Automated Theorem Proving

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

Arnon Avron

Mooly Sagiv

Based on a presentations by Jonathan Aldrich ,Sorin Lerner & Tevfik Bultan

Course prerequisites

- Undergrad level logic
- Some familiarity with functional programming
  - mini-project will require you to write some LISP code

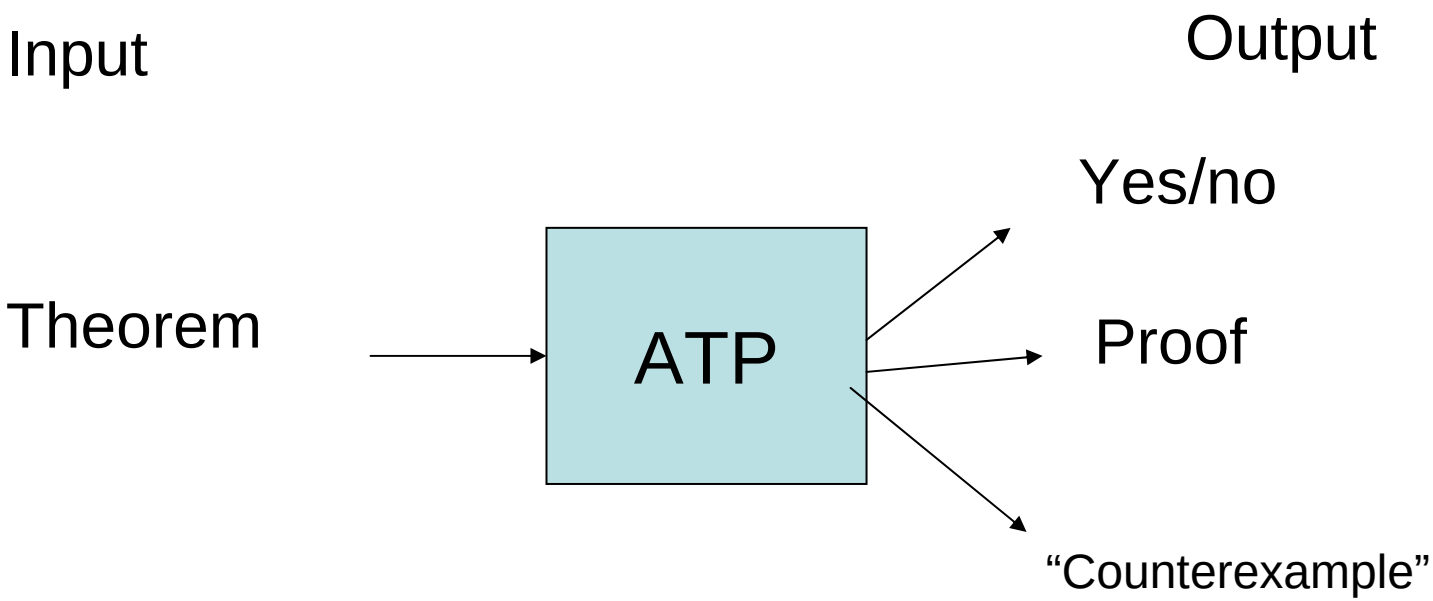
Course Requirements

- Summarize one class (10%)
- Homework
- Course project (35%)
- Exam (55%)

Course project

- Given a problem, you are asked to encode it in two (maybe three?) theorem provers
- Written report stating what worked, what didn’t, and what the differences were between the various theorem provers

What is an automated theorem prover?



Example theorems

- Pythagoras theorem: Given a right triangle with sides A B and C, where C is the hypotenuse, then $C^2 = A^2 + B^2$
- Fundamental theorem of arithmetic: Any whole number bigger than 1 can be represented in exactly one way as a product of primes

Example Theorem

- The program “z = x; z = z + y;” computes the sum of ‘x’ and ‘y’ in ‘z’ according to the semantics of C
- Program-Semantics $\Rightarrow$ Specification

Theorem

- Theorem must be stated in formal logic
  - self-contained
  - no hidden assumptions
- Many different kinds of logics (propositional logic, first order logic, higher order logic, linear logic, temporal logic)
- Different from theorems as stated in math
  - theorems in math are informal
  - mathematicians find the formal details too cumbersome

Human assistance

- Some ATPs require human assistance
  - e.g.: programmer gives hints a priori, or interacts with ATP using a prompt
- Hardest theorems to prove are “mathematically interesting” theorems (eg: Fermat’s last theorem)

Output

- Can be as simple as a yes/no answer
- May include proofs and/or counterexamples
- These are formal proofs, not what mathematicians refer to as proofs
- Proofs in math are
  - informal
  - “validated” by peer review
  - meant to convey a message, an intuition of how the proof works -- for this purpose the formal details are too cumbersome

Output: meaning of the answer

- If the theorem prover says “yes” to a formula, what does that tell us?
  - Soundness: theorem prover says yes implies formula is correct
  - Subject to bugs in the Trusted Computing Base (TCB)
  - Broad defn of TCB: part the system that must be correct in order to ensure the intended guarantee
  - TCB may include the whole theorem prover
  - Or it may include only a proof checker

Output: meaning of the answer

- If the theorem prover says “no” to a formula, what does that tell us?
  - Completeness: formula is correct implies theorem prover says yes
  - Or, equivalently, theorem prover says no implies formula incorrect
  - Again, as before, subject to bugs in the TCB

Output: meaning of the answer

- ATPs first strive for soundness, and then for completeness if possible
- Some ATPs are incomplete: “no” answer doesn’t provide any information
- Many subtle variants
  - refutation complete
  - complete semi-algorithm

A bit of early history

- 1929 M. Presburger shown that linear arithmetic is decidable
- 1954 M. Davis programmed this algorithm
  - Very slow
  - Even showing that the sum of even numbers is even is hard

More history

- The field of automated theorem proving started in the 1960s
  - SAT and reduction to SAT (early 60s)
  - Resolution (Robinson 1965)
  - Lots of enthusiasm, and many early efforts
  - Was considered originally part of AI
- In the 70s
  - some of the original excitement settles down, with the realization that “interesting” theorems are hard to prove automatically

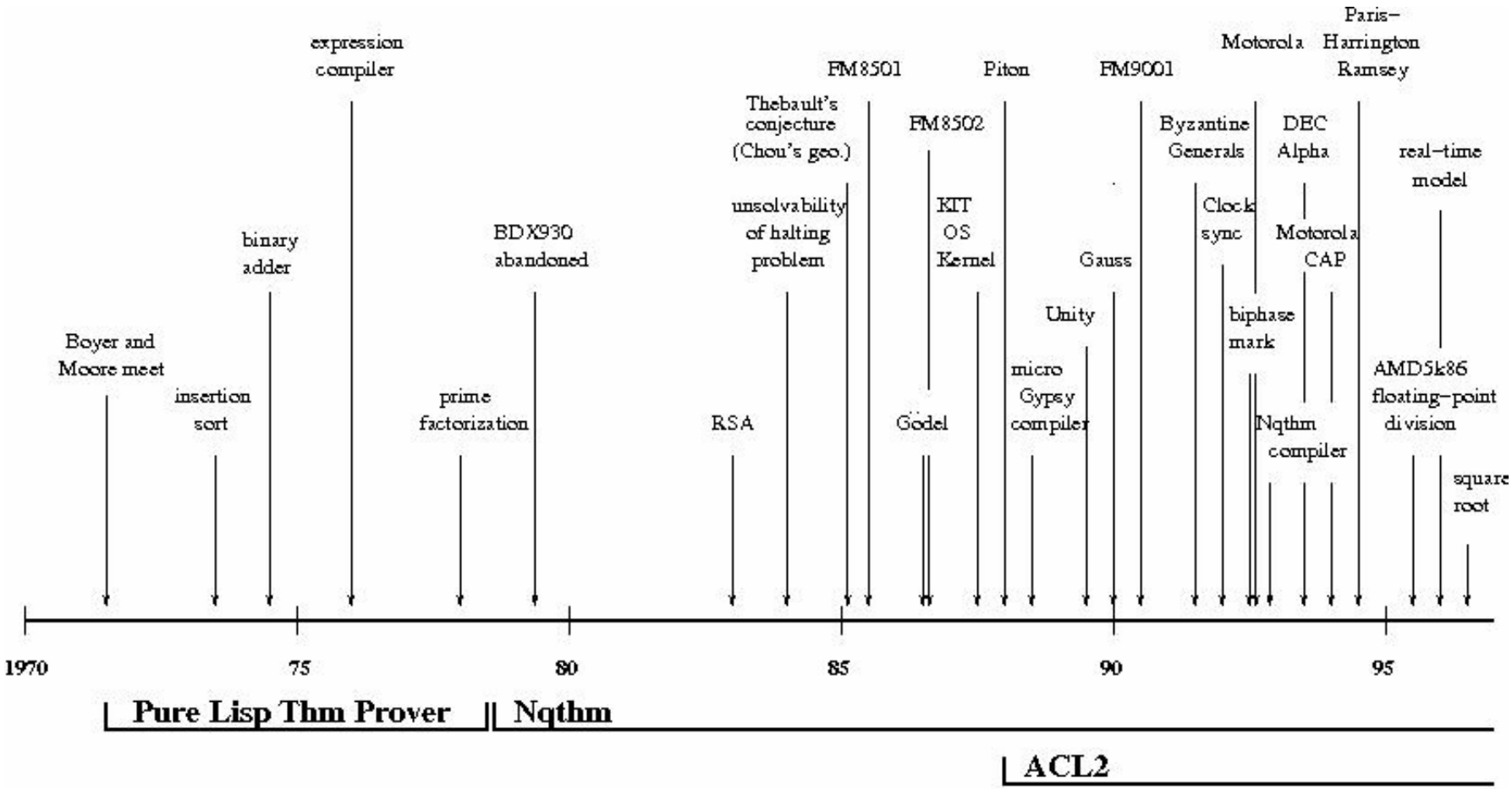
Over the next three decades

- Many large theorem proving systems are born
  - Otter (1960)
  - Boyer-Moore (1971)
  - NuPrl (1985)
  - Isabelle (1989)
  - Coq (1989)
  - PVS (1992)
  - Simplify (1990s)
  - SPASS (1994)
- The list of theorems proven automatically/semi-automatically grows

On the math side

- 1976: Appel and Haken prove the four color theorem using a program that performs a gigantic case analysis (billions of cases)
- First use of a program (essentially a simple “theorem prover”) to solve an open problem in math
  - The proof was controversial and attracted a lot of criticism
- Other open problems have since been solved using theorem provers (Otter)

On the verification side



And this is just one theorem prover!

And yet…

- In 1979, DeMillo, Lipton and Perlis, in a now famous paper, argue that software verification is doomed
- Why?
  - Too hard to verify by hand
  - Too hard to verify automatically
  - Nobody will check your verifications anyway
  - As opposed to math, where a proof becomes a proofs only after it has been validated by the community

And then… the internet happens

- Amount of code exposed to malicious attacks skyrockets
  - Vulnerabilities are widely exploited
  - And are worthy of the NYTimes front page
- At the same time, state of the art improves
- Result: technological readiness + increased cost of bugs leads to a renewed interest in software verification, and the use of analysis techniques and/or theorem provers to verify software
- Can identify subtle bugs early

Recent uses of theorem provers

- ESC/Java: pre- and post-condition checker for Java
  - ESC/Java is a tool to check that the pre-condition of a method implies the post-condition of the method. Underneath the covers, ESC/Java uses a fully automated theorem prover
- SLAM: verifying C software
  - SLAM verifies that C programs adhere to API usage rules, such as “a lock can be released only if it was previously acquired”, or “a file can be written to only if it was previously opened”. SLAM uses a theorem prover to perform predicate abstraction.

Recent uses of theorem provers

- Verisoft: end-to-end correctness
  - This porject uses interactive theorem provers to show the correctness of the software itself, but also of all the artifacts needed to execute the software (e.g. hardware and compiler)
- Rhodium: automatically proving compilers correct
  - Rhodium is a language for writing compiler optimizations that can be proven correct automatically using a theorem prover

The Grand Verification Challenge Hoare 2003

- Develop a compiler which verifies that the program is correct
- Grand Challenges for Computing Research

Course topics

- Classical techniques for automated deduction
  - Resolution
  - Gentzen calculi
  - Paramodulation
  - …
- Efficient heuristics
  - SAT solvers
- Decision procedures
- Existing theorem provers
- Applications

Tentative Schedule

09/03	Arnon/Mooly	Intro Natural Deduction
16/3	Arnon	Natural Deduction
23/3	Mooly	SAT
30/3	Mooly	DPLL(T) & SMT
6/4	Arnon	Gentzen Calculi
27/4 4/5	Arnon	Tableaux & Resolution
18/5	Mooly	Saturation Based
25/5 8/6	Arnon & Mooly	Equality
15/6	Mooly	Decision Procedures?

Homework

- Read DeMillo, Lipton and Perlis
- Read Moore’s talk from Zurich 05
- Read Tony Hoare (Grand Verification Challenge)

Extended Static Checking

- Extended Static Checking is a static analysis technique that relies on automated theorem proving

- Goals of Extended Static Checking:
  - prevent common errors in programs
  - make it easier to write reliable programs
  - increase programmer productivity

Extended Static Checking

- Help programmer in writing reliable programs by generating warning messages pointing out potential errors at compile time

- The programmer annotates the code using design-by-contract style assertions
  - These assertions can be added to clarify the contract of a method based on the warnings generated by the ESC tool
  - Big difference between ESC and Dynamic design by contract monitoring tools:
  - ESC checks errors at compile time, before running the program, for **all** possible executions of the program!

Extended Static Checking

- Extended Static Checking approach and tools have been developed by researchers at
  - Systems Research Center (SRC) (now known as HP SRC Classic (before that known as Compaq's Systems Research Center (before that known as Digital Equipment Corporation Systems Research Center)))

- Extended Static Checking website:
  - http://research.compaq.com/SRC/esc/

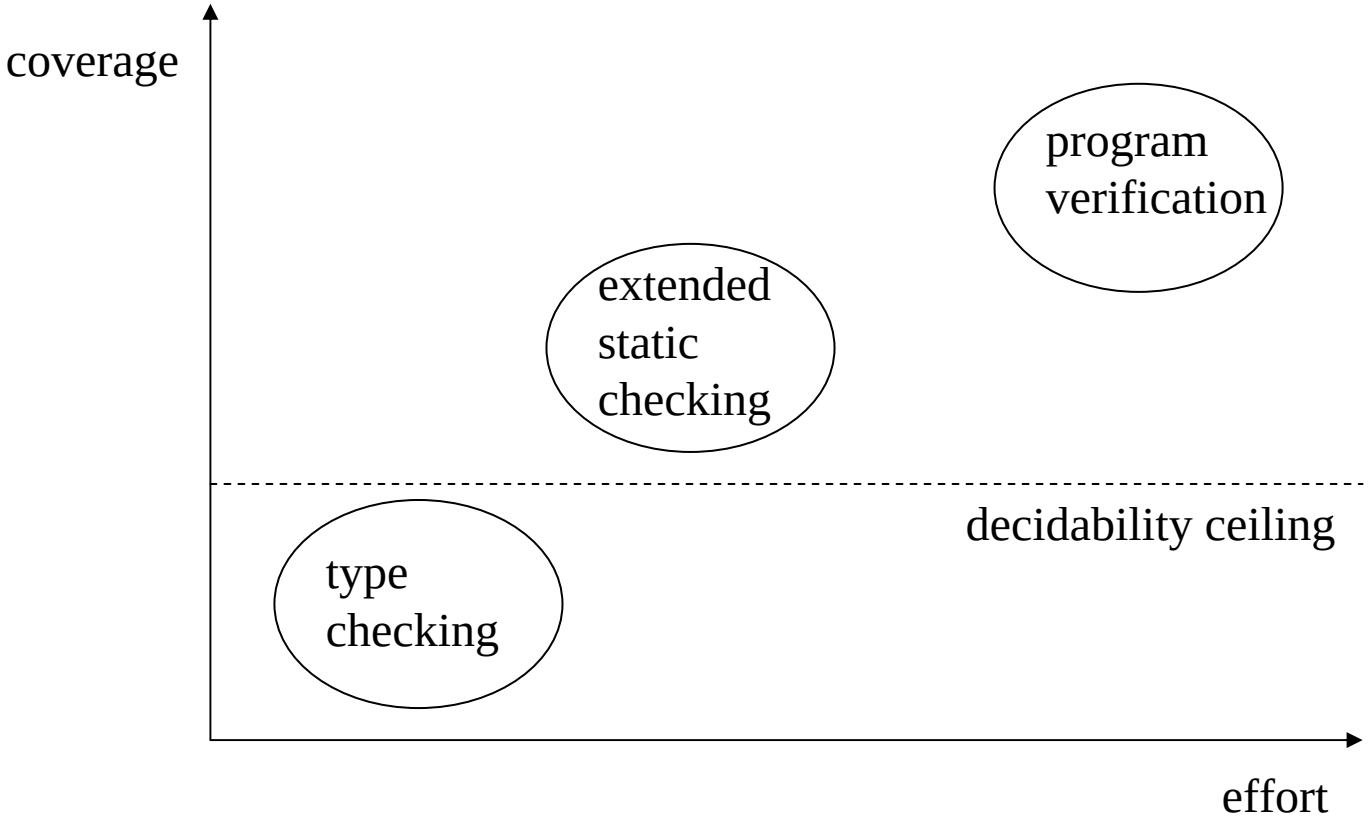
Extended Static Checking

- ESC/Java is neither sound nor complete
- When ESC/Java does not generate any warnings this does not imply that your program is free of errors
- When ESC/Java generates a warning you have to validate that there is indeed an error
- ESC/Java is a debugging or testing tool, which actually covers much more cases than you can achieve using debugging or testing

Extended Static Checking

- ESC/Java also restricts the types of properties that it checks
- Verification of general properties of programs may require a lot of input from the programmer
  - Programmer may need to enter a lot of assertions to remove the false negatives: loop invariants etc.
  - ESC/Java tries to achieve a balance between the properties it covers and the effort required by the programmer
  - In this sense it is different than the more ambitious program verification tools based on Hoare Logic and automated theorem provers

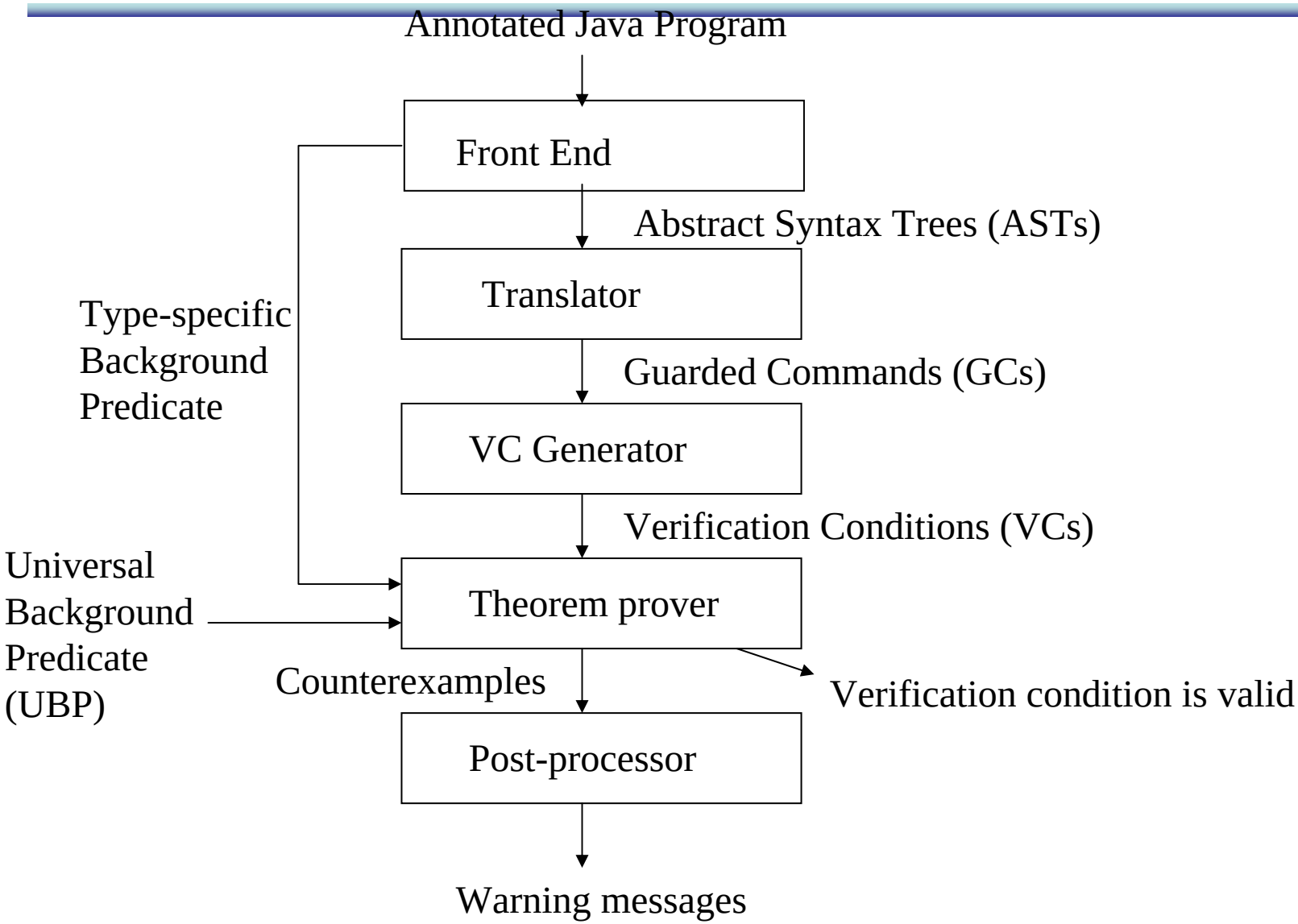
Extended Static Checking



Extended Static Checking/Ideal

- Given a program, ESC tool generates a logical formula, called a *verification condition,* that is valid when the program is free of the classes of errors under consideration
- An automated theorem prover is used to check if the negation of the verification condition is satisfiable
  - Any satisfying assignment to the negation of the verification condition is a *counterexample* behavior that demonstrates a bug

ESC/Java Architecture



Types of Errors

- ESC/Java checks three types of errors

1. Common programming errors such as null dereferences, array index bound errors, type-cast errors, division by zero, etc.

2. Common synchronization errors such as race conditions and deadlocks

3. Violations of program annotations, i.e., static checking of contracts such as preconditions, postconditions or invariants

An Example: Bag.java

```
class Bag {
  int size;
  int[] elements; // valid: elements[0..size-1]

  Bag(int[] input) {
    size = input.length;
     elements = new int[size];
     System.arraycopy(input,0,elements,0,size);
  }

  int extractMin() {
     int min = Integer.MAX_VALUE;
     int minIndex = 0;
    for (int i=1; i <=size; i++) {
        if (elements[i] < min) {
          min = elements[i];
           minIndex = i;
        }
     }
     size--;
     elements[minIndex] = elements[size];
     return min;
  }
}
```

ESC/Java

- We run ESC/Java on Bag.java

```
% escjava Bag.java
```

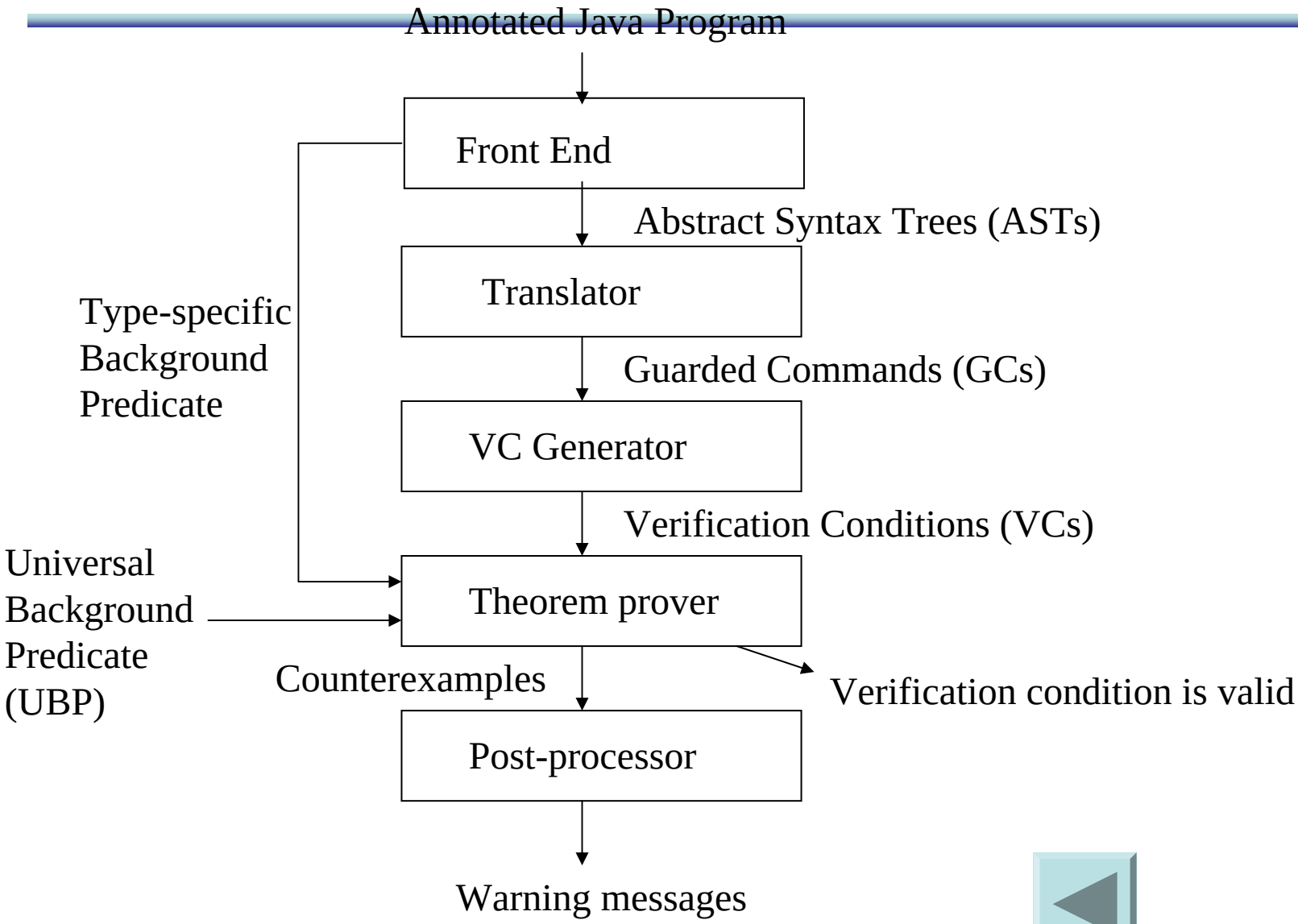
- We get the following warning messages

```
Bag.java:6: Warning: Possible null dereference (Null)
    size = input.length;
                 ^
Bag.java:15: Warning: Possible null dereference (Null)
      if (elements[i] < min) {
                   ^
Bag.java:15: Warning: Array index possibly too large (..
      if (elements[i] < min) {
                   ^
Bag.java:21: Warning: Possible null dereference (Null)
    elements[minIndex] = elements[size];
                                  ^
Bag.java:21: Warning: Array index possibly too large (..
    elements[minIndex] = elements[size];
                                  ^
```

An Example: Bag.java

```
1: class Bag {
2:   int size;
2.1: //@ invariant 0 <= size
             && size <= elements.length
3':   /*@non_null*/ int[] elements;
          // valid: elements[0..size-1]
4:
4.1: //@ requires input != null
5':   Bag(/*@non_null*/ int[] input) {
6:     size = input.length;
7:      elements = new int[size];
8:      System.arraycopy(input,0,elements,0,size);
9:   }
10:
11:   int extractMin() {
12:      int min = Integer.MAX_VALUE;
13:      int minIndex = 0;
14':     for (int i=0; i <size; i++) {
15:         if (elements[i] < min) {
16:           min = elements[i];
17:            minIndex = i;
18:         }
19:      }
19.1:   if (size > 0) {
20:         size--;
21:         elements[minIndex] = elements[size];
21.1:   }
22:      return min;
23:   }
24: }
```

How does ESC/Java Work?



Front End

- Front end for ESC/Java is just like a Java compiler but it also parses and type checks annotations as well as Java code

- The front end produces
  - Abstract Syntax Trees (ASTs) and
  - a type-specific background predicate for each class whose methods are to be checked

Front End

- The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that methods in that class use

- For example, if T is a final class (i.e., T cannot have any subclasses) then the type-specific background predicate for class T or any client of class T will include the conjunct

  $(\forall S :: S <: T \Rightarrow S = T)$

Translator

- The next stage translates each method body to be checked into a simple language based on Dijkstra's guarded commands (GCs)
- A simple guarded command syntax is

  guard $\rightarrow$ *guarded-list*

which means that: execute the guarded-list if the guard evaluates to true. For example:

$x \geq y \rightarrow max := x$

- Guarded commands allow non-determinism where typically [] is used as the non-deterministic choice operator

Translator

- ESC/Java's command language includes commands such as

  assert E

where E is a boolean expression

- Execution of a guarded command is said to *go wrong* if control reaches a subcommand of the form assert E when E is false

Translator

- Ideally the translator should translate the body of a method M into a guarded command G such that
  - G has at least one potential execution that starts in a state satisfying the background predicate of M's class and goes wrong

  if and only if
  - there is at least one way that M can be invoked from a state satisfying its preconditions and then behave erroneously by, for example, dereferencing null or terminating in a state that violates its specified post-conditions
- Unfortunately for ESC/Java neither "if" nor the "only if" part holds
  - The fact that "if" part does not hold means that the tools is unsound
  - The fact that the "only if" part does not hold means that the tool is incomplete

Modular Verification

- ESC/Java uses a modular verification strategy
- ESC/Java uses Design by contract style specifications to achieve modularity
- When ESC/Java produces the guarded command for a method M it translates each method call in M according to the specification (of its contract) rather than the implementation of the called method
- Hence, the resulting non-deterministic guarded command G may be able to go wrong in ways involving behaviors of called routines that are permitted by their specification but can never occur with the actual implementations
- Note that this is a sign of incompleteness, i.e., ESC/Java can generate false negatives

Modular Verification

- Modular verification modularizes and hopefully simplifies the verification task
  - specifications are hopefully simpler than the implementations
- Another nice side effect of the modular verification is that the methods are verified against the specifications of the methods that they are using
  - In the future if the implementation of a method that is called by method M changes but if its specification remains the same we do not have to verify M again since the verification is based on the specification not the implementation of the called methods

Overflows

- ESC/Java does not model arithmetic overflows
  - Allowing addition of two integers generate a negative value (i.e. considering the possibility of overflow) generates a lot of spurious warnings, hence they do not consider the possibility of an overflow

- Note that this is s sign of unsoundness
  - There are programs which may have overflow and cause null dereference because of that for which ESC/Java will not report any error

Loops

- A precise semantics of loops can be defined as the least fixpoints of weakest-preconditions (predicate-transformers)
- Unfortunately least fixpoints characterizing the semantics of loops are
  - uncomputable
  - and they are hard to compute even for restricted cases
- ESC/Java approximates the semantics of loops by unrolling them a fixed number of times and replacing the remaining iterations by code that terminates without ever producing an error

Loops

- The user can control the amount of loop unrolling
- Or the user can substitute a sound alternative translation for loops that relies on on the user to supply explicit loop invariants
- The default in ESC/Java is to unroll the loop body once (which evaluates the loop guard twice)
  - In a case study unrolling loops twice increased the execution time 20% but generated only one new interesting warning
  - In the same case study five unrollings doubled the time but produced no new warnings
- In their experience ESC/Java developers said that even experts have difficulty in providing correct and sufficiently strong loop invariants

Verification Condition Generator

- Verification Condition (VC) Generator generates verification conditions from the guarded commands
- The output of the VC generator for a guarded command G is a predicate in first-order logic that holds for precisely those programs states from which no execution of the command G can go wrong
- VC generator in ESC/Java is an efficient weakest-precondition generator

Reminder: Weakest Preconditions

- Dijkstra added a tool to Hoare's formalism for reasoning about programs called weakest precondition.
  - It is another useful tool in reasoning about programs
- Given an assertion P and a program segment S weakest precondition of S with respect to P written wp(S,P) is defined as:
  - the weakest initial condition such that if S starts executing in a state which satisfies that condition, when it terminates it is guaranteed that P will hold.
- Note that the Hoare triple {P}S{Q} is correct if and only if $P \Rightarrow wp(S,Q)$
  - this is why it is called the weakest precondition, every other assertion P where we can show {P}S{Q} implies (i.e., is stronger than) wp(S,Q)

Weakest Preconditions

- Dijkstra calls wp(S,Q) a predicate transformer
  - wp(S,Q) takes a predicate (assertion, same thing) Q and a program segment S, and transforms it to another predicate
  - For example, for simple assignments x:=exp the predicate transformer wp is defined as:
  - $wp(x:=exp,Q) = Q[x \leftarrow exp]$
    - where exp is a simple expression (no procedure calls in exp)

Some rules about weakest preconditions

- Some rules about weakest preconditions

- If $P \Rightarrow Q$ then $wp(S,P) \Rightarrow wp(S,Q)$
- $wp(S, P \wedge Q) \equiv wp(S,P) \wedge wp(S,Q)$
- $wp(S, P \vee Q) \equiv wp(S,P) \vee wp(S,Q)$
- $wp(S_1 ; S_2 , P) \equiv wp(S_1, wp(S_2, P))$
- $wp(\text{if } B \text{ then } S_1 \text{ else } S_2 , P) \equiv$
  $(B \Rightarrow wp(S_1, P)) \wedge (\neg B \Rightarrow wp(S_2, P))$

Examples

- $wp(x:=x+1, x\geq 1)$

$\equiv \quad x\geq 1[x\leftarrow x+1]$

$\equiv \; x+1\geq 1$

$\equiv \; x\geq 0$

- $wp(x:=x+1;x:=x+2,x<10)$

$\equiv \quad wp(x:=x+1, wp(x:=x+2, x<10))$

$\equiv \; wp(x:=x+1, x<10[x\leftarrow x+2])$

$\equiv \; wp(x:=x+1, x+2<10)$

Examples

- wp(if (x >y) max:=x else max:=y, $max \geq x \wedge max \geq y$)

$\equiv$ $(x>y \Rightarrow wp(max:=x, max \geq x \wedge max \geq y)) \wedge (\neg(x>y) \Rightarrow wp(max:=y, max \geq x \wedge max \geq y))$

$\equiv$ $(x>y \Rightarrow max \geq x \wedge max \geq y[max \leftarrow x]) \wedge (x \leq y \Rightarrow max \geq x \wedge max \geq y[max \leftarrow y])$

$\equiv$ $(x>y \Rightarrow x \geq x \wedge x \geq y) \wedge (x \leq y \Rightarrow y \geq x \wedge y \geq y)$

$\equiv$ $(x>y \Rightarrow x \geq y) \wedge (x \leq y \Rightarrow y \geq x)$

$\equiv$ true

Loops

- Loops are more complicated
- We want to compute wp(while B do S, P)
- We will need the following definitions:
  - Let $H_0(P) \equiv \neg B \wedge P$
  - Let $H_k(P) \equiv$ wp(if B then S, H_{k-1}) $\vee H_{k-1}(P)$
- The weakest precondition of the loop is the *fixpoint* of this iteration, i.e., if there is a k, such that $H_n(P) \equiv H_{n-1}(P)$ then
  - wp(while B do S, P) $\equiv H_n(P)$
- Actually the weakest precondition is the disjunction of all the iterations:
- wp(while B do S, P) $\equiv H_0(P) \vee H_1(P) \vee H_2(P)$ ...
- wp(while B do S, P) $\equiv \exists m, m \geq 0, H_m(P)$

Loops: Example

- wp(while (i<=10) do i:=i+1, i=11)

$H_0(i=11) \equiv i>10 \wedge i=11 \equiv i=11$

$H_1(i=11) \equiv wp(if(i<=10) \text{ then } i:=i+1, i=11) \vee i=11$

$\equiv i=10 \vee i=11$

$H_2(i=11) \equiv i=9 \vee i=10 \vee i=11$

$H_k(i=11) \equiv \bigvee_{0\leq j \leq k} i = 11- j$

$wp(\text{while } (i<=10) \text{ do } i:=i+1, i=11) \equiv \exists m, m\geq 0, H_m(P)$

$\equiv \exists m, m\geq 0, \bigvee_{0\leq j \leq m} i = 11- j \equiv i \leq 11$

VC Generator

- A big part of VC Generator is dealing with Java semantics
- Consider the following program fragment

```
class T extends S { ... }

t = (T) s;
```

where t is a variable of type T and s is a variable of type S.

- The Java type-cast expression (T)s returns the value of s after checking that this value is assignable to type T

VC Generator

- The background predicate includes
  - a relation $<:_1$ which models the direct subtype relation
    - for example $T <:_1 S$

- The background predicate defines the subtype relation <: which is the reflexive transitive closure of $<:_1$
- The background predicate also include a predicate is, where is(o,U) means that the value o is assignable to type U, defined as:
  - $(\forall\ o, U :: is(o, U) \equiv o = null \vee typeof(o) <: U)$

  where typeof maps non-null objects to their types

VC Generator

- Then given the code

```
class T extends S { ... }

t = (T) s;
```

the generated guarded command will be:

```
assert is(s, T); t =s
```

hence, the command explicitly checks if the value of s is assignable to type T

- When the weakest condition is generated it looks like

Theorem Prover

- After generating the verification condition ESC/Java uses the automated theorem prover Simplify to check the validity of the following formula:

  $UBP \wedge BP_T \Rightarrow VC_M$

  where

  UBP is the universal background predicate

  BP_T is the type-specific background predicate for the class T in which method M is defined

  VC_M is the generated verification condition for M

Post Processor

- Produces warnings when the prover is unable to prove the verification conditions
- Simplify theorem prover has some features which help the postprocessor to provide some warnings rather then just printing a message that the method is not correct
- When Simplify cannot prove the validity of a formula, it finds and reports one or more counterexample contexts

Post Processor

- A counterexample context is a conjunction of conditions
  - that collectively imply the negation of the verification condition
  - and have not been shown by the prover to be mutually inconsistent
- These conditions are mapped to the program source code and translated to warnings by the postprocessor
- There maybe more than one reason for the verification of a method to fail
- ESC/Java reports multiple (different) warning messages based on the counterexample contexts
- There is a limit (10) on the number of warnings it produces
- Simplify can produce spurious counterexamples which result in spurious warnings