



INTERMEZZO

TYPE ANALYSIS

La volta scorsa abbiamo visto i tipi statici di un linguaggio di natura funzionale perchè erano i tipi che noi avevamo creati come annotazioni delle espressioni che catturavano i comportamenti astratti del programma. Esistono altre nozioni che emergono a tempo di esecuzione con i dati concreti e non quelli astratti. Nei linguaggi ad oggetti uno ha la nozione di tipo della classe a tempo di compilazione.

Static types and dynamic types

- The dynamic type of an object is the class *C* that is used in the “*new C*” expression that creates the object
 - - A run-time notion
 - Even languages that are not statically typed have the notion of dynamic type
- The static type of an expression is a notation that captures all possible dynamic types the expression could take
 - A compile-time notion

Il polimorfismo di sottotipo, nei linguaggi come per esempio il Java, il tipo statico è un tipo più generale che contiene tutti i sottotipi dinamici contenuti in quella classe

Qui vedete un qualcosa scritto in un linguaggio tipo Java e vedete, ci sono tutte le informazioni di natura statica che io posso associare all'oggetto ad esempio, vedete, abbiamo quella, quella string che si chiama greeting che è una costante che è privata, quindi è un'informazione di tipo statico che ci dice che è accessibile soltanto all'interno della classe demo e quindi dall'esterno di questa classe non è visibile e il fatto che venga messo che abbiamo definito una costante in un valore intero che chiamiamo con CONST con Final static Int ci dice che quella è una costante che è una variabile d'istanza perché c'è il costrutto statico, il costrutto statico che vuol dire che appartiene a tutte le istanze degli oggetti della classe e non alle variabili d'istanza.

Example

```
public class Demo{
    static private string greeting = "Hello";
    final static int CONST = 43;

    static void Main (string[] args){
        foreach (string name in args){
            Console.WriteLine(sayHello(name));
        }
    }

    public static string sayHello(string name){
        return greeting + name;
    }
}
```

**greeting only accessible
in class Demo**

CONST will *always* be 43

**sayHello will always return
a string**

**sayHello will always be called
with 1 parameter
of type string**

Un'altra informazione statica è la segnatura dei metodi, ad esempio il metodo sayHello ha una segnatura che ci dice che prende una stringa come parametro e restituisce una stringa. Queste sono tutte nozioni di tipo statico che sono gestite dal compilatore.

Example

```
class A
{
    void m1() { ... }
}

class B extends A
{
    // overriding m1()
    void m1() { ... }
}

class C extends A
{
    // overriding m1()
    void m1() { ... }
}
```

//DRIVER

```
A a = new A();
B b = new B();
C c = new C();
:
A ref;
ref.m1()
ref = b;
ref.m1();
:
ref = c;
c.m1()
```

The version of m1() executed is determined by the dynamic type of object being referred to at the time of the invocation.

Questo esempio mi crea una classe che ha solo una variabile di istanza privata e che è una variabile intera a cui associo il valore. 42 non ha niente, non ha metodi d'accesso, quindi se io vado a prendere strettamente le caratteristiche del tipo statico delle informazioni sui modificatori che vado a vedere la classe sto sostanzialmente definendo una variabile d'istanza che non potrà mai essere utilizzata perché l'unica entità che possono accedere alla variabile privata x sono quelle all'interno della classe, ma l'interno della classe non hanno alcun metodo per operare. Però ad esempio potrei definire questa classe che definisce una classe pubblica che ha un unico metodo pubblico e il metodo m. Supponiamo che la classe A sia all'interno della stessa unità di compilazione dove ha definito la classe Secret, quindi vuol dire che può vedere la definizione della classe Secret, quindi possono essere anche compilate separatamente dal compilatore di Java, ma sicuramente quando andando a mandare in esecuzione la classe ho tutte le informazioni nei file presenti nel run time che mi permettono di accedere a la classe Secret.

However

```
class Secret { private int x = 42; }
```

```
public class A {  
    public void m() {  
        try {  
            Secret o = new Secret();  
            Class c = o.getClass();  
            Field f = c.getDeclaredField("x");  
            f.setAccessible(true);  
            System.out.println("x="+f.getInt(o)) ;  
        }  
        catch (Exception e) {System.out.println(e); }
```

```
import java.lang.reflect.*;
```

```
jshell> ref = new A()  
ref ==> A@41cf53f9
```

```
jshell> ref.m()  
x=42
```

Questo metodo crea un oggetto della classe Secret e poi usa l'introspezione ovvero usa un API di Java che permette a tempo di esecuzione di invocare dei metodi che sono dei metodi meta, sono dei metodi che permettono di operare sulla struttura degli oggetti della classe. Perché a run time nel caso di Java abbiamo la possibilità di avere quelli che si chiamano i file .class che contengono relativamente agli oggetti di una classe, quella che si chiama la Constant Pool che è una struttura del byte code di Java che permette di avere tutte le informazioni relative a gli oggetti della classe e permette di usare queste API che permettono di operare con questa informazione. Allora in modo particolare quello che uno potrebbe fare può andare a crearsi un oggetto di tipo classe e quindi si vede bene che siamo a livello meta e si crea un oggetto di tipo classe, andando a prendere dall'oggetto che ho creato di tipo Secret l'informazione associata della classe. Dove la vado a beccare? La vado a beccare esattamente nel runtime, nell'informazione associata alla classe poi vado a vedere il campo e quindi ovviamente qui devo sapere che il campo si chiama x perché devo dargli come parametro alla getDeclaredField il campo x. Poi dico che a questo punto vado a modificare i meccanismi di accesso, cioè sto dicendo che F diventa accessibile quindi non è più privato e a questo punto, vado a stampare il valore associato a F perdendo le informazioni dall'oggetto. Quindi questo cosa vuol dire? Vuol dire che uno deve fare particolare attenzione al linguaggio di programmazione che sta utilizzando e all'informazione che mette in questo linguaggio di programmazione perché le caratteristiche del linguaggio di programmazione, come quelle che abbiamo visto qui di natura statica che dicono io do l'accesso soltanto ai valori della classe, se io uso la

Type safety

- Type-safety programming language guarantees that programs that pass the type-checker can only manipulate data in ways allowed by their types
 - One cannot multiply booleans, dereference an integer, take the square root of reference,
- Type safety avoids lots of room for undefined behaviour
 - In OO languages: no “Method not found” errors at runtime

Allora a questo punto affrontiamo il problema della type safety. Se il linguaggio di programmazione mette un sistema di tipo staticamente, il sistema di tipo fa tra virgolette passare, cioè controlla che il linguaggio è conforme tramite un type checker all'uso dei tipi. allora noi diciamo che il linguaggio è Type safety se la manipolazione che ho degli operatori del linguaggio sui valori presenti a tempo di esecuzione è compatibile con lui, con i tipi. Quindi che cosa vuol dire? Vuol dire che uno non può moltiplicare come ho scritto qui, due booleani non può dereferenziare un intero etc... Esempio in molti linguaggi di programmazione, pur avendo un controllo statico il metodo not found è disponibile a runtime, quindi avere la Type safety aiuta.

Discussion

- Programming languages can be
 - memory-safe, typed, and type sound:
 - Java, C#, Rust, Go
 - though some of these have loopholes to allow unsafety
 - Functional languages such as Haskell, ML, OCaml, F#
 - memory-safe and untyped
 - LISP, Prolog, many interpreted languages
 - memory-unsafe, typed, and type-unsafe
 - C, C++
 - Not type sound: using pointer arithmetic in C, you can break any guarantees the type system could possibly make



Breaking type soundness (in C++)

For a C(++) program we can make no guarantees whatsoever in the presence of untrusted code.

a buffer overflow in some library can be fatal
in a code review we have to look at all code to make guarantees

```
class DiskQuota {  
    private:  
        int MinBytes;  
        int MaxBytes;  
};
```

```
void EvilCode(DiskQuota* quota) {  
    // use pointer arithmetic to access  
    // the quota object in any way we like!  
    ((int*)quota)[1] = MAX_INT;
```


Avoiding buffer overflow (Java, C#)

- Language-based solution: Buffer overflow is ruled out at **language-level**, by combination
 - **compile-time** typechecking (static checks)
 - **at load-time**, by bytecode verifier (bcv)
 - **runtime** checks (dynamic checks)

JAVA RUN TIME CHECK

```
public class A extends Super{  
    protected int[] d;  
    private A next;  
  
    public A() { d = new int[3]; }  
    public void m(int j) { d[0] = j; }  
    public setNext(Object s)  
        next = (A) s;  
    }  
}
```

runtime checks for
1) non-nullness of d,
and 2) array bound

runtime check for
type (down)cast

Come viene evitato ad esempio, il buffer overflow in Java e C#? Viene evitato con una combinazione di controlli statici di controlli a tempo di compilazione e di controlli a tempo di esecuzione. Allora il controllo statico, va a vedere se ad esempio i valori degli indici degli array sono conformi alle caratteristiche, però non va ancora a calcolare, a fare il controllo sulla dimensione se esce fuori perché quella è una nozione dinamica, quindi quello non lo fa però in tutte quelle situazioni che ad esempio si riescono a calcolare staticamente lo fa. Quando il bytecode verifier va a caricare un .class va a controllare se il tipo del byte coded che ho generato è compatibile con le caratteristiche del linguaggio. Ogni volta che io faccio un accesso ad un array in Java, quando quindi crea un oggetto della classe, tra le meta informazioni che a tempo di esecuzione vado a mettere nelle meta informazioni di tipo associate agli oggetti che ho creato c'è un'informazione che è la dimensione dell'array. Tutte le operazioni dell'accesso fanno riferimento a questa informazione che tipicamente si chiama length. A runtime vado a controllare se l'array è non nullo e a controllare gli indici delle array queste due cose se non sono verificate generano delle eccezioni a tempo di esecuzione e similmente ad esempio io posso definirmi un metodo set next che è parametrico rispetto al tipo del parametro, allora a questo punto quando vado ad associare questo valore alla variabile d'istanza, faccio un'operazione di casting, quindi dico che l'oggetto S viene fatto il casting su A e quello viene associato alla variabile d'istanza next. Bene a questo punto noi abbiamo ancora dei check a tempo di esecuzione che vanno a controllare se effettivamente il casting, in questo caso il downcasting è ammissibile quindi vedete che abbiamo un overhead a tempo di esecuzione.

Buffer overflow still exists in Java and C#

- Buffer overflows can still exist:
 - in native code
 - C#: code blocks declared as unsafe
 - through bugs in the Virtual Machine (VM) implementation, which is typically written in C++....
 - through bugs in the implementation of the type checker, or worse, bugs in the type system (unsoundness)
- The VM (incl. the type checker, byte code verifier) is part of the **Trusted Computing Base** (TCB) for memory and type-safety,

Però, come dicevo poc anzi, e usando i metodi nativi, ovviamente posso avere ancora buffer overflow in Java e tipicamente quello che succede è che anche in c# posso avere dei metodi che possono fare overflow e che posso marcare come unsafe e quindi il sistema in un qualche modo ha un po di informazione, ciò nonostante possa avere dei metodi overflow. Quando abbiamo linguaggi con queste caratteristiche tipo Java e C# ad esempio a questo punto tutte le informazioni relative al controllo dei tipi e quindi tutte quelle informazioni che mi permettono di avere la type soundness e la memory safedance sono meccanismi che devono essere presenti nella TCB. Se si vuole avere un linguaggio con caratteristiche di safety il prezzo da pagare è avere una TCB allargata rispetto a quella precedente.

Discussion

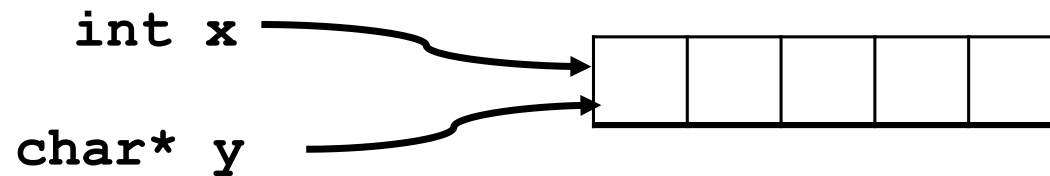
Type safety is an extremely **fragile** property:
one tiny flaw brings
the whole type
system crashing
down

In alcuni linguaggi di programmazione non si fa particolare attenzione all' aliasing e quello che potrebbe accadere (spesso nei linguaggi di scripting) è che io possa avere due cammini di accesso alla stessa struttura dati, proprio una nozione di riferimento come abbiamo nei linguaggio oggetti, ma immaginate che di averla nei linguaggi di scripting dove a questo punto ho due riferimenti di tipo diverso che puntano alla stessa struttura di memoria sull'heap. Allora se il linguaggio permette di fare questa operazione tutto quello che abbiamo detto pocanzi sul sistema dei tipi si va a fare benedire. Questo è il motivo, ad esempio, per cui in Rust ci sono dei controlli molto forti per evitare che ci sia l' alias, perché l' alias è una nozione complicata.



Example: type confusion

Data values and objects are just blobs of memory. If we can create **type confusion**, by having **two references with different types pointing the same blob of memory**, then *all* type guarantees are gone.



Type confusion attacks

```
public class A{  
    public Object x;  
    ...  
}
```



What if we could compile **B** against **A**
but we run it against **A**?

We can do pointer arithmetic again!

If Java Virtual Machine would allow
such so-called *binary incompatible*
classes to be loaded, the whole
type system would break.

```
public class A{  
    public int x;  
    ...  
}  
  
public class B{  
    void setX(A a) {  
        a.x = 12;  
    }  
}
```

Type system (soundness)

- **Representation independence**

- Booleans: it does not matter if we represent true as 0 and false as 1 (or viceversa),
- if we execute a given program with either representation, the result is guaranteed to be the same.

- **Theory helps: Given a formal mathematical definition of the programming language, we could prove that it does not matter how true and false are represented for all programs**

Altra caratteristica che il sistema dei tipi e i meccanismi dei linguaggi sui tipi permettono di avere è una indipendenza delle astrazioni della rappresentazione utilizzata. Questo cosa vuol dire? Vuol dire che se io so che il mio linguaggio opera su booleani interi, float, vado a fare il controllo e ha una consistenza di tipo rispetto ai valori degli elementi allora questo vuol dire che nella macchina virtuale dell'esecuzione del linguaggio posso rappresentare i booleani con un byte con 64 byte o con quello che si ha e a questo punto però io sono astratto, cioè sono indipendente dalla scelta di rappresentazione e ovviamente questo è importante perché se noi assicuriamo che i programmi sono indipendenti dalla scelta di rappresentazione dei dati e quindi il comportamento non è influenzato dalla scelta di rappresentazione dei dati, abbiamo una bella proprietà, ovviamente qui c'è un mucchio di teoria dietro nella teoria generale dei linguaggi di programmazione.

Tipicamente quello che succede quando uno deve progettare un linguaggio ha un modello del comportamento del linguaggio come se io lo eseguisse facendo dei controlli dei tipi mentre lo vado a eseguire. In termini formali significa che io uso una semantica operativa quindi un modello del comportamento dell'esecuzione del linguaggio con i tipi. Quindi questo cosa vuol dire? Vuol dire che vado a vedere quali sono i controlli che posso fare a run time e come li faccio usando le meta informazioni dei tipi. Parallelamente uno ha un modello del comportamento che non usa i tipi, cioè ha un modello del comportamento e quindi dell'esecuzione del linguaggio dove assume che i controlli di tipo vengono fatti staticamente. Vuol dire che a tempo di esecutore non si deve controllare la conformità del comportamento di esecuzione dei programmi rispetto ai tipi, perché è controllata staticamente.

Type systems (summary)

- Two formal definitions of the programming language
 - a **typed operational semantics**, which records and checks type information at runtime
 - an **untyped operational semantics**, which does not
- Prove their equivalence for all well-typed programs:
 - **prove the equivalence of a defensive execution engine (which records and checks all type information at runtime) and a normal execution engine which does not for any program that passes the type checker.**
- Research: Several works have formalised the semantics and type system of Java, using theorem provers (Coq, Isabelle/HOL), to prove such results.



Richer forms of types

- Distinguish **non-null** & **possibly-null** (nullable) types

public @NonNull String hello = "hello";

- to improve efficiency
- to prevent null pointer bugs or detect (some) of them earlier, at compile time
- Programming language mainstream:
- C# supports nullable types written as `A?` or `Nullable<A>`
 - In Java you can use type annotations `@Nullable` and `@NonNull`
 - Scala, Rust, Swift have non-null vs nullable option types

Uno a questo punto capito l'uso dei tipi può dire ah ma i tipi che ho nel linguaggio di programmazione possono averne di migliori, ad esempio chi ha avuto un po' di esperienza nella programmazione orientata ad oggetti sa che se non fa attenzione e non mette particolare attenzione nella descrizione dei comportamenti del linguaggio, la tipica cosa che farà saranno degli errori a run time perché cerca di accedere a un oggetto che è ancora null. Null non vuol dire che l'oggetto è vuoto, null vuol dire che non è stato creato in memoria, non è presente nell'heap. Ci sono un mucchio di esempi di situazioni e di errori famosi che sono nati perché uno usa valori null. Ad esempio in Java uno può annotare con nullable e non nullable e Scala Rust Swift hanno esattamente gli option Type, esattamente come Ocaml. Vuol dire che la scelta di avere dei meccanismi di tipo più ricchi rispetto a quello che uno si aspetta, è una cosa che sta ancora andando avanti.

Altre cose che sono presenti nei linguaggi di programmazione prima citavo Rust, quello nato da sostanzialmente dalla Mozilla Foundation, ammette dei meccanismi di alias controllo, cioè vuole evitare che ci siano delle interferenze dovute al fatto che ho diversi cammini di accesso, alle stesse strutture sullo heap e vuole evitare questa caratteristica.

Richer forms of types

- **Alias control:** restrict possible interferences between modules due to aliasing.
- **Information flow:** controlling on the way tainted information flows through an implementation.
- **Resource usage and access control:** controlling accesses to resources

We will discuss Information flow, Resource usage and access control in further lectures

Vedremo nel dettaglio la caratteristica di avere delle nozioni di tipo che permettono di avere all'interno del linguaggio la possibilità di definire delle politiche di uso delle risorse di controllo degli accessi all'interno del linguaggio di programmazione e poi avere delle informazioni di tipo che vanno a controllare come fluiscono i dati all'interno di un applicazione per evitare che dei dati privati vengano messi a disposizione di entità che non hanno accesso a dati privati.

Allora andiamo a vedere un discorso che è un discorso relativo a la mutabilità e la non mutabilità. Le strutture dati immutabili sono quelle strutture che io non posso avere delle operazioni che fanno delle modifiche le strutture dati immutabili, ad esempio un array è una struttura che se io ho un accesso a un indice dell'array lo possa modificare.

Language-based guarantees

- **visibility**: public, private, ...
 - private fields not accessible from outside a class
- **immutability**
 - In Java: `final int i = 5;`
 - in C(++) : `const int BUF_SIZE = 128;`
 - In Java String objects are immutable
- Scala, Rust provides a more systematic distinction between mutable and immutable data to promote the use of immutable data structures

Le stringhe sono immutable, gli interi ugualmente e ugualmente per gli array. Nei linguaggi funzionali le liste sono immutable e così via e c'è una distinzione molto netta, in modo particolare scala e rust sono due linguaggi di programmazione che fanno dell'operare con strutture dati immutable la loro caratteristica principale.

A blue brushstroke graphic with a rough, hand-painted edge, containing the text.

Thread Safety & Aliasing

Races

Supponiamo di avere un linguaggio che permette di avere la possibilità di definire dei thread che vengono mandati in esecuzione. Supponiamo di avere due thread banali banali che incrementano il valore di una variabile che inizialmente è inizializzata a zero.

- Two threads both execute the statement
 $x = x + 1;$
where **x** initially has the value **0**.
- What is the value of x in the end?
 - **x can have value 2 or 1**
- **Data race**: $x = x + 1$ is not an **atomic operation**, but happens in two steps - reading x and assigning it the new value - which may be interleaved in unexpected ways

Mandato questi due thread in esecuzione, a questo punto quello che uno si aspetta dice quale sarà il valore alla fine che ho? Il valore che alla fine ho è 2. Perché incrementa ma in realtà potrebbe avere anche uno. Questo perché? Perché il punto è che nell'implementazione dei linguaggi l'operazione $x = x + 1$ non è un'operazione atomica. Perché? Viene fatta in due fasi, prima si legge il valore della variabile e poi lo si scrive e queste due fasi possono essere intercalate tra i due processi in esecuzione.

```
class A {  
    private int i ;  
    A() { i = 5 ;}  
    int geti() { return i; }  
}
```

Can geti() ever return
something else than 5?

Yes!

Thread 1, initialising x

```
static A x = new A();
```

Thread 2, accessing x

```
j = x.geti();
```

You'd think that here x.geti() returns 5 or
throws an exception, depending on
whether thread 1 has initialised x

Hence: x.geti() in thread 2
can return 0 instead of 5

Execution of thread 1 takes in 3 steps

1. allocate new object m
2. m.i = 5;
3. x = m;

the compiler or VM is allowed to swap the order of these
statements, because they don't affect each other



Solution

```
class A {  
    private final int i ;  
    A() { i = 5 ;}  
    int geti() { return i;}  
}
```

Now geti() always return 5.

Declaring a private field as **final** fixes this particular problem

- due to ad-hoc restrictions on the initialisation of final fields

Un esempio di data race è quando ho dei comportamenti di thread che accedono a delle variabile o più in generale a delle strutture dati condivise. e a questo punto il comportamento dei thread è non deterministico perché dipende dalle politiche di accesso e dalle politiche di assicurazione che i dati condivisi non siano accessibili in un modo strano ai thread d'esecuzione.

Data races & thread safety

- A program contains a **data race** if **two execution threads simultaneously access the same variable and at least one of these accesses is a write**
 - data races are highly non-deterministic, and a pain to debug!
- **thread-safety** = **the behaviour of a program consisting of several threads can be understood as an interleaving of those threads**
- In Java, the semantics of a program with data races is effectively undefined:
- **Moral of the story: Even “safe” programming languages can have very weird behaviour in presence of concurrency**

Java Bug

the method `getSigners` returns an alias to the private array `signers`. Because arrays in Java are always mutable, this allows untrusted code to obtain a reference to this array and then, change the content.

```
package java.lang;
public class Class {
    private String[] signers;

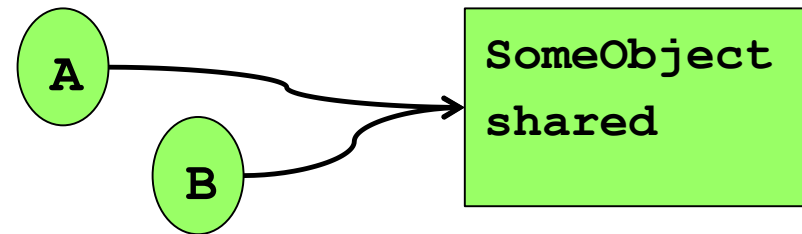
    /** Obtain list of signers of given class */
    public String[] getSigners() {
        return signers;
    }
    ...
}
```



The solution is for `getSigners` to return a copy of the array `signers` rather than an alias.

Threads and aliasing

Aliasing: two threads or objects
A and B both have a reference
to the same object shared



programmazione concorrente questa è una cosa complicatissima perché per gestire la sincronizzazione dei thread su strutture dati condivise, ho bisogno di avere dei meccanismi di sincronizzazione, infatti tutti i linguaggi di programmazione concorrente, ad esempio in Java hanno i metodi `synchronize`, che quello che fa è mettere un lock all'informazione dell'oggetto e vuol dire che ho accesso in mutua esclusione, ma sono costose. Quello che molto spesso succede nei linguaggi moderni è fare quelle che si chiamano strutture lock free che permettono di non avere lock ma hanno un meccanismo di gestione del recovery, quindi ci sono quelle ottimistiche e quelle che non sono ottimistiche, ad esempio in quelle ottimistiche sono strutture a cui riesco ad accedere e a questo punto poi in caso di errore devo riportare la struttura a un meccanismo consistente.

Aliasing (Issues)

- Concurrent (multi-threaded) context: data races
 - Locking objects (synchronized methods in Java) can help...
expensive & risk of deadlock & lock-free data structure
- Single-threaded context: dangling pointers
- Single-threaded context: broken assumptions
 - If A changes the shared object, this may break B's code, because B's assumptions about shared are broken

Però gli alias sono già complicati anche nel caso sempre sequenziale, il singolo thread, perché se io ho degli alias e non ha un controllo della memoria ragionevole e quello che potrei avere potrei avere riferimenti più lenti perché immaginate che ho un linguaggio di programmazione dove la gestione della memoria dinamica è data dal programmatore, non c'è un sistema allora questo punto immaginate di avere A e B che sono due alias della stessa struttura e immaginate che il programmatore decide: non userò più questa struttura quindi fa il free di A, ma questo punto B rimane appeso, cioè mi si è stato tolto sotto il tappeto l'accesso alla struttura, dato che ha fatto la free magari il gestore della memoria dinamica l'ha data a qualcun altro e a questo punto col meccanismo di accesso dovuto all' alias, io entrosupponendo di avere un dato sensibile a quello che mi interessava a me e invece trovo qualunque altra cosa.

```
1  public void f(char[] x){
2      if (x[0] != 'a') { throw new Exception(); }
3      // Can we assume that x[0] is the letter 'a' here?
4      // No!! Another concurrent execution thread could
5      //      change the content of x at any moment
```

If there is aliasing, another thread can modify the content of the array at any moment





Supponendo di avere questo metodo che prende un array di caratteri e a, questo punto di avere un controllo dice: se alla posizione zero ho un valore diverso dal carattere a sollevo eccezione. Posso assumere alla fine dell'esecuzione del punto dell'istruzione 2 che a questo punto, alla posizione zero ho la lettera a? Assolutamente no. Se sono in un meccanismo di programmazione concorrente e un meccanismo di alias che mi opera su questa struttura dipende da quello che hanno fatto gli altri, thread. Quindi questo cosa vuol dire che l'alias e la e il threading sono fatti in modo tale di avere delle assunzioni molto più complicate, però se io faccio l', alias con strutture dati immutabili, ad esempio le stringhe e dico: ma se la stringa alla posizione x ha un determinato carattere diverso da zero allora lancio un'eccezione a questo punto posso assumere che la posizione alla posizione zero della stringa X ho la lettera a, perché? Perché la struttura dati è immutabile quindi anche se ho un alias anche se ho thread che operano su questa struttura dati non possono essere mutati e questo è il motivo per cui nella programmazione concorrente nei linguaggi di programmazione concorrente vengono utilizzate strutture immutabili esattamente per affrontare questi aspetti

In a multi-threaded program, **aliasing of immutable data structures** are safer.

```
1  public void f(String x){  
2      if (x.charAt(0) != 'a') { throw new Exception(); }  
3      // We CAN assume that x[0] is the letter 'a' here?  
4      // Yes, as Java Strings are immutable  
5      ...
```



(Some) Temporary Conclusions

-  Some important programming language features:
 - type safety: the actual (runtime) type **matches** with the expected one
 - memory operations are compatible with the source-level abstraction (may forbid the use of un-initialized variables)
-  memory safety: no **unintended/invalid** memory access
-  thread safety: no **unintended** operations between threads
 - no *race conditions*, safe synchronization facilities, etc.
- no undefined behaviors (~ “time bombs”)
 -  no need for the compiler to detect or mitigate them !
 - aggressive optimizations, able to suppress security checks !

bbiamo detto che il linguaggio è caratterizzato da sintassi, regole semantiche, e run time. Però poi il run time è scritto anche con linguaggio di programmazione che non sono safety. Ad esempio la JVM è scritta in c, allora che cosa dobbiamo garantire? Dobbiamo garantire che nella parte del front end dei compilatori ci siano dei meccanismi di analisi statica che garantiscano delle buone proprietà di sicurezza e quindi, anche se questi meccanismi di analisi statica non effettuano delle ottimizzazioni aggressive, ma devono garantire delle buone proprietà di comportamento. L'esempio, CFI, che è esattamente quello che intendevo, cioè un meccanismo statico che serve per garantire poi a tempo di esecuzione dei buoni comportamenti. Perché a questo punto, staticamente uno riesce a determinare quali sono i cammini validi nel CFG. Quindi abbiamo un'idea statica dei comportamenti ammissibile. Vuol dire che c'è nel caso del progetto dei linguaggi di programmazione, la ricerca è molto focalizzata su questa parte qui, ovvero capire quali controlli statici si può fare. Poi c'è una parte di interoperabilità col codice a basso livello. Questa è una cosa molto complicata perché il codice a basso livello è scritto o in linguaggi di programmazione che sono unsafe oppure se è scritto in termini di codice della macchina sottostante, ora le macchine hanno n mila processori e sono core e quello che succede è che fanno dell'esecuzione speculativa, dove l'esecuzione speculativa significa che eseguono prima ad esempio di esser sicuri che effettivamente quell'istruzione è eseguita. Perché? Perché il tempo di esecuzione a livello del processore di un'istruzione è ordini inferiori rispetto al trasferimento dei dati dalla memoria e quindi dobbiamo garantire contro degli attacchi che operano proprio a livello di architettura e dobbiamo garantire questo aspetto. Più in generale deve garantire di avere delle macchine astratte dell'esecuzione del linguaggio con una TCB ben definita. Vuol dire delle componenti di sicurezza presenti a tempo di esecuzione.

- **Build secure compilation chains**

- **Compiler (static analyses)**

- control-flow integrity: preserves intended control-flow method call/returns (e.g, Java), valid paths in the control-flow graph

- **Runtime**

- Secure interoperability with lower-level code
 - abstract machine with **build-in security components**

La morale di questa seconda trincea di conclusioni è che non solo bisogna definire linguaggi di programmazione che hanno primitive di sicurezza, ma bisogna garantire tutta la chain della definizione del linguaggio, ovvero dalla sintassi all'esecuzione nel run time che tenga presente che la sicurezza è importante. In realtà bisogna anche garantire un altro aspetto che è un aspetto di formazione di awareness, cioè? Noi stiamo vedendo queste problematiche a un corso dalla magistrale. In realtà molte di queste problematiche ora dovrebbero essere portate nei corsi della triennale perché tutti i problemi che uno ha sull'aritmetica dei puntatori e sull'uso di riferimenti.

(Other) Temporary Conclusions ... hopefully

- Some prog. language features lead to unsecure code . . .
 - how do you choose a programming language ?
 - what about security ???
 - no “perfect language” yet . . . but security should become a (much) more important design concern . . .