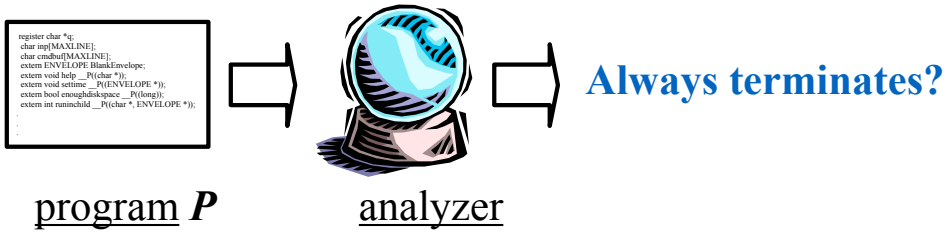
What is **Static Analysis?**

The Halting Problem

- Can we write an analyzer that can prove, for any program *P* and inputs to it, *P* will terminate
  - Doing so is called the **halting problem**



- Unfortunately, the halting problem is **undecidable**
  - That is, it is **impossible** to write such an analyzer: it will fail to produce an answer for at least some programs (and/or some inputs)

Some material inspired by work of Matt Might: http://matt.might.net/articles/intro-static-analysis/

Other properties?

- Perhaps security-related properties are feasible
  - E.g., that all accesses `a[i]` are in bounds
- *But* these **properties can be converted into the halting problem** by transforming the program
  - I.e., a perfect array bounds checker could solve the halting problem, which is impossible!
- Other undecidable properties (Rice's theorem)
  - Does this **SQL string** come from a **tainted source**?
  - Is this **pointer used after** its memory is **freed**?
  - Do any variables experience **data races**?

Halting ≈ Index in Bounds

- Proof by transformation
  - Change indexing expressions `a[i]` to exit
    - `(i >= 0 && i < a.length) ? a[i] : exit()`
    - Now all array bounds errors instead result in termination
  - Change program exit points to out-of-bounds accesses
    - `a[a.length+10]`

- Now if the array bounds checker
  - … **finds an error**, then the original program **halts**
  - … claims there are **no such errors**, then the original program **does not halt**
  - … **contradiction!**
    - with undecidability of the halting problem

Static analysis is impossible?

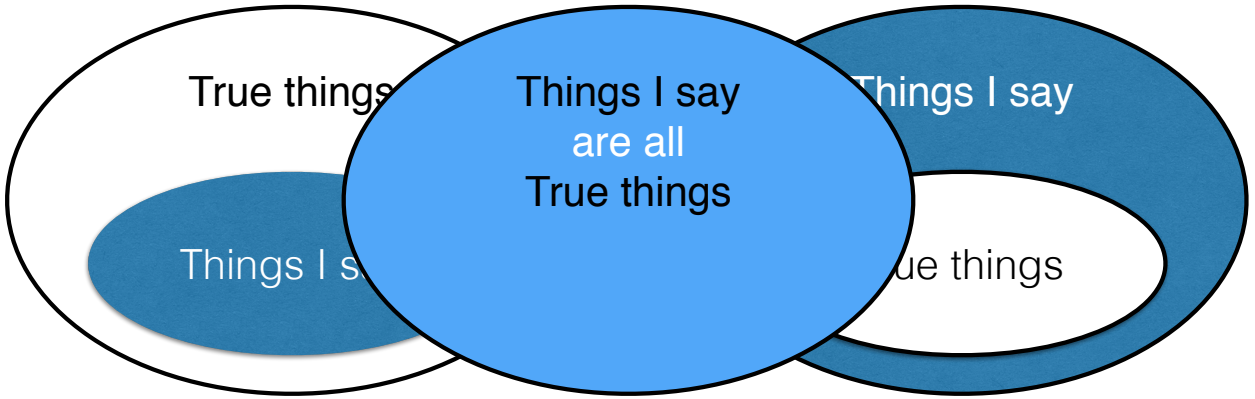
- **Perfect** static analysis is **not possible**
- **Useful** static analysis is **perfectly possible**, despite
  1. **Nontermination** - analyzer never terminates, or
  2. **False alarms** - claimed errors are not really errors, or
  3. **Missed errors** - no error reports ≠ error free
- Nonterminating analyses are confusing, so tools tend to exhibit only false alarms and/or missed errors
  - Fall somewhere between **soundness** and **completeness**

Soundness

If analysis says that X is true, then X is true.

Completeness

If X is true, then analysis says X is true.



Trivially Sound: Say nothing

Trivially Complete: Say everything

Sound and **Complete**:
Say exactly the set of true things

Stepping back

- **Soundness**: if the program is claimed to be error free, then it really is
  - *Alarms do not imply erroneousness*
- **Completeness**: if the program is claimed to be erroneous, then it really is
  - *Silence does not imply error freedom*
- Essentially, most interesting analyses
  - are neither **sound** nor **complete** (and not **both**)
  - … usually *lean* toward soundness (“soundy”) or completeness

The Art of Static Analysis

- Analysis design tradeoffs
  - **Precision**: Carefully model program behavior, to minimize false alarms
  - **Scalability**: Successfully analyze large programs
  - **Understandability**: Error reports should be actionable
- Observation: **Code style is important**
  - Aim to be precise for “good” programs
    - It’s OK to forbid yucky code in the name of safety
  - False alarms viewed positively: reduces complexity
    - Code that is more understandable to the analysis is more understandable to humans