Automated Testing & Verification

Richer type systems

Galeotti/Gorla/Rau
Saarland University

Types

- 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.

What are they good for?

- Classify/Filter programs
  - <u>If a program does not type checks, then it is not part of the language</u>
- Forbid undesirable behaviors
  - Adding "dates" to "words"
  - Unsafe handling of a pointer
- Force the usage of user-defined interfaces
  - Abstract types

Basic type checking

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

$$\frac{A \vdash E_1:int \quad A \vdash E_2:int}{A \vdash E_1+E_2:int} \text{(sum)}$$

- Where **A |- E: T** is a type judgement
  - given $A=[x_1:T_1,\dots,x_n:T_n]$ a set of type assumptions, it can be derived from these typing rules that **E** is of type **T**
- We say a term is **well typed** if [] |- E:T'

Type-system rules example

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

$$\frac{A(a) = T}{A \vdash a : T}\ (var)$$

$$\frac{A \vdash T2\ m(T1\ a)\{\ldots\} : T1 \rightarrow T2 \quad A \vdash E : T1}{A \vdash m(E) : T2}\ (app)$$

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

$$\frac{A \vdash E1 : int \quad A \vdash E2 : int}{A \vdash E1 + E2 : int}\ (sum)$$

Type-system rules example

$$\frac{A(a) = T}{A \vdash a : T}\ (var)$$

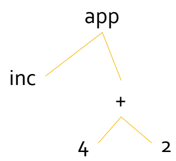
$$\frac{A \vdash T2\ m(T1\ a)\{\ldots\} : T1 \rightarrow T2 \quad A \vdash E : T1}{A \vdash m(E) : T2}\ (app)$$

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

$$\frac{A \vdash E1 : int \quad A \vdash E2 : int}{A \vdash E1 + E2 : int}\ (sum)$$

```
int m()
{
  return inc(4+2);
}

int inc(a: int)
{
  return a+1;
}
```

$$\frac{\dfrac{\vdash 4 : int\ (const) \qquad \vdash 2 : int\ (const)}{\vdash 4+2 : int\ (sum)} \qquad \vdash int\ inc(a : int)\{\ldots\} : int \rightarrow int}{\vdash inc(4+2) : int\ \ (app)}\ ✓$$



Type-system rules example

$$\frac{A(a) = T}{A \vdash a : T}\ (var)$$

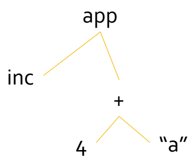
$$\frac{A \vdash T2\ m(T1\ a)\{\ldots\} : T1 \rightarrow T2 \quad A \vdash E : T1}{A \vdash m(E) : T2}\ (app)$$

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

$$\frac{A \vdash E1 : int \quad A \vdash E2 : int}{A \vdash E1 + E2 : int}\ (sum)$$

```
int m()
{
  return inc(4+"a");
}

int inc(a: int)
{
  return a+1;
}
```

$$\frac{\vdash 4 : int\ (const) \qquad \vdash \text{"a"} : ?}{\vdash 4 + \text{"a"} : Error!\ (sum)}$$



Type Inference

- We call ***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 *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 consistely

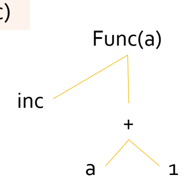
```
int m()
{
  return inc(4+2);
}

? inc(a: ?)
{
   return a+1;
```

$$\frac{A \vdash E_1{:}int \quad A \vdash E_2{:}int}{A \vdash E_1{+}E_2{:}int} \text{ (sum)}$$

$$\frac{A, x{:}T_1 \vdash E : T_2}{A \vdash T_2 \; func(x{:}tT_1) \{E\} : T_1 \rightarrow T_2} \text{ (func)}$$

$$\frac{\dfrac{\{a{:}int\} \vdash a{:}int \qquad \vdash 1{:}int}{\{a{:}A\} \vdash a{+}1{:}B \quad {=}int \text{ (sum)}}}{\vdash \mathbf{B} \; inc(a{:}\mathbf{A}) \{a{+}1;\}{:} \; \mathbf{A \rightarrow B} \quad \text{(func)}}$$

A = int
B = int



|- **int** inc(a:**int**) {a+1; }

A richer type system

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

Non-null types

```
class C {
  D f;

  C() {
    if (bluemoon()) {
      f = new D();
    }
  }

  C(D x) {
    f = x;
  }

  void m() {
        f.q();
  }
}
```

Was *f* really initialized?

- It depends on a condition

Does *f* point to an object?

- It depends on the value x

Can field *f* be null?

Non-null types

```
class C {
  D! f;

  C() {
    if (bluemoon()) {
      f = new D();
    }
  }

  C(D x) {
    f = x;
  }

  void m() {
    f.q();
  }
}
```

Field *f* is non-null !

Now the errors are explicit

1. Paths do not initialize *f*
2. *x* can **not** be assigned to *f*

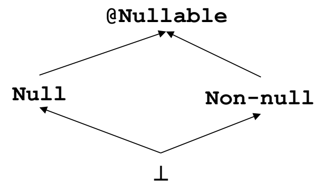
Now if a program typecheck, then it guarantees no null dereference.

Non-null Annotation

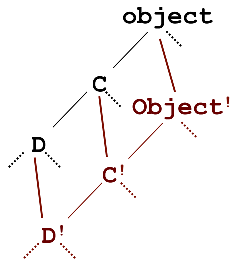
- Extend the type system to:
  - Improve documentation
  - Record intention
- Usage:
  - Detect errors during source compilation
  - Detect errors sooner, possibly before dereference
  - Low the number of runtime exceptions
  - Optimization
  - Useful for other analysis

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



Examples

```
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;          // requires t non-null
...
t.m(...);             // requires t non-null
...
throw t;              // requires t non-null
```

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

```
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

Example (cont.)

```
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**!

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

Solution

- The "**raw**" type
  - $x : T^{raw!}$ 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 $x : T^{raw!}$. 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

```
class A {
    string! name;
    public A(string! s) {
        this.name = s;
        this.m(55);
    }

    [Raw]
    virtual void m(int x) { … }
}
```

```
class B : A {
    string! path;
    public B(string! p, string! s)
        : base(s) { this.path = p; }

    [Raw]
    override void m(int x) {
        … a = this.path …
    }
}
```

Now *a* is @Nullable
So any dereference of *a* fails!

An implementation

- Spec# type system
  - http://research.microsoft.com/en-us/projects/specsharp/

A richer type system

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

Reference immutability

- Any problem with this program?

```
class C {
  private int data;

  public int getData() {
    return data;
  }
}

int i = myClass.getData();
i++;
```

Reference immutability

- And with this program?

```
class C {
  private string data;

  public string getData() {
    return data;
  }
}

String s = myClass.getData();
s.trim();
```

Reference immutability

- And with this one?

```
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.

Information Leak

- A security leak in JDK 1.1

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

Example

- A possible solution

```
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

```
class C {
  private List data;

  public @ReadOnly List getData() {
    return data;
  }
}

List l = myClass.getData();
l.add(); // compilation error!
```

Protecting arguments

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

- We want to **forbid** changes to *g*

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

- Is this enough?
- No! "final" only avoids a reference from being **modified**.

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

- Is it enough now?
- Yes! Readonly protects the reference

Kinds of Immutability

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

```
Graph temp = new Graph();

// construct the graph

readonly Graph g = temp;
temp = null;
```

- **Reference immutability**: **independent** control for each reference

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

Mutability

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

```
class Date {
  int year;
}

/*mutable*/ Date  d;
readonly    Date rd;
 d.year = 2005; // OK
rd.year = 2005; // Error
```

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

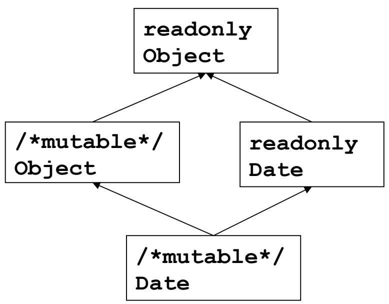
Mutability vs. Assignability

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

```
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
```

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.



`readonly`

```
class Account {
  @ReadOnly Customer owner;
  @Mutable RequestLog requests;
  Balance bal; //def:this-mutable
}
...
Account a; //def:mutable
@ReadOnly Account ra;
 a.owner.setName("Bob"); // Error
ra.owner.setName("Bob"); // Error
```

Mutability of `ref.f`		
Declared mutability of `f`	Resolved mutability of `ref`	
	mutable	readonly
readonly	readonly	readonly

mutable

```
class Account {
  @ReadOnly Customer owner;
  @Mutable RequestLog requests;
  Balance bal; //def:this-mutable
}
...
Account a; //def:mutable
@ReadOnly Account ra;
a.requests.add("checkBalance"); // OK
ra.requests.add("checkBalance"); // OK
```

mutable excludes **requests** from the abstract state of the object.

Mutability of ref.f		
Declared mutability of f	Resolved mutability of ref	
	mutable	readonly
readonly	readonly	readonly
mutable	mutable	mutable

this-mutable

```
class Account {
  @ReadOnly Customer owner;
  @Mutable RequestLog requests;
  Balance bal; //def:this-mutable
}
...
Account a; //def:mutable
@ReadOnly Account ra;
a.balance.withdraw(1000); // OK
```

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

Mutability of ref.f		
Declared mutability of f	Resolved mutability of ref	
	mutable	readonly
readonly	readonly	readonly
mutable	mutable	mutable
this-mutable	mutable	<? readonly>

this-mutable

```
class Account {
  @ReadOnly Customer owner;
  @Mutable RequestLog requests;
  Balance bal; //def:this-mutable
}
...
Account a; //def:mutable
@ReadOnly Account ra;
a.balance.withdraw(1000); // OK
ra.balance.withdraw(1000); // Error
```

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

Mutability of ref.f		
Declared mutability of f	Resolved mutability of ref	
	mutable	readonly
readonly	readonly	readonly
mutable	mutable	mutable
this-mutable	mutable	<? readonly>

Recap

```
class Account {
  @ReadOnly Customer owner;
  @Mutable RequestLog requests;
  Balance bal; //def:this-mutable
}
...
Account a; //def:mutable
@ReadOnly Account ra;
```

Mutability of ref.f		
Declared mutability of f	Resolved mutability of ref	
	mutable	readonly
readonly	readonly	readonly
mutable	mutable	mutable
this-mutable	mutable	<? readonly>

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

Assignability

Assignability (ref.f)		
Declared assignability of f	Mutability (ref)	
	mutable	readonly
final	no-assignable	no-assignable
assignable	assignable	assignable
this-assignable	assignable	no-assignable

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

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

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

Assignability

Assignability (ref.f)		
Declared assignability of f	Mutability (ref)	
	mutable	readonly
final	no-assignable	no-assignable
assignable	assignable	assignable
this-assignable	assignable	no-assignable

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

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

- **b.gear = 5;**
- **rb.gear = 5;**

Problematic example

```
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?

Problematic example

```
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.

This-mutables refs from readonly refs

- Solution: Forbid a readonly reference from being copied to a this-mutable field.

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

This-mutable fields are:

- Read as `readonly GradeReport`
- Written as `mutable GradeReport`

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

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 t change the state of its referent
- Javari compile-time checker verifies this property.

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

- **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 inmmutability / ownership types
  - A certain protocol is not obeyed
    - Type states

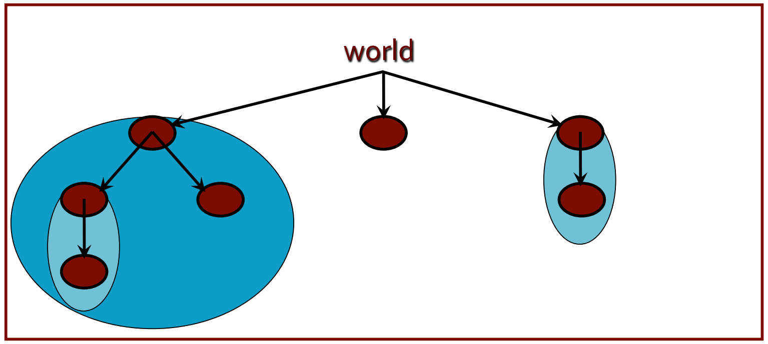
Encapsulation

- **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
    - Foundamental for reasoning on complex systems
    - And also for automatic analysis!
  - Security

Ownership types

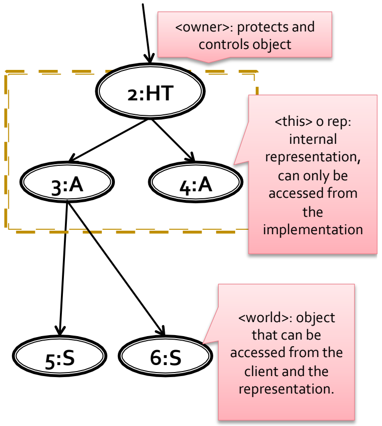
- Types for Flexible Alias protection
- **Property**: Each object has an owner
  - Owners control access to objects
- Use **type-checking** to enforce this property on the programs

Ownership Types



Types for alias protection

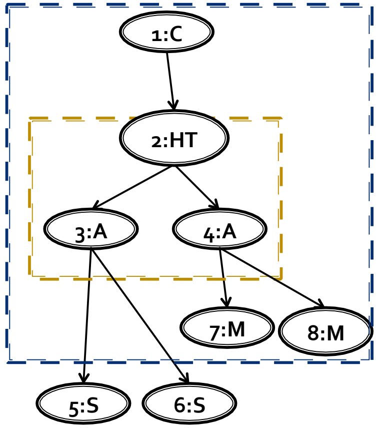
```
class HT<o, k, i> {
  private Array<this,k> keys;
  private Array<this,i> items;
  public void put(H<k> key, O<i> value);
  public O<i> get(H<k> key);
}
class Student {
}
```



- The first parameter is the owner
- <this> means internal representation object (also rep)
- <o> Owner passed as parameter
- <world> Default, means no restrictions

Types for alias protection

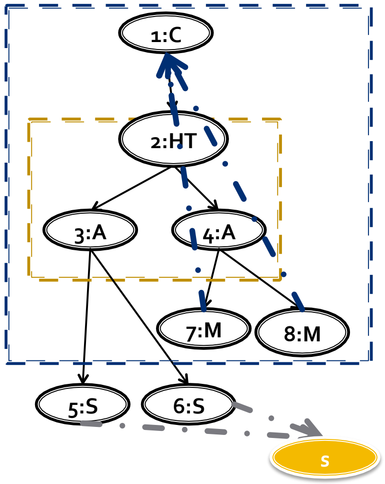
```
class HT<o, k, i> {
  private Array<this,k> keys;
  private Array<this,i> items;
  public void put(H<k> key, O<i> value);
  public O<i> get(H<k> key);
}
class Student {
}
class Mark {
}

class Course<s> {
   HT<this,s,this> marks;
}
```



- The first parameter is the owner
- <this> means internal representation object (also rep)
- <o> Owner passed as parameter
- <world> Default, means no restrictions

Types for alias protection

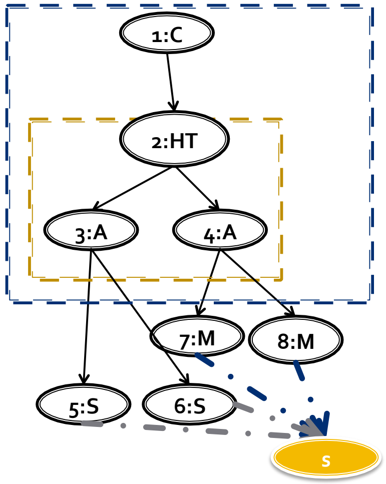
```
class HT<o, k, i> {
  private Array<this,k> keys;
  private Array<this,i> items;
  public void put(H<k> key, O<i> value);
  public O<i> get(H<k> key);
}
class Student {
}
class Mark {
}

class Course<s> {
   HT<this,s,this> marks;
}
```



- The first parameter is the owner
- <this> means internal representation object (also rep)
- <o> Owner passed as parameter
- <world> Default, means no restrictions

Types for alias protection

```
class HT<o, k, i> {
  private Array<this,k> keys;
  private Array<this,i> items;
  public void put(H<k> key, O<i> value);
  public O<i> get(H<k> key);
}
class Student {
}
class Mark {
}

class Course<s> {
   HT<this,s,s> marks;
}
```



- The first parameter is the owner
- <this> means internal representation object (also rep)
- <o> Owner passed as parameter
- <world> Default, means no restrictions

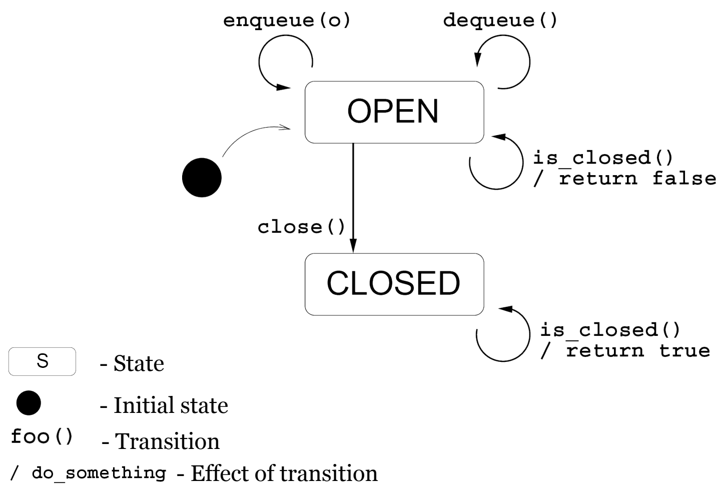
A richer type system

- Idea: Use types to filter out programs with undesirable behavior
- Examples:
  - Potential runtime exceptions
    - Non-null types
  - Security/Protection
    - Reference inmmutability / 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 conections APIs, etc.
- **Problem**: how to correctly use them?

Example: queue

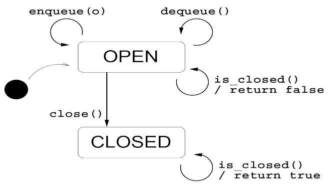


Typestate: dynamic types

- A typestate is a dynamic version of a type

Traditional Type	Typestate
Indicates what actions are allowed on an instance.	Indicates what actions are allowed on an instance **at a given instant**

- Each state in a typestate prunes functionality



Controlling access to an object

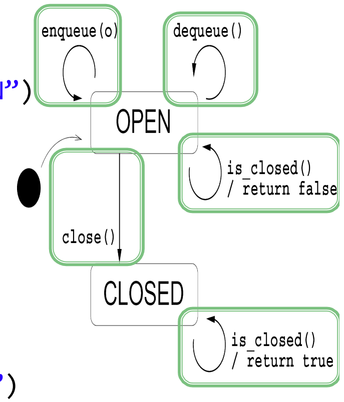
```
public static void f1(@Unique Object myObj) {
    // myObj is the only reference to the object
}
public static void f2(@Full Object myObj) {
    // we can modify the state of myObj
    // (only “read-only” references can exist)
}
public static void f3(@Pure Object myObj) {
    // we can not modify 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)
}
```

Permisions can be modified

- `@Unique` => 1×`@Full` & N×`@Pure`
- `@Full` => N×`@Imm`
- `@Full` => N×`@Share` & M×`@Pure`
- …

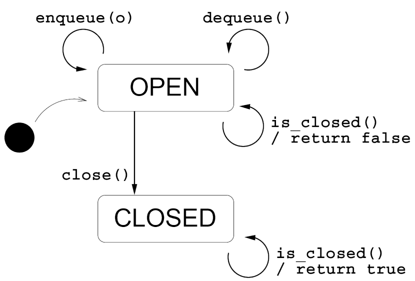
A queue protocol

```
@Full(requires=“OPEN”, ensures=“CLOSED”)
void close()

@Full(requires=“OPEN”, ensures=“OPEN”)
void enqueue(@Share Object o)

@Pure
@TrueIndicates(“CLOSED”)
@FalseIndicates(“OPEN”)
boolean is_closed()

@Pure(requires=“OPEN”,ensures=“OPEN”)
Object dequeue()
```



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();

for( int i=0;i<5;i++ )
  queue.enqueue(”Object" + i);

queue.close();
```

Any problem?

Race condition

Verifying Producer

```
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();
```

@Full(queue)
CLOSED

Satisfies the precondition

```
@Pure
boolean is_closed()
@Full(requires="OPEN",
@Full(requires="OPEN", ensures="CLOSED")
void close()
```

Verifying Consumer

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

```
@Pure
@Pure(requires="OPEN",ensures="OPEN")
Object dequeue()
boolean is_closed()
```

@Pure(queue)

Precondition is not satisfied
ERROR

Plural (Aldrich et al.)

- Eclipse plugin
  - 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)*