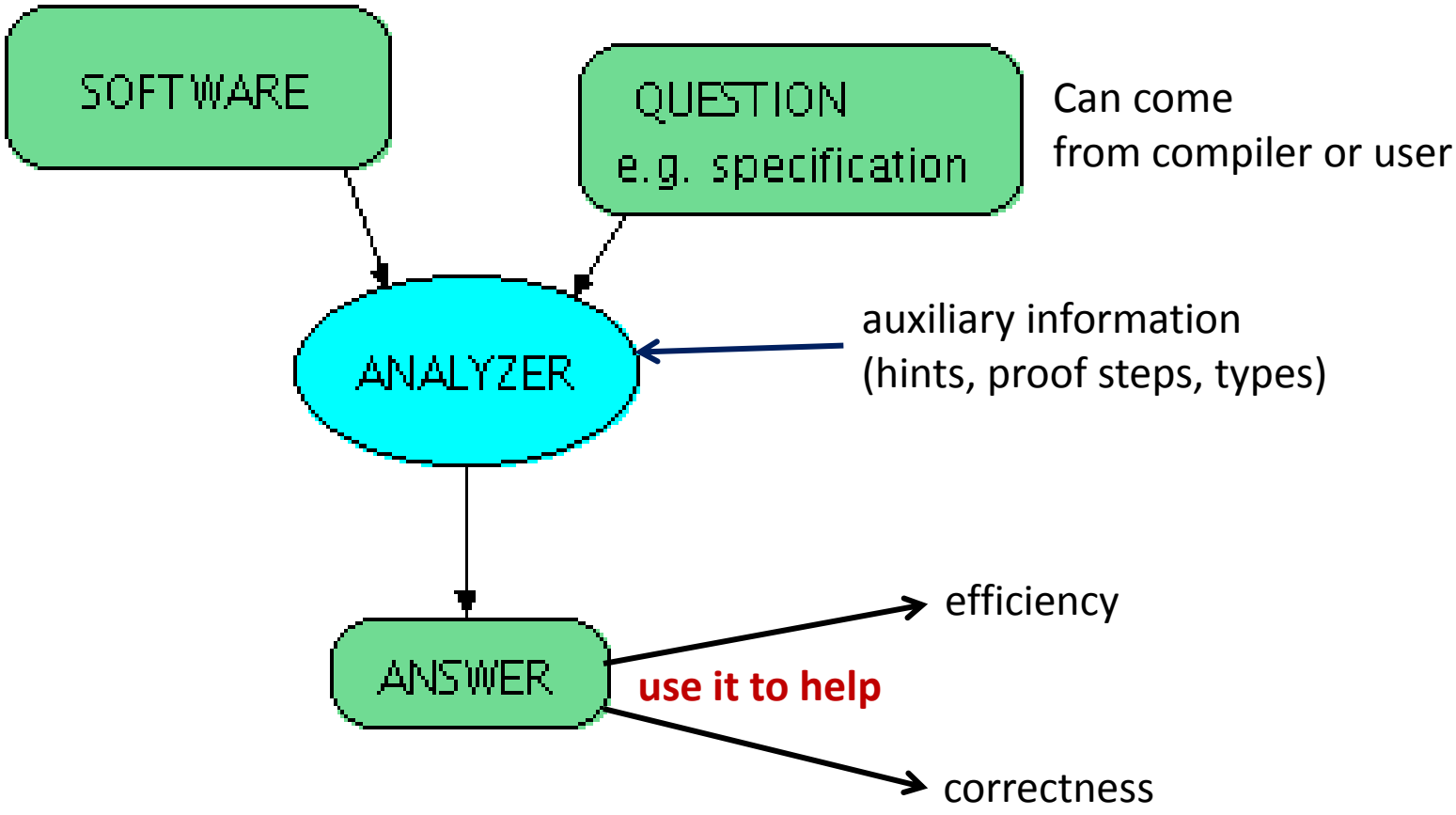
Program Analysis

Goal:

Automatically computes potentially useful information about the program.



Uses of Program Analysis

Compute information about the program and use it for:

- efficiency (codegen): Program transformation
  - Use the information in compiler to **transform the program**, make it more efficient (“optimization”)
- correctness: Program verification
  - Provide **feedback to developer** about possible errors in the program

Example Transformations

- Common sub-expression elimination using available expression analysis
  - avoid re-computing (automatically or manually generated) identical expressions
- Constant propagation
  - use constants instead of variables if variable value known
- Copy propagation
  - use another variable with the same name
- Dead code elimination
  - remove unnecessary code
- Automatically generate code for parallel machines

Examples of Verification Questions

Example questions in analysis and verification (with sample links to tools or papers):

- Will the program crash?
- Does it compute the correct result?
- Does it leak private information?
- How long does it take to run?
- How much power does it consume?
- Will it turn off automated cruise control?

Space Missions

Arithmetic Overflow

According to a presentation by Jean-Jacques Levy (who was part of the team who searched for the source of the problem), the source code in Ada that caused the problem was as follows:

```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
  P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
  P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
  P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_BV_32));
end if;
P_M_DERIVE(T_ALG.E_BH) :=
  UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH)*G_M_INFO_DERIVE(T_ALG.E_BH)));
```

http://en.wikipedia.org/wiki/Ariane_5_Flight_501

August 2005

Gerardo Dominguez/zrh.airlinerpictures.net

As a Malaysia Airlines jetliner cruised from Perth, Australia, to Kuala Lumpur, Malaysia, one evening last August, it suddenly took on a mind of its own and zoomed 3,000 feet upward. The captain disconnected the autopilot and pointed the Boeing 777's nose down to avoid stalling, but was jerked into a steep dive. He throttled back sharply on both engines, trying to slow the plane. Instead, the jet raced into another climb. The crew eventually regained control and manually flew their 177 passengers safely back to Australia.

Investigators quickly discovered the reason for the plane's roller-coaster ride 38,000 feet above the Indian Ocean. A defective software program had provided incorrect data about the aircraft's speed and acceleration, confusing flight computers.

ASTREE Analyzer

“In Nov. 2003, ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1h20 on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon™ 64 using 580 Mb of memory).”

- http://www.astree.ens.fr/

AbsInt

- 7 April 2005. AbsInt contributes to guaranteeing the safety of the A380, the world's largest passenger aircraft. The Analyzer is able to verify the proper response time of the control software of all components by computing the worst-case execution time (WCET) of all tasks in the flight control software. This analysis is performed on the ground as a critical part of the safety certification of the aircraft.

Coverity Prevent

- SAN FRANCISCO - January 8, 2008 - Coverity®, Inc., the leader in improving software quality and security, today announced that as a result of its contract with US Department of Homeland Security (DHS), **potential security and quality defects** in 11 popular open source software projects were **identified and fixed**. The 11 projects are **Amanda, NTP, OpenPAM, OpenVPN, Overdose, Perl, PHP, Postfix, Python, Samba, and TCL.**

Microsoft's Static Driver Verifier

Static Driver Verifier (SDV) is a thorough, compile-time, static verification tool designed for kernel-mode drivers. SDV finds serious errors that are unlikely to be encountered even in thorough testing. SDV systematically analyzes the source code of Windows drivers that are written in the C language. SDV uses a set of interface rules and a model of the operating system to determine whether the driver interacts properly with the Windows operating system. SDV can verify device drivers (function drivers, filter drivers, and bus drivers) that use the Windows Driver Model (WDM), Kernel-Mode Driver Framework (KMDF), or NDIS miniport model. SDV is designed to be used throughout the development cycle. You should run SDV as soon as the basic structure of a driver is in place, and continue to run it as you make changes to the driver. Development teams at Microsoft use SDV to improve the quality of the WDM, KMDF, and NDIS miniport drivers that ship with the operating system and the sample drivers that ship with the Windows Driver Kit (WDK).

SDV is included in the Windows Driver Kit (WDK) and supports all x86-based and x64-based build environments.

Further Reading on Verification

- Recent *Research Highlights* from the **Communications of the ACM**
  - A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World
  - Retrospective: An Axiomatic Basis for Computer Programming
  - Model Checking: Algorithmic Verification and Debugging
  - Software Model Checking Takes Off
  - Formal Verification of a Realistic Compiler
  - seL4: Formal Verification of an Operating-System Kernel

  (click on the links to see pointers to papers)

Type Inference

Example Analysis: Type Inference

- Avoid the need for some type declarations, but still know the type
- Infer types that programmer is not willing to write (e.g. more precise ones)
- We show a simple example: inferring types that can be simple values or functions
  - we assume no subtyping in this part
  - corresponds to *Simply Typed Lambda Calculus*

Subset of Scala

- Int, Boolean (unless otherwise specified)
  - These are two disjoint types
- arithmetic operations (+, -, …), Int x Int => Int
- relations relate Int and give Boolean
- boolean operators
- functions
  - also anonymous functions x=>E
- if-then-else statements

Example

```
object Main {
  val a = 2 * 3
  val b = a < 2
  val c = sumOfSquares(a)
  val d = if(b) c(3) else square(a)
}

def square(x) = x * x

def sumOfSquares(x) = {
  (y) => square(x) + square(y)
}
```

Can it type-check?

Do there exist some type declarations for which it type checks

Find assignment
{TA -> Int, TB -> Boolean …}

```
object Main {
  val a: TA = 2 * 3
  val b: TB = a < 2
  val c: TC = sumOfSquares(a)
  val d: TD = if(b) c(3) else square(a)
}

def square(x: TE): TF = x * x

def sumOfSquares(x: TG): TH = {
  (y: TI) => square(x) + square(y)
}
```

Type constraints in example

```
object Main {
  val a: TA = 2 * 3
  val b: TB = a < 2
  val c: TC = sumOfSquares(a: TA)
  val d: TD =
   if(b) c(3): S1 else square(a): S2
}

def square(x: TE): TF = x * x

def sumOfSquares(x: TG): TH = {
  (y: TI) => (square(x) + square(y)): S3
}
```

2: Int, 3: Int

TA = Int

TB = Boolean

TC = TH

TA = TG

S1 = S2

TD = S2

TD = S1

TA = TE

TF = Int

TE = TF

TE = TG

TI = TE

TH = TI -> S3

S3 = Int

S3 = TF

Hindley-Milner algorithm, intuitively

1. Record type constraints

```
val a: A = 3
val b: B = a
```

constraints:
{ A = Int, A = B}

2. Solve type constraints
   - obvious in the case above: {A= Int, B = Int}
   - in general use unification algorithm

3. Return assignment to type variables or failure

Recording type constraints

$$\frac{\Gamma \vdash b : T_1 \quad \Gamma \vdash e_1 : T_2 \quad \Gamma \vdash e_2 : T_3}{\Gamma \vdash (\text{if (b) } e_1 \text{ else } e_2) : T_4}$$

T1 = Boolean
T2 = T3 = T4

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1 + e_2) : T_3}$$

T1 = T2 = T3 = Int

$$\frac{\Gamma \vdash e_i : T_i \quad \Gamma \vdash f : T_0}{\Gamma \vdash f(e_1, \ldots, e_n) : T}$$

$T_0 = T_1 \times \ldots \times T_n \to T$

Rules for Solving Equations

$$\frac{T_1 = T_2}{E[\ldots T_1 \ldots] = E[\ldots T_2 \ldots]}$$

substitute equals for equals

$$\frac{T_1 \times T_2 = S_1 \times S_2}{T_1 = S_1 \;,\; T_2 = S_2}$$

$$\frac{T_1 \to T_2 = S_1 \to S_2}{T_1 = S_1 \;,\; T_2 = S_2}$$

$$\frac{f(t_1, t_2) = f(s_1, s_2)}{t_1 = s_1 \;,\; t_2 = s_2}$$

$T_1 \times T_2 \times T_3$

$T_1 \times (T_2 \times T_3)$

Unification

Finds a solution (substitution) to a set of equational constraints.

- works for any constraint set of equalities between (type) constructors
- finds the most general solution

Definition

A set of equations is in *solved form* if it is of the form

$\{x1 = t1, \ldots x_n = t_n\}$ iff variables x_i do not appear in terms t_i, that is

$\{x1, \ldots, x_n\} \cap (FV(t1) \cup \ldots \cup FV(t_n)) = \emptyset$

$FV(TA \to TB) = \{TA, TB\}$

$FV(TA \to Int) = \{TA\}$

In what follows,

- x denotes a type variable (like TA, TB before)
- t, t_i, s_i denote terms, that may contain type variables

Unification Algorithm

We obtain a solved form in finite time using the non-deterministic algorithm that applies the following rules as long as no clash is reported and as long as the equations are not in solved form.

Orient: Select `t = x, t ≠ x` and replace it with `x = t`.

Delete: Select `x = x`, remove it.

Eliminate: Select `x = t` where `x` does not occur in `t`, put it aside, substitute `x` with `t` in all remaining equations

Occurs Check: Select `x = t`, where `x` occurs in `t`, report clash.

Decomposition: Select $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$, replace with $t_1 = s_1, \ldots, t_n = s_n$.

e.g. $(T_1 \times T_2) = (S_1 \times S_2)$ becomes $T_1 = S_1$, $T_2 = S_2$

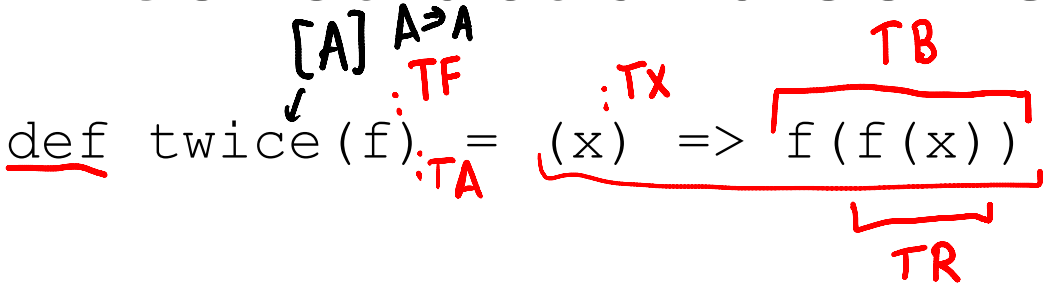
Decomposition Clash: $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_n)$, $f \neq g$, report clash.

e.g. $(T_1 \times T_2) = (S_1 \to S_2)$ is $f(T_1, T_2) = g(S_1, S_2)$ so it is a clash

f and g can denote x, ->, as well as constructor of polymorphic containers:

`Map[A, B] = Map[C, D]` will be replaced by `A = C` and `B = D`

Example 2
Construct and Solve Constraints



TA = TB → TB

TA = TX → TB

TF = TB → TB

TF = TX → TR

TF = TR → TB

TX → TR = TR → TB

TX = TR

TX = TB

TR = TB

twice : TT = TF → TA

TT = TF → (TX → TB)

TT = (TX → TR) → (TX → TB)

TT = (TB → TB) → (TB → TB)

Example 2, cleaned up

def twice(f) = (x) => f(f(x))

add type variables:

def twice(f:TF):TA = (x:TX) => f(f(x):TR):TB

constraints:

TA=TX->TB, TF=TX->TR, TF=TR->TB

consequences derived:

TX=TR, TR=TB

replace TR,TB with TX:

TR=TX, TB=TX, TA=TX->TX, TF=TX->TX

twice: TT = TF->TA = (TX->TX)->(TX->TX)

Most General Solution

What is the general solution for

```
def f(x) = x
def g(a) = f( f( a))
```

[TX] :TX

[TX] :TA

TC

TB

f : TF TF = TX → TX

g : TG TG = TA → TC

TX → TX = TA → TB

TX → TX = TB → TC

Example solution: `a:Int, f,g : Int -> Int`

Are there others? How do all solutions look like?

TF = TX → TX

TG = TX → TX

TA = TX

TB = TX

TC = TX

Instantiating Type Variables

```
def f(x) = x
def test() = if (f(true)) f(3) else f(4)
```

Generate and solve constraints.

Is result different if we clone f for each invocation?

Generalization Rule

- If after inferring top-level function definitions certain variables remain unconstrained, then generalize these variables
- When applying a function with generalized variables, rename variables into fresh ones

```
def f(x) = x
def test() = if (f(true)) f(3) else f(4)
```

Individual exercise 1:

```
def length(s : String) : Int = {...}
def foo(s: String) = length(s)
def bar(x, y) = foo(x) + y
```

Individual exercise 2:

```
def CONS[T](x:T, lst:List[T]):List[T]={...}
def listInt() : List[Int] = {...}
def listBool() : List[Bool] = {...}

def baz(a, b) = CONS(a(b), b)
def test(f,g) =
      (baz(f,listInt), baz(g,listBool))
```