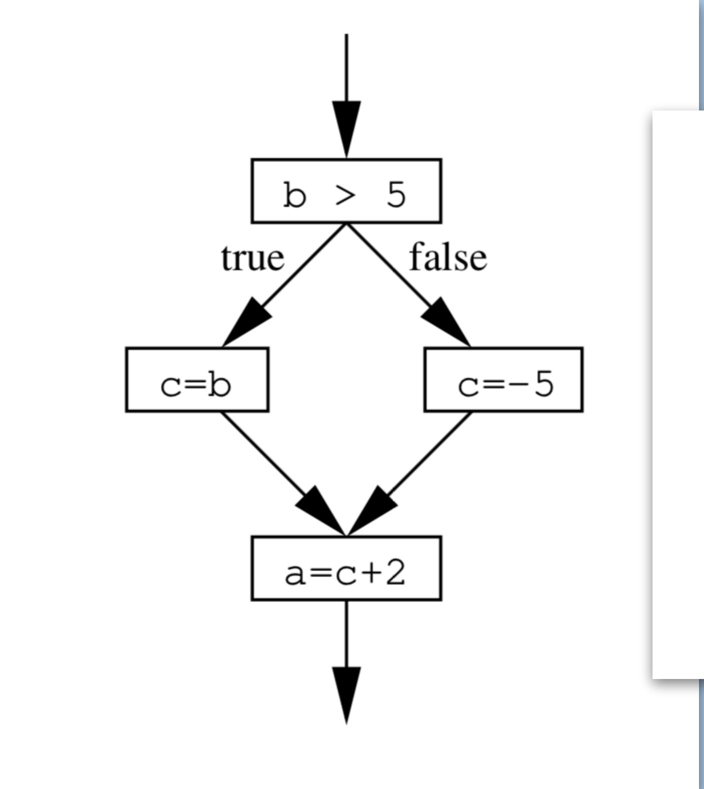
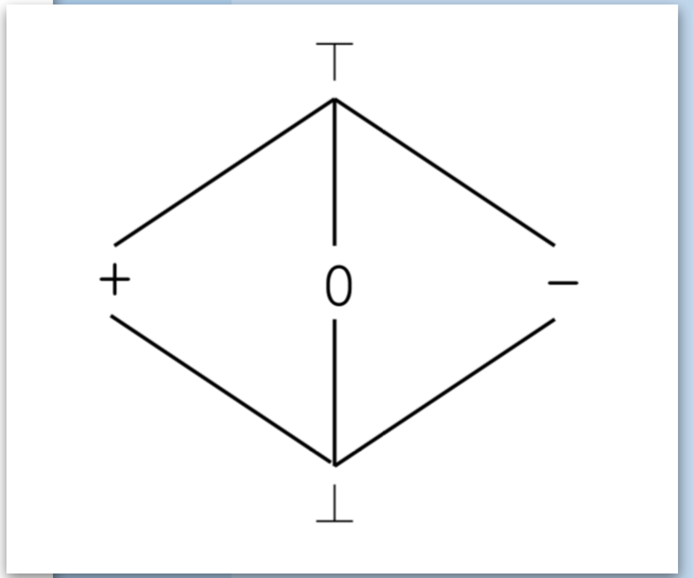
UNIVERSITÄT BERN

Static Analysis

Part 1: Static Analysis Techniques

Oscar Nierstrasz





Roadmap

- What is static analysis?
- Type checking
- Dataflow analysis
- Common static analysis techniques

Resources

- Lecture notes on Static Program Analysis, Anders Møller and Michael I. Schwartzbach
  - http://cs.au.dk/~amoeller/spa/

These lecture notes offer a very thorough and gentle introduction to the main static analysis techniques.

Roadmap

> **What is static analysis?**
> Type checking
> Dataflow analysis
> Common static analysis techniques

What is Static Analysis?

Static program analysis is the *systematic examination* of an *abstraction* of a *program's state space*

Instead of attempting to analyze the full behaviour of a program (which is in general undecidable), the idea is to create a more abstract representation, typically a finite one, which can be fully analyzed.

Questions about programs

- Does the program terminate on every input?
- Does the program contain dead code?
- Can secret information become publicly observable?
- Can the program deadlock?
- Does there exist an input that leads to a null pointer dereference, division by zero, or arithmetic overflow?
- Are all assertions guaranteed to succeed?
- Are all variables initialized before they are read?

Rice’s Theorem

> “Any nontrivial property about the language recognized by a Turing machine is undecidable.”
>
> — Henry Gordon Rice, 1951

Implies that many interesting static analysis questions are incomplete, or unsound, or undecidable.

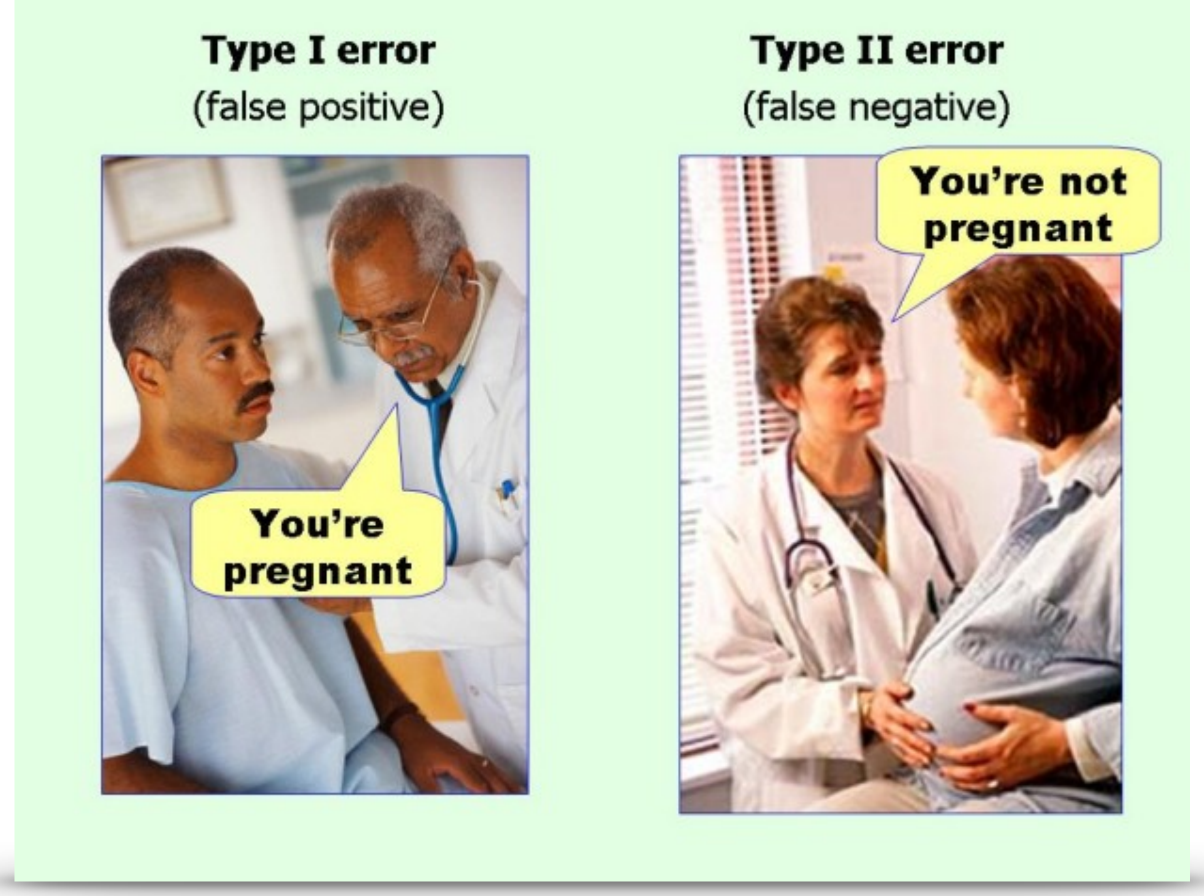
Rice’s theorem argues that there is no Turing machine that can always decide that the language of (another) given Turing machine has a particular nontrivial property.

For this reason we do not attempt static analysis over the full behaviour of programs, but instead over *abstractions* of the possible program states.

https://en.wikipedia.org/wiki/Rice%27s_theorem

Approximation

- *Approximate answers* may be decidable
- Approximations must be conservative
  - i.e., err on the "safe side"
  - In practice, either *no false positives*, or *no false negatives*



By abstracting over the program state, we avoid the dilemma of Rice's theorem. However the answers given by our static analyses will in general be “wrong”. This means that there will be false positives (programs that the analysis concludes have a certain property while in fact they don’t) as well as false negatives (programs that we fail to detect have a given property).

We want our static analyses to be conservative, i.e., either they have no false positives or no false negatives (depending on the particular problem).

An analysis with both false negatives and positives is pretty much useless.

Example approximations

> Code is live: don't remove it
  - False positives are ok (dead code left in)
  - False negatives are not ok (removing live code)

> Downcast (A) x will succeed: skip run-time check
  - False negatives are ok (check safe casts)
  - False positives are not (skip test of unsafe cast)

The engineering challenge: minimize the errors with reasonable performance (time and space)

The engineering challenge is to provide useful and accurate analyses with reasonable cost and performance. To obtain more accurate results, an analysis must typically work with a fairly detailed state space, but this can require large amounts of memory, and can be expensive to analyze.

See:

> "The Use and Limitations of Static-Analysis Tools to Improve Software Quality", Paul Anderson, 2008
>
> http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.639.624&rep=rep1&type=pdf

Static vs Dynamic Analysis

> Static analysis:
  - analyze a representation of the source code; don't run the program
  - may over- or under-approximate (false positives or negatives)

> Dynamic analysis:
  - execute the program, possibly with multiple inputs
  - limited to specific runs (false negatives)

> Hybrid approaches
  - use dynamic analysis results to augment static analysis
  - common in recommender systems

Static analysis works with a representation of the source code alone, e.g., the program text, the abstract syntax tree, a graph representing the control flow, etc.

Dynamic analysis, on the other hand, works with a representation of a concrete run of the program, e.g., a log of the methods or procedures that have been executed, generated by an instrumented version of the compiled program.

Although dynamic analysis works with precise data rather than an approximation, it only has access to individual traces that represent a *subset* of all possible runs of the program.

Hence both techniques work with partial information.

Hybrid approaches take results from dynamic analysis of the same system or related systems, and use them to offer insights into the software being statically analyzed.

Roadmap

> What is static analysis?
> **Type checking**
> Dataflow analysis
> Common static analysis techniques

Type checking

> Type checking ensures that operations are only applied to valid arguments
  - arithmetic operations only applied to numbers
  - only invoke valid methods on objects
  - only pass valid arguments to methods or functions

Static and dynamic types

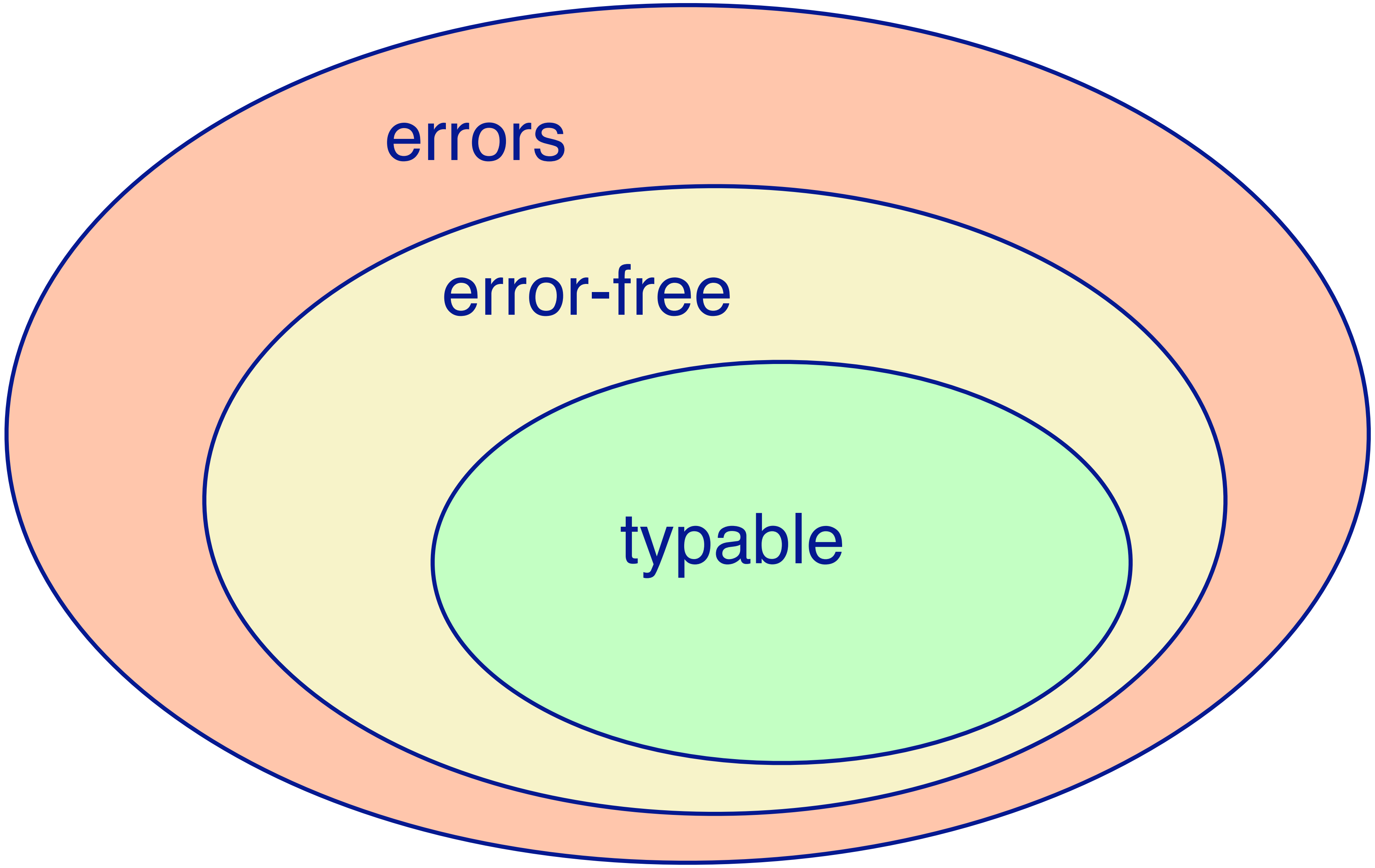
> The static type of a variable is its declared type
> The dynamic type of a variable is the run-time type of the value it holds

> In statically-typed languages types are checked at compile time
  - The compiler will forbid code to be compiled that cannot be proven type-safe
> In dynamically-typed languages types are checked only at run time
  - A static analysis tool can (attempt to) infer types and skip run-time checks that are deemed unnecessary

Note that there is no such thing as an “untyped” language, just languages which check types at compile time and those that check them at run time.

Type safety

Static type safety may exclude some programs that do not lead to errors, but are not provably type-safe



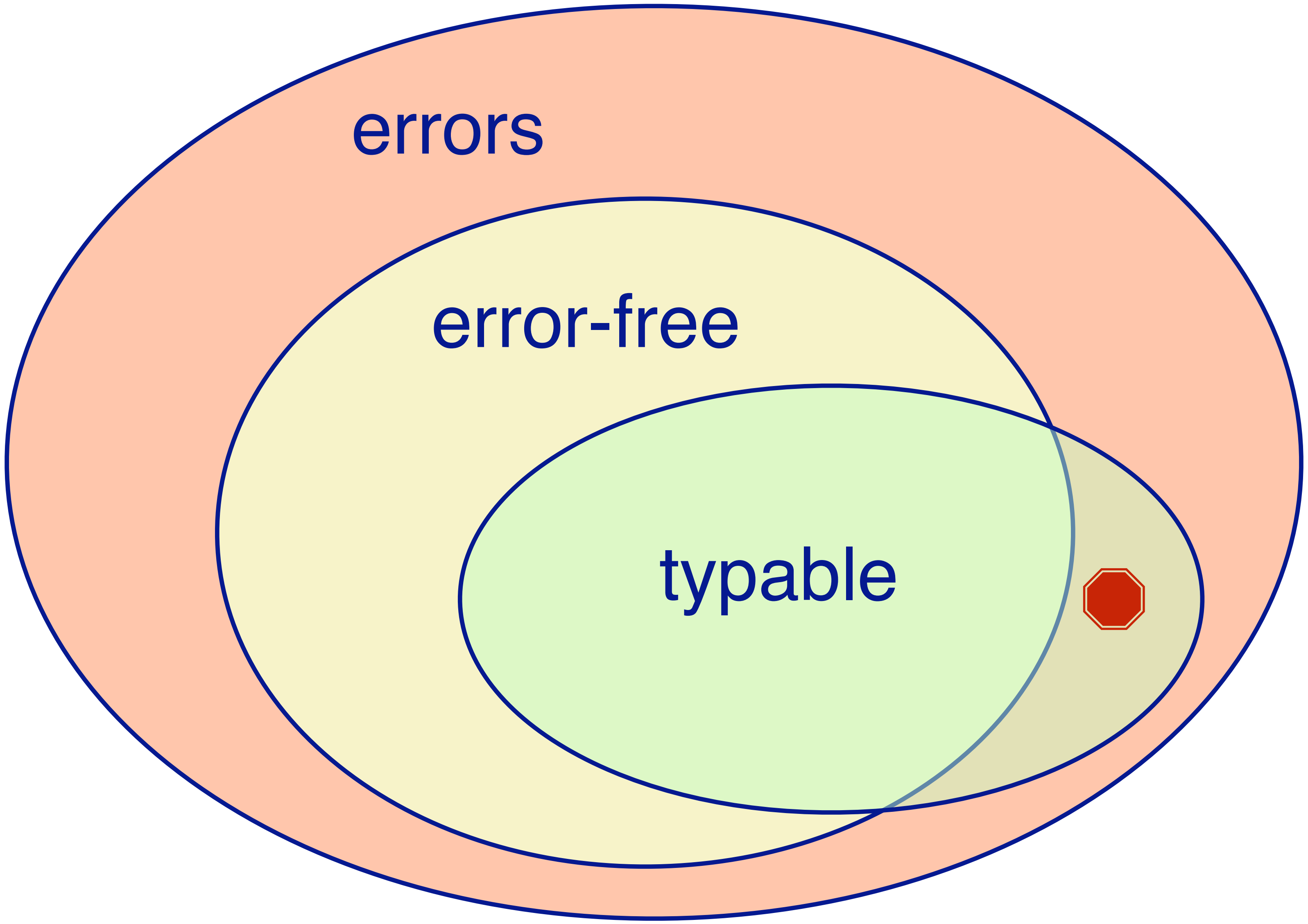

```
Object cowboy = new Cowboy();
cowboy.draw();
```

Møller and Schwartzbach refer to the yellow area as “slack”. The idea is to have a type system that is fast and efficient, and reduces as much as possible the “slack” area of programs that are actually ok, but cannot be statically type-checked.

The sample code is in fact safe, but will not pass the Java type-checker. We can fix this either by changing the declaration of `cowboy` to `Cowboy` in the first line, or by inserting a dynamic downcast in the second line of `cowboy` to `Cowboy`.

Type soundness

If a type system is unsound, it can accept code that may lead to run-time errors.



Example: co-variant arrays in Java.

In general type systems should be sound, but sometimes for pragmatic reasons unsound type rules can be useful. In such cases, dynamic checks should be generated to avoid catastrophic errors at run time.

Type-checking constraints

The "static semantics" of expressions in the programming language denote their static types. The static analysis tool resolves constraints over the static types.

Constraints:

$[\![n]\!] = \text{int}$

$[\![E1]\!] = [\![E2]\!] = [\![E1 \text{ op } E2]\!] = \text{int}$

Code:

3 + 4

3 + "hello"

Constraints are defined for all syntactic constructs of the programming language. For a given program, constraints are generated for every identifier and expression. The constraint solver then attempts to resolve the constraints.

In the example we would generate the constraints:

$[\![3]\!] = [\![4]\!] = [\![3+4]\!] = \text{int}$

$[\![3]\!] = [\![\text{“hello”}]\!] = [\![3+\text{“hello”}]\!] = \text{int}$

The first expression clearly passes while the second does not.

This is just one of many possible type-checking strategies. Here we also do not discuss techniques to statically infer types for dynamically-typed languages.

Type rules

$$\frac{}{\vdash \text{num} : \text{int}}$$

Numbers have the type "int"

$$\frac{A \vdash e_1 : \text{int} \quad A \vdash e_2 : \text{int}}{A \vdash e_1 < e_2 : \text{bool}}$$

A comparison of two ints is of type "bool"

A classical way to encode type checking rules is as natural deductions.

The type rules for a programming language can be described as a set of deduction rules, where the top part of the rule encodes what is known about some syntactic elements, and the bottom encode what can be concluded.

In the first rule, there is no precondition, so the top part is empty. A number is always of type “int”.

In the second rule, A represents the type environment, i.e., all the type declarations in the program. It says, if in this program we can conclude that e_1 and e_2 are ints, then in the same environment A we can also conclude that $e_1 < e_2$ is a bool.

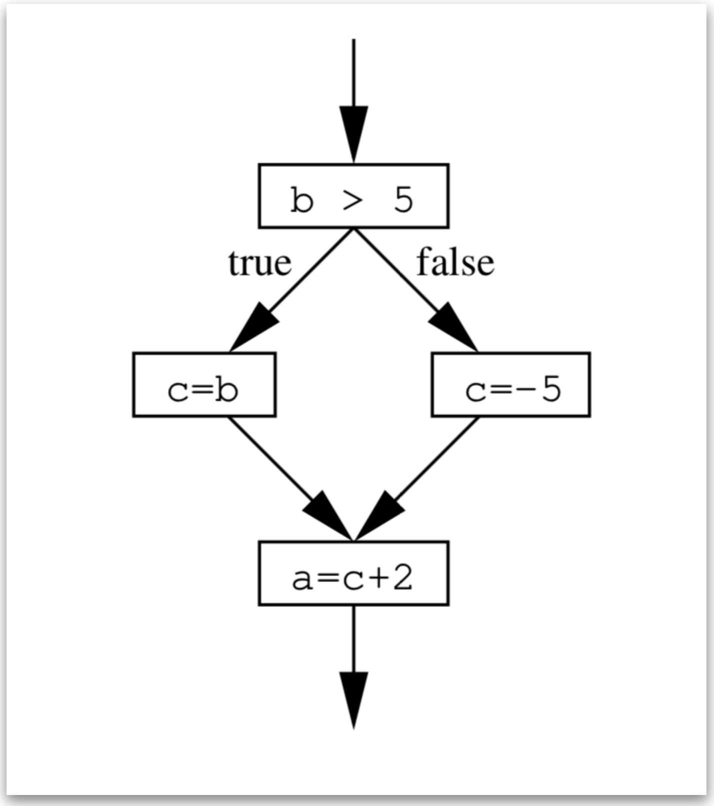
These rules can be translated to code that performs the type checking, for example by traversing the abstract syntax tree of the program.

https://en.wikipedia.org/wiki/Type_rule

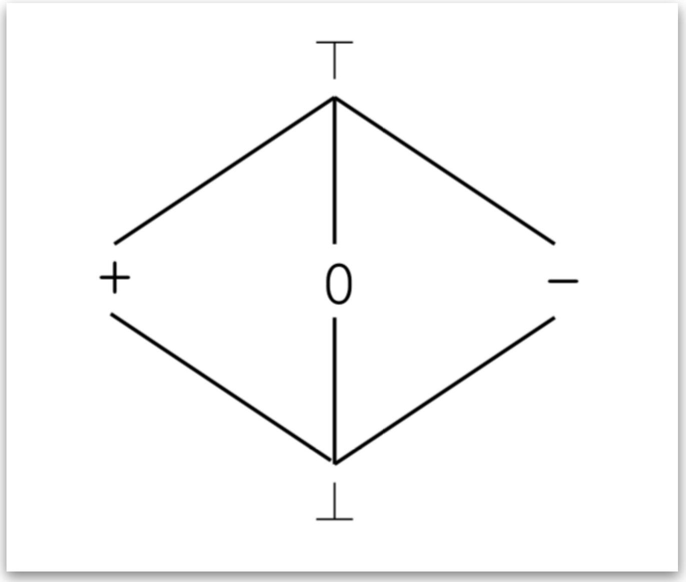
Roadmap

- What is static analysis?
- Type checking
- **Dataflow analysis**
- Common static analysis techniques

Dataflow analysis



Control-flow graph



Lattice of abstract values

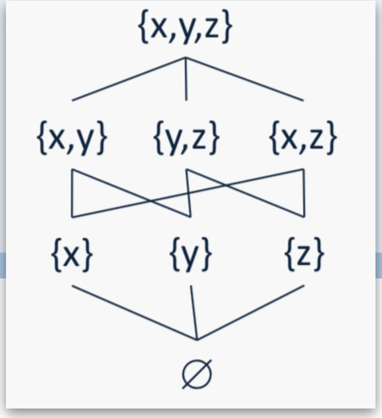
Dataflow analysis used used to reason about abstract values of variables. Here we want to reason about the signs of the variable a, b and c at various points in the program.

Dataflow analysis makes use of a control flow graph (CFG) that represents the possible flows through a program as a directed graph. Each node typically represents a "basic block" of straight-line code (i.e., without any jumps or jump targets).

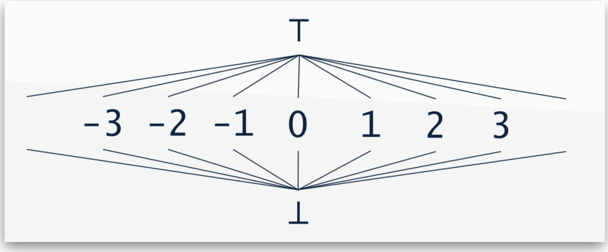
In addition, a lattice represents the abstract values of interest, in this case the possible signs of the variables. These may be positive, negative, or zero (+, –, 0). A lattice is a graph structure in which every subset of nodes has a *least upper bound*, and a *greatest lower bound*. The top element ⊤ here represents "any possible sign" while the bottom ⊥ represents "no possible sign".

For each node in the graph we generate constraints from the inputs to the outputs. After node c=–5, for example, we know that c has the abstract value –.

Applications of data flow analysis

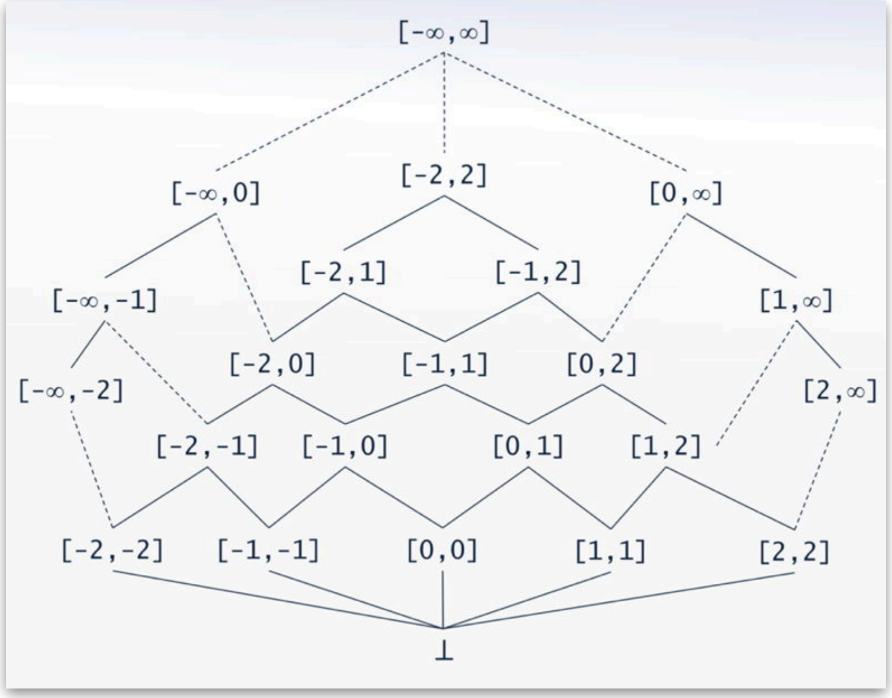


> Liveness
  — Which variables are “live” (i.e., hold values that may later be read)?
  — Uses subset lattice of all possibly live variables



> Constant propagation analysis
  — Which expressions have compile-time constant value?
  — Uses flat lattice of constant values

> Interval analysis
  — What intervals of integer values may variables hold?
  — Uses lattice of intervals



In each case a different kind of lattice is used to reason about the possible “values” of interest variables or expressions may hold. In general the top element represents no useful information (all possible values), while the bottom element represents an impossible value (none of the values).

Interprocedural analysis

- Intraprocedural analysis
  - analyze the body of *a single function*
- Interprocedural analysis
  - analyze the *whole program* with function calls

- *Key idea*
  - construct a CFG for each function
  - glue them together in a graph
  - need to handle
    - *parameter passing*
    - *return values*
    - *values of local variables across calls*

Roadmap

> What is static analysis?
> Type checking
> Dataflow analysis
> **Common static analysis techniques**

Pointer analysis

Andersen's algorithm uses constraints to reason about what variables a pointer may point to.

```
p = &a;
q = p;
p = &b;
r = p;
```

$[\![p]\!] \supseteq \{a\}$

$[\![q]\!] \supseteq p$

$[\![p]\!] \supseteq \{b\}$

$[\![r]\!] \supseteq p$

p points-to {a,b}

q points-to {a,b}

r points-to {a,b}

a points-to {}

b points-to {}

Pointer analysis (or “points-to analysis”) attempts to determine what variables (i.e., areas in memory) a pointer may end up pointing to. Such analysis should be *conservative*, i.e., may include false positives, but no false negatives.

Andersen’s algorithm generates a set of constraints for each point p, for ⟦p⟧, the set of variables that p might point to.

In the example, p points to a, q points to the same thing p points to (q *aliases* p), p points to b, and finally r aliases p.

Ignoring control flow, we conclude that p, q and r may point to a and b.

“Program Analysis and Specialization for the C Programming Language”, Lars Ole Andersen, Ph.D. Thesis, U Copenhagen, 1994

Symbolic execution

```
main () {
    print fact(read());      n=IN
}

fact (n) {                   n=IN
    if (n<0) {               n=IN, n<0
        throw ERROR;
    }
    if (n == 0) {            n=IN, n≥0
        return 1;
    } else {
        return n * fact(n-1);   n=IN, n>0
    }
}
```

Execute program with symbolic values. Generate constraints and solve to determine outcomes.

Error only if IN<0

n–1≥0 cannot lead to error

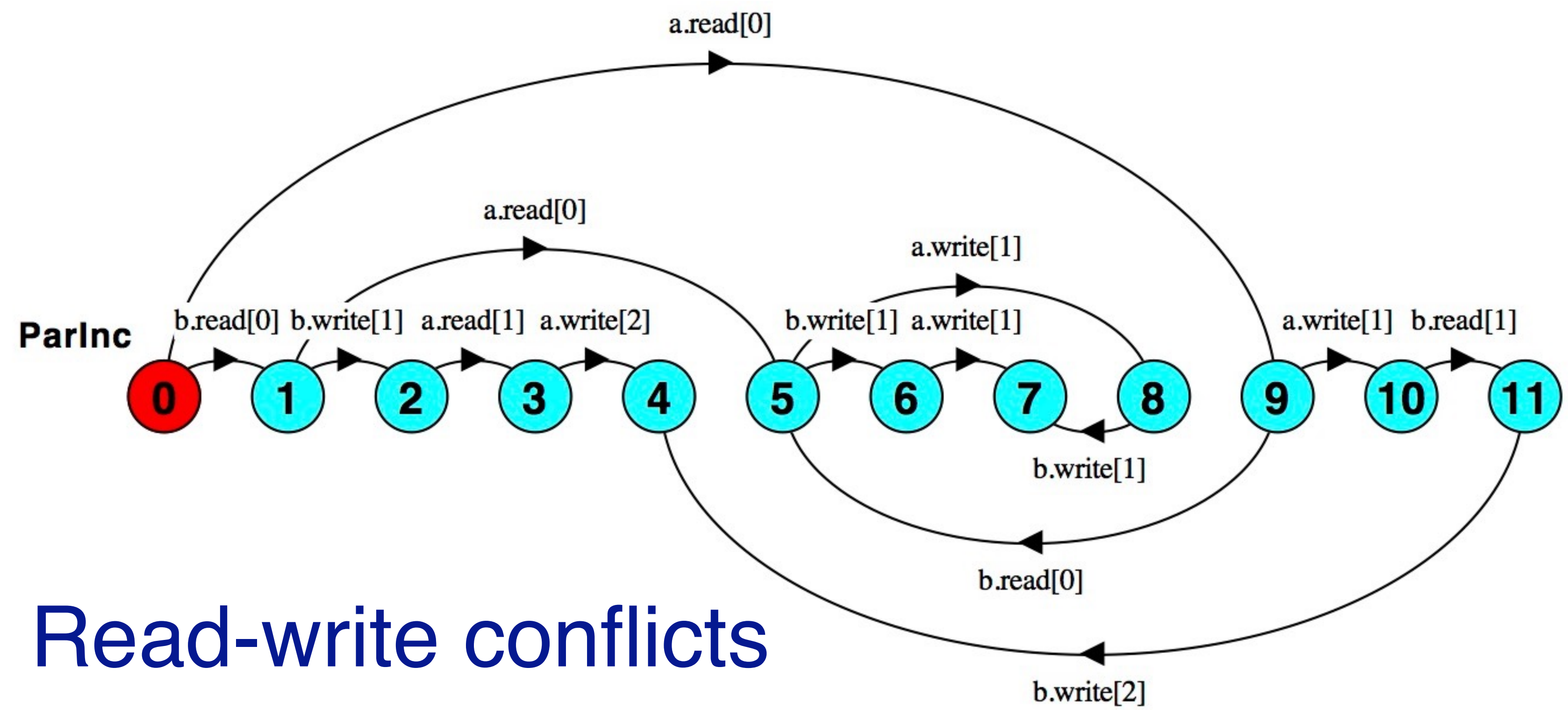
Symbolic execution is a form of *abstract interpretation* used to analyze programs by simulating the execution of a program using symbolic values for the inputs, and constraints over these values to represent outcomes of conditional branches.

Here we trace the execution of a factorial function, and see that an error can arise if the input is less than zero.

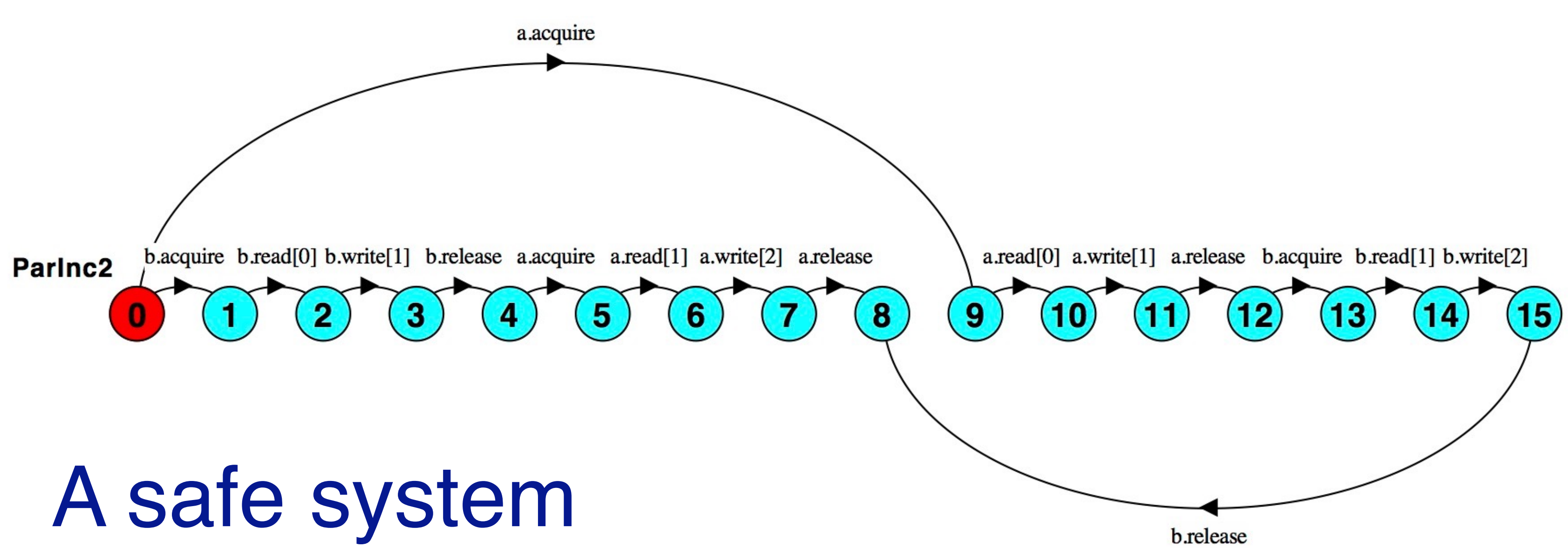
Note that if we enter the recursive call, we establish that $n>0$, so the input to the recursive call is ≥ 0 and cannot lead to any error.

Model checking

Model-checking approximates a software system as a finite-state system and exhaustively explores those states to verify some desirable property.



Read-write conflicts



A safe system

Model-checking refers to a general approach to verifying software systems by modeling them as finite state systems.

The example shows LTSA (labeled transition system analyzer), a tool that models concurrent software as a set of interacting finite state processes. It can be used to verify *safety* properties (no bad state is reachable) as well as *liveness* properties (some desired actions are always possible).

In the example, the system above is unsafe as there is a reachable state in which processes A and B conflict in reading, incrementing and writing a variable. The system below uses locks to avoid the unsafe state.

LTSA web site:

https://www.doc.ic.ac.uk/ltsa/

What you should know!

- Why do static analysis techniques work with an *abstraction* of a program's state space?
- Why must approximations be *conservative*?
- How does static analysis *differ* from dynamic analysis?
- Why should type systems generally be *sound* (but not necessarily complete)?
- What kinds of questions can be answered by data flow analysis?

Can you answer these questions?

> What technique(s) would you use to detect *dead code*?
> Can a type system reject *only* programs that lead to type errors?
> Why is symbolic execution considered to be a *static* (and not dynamic) analysis technique?
> Why is it crucial that model-checkers work with *finite state* models of software?