Introduction to program analysis

Material covered in chapter 1 of
Introduction to Static Analysis: an Abstract Interpretation Perspective

Outline

Verification: a first definition

In this course, we consider how to **verify specific properties about program executions**:

- absence of execution errors
  i.e., crashes due to pointer or arithmetic errors
- preservation of invariants
- termination
- absence of information flows and other security breaches...

We are not interested in purely syntactic properties

Verification

Make sure that $[\![P]\!] \subseteq \mathcal{S}$ where

- **the semantics** $[\![P]\!]$ describes the **set of behaviors of** P,
- **the specification** $\mathcal{S}$ describes the set of acceptable behaviors

Behaviors are still an abstract notion at this point

Semantics and semantic properties

There exists **several forms of semantics** $[\![P]\!]$ that convey:

- reachable states, input/output relations (e.g., described by a function), execution traces of program states (finite, infinite, or both)

We will consider two styles of semantics:

- **compositional style** ("denotational")
  - intuitively, $[\![AB]\!] = \cdots [\![A]\!] \cdots [\![B]\!] \cdots$
- **transitional style** ("operational")
  - intuitively, $[\![AB]\!] = \{s_0 \hookrightarrow s_1 \hookrightarrow \cdots, \cdots\}$
- **A right semantics style facilitates the design of static analysis**

Specification (or **semantic properties** of interest):

- sets of executions (that are considered to satisfy a specification)
- property can be expressed by $[\![P]\!] \subseteq \mathcal{S}$
- there exist **several interesting classes of semantic properties**

Safety

Intuitive definition: safety

A safety property asserts that some kind of behaviors that are observable in finite time will never occur.

Examples:

- absence of some class of crashing error
  e.g., null pointer exception in Java, arithmetic or memory error in C
- preservation of a general invariant
  e.g., some data structure should never get broken
- assertion on output value
  e.g., the output value of a function should always lie in a given range

Proof method: **by invariance**

i.e., a safety property $\mathcal{S}$ holds if and only if there exists a program **invariant** stronger than $\mathcal{S}$

Liveness

Intuitive definition: liveness

A livenesss property asserts that some kind of behaviors that are only observable in infinite time will never occur.

Examples:

- non termination
- live lock
- unbounded repetition of a given behavior
  note termination and live lock are special cases of this one

Proof method: **with a variance argument**

e.g., for termination, **ranking functions**:

search for a measure that decrease during execution
and that cannot decrease forever

Trace properties

Not all trace properties are safety or liveness. **What about others ?**

Theorem (Alpern and Schneider)

Given a trace property $\mathcal{T}$ (i.e., a set of finite or infinite program executions), then there exists two trace properties $\mathcal{S}$ and $\mathcal{L}$ such that:

- $\mathcal{T} = \mathcal{S} \cap \mathcal{L}$
- $\mathcal{S}$ **is a safety property**
- $\mathcal{L}$ **is a liveness property**

Application: the proof of safety property boils down to

1. a proof of safety, by variance
2. a proof of liveness, by invariance

Example: total correctness = partial correctness and absence of crashes + termination

How to make such proofs automatic ?

Beyond trace properties: security, dependences...

Many important semantic properties **cannot be described only by a set of executions**.

For instance,

- **dependence**:
  `y` depends on `x` if and only running the program with distinct values for `x` yields distinct observations for `y`
- **absence of information flow** (**security** property):
  absence of dependences of public outputs on private data

To prove/disprove these properties, one needs to reason simultaneously on **pairs of traces**

How to make such proofs automatic ?

Outline

The termination problem

Termination

Program P terminates on input X if and only if
any execution of P, with input X eventually reaches a final state

- **Final state:** final point in the program (i.e., not error)
- **We may want to ensure termination:**
  - processing of a task, such as, e.g., printing a document
  - computation of a mathematical function
- **We may want to ensure *non*-termination:**
  - operating system
  - device drivers

The termination problem

Can we find a program `Pt` that **takes as argument a program P and data X and that returns "`true`" if P terminates on X and "`false`" otherwise ?**

The termination problem is not computable

- **Proof by reductio ad absurdum**, using a *diagonal argument*
  We assume **there exists a program** `Pa` **such that:**
  - `Pa` always terminates
  - $\texttt{Pa}(P, X) = 1$ **if** P **terminates** on input X
  - $\texttt{Pa}(P, X) = 0$ **if** P **does not terminate** on input X
- We consider the following program:

```
void P0( P ){
  if( Pa( P, P ) == 1 ){
    while( 1 ){
      // loop forever
    }
  } else {
    return; // do nothing
  }
}
```

- **What is the return value of** $\texttt{Pa}(\texttt{P0}, \texttt{P0})$ **?**
  i.e., **does** `P0` **terminate on input** `P0` **?**

The termination problem is not computable

- **What is the return value of `Pa(P0, P0)` ?**
  We know `Pa` always terminates and returns either 0 or 1 (assumption).
  Therefore, we need to consider only two cases:
  - if `Pa(P0, P0)` returns 1, then `P0(P0)` **loops forever**,
    thus `Pa(P0, P0)` should return 0, so we have reached a **contradiction**
  - if `Pa(P0, P0)` returns 0, then `P0(P0)` **terminates**,
    thus `Pa(P0, P0)` should 1, so we have reached a **contradiction**

- In both cases, we **reach a contradiction**

- Therefore we conclude **no such a `Pa` exists**

The termination problem is not decidable

There exists no program `Pt` that always terminates and always recognizes whether a program P terminates on input X

Undecidability of interesting verification problems

We assume a **Turing complete language** $\mathbb{L}$.

There is no computable algorithm **Exact** such that

$$\text{For all } P \in \mathbb{L},\ \mathbf{Exact}(P) = [\![P]\!]$$

Otherwise, we could solve the termination problem by using such **Exact**.

Undecidability of non trivial semantic properties

Let $\mathcal{S}$ be a non trivial semantic property (non trivial: neither true for all programs nor false for all programs).

Then $\mathcal{S}$ is not decidable on $\mathbb{L}$.

There is no fully automatic and exact algorithm deciding $\mathcal{S}$.

For instance:

- The halting problem is not decidable
- The absence of runtime errors is not decidable...
- Total correctness is not decidable...

Outline

Inexact verification: soundness and completeness

As we have seen automatic and exact verification is impossible.

How to retain automation, while still verifying programs ?

Approximate verification, reaches for a **weaker goal** than exact verification.

Two important notions:

- **Soundness**: analysis(P) = yes $\Longrightarrow$ P satisfies the specification
  i.e., rejects any program that violates the specification
- **Completeness**: analysis(P) = yes $\Longleftarrow$ P satisfies the specification
  i.e., accepts any program that satisfies the specification

In the following, we consider various verification techniques, that give up partially on either automation, soundness or completeness.

Testing

Principle

1. **Consider finitely many, finite executions**
2. For each of them, **check whether it violates the specification**

- Very natural idea, used on all software projects, at all levels (from unit testing to integration testing)
- Many advanced techniques (e.g., to choose "good" test samples)
- Challening to apply in presence of non-determinism (reproducibility issue) or for hyperproperties (need to talk about several executions in one)...
- **In general unsound**: when state space is infinite or even finite, but just too big (testing does not scale), soundness cannot be ensured
- **Complete**: when a violation is discovered, a counter-example can be produced

Machine assisted proving

Principle

1. Use a **specific language** to **formalize verification goals**
2. **Manually supply proof arguments**
3. Let the proofs be **automatically verified**

- Example of tools: Coq, Isabelle/HOL, PVS...
- **Applications**: CompCert (certified compiler), SeL4 (secure micro-kerne)...
- **Not automatic**: key proof arguments need to be found by users
- **Proof search algorithms** often reduce the amount of proof arguments that need to be supplied manually
- **Sound**, if the formalization is correct
- **Quasi-complete** (only limited by the expressiveness of the logics)

Finite state model checking

Principle

1. Focus on **finite state models** of programs and systems
2. Perform **exhaustive exploration** or some **optimised form of it**

- Example: Uppaal
- **Automatic**
- **Sound** and **complete** **with respect to the model**
- However, general programs require **approximate models**
  at this stage, one **loses either soundness or completeness**

Conservative static analysis

Principle

1. Perform **automatic verification, yet which may fail**
2. Compute a **conservative approximation of the program semantics**

Two kinds of approximations are possible (with math. guarantee):

- **Sound, incomplete**: the most common case
- **Complete, unsound**: rare

Sound, incomplete static analysis very widely used:

- Examples: type systems, Astrée, Facebook Infer, Sparrow...
- Most compilers use it without users even noticing
  (type system, analyses for optimization or code generation)
- **Automatic**
- Incompleteness means that **safe programs may be rejected**
  or that **false alarms** may be raised
- Analysis algorithms **reason over program semantics**

Bug finding

Principle

Automatic, **unsound** and **incomplete** algorithms

- Examples: Coverity, CodeSonar...
- **Automatic** and **generally fast**
- **No mathematical guarantee about the results**
  may reject a correct program, and accept an incorrect one
  may raise false alarm and fail to report true violations
- Typially used to increase software quality without trying to provide any strong guarantee

High-level comparison

	automatic	sound	complete
testing	yes	no	yes
assisted proving	no	yes	yes/no
model checking of finite state model	yes	yes	yes
model checking, at program level	yes	yes	no
conservative static analysis	yes	yes	no
bug finding	yes	no	no

No program level approach can be automatic, sound and complete

Outline

Scope and objectives

We consider **automatic, conservative static analyses**, that compute **some abstraction of the semantics of programs**

To achieve a good understanding of this family of works, we need to study:

- **the semantics of programs**
  indeed, it serves as a basis for the definition of abstractions
- **the notion of conservative approximation** of a semantics
  i.e., what it means to be conservative, how it can be formalized
- **the computation of conservative approximations**
  using abstract interpretation techniques, step-by-step abstract execution, and widening

The lectures focus on foundations (intuition and formalization).
The book also exposes advanced topics.
We encourage to look at practical chapters (chapters 6 and 7) in the same time as the corresponding notions are considered in the lectures

Outline of the next lectures

1. Introduction to static analysis (this course)
   (chapter 1)
2. A gentle introduction to static analysis by abstract interpretation
   (chapter 2)
3. Basic notions of semantics
   (sections 3.1 and 4.1)
4. Semantic abstraction
   (section 3.2)
5. Static analysis based on a compositional semantics
   (section 3.3)
6. Static analysis based on a transitional semantics
   (sections 4.2 and 4.3)
7. Specialized static analysis frameworks
   (chapter 10)