Translate This While Loop using Rules that Explicitly Put Booleans on Stack

```
static void count(int from,
                  int to,
                  int step) {
  int counter = from;
  while (counter < to) {
    counter = counter + step;
  }
}
```

Towards More Efficient Translation Compiling by Passing Destinations

Macro ‘branch’ instruction

Introduce an imaginary big instruction

branch(c,nThen,nElse)

Here

c is a potentially *complex* Java boolean expression,
that is the main reason why branch is not a real instruction

nThen is label to jump to when c evaluates to true

nFalse is label to jump to when c evaluates to false

no “fall through” – always jumps (for symmetry)

We show how to:

- use **branch** to compile if, while, etc.
- expand **branch** recursively into concrete bytecodes

Using **branch** in Compilation

[if (c) t **else** e **]** =

```
        branch(c,nThen,nElse)
nThen: [ t ]
        goto nAfter
nElse: [ e ]
nAfter:
```

[while (c) s **]** =

```
lBegin: branch(c,start,lExit)
start:  [ s ]
        goto lBegin
lExit:
```

Decomposing **branch**

branch(**!**c,nThen,nElse) =
        branch(c,nElse,nThen)

branch(c1 **&&** c2,nThen,nElse) =
        branch(c1,nNext,nElse)
nNext: **branch**(c2,nThen,nElse)

branch(c1 **||** c2,nThen,nElse) =
        branch(c1,nThen,nNext)
nNext: **branch**(c2,nThen,nElse)

branch(**true**,nThen,nElse) =
    goto nThen

branch(**false**,nThen,nElse) =
    goto nElse

boolean var b with slot N

branch(b,nThen,nElse) =
    iload_N
    ifeq nElse
    goto nThen

Compiling Relations

branch(e1 **R** e2,nThen,nElse) =

> **[** e1 **]**
>
> **[** e2 **]**
>
> **if_cmpR** nThen
>
> **goto** nElse

Putting boolean variable on the stack

Consider storing x = c

where x,c are boolean and c has &&,||

How to put result of **branch** on stack to allow **istore**?

[c **]** = **branch**(c,nThen,nElse)

```
nThen: iconst_1
       goto nAfter
nElse: iconst_0
nAfter:
```

Compare Two Translations of This While Loop

```
while (counter < to) {
   counter = counter + step;
 }
```

old one:

```
nbegin:  iload #counter
         iload #to
         if_icmplt ntrue
         iconst_0
         goto nafter
ntrue:   iconst_1
nafter:  ifeq nexit
         iload #counter
         iload #step
         iadd
         istore #counter
         goto nbegin
nexit:
```

Complex Boolean Expression: Compare

Old code for assignment

y = (x && y) && !z

New code:

Code Compiled with javac

```
static int k = 0;
static boolean action(int si,
                      boolean ob,
                      int sm, int pr) {
        if (sm + 2*pr > 10 &&
        !(si <= 5 && ob)) {
          k++; return true;
        } else {
          return false;
        } }
```

```
0:   iload_2
1:   iconst_2
2:   iload_3
3:   imul
4:   iadd
5:   bipush  10
7:   if_icmple      29
10:  iload_0
11:  iconst_5
12:  if_icmpgt      19
15:  iload_1
16:  ifne    29
19:  getstatic      #2; //Field k
22:  iconst_1
23:  iadd
24:  putstatic      #2; //Field k
27:  iconst_1
28:  ireturn
29:  iconst_0
30:  ireturn
```

Compared to our current translation:

if 'sm+2*pr > 10' false, immediately ireturns

if 'si > 5' is true, immediately goes to 'then' part

no intermediate result for if condition - do branches directly

negation sign eliminated and pushed through

only one iconst_0 and one iconst_1

Implementing **branch**

- Option 1: emit code using branches, then rewrite
- Option 2: **branch** is a compilation function in the compiler that expands

branch(c,nTrue,nFalse)

$\downarrow$

```
def compileBranch(c:Expression,
    nTrue : Label, nFalse : Label) : List[Bytecode] =
{ … }
```

More Complex Control Flow

break statement

A common way to exit from a loop is to use a 'break' statement e.g.

```
while (true) {
  code1
  if (cond) break
  cond2
}
```

Consider a language that has expressions, assignments, the {…} blocks, 'if' statements, while, and a 'break' statement. The 'break' exits the innermost loop and can appear inside arbitrarily complex blocks and if conditions. How would translation scheme for such construct look like?

Destination Parameters in Compilation

- To compilation functions **[…]** pass the label to which instructions should jump when they finish.
  - No fall-through

```
[ x = e ] dest =                  // new parameter dest
   [ e ]
   istore_slot(x)
   goto dest                      // at the end jump to it


[ s1 ; s2 ] dest brk =
                 [ s1 ] freshL
   freshL:       [ s2 ] dest
```

we could have any junk in here because ([s1] freshL) jumps

More Control, More Destination Paramameters

[s1 ; s2 **]** dest brk =

```
                **[** s1 **]** freshL brk
    freshL:     **[** s2 **]** dest brk
```

[x = e **]** dest brk =

```
    [ e ]
    istore_slot(x)
    goto dest
```

[break] dest brk =

```
    goto brk
```

[while (c) s **]** dest brk =

```
test:   branch(c,body,dest)
body:   [ s ] dest dest
```

this is where the second parameter gets bound to the exit of the loop

break and **continue** statements?

- Describe how to modify previous translation

Some High-Level Instructions for JVM

Method Calls

Invoking methods (arguments pushed onto stack)

invokestatic

invokevirtual

Returning value from methods:

ireturn – take integer from stack and return it

areturn – take reference from stack and return it

return – return from a method returning ‘**void**’

invokestatic

invokestatic
indexbyte1
indexbyte2

Operand Stack ..., [arg1, [arg2 ...]] → ...

The unsigned **indexbyte1** and **indexbyte2** are used to construct an index into the run-time **constant pool** of the current class (§2.6), where the value of the index is **(indexbyte1 << 8) | indexbyte2**. The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). It must be static, and therefore cannot be abstract.

On successful resolution of the method, the class that declared the resolved method is initialized (§5.5) if that class has not already been initialized.

The operand stack must contain nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is synchronized, the monitor associated with the resolved Class object is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The nargs argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 (or, if arg1 is of type long or double, in local variables 0 and 1) and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The nargs argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

If the native method is synchronized, the monitor associated with the resolved Class object is updated and possibly exited as if by execution of a monitorexit instruction (§monitorexit) in the current thread.

If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

invokevirtual

invokevirtual

indexbyte1

indexbyte2

..., objectref, [arg1, [arg2 ...]] →

...

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is **(indexbyte1 << 8) | indexbyte2**. The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of objectref must be either the current class or a subclass of the current class.

If the resolved method is not signature polymorphic (§2.9), then the invokevirtual instruction proceeds as follows.

Let C be the class of objectref. The actual method to be invoked is selected by the following lookup procedure:

If C contains a declaration for an instance method m that overrides (§5.4.5) the resolved method, then m is the method to be invoked, and the lookup procedure terminates.

Otherwise, if C has a superclass, this same lookup procedure is performed recursively using the direct superclass of C; the method to be invoked is the result of the recursive invocation of this lookup procedure.

Otherwise, an AbstractMethodError is raised.

The objectref must be followed on the operand stack by nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with objectref is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values and objectref are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The objectref and the argument values are consecutively made the values of local variables of the new frame, with objectref in local variable 0, arg1 in local variable 1 (or, if arg1 is of type long or double, in local variables 1 and 2), and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

...

Translation Rule for Method Calls

[x = objExpr.myMethodName(e1,e2) **]** =

- **[** objExpr **]**
- **[** e1 **]**
- **[** e2 **]**
- **invokevirtual** #13
- **istore** #x

constant pool area:

0: "hello, world"

1:

...

13: className.myMethodName/(II)I

...

Objects and References

ifnull label - consume top-of-stack reference and jump if it is null

ifnonnull label - consume top-of-stack reference, jump if *not* null

new #className - create fresh object of class pointed to by the offset
#className in the constant pool
(does not invoke any constructors)

getfield #field – consume object reference from stack,
then dereference the field of that object given
by (field,class) stored in the #field pointer in the constant pool
and put the value of the field on the stack

putfield #field - consume an object reference obj and a value v
from the stack and store v it in the #field of obj

"If the field descriptor type is boolean, byte, char, short, or int, then the value must be an int."

Array Manipulation

a = reference - “address” arrays

i = int arrays (and some other int-like value types)

Selected array manipulation operations:

- **newarray**, **anewarray**, **multianewarray** – allocate an array object and put a reference to it on the stack
- **aaload**, **iaload** – take: a reference to array and index from stack load the value from array and push it onto the stack
- **aastore**, **iastore** – take: a reference to array, an index, a value from stack, store the value into the array index
- **arraylength** – retrieve length of the array

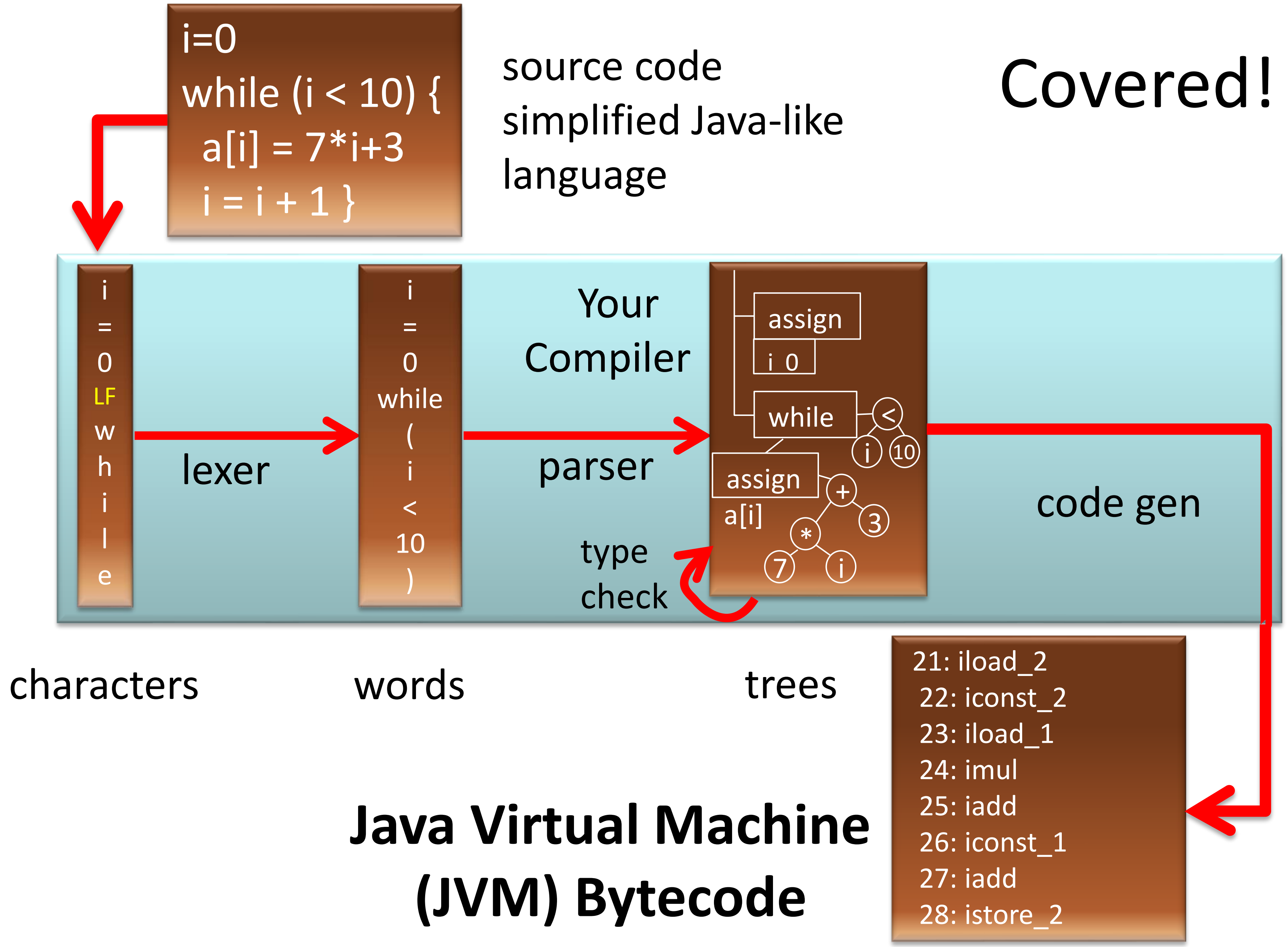
Java arrays store the size of the array and its time, which enables run-time checking of array bounds and object types.

There are Floating Point Operations…

- fadd
- faload (for floating point arrays)
- fastore (for floating point arrays)
- fcmp<op>
- fconst_<f>
- fdiv
- fload
- fload_<n>
- fmul
- fneg
- frem
- freturn
- fstore
- fstore_<n>
- fsub

When needed,
READ THE JVM Spec ☺

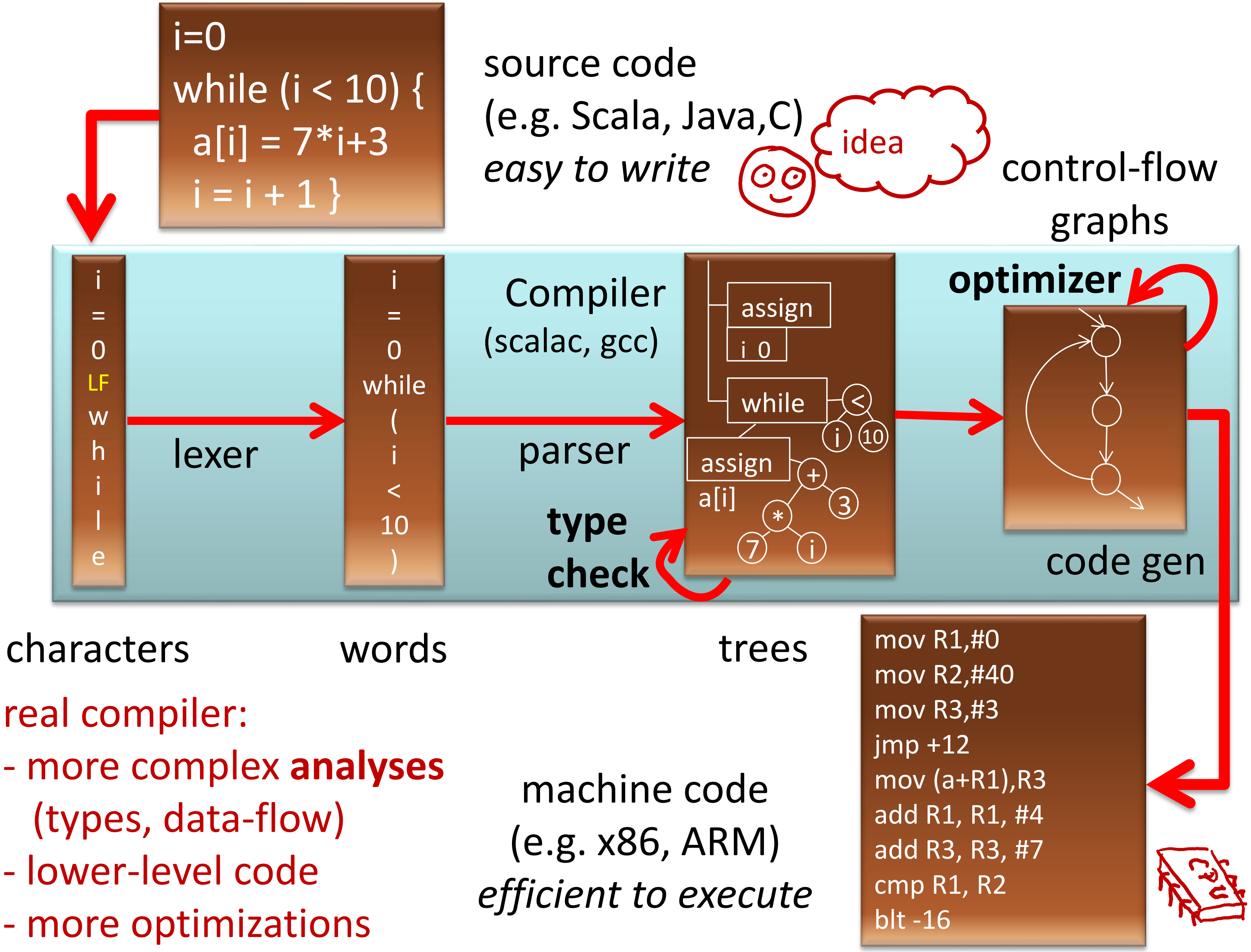
Covered!



```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

```
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 }
```

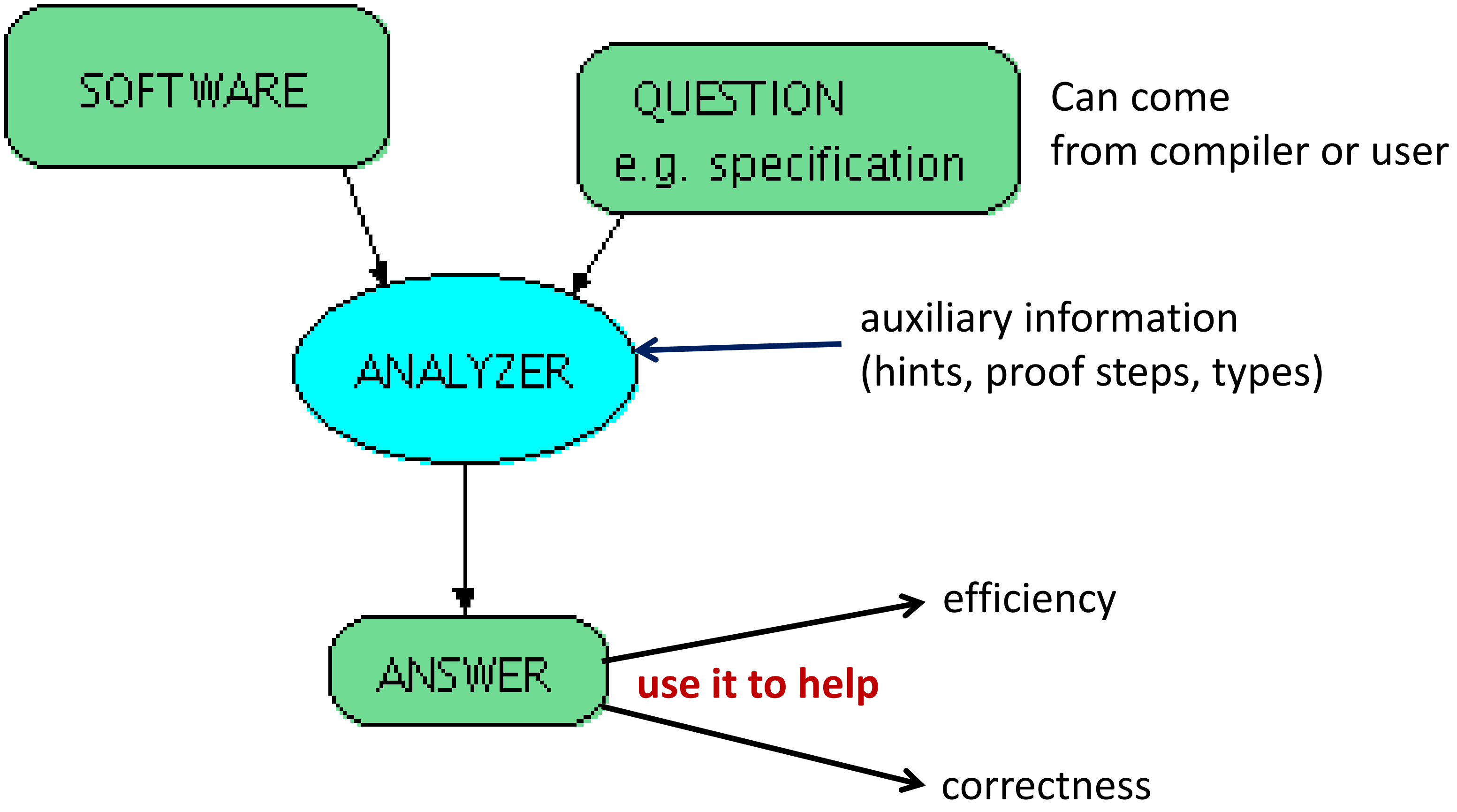
```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

real compiler:

- more complex **analyses** (types, data-flow)
- lower-level code
- more optimizations

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

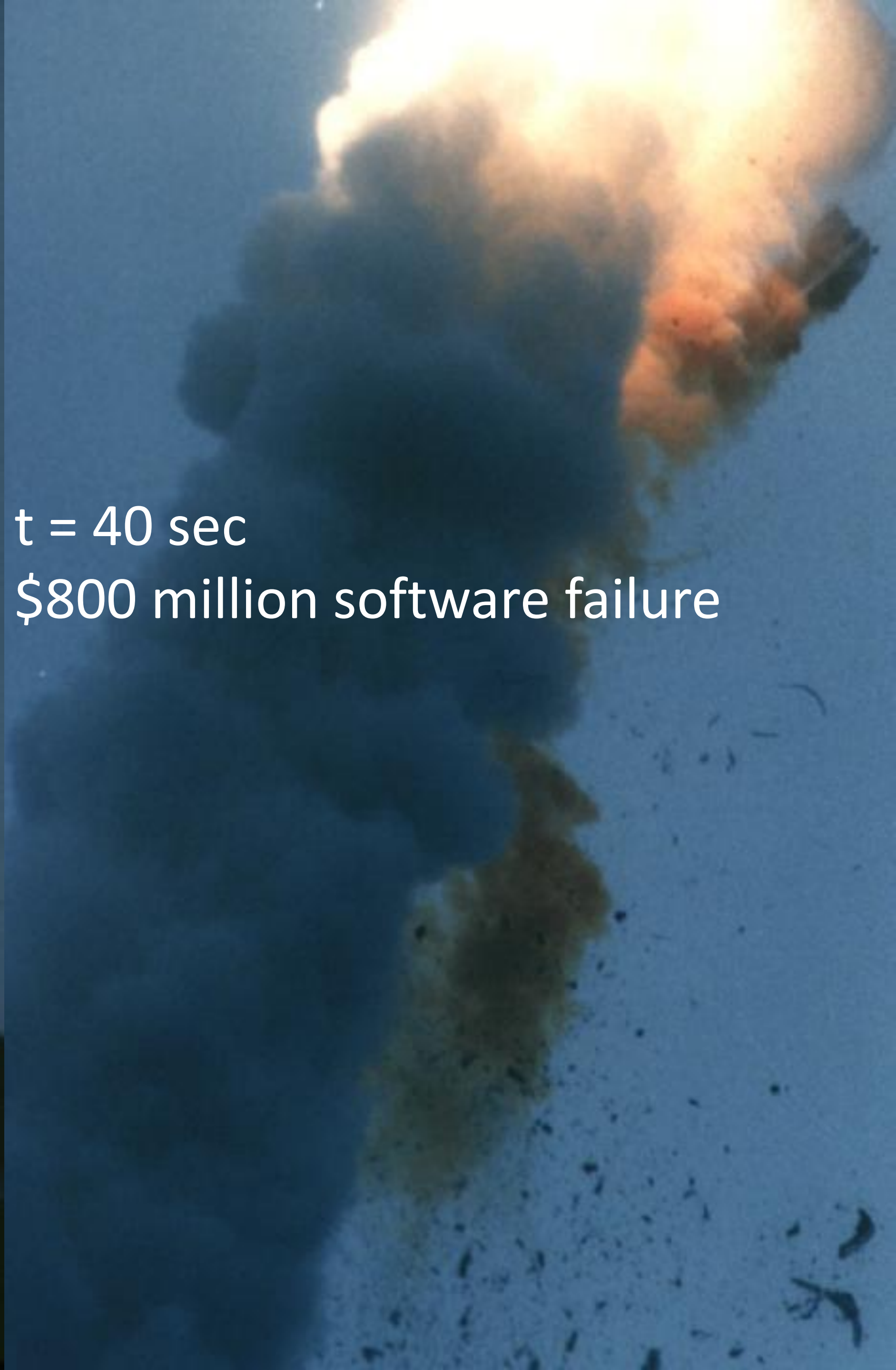
Examples of Verification Questions

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?

French Guyana, June 4, 1996

t = 0 sec


t = 40 sec

$800 million software failure

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.

Air Transport

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

Can it type-check?

```
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)
}
```

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

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

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

Rules for Solving Equations

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

substitute
equals for equals

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

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

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

$$T_1 \times T_2 \times T_3$$

$$T_1 \times (T_2 \times T_3)$$

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

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

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

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

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

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

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

Decomposition Clash: $f(t1,…,t_n) = g(s1,…,s_n)$, $f \neq g$, 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

$$\overset{[A]\ A\to A}{}$$

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

$$\text{twice}(f : TF) : TA = \underbrace{(x : TX) \Rightarrow \overbrace{f(\underbrace{f(x)}_{TR})}^{TB}}_{TA}$$

$TA = TB \to TB$

$\underbrace{TA = TX \to TB}$

$TF = TB \to TB$

$\underbrace{TF = TX \to TR}$

~~$TF = TR \to TB$~~

~~$TX \to TR = TR \to TB$~~

~~$TX = TR$~~

$TX = TB$

$\underbrace{TR = TB}$

twice : ~~$TT = TF \to TA$~~

~~$TT = TF \to (TX \to TB)$~~

$TT = (TX \to TR) \to (TX \to TB)$

$TT = (TB \to TB) \to (TB \to TB)$

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))
```

$\overset{[TX]}{f}(x) : TX$

$\overset{[TX]}{g}(a) : TA$

$\overbrace{f(\underbrace{f(a)}_{TB})}^{TC}$

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`

$\boxed{\begin{array}{l} TA = TX \\ TB = TX \\ TC = TX \end{array}}$

Are there others? How do all solutions look like?

$\boxed{\begin{array}{l} TF = TX \to TX \\ TG = TX \to TX \end{array}}$

Instantiating Type Variables

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

$\to [TX] : TX$

f_1

$f_2(x) = x$

f_1 f_2 f_2

$f : TF$

$TF = \underbrace{TX \to TX}$

TT TE

$TX_1 = \text{Boolean}$

$TX_1 = \text{Boolean}$

Generate and solve constraints.

Is result different if we clone f for each invocation?

$\underbrace{f : \forall TX.\ TX \to TX}$

$TX_2 = \text{Int}$

$TX_2 = TT$

$TX_3 = \text{Int}$

$TX_3 = TE$

$TE = TT$

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))
```