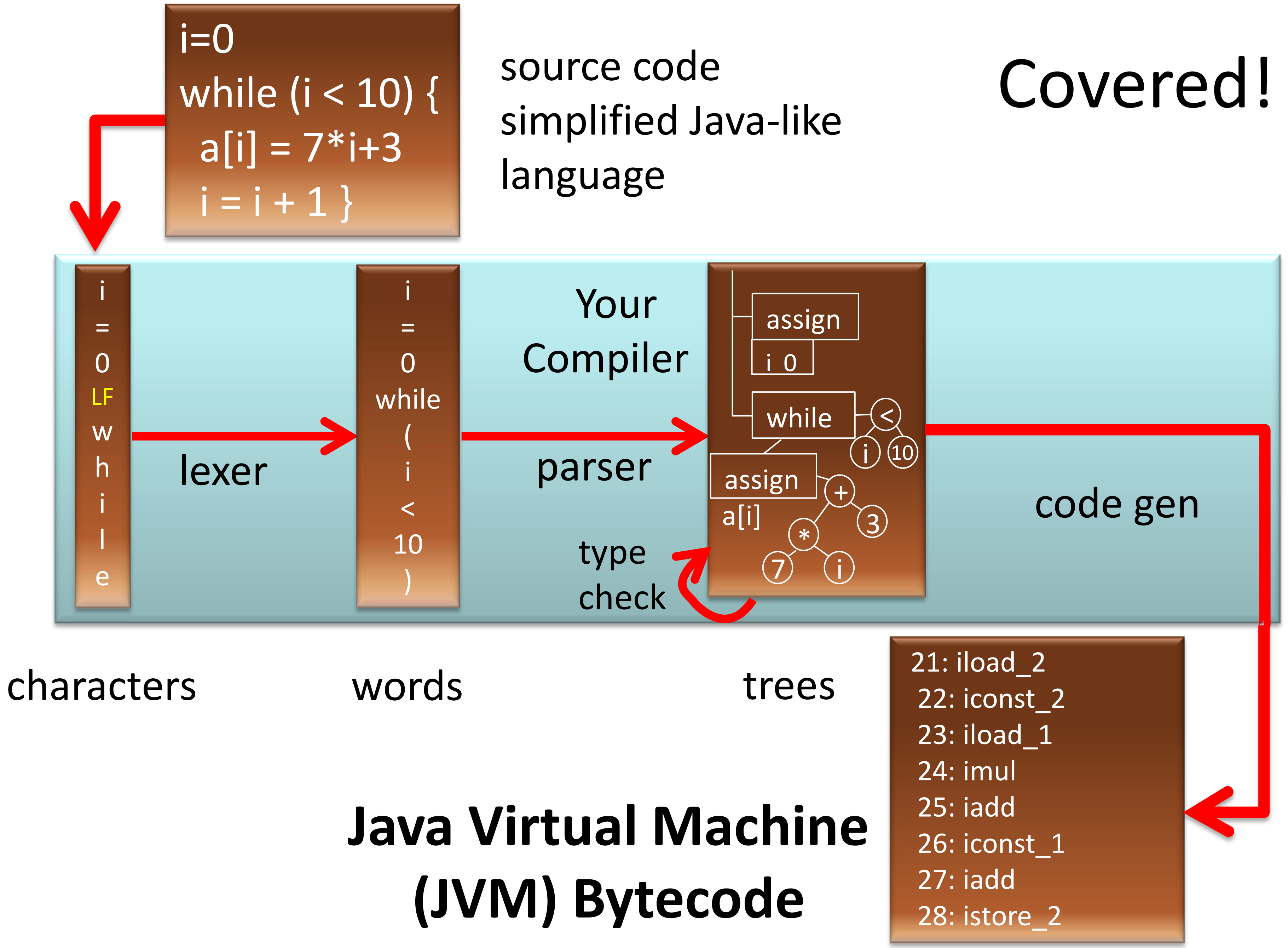
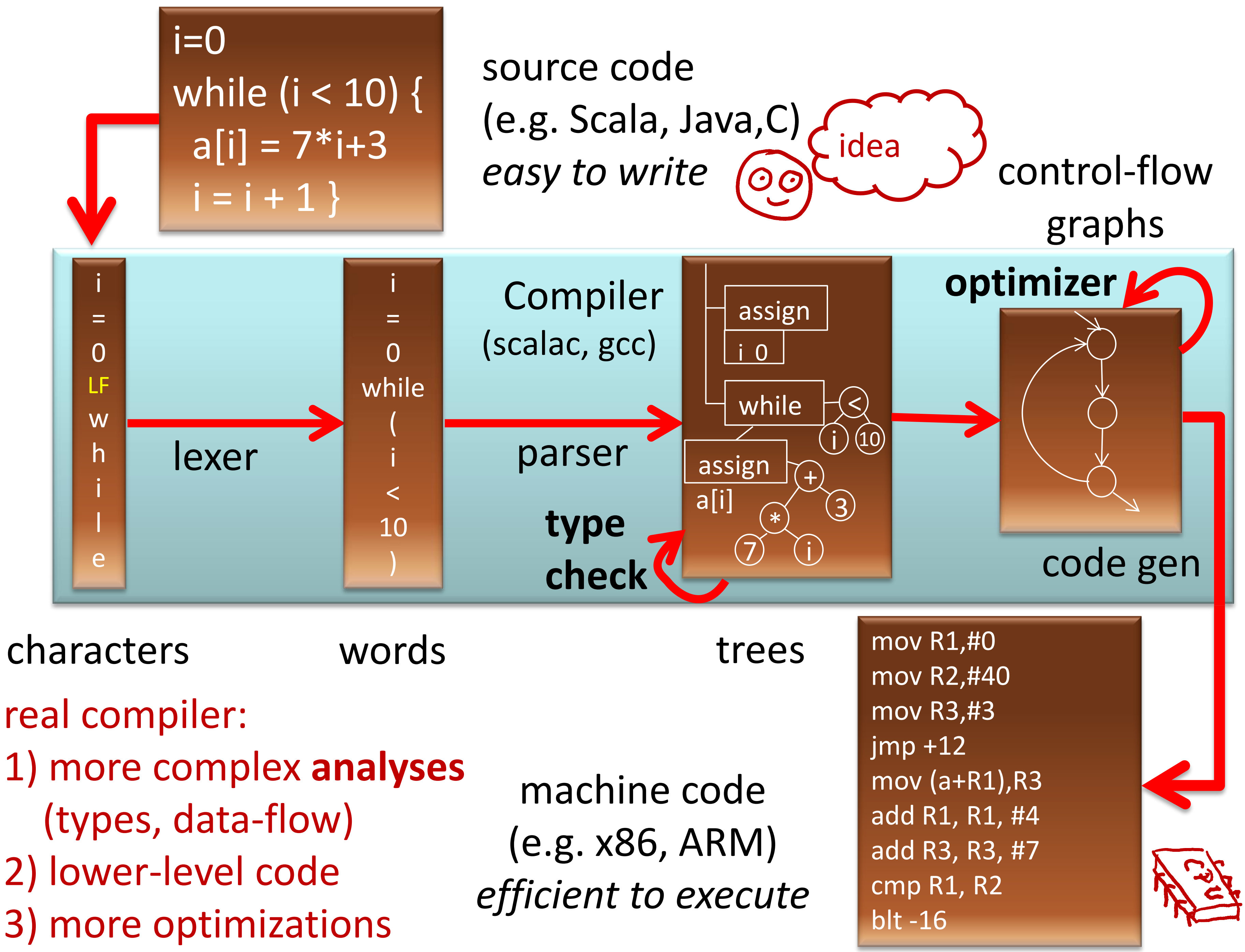
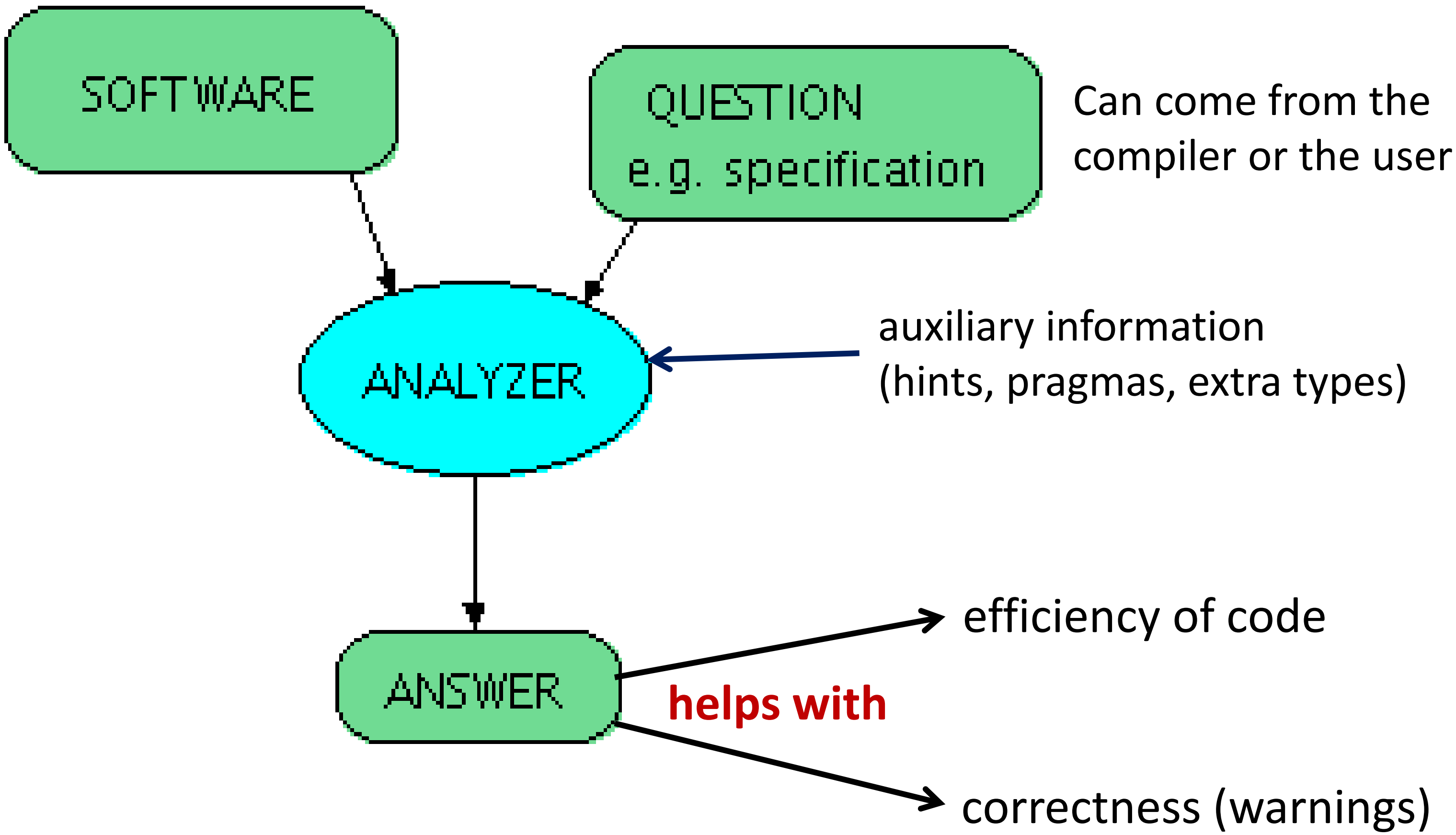


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
simplified Java-like
language
Covered!
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Your
Compiler
parser
type
check
assign
i 0
while
assign
a[i]
code gen
characters
words
trees
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd
26: iconst_1
27: iadd
28: istore_2
Java Virtual Machine
(JVM) Bytecode


i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
idea
control-flow
graphs
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Compiler
(scalac, gcc)
parser
type
check
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
optimizer
code gen
characters
words
trees
real compiler:
1) more complex analyses
(types, data-flow)
2) lower-level code
3) more optimizations
machine code
(e.g. x86, ARM)
efficient to execute
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16

Program Analysis

Goal:
Automatically computes potentially useful information about the program.



Uses of Program Analysis

Compute information about the program; use it for:

- efficiency (codegen): Program **transformation**
  - Use the information in compiler to **transform the program**, make it more efficient (“optimization”)

    bipush 9; bipush 7; imul →
    bipush 63

- correctness: Program **verification**
  - Provide **feedback to developer** about possible errors in the program

    a[k] = v

    warning: out of-bounds reference possible for k=100

Example **Transformations**

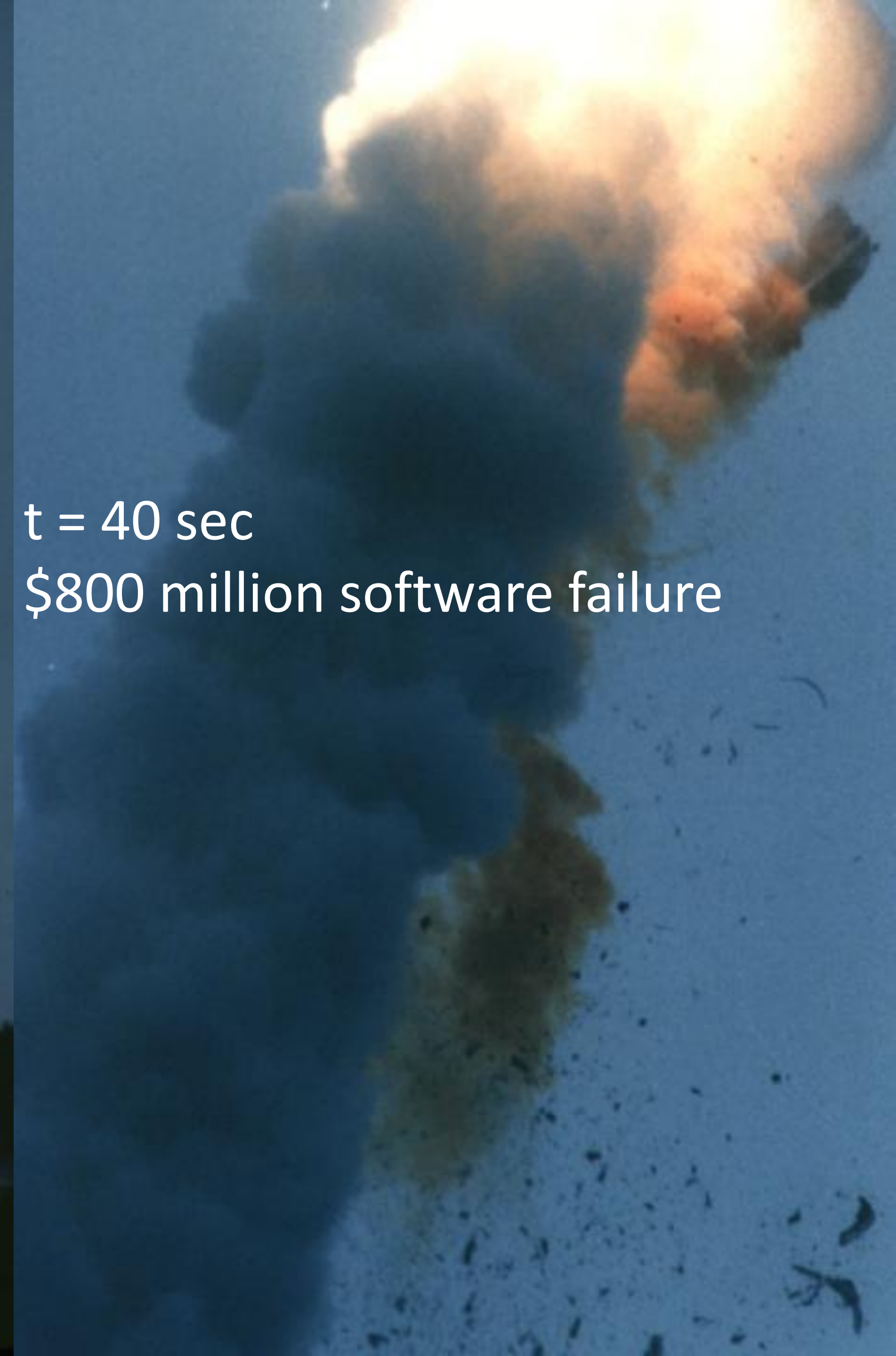
- Common sub-expression elimination using available expression analysis
  - avoid re-computing (automatically or manually generated) identical expressions:
    a[c[k]] = a[c[k]] + a[k] →
    { **val** x1 = c[k]; a[x1] = a[x1] + a[k] }
- Constant propagation
  - use constants instead of variables if variable value known
- Copy propagation
  - use another variable with the same name
- Dead code elimination
  - remove code that is never reached
- Automatically generate good code for parallel machines

Examples of **Verification** Questions

Example questions in analysis and verification

- Will the program crash?
  - null dereference, array bounds, exception
- Does it compute the correct result?
  - satisfy given assertions, numerical value close enough
- Does it leak private information?
  - sends passwords over the network?
- How long does it take to run?
  - will airplane controller react fast enough
- How much power does it consume?
  - which version of code consumes less power?


French Guyana, June 4, 1996
t = 0 sec
Space Missions


t = 40 sec
$800 million software failure

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.

Air Transport

ASTREE Analyzer

"In Nov. 2003, ASTRÉE [analyzer] was able to **prove completely automatically the absence of any run-time errors** 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)

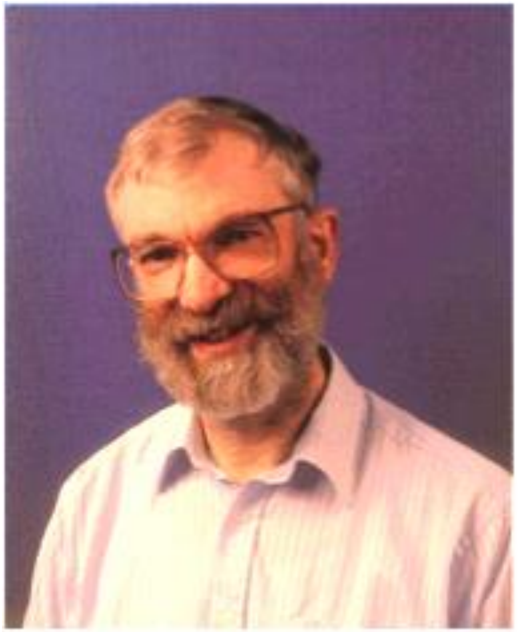
Type Inference

Example Analysis: Type Inference

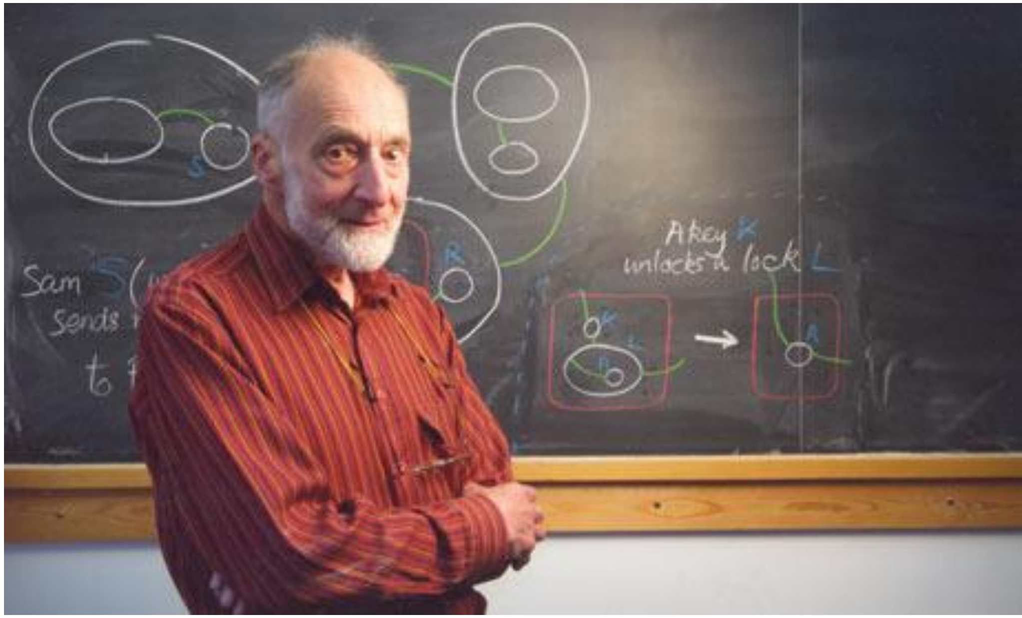
- Reduce the need to write type declarations, yet detect type errors statically
  - best of static and dynamic typing
- Infer types that programmer is not willing to write (e.g. more precise types)
- Today: a simple example: inferring types that can be: *simple values*, *pairs*, or *functions*
  - we assume *no subtyping* in this part

Hindley-Milner Type Inference

J. **Roger Hindley** (1938-)



Arthur John **Robin** Gorell **Milner**
13 January 1934 – 20 March 2010



http://www.users.waitrose.com/~hindley/

A Small Language

- Int, Bool (could be any finite set of base types)
  - Disjoint – no overlap between values
- functions on primitive types given by declarations
  - +, - : Int x Int -> Int   <,> : +, - : Int x Int -> Boolean
  - &&, || : Boolean x Boolean -> Boolean
- Pairs: (7,9) : (Int,Int)   Pair[A,B]
  - records - same: { f = 7, g = false } : { f : Int, g : Boolean }
- Lists: List(1,2,3) : List[Int],   List(true,false) : List[Bool]
- User-defined functions   Function[A,B]
  - including anonymous functions: (x=> x*x + 1): (Int => Int)
- **val**-s and blocks similar to Scala: { **val** x:T= x0 ; body}

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

named function without parameter type declaration

anonymous function without argument type declaration

Can we assign types so it type checks?

```
object Main {
  val a = 2 * 3        a:Int
  val b = a < 2        b:Bool          Int
  val c = sumOfSquares(a)          c : Int => Int
  val d = if(b) c(3) else square(a)          d : Int
}                Int              Int
                z: Int
def square(z) =(z * z):Int
                                         x: Int
def sumOfSquares(x) = {
                                            y:Int
  (y) => square(x) + square(y)
}  y:Int              Int
                 Int              Int
                          Int
                    Int => Int
```

solution not always unique

X => X

Introduce type variables for unknown types

```
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)
}
```

Write relationships (constraints) between variables – here a subset written

```
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
}
```

TA = Int
TB = Bool

TC = TH
TA = TG

TD = S1
TD = S2
TA = TE
S2 = TF
TC = (Int => S1)

* : Int x Int => Int
TF = Int
TE = Int

TE = TG
TE = TI
TF = Int
S3 = Int
TH = (TI => S3)

λ-calculus

Generated type constraints: no program expressions, only types

TA = Int
TB = Bool
TC = TH
TA = TG
TF = Int
TE = Int

TD = S1
TD = S2
TA = TE
S2 = TF
TC = (Int => S1)
TE = TG
TE = TI
TF = Int
S3 = Int
TH = (TI => S3)

If left side is a variable, replace left side by right everywhere

Solving (Equality) Constraints

TA = Int
TB = Bool
TC = TH
TA = TG
TD = S1
TD = S2
TA = TE
S2 = TF
TC = (Int => S1)

If left side is a variable, replace left side by right everywhere

TF = Int
TE = Int
TE = TG
TE = TI
TF = Int
S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TC = TH
Int = TG

TD = S1
TD = S2
Int = TE
S2 = TF
TC = (Int => S1)

1) If left side is a variable, replace left side by right everywhere

2) if right side is a variable, swap left and right

TF = Int
TE = Int

TE = TG
TE = TI
TF = Int
S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

Like Gaussian elimination

TA = Int
TB = Bool

TD = S1
TC = TH
TD = S2
TG = Int
TE = Int
S2 = TF
TC = (Int => S1)

1) If left side is a variable, replace left side by right everywhere

2) if right side is a variable, swap left and right (so you can apply 1) again

TF = Int
TE = Int

TE = TG
TE = TI
TF = Int
S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TC = TH
TG = Int

TD = S1
TD = S2
Int = Int
S2 = Int
TH = (Int => S1)

1) If left side is a variable, replace left side by right everywhere

2) if right side is a variable, swap left and right (so you can apply 1) again

3) delete equations of form T=T

TF = Int
TE = Int

Int = Int
Int = TI
Int = Int
S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TD = S1

TC = TH
TD = S2

TG = Int

S2 = Int
TH = (Int => S1)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

TF = Int
TE = Int

2) if right side is a variable, swap left and right (so you can apply 1) again

TI = Int

3) delete equations of form T=T

S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TD = S1
TC = TH
S1 = S2
TG = Int

S2 = Int
TH = (Int => S1)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

TF = Int
TE = Int

2) if right side is a variable, swap left and right (so you can apply 1) again

TI = Int

3) delete equations of form T=T

S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TD = S2
TC = TH
S1 = S2
TG = Int

S2 = Int
TH = (Int => S2)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

2) if right side is a variable, swap left and right (so you can apply 1) again

3) delete equations of form T=T

TF = Int
TE = Int

TI = Int

S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int
TB = Bool

TD = Int
TC = TH
S1 = Int
TG = Int

S2 = Int
TH = (Int => Int)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

2) if right side is a variable, swap left and right (so you can apply 1) again

3) delete equations of form T=T

TF = Int
TE = Int

TI = Int

S3 = Int
TH = (TI => S3)

Solving (Equality) Constraints

TA = Int

TB = Bool

TD = Int

TC = (Int => Int)

S1 = Int

TG = Int

S2 = Int

TH = (Int => Int)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

2) if right side is a variable, swap left and right (so you can apply 1) again

3) delete equations of form T=T

4) decompose complex types:

TF = Int

TE = Int

TI = Int

S3 = Int

(Int => Int) = (TI => S3)

Decompose Types

(A => B) = (A' => B')

if and only if

A=A' and **B=B'**

Int = TI and **Int = S3**

4) decompose complex types: **(Int => Int) = (TI => S3)**

Solving (Equality) Constraints

TA = Int
TB = Bool

TD = Int

TC = (Int => Int)

S1 = Int

TG = Int

S2 = Int

TH = (Int => Int)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

2) if right side is a variable, swap left and right (so you can apply 1) again

3) delete equations of form T=T

4) decompose complex types:

TF = Int
TE = Int

TI = Int

S3 = Int

Int = TI

Int = S3

Solving (Equality) Constraints

TA = Int

TB = Bool

TD = Int

TC = (Int => Int)

S1 = Int

TG = Int

S2 = Int

TH = (Int => Int)

1) If left side is a variable, replace left side by right everywhere (RHS can be any)

TF = Int

TE = Int

2) if right side is a variable, swap left and right (so you can apply 1) again

TI = Int

3) delete equations of form T=T

S3 = Int

4) decompose complex types

Int = Int   Int = Int

Substitute back solution into the program

```
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
}
```

TA = Int

TB = Bool

TC = (Int => Int)

TG = Int

TD = Int

S1 = Int

S2 = Int

TH = (Int => Int)

TF = Int

TE = Int

TI = Int

S3 = Int

Obtained program fully annotated with types!

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

def square(x: Int): Int = x * x

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

Hindley-Milner Algorithm Sketch

1. Generate type constraints
   - introduce type variable for each sub-tree
   - applicable type rule for the tree node gives a
     constraint between type variables in the tree
2. Solve type constraints
   - systematically use rules for equality (substitution)
   - decomposition handles cases when both sides are non-variables
3. If constraints have solution, put it into tree, otherwise report a type error

From Type Rule to Constraint: *

type rule:

$$\frac{e1 : \text{Int} \qquad e2 : \text{Int}}{e1 * e2 : \text{Int}}$$

equivalent constraint form
(each subtree has a distinct type variable)

$$\frac{e1 : T1 \qquad e2 : T2}{e1 * e2 : T3} \quad \underbrace{T1{=}\text{Int},\ T2{=}\text{Int},\ T3{=}\text{Int}}_{\text{“where clause” of the type rule}}$$

From Type Rule to Constraint: **if**

type rule:

$$\frac{c : \mathrm{Bool} \quad e1 : T \quad e2 : T}{(\textbf{if}\ (\ c\ )\ e1\ \textbf{else}\ e2) : T}$$

equivalent constraint form:

$$\frac{c : T1 \quad e1 : T2 \quad e2 : T3}{(\textbf{if}\ (\ c\ )\ e1\ \textbf{else}\ e2) : T4}$$

T1=Bool, T2=T3, T4=T2

Type variables are local for each rule application. T1,T2,T3,T4 for one "**if**" expression have nothing to do with those variables for another "**if**"

General Function Application Rule

$$\frac{f : ((T_1 \text{ x } \ldots \text{ x } T_n) \Rightarrow T \qquad e_1 : T_1 \quad \ldots \quad e_n : T_n}{f(e_1,\ldots,e_n) : T}$$

equivalent constraint form:

$$\frac{f : T_f \qquad e_1 : T_1 \quad \ldots \quad e_n : T_n}{f(e_1,\ldots e_n) : T} \qquad T_f = ((T_1 \text{ x } \ldots \text{ x } T_n) \Rightarrow T)$$

Variable Rule

$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T}$$

equivalent constraint form:

$$\frac{(x,T_1) \in \Gamma}{x : T_2} \quad T_1 = T_2$$

These Rules Cover Primitives, Too

$$\frac{f : T_f \quad e_1 : T_1 \quad \ldots \quad e_n : T_n}{f(e_1, \ldots e_n) : T} \qquad T_f = ((T_1 \text{ x } \ldots \text{ x } T_n) \Rightarrow T)$$

Now assume (f, Int x Int => Int) $\in \Gamma$ (f is e.g. *,+)

$$\frac{(f, \text{Int x Int} \Rightarrow \text{Int}) \in \Gamma}{x : T_f} \qquad T_f = (\text{Int x Int} \Rightarrow \text{Int})$$

$$\frac{f : T_f \quad e_1 : T_1 \quad e_2 : T_2}{f(e_1, e_n) : T} \qquad T_f = ((T_1 \text{ x } T_2) \Rightarrow T)$$

$(\text{Int x Int} \Rightarrow \text{Int}) = ((T_1 \text{ x } T_2) \Rightarrow T)$

$((\text{Int x Int}) = (T_1 \text{ x } T_2) \qquad \text{Int} = T$

$\text{Int} = T_1$

$\text{Int} = T_2$

$\text{Int} = T$

Equality between Types

(A=>B)=(A'=>B')	iff	A=A' and B=B'
(A x B) = (A' x B')	iff	A=A' and B=B'
List[A] = List[A']	iff	A=A'
(A=>B) = (C x D)	iff	**false**
(A => B) = List[C]	iff	**false**

Type constructor: constructs types from types

Unary: List[A] – one type argument

A => B (Function[A,B]), AxB (Pair[A,B]) - two type args

$f(t_1,\ldots,t_n)$ – type constructor applied to types

$f(t_1,\ldots,t_n) = f(t'_1,\ldots t'_n)$ iff $t_1=t'_1$ and … and $t_n = t'_n$

$f(t_1,\ldots,t_n) = g(t'_1,\ldots t'_m)$ iff false (f != g)

like in algebraic data types | term algebras

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

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

Unification

Finds a solution (substitution) to a set of equations

- works for any constraint set of equalities between (type) constructors: they are injective functions f(x)=f(y)-> x=y
- finds the most general solution

Definition

A set of equations is in *solved form* (compare to Gaussian elimination!) if it is of the form

$\{x_1 = t1, \ldots x_n = t_n\}$ and variables x_i do not appear in terms t_i, that is $\{x1, \ldots, x_n\} \cap (FV(t1) \cup \ldots \cup FV(t_n)) = \emptyset$

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 $f(t1, \ldots, t_n) = f(s1, \ldots, s_n)$,
replace with $t1 = s1, \ldots, t_n = s_n$.

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

Decomposition Clash: $f(t1,\ldots,t_n) = g(s1,\ldots,s_n)$, $f \neq g$, report clash.

e.g. $(T_1 \times T_2) = (S_1 \text{->} S_2)$ 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:

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

Example 2
Construct and Solve Constraints

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

$:TF$ $:TR$ $:TX$

$\underbrace{f(\underbrace{f(x)}_{TA})}_{TB}$

$TR = TX \Rightarrow TX$
$TF = TX \Rightarrow TX$
$TA = TX$
$TB = TX$

$TR = (TX \Rightarrow TB)$ ✓

$TF = (TX \Rightarrow TA)$

$(TF = (TA \Rightarrow TB))$

$TX \Rightarrow TA = TA \Rightarrow TB$

$(TX = TA)$ $(TA = TB)$

$TA = TX$ $TB = TA$

Example 2

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))
```

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

Are there others? How do all solutions look like?

$f : TX_1 \Rightarrow TX_1$ $\quad$ $f : TX_2 \Rightarrow TX_2$

Instantiating Type Variables

$[TX]$ $:TX$ $\quad$ $:TX$

```
def f(x) = x
```

$TX_1 = Bool$ $\quad$ TX_1 $\quad$ $TX_2 = Int$ $\quad$ $Bool = Int$

```
def test() = if (f(true)) f(30)
                else f(42)
```

Int $\quad$ Int

Generate and solve constraints.

Is result different if we clone f for each invocation?

```
def f1(x) = x
def f2(x) = x
def f3(x) = x
def test() = if (f1(true)) f2(30)
                else f3(42)
```

Generalization Rule

- If after inferring top-level (immutable) function definitions certain variables remain unconstrained, then generalize these variables and make them into type parameters T:

def f[T](…)   if T was not constrained

- When applying a function with generalized variables, rename these variables into fresh ones

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

Exercise

$\forall T.\ T \times List[T] \Rightarrow List[T]$

```
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))
```

$:TA$ $\quad :TB$ $\quad TD$ $\quad TC. \quad TB.$

$TA = (TB \Rightarrow TC)$

$T_1 \times List[T_1] \Rightarrow List[T_1]$

$= TC \times TB \Rightarrow TD$

$TB = List[T_1] \qquad TD = List[T_1]$

$TC = T_1$