Formal Reasoning about Software Systems

Mooly Sagiv
msagiv@acm.org
03-640-7606
Tel Aviv University
Sunday 14-16

http://www.cs.tau.ac.il/~msagiv/courses/rsys.html

Seminar Perquisites

- A course in formal methods
  - Courses by Shachar Maoz, Noam Rinetzky, Alex Rabinovich, Mooly Sagiv
    - Hardware & Software verification
    - Static analysis
    - Modeling systems
    - Automatic software verification

Seminar Requirements

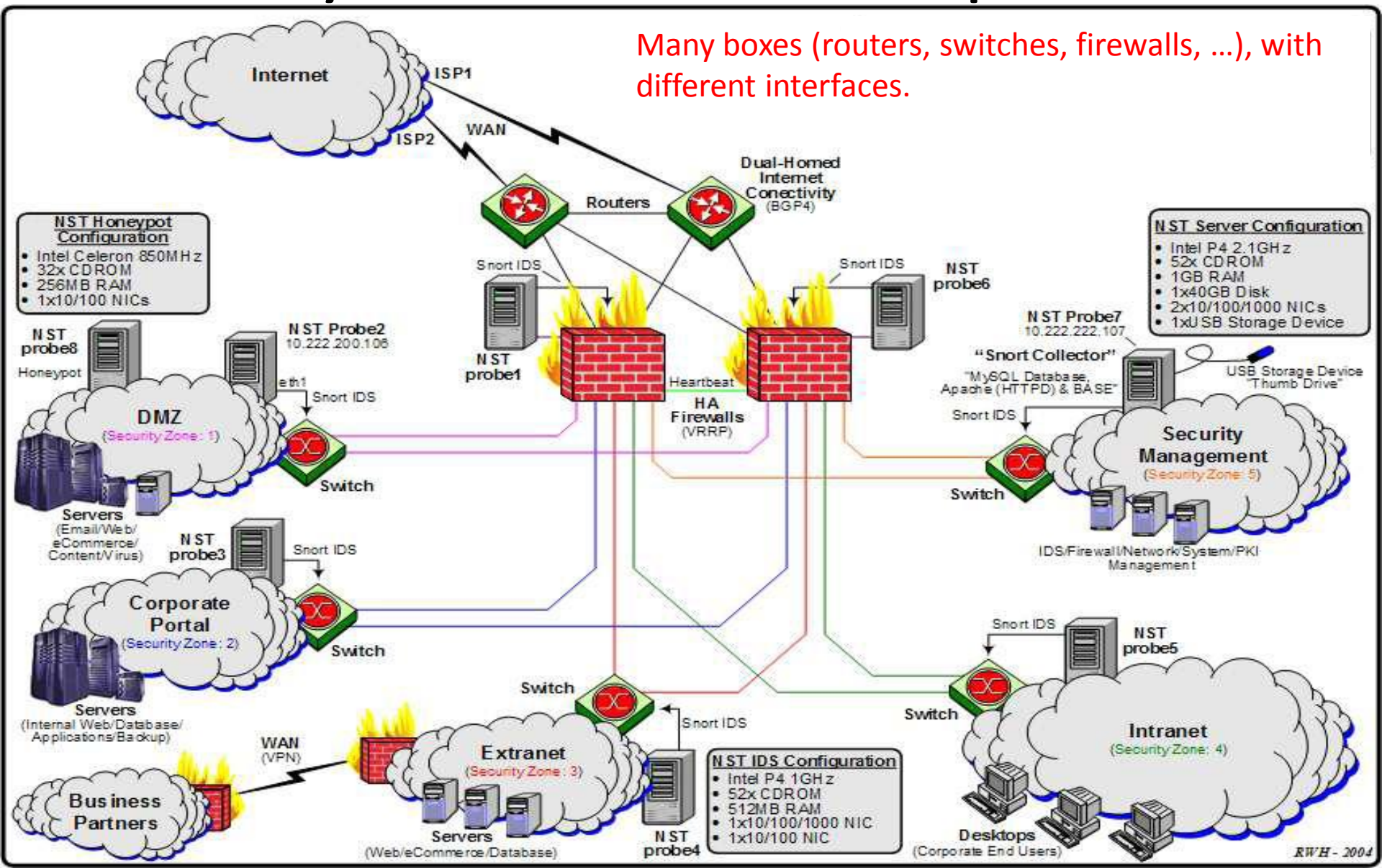
- Suggest three articles by Sunday 6/3 by email to msagiv@acm.org
- Receives selected article and date on Thursday 3/3
- Critically read the article
- Present the article (with slides) in the seminar
  - Present the article to instructor on Tuesday week before class with slides
- Actively participate in all seminars
  - Can skip one with reason

System's Research

- Develop reliable systems
- Long tradition of innovations:
  - Compilers
  - Operating Systems
  - Networks
    - Internet: 8 orders of magnitude of scaling
    - Software defined networks
  - Virtual machines
  - Garbage collectors
- Several highly competitive publication venues: SOSP, OSDI, NSDI, PLDI, …

Systems can be complex

Many boxes (routers, switches, firewalls, …), with different interfaces.



Why formally verify Systems?

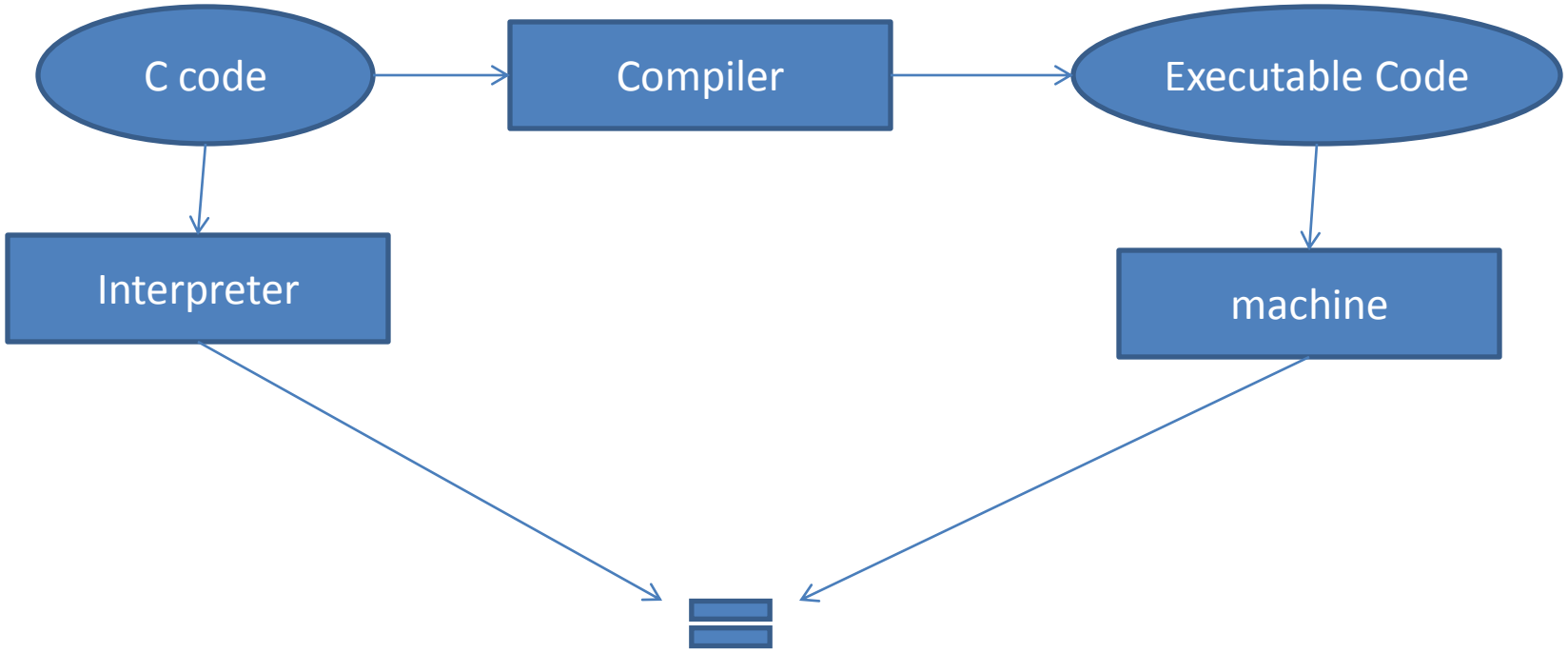
- Next level after the hardware
- Complexity
- Bugs in system affects all applications:
  - A bug in the compiler / OS / network can hurt every application
- Reliability
- The cost of bugs
- Stability of systems
- Maturity of the field
- Enable more innovation
- Maturity of formal method tools

Formal methods

- Formally prove that a system implementation satisfies the required specification
- Automatically identify bugs before the program executes
- Challenges:
  - Specification: How to formally specify what is the correct behavior (can you formally specify what MS-Excel should do? How about autonomous car?)
  - Verification: Given a system and a specification, check if the system meets the specification

Verified compilers

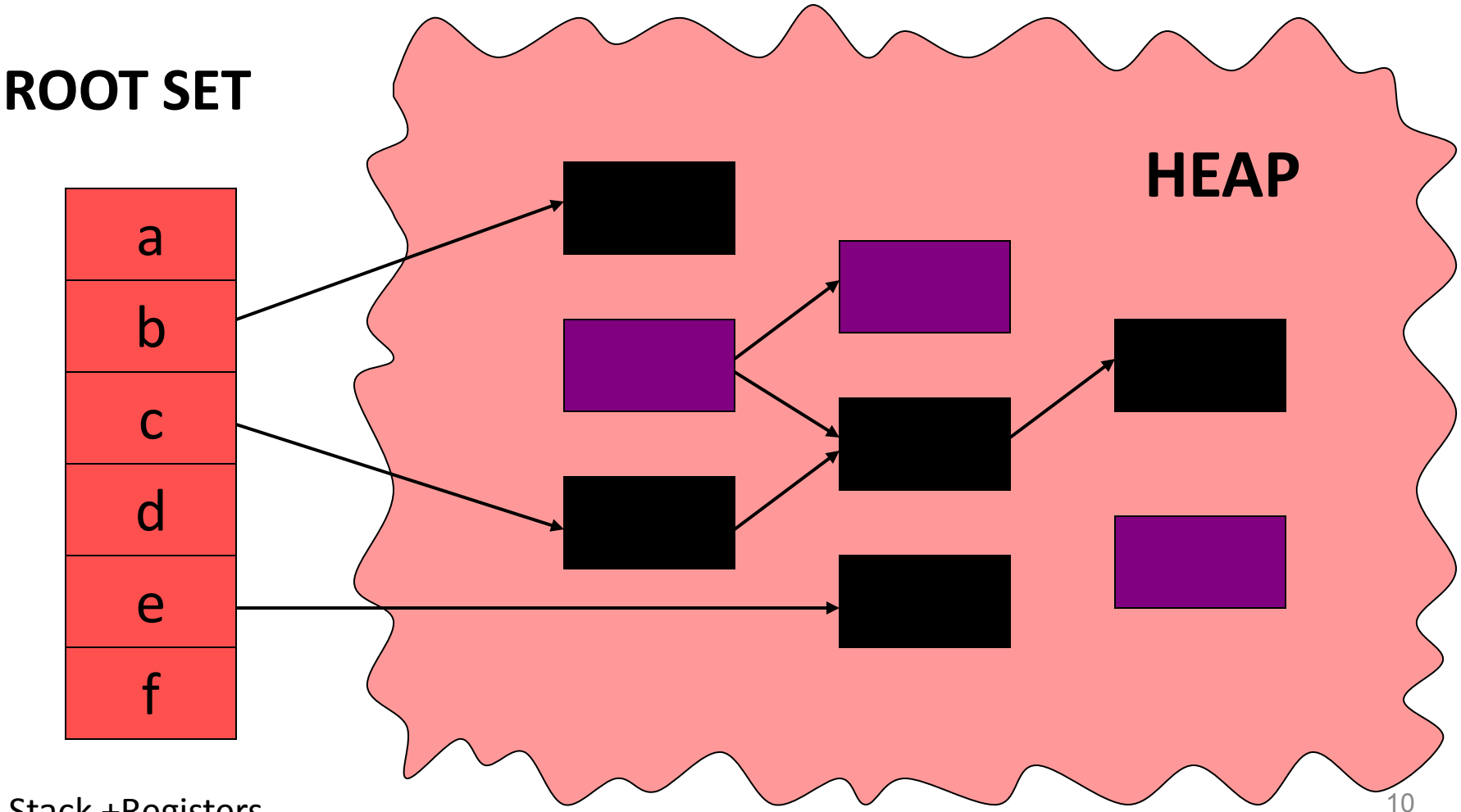
- The compiler maintains the abstraction



Verified compilers: Why bother?

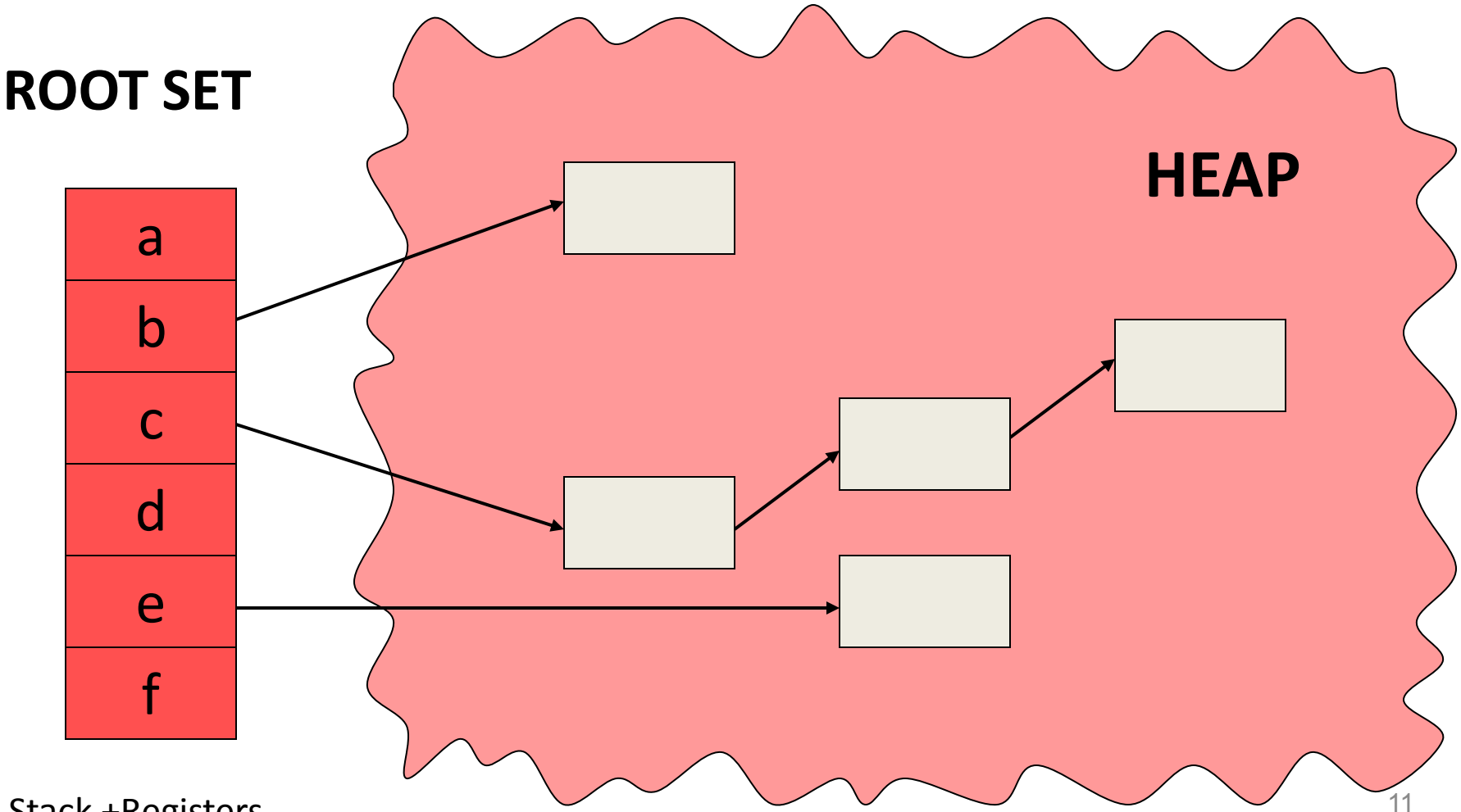
- If the compiler has a bug – every complied program is (possibly) wrong
- The compiler is a complicated program
  - Many components
  - Tricky optimizations
  - Tricky machine semantics
- Many tricky bugs
  - Even after two decades of testing
- Especially important for safety critical code

Garbage collection



Stack +Registers

Garbage collection



Stack +Registers

What is garbage collection

- The runtime environment reuses chunks that were allocated but are no longer needed
- Garbage chunks
  - not live
- It is undecidable to find the garbage chunks:
  - Decidability of liveness
  - Decidability of type information
- Conservative collection
  - every live chunk is identified
  - some garbage runtime chunk are not identified
- Find the reachable chunks via pointer chains
- Often done in the allocation function

```
int *p; int x;
p = malloc();
x = (int) p;
p= null;
// is it safe to free  here?
*((int*)x) = 5
```

Garbage collection vs. Explicit memory deallocation

- Faster program development
- Less error prone
- Can lead to faster programs
  - Can improve locality of references
- Support very general programming styles, e.g. higher order and OO programming
- Standard in ML, Java, C#
- Supported in C and C++ via separate libraries

- May require more space
- Needs a large memory
- Can lead to long pauses
- Can change locality of references
- Effectiveness depends on programming language and style
- Hides documentation
- More trusted code

Interesting aspects of garbage collection

- Data structures
- Non constant time costs
- Amortized algorithms
- Constant factors matter
- Interfaces between compilers and runtime environments
- Interfaces between compilers and virtual memory management
- Tricky code

Verified garbage collection

- Any reachable object is not reclaimed
- Any non-reachable object is reclaimed

Operating Systems

- Implement an abstraction of the machine
- “Single-user”/”Single task” illusion
- Unbounded unshared local memory
- Persistent memory (filesystem)
- A lot of useful functions for resource allocation
- Many security and permissions aspects

What does the operating system provide?

- Abstract machine
- Scheduling
- Isolation
  - Virtual memory
- Filesystem
  - Consistency
  - Failures
- …

Virtual machines

- Abstracts the operating system
- Enables portability
- Some support for high level programming
- Examples: JVM, .NET

Network verification

- Relatively new topic
- Every packet sent will eventually received
- No SSL packet from A will arrive to B unless B requested
- Becomes more important as network is shifted from hardware to software
  - Software defined networks
  - Network function virtualization
  - Middleboxes

What can you expect to learn from the seminar

- Critical thinking
- Specifying interesting properties of systems
- Verifying interesting properties

Verification Techniques

Formal Verification Techniques

- Model checking
  - Finite state model checking
  - Bounded model checking
- Concolic testing

- Abstract interpretation
- Deductive verification
- Constructive mathematics

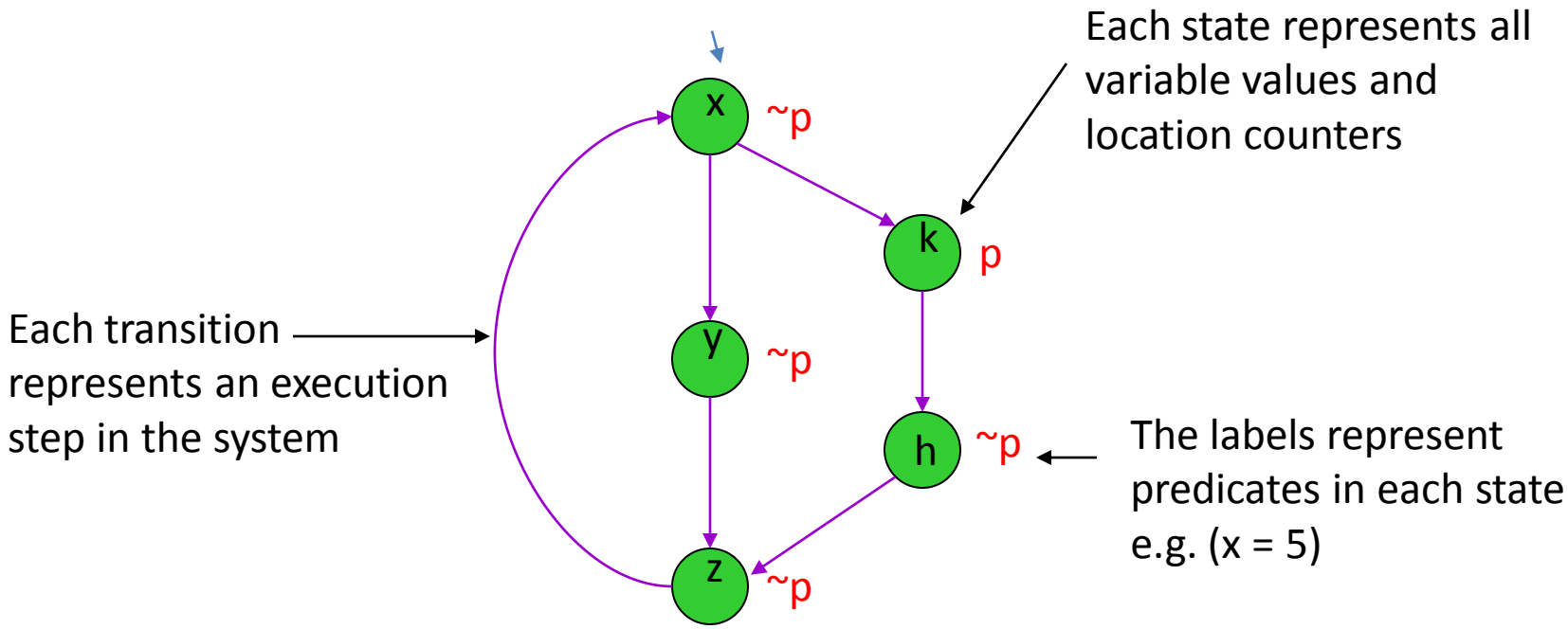
Model Checking

Taken from Willem Visser(NASA)

Model Checking
The Intuition

- Calculate whether a system satisfies a certain behavioral property:
  - Is the system deadlock free?
  - Whenever a packet is sent will it eventually be received?
- So it is like testing? No, major difference:
  - Look at *all* possible behaviors of a system
- Automatic, if the system is finite-state
  - Potential for being a push-button technology
  - Almost no expert knowledge required
- How do we describe the system?
- How do we express the properties?

Labeled State Graph
Kripke Structure



K = ({p,~p},{x,y,z,k,h},R,{x},L)

Safety and Liveness

- Safety properties
  - Invariants, deadlocks, reachability, etc.
  - Can be checked on finite traces
  - “something bad never happens”
- Liveness Properties
  - Fairness, response, etc.
  - Infinite traces
  - “something good will eventually happen”

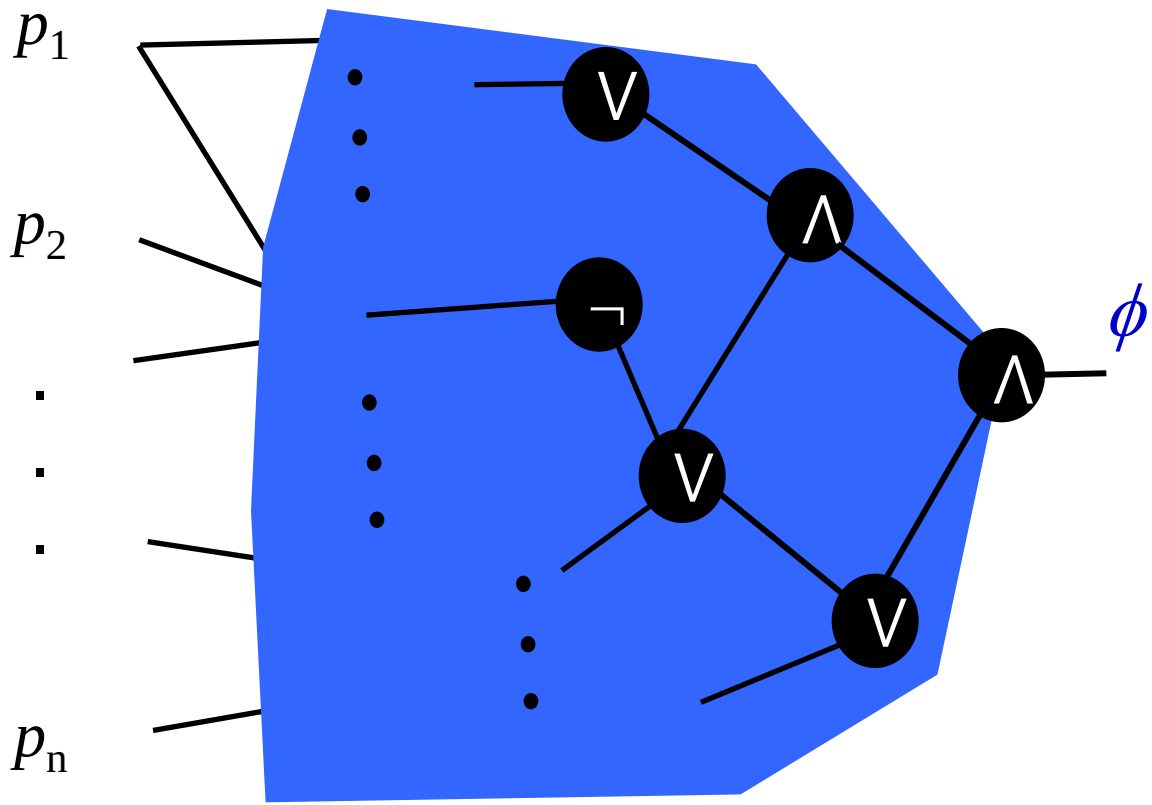
Model Checking

- Does a given model M satisfy a property P, $M \models P$
  - M is usually a finite directed graph
  - P is usually a formula in temporal logic
- Examples:
  - Is every request to this bus arbiter eventually acknowledged?
  - Does this program every dereference a null pointer?

Bounded Model Checking

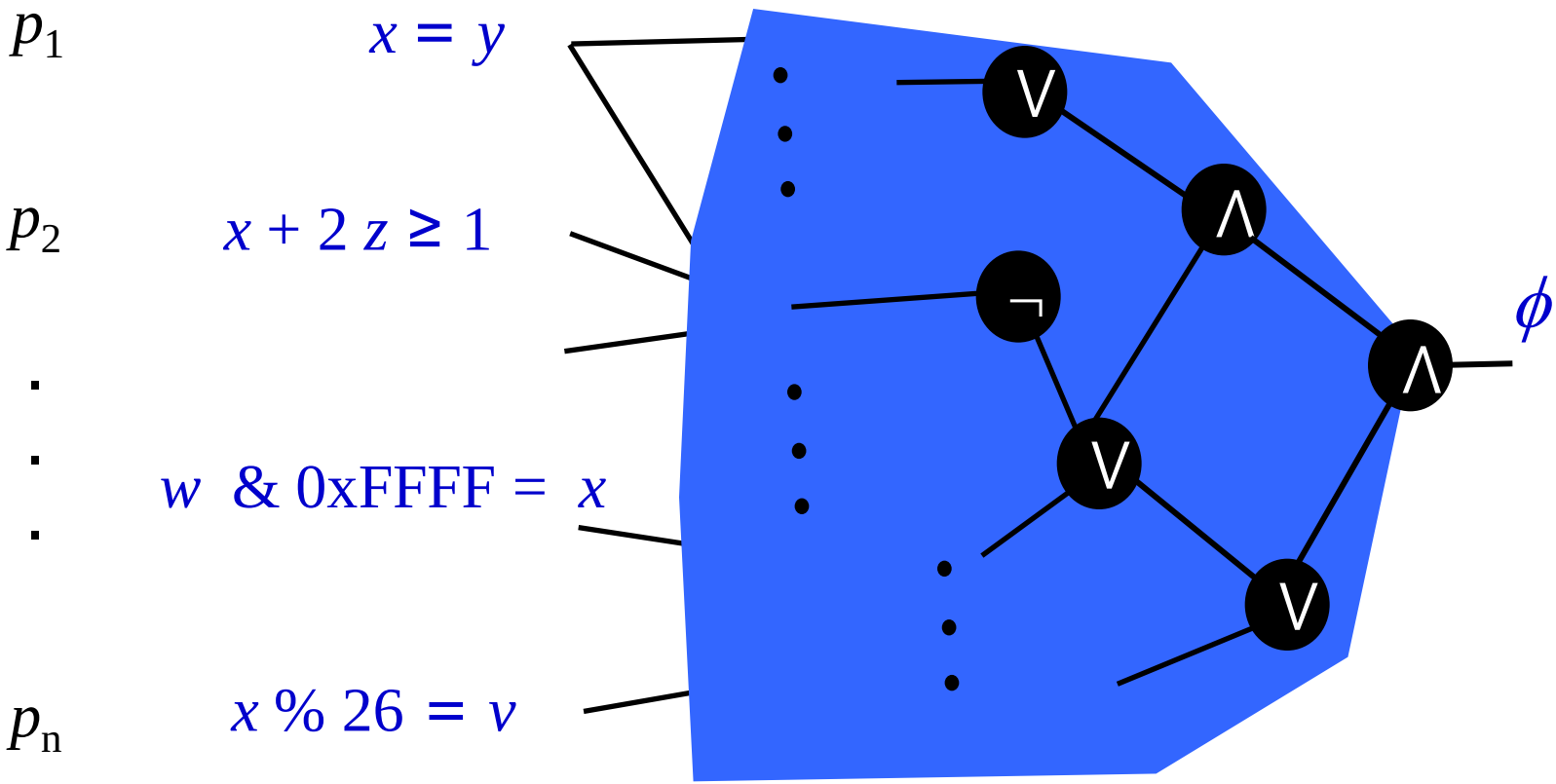
Taken from Arie Gurfinkel (CMU)

Boolean Satisfiability (SAT)



Is there an assignment to the $p_1, p_2, \ldots, p_n$ variables such that ϕ evaluates to 1?

Satisfiability Modulo Theories



Is there an assignment to the x,y,z,w variables s.t. ϕ evaluates to 1?

Bounded Model Checking

- Given
  - A finite transition system M
  - A property P
- Determine
  - Does M allow a counterexample to P of *k transitions of fewer*?

This problem can be translated to a SAT problem

Bounded Model Checking of Loops

- Does the program reach an error within at most k unfolding of the loop
- Special kind of symbolic evaluation

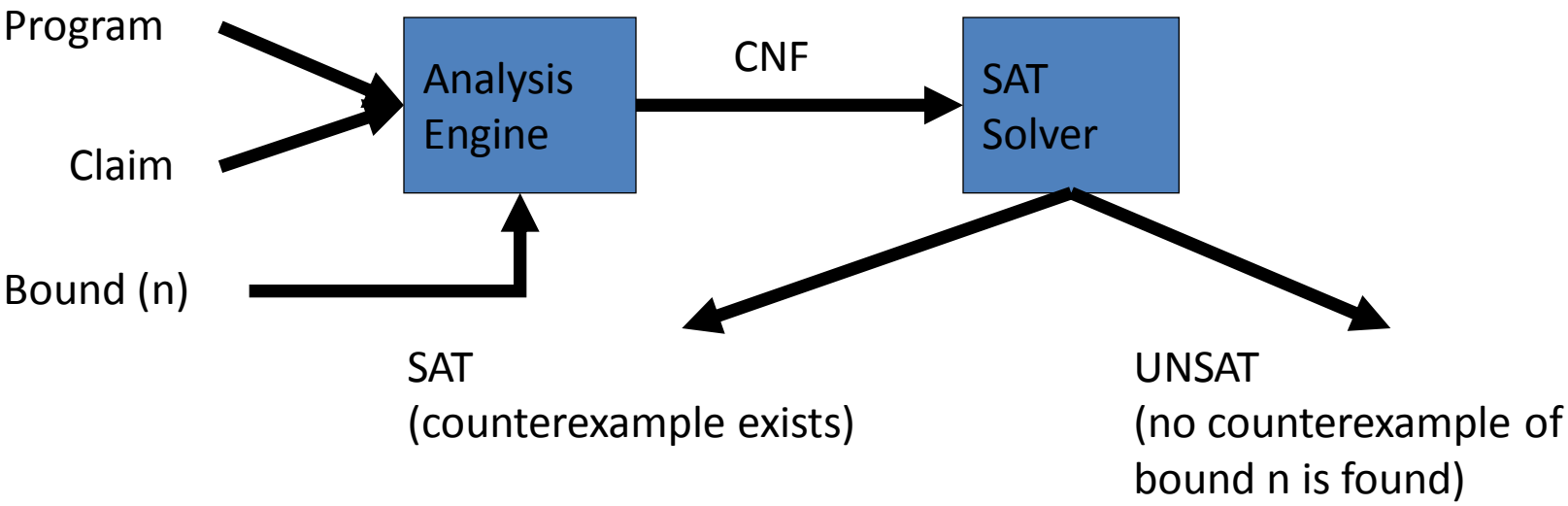
CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cs.cmu.edu/~modelcheck/cbmc/
- Supported platafoms: Windows (requires VisualStudio's` CL), Linux
- Provides a command line and Eclipse-based interfaces

- Known to scale to programs with over 30K LOC
- Was used to find previously unknown bugs in MS Windows device drivers

What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)



How does it work

- Transform a programs into a set of equations

1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Bit-blast
6. Solve with a SAT Solver
7. Convert SAT assignment into a counterexample

Control Flow Simplifications

- All side effect are removal
  - `e.g., j=i++` becomes `j=i;i=i+1`
- Control Flow is made explicit
  - `continue`, `break` replaced by `goto`
- All loops are simplified into one form
  - `for`, `do while` replaced by `while`

Loop Unwinding

- All loops are unwound
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special “unwinding assertion” claims are added
- If a program satisfies all of its claims and all unwinding assertions then it is correct!
- Same for backward `goto` jumps and recursive functions

Loop Unwinding

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

        }
      }
    }
  }
  Remainder;
}
```

Unwinding assertion

- while() loops are unwound iteratively
- Break / continue replaced by goto
- Assertion inserted after last iteration: violated if program runs longer than bound permits
- Positive correctness result!

Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
        }
      }
    }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 10)
    j = j + 1;
  Remainder;
}
```

unwind = 3

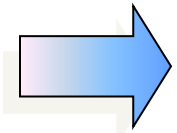
```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
        }
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (1)

- Easy to transform when every variable is only assigned once!

Program

```
x = a;
y = x + 1;
z = y - 1;
```



Constraints

x = a &&
y = x + 1 &&
z = y – 1 &&

Transforming Loop-Free Programs Into Equations (2)

- When a variable is assigned multiple times,
- use a new variable for the RHS of each assignment

Program

```
x=x+y;
x=x*2;
a[i]=100;
```

ρ

SSA Program

$x_1 = x_0 + y_0;$

$x_2 = x_1 * 2;$

$a_1[i_0] = 100;$

What about conditionals?

Program

```
if (v)
  x = y;
else
  x = z;

w = x;
```

ρ

SSA Program

```
if (v0)
  x0 = y0;
else
  x1 = z0;

w1 = x??;
```

What should ‘x’ be?

What about conditionals?

Program

```
if (v)
  x = y;
else
  x = z;

w = x;
```

ρ

SSA Program

```
if (v0)
  x0 = y0;
else
  x1 = z0;
x2 = v0 ? x0 : x1;
w1 = x2
```

- For each join point, add new variables with selectors

Example

```
int main() {
   int x, y;
   y=8;
   if(x)
     y--;
   else
     y++;

   assert
       (y==7 ||
       y==9);
}
```

ρ

```
int main() {
   int x, y;
   y1=8;
   if(x0)
     y2=y1-1;
   else
     y3=y1+1;
   y4= x0 ?y2:y3;
   assert
       (y4==7 ||
       y4==9);
}
```

$$
\begin{aligned}
&(\quad y_1 = 8 \\
&\wedge \quad y_2 = y_1 - 1 \\
&\wedge \quad y_3 = y_1 + 1 \\
&\wedge \quad y_4 = x_0 \,?\, y_2 \,:\, y_3 \,) \\
&\Longrightarrow \quad (y_4 = 7 \vee y_4 = 9)
\end{aligned}
$$

Concolic Testing

Concolic Testing

- Combine runtime testing and symbolic execution
- Runtime testing
  - Effectiveness depends on input test
- Symbolic Execution

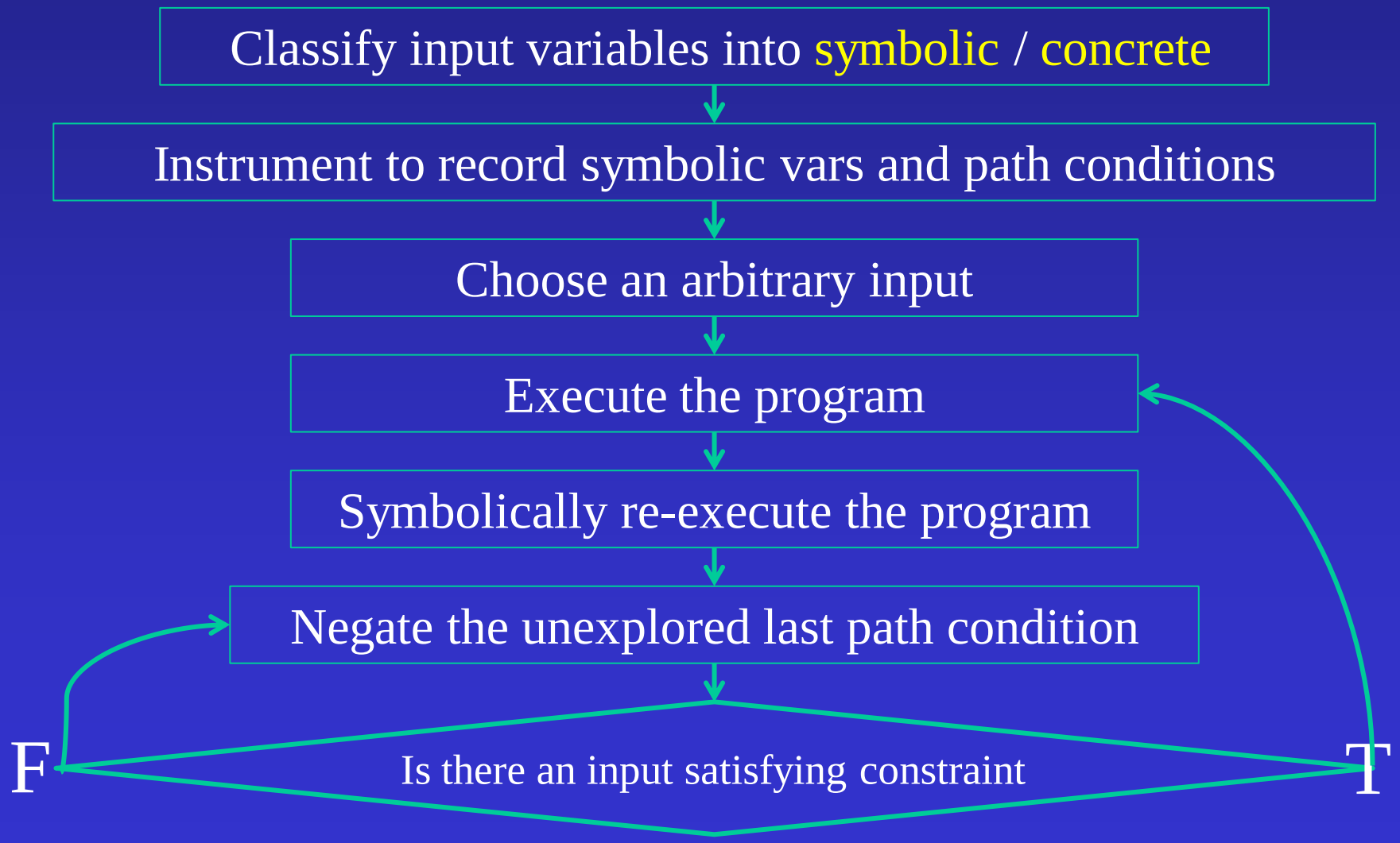
```
read(x);
y = 2 * x ;
assert y != 12;
```

  - Need constraint solver
  - Can be complex
- Concolic testing aims to improve both

A Motivating Example

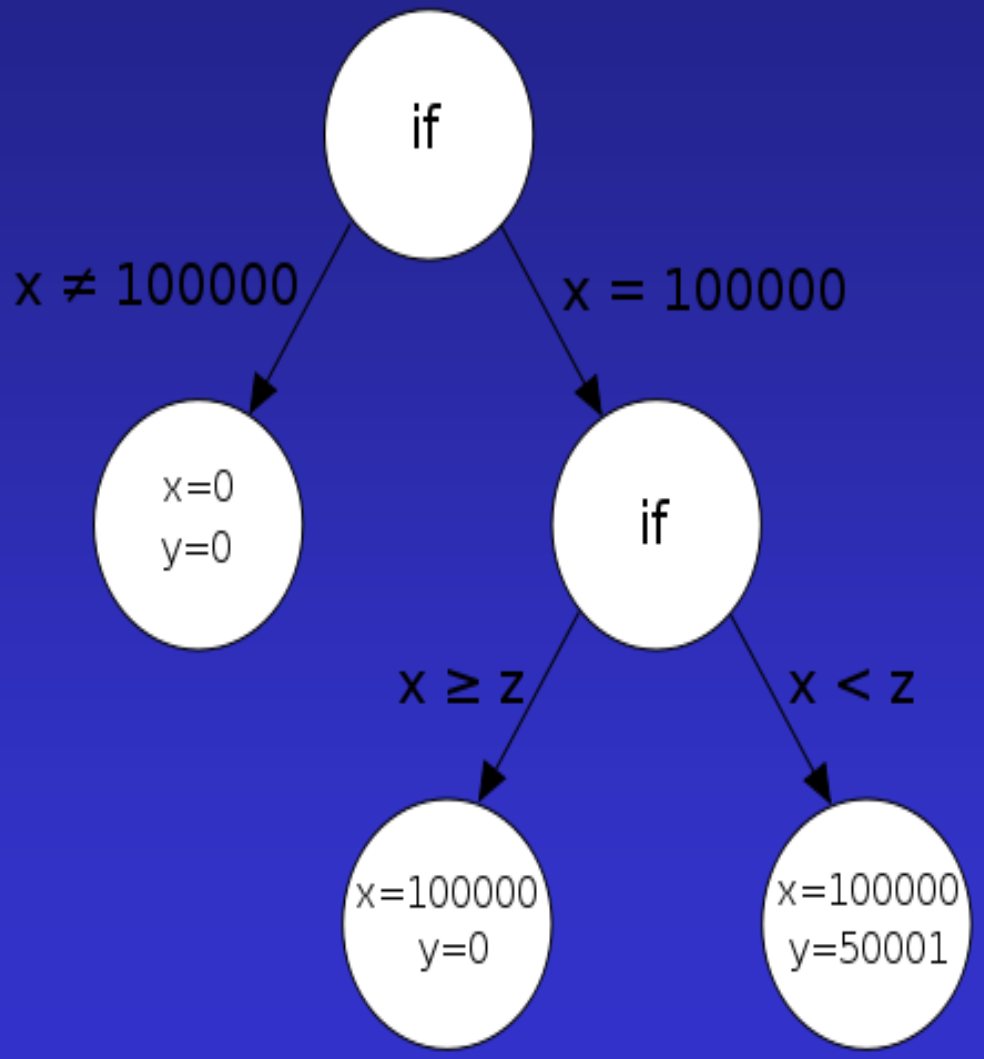
```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```

The Concolic Testing Algorithm



Example Concolic Testing

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```



Summary Concolic Testing

- Quite effective:
  - SAGE (Microsoft Research)
  - Datarace detection (Candea, EPFL)
- Instrumentation can be tricky
- Scalability is an issue
- Coverage is an issue
- Limitations of theorem provers
- Data structures

Abstract Interpretation

Abstract Interpretation

- Automatically prove that the program is correct by also considering infeasible executions
- Abstract interpretation of program statements/conditions
- Conceptually explore a superset of reachable states
- Sound but incomplete reasoning
- Automatically infer sound inductive invariants

Automatic Program Verification

Program
P

Desired
Properties φ

Solver

Is there a behavior of P that violates φ?

Counterexample

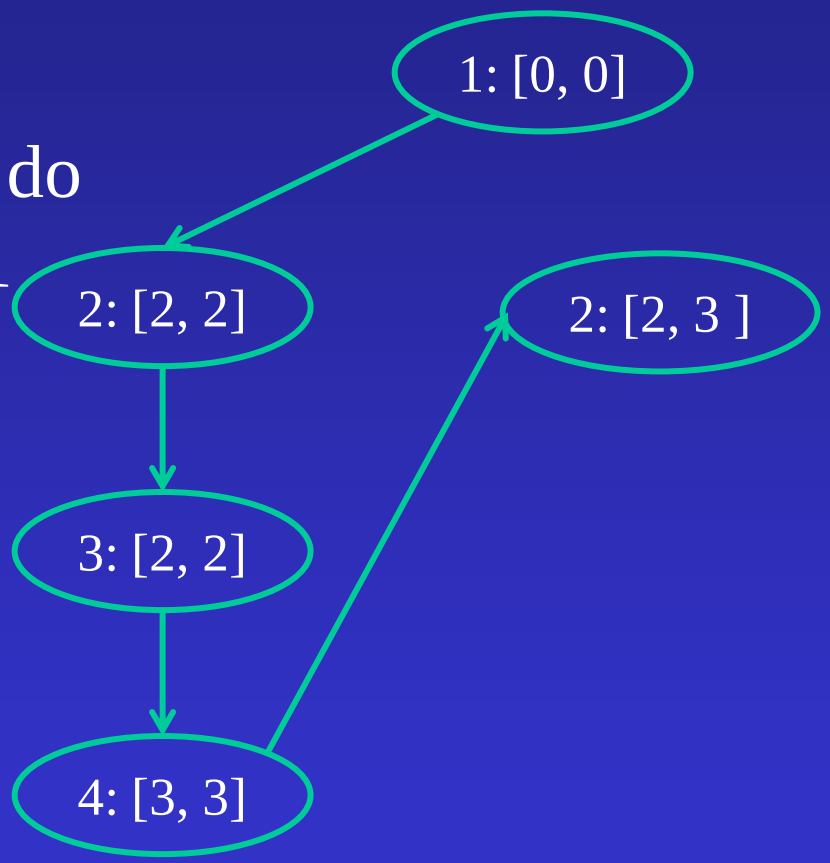
Unknown

Proof

Interval Based Abstract Interpretation

pc: int(x)

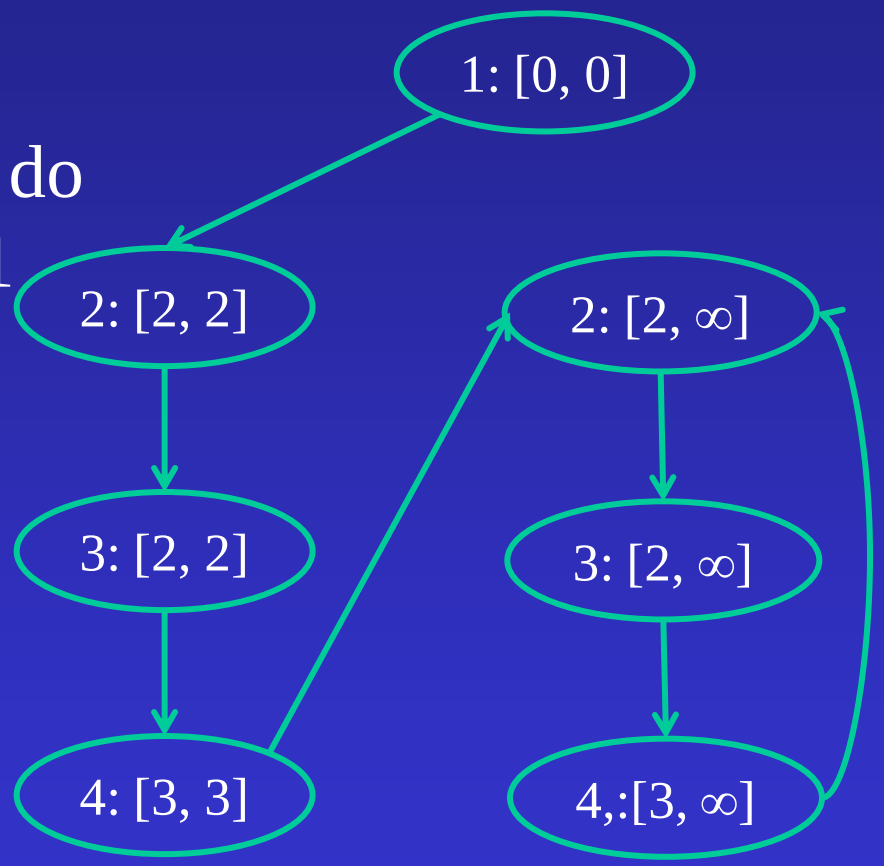
1: x = 2;
2: while true {x > 0} do
3: x = 2* x – 1
4:



Interval Based Abstract Interpretation

pc: int(x)

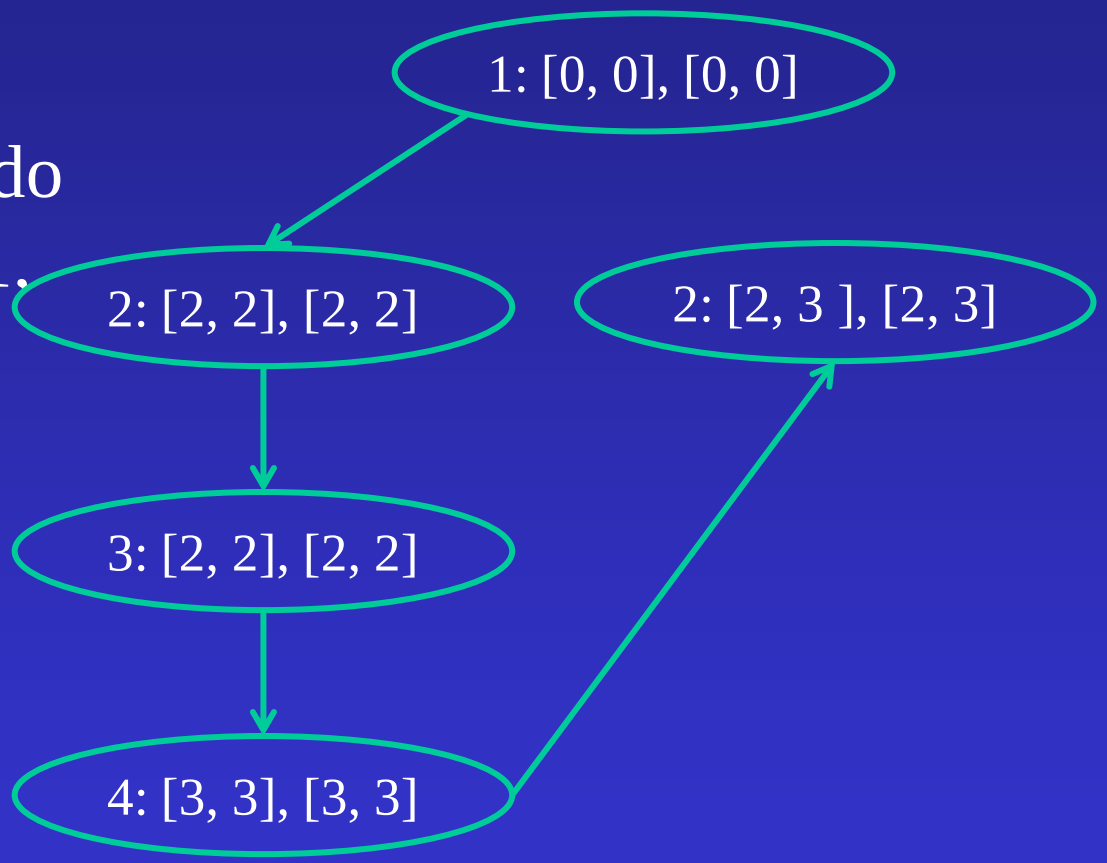
1: x = 2;
2: while true {x > 0} do
3: x = 2* x – 1
4:

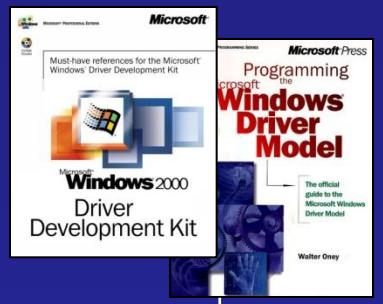


Interval Based Abstract Interpretation

pc: int(x), int(y)

1: x = 2, y = 2
2: while true {x =y} do
3: x = 2* x – 1,
y = 2*y -1
4:





Rules

Read for understanding

New API rules

Static Driver Verifier

Precise API Usage Rules (SLIC)

Drive testing tools

Development

Defects

SLAM

Software Model Checking

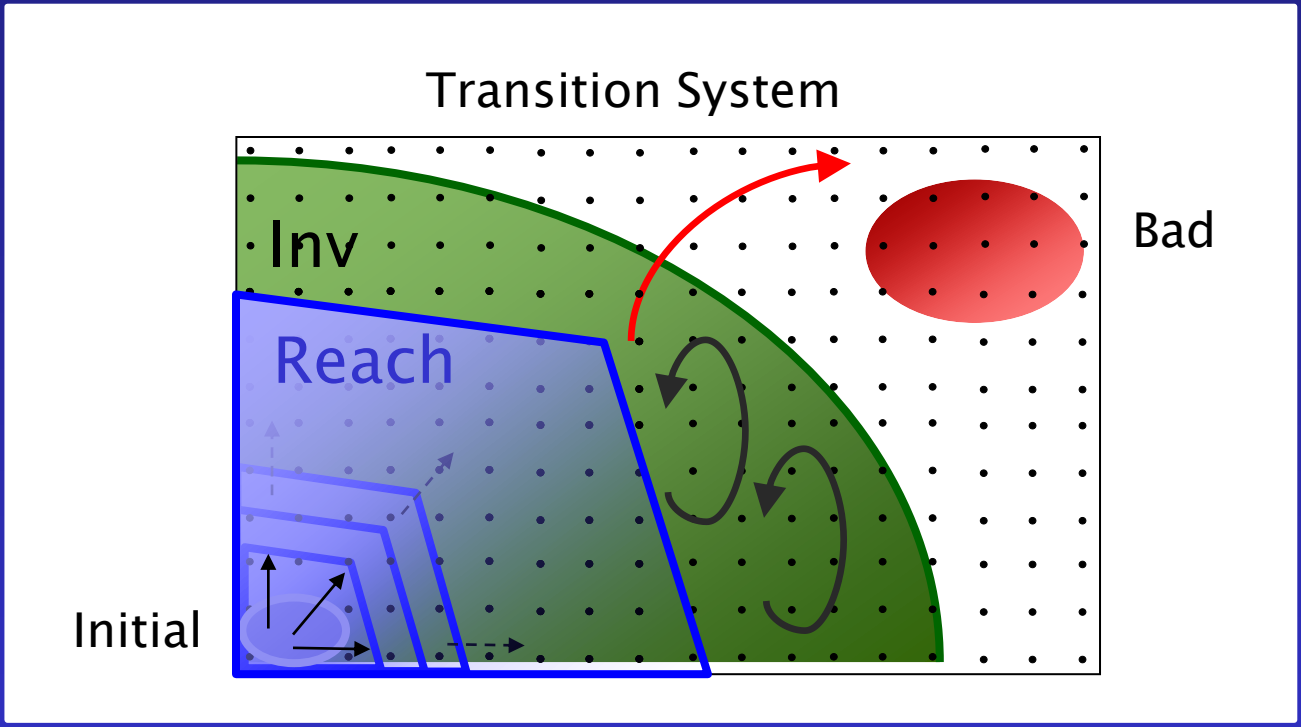
Testing

100% path coverage

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability" ***Bill Gates, 2002***

Deductive Verification

Inductive Invariants



System S is **safe** if all the reachable states satisfy the property

System S is safe iff there exists an **inductive invariant** Inv:

Inv $\Rightarrow$ $\varphi = \neg$Bad (Safety)
Init $\Rightarrow$ Inv (Initiation)
if $\sigma \vDash$ Inv and $T(\sigma, \sigma')$ then $\sigma' \vDash$ Inv (Consecution)

Deductive Verification

Program
P

Candidate Invariant I

Safety
Property φ

Solver

Is there a behavior
of P that violates the inductiveness of I?

Counterexample to induction
(CTI)

Proof

Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥1;
   4:  x:= x + y;
   5:  y := y + 1
}
6:
```

$at(3) \Rightarrow x \geq 1$

$at(3) \Rightarrow x \geq 1$

Solver

Is there a behavior of P that violates the inductiveness of I?

3: <1, -2>

x := x+y; y :=y+1

$\neg (at(3) \Rightarrow x \geq 1)$

Deductive Verification

```
1: x := 1;
2: y := 2;
while * do {
   3: assert x ≥1;
   4:  x:= x + y;
   5:  y := y + 1
}
6:
```

$at(3) \Rightarrow x \geq 1 \wedge y \geq 0$

$at(3) \Rightarrow x \geq 1$

Solver

Is there a behavior of P that violates the inductiveness of I?

Proof

IronFleet: Proving Practical Distributed Systems Correct [SOSP'15]

- Employs deductive verification
- Useful for verifying real systems

- Writing invariants is hard
- Deduction is hard
  - Quantifier alternations leads to matching loops
  - Complicated arithmetic

Challenges

1. Specifying safety properties
2. Inductive Invariants for Deductive Verification
   - Hard to express
   - Hard to change
   - Hard to infer
3. Deduction
   - Reasoning about inductive invariants
     - Undecidability of implication checking

The Coq Proof Assistant

Xavier Rival

Wei Hu

Constructive Mathematics

- The designer expresses the correctness of the system and a “constructive” proof
  - No proof by contradiction
- The system checks each step of the proof
- If the proof succeeds a program which implements the system is automatically generated
- Existing tools Isabelle, Coq

What is a proof assistant ?

- Assists the user in:
  - defining the proof goals formally
  - setting up the structure of the proofs
  - making the proof steps
  - checking the consistency of the proof
  - extract executable code

Overall workflow

- Define the objects properties need be proved about Data-structures, base types, programs written in the Coq language
- Write and prove intermediate lemmas
- Write and prove the main theorems
- If needed, extract programs

Coq as a Programming Language

- ML-like

```
Fixpoint is_even (n:nat) : bool :=
  match n with
  | 0        => true
  | 1        => false
  | S (S n') => is_even n’
  end.

Eval compute in is_even 3.
false
```

- Restrictions
  - No side effects
  - No non-terminating programs (to avoid inconsistency)

Program Extraction

```
Fixpoint is_even (n:nat) : bool :=
  match n with
  | 0        => true
  | 1        => false
  | S (S n') => is_even n’
  end.
```

Coq

```
let rec is_even = function
  | O -> True
  | S n0 -> (match n0 with
              | O -> False
              | S n' -> is_even n')
```

OCaml

```
is_even n =
  case n of
    O -> True
    S n0 -> (case n0 of
              O -> False
              S n' -> is_even n')
```

Haskell

Summary (Techniques)

- A lot of usable techniques
  - Range from fully manual to fully automated
- Public domain software
- Interesting algorithms
- Order of magnitudes improvement in scaling (e.g. SAT)
- Still mainly academic topic

Seminar Summary

- A renaissance for formal methods
- Seminar topics combine complex software applications and deep mathematical theories
  - Can be very enjoyable – if you make an effort!
- Timeline
  - Sunday 6/3 (3 articles selected)
  - Monday 7/3 (receive selected article)