What is a type?
- Is a set of values of an expression
  - Example: int x; \{ x ∈ Z | MIN_INT ≤ x ≤ MAX_INT \}
- Together with the operations that can be applied to these values
  - int x
  - x + 4 : int
  - x + "hello"

What is a type system?
- A set of types and the rules for creating and using them.

Classify/Filter programs
- If a program does not type checks, then it is not part of the language

Forbid undesirable behaviors
- Adding "dates" to "words"
- Unsafe handling of a pointer

Force the usage of user-defined interfaces
- Abstract types

The type checking follows the syntactic structure of the term to check
- There is an inference rule for each node in the syntax tree

\[ \Gamma \vdash E : \tau \]

Where \( \Gamma \vdash E : \tau \) is a type judgement
- given \( \Gamma = [x_1 : T_1, \ldots, x_n : T_n] \) a set of type assumptions, it can be derived from these typing rules that \( E \) is of type \( \tau \)

We say a term is well typed if \( \emptyset \vdash E : \tau' \)
Type-system rules example

- Terms:
  - $a$ is a variable
  - ..., $0$, $1$, ..., are integer expressions
  - $E + E$ is the addition
  - $T \, m(T \, a) \, \{\text{Body}\}$ is a method declaration
  - $m(E)$ is an invocation to method $m$

\[
\frac{A(a) = T}{A \vdash a : T \ (\text{var})}
\]

\[
\frac{n = \ldots, 0, 1, \ldots}{[] \vdash n : \text{int} \ (\text{const})}
\]

\[
\frac{A \vdash T \, m(T \, a) \, \ldots : T_1 \rightarrow T_2}{A \vdash E : T_1}
\]

\[
\frac{A \vdash \text{m}(E) : T_2 \ (\text{app})}{A \vdash E : T_2}\]

Type System rules example

\[
\frac{A(a) = T}{A \vdash a : T \ (\text{var})}
\]

\[
\frac{n = \ldots, 0, 1, \ldots}{[] \vdash n : \text{int} \ (\text{const})}
\]

\[
\frac{A \vdash T \, m(T \, a) \, \ldots : T_1 \rightarrow T_2}{A \vdash E : T_1}
\]

\[
\frac{A \vdash \text{m}(E) : T_2 \ (\text{app})}{A \vdash E : T_2}\]

We call $\text{erase}(e)$ to a function taking a “well-typed” $e$ that returns $e$ with no type-information.

- Given a non-typed term $e_u$: Is this the result of $\text{erase}(e)$ for some $e$? Which are the types of $e$?

This is the type inference problem. We have to find valid types instead of just checking their validity.
**Type Inference**

- Discover all type annotations such that the program typechecks.
  - Idea: Introduce unknowns to the rules and see if the rules can be solved consistently

```
{a:int} |- X a:int
| X 1:int
{a+A} |- X a+1:B
| X B
inc(a: A) {a+1:B} A B
(func)
```

```java
class C {
    D f;
    C() {
        if (bluemoon()) {
            f = new D();
        }
    }
    C(D x) {
        f = x;
    }
    void m() {
        f.q();
    }
}
```

**A richer type system**

- Idea: Use types to filter out programs with undesirable behavior
- Examples:
  - Potential runtime exceptions
    - Non-null types
  - Security/Protection
    - Reference immutability / ownership types
  - A certain protocol is not obeyed
    - Type states

```
class C {
    D f;
    C() {
        if (bluemoon()) {
            f = new D();
        }
    }
    C(D x) {
        f = x;
    }
    void m() {
        f.q();
    }
}
```

**Non-null types**

- Field f is non-null!
  - Now if a program typecheck, then it guarantees no null dereference.

```
class C {
    D f;
    C() {
        if (bluemoon()) {
            f = new D();
        }
    }
    C(D x) {
        f = x;
    }
    void m() {
        f.q();
    }
}
```
**Non-null Annotation**

- Extend the type system to:
  - Improve documentation
  - Record intention

- Usage:
  - Detect errors during source compilation
  - Detect errors sooner, possibly before dereference
  - Low the number of runtime exceptions
  - Optimization
  - Useful for other analysis

**Examples**

```java
T! t = new T(...); // create an object of type non-null
T n = t; // OK: nullable super type of non-null
...
if (n != null)
  t = n; // here n has type T!
t = (T!)n; // requires a cast (runtime check)
int x = t.f;
...
t.m(...); // requires t non-null
throw t; // requires t non-null
```

**Extensions**

- Types given a class $T$
  - Non-null: $T! (@nonNull T)$
  - Possibly-null: $T (Nullable T)$

- Use $T!$ for arguments, return, fields, variables ...

- Explicity type cast from $T$ to $T!$

- New type hierarchy

- Changes in semantic of constructors

**Problems**

- Component initialization (constructors, arrays)
  - Default reference initialization is to null!

- Constructors must enforce object invariants
  - Each non-null field must be initialized

- Do we grant access to partially initialized objects?
  - No: simpler, but more limited
    - No method invocation is allowed from the constructor!
  - Yes: what is the type for those objects?
Example

```csharp
class A {
    string name;
    public A(string s) {
        this.name = s;
        this.m(55);
    }
    virtual void m(int x) { ... }
}
```

OK: name is initialized before its use

Solution

- The “raw” type
  - `x : T` object partially initialized of type `T`
  - A constructor can only call methods accepting Raw data types.
- For raw objects, the rules reading and writing fields change:
  - Given `x : T`, if the field `x.f` has type `C`:
    - read `(t = x.f)` returns `t : C` (it might be null)
    - write `(x.f = b)` requires `b : C` (whatever we write, it might be non-null)

Example (cont.)

```csharp
class B : A {
    string path;
    public B(string p, string s) {
        base(s) {
            this.path = p;
        }
        override void m(int x) {
            ... a = this.path ...
        }
    }
}
```

m() is invoked from A's constructor, before B() initialized path!

Example

```csharp
class A {
    string name;
    public A(string s) {
        this.name = s;
        this.m(55);
    }
    virtual void m(int x) { ... }
}
```

```
class B : A {
    string path;
    public B(string p, string s) {
        base(s) {
            this.path = p;
        }
        override void m(int x) {
            ... a = this.path ...
        }
    }
}
```

Non-null
Null
⊥
Raw

Now `a` is Nullable!
So any dereference of `a` fails!
An implementation

- Spec# type system

A richer type system

- Idea: Use types to filter out programs with undesirable behavior
- Examples:
  - Potential runtime exceptions
    - Non-null types
  - Security/Protection
    - Reference immutability / ownership types
  - A certain protocol is not obeyed
    - Type states

Reference immutability

- Any problem with this program?

```csharp
class C {
    private int data;
    public int getData() {
        return data;
    }
}
int i = myClass.getData();
i++;
```

- And with this program?

```csharp
class C {
    private string data;
    public string getData() {
        return data;
    }
}

String s = myClass.getData();
s.Trim();
```
Reference immutability

- And with this one?

```java
class C {
    private List data;
    public List getData() {
        return data;
    }
}
List l = myClass.getData();
l.add();
```

- Mutation error: a side-effect leads to an undesired update.

Example

- A possible solution

```java
class C {
    private List data;
    public List getData() {
        return new List(data);
    }
}
List l = myClass.getData();
l.add();
```

- Is this what we want?
- Is always that simple?

- It does not seem to be a feasible solution

Enriching our type system

- We might indicate that a given reference is immutable

```java
class C {
    private List data;
    public @ReadOnly List getData() {
        return data;
    }
}
List l = myClass.getData();
l.add(); // compilation error!
```

Information Leak

- A security leak in JDK 1.1

```java
class Class {
    private Object[] signers;
    public Object[] getSigners() {
        return signers;
    }
}
```

- A possible solution

```java
class C {
    private List data;
    public List getData() {
        return new List(data);
    }
}
List l = myClass.getData();
l.add();
```

- Is this what we want?
- Is always that simple?

- It does not seem to be a feasible solution
We want to forbid changes to \( g \)

```java
public void m(Graph g) {
    ...
    g.addEdge(n1,n2);
}
```

Is this enough?

No! “final” only avoids a reference from being modified.

```java
public void m(final Graph g) {
    ...
    g.addEdge(n1,n2);
}
```

Is it enough now?

Yes! `ReadOnly` protects the reference

```java
public void m(ReadOnly Graph g) {
    ...
    g.addEdge(n1,n2); // compilation error
}
```

Object immutability: an object can not be modified by any reference

```java
Graph temp = new Graph();
// construct the graph
readonly Graph g = temp;
temp = null;
```

Reference immutability: independent control for each reference

Mutability

It defines if the (abstract) state of an object can be modified.

```java
class Date {
    int year;
}
/*mutable*/ Date d;
readonly Date rd;
d.year = 2005; // OK
rd.year = 2005; // Error
```

Abstract state mutation

Mutation: any change to the abstract state of the object.

- Abstract state: by default all fields. Some fields can be excluded. The abstract state is recursive over all reachable objects.

- Two control mechanisms:
  - Mutability
  - Assignability

Kinds of Immutability

- Object immutability: an object can not be modified by any reference

- Reference immutability: independent control for each reference
Mutability annotations

- They are applied on fields and variables
  - `readonly`: The referent abstract state can not be modified
  - `mutable`: The abstract state can be modified
    - Default for all local variables
  - `this-mutable`: (all for fields)
    - Mutability depends on container class
    - It can be modified if the container instance is mutable
    - It can’t be modified if the container class is read-only
      - `this.f` is mutable if `this` is mutable and `f` is this-mutable
      - `this.f` is readonly if `this` is readonly `y` `f` is this-mutable

Type system

- Each (mutable) type `T` has `readonly T` as super type
  - In other words, we can assign a mutable to a readonly reference, but not the other way around.

Mutability vs. Assignability

- **Mutable**
  - It is a part of the type
    - `readonly`
- **Assignable**
  - It’s not a part of the type
    - `final`

```java
final Date fd = null;
@readonly Date rd = null;

fd = new Date(); // Error: final
rd = null; // OK

Date d1 = fd; // OK
Date d2 = rd; // Error: wrong type
```

```java
class Account {
    @ReadOnly Customer owner;
    @Mutable RequestLog requests;
    //def:this-mutable

    ... Account a; //def:mutable
    @ReadOnly Account ra;
    a.owner.setName("Bob"); // Error
    ra.owner.setName("Bob"); // Error
}
```

<table>
<thead>
<tr>
<th>Mutability of ref.f</th>
<th>Declared mutability of f</th>
<th>Resolved mutability of ref</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>readonly</code></td>
<td><code>mutable</code></td>
<td><code>readonly</code></td>
</tr>
<tr>
<td><code>readonly</code></td>
<td><code>readonly</code></td>
<td><code>readonly</code></td>
</tr>
</tbody>
</table>
Mutability of `ref.f`

<table>
<thead>
<tr>
<th>Declared mutability of <code>f</code></th>
<th>Resolved mutability of <code>ref</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

Mutability of `this-mutable`

<table>
<thead>
<tr>
<th>Declared mutability of <code>f</code></th>
<th>Resolved mutability of <code>ref</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

Recap

<table>
<thead>
<tr>
<th>Declared mutability of <code>f</code></th>
<th>Resolved mutability of <code>ref</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>mutable</td>
<td>mutable</td>
</tr>
</tbody>
</table>

- `a.requests.add("checkBalance");` // OK
- `ra.requests.add("checkBalance");` // OK
- `a.balance.withdraw(1000);` // OK
- `ra.balance.withdraw(1000);` // Error

**this-mutable**: the mutability of `bal` depends on the mutability of this

**this-mutable**: the mutability of `bal` depends on the mutability of this

**a.balance.withdraw(1000);** // Error

**a.requests.add("checkBalance");** // OK
**ra.requests.add("checkBalance");** // OK
Assignability

<table>
<thead>
<tr>
<th>Declared assignability of f</th>
<th>Mutability (ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
<td>no-assignable</td>
</tr>
<tr>
<td>assignable</td>
<td>assignable</td>
</tr>
<tr>
<td>this-assignable</td>
<td>assignable</td>
</tr>
</tbody>
</table>

Class Bicycle

```java
class Bicycle {
    final int id;
    @Assignable int hashCode;
    int gear; //def:this-assignable
} Bicycles b; //def:mutable
@ReadOnly Bicycle rb;
```

- b.id = 5;
- rb.id = 5;
- b.hashCode = 5;
- rb.hashCode = 5;
- b.gear = 5;
- rb.gear = 5;

Mutability (ref)

<table>
<thead>
<tr>
<th>Declared assignability of f</th>
<th>Mutability (ref)</th>
</tr>
</thead>
<tbody>
<tr>
<td>final</td>
<td>no-assignable</td>
</tr>
<tr>
<td>assignable</td>
<td>assignable</td>
</tr>
<tr>
<td>this-assignable</td>
<td>assignable</td>
</tr>
</tbody>
</table>

Problematic example

```java
class Student {
    @Assignable GradeReport grades; //this-mutable
}

myMethod(@ReadOnly GradeReport rg ...) {
    Student s = new Student(); //mutable
    @ReadOnly Student rs = s;
    GradeReport g; //mutable
    rs.grades = rg;
    g = s.grades;
}
```

- A this-mutable reference from readonly should be readonly?

```
class Student {
    @Assignable GradeReport grades; //this-mutable
}

myMethod(@ReadOnly GradeReport rg ...) {
    Student s = new Student(); //mutable
    @ReadOnly Student rs = s; //Valid
    GradeReport g; //mutable
    rs.grades = rg; //readonly assigned to this-mutable
    g = s.grades; //now g has rg as mutable!
}
```

- A this-mutable reference from readonly should be readonly?
  - No! It might turn a readonly reference to a mutable reference without explicitly stating that.
Solution: Forbid a readonly reference from being copied to a this-mutable field.

```java
class Student {
    assignable /*this-mut*/ GradeReport grades;
}
```

This-mutable fields are:
- Read as `readonly GradeReport`
- Written as `mutable GradeReport`

```java
rs.grades = rg; // error! readonly (rg) assigned to a mutable (rs.grades)
```

Javari: reference immutability

- A `reference` is immutable if we can not use this reference to modify the object
  - Others references might modify it

- An extension of the type system to deal with updates through references.

- “Depth” immutability control
  - All the reachable state

Javari

- Javari is a backward-compatible extension of the Java language.

- The programmer can specify that a particular reference is read-only
  - Cannot be used to change the state of its referent

- Javari compile-time checker verifies this property.

Javari: Static typing

- It is checked at compilation time
- Type casting delegates checking at execution time

- Uses
  - Documentation checkeable by a computer
  - Prevent/detect errors
  - Useful information for other analyses
A richer type system

- Idea: Use types to filter out programs with undesirable behavior
- Examples:
  - Potential runtime exceptions
    - Non-null types
  - Security/Protection
    - Reference immutability / ownership types
  - A certain protocol is not obeyed
    - Type states

Encapsulation

- **Encapsulation**: Restrict access to object internal representation
  - The inner state of an object is hidden to the external objects
- **Goals**:
  - Independent from representation
  - Side-effects (preserve invariants)
  - Modular reasoning
    - Think of objects as components
    - Fundamental for reasoning on complex systems
    - And also for automatic analysis!
  - Security

Ownership types

- Types for Flexible Alias protection
- **Property**: Each object has an owner
  - Owners control access to objects
- Use **type-checking** to enforce this property on the programs

Ownership Types
• The first parameter is the owner
• <this> means internal representation object (also rep)
• <o> Owner passed as parameter
• <world> Default, means no restrictions
A richer type system

- Idea: Use types to filter out programs with undesirable behavior
- Examples:
  - Potential runtime exceptions
    - Non-null types
  - Security/Protection
    - Reference immutability / ownership types
  - A certain protocol is not obeyed
    - Type states

API protocols

- Increasing complexity of APIs
  - Dozens or hundreds of functions
  - Not always well documented
- Programs are more dependent on them
  - Framework APIs
  - Library APIs
  - Database connections APIs, etc.
- Problem: how to correctly use them?

Example: queue

Typestate: dynamic types

- A typestate is a dynamic version of a type

<table>
<thead>
<tr>
<th>Traditional Type</th>
<th>Typestate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates what actions are allowed on an instance.</td>
<td>Indicates what actions are allowed on an instance at a given instant</td>
</tr>
</tbody>
</table>

- Each state in a typestate prunes functionality
### Controlling access to an object

- `public static void f1(@Unique Object myObj) {`  // myObj is the only reference to the object
- `public static void f2(@Full Object myObj) {`  // we can modify the state of myObj (only "read-only" references can exist)
- `public static void f3(@Pure Object myObj) {`  // we can not modify the state of myObj (other references may modify it)
- `public static void f4(@Share Object myObj) {`  // we can modify myObj (other share references can also modify it)

Permissions can be modified:
- `@Unique => 1`  // `@Full & N`  // `@Pure`  // `@Share & M`  // `@Pure`  // `null`

### A queue protocol

- `@Full` (requires=“OPEN”, ensures=“CLOSED”)  void close()
- `@Full` (requires=“OPEN”, ensures=“OPEN”)  void enqueue(@Share Object o)
- `@Pure`  void is_closed()
- `@TrueIndicates (“CLOSED”)`  `@FalseIndicates (“OPEN”)`

### Analyzing Queue usage

```
final Blocking_queue queue = new Blocking_queue();
// OPEN
for( int i=0; i<5; i++ )
// OPEN
queue.enqueue("Object " + i);
// OPEN
queue.close();
// CLOSED
```

### Queue and multithreads...

```
final Blocking_queue queue = new Blocking_queue();
(new Thread() {
  @Override
  public void run() {
    while( !queue.is_closed() )
      System.out.println("Got object: " + queue.dequeue());
    System.exit();
  }
}).start();
```

Any problem?

Race condition
**Verifying Producer**

```java
final Blocking_queue queue = new Blocking_queue();
// @Unique(queue) OPEN
(...).start();
// @Full(queue) OPEN
for( int i=0;i<5;i++ )
    queue.enqueue("Object " + i);
// @Full(queue) OPEN
queue.close();
```

**Verifying Consumer**

```java
@Override
public void run() {
    // @Pure(queue)
    while( !queue.is_closed() )
        // @Pure(queue)
        System.out.println("Got object: “ +
        queue.dequeue());
    System.exit();
}
```

**Plural** (Aldrich et al.)

- Eclipse plugin
  - [http://code.google.com/p/pluralism/](http://code.google.com/p/pluralism/)
- Typechecking is done through dataflow
  - Modular: type annotations
- User is able to specify:
  - Access permissions (*aliasing control*)
  - Object abstract states (*typestates*)