Counterexample-Guided Abstraction Refinement

E. Clarke et al., CAV ’00

Presenter: Hyunsu Kim

Checking program is important

$\supseteq$

Specification
(High-level)

Implementation
(Low-level)

Checking program is important

$\supseteq$

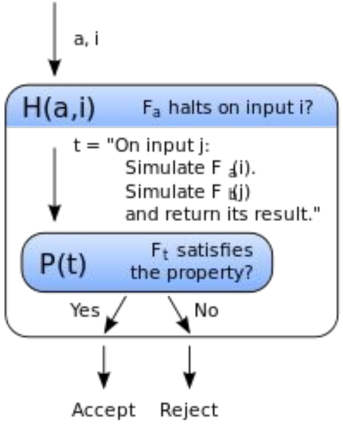
Specification
(High-level)

Implementation
(Low-level)

E.g. OS kernel developer wants to know multi-threaded system call terminates (liveness property)

Checking program is non-trivial

According to Rice’s theorem, if it were trivial, we could solve halting problem i.e. contradiction. (Proof by contradiction)



https://en.wikipedia.org/wiki/Rice%27s_theorem#/media/File:Rice_reduction.svg

We can still check our program

SmaLLVM Dataflow Analysis

The goal of this homework is to implement a static analyzer based on dataflow analysis that detects potential division-by-zero errors in SmaLLVM programs. Students will write the following modules based on the definitions in this document:

- abstract domains for memories (module `Memory` in src/domain.ml)
- abstract domains for integer values (module `Sign` in src/domain.ml)
- abstract semantics for selected LLVM instructions (`transfer` , `transfer_cond` , `transfer_phi` , `eval` , and `filter` in src/semantics.ml)
- a generic fixed point computation engine (`run` in src/analysis.ml)

In short, replace all `failwith "Not implemented"` with your own implementation. You can ignore the other LLVM instructions marked as `raise Unsupported` , and assume that input programs are always **syntactically** valid.

Notice that students are not permitted to change directory structures and types of the functions. All the functions you implement must have the same types as described in the module signatures. However, students are allowed to change `let` to `let rec` if needed.

SmaLLVM Constraint-based Analysis

The goal of this homework is to implement a static analyzer based on constraint-based analysis that detects potentially vulnerable data flow from source points to sink points (so called taint analysis) in SmaLLVM programs. Students will write the following definitions based on the definitions in the lecture slides:

- the function `extract_instr` in src/extractor.ml that extracts initial Datalog facts from LLVM IR
- the Z3 expressions `r1` , `r2` , and `r3` that defines the Datalog rules for the inductive cases of the analysis rules

In short, replace all `failwith "Not implemented"` with your own implementation. You assume that input programs are always **syntactically** valid.

Notice that students are not permitted to change directory structures and types of the functions. All the functions you implement must have the same types as described in the module signatures. However, students are allowed to change `let` to `let rec` if needed.

Two recent assignments

Type systems

Target property + Abstraction

https://en.wikipedia.org/wiki/Rice%27s_theorem#/media/File:Rice_reduction.svg

Abstraction is required for model checking

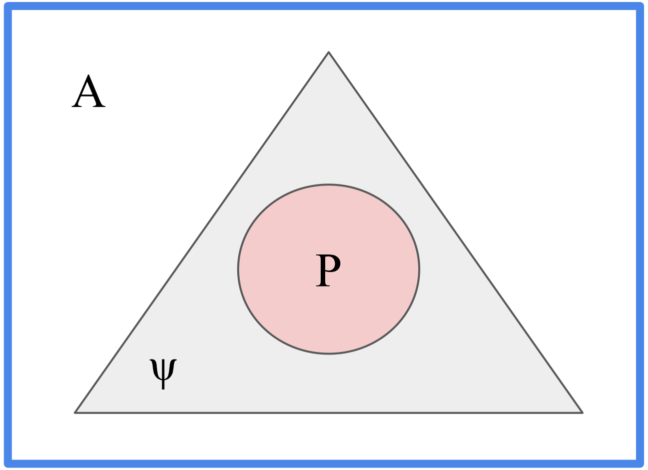
Abstraction

- **Coarse** up to high-level idea
- Reason in **finite** time
- cf. Static analysis

Concretization

- **Refined** up to implementation details
- May take **infinite** time for enumeration
- cf. Fuzzing

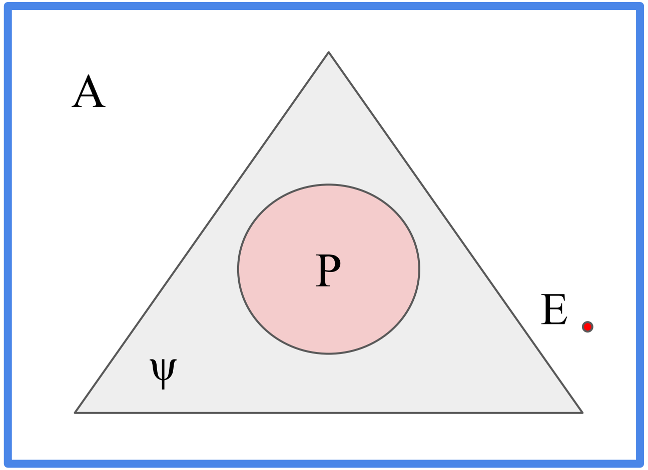
CEGAR automates model checking



Underlying fact:

- P satisfies ψ

CEGAR automates model checking

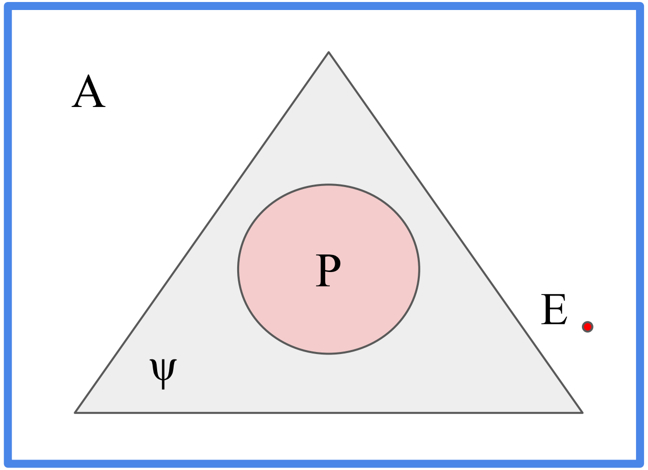


Find a counterexample E

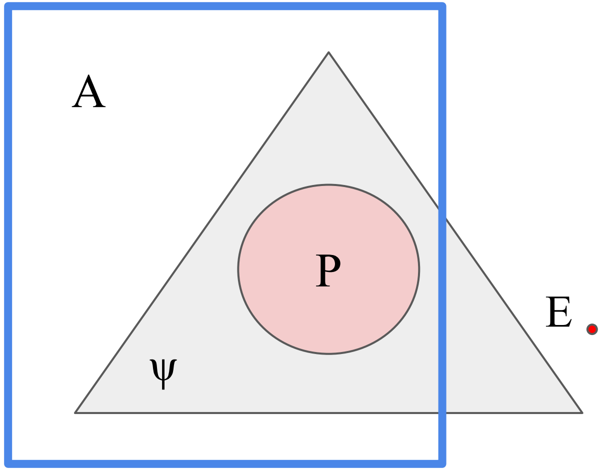
Underlying fact:

- P satisfies ψ

CEGAR automates model checking

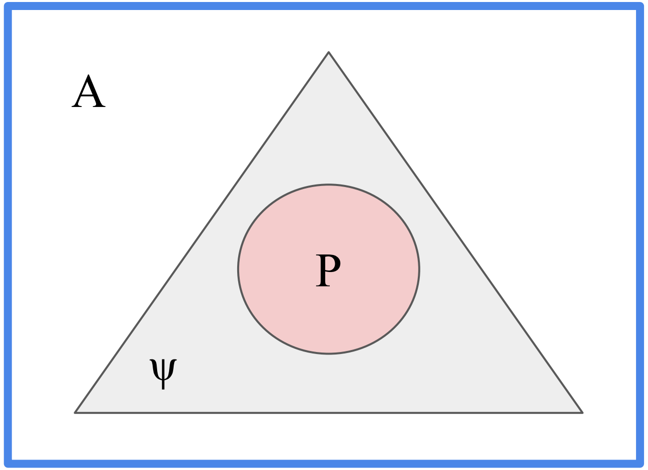


Underlying fact:
- P satisfies ψ

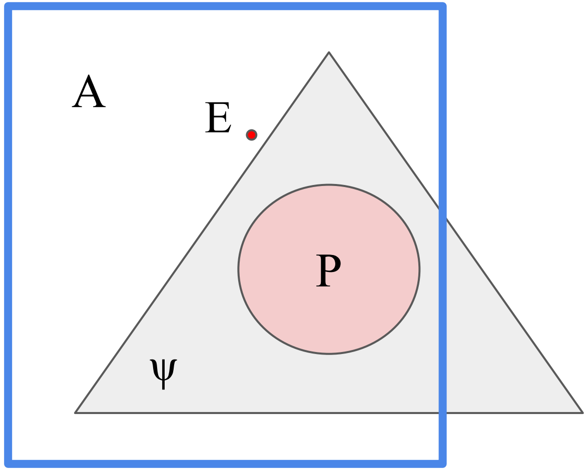


Refine A that eliminates E

CEGAR automates model checking

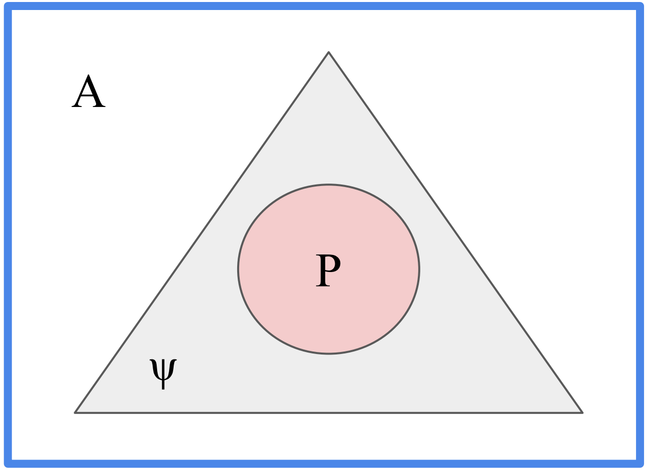


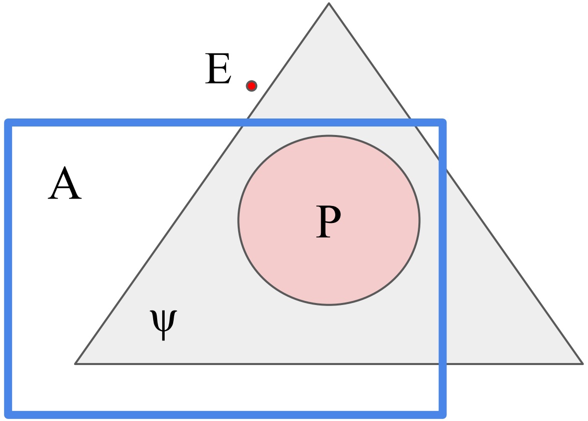
Underlying fact:
- P satisfies ψ



Find another counterexample E

CEGAR automates model checking

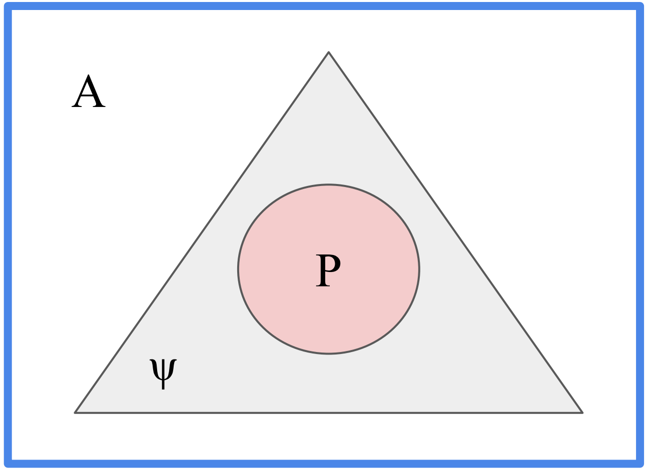


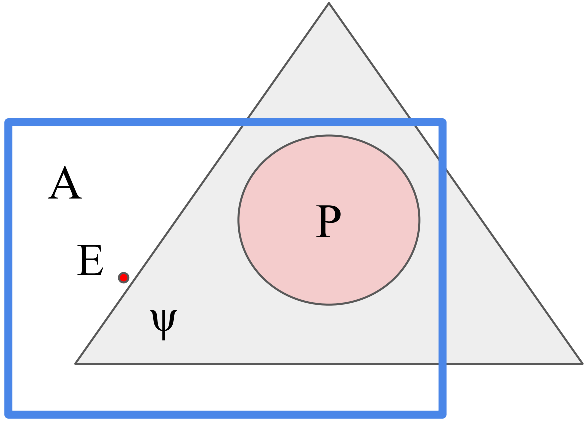


Underlying fact:

- P satisfies ψ

CEGAR automates model checking

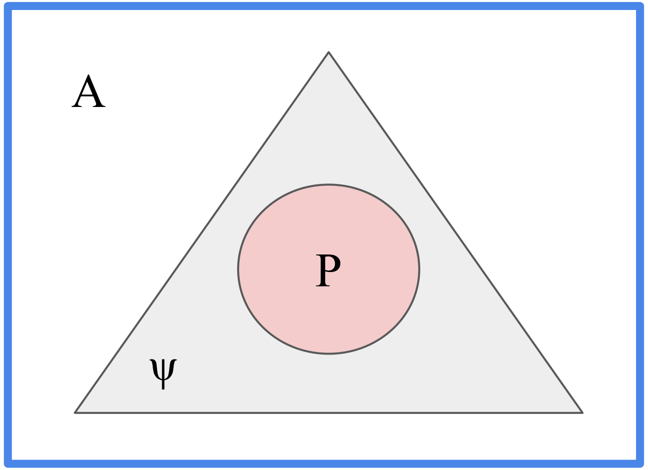




Underlying fact:

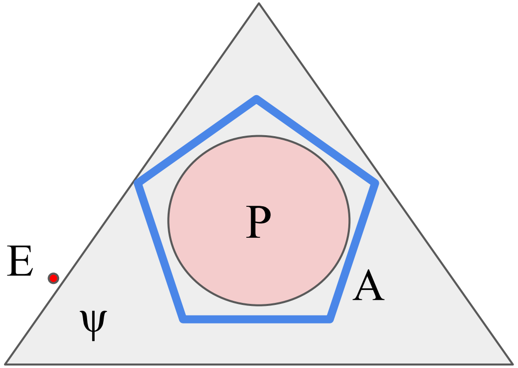
- P satisfies ψ

CEGAR automates model checking



Underlying fact:

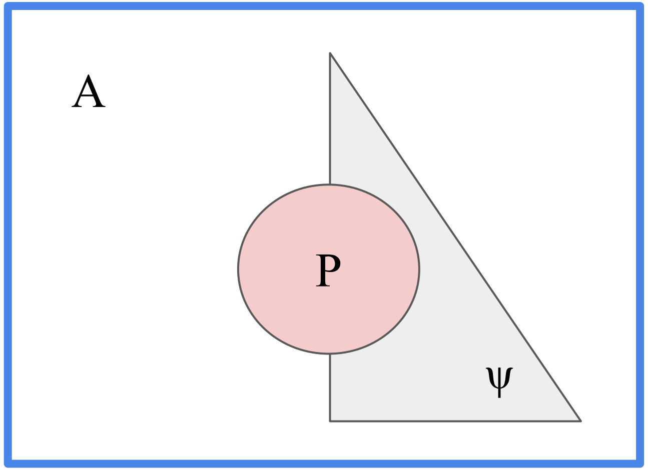
- P satisfies ψ



Derived fact:

- A satisfies ψ, so then P satisfies ψ

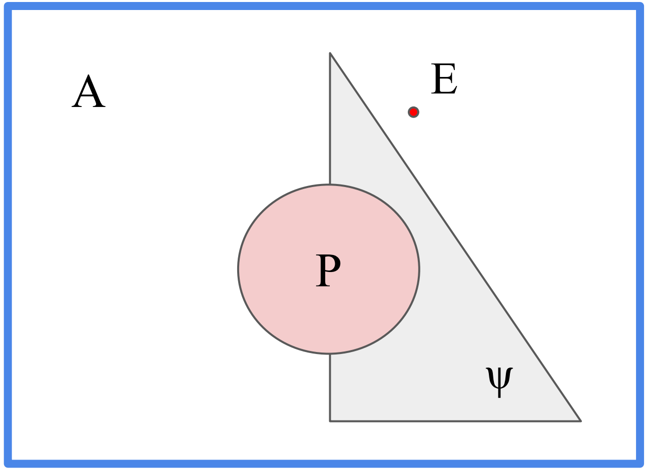
CEGAR automates model checking



Underlying fact:

- P does not satisfy ψ

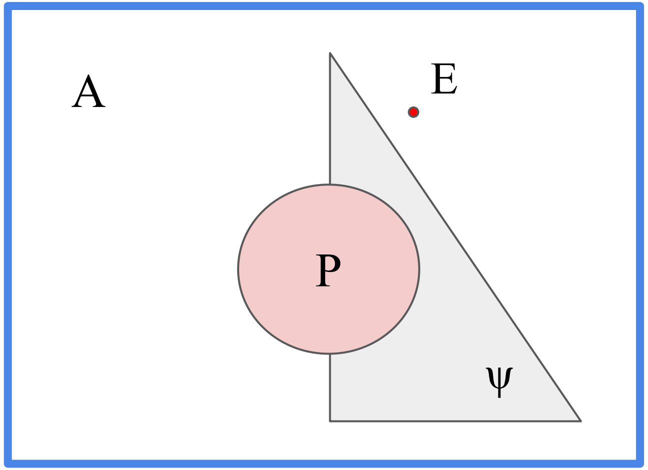
CEGAR automates model checking



Underlying fact:

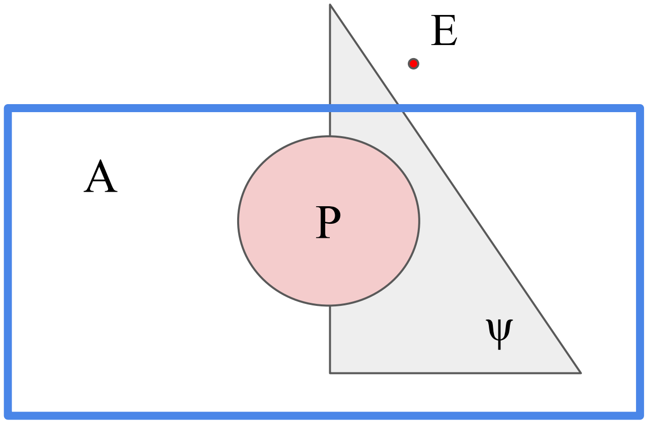
- P does not satisfy ψ

CEGAR automates model checking



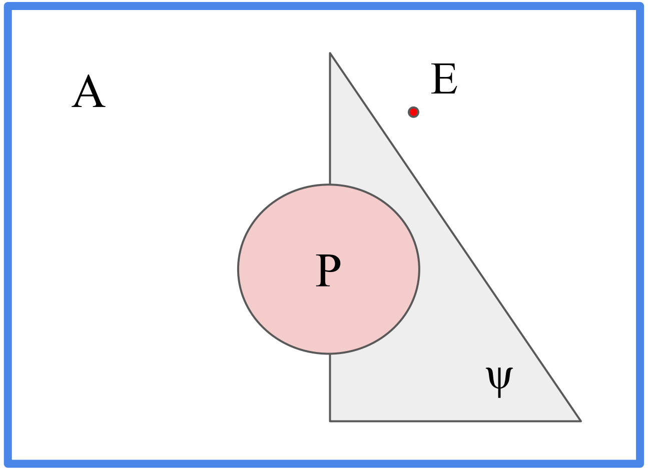
Underlying fact:

- P does not satisfy ψ



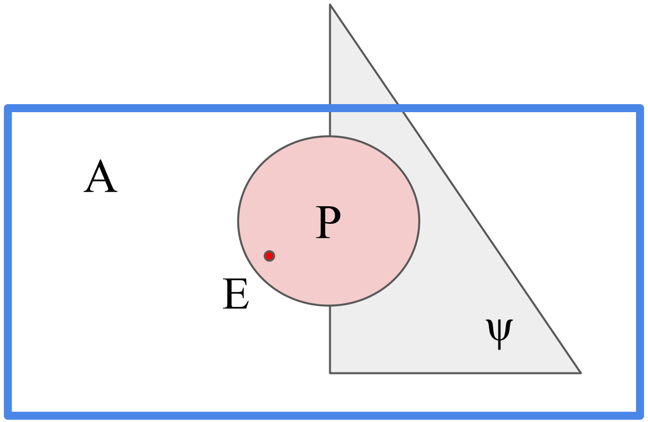
Refine A that eliminates E

CEGAR automates model checking



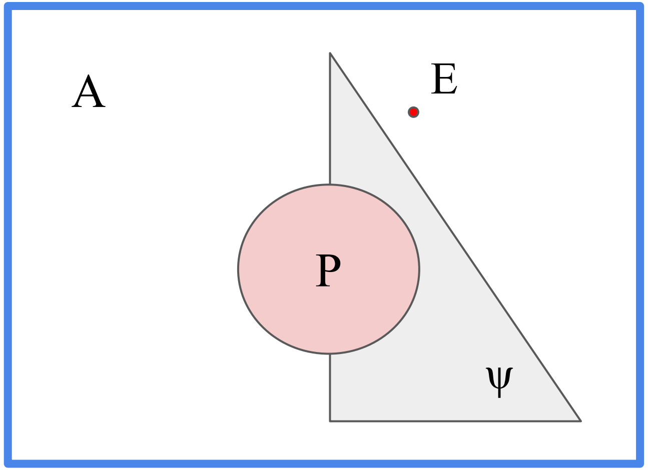
Underlying fact:

- P does not satisfy ψ



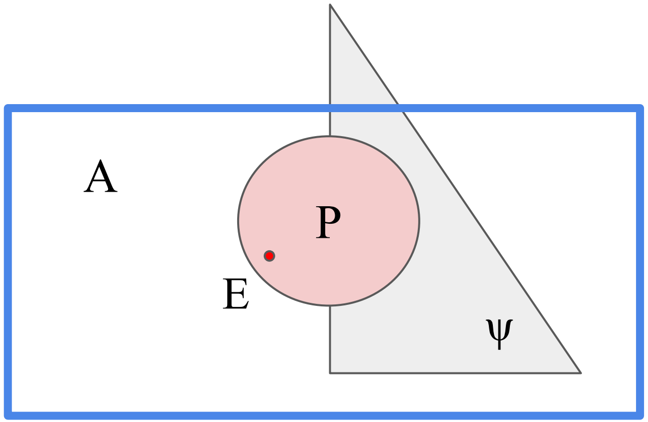
Find another counterexample E

CEGAR automates model checking



Underlying fact:

- P does not satisfy ψ



Derived fact:

- P does not satisfy ψ

CEGAR automates model checking

Given arbitrary property ψ and a program P, check ψ is satisfied in P.

1. Initialize abstraction A
2. Check if A satisfies ψ. If SATISFIED, done. Else, goto 3
3. Find counter-example E that brought UNSAT
4. Check if E is reachable in P. If so, report the instance. Else, goto 5
5. Refine abstraction $\mathrm{A} \coloneqq \mathrm{A}'$ that eliminates E. Goto 2

CEGAR is able to verify design and find bug

- Verified several designs including Fujitsu IP core and PCI-bus design.
- Verified a benchmark of which existing techniques were failed to do so.
- Found a bug that had not been discovered.

CEGAR should deal with NP-hardness

- Coarsest abstraction refinement is NP-hard.
- Sacrificing coarseness and using heuristics alleviates the hardness.

What is omitted

- Preliminaries about temporal logic, ACTL
- Finding counterexamples in ACTL settings
- Abstraction refinement algorithm in ACTL settings

Questions