Concepts (Principles) of Object-Oriented Programming Study Guide

Karl Marx

December 16, 2023

Object-oriented programming is an exceptionally bad idea which could only have originated in California.

Edsger Dijkstra

Contents

1 Introduction

This study guide takes its information from the Concepts of Object-Oriented Programming course website.

Objects have:

- State
- Location in memory
- Behavior

Core concepts:

- Object model
- Interfaces and encapsulation
- Classification and Polymorphism

Language concepts:

- Classes
- Inheritance
- Subtyping
- Dynamic binding

Methods are invoked on a receiver object:

Example 1.1 (receivers)**.** Imagine there is a dog class and a human class. Here the dog instance is the receiver.

```
Dog dog = new Dog();
Human human = new Human();
dog.peeOn(human);
```

2 Typing

2.1 Types

> *Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.*
> *A: The definitions are almost always wrong.*
>
> Benjamin C. Pierce

Definition 2.1 (Nominal Types)**.** Type equivalence based on the type's name.

Definition 2.2 (Structural Types)**.** Type equivalence based on availability of methods and fields.

$$\boxed{S <: T} \qquad \frac{S \text{ extends } T}{S <: T} \qquad \frac{S \text{ implements } T}{S <: T} \qquad \frac{}{T <: \top} \qquad \frac{}{T <: T} \qquad \frac{R <: S \qquad S <: T}{R <: T}$$

Figure 1: Basic subtyping rules

Definition 2.3 (Static Typing)**.** Types are determined at compile time, may be inferred or declared explicitly.

Definition 2.4 (Dynamic Typing)**.** Types are determined at run time.

Sometimes *run time checks* are needed even in so-called "Static" languages.

Example 2.1 (Downcasts)**.** Downcasts could fail so they need a run time check.

```
String foo(Object o){
    String s = (String) o;
    return s;
}
```

2.2 Syntactic Subtyping

Definition 2.5 (Substitution Principle)**.** Subtype objects may be used wherever supertype objects are expected.

In languages such as Java and C# we say a type is a subtype of another if it satisfies the rules 1.

Definition 2.6 (Syntactic Subtyping)**.** Subtype receivers S understand the messages of supertype receivers T.

$$values(S) \subseteq values(T) \tag{1}$$

S has a wider interface than T.

$$interface(T) \subseteq interface(S) \tag{2}$$

We then say that $S <: T$.

Definition 2.7 (Nominal Subtyping)**.** $S <: T$ when S explicitly declares T as a supertype.

Definition 2.8 (Structural Subtyping)**.** $S <: T$ when S has the same available methods and fields as T.

For $S <: T$, then any field or method in S that overrides one in T must have at least the same visibility.

Example 2.2 (Java Visibility)**.**

$$\texttt{public} <: \texttt{protected} <: \texttt{default} <: \texttt{private} \tag{3}$$

Definition 2.9 (Function Subtyping)**.** Recall from lambda calculi with subtyping that parameter types should be *contravariant* and that return types should be *covariant*:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

In OO this applies to methods.

Example 2.3 (Why contravariant parameters?)**.** Consider the following:

```
class Animal {}

class Cat <: Animal {
    public void squish() { ... }
}

class Human {
    public void pet(Animal a) {}
}

class CatLover <: Human {
    public void pet(Cat c) {
        c.squish();
    }
}
```

We could have client code:

```
void client(Human h) {
    h.pet(new Animal())
}
```

That is called as:

```
client(new CatLover());
```

This would invoke `CatLover`'s pet method, which will call `squish` on an `Animal`. However we don't know if `Animal` in general may be squished? This message will not be understood.

In pretty-much all OO languages that anyone cares about return types are covariant but parameter types are invariant because this would make *overloading* more troublesome.

Definition 2.10 (Reference Subtyping)**.** Recall from lambda calculi with subtyping that mutable reference types must be invariant:

$$\frac{S <: T \qquad T <: S}{\texttt{ref}\ S <: \texttt{ref}\ T}$$

In OO this applies to class's mutable fields.

Example 2.4 (Why invariant fields?)**.** We can think of a mutable field has having a `read` and a `write` method.

```
class Super {
    T field;

    T read() {
        return this.field;
    }

    void write(T t) {
        this.field = t;
    }
}

class Sub <: Super {
    S field;

    S read() {
```

```
        return this.field;
    }

    void write(S s) {
        this.field;
    }
}
```

From doing function subtyping on `read` we get that $S <: T$. From doing function subtyping on `write` we get that $T <: S$.

Arrays in java are covariant because Java is stupid. So *writes* to arrays require a run time check.

Example 2.5 (covariant arrays in Java)**.** Consider the following:

```
1 String[] strs = {"hello", "world"};
2 Object[] objs = strs;
3 for (Object obj : objs) {
4     System.out.println(obj);
5 }
6 System.out.println()
7 objs[0] = new Object();
```

This fails at line 7. There is a run time check that throws a `java.lang.ArrayStoreException` for array writes.

Example 2.6 (contravariant arrays?)**.** What if Java had contravariant arrays? Consider:

```
1 Object[] objs = [new Object(), new Object()];
2 String strs = objs;
3 strs[0] = "hello";
4 for (String str : strs) {
5     System.out.println("bro" + str);
6 }
```

Here this would require a run time check at line 5 when we read from the array.

In general we want covariant for reads, or types in *positive* positions whereas we want contravariant for writes or types in *negative* positions. If what should be an invariant relation is treated as covariant, then we need a run time check for *writes*. If what should be an invariant relation is treated a contravariant, then we need a run time check for *reads*.

Example 2.7 (Double pointers in C++)**.** How should subtyping relate to pointers, and in particular double pointers? Consider the following, which assigns a value of `X` to what is passed into `init`.

```
class SuperX {};

class X : public SuperX {
    public: int a;
};

class SubX : public X {
    public: int b;
};

class Initializer {
    public:
        void init(X** x) {
            *x = new X();
        }
};
```

For example we could use an `Initializer` to assign a non-`null` value to a field of type `X*`.

```
class Value {
    private: X* x = nullptr;

    public: Value(Initializer* i) {
        i->init(&x); // The initializer object will set the value of x
    }
};
```

Could we pass in values of type `SuperX` or `SubX` to `init`. May we have either of the following rules?

$$\frac{S <: T}{S** <: T**}\ \textsc{CovarDoublePtr} \qquad\qquad \frac{S <: T}{T** <: S**}\ \textsc{ContraDoublePtr}$$

Let's test CovarDoublePtr. Consider if we modify `Value` as

```
class Value {
    private: SubX* x = nullptr;

    public: Value(Initializer* i) {
        i->init(&x); // The initializer object will set the value of x
        cout << x.b;
    }
};
```

This will immediately fail when we try to perform `i->init(x)`. The method will attempt to assign a `X*` to a `SubX*`. This write fails because $X \not<: SubX$. This is equivalent to allowing:

```
class Initializer {
    public:
        void init(SubX** x) {
            *x = new X(); // obviously wrong
        }
};
```

Let's instead test ContraDoublePtr. Consider if we modify both `Initializer` and `Value` as

```
class Initializer {
    public:
        void init(X** x) {
            cout << x.a;
            *x = new SubX();
        }
};

class Value {
    private: SuperX* x = new SuperX();

    public: Value(Initializer* i) {
        i->init(&x); // The initializer object will set the value of x
    }
};
```

When `i->init(x)` is invoked, `init` will attempt to read field a from what it thinks is an `X`. But because it is actually a `SuperX` it does not have an a. Thus this read fails. Thus the only safe rule for double pointers is an one.

$$\frac{S <: T \qquad T <: S}{S** <: T**}\ \textsc{InvarDoublePtr}$$

Example 2.8 (Read only types)**.** Types with values that may only be read from generally follow *covariance*. Basic products (tuples) and sums (left or right) have simple rules.

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 \times S_2 <: T_1 \times T_2}\ \textsc{ProductSubtyping} \qquad\qquad \frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 + S_2 <: T_1 + T_2}\ \textsc{SumSubtyping}$$

Record types (tagged products) and variant types (tagged sums) have a similar *depth* subtyping rules.

$$\frac{S_1 <: T_1 \qquad ... \qquad S_n <: T_n}{\{x_1 : S_1; ...; x_n : S_n\} <: \{x_1 : T_1; ...; x_n : T_n\}}\ \textsc{RecordDepth}$$

$$\frac{S_1 <: T_1 \qquad ... \qquad S_n <: T_n}{\{x_1 : S_1 | ... | x_n : S_n\} <: \{x_1 : T_1 | ... | x_n : T_n\}}\ \textsc{VariantDepth}$$

However records and variants have different *width* subtyping rules. A record type may subtype another if it has *at least as many* members. This is analogous to subclasses having more fields than superclasses. A variant type may subtype another if it has *at most as many* cases. If the opposite were allowed it would violate the Substitution Principle, as not all cases could be covered by code expecting the supertype.

$$\frac{}{x_1 : T_1; ...; x_n : T_n; ...; x_m : T_m <: x_1 : T_1; ...; x_n : T_n}\ \textsc{RecordWidth}$$

$$\frac{}{\{x_1 : T_1 | ... | x_n : T_n\} <: \{x_1 : T_1 | ... | x_n : T_n | ... | x_m : T_m\}}\ \textsc{VariantDepth}$$

Example 2.9 (Union types)**.** Values of union types may take the form of any of its constituent types. These are different from variant types because they are not tagged. A union type may subtype another type if all of the constituent types are a subtype of some type in the supertype (How do I say this without "type" appearing so many times, suggestions welcome).

$$\frac{\forall i, \exists j, S_i <: T_j}{\sum_{i}^{n} S_i <: \sum_{j}^{m} T_j}\ \textsc{UnionSubtype}$$

When reasoning on paper it is easy to make mistakes when a given union type has several constituent types. When determining if a subtype relationship hold between two union types, it is sufficient to reduce each to the common super types among them. For example, observe that given $B <: A$, $C <: B$ and we have some type $A \mid\mid B \mid\mid C \mid\mid D$, where D is unrelated to the others, we can reduce $A \mid\mid B \mid\mid C \mid\mid D$ to $A \mid\mid D$.

Example 2.10 (Dynamic types are screwy)**.** If we know that some term t has type T, $t : T$, in a normal language without subtyping we can make assumptions about the form t takes. However in a setting with subtyping, knowing $t : T$ only tells us that there's some S that t really is, where $S <: T$. Consider a language that has a `MyType` to represent the type of a term. We may define:

```
class Point {
    int x,y;

    boolean equals(MyType other) {
        return x == other.x && y == other.y;
    }
}

class ColorPoint extends Point {
    int color;
    override boolean equals(MyType other) {
        return super.equals(other) && color == other.color;
    }
}
```

In these definitions we require that Dynamic type of the parameter of `equals` is a subtype of the dynamic type of `this`. This seems innocent enough. However this restriction is very screwy when you consider that some subclass of `Point` or `ColorPoint` may inherit `equals` and call it. In this setting we have no idea of the lower bound on `this`. Thus all of the following calls are bad:

```
Point p       = ...;
ColorPoint cp1 = ...;
ColorPoint cp2 = ...;
p.equals(cp1)   // Bad, p may be some other subtype of Point not related to ColorPoint
p.equals(cp2)   // Bad, p may be some other subtype of Point not related to ColorPoint
cp1.equals(p)   // Bad, p could be Point and cp1 could be ColorPoint
cp2.equals(cp1) // Bad, cp2 could be some other subtype of ColorPoint
cp1.equals(cp2) // Bad, cp1 could be some other subtype of ColorPoint
```

What if we make `ColorPoint final`; that is no class may `extend ColorPoint`.

```
final class ColorPoint extends Point {
    int color;
    override boolean equals(MyType other) {
        return super.equals(other) && color == other.color;
    }
}
```

That helps us with the last two calls.

```
Point p       = ...;
ColorPoint cp1 = ...;
ColorPoint cp2 = ...;
p.equals(cp1)   // Bad, p may be some other subtype of Point not related to ColorPoint
p.equals(cp2)   // Bad, p may be some other subtype of Point not related to ColorPoint
cp1.equals(p)   // Bad, p could be Point and cp1 could be ColorPoint
cp2.equals(cp1) // Okay, both must be ColorPoint
cp1.equals(cp2) // Okay, both must be ColorPoint
```

Instead what if we could guarantee that a term has *exactly* its declared type. Say the syntax is `@T t` where t dynamically does not have a subtype of T. We could say the following

```
@Point p       = ...;
@ColorPoint cp1 = ...;
ColorPoint cp2  = ...;
p.equals(cp1)   // Okay, p must be Point
p.equals(cp2)   // Okay, p must be Point
cp1.equals(p)   // Bad, p is Point and cp1 is ColorPoint
cp2.equals(cp1) // Bad, cp2 could be some other subtype of ColorPoint
cp1.equals(cp2) // Okay, cp1 must be ColorPoint
```

2.3 Behavioral Subtyping

Definition 2.11 (Semantic Classification of Subtyping)**.**

$$S <: T \to behaviors(T) \subseteq behaviors(S) \tag{4}$$

We would like to constrain how overriding method's pre- and postconditions may be logically related.

Definition 2.12 (Overriding Preconditions)**.** Overriding methods may *weaken* preconditions.

$$S <: T \to \forall\ S_m\ override\ T_m \to pre(T_m) \to pre(S_m) \tag{5}$$

Definition 2.13 (Overrding Postconditions)**.** Overriding methods may *strengthen* postconditions.

$$S <: T \to \forall\ S_m\ override\ T_m \to post(S_m) \to post(T_m) \tag{6}$$

We may allow a stronger result:

$$S <: T \to \forall\ S_m\ override\ T_m \to old(pre(T_m)) \to post(S_m) \to post(T_m) \tag{7}$$

These rules for pre- and postconditions are reminiscent of the *rule of consequence* from Hoare Logic:

$$\frac{P \to P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \to Q}{\{P\}\ c\ \{Q\}}$$

We may think of P as the precondition of the supertype, Q as the postcondition of the supertype, P' as the precondition of the subtype, and Q' as the postcondition of the subtype.

Definition 2.14 (Overriding Invariants)**.** Subtypes may have stronger class invariants for fields:

$$S <: T \to inv(S) \to inv(T) \tag{8}$$

Definition 2.15 (History Constraints)**.** Properties of class fields that describe how objects evolve over time. Such a relation on a class's field must be *reflexive* and *transitive*.

$$constraint(T_f) := old(T_f) \sqsubseteq T_f \tag{9}$$

where $\sqsubseteq$ is reflexive and transitive. Relations in history constraints may be represented as pure boolean functions.

Example 2.11 (History Constraint)**.** Consider a stopwatch class that has the time.

```
class Stopwatch {
    // constraint: old(time) <= time
    int time;

    void tick() {
        this.time++;
    }
}
```

Definition 2.16 (Overriding History Constraints)**.** Subtypes may *strengthen* history constraints for fields.

$$S <: T \to \forall\ S_f\ override\ T_f \to constraint(S_f) \to constraint(T_f) \tag{10}$$

Example 2.12 (Unique key generation)**.** We may specify a class that generates unique keys as an incrementing counter, that is keys it has not generated before, as follows

```
class IncCounter {
    // constraint old(key) <= key
    int key;

    IncCounter (int k) {
        this.key = k;
    }

    // ensures key = old(key) + 1 /\ result = old(key)
    int generate () {
        return this.key++;
    }
}
```

The conjunction of the history constraint and postcondition of `generate` ensure that it actually generates new keys. Of course we are assuming no overflow. We could do something similar with a decrementing counter.

```
class DecCounter {
    // constraint key <= old(key)
    int key;

    DecCounter (int k) {
        this.key = k;
    }

    // ensures key + 1 = old(key) /\ result = old(key)
    int generate () {
        return this.key--;
    }
}
```

Each is acceptable as a unique key generator, but neither is a behavioral subtype of the other. We may unify each with a common abstract class. A first attempt at generalizing some transitive relation to be used for uniqueness and a reflexive closure of that relation is as follows.

```
abstract class GenerateUniqueKey {
    // constraint reflTransRel(old(key),key)
    int key;

    // Some transitive relation.
    boolean transRel(int, int);

    // The reflexive closure of the transitive relation.
    boolean reflTransRel(int x, int y) {
        return transRel(x, y) || x == y
    }

    // ensures transRel(old(key), key) /\ result = old(key)
    int generate ();
}

class IncCounter extends GenerateUniqueKey {
    // constraint old(key) <= key
    int key;

    IncCounter (int k) {
        this.key = k;
    }

    boolean transRel(int x, int y) {
        return x < y
    }

    // ensures key = old(key) + 1 /\ result = old(key)
    int generate () {
        return this.key++;
    }
}
```

```
class DecCounter extends GenerateUniqueKey {
    // constraint key <= old(key)
    int key;

    DecCounter (int k) {
        this.key = k;
    }

    boolean transRel(int x, int y) {
        return y < x
    }

    // ensures key + 1 = old(key) /\ result = old(key)
    int generate () {
        return this.key++;
    }
}
```

A more precise version is as follows.

```
abstract class GenerateUniqueKey {
     // constraint forall z, old(used(z)) -> used(z)

     abstract boolean used (int);

     // ensures ~ old(used(result)) /\ used(result)
     int generate ();
}

class IncCounter extends GenerateUniqueKey {
    // constraint old(key) <= key
    int key;

    IncCounter (int k) {
        this.key = k;
    }

    boolean used (int z) {
        return z < this.key;
    }

    // ensures key = old(key) + 1 /\ result = old(key)
    int generate () {
        return this.key++;
    }
}

class DecCounter {
    // constraint key <= old(key)
    int key;

    DecCounter (int k) {
        this.key = k;
    }
```

```
    boolean used (int z) {
        return this.key < z;
    }

    // ensures key + 1 = old(key) /\ result = old(key)
    int generate () {
        return this.key--;
    }
}
```

Note that the history constraint is defined in terms of some function `used`. This is okay as long as `used` is *pure*. Also note that we have old() wrapped around the result of a function without any fields. old(used(z)) means any previous value `used` gave for z. The history constraint means that any int was used in the past, it is still used. We can show that `IncCounter` and `DecCounter` are behavioral subtypes of `GenerateUniqueKey`. For instance we can show that `IncCounter`'s history constraint is stronger than `GenerateUniqueKey`'s.

$$\begin{aligned}
&\vdash old(key) \leq key \rightarrow \forall z, old(used(z)) \rightarrow used(z) \\
old(key) \leq key, old(used(z)) &\vdash used(z) \\
old(key) \leq key, old(z < key) &\vdash z < key \\
old(key) \leq key, z < old(key) &\vdash z < key
\end{aligned}$$

We can demonstrate that `DecCounter`'s history constraint is okay similarly. We can also show that `IncCounter.generate`'s postcondition is stronger than `GenerateUniqueKey.generate`'s.

$$\begin{aligned}
&\vdash key = old(key) + 1 \wedge result = old(key) \rightarrow \neg old(used(result)) \wedge used(result) \\
key = old(key) + 1, result = old(key) &\vdash \neg old(used(result)) \wedge used(result) \\
key = old(key) + 1, result = old(key) &\vdash \neg old(result < key) \wedge result < key \\
key = old(key) + 1, result = old(key) &\vdash \neg result < old(key) \wedge result < key \\
key = old(key) + 1, result = old(key) &\vdash old(key) \leq result \wedge result < key \\
&\vdash result \leq result \wedge result < result + 1
\end{aligned}$$

3 Inheritance

> *Inheritance is not a core concept [of OOP].*
>
> Peter Müller

> *Specifically, programming without inheritance is distinctly not object-oriented; that would merely be programming with abstract data types.*
>
> Grady Booch

Subtyping expresses *classification*, whereas inheritance is a means of *code reuse*. *Subclassing* is both subtyping and inheritance. In inheritance there is only 1 super object at run time The sub object *is* an instance of the super object. In aggregation an object *has* another object as a field, and delegates to that object. It's possible to simulate subclassing with subtyping and delegation.

Example 3.1 (Immutabe and Mutable Types)**.** Consider the following java code for an immutable cell:

```
class ImmutableCell {

    // constraint: old(val) = val
    private int val;
```

```
    public ImmutableCell (int val) {
        this.val = val;
    }

    public int get () {
        return this.val;
    }
}
```

We could want to write a mutable cell. How should it be related to `ImmutableCell`? Ideally a `MutableCell` could inherit all of `ImmutableCell`'s functionality and add a setter:

```
class MutableCell extends ImmutableCell {

    public MutableCell (int val) {
        super(val);
    }

    public void set (int val) {
        this.val = val;
    }
}
```

However `MutableCell` breaks the history constraint on `val`, so it cannot be a Behavioral subtype! In Java it would be best to use aggregation:

```
class MutableCell {
    ImmutableCell cell;

    public MutableCell (int val) {
        this.cell = new ImmutableCell(val);
    }

    public int get () {
        this.cell.get();
    }

    public void set (int val) {
        this.cell = new ImmutableCell(val);
    }
}
```

Admittedly this is stupid: it could just have an `int` field and do away with the overhead of storing another object.

Example 3.2 (Private Inheritance)**.** In C++ it's possible for a class to inherit *privately* from another class. This achieves code reuse without introducing a subtyping relation. Now we can implement the immutable and mutable cell example as we wanted to before.

```
class ImmutableCell {
    private:
        // constraint: old(val) = val
        int val;

    public:
        ImmutableCell (int v) {
            val = v;
```

```
        }

        int get () {
            return val;
        }
};

class MutableCell : private ImmutableCell {
    public:
        MutableCell (int v) : ImmutableCell(v) {}

        ImmutableCell::get

        void set (int v) {
            val = v;
        }
}
```

3.1 Dynamic Method Binding

> *The object-oriented model makes it easy to build up programs by accretion. What this often means, in practice, is that it provides a structured way to write spaghetti code.*
>
> Paul Graham

Definition 3.1 (Static binding)**.** At compile time a method is selected based on the *Static* type of the receiver's *Syntactic* expression.

Definition 3.2 (Dynamic binding)**.** At run time a method is selected based on the *Dynamic* type of the receiver object's *value*.

Java and Scala have Dynamic binding by default, whereas C++ and C# have Static binding by default.

Definition 3.3 (Two-step Method)**.** Given a method call on some receiver, to determine what method will be called given an OO program do:

1. Find the best-matching method based on the *Static* type of the receiver and arguments.
2. Based on the *Dynamic* type of receiver see if this method is overriden.

This is different in each language. In Java for step 1 any method in the Static class and any of its superclass's are equally valid options. In C++ and C# it starts at the Static and goes up the class hierarchy until a suitable method is found. This is each languages *overloading resolution*.

Definition 3.4 (Fragile Base-class Scenario)**.** In general any potentially overriden methods may cause problems. A subclass's behavior may be affected by changes to the superclass, and the way a subclass overrides a superclass may break the superclass's behavior. In particular:

- A superclass should not change calls to Dynamically bound methods.
- subclass's should override any method that would otherwise break an invariant.
- A superclass should never call a Dynamically bound method in a constructor.
- Adding an overloading method to a superclass may affect the subclass (particularly in Java).

In general a superclass does *not* know what method may be invoked at run time for any overridable method.

In C++ and C# a superclass may declare a method as `virtual` if a subclass may override it. In C# a subclass may specify a method with `override` if a superclass declares it with `virutal`. In C# a subclass may specify a method with `new` if it wants to specify that the method doesn't have anything to do with the superclass's method; it does not override and it has different behavior.

3.1.1 Binary Methods and the Visitor Pattern

A Binary method takes the receiver and one parameter (usually of the same type or one related in a subtype hierarchy). We want behavior to depend on the *Dynamic* types of both. However simple overriding is not sound because parameter types should be contravariant.

Example 3.3 (Equals)**.** Consider that when we want to invoke an `equals` method we would like to have the most specialized version invoked.

```
class Animal {
    public boolean equals (Animal a) { ... }
}

class Cat extends Animal {
    public boolean equals (Cat c) { ... }
}
```

`Cat`'s `equals` does *not* override `Animal`'s. `Cat`'s `equals` *overloads* `Animal`'s. Consider:

```
Cat c1 = new Cat();
Cat c2 = new Cat();
Animal a1 = c1;
Animal a2 = c2;
a1.equals(a2);
```

Here `Animal`'s `equals` method is called even though `Cat`'s would be safe and what we want. If we allowed an override than the following unsafe code would be accepted:

```
Cat c = new Cat();
Animal a = new Animal();
c.equals(a);
```

Since method resolution is determined by the Dynamic type of the receiver `c`, `Cat`'s `equals` could be called which makes bad assumptions about its parameter in this case. What we would like is resolution to be determined by both the Dynamic type of the receiver and that of its argument.

In languages such as Java, C++, and C# how does one specialize behavior based on the Dynamic type of both the receiver and that of its argument?

Example 3.4 (Type tests)**.** We can check the type of the parameter:

```
class Cat extends Animal {
    public boolean equals (Animal a) {
        if (a instanceof Cat) {
            Cat c = (Cat) a;
            ...
        } else {
            return super.equals(a);
        }
    }
}
```

Example 3.5 (Double invocation)**.** Both the super and subclass may have appropriate methods to dispatch to depending on the parameter types. Consider the following:

```
class Animal {
    public boolean equals (Animal a) {
        // Give up
        return a.equalsAnimal(this);
    }

    // Terminal method
    public boolean equalsAnimal(Animal a) {
        // General for all Animals, Gives up
        ...
    }

    public boolean equalsBirb(Birb b) {
        // Give up
        return this.equalsAnimal(b);
    }

    public boolean equalsCat(Cat c) {
        // Give up
        return this.equalsAnimal(c);
    }
}

class Birb extends Animal {
    public boolean equals (Animal a) {
        return a.equalsBirb(this);
    }

    public boolean equals (Birb b) {
        // Specific code for Birbs
        ...
    }
}

class Cat extends Animal {
    public boolean equals (Animal a) {
        return a.equalsCat(this);
    }

    public boolean equalsCat(Cat c) {
        // Specific code for Cats
        ...
    }
}
```

In general if we want to specialize a binary method $m : receiver \times T \to A$ where there is a super class T and subclasses S_i, $\forall i, S_i <: T$, we need the following:

- $m \in T \wedge \forall i, m \in S_i$.
- $\forall i, m_i : receiver \times S_i \to A \wedge m_i \in T \wedge m_i \in S_i$
- $m_{general} : receiver \times T \to A \wedge m_{general} \in T$

Where $\in$ is used to denote a method is explicitly implemented in a particular class. In Java $m_{general}$ is inherited by the S_i. $m \in T$ and each $m_i \in T$ will dispatch to $m_{general}$. m in each S_i will dispatch to m_i flipping the arguments.

Example 3.6 (Dynamic resolution in C#)**.** C# allows one to Dynamically resolve the calling method:

```
class Animal {
    boolean equals (Animal a) {
        // General code for Animals
    }
}

class Cat : Animal {
    boolean equals (Cat c) {
        // Specific code for Cats
    }
}

static boolean equals (Animal a1, Animal a2) {
    return (a1 as dynamic).equals(a2 as dynamic);
}
```

`Cat`'s `equals` *overloads* `Animal`'s. Which method is called depends on the *run time* type of each argument. This requires run time checks.

Example 3.7 (Multiple Dispatch)**.** Some (experimental) languages support *multiple dispatch*.

```
class Animal {
    boolean equals (Animal a) {
        // General code for Animals
    }
}

class Cat : Animal {
    boolean equals (Animal@Cat c) {
        // Specific code for Cats
    }
}
```

In a call to `equals` the method is determined by the Dynamic type of the argument as well. In `Cat`'s `equals`, the parameter `c` has Static type `Animal` and *dispatch* type `Cat`. Every call needs to have a unique best method, or else typechecking fails.

3.2 Multiple Inheritance

> *I made up the term "object-oriented," and I can tell you I did not have C++ in mind.*
>
> Alan Kay

> *C makes it easy to shoot yourself in the foot. In C++ it's harder, but when you do, you blow off your whole leg.*
>
> Bjarne Stroustrup

A type may have several subtypes and (direct) supertypes. For instance, it makes sense to say $Tabby <: Cat$ and $Tiger <: Cat$, as well as $Cat <: Animal$ and that $Cat <: Furball$, where $Animal$ and $Furball$ are not necessarily related. Sometimes we would like to reuse code from several superclasses as well. In OOP, why not combine both into *multiple inheritance*?

Remark. *Java and C# only support single inheritance but using aggregation may simulate single inheritance and interfaces.*

Multiple inheritance is just bad. Here's why.

3.2.1 Ambiguities

A subclass S may inherit directly from both T_1 and T_2, and T_1 and T_2 may both have a field f or method m. When an instance of S is the receiver for m or f how does it choose which one? The client may need to explicitly refer to which field or method. Sometimes methods may be combined into one overriding method in the subclass.

3.2.2 Repeated Inheritance

Sometimes a subclass S inherits directly from both T_1 and T_2, where T_1 and T_2 both inherit directly from Top. Top may have some field f. How many copies of f should S have? This is the infamous *diamond problem*.

By default C++ has *non-virtual* inheritance, where two copies of each Top class exist in memory. The constructor for Top is called twice, once for T_1, and once for T_2.

In C++ one may also inherit *virtually*, where only one copy of Top exists in memory and its constructor is just called once. In this case, the smallest subclass S must call the constructor for Top directly. If there is a constructor for Top without arguments, this will be called first by default. Constructors should not realy upon the virutal superclass constructors they call to maintain invariants.

3.3 Linearization

Scala has *mixins* or *traits* that may help specialize classes. Traits extend exactly one superclass and some number of traits. When mixing in a trait in a class declaration the class *must* be a subclass of the direct superclass of each trait.

Remark.

$$forall\ classes\ C, traits\ T\ s.t.\ T\ extends\ C, T <: C$$

The dynamic dispatch order is determined by *linearization*.

Definition 3.5 (Linearization)**.** L determines the dispatch order.

$$L(C\ extends\ C'\ with\ T_1 ... with\ T_n) = C, L(T_n) \bullet ... \bullet L(T_1) \bullet L(C') \tag{11}$$

$$\varepsilon \bullet B = B \tag{12}$$

$$(a, A) \bullet B = \begin{cases} A \bullet B & a \in B \\ a, A \bullet B & a \notin B \end{cases} \tag{13}$$

The initialization order is the opposite.

A subclass only has one copy of a superclass in memory. In initialization each constructor is called exactly once, where arguments to a superclass constructor are provided by the immediately preceding class in the linearization order.

Behavioral subtyping can only be checked when a trait is mixed in. Traits are difficult to reason about because:

- Like in C++ virutal inheritance they don't know how their superclasses get initialized.
- Traits do not know which methods they override.
- Traits do not know Statically which class `super` calls will go to.

instr	::=	`iconst` z	push `int` z onto the stack
	|	$\texttt{load}_\tau$ r	push value in register r onto the stack
	|	$\texttt{store}_\tau$ r	pop top value from stack into register r
	|	$\texttt{op}_\otimes$	pop top 2 values v_1, v_2 from stack and push $v_1 \otimes v_2$
	|	`invokevirtual` $C.m.\tau_r(\tau_1, ..., \tau_n)$	pop args $v_n, ..., v_1$ receiver v then push $C.m.v(v_1, ..., v_n)$
	|	`invokeinterface` $I.m.\tau_r(\tau_1, ..., \tau_n)$	pop args $v_n, ..., v_1$ receiver v then push $I.m.v(v_1, ..., v_n)$
	|	$\texttt{return}_\tau$	pop v and method return v
	|	`ifeq` l	pop v and jump to l if $v = 0$
	|	`goto` l	jump to program line l

Figure 2: JVM instructions

4 More Typing

If Sun were to hand the management of Java over to a committee of monkeys, would it be more successful?

Eric Sink

4.1 Bytecode Verification

The Java Virtual Machine (JVM) runs *java bytecode*. In order to make some guarantees about some arbitrary bytecode loaded by a program, the bytecode must be typechecked.

Definition 4.1 (Java Virtual Machine)**.** JVM is a stack-based language that consists of the following:

- A stack (duh). Operations push elements onto the stack and pop them off.
- A (potentially finite) set of registers. The first stores `this`, then method parameters, then local variables.
- A heap where objects are stored in memory. The stack and registers may have pointers to the heap.

A sample of the JVM instruction set is provided in 2.

JVM instructions are typechecked. There's a max stack size MS, input type list for the stack S, input type list for registers R, and output type list for the stack S', and an output type list for the registers R'. Some of the basic rules are given in 3.

Remark. *Note that the rule for stores does not check that the previous type in the register is related. Thus bytecode typing allows strong updates, something not allowed in java source code.*

Instructions may have several predecessors because of jumps. Thus the *smallest common supertype*, or *join* $\sqcup$ needs to be selected.

Remark. *With multiple subtyping (such as with interfaces in Java) it may not be possible to calculate the join. A run time check is necessary for interface method invocation. This is why the rule for* ***`invokeinterface`*** *does not check that the expected type on the static satisfies the interface, only that it is a Object.*

(Old) Java needed to perform type inference on bytecode. Because joins need to be computed, a *worklist algorithm* is necessary. In this dataflow analysis, for each instruction $instr(p)$ at a program point (line) p, we have $in(p)$ for the types of the stack and registers before the instruction, and $out(p)$ for the types of the stack and registers after the instruction. Thus we have:

$$instr(p) : in(p) \to out(p) \tag{14}$$

The dataflow equation we solve for is:

$$in(p) = \bigsqcup \{out(q) \mid q \text{ precedes } p\} \tag{15}$$

$$\boxed{MS \vdash instr : (S, R) \to (S', R')}$$

$$\frac{|S| < MS}{MS \vdash \texttt{iconst}\ z : (S, R) \to (\texttt{int}.S, R)}\ \textsc{Const} \qquad \frac{|S| < MS \qquad R(r) <: \tau}{MS \vdash \texttt{load}_\tau\ r : (S, R) \to (\tau.S, R)}\ \textsc{Load}$$

$$\frac{\tau' <: \tau \qquad R'(r) = \tau}{MS \vdash \texttt{store}_\tau\ r : (\tau'.S, R) \to (S, R')}\ \textsc{Store}$$

$$\frac{1 + |S| < MS \qquad \otimes : \tau_1 \times \tau_2 \to \tau \qquad \tau_1' <: \tau_1 \qquad \tau_2' <: \tau_2}{MS \vdash \texttt{op}_\otimes : (\tau_1'.\tau_2'.S, R) \to (\tau.S, R)}\ \textsc{Op}$$

$$\frac{n + |S| < MS \qquad \tau' <: C \qquad \tau_1' <: \tau_1\ ...\ \tau_n' <: \tau_n}{MS \vdash \texttt{invokevirtual}\ C.m.\tau_r(\tau_1, ..., \tau_n) : (\tau_n'...\tau_1'.\tau'.S, R) \to (\tau_r.S, R)}\ \textsc{InvokeVirutal}$$

$$\frac{n + |S| < MS \qquad \tau' <: Object \qquad \tau_1' <: \tau_1\ ...\ \tau_n' <: \tau_n}{MS \vdash \texttt{invokeinterface}\ I.m.\tau_r(\tau_1, ..., \tau_n) : (\tau_n'...\tau_1'.\tau'.S, R) \to (\tau_r.S, R)}\ \textsc{InvokeInterface}$$

$$\frac{}{MS \vdash \texttt{ifeq}\ l : (\texttt{int}.S, R) \to (S, R)}\ \textsc{IfEq} \qquad \frac{}{MS \vdash \texttt{goto}\ l : (S, R) \to (S, R)}\ \textsc{Goto}$$

$$\frac{\tau' <: \tau}{MS \vdash \texttt{return}_\tau : (\tau'.S, R) \to (\emptyset, \emptyset)}\ \textsc{Return}$$

Figure 3: JVM instruction typing

We want to take all of the predecessor instructions of p, take the joins of their types in their stacks and registers. This join is exactly what we want to compute for $in(p)$.

To start out we want the stack to be empty, and the registers to contain the known parameter types P_i and $\top$ for the local variables. The algorithm is given in 1. Note that $\sqcup$ fails for different stack sizes.

Algorithm 1 JVM Type Inference

$in(0) \leftarrow (\varepsilon, [P_1, ..., P_n, \top, ..., \top])$
$worklist \leftarrow \{p \mid instr(p) \text{ in method}\}$
while $worklist \neq \emptyset$ **do**
  $p \leftarrow \min worklist$
  $worklist \leftarrow worklist \setminus \{p\}$
  $instr(p) : in(p) \rightarrow out(p)$
  for all q succeeding p **do**
    $in(q) \leftarrow in(q) \sqcup out(p)$
    if $in(q)$ has changed **then**
      $worklist \leftarrow worklist \cup \{q\}$
    end if
  end for
end while

Read this to learn more about java bytecode verification.

Example 4.1 (Bytecode verification example)**.** Suppose we have a class hierarchy $D <: C$ and $E <: C$. Consider the program:

```
0  aload 0;
1  astore 2;
2  aload 1;
3  goto 1;
```

Lets say that the signature that the start of the program is $[], [D, E, \top]$, so we set $in(0) = [], [D, E, \top]$. The worklist at first has all instructions in order.

1. We pop instruction $instr(0) =$ `aload 0` from the worklist. We have $in(0) = [], [D, E, \top]$. Applying LOAD we get $out(0) = [D], [D, E, \top]$. The only successor of $instr(0) =$ `aload 0` is $instr(1) =$ `astore 2`. Since we do not yet have $in(1)$ we set it as $in(1) = out(0) = [D], [D, E, \top]$. Since $in(1)$ has changed we still need $instr(1) =$ `astore 2` in the worklist.

2. We pop instruction $instr(1) =$ `astore 2` from the worklist. We have $in(1) = [D], [D, E, \top]$. Applying STORE we get $out(1) = [], [D, E, D]$. The only successor of $instr(1) =$ `astore 2` is $instr(2) =$ `aload 1`. Since we do not yet have $in(2)$ we set it as $in(2) = out(1) = [], [D, E, D]$. Since $in(2)$ has changed we still need $instr(2) =$ `aload 1` in the worklist.

3. We pop instruction $instr(2) =$ `aload 1` from the worklist. We have $in(2) = [], [D, E, D]$. Applying LOAD we get $out(2) = [E], [D, E, D]$. The only successor of $instr(2) =$ `aload 1` is $instr(3) =$ `goto 1`. Since we do not yet have $in(3)$ we set it as $in(3) = out(2) = [E], [D, E, D]$. Since $in(3)$ has changed we still need $instr(3) =$ `goto 1` in the worklist.

4. We pop instruction $instr(3) =$ `goto 1` from the worklist. We have $in(3) = [E], [D, E, D]$. Applying GOTO we get $out(3) = [E], [D, E, D]$. The only successor of $instr(3) =$ `goto 1` is $instr(1) =$ `astore 2`. We assign $in(1)$ as:

$$\begin{aligned} in(1) &= in(1) \sqcup out(3) \\ &= ([D], [D, E, \top]) \sqcup ([E], [D, E, D]) \\ &= [D] \sqcup [E], [D, E, \top] \sqcup [D, E, D] \\ &= [D \sqcup E], [D \sqcup D, E \sqcup E, \top \sqcup D] \\ &= [C], [D, E, \top] \end{aligned}$$

Since $in(1)$ has changed we add $instr(1) = \texttt{astore 2}$ back into the worklist.

5. We pop instruction $instr(1) = \texttt{astore 2}$ from the worklist. We have $in(1) = [C], [D, E, \top]$. Applying STORE we get $out(1) = [], [D, E, C]$. The only successor of $instr(1) = \texttt{astore 2}$ is $instr(2) = \texttt{aload 1}$. We assign $in(2)$ as:

$$\begin{aligned} in(2) &= in(2) \sqcup out(1) \\ &= ([], [D, E, D]) \sqcup [], [D, E, C] \\ &= [] \sqcup [], [D, E, D] \sqcup [D, E, C] \\ &= [], [D \sqcup D, E \sqcup E, D \sqcup C] \\ &= [], [D, E, C] \end{aligned}$$

   Since $in(2)$ has changed we add $instr(2) = \texttt{aload 1}$ back into the worklist.

6. We pop instruction $instr(2) = \texttt{aload 1}$ from the worklist. We have $in(2) = [], [D, E, C]$. Applying LOAD we get $out(2) = [E], [D, E, C]$. The only successor of $instr(2) = \texttt{aload 1}$ is $instr(3) = \texttt{goto 1}$. We assign $in(3)$ as:

$$\begin{aligned} in(3) &= in(3) \sqcup out(2) \\ &= ([E], [D, E, D]) \sqcup ([E], [D, E, C]) \\ &= [E] \sqcup [E], [D, E, D] \sqcup [D, E, C] \\ &= [E \sqcup E], [D \sqcup D, E \sqcup E, D \sqcup C] \\ &= [E], [D, E, C] \end{aligned}$$

   Since $in(3)$ has changed we add $instr(3) = \texttt{goto 1}$ back into the worklist.

7. We pop instruction $instr(3) = \texttt{goto 1}$ from the worklist. We have $in(3) = [E], [D, E, C]$. Applying GOTO we get $out(3) = [E], [D, E, C]$. The only successor of $instr(3) = \texttt{goto 1}$ is $instr(1) = \texttt{astore 2}$. We assign $in(1)$ as:

$$\begin{aligned} in(1) &= in(1) \sqcup out(3) \\ &= ([C], [D, E, \top]) \sqcup ([E], [D, E, C]) \\ &= [C] \sqcup [E], [D, E, \top] \sqcup [D, E, C] \\ &= [C \sqcup E], [D \sqcup D, E \sqcup E, \top \sqcup C] \\ &= [C], [D, E, \top] \end{aligned}$$

   $in(1)$ has *not* changed so we do not add it back to the worklist.

8. The worklist is empty, so the algorithm is done.

4.2 Parametric Polymorphism

Classes and methods may be parameterized with types (*yay!*). In OOP this is called *generic types*. It is possible to specify upper bounds on these type parameters. Upper bounds via subtyping on type parameters is the lame OOP way of accomplishing ad-hoc polymorphism, like typeclasses in Haskell or traits in Rust. If $S <: T$, how should we relate $G\langle S \rangle$ and $G\langle T \rangle$?

Example 4.2 (Covariant Generics)**.** Suppose:

$$\frac{S <: T}{G\langle S \rangle <: G\langle T \rangle}$$

In Java we could define a class for lists:

```
class List<A> {
    void add (A a) { ... }
    A head () { ... }
}
```

Suppose we had client code:

```
List<String> ls = new List<String>();
List<Object> lo = ls;
ls.add("Hello there");
Object o = lo.head();
lo.add(new Object);
```

Everything works fine until we add a `Object` to the `String` list. recall the principle that covariant types work for reads but not for writes! Thus in general covariant generics are not sound. If we allowed covariant generics than we would need a *run time check* for *writes*.

Example 4.3 (Contravariant Generics)**.** Suppose:

$$\frac{S <: T}{G\langle T\rangle <: G\langle S\rangle}$$

Consider the previous `List` class. Now we could write client code:

```
List<Object> lo = new List<Object>();
List<String> ls = lo;
lo.add(new Object);
ls.add("General␣Kenobi");
String s = ls.head();
```

This time everything works until we try to get the head from the list as a `String`, because it is an `Object`. The recall the principle that contravariant types work for writes but not for reads! Thus in general contravariant generics are unsound. If we allowed contravariant generics than we would need a *run time check* for *reads*.

Java has chosen to only allow *invariant* generics. By contrast Scala only has invariant generics by default, allowing one to specify if a type parameter is covariant or contravariant. Scala allows contravariant type parameters when they only occur in a positive position, are only read from. Scala allows contravariant type parameters when they only occur in a negative positive, i.e. are only written to.

Example 4.4 (Scala covariant parameter)**.** Consider an immutable cell in Scala:

```
class ImmutableCell[+A] (val : A) {
    def get() : A { val }
}
```

Here `A` is only used as a return type, i.e. a positive position.

Question: what about constructor?

Example 4.5 (Scala contravariant parameter)**.** Consider a comparison interface:

```
interface Comparison[-A] {
    int compare(A);
}
```

Here `A` is only used as a method parameter to be written to, i.e. a negative positive.

When we are stuck with *invariant* generics how do we write code that works with different instantiations of a type parameter?

In OOP (especially Java) this is an issue when what you want is ad-hoc polymorphic functionality for a supertype that works on subtypes, but since you don't have ad-hoc polymorphism you need to create another interface and class(es) that have their own subtype relations. Also you cannot pass functions as arguments in these languages in general so ... here's a convoluted example to motivate a kind of problem one may encounter.

Example 4.6 (Additional type parameters)**.** Here we are going to try to demonstrate that Java forces one to twist themselves into a pretzel if they just want to use standard type parameters. Suppose there's some `Comparator` interface and generic `TreeSet` class that represents a set implemented in a tree data structure. In order to search for elements it needs to compare elements in binary nodes of the tree. Thus it needs some way to have a `compare` function for its elements. Because we are in Java we cannot pass functions as arguments.

```
interface Comparator<T> {
    int compare(T x, T y);
}

class TreeSet<E extends Comparator<E>> {
    ...
    TreeSet () {}
    void add (E elem) { ... }
    boolean member (E elem) { ... }
    void remove (E elem) { ... }
}
```

Suppose that we would like to create a `TreeSet<Student>` where `Student`s are a `Person`, and `Person` has a general comparison operation for people.

```
class Person implements Comparator<Person> {
    ...
    int compare (Person x, Person y) { ... }
}

class Student extends Person {
    ...
}
```

Where we would like to have client code:

```
TreeSet<Student> ts = new TreeSet<Student>();
```

Well too bad. Java gets angry because `Student` is "not within the bounds of the type variable (of `TreeSet`)." That is, we don't know $Student <: Comparator\langle Student\rangle$. If we try to have `Student` directly implement the `Comparator` interface Java still isn't satisfied:

```
class Student extends Person implements Comparator<Student> {
    int compare(Student x, Student y) { ... }
}
```

Java complains that "`Comparator` cannot be inherited with different arguments: `Student` and `Person`." If `Person` does not implement `Comparator` then Java is satisfied, but we would like it to. If we would like to have our cake and eat it too we need to modify our `TreeSet` class as follows:

```
class TreeSet<E extends F, F extends Comparator<F>> {
    ...
    TreeSet () {}
    void add (E elem) { ... }
    boolean member (E elem) { ... }
    void remove (E elem) { ... }
}
```

And then out client code would need to look like:

```
TreeSet<Person, Student> ts = new TreeSet<Person, Student>();
```

Here we needed to make explicit in `TreeSet`'s type parameters that the element type does not necessarily need to have its own `Comparator`, but that it has a supertype that implements `Comparator`. As shown this is very tedious.

How can we avoid addition type parameters and constraints?

4.2.1 Wildcards

Definition 4.2 (Wildcards)**.** Java has so called *wildcard*s that represent *existential types*.

$$G\langle ? \rangle \triangleq \exists A, G\langle A \rangle \tag{16}$$

In Java there is no way to explicitly instantiate these existential types. Java's type system does this automatically. The type argument to the instance of an existential type is abstract to clients.

Example 4.7 (Wildcard identity)**.** Java accepts the following identity function:

```
static List<?> id(List<?> l) {
    return l;
}
```

In the body of `id` the Java compiler is able to show $\exists E, List\langle E \rangle$, where `l` is a *proof* of $\exists E, List\langle E \rangle$. However it does not know that the parameter type and return type are the same.

```
List<String> l = id(new ArrayList < String > );
```

The above fails because java only knows that the result of `id(new ArrayList<List>)` is $\exists E, List\langle E \rangle$. Java does not remember that the E in question is a `String`. The best we can do is:

```
List<?> l = id(new ArrayList < String > );
```

Java does not even allow:

```
List<Object> l = id(new ArrayList < String > );
```

Since Java generics are *invariant* in general $\exists E, G\langle E \rangle \not<: G\langle Object \rangle$.

Example 4.8 (Reading from and writing to wildcards)**.** Java rejects the following:

```
static void merge (Collection<?> x, Collection<?> y) {
    x.addAll(y);
}
```

We have the following types:

$$\begin{aligned} \texttt{x} &: \exists A, Collection\langle A \rangle \\ \texttt{y} &: \exists B, Collection\langle B \rangle \\ \texttt{addAll} &: \forall C, Collection\langle C \rangle \times (\exists D, Collection\langle D \rangle \wedge D <: C) \to () \end{aligned}$$

When `x` is the receiver of `addAll`, its universal type C is instantiated with some unknown parameter type. Java is implicitly unpacking `x`'s type as some unknown A.

$$\texttt{addAdll}\langle A \rangle : Collection\langle A \rangle \times (\exists D, Collection\langle D \rangle \wedge D <: A) \to ()$$

When `y` is passed as an argument to `x.addAll` `y`'s type is implicitly unpacked with some unknown type parameter B. Java will not be able to satisfy the type bounds of

$$\texttt{addAdll}\langle A \rangle : Collection\langle A \rangle \times (genericCollectionB \wedge B <: A) \to ()$$

Since there's no relationship between A and B. Even if we try:

```
static void merge (Collection<?> x, Collection<?> y) {
for (Object elem : y) {
        x.add(elem);
    }
}
```

Java will still be unconvinced. We have:

$$\texttt{add} : \forall C, Collection\langle C\rangle \times C \to ()$$
$$\texttt{add}\langle A\rangle : Collection\langle A\rangle \times A \to ()$$

Which is even more strict. Of course $A <: Object$ and $B <: Object$. However in order to pass `elem` into `x.add`, we need that `Object <: A`.

We would like to be able to constrain wildcards in some way.

Definition 4.3 (Bounding wildcards)**.** Java offers the following:

$$G\langle ?\ \texttt{extends}\ T\rangle \triangleq \exists A, G\langle A\rangle \wedge A <: T$$
$$G\langle ?\ \texttt{super}\ T\rangle \triangleq \exists A, G\langle A\rangle \wedge T <: A$$

Example 4.9 (Reading from and writing to wildcards revisited)**.** We may now constrain the existential type parameters in our `merge` example:

```
static <E> void merge (Collection<? super E> x, Collection<? extends E> y) {
    x.addAll(y);
    for (A elem : y) {
        x.add(elem);
    }
}
```

Everything works now because we have the following:

$$\texttt{x} : \exists A, Collection\langle A\rangle \wedge E <: A$$
$$\texttt{y} : \exists B, Collection\langle B\rangle \wedge B <: E$$
$$\texttt{add} : \forall C, Collection\langle C\rangle \times C \to ()$$
$$\texttt{addAll} : \forall C, Collection\langle C\rangle \times (\exists D, Collection\langle D\rangle \wedge D <: C) \to ()$$

When `x`'s type gets implicitly unpacked we get some type A and the knowledge that $E <: A$. Similarly when `y`'s type is unpacked we get some type B and the knowledge that $B <: E$. When `x` is the receiver of `add` and `addAll` we get:

$$\texttt{add}\langle A\rangle : Collection\langle A\rangle \times A \to ()$$
$$\texttt{addAll}\langle A\rangle : Collection\langle A\rangle \times (\exists D, Collection\langle D\rangle \wedge D <: A) \to ()$$

Thus when we pass `elem` into `x.add` we need to show that $B <: A$. This is true by transitivity of subtyping. The same evidence is used when we pass `y` into `x.addAll`.

In java, given some generic class G, we can use wildcards to make certain instances of G covariant or contravariant. This is called *use-site variance*.

Example 4.10 (Use-site variance)**.** Suppose we have some cell class:

```
class Cell<E> {
    private E elem;
    Cell (E elem) { this.elem = elem; }
    E get () { return this.elem; }
    void set (E elem) { this.elem = elem; }
}
```

Say we have that `Student extends Person`. We can create a covariant `Cell` that only allows us to perform reads:

```
1 Cell<? extends Person> readOnly = new Cell<Student>(new Student());
2 Object o  = readOnly.get();   // good
3 Person p  = readOnly.get();   // good
4 Student s = readOnly.get();   // bad
5 readOnly.set(new Object());   // bad
6 readOnly.set(new Person());   // bad
7 readOnly.set(new Student()); // bad
```

Both of the final `set` calls fail, as Java is unable to correctly constrain the type parameters. We have the type information:

$$\begin{aligned}\texttt{readOnly} &: \exists E, Cell\langle E\rangle \wedge E <: Person\\ \texttt{readOnly.get} &: () \to E \wedge E <: Person\\ \texttt{readOnly.set} &: E \wedge E <: Person \to ()\end{aligned}$$

Where for `readOnly.get` and `readOnly.set` E is some unpacked type whose only constraint is that $E <: Person$. Let's take this line by line.

1. Declare a covariant-use-site `Cell` with an upper bound `Person`.
2. Typechecks because the result E of `readOnly.get` is bounded above by `Person`, so it is acceptable to assign it to an `Object`.
3. Typechecks because the result E of `readOnly.get` is bounded above by `Person`, so it is acceptable to assign it to a `Person`.
4. Reading as a `Student` fails because we need to somehow show that $E <: Student$, but all we have is that $E <: Person$. In principle E could be a `Person`, which does not work as a `Student` in general.
5. Setting the cell to `Object` fails because we would need to show that $Object <: E$ and $Object <: Person$, which does not hold.
6. Attempting to set the cell to a `Person` fails because we do not know that $Person <: E$.
7. Setting the cell to a `Student` fails because of the same principle.

Intuitively writing to `readOnly` fails because we have information about the lower bound of E.

We may also create a contravariant cell that only allows us to perform writes. Suppose that there is also some class `MasterStudent extends Student`.

```
1 Cell<? super Student> writeonly = new Cell<Person>(new Person());
2 writeonly.set(new MasterStudent()); // good
3 writeonly.set(new Student());       // good
4 writeonly.set(new Person());        // bad
5 writeonly.set(new Object());        // bad
6 MasterStudent m = writeonly.get();  // bad
7 Student s       = writeonly.get();  // bad
8 Person p        = writeonly.get();  // bad
9 Object o        = writeonly.get();  // good
```

We have the type information:

$$\begin{aligned}\texttt{readOnly} &: \exists E, Cell\langle E\rangle \wedge Person <: E\\ \texttt{readOnly.get} &: () \to E \wedge Person <: E\\ \texttt{readOnly.set} &: E \wedge Person <: E \to ()\end{aligned}$$

Where for `readOnly.get` and `readOnly.set` E is some unpacked type whose only constraint is that $Person <: E$. Let's take this line by line.

1. Declares a contravariant-use-site `Cell` with a lower bound `Student`.
2. Typechecks because we have that $MasterStudent <: E$ from $MasterStudent <: Student$ and $Student <: E$.
3. Typechecks because we have that $Student <: E$.
4. Fails because $Person <: E$ does not hold.
5. Fails because $Object <: E$ does not hold.
6. Fails because $E <: MasterStudent$ does not hold in general.
7. Fails because $E <: Student$ does not hold in general.
8. Fails because $E <: Person$ does not hold in general.
9. Typechecks because $E <: Objects\ always$ holds.

In principle it is possible to downcast to a more specific type.

Example 4.11 (TreeSet revisited)**.** We may now only have a single type parameter for `TreeSet` via wildcards.

```
class TreeSet<E extends Comparator<?>> {
    ...
    TreeSet () {}
    void add (E elem) { ... }
    boolean member (E elem) { ... }
    void remove (E elem) { ... }
}
```

We may now define client code:

```
TreeSet<Student> ts = new TreeSet<Student>();
```

In general wildcards allow more flexibility than vanilla type parameters.

- Java does not allow lower bounds `super` for normal type parameters.
- Instantiation of a wildcard $G\langle ? \rangle$ may change over time: it may be reassigned an instance with a different type argument. However when reassigning a normal generic instance of $G\langle T \rangle$ it must always have a type argument of exactly T.

For some generic type G, we may define the set of instance types given some type argument. TODO: define this better or remove.

$$[\![G\langle T \rangle]\!] = \{H\langle T \rangle \mid \forall H, H <: G\} \tag{17}$$
$$[\![G\langle ? \rangle]\!] = \{H\langle T \rangle \mid \forall H, T, H <: G\} \tag{18}$$
$$[\![G\langle ? <: T \rangle]\!] = \{H\langle S \rangle \mid \forall H, S, H <: G \land S <: T\} \tag{19}$$
$$[\![G\langle S <: ? \rangle]\!] = \{H\langle T \rangle \mid \forall H, T, H <: G \land S <: T\} \tag{20}$$

Definition 4.4 (Subtyping generics)**.** Given generic types H and G, we say that:

$$H\langle \mathcal{I}_1 \rangle <: G\langle \mathcal{I}_2 \rangle \equiv [\![H\langle \mathcal{I}_1 \rangle]\!] \subseteq [\![G\langle \mathcal{I}_2 \rangle]\!]$$

where $\mathcal{I}$ is either a type parameter or a wildcard bound.

We may write the subtyping rules for generics as in 4.

Remark. *Subtyping in Java is undecidable do to wildcards.*

$$\frac{H\langle\mathcal{I}\rangle <: G\langle\mathcal{I}\rangle \qquad G\langle\mathcal{I}\rangle <: F\langle\mathcal{I}\rangle}{H\langle\mathcal{I}\rangle <: F\langle\mathcal{I}\rangle}$$

$$\frac{S <: T}{G\langle T\rangle <: G\langle S <:?\rangle} \qquad \frac{S <: T}{G\langle ? <: S\rangle <: G\langle ? <: T\rangle} \qquad \frac{S <: T}{G\langle T <:?\rangle <: G\langle S <:?\rangle}$$

Figure 4: Generic and wildcard subtyping rules for Java

Example 4.12 (Java example rules)**.** Here is a concrete Java program that attempts to use certain inference rules. It is interesting to see what Java allows and disallows.

```
import java.util.*;

class Animal {

    Animal () {}
}

class Cat extends Animal {
    Cat () {}

    void meow () {
        System.out.println("meow");
    }

}

class Dog extends Animal {
    Dog () {}

    void bark () {
        System.out.println("bark");
    }

}

public class MyClass {
    public static void main(String args[]) {
        ArrayList<Animal> lAnimal = new ArrayList<Animal>();
        ArrayList<Cat>    lCat    = new ArrayList<Cat>();

        // Java forbids covariant generics
        // lAnimal = lCat;
        // lAnimal.add(new Dog());

        // Java forbirds contravariant generics
        // lCat = lAnimal;


        // Java forbids
        //            S <: T
        // ----------------------
```

```
// G < ? <: S > <: G < T >

ArrayList<Animal>         animalList = new ArrayList<Animal>();
ArrayList<? extends Cat> catList    = new ArrayList<Cat>();

// Java forbids this?
// animalList = catList;

// Java allows
//       S <: T
// ------------------------
// G < T > <: G < S <: ? >

ArrayList<? super Cat> catlist    = new ArrayList<Object>();
ArrayList<Animal>      animallist = new ArrayList<Animal>();

catlist = animallist;

// Java allows
//            S <: T
// ---------------------------
// G < ? <: S > <: G < ? <: T >

ArrayList<? extends Animal> ldog = new ArrayList<Dog>(Arrays.asList(new Dog()));
ArrayList<? extends Cat>    lcat = new ArrayList<Cat>(Arrays.asList(new Cat()));
ldog = lcat;

// Java forbids writes to covariantly declared object
// ldog.add(new Dog());


// Java allows
//            S <: T
// ---------------------------
// G < T <: ? > <: G < S <: ? >
ArrayList<? super Animal> lsub = new ArrayList<Animal>();
ArrayList<? super Cat>    ltop = new ArrayList<Object>();
ltop = lsub;

// Put a Cat in a list with lowerbound as Animal
ltop.add(new Cat());
Object element = lsub.get(0);
System.out.println(element instanceof Cat);
((Cat) element).meow();

// Java forbids
//            S <: T
// ---------------------------
// G < ? <: S > <: G < T <: ? >
ArrayList<? super Animal> lTop = new ArrayList<Object>();
ArrayList<? extends Cat>  lSub = new ArrayList<Cat>(Arrays.asList(new Cat()));

// Java forbids
// lTop = lSub;
```

```
        Cat cat1 = lSub.get(0);

        // Put a Cat in a list with a lower bound as Animal
        lTop.add(cat1);

        Cat cat = (Cat) lTop.get(0);
        cat.meow();
    }
}
```

It was not possible to obtain assurance as to which inference rules are allowed by Java and those that are not so hopefully this example will illuminate any issues.

4.2.2 Type Erasure

For backwards compatibility, Sun did not want to change the JVM after generics were introduced. So generic type information is erased by the compiler. $G\langle T\rangle$ becomes just G in java bytecode. Occasionally the compiler needs to insert casts to the class used for the type argument for instances of generic types. Because of this, Java disallows the following:

- Performing `instanceOf` on a generic instance type $G\langle T\rangle$.
- `.class` of a generic instance type.
- arrays of generic types $G\langle T\rangle[\,]$.

C# allows the above.

Example 4.13 (Erasure and overloading)**.** Java disallows the following because there is a name clash between each `foo` as a result of type erasure.

```
class Erasure<S, T> {
    void foo (S p) {}
    void foo (T p) {}
}
```

However C# does allow this in general. If one chooses to instantiate `Erasure` with the same types any call to foo will be ambiguous and cause a typechecking error.

```
Erasure<Object, Object> erasure = new Erasure<Object, Object>();
erasure.foo(new Object()); // ambiguous!
```

In Java one can resolve the issue by giving the type parameters different upper bounds.

In Java static fields are shared by all instantiations of a generic class, but not in C#.

4.2.3 C++ Templates

In C++ one provides type parameters to classes and methods via *templates*. As you will see, intuitively templates are glorified macros.

Every time a template is instantiated the C++ generates an instantiated version of the code for the class or method for the given type arguments. This causes *code bloat*. Furthermore C++ templates are not typechecked. Only the parts of instance classes that get used after the class is instantiated get typechecked. In principle one could make all kinds of assumptions about the type parameter in a template class (about what methods and fields it has) and C++ will not complain unless its instantiation doesn't provide suitable type arguments.

C++ later added *concepts*. Concepts provide a way to specify constraints about a template's type parameter. These constraints are checked upon instantiation. To read more about concepts look here.

5 Information Hiding and Encapsulation

> *The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.*
>
> Joe Armstrong

5.1 Information Hiding

Definition 5.1 (Information Hiding)**.** A technique to *reduce* dependencies between modules. A module only provides clients with the fields and methods such that no invariants can be broken by clients. This is a *Static* concept.

Ideally class interfaces are thin and hide implementation details. To understand a single class hopefully one just has to understand the interfaces of any dependencies without looking at their implementations.

To use a class a client should only need to know:

- The class name.
- The type parameters and their bounds.
- The signatures and pre and postconditions of exposed methods.
- The types, invariants, and constraints of exposed fields.
- The above for any superclass or superinterface.

Example 5.1 (Subclass interface)**.** We for some class S we can consider the interface it has to its superclass T. This includes any `protected` fields and methods of the superclass.

Example 5.2 (Friend (package) interface)**.** Classes defined in the same package have access to each others `default` and `protected` fields and methods. They may be mutually defined.

Java offers the following access modifiers:

- `private`: Only the implementing class has access to these methods and fields. They can not even be overriden by subclasses. The most restrictive.
- `default`: Only classes defined in the same package has access to these fields and methods. In general they cannot be overriden by subclasses, unless the subclass is defined in the same package. Available in the friend interface.
- `protected`: These fields and methods are available to classes in the same package and any potential subclasses. Thus they may be overriden. Available in the subclass and friend interface.
- `public`: Everyone and their mother has access to these fields and methods. Of course may be overriden.

Question: how do access modifiers interact with overloading?

Recall 3.4. In principle any non-`private` field or method, anything that can be overriden may cause problems. Thus when changing a base class, any *observable* behavior, i.e. **not** `private` must be preserved.

Definition 5.2 (JLS1 Java method selection)**.** When some receiver o invokes a method m, Statically/at compile time:

1. Find the static declaration of m.
2. Determine if m is accessible at the call site.
3. Determine if m is virtual (may be overriden) or non-virtual.

Then at run time

1. Compute the receiver's value.
2. Based on the Dynamic type of the receiver locate the method m to invoke.

Definition 5.3 (The Access rule)**.** The access modifier of an overriding method must be at least as open as that of the overriden method.

Definition 5.4 (Rules for overriding)**.** If $S <: T$, $S.m$ overrides $T.m$ only if $T.m$ is accessible in $S.m$.

Example 5.3 (Default access methods)**.** Consider the class `Super` defined in package `A`.

```
package A;
class Super {
    void foo() {
        System.out.println("A");
    }
}
```

And the class `Sub` in package `B`.

```
package B;
class Sub extends A.Super {
    public void foo() {
        System.out.println("B");
    }
}
```

Suppose we have the following client code that invokes `foo` in package `A`.

```
Super soup = new Sub();
soup.foo();
```

Since `Super.foo()` is not accessible in `Sub.foo()`, we would expect "A" to be printed. This is because `Super.foo` is `default`, thus is not available in `Sub`, and thus cannot be overriden by `Sub.foo`. However we would be wrong under the rules in 5.2. Under these rules `Sub.foo` will be selected and "B" will be printed. Let's step through the rules. First, at compile time:

1. We find the Static declaration of `foo` in `Super`.
2. We find that `Super.foo` is accessible at the call site.
3. The invocation is virtual.

Then at run time:

1. We find the Dynamic type of the receiver `soup` is `Sub`.
2. We locate `Sub.foo` since it is `public` and in the class `Sub` of the receiver `soup`.

Definition 5.5. [JLS2 Java method selection] Java fixed their broken 5.2 by ensuring that the Dynamically selected method is able to override the Statically determined method. When some receiver o invokes a method m, Statically/at compile time:

1. Find the static declaration of m.
2. Determine if m is accessible at the call site.
3. Determine if m is virtual (may be overriden) or non-virtual.

Then at run time

1. Compute the receiver's value.

2. Based on the Dynamic type of the receiver locate the method m to invoke **that overrides the Statically determined method!**

This is a refinement of 3.3 that more precisely considers access modifiers.

Example 5.4 (Don't always use protection)**.** Let's consider the previous example, but now the modifiers are `protected`.

```
package A;
class Super {
    protected void foo() {
        System.out.println("A");
    }
}
```

And the class `Sub` in package `B`.

```
package B;
class Sub extends A.Super {
    protected void foo() {
        System.out.println("B");
    }
}
```

Suppose we have the following client code that invokes `foo` in package `A`.

```
package A;
class Client {
    void client() {
        Super soup = new Sub();
        soup.foo();
    }
}
```

Again we would expect `Super.foo` to be invoked, printing "A", since `Sub.foo` is `protected`, and this unavailable to `Client` as `Client` is not a subclass of `Sub` nor is it in the same package. And again, even under the refined 5.5 we would be wrong. Let's go through the steps. At compile time:

1. We determine that the Static declaration is `Super.foo`.
2. Since `Super.foo` is `protected` and `Client` is in the same package as `Super`.
3. `Super.foo` may be overriden.

Then at run time:

1. The Dynamic value of `soup` is `Sub`.
2. Indeed `Sub.foo` overrides `Super.foo`, thus `Sub.foo` is called.

Thus `Client` was able to call a method that was not even available to it!.

5.2 Encapsulation

Ideally, any module behaves the same way, according to its specification, no matter what the calling context it. The client relies upon the **consistency of internal representations** of the invoked module. Furthermore the module should prevent clients from being able to violate its invariants. For instance, non-`private` fields are a way to set yourself up for failure, so stay outta trouble kids.

Definition 5.6 (Encapsulation)**.** A technique to separate the memory state available to different *capsules*, so the memory states available to each capsule are disjoint. The boundaries delimiting memory are achieved via clear interfaces to these capsules so these capsules may guarantee invariants about their internal states. In this course encapsulation relates to the state of the program at run time, thus is a *Dynamic* concept.

Often the terms information hiding and encapsulation are used interchangeably, but here they are distinct. What is a capsule? It can be:

- An individual object.
- A structure composed of many objects.
- A class (with all of its objects).
- A package (with all of its classes and objects).

To provide encapsulation for a capsule we requires a precisely defined boundary and an interface for that boundary. The internal representation of a capsule is encapsulated if it can only be manipulated using its interface. To achieve internal consistency (invariants) of objects we should:

1. Apply information hiding.
2. Make consistency criteria explicit, via the use of contracts, i.e. with invariants, history constraints, pre- and postconditions.
3. Check that anything exposed by the interface, including those of possible subclasses, preserve consistency and prevent malicious clients from breaking consistency.

In practice this means that invariants should only refer to `private` fields, since anything else can be set by a client.

All this begs the question: what is an object structure?

6 Object Structures and Aliasing

> *Object-oriented programming had boldly promised "to model the world." Well, the world is a scary place where bad things happen for no apparent reason, and in this narrow sense I concede that OO does model the world.*
>
> Dave Fancher

Single objects are the atomic building blocks of OOP. From these we can build more complex objects or *object structures* when their classes have fields that are other objects.

Definition 6.1 (Object structure)**.** A set of objects connected via references.

6.1 Aliasing

Definition 6.2 (Aliasing)**.** We say that an object o is *aliased* by the aliases x and y if they both hold references to o. That is

$$x \mapsto o \land y \mapsto o \land x \neq y$$

An alias may be

- Local variables of methods, including `this`.
- Class fields (including static).
- Left expressions in general.

Definition 6.3 (Capturing)**.** A form of unintended aliasing where a reference gets stored in a data structure. Often this occurs in constructors.

Aliases may be used to get around an object's boundary/interface.

Definition 6.4 (Leaking)**.** Occurs when a data structure passes out or exposes a reference to some ostensibly internal object. This usually happens by mistake. Leaking can be exploited to bypass an object's boundary/interface.

6.2 Problems of Aliasing

The big lie of object-oriented programming is that objects provide encapsulation.
John Hogg.

Remark. *If an interface has a method that accepts some object or object structure as an argument, and it captures that argument, its interface may be bypassed if the caller decides to mutate what was passed in. Just making fields private that represent the internal state of the callee does nothing.*

Remark. *If an interface has a method that returns an object, array, or reference part of its internal data structure this is leaking, and any client may modify this object and thus the internal state.*

Here's a list of some other specific issues with aliasing:

- When performing explicit memory management one needs to be sure that when one is freeing a pointer that there are no other aliases. This is the so-called dangling pointer problem.
- Compilers should not inline objects that are aliased.

6.3 Unique References

For some data structures we would like to be sure there is no aliasing. Some languages/libraries provide some support for this with *unique pointers*.

6.3.1 C++ Unique Pointers

C++ offers a library for unique pointers. Direct leaking is prevented by the omission of the copy constructor. Direct capturing is prevent by omission of the assignment operator.

To pass unique pointers around in C++ one needs to explicitly *move* them. Moving them prevents aliasing by rendering the old pointer unusable. Accessing the old pointer leads to undefined behavior, such as nontermination.

Because unique pointers are in general not aliased, when they are deleted C++ deallocates the underlying objects as well. Because there are (in theory) no aliases one does not need to worry out the dangling pointer problem.

Example 6.1 (Direct leaking and capturing)**.** Consider the following code for an `Address` and `Person` class.

```
class Address {
    public:
        std::string city;

        Address(std::string city) {
            this->city = city;
        }

        std::string getCity() {
            return this->city;
        }
}

class Person {
    std::unique_ptr<Address> addr;

    std::unique_ptr<Address> getAddr() { return addr; }

    void setAddr(std::unique_ptr<Address> a) { addr = a; }
}
```

C++ forbids both `getAddr` and `setAddr`. C++ rejects `getAddr` because `unique_ptr` deletes the copy constructor, so a pointer cannot be copied out of the class. This is good behavior because it prevents a client from aliasing `addr`, thus no leaking. It rejects `setAddr` because `unique_ptr` deletes the assignment operator, preventing the class from having an alias to some pointer the client passed in, thus preventing capturing. To get C++ to accept `Person` we need to use the `move` operator.

```
class Person {
    std::unique_ptr<Address> addr;

    std::unique_ptr<Address> getAddr() { return std::move(addr); }

    void setAddr(std::unique_ptr<Address> a) { addr = std::move(a); }
}
```

In `getAddr` the pointer in `addr` is moved out of the class and given to the client. Attempting to read from `Person`'s `addr` will result in undefined behavior, probably nontermination. In `setAddr` the pointer the client passes will be moved into the `addr` field, rendering it unusable to the client.

Example 6.2 (Ownership transfer)**.** When a unique pointer is passed into a function it needs to be moved. This means that the client will no longer be able to use it after the call. Imagine that there is some compare function on unique pointers of addresses.

```
int compare(std::unique_ptr<Address> x, std::unique_ptr<Address> y) { ... }
```

Say some client code would like to compare two addresses and do something with them afterwards.

```
auto x = std::make_unique<Address>("Hagen");
auto y = std::make_unique<Address>("Zurich");
int  c = compare(std::move(x), std::move(y));
auto small_address = c < 0 ? std::move(x) : std::move(y); // C++ mad
```

We would've liked to use `compare` to figure out the smaller address, but since both `x` and `y` were moved into `compare` the client can no longer access them. We could imagine some alternative `compare` that returns `x` and `y` in addition to the `int`.

```
std::tuple<int, std::unique_ptr<Address>, std::unique_ptr<Address>>
compare(std::unique_ptr<Address> x, std::unique_ptr<Address> y) { ... }
```

And then use this in the client:

```
auto x = std::make_unique<Address>("Los Angeles");
auto y = std::make_unique<Address>("Reno");
auto t = compare(std::move(x), std::move(y));
int  c = std::get<0>(t);
x = std::move(get<1>(t));
y = std::move(get<2>(t));
auto small_address = c < 0 ? std::move(x) : std::move(y);
```

C++ accepts this. However `x` and `y` are moved when we assign `small_address`. We've merely kicked the problem down the road. One could imagine programming in this style but C++ offers another solution. Compare could take a pointer to the unique pointer.

```
int compare(std::unique_ptr<Address> *x, std::unique_ptr<Address> *y) { ... }
```

Now the client will pass `x` and `y` as references into compare.

```
auto x = std::make_unique<Address>("Tokyo");
auto y = std::make_unique<Address>("Cairo");
int  c = compare(&x, &y);
auto small_address = c < 0 ? std::move(x) : std::move(y); // C++ mad
```

C++ allows one to create an alias to the data in a unique pointer with a `get` method. Obviously this causes problems.

Example 6.3 (Dangling unique pointer)**.** One can create a dangling pointer using `get`. Imagine there is some method that returns `get` on a newly created unique address.

```
Address* make_addr() {
    auto addr = std::make_unique<Address>("Zurich");
    return addr.get();
}
```

And we have a client that tries to call `getCity` on the result.

```
std::cout << make_addr()->getCity() << std::endl;
```

Since C++ deletes the data `addr` points to in `make_addr`, calling `getCity` on the result is undefined behavior.

Example 6.4 (Aliasing unique pointers)**.** Other than `get`, one can alias a unique pointer by constructing one with previously existing data.

```
Address peter_address = new Address("Zurich");
std::unique_ptr<Address> alias(peter_address);
alias->city = "Hagen";
std::cout << peter_address->getCity() << std::endl;
```

We have sent someone to Hagen, which is the worst thing that could happen.

Furthermore, unique pointers in C++ are shallow. One may easily alias any inner objects or pointers. To get deep ownership one needs to make all fields unique pointers recursively.

6.3.2 Rust Ownership

Rust's type system enforces ownership. Moves occur implicitly because all assignments are moves. When one tries to read from a moved value, instead of undefined behavior at run time, the typechecker rejects the program. Ownership in Rust is deep. In Rust the `Box` type allows one to have deeply owning pointers to memory: a better version of `unique_ptr`.

One can transfer values back and force in Rust like in C++, but it is more idiomatic pass by reference. This is called *borrowing*. Borrowing creates a temporary alias. These aliases come with more restrictions. Read-only references `& T` may be shared and have many aliases but cannot be written to. Mutable references `&mut T` may be written to but there may only be one. Rust's borrow checker analyzes the program syntax and determines the availability of references, or *lifetimes*. A reference's lifetime is the span of program syntax from when it is born into this world and when it is last used. Leaking is possible in Rust if a function returns a mutable reference. However capturing is safe because the capturing data structure owns the captured value.

Remark. *Rust can safely deallocate memory of variables at the end of their scope because of the uniqueness of ownership.*

Remark. *In safe Rust it is not possible to construct cyclic data structures.*

6.4 Read-only References

In stateful programming aliases require much less memory overhead than deep-copying data structures. So despite their many drawbacks we would like to use aliases, but with much better safety guarantees. Intuitively we can avoid many of the problems with aliases if we require that they are *read-only*. We would like to have the following properties for read-only references.

- Distinguish between clients that may mutate data via a reference verses those who may not. The data itself isn't read-only, the references to it are.

- Effectively prevent clients from modifying some state.
- We would like read-only access to extend to fields of data, that is we would like *transitivity*.

Rust's read-only references satisfy these properties.

C++ supports `const` pointers. On the surface they are read-only. However inner fields may be updated, and they may be cast to non-constant pointers.

C++ also supports `const` methods that may not modify their receiver object. Question: Can const methods update fields within const fields?

TODO: example of rust borrow checking, when a mutable reference may be created after an immutable reference is no longer used.

7 Initialization

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965.*
>
> Tony Hoare

> *Null is very nice and intuitive to work with. There's no alternative more expressive, safe, nor efficient.*
>
> Nobody

7.1 Simple Non-null Types

`null` seems convenient. Programmers like to use `null` to

1. Indicate the absence of a value. For example as a terminal node in a recursively defined type.
2. Initialize fields by default.

In other words:

1. The language doesn't have algebraic data types or the programmer chose not to use them.
2. It's an object-oriented setting and the programmer was too lazy to write a constructor that initializes the fields.

`null` is cringe. Algebraic data types give you more expressiveness and safety. But some *certain* languages (*cough, cough* Java, C, C++, etc.) thought `null` would be simpler to implement and program with.

Unfortunately will not be getting rid of `null` all together. Instead we will distinguish between `null` types and *non-`null`* types. $T!$ denotes references to T objects. $T?$ denotes references to T objects *or* `null`.

$$values(T?) = \{\texttt{null}\} \cup values(T!)$$

Definition 7.1 (Type safety invariant)**.** We would like to Statically guarantee:

$$\forall e, v, T, e : T! \to e \downarrow v \to v \neq \texttt{null}$$

We would like to prevent `null` dereferencing at compile time (while still having `null` in the language). I.e. we would like to prevent the receiver from ever being `null`. Because $values(T!) \subset values(T?)$ we may have the following subtyping rules:

$$\frac{S \texttt{ extends } T}{S! <: T!} \qquad \frac{S \texttt{ extends } T}{S? <: T?} \qquad \frac{}{T! <: T?}$$

Downcasts from $T?$ to $T!$ require a *run time check*.

Because downcasts are inherently unsafe but needed if we every want to do something with a `null` type, we need to first check whether or not an instance of a `null` type is really indeed `null`. To prevent compile time errors for potential `null` dereferencing our type system will use a *dataflow analysis*.

Example 7.1 (Checking `null`)**.** Consider some `Map` class that represents a key-value map.

```
class Map {
    private Object! key;
    private Object! val;
    private Map? next;

    public Map(Object! key, Object! val, Map? next) {
        this.key = key;
        this.val = val;
        this.next = next;
    }

   public Object? get(Object! key) {
       if (this.key == key) return this.value;
       return this.next.get(key);
   }
}
```

`get` is not `null`-safe because `this.next` could be `null`. So we need to first check whether it is `null` before we use it as a receiver.

```
public Object? get(Object! key) {
    if (this.key == key) return this.value;
    if (this.next == null) return null;
    return this.next.get(key);
}
```

This seems like a reasonable solution, and in a sequential setting it could be. However interleavings are possible that allow some other thread to set `this.next` to `null` before `this.next.get(key)` is invoked. Thus we need to assign to a local variable first.

```
public Object? get(Object! key) {
    if (this.key == key) return this.value;
    Map? nextUp = this.next;
    if (nextUp == null) return null;
    return nextUp.get(key);
}
```

There is no way for another thread to set `nextUp` to anything. Thus a dataflow analysis is powerful enough to see that when `nextUp.get(key)` is called we can guarantee that `nextUp` won't be `null`. In general dataflow analyses are only powerful enough to reason about local variables like `nextUp`, not heap locations such as `this.next`.

7.2 Object Initialization

Constructing new objects poses many challenges when attempting to guarantee 7.1. Consider:

- We need to ensure that non-`null` fields are initialized to a non-`null` value when the constructor has completed.
- Non-`null` fields may be `null` before construction is finished.

Remember that in general it is okay for invariants to be violated in the middle of a function's execution. This may pose a problem especially in concurrent settings.

Definition 7.2 (Definite assignment rule)**.** Every local variable must assigned to prior to its first use, particularly as a receiver. For object types this is especially important to prevent `null`-dereferencing. This may be checked by a dataflow analysis and during bytecode verification.

Remark. *Languages such as Java and C# do not initialize declared variables, but do let you use them.*

We would like to extend definition:defassrule to fields as well, but this is difficult to enforce in constructors. Especially because subobjects will implicitly or explicitly invoke the superobjects constructor during their constructor.

Example 7.2 (Dynamically-bound problems.)**.** Consider the didactically named class `Demo`.

```
class Demo {
    private Vector! cache;

    Demo () {
        int size = optimalSize();
        this.cache = new Vector(size);
    }

    int optimalSize () { return 16; }
}
```

By this point, the OO guru in you is observing that calling a potentially dynamically-bound method, `optimalSize`, in the constructor is asking for trouble. You may also be asking why not make `optimalSize` a constant field instead of a method. Don't think about. Don't ask questions. As you would expect a subclass can cause problems.

```
class SubDemo extends Demo {
    private Vector! data;

    SubDemo (Vector! d) {
        // implicit call super()
        this.data = d.clone();
    }

    int optimalSize () { return this.data.size() * 2; }
}
```

Every time we attempt to invoke the constructor for `SubDemo` bad things will happen: we will get a `null`-exception. You may ask, where did we go wrong (other than that we used OOP). All receivers have non-`null` types!

1. When the constructor for `SubDemo` is invoked it first (implicitly) invokes `Demo`'s constructor.
2. `Demo`'s constructor's call to `optimalSize` is n by `SubDemo`'s.
3. In `SubDemo.optimalSize` we call `this.data.size()`.
4. But `this.data` has not yet been initialized to a non-`null` value!
5. So we get a `null`-pointer exception!

Example 7.3 (Callback problems)**.** Problems may occur if a class's constructor calls a method that call's one of its own dynamically-bound methods. Once we again have an appropriately named `Demo` class.

```
class Demo implements Observer {
    static Subject! subject;

    Demo () {
```

```
        // register this as an observer of this.subject
        this.subject.register(this);
    }

    void update () { ... }
}
```

Question: when does `this.subject` get initialized?

No dynamically-bound calls occur within `Demo`'s constructor. The `Subject` class looks like:

```
class Subject {
    void register (Observer! o) {
        ...
        o.update();
        ...
    }
}
```

However again we may write a subclass that causes problems:

```
class SubDemo extends Demo {
     private Vector! data;

     SubDemo (Vector! d) {
         // implicit super() call
         this.data = d.clone();
     }

    void update () {
        ...
        something = this.data.size();
        ...
    }
}
```

Once again a call to `SubDemo`'s constructor will throw a `null`-pointer exception!

1. `SubDemo`'s constructor begins by implicitly invoking `Demo`'s constructor.
2. `Demo`'s constructor invokes `this.subject.register(this)`.
3. In `Subject.register` `update` is called, which is overriden in SubDemo.
4. In `SubDemo.update` `this.data.size()` is invoked.
5. But `this.data` is has not been initialized to a non-`null` value so it is still `null`!
6. A `null` value as a receiver throws a `null`-pointer exception!

Example 7.4 (Concurrency woes)**.** So maybe the constructor doesn't even indirectly call one of its own Dynamically bound methods. Concurrent execution still has something to say.

```
class Demo implements Observer {
    static Subject! subject;

    Demo () {
        // register this as an observer of this.subject
        this.subject.register(this);
    }
```

```
    void update () { ... }
}

class SubDemo extends Demo {
     private Vector! data;

     SubDemo (Vector! d) {
         // implicit super() call
         this.data = d.clone();
     }

    void update () {
        ...
        something = this.data.size();
        ...
    }
}

class Subject extends Thread {
    List<Observer!>! observers;

    void register (Observer! o) { observers.add(o);  }

    void run () {
        for (Observer! o : this.observers) o.update();
    }
}
```

Here the problem arises from an unlucky interleaving. Consider the following execution.

1. `SubDemo`'s constructor is invoked.
2. The implicit `super()` call is invoked.
3. In `Demo`'s constructor `this` is registered as an observer.
4. In `Subject.register` out object under construction is added to `observers`.
5. Our constructor's execution is preempted by the thread calling `Subject.run`.
6. `update` is called upon our object under constructor, which is ally bound to that in `SubDemo`.
7. In `Subject.update` `this.data.size()` is invoked.
8. But `this.data` is still `null`, so a `null`-pointer exception is thrown.

Example 7.5 (Concurrent woes continued)**.** Okay. I give up. I will not make *any* method calls during a constructor at all. Here's an *innocent* class with a cycle of processes.

```
class Cycle {
    Cycle! next;
    Process! proc;

    Cycle (Process! p) {
        this.next = this;
        this.proc = p;
    }
```

```
    Cycle (Cycle! neighbor, Process! p) {
        this.next = neighbor.next;
        neighbor.next = this;
        this.proc = p;
    }
}
```

Imagine we have some scheduler that goes through a cycle.

```
class Scheduler extends Thread {
    Cycle! current;

    Scheduler (Cycle! cycle) {
        this.current = cycle;
    }

    void run () {
        while (true) {
            this.current.proc.preempt();
            this.current = this.current.next;
            this.current.proc.resume();
            Thread.sleep(1000);
        }
    }
}
```

Say we have some client code that creates process cycle for a scheduler, runs the scheduler, then decides to add another process to the cycle.

```
Process! eat   = new Process(...);
Process! poop  = new Process(...);
Process! sleep = new Process(...);

Cycle! eats    = new Cycle(eat);
Cycle! poops   = new Cycle(eats, poop);
Cycle! sleeps  = new Cycle(poops, sleep);

Scheduler dailySchedule = new Scheduler(eats);

// Spawn a new thread, calls dailySchedule.run()
dailySchedule.spawn();

Process! game  = new Process(...);

Cycle! games   = new Cycle(eats, game);
```

Here we create a daily scheduling that includes eating, pooping, and sleeping. Later we decide to add gaming to our schedule. This will be totally okay, right...? Imagine the following execution:

1. The `eat`, `poop`, and `sleep` processes are initialized without issue.
2. These processes are used to create a daily schedule `dailySchedule`, that first eats, poops, sleeps, eats, etc.
3. A new thread is spawned that runs the scheduler.
4. We decide to add a new process `game` for gaming to our cycle after eating and before pooping.
5. In the constructor for `Cycle` `eats` is passed for `neighbor` and `game` is passed for `proc`.

6. `this.next` is assigned to `eats.next` which is `poops`.
7. `neighbor.next`, which is `eats.next`, is assigned to `this`.
8. Uh-oh, we've been preempted by the scheduler!
9. For `dailySchedule.run`, at the moment of preemption `this.current` was set to `eats`.
10. Now `eats.next`, which is our object under construction, becomes `this.current`.
11. `this.current.proc.resume()` is invoked.
12. `this.current.proc` is the `proc` field for our object under construction, which has not yet been assigned and is still `null`!
13. A `null`-pointer exception is thrown!

Currently our type system for non-`null` types is not powerful enough to Statically guarantee that no `null`-pointer exceptions occur when a class field of a partially-initialized object escapes from its constructor.

It *cannot* ensure that during initialization that uninitialized fields are not used a receivers when

- The object under construction calls a dynamically-bound method.
- The object under construction passes itself or one of its uninitialized fields into another function that calls a dynamically-bound method.
- The object under construction or one of its uninitialized fields is stored in some data structure that concurrently that is available to another thread.

We may be able to solve this problem by prevent constructors from making dynamically bound calls, passing itself or any of its uninitialized fields into another function call, and synchronizing constructors. However this is much less expressive and Dynamically has more overhead. We would like to adopt a more fine-grained approach. To do this we need to consider the initialization phases for an object.

When a constructor is invoked, we can consider the following stages:

1. When `new` is invoked the object now exists any may be used, even though it has not yet been initialized!
2. At some point during the initialization process all of the object's fields become initialized.
3. The constructor call has completed execution.

Before step 1 the object cannot be used because it does not yet exist. Between steps 1 and 2 we cannot make any assumptions about the `null`ness of the object's fields but the object and its fields may be accessed. After step 2 we know that all of the object's non-`null` fields are not `null`.

We would like our type system to work based on understanding which stage the object's construction is at. For our purposes the second step is too fine-grained so we may simply consider that

1. When `new` is invoked the object now exists any may be used, even though it has not yet been initialized!
2. The constructor call has completed execution and all of its fields are initialized.

Between steps 1 and 2 the object and its fields occupy some nether state where they may be accessed but we know nothing about their `null`ness. I.e. for some field $this.f : T!$, $this.f$ may actually be `null` even though its type says otherwise. Our type system needs to be expressive enough to reason about the fact that 7.1 may be temporally violated within a call to a constructor. Thus the type $T!$ on its own cannot express this. Thus we need to introduce types for different stages of object construction. We call these *construction types*.

- Free types `free` T represents values of type T during object constructions, in particular for uninitialized fields.

- Committed types $\texttt{com}\ T$ represents values of type T after an object constructor has completed execution.
- Unclassified types $\texttt{unc}\ T$ represents values of type T that may be either free or committed.

Note that $\texttt{com}\ T$ is allowable, it just expresses fully initialized values of T that may be `null`. In general we take that $\texttt{com}\ T$ is synonymous with T.

We may introduce the following subtyping rules.

$$\frac{}{\texttt{com}\ T! <: \texttt{unc}\ T!} \qquad \frac{}{\texttt{free}\ T! <: \texttt{unc}\ T!} \qquad \frac{}{\texttt{com}\ T? <: \texttt{unc}\ T?} \qquad \frac{}{\texttt{free}\ T? <: \texttt{unc}\ T?}$$

Note that we must ban casts from unclassified types to free and committed types. Question: but why? What role do unclassified types have anyway?

Our new type systems needs certain requirements.

Definition 7.3 (Local initialization)**.** We say that an object is *locally initialized* if and only if its non-`null` fields are not `null`. If the Static type of a term is committed then its value at run time is locally initialized. See 1 for the possibilities for `null` reads of a field of some term.

construction	!	?
committed	!	?
free	?	?
unclassified	?	?

Table 1: Possible `null` reads for each combination of (non-) `null` and construction types.

$$\text{locally\_initialized}(o) = \Gamma \vdash o : \texttt{com}\ T! \rightarrow \forall f : T! \in fields(A) \rightarrow o.f \neq \texttt{null}$$

Local initialization is a *partial* property.

Definition 7.4 (Transitive initialization)**.** We say that an object is *transitively initialized* if all reachable objects from its fields satisfy 7.3. We would like to guarantee that if a term's Static type is committed then its value at run time is transitively initialized.

$$\frac{\text{locally\_initialized}(o) \qquad \forall f \in fields(o) \rightarrow \text{transitively\_initialized}(o.f)}{\text{transitively\_initialized}(o)}$$

Transitive initialization may be considered to be a property that holds for an *entire* object. A committed object is transitively initialized.

Definition 7.5 (Typing field writes)**.** We may type field writes as:

$$\frac{\Gamma \vdash e_1 : C_1\ A! \qquad \Gamma \vdash e_2 : C_2\ S \qquad f : T \in fields(A) \qquad S <: T \qquad C_1 = \texttt{free} \vee C_2 = \texttt{com}}{\Gamma \vdash e_1.f = e_2 : unit}$$

The condition $C_1 = \texttt{free} \vee C_2 = \texttt{com}$ is depicted for all combinations of construction types in 2.

	$\Gamma \vdash e_2 : \texttt{com}\ S$	$\Gamma \vdash e_2 : \texttt{free}\ S$	$\Gamma \vdash e_2 : \texttt{unc}\ S$
$\Gamma \vdash e_1 : \texttt{com}\ A!$	Ok	NO	NO
$\Gamma \vdash e_1 : \texttt{free}\ A!$	Ok	Ok	Ok
$\Gamma \vdash e_1 : \texttt{unc}\ A!$	Ok	NO	NO

Table 2: $C_1 = \texttt{free} \vee C_2 = \texttt{com}$

Definition 7.6 (Typing field reads)**.** we may type field reads as:

$$\frac{\Gamma \vdash e : C\ A! \qquad f : T \in fields(A)}{\Gamma \vdash e.f : \mathcal{F}(C, T)}$$

Where $\mathcal{F}$ is defined as:

$$\begin{aligned}\mathcal{F}(\texttt{com}, T!) &= \texttt{com}\ T! \\ \mathcal{F}(\texttt{com}, T?) &= \texttt{com}\ T? \\ \mathcal{F}(\texttt{free}, T\_) &= \texttt{unc}\ T? \\ \mathcal{F}(\texttt{unc}, T\_) &= \texttt{unc}\ T?\end{aligned}$$

$\mathcal{F}$ may be expressed as 3. At first glance it may seem like a type $\texttt{com}\ T?$ is a paradox: does such a type

	$f : T! \in fields(A)$	$f : T? \in fields(A)$
$\Gamma \vdash e : \texttt{com}\ A!$	$\Gamma \vdash e.f : \texttt{com}\ T!$	$\Gamma \vdash e.f : \texttt{com}\ T?$
$\Gamma \vdash e : \texttt{free}\ A!$	$\Gamma \vdash e.f : \texttt{unc}\ T?$	$\Gamma \vdash e.f : \texttt{unc}\ T?$
$\Gamma \vdash e : \texttt{unc}\ A!$	$\Gamma \vdash e.f : \texttt{unc}\ T?$	$\Gamma \vdash e.f : \texttt{unc}\ T?$

Table 3: $\mathcal{F}$ as a table.

violate 7.4? Such a type is actually okay because it is all right for a committed non-`null` type to have a `null` field. The idea is that a type being non-`null` is a property of a local variable, and it being committed is a property of the object that a local variable points to. When you read from free field it is unclassified (either free or committed). Fields of free refs are unclassified. When we read a field from a free reference its possibly null, so we need to check.

Definition 7.7 (Typing constructor declarations)**.** When we type the body of a constructor, we assume that `this` has a free, non-`null`type. By the end non-`null` values must be assigned to non-`null` fields. That is the constructor must have the explicit syntax `this.field = <not null>` for each field of a non-`null` type. This is called *the definite assignment check*. Constructors may either call free or unclassified methods

Definition 7.8 (Typing method calls)**.** The receiver of a method call must be non-`null`. Construction types for methods are those for receiver, so subtyping with them is contravariant.

How do we know when construction is complete? Naively we would say that construction is complete at the end of a call to a constructor. However this is not true in general: a subobject could just have finished calling `super()` but still has more to do. Knowing if object construction is complete is generally *non-modular*.

We might say that construction is complete when the `new` expression terminates. However this is only valid for *surface* calls to `new`. In other words, a constructor may call `new` within its declaration. This is particularly a problem if such a call to `new` contains terms that are free, such as `this`. In general objects created with `new` do not satisfy 7.4 when done. However they should always satisfy 7.3 when done.

Remark. *At the end of a call to `new` fields will have a strong update to their types to become committed.*

We must require surface-level `new` expressions to only take committed arguments. This is because assigning `this`'s fields to under-initialized arguments, i.e. not initialized or merely those that satisfy 7.3 is banned because there is no way that they will become committed or satisfy 7.4 by the end of construction. However we will allow `new` expressions invoked in constructors to take arguments of any construction type.

Thus we may make the following observations about what type of arguments `new` expressions may or may not allow. First concerning surface-level calls to `new`:

- The `this` under construction may only be given committed arguments to assign to fields. I.e, all arguments must satisfy 7.4.
- Any committed argument may not have a field assigned to `this`. This would violate 7.5 and the argument would no longer satisfy 7.4.

- An argument merely satisfying 7.3 is banned because it is not fully committed. Furthermore updating any of its fields to `this` is also forbidden because it would no longer satisfy 7.3.
- An argument that is uninitialized is forbidden because it is not fully committed.

Not let us consider nested calls to `new`.

- The inner `this` under construction may not have any of its fields assigned to *external* under-initialized values, as there would be no way to ensure that the final object satisfies 7.4.
- The inner `this` may have some of its fields assigned to the surface-level `this`, even though both are free.
- Once the inner `this` has completed its own construction, it satisfies 7.3.
- The outer `this` may assign one of its fields to the result of the inner `new`.
- Once all of the inner `this`'s fields have been assigned, it satisfies 7.3.
- Observe that all of these `new` objects when done only refer to objects that satisfy 7.4, or each other, which satisfy 7.3.

Thus at the end of surface-level constructor call the constructed object will satisfy 7.4.

Definition 7.9 (Typing `new` expressions)**.**

$$\frac{\textit{constructor}(A) : C_i'\ T_i \to A! \qquad \begin{array}{c}\forall i, \Gamma \vdash e_i : C_i\ S_i \\ \forall i, C_i\ S_i <: C_i'\ T_i\end{array} \qquad ((\forall i, C_i = \texttt{com}) \to C = \texttt{com}) \vee C = \texttt{free}}{\Gamma \vdash \texttt{new}\ A(e_i) : C\ A!}$$

Now in general methods need to be annotated with `free` to invoke them during construction. Such methods may not use `this` as a receiver.

Free and unclassified arguments to nested new must have come from within the surface constructor.

Example 7.6 (Declared vs actual types)**.** Formal construction type of parameter doesn't matter as much as static type of argument.

```
public C {
    public C (unc String c) {...}
}
```

We may do the following since committed types are subtypes of unclassified types.

```
String s = "Hello";
C x = new C(s); // ok, will know s is committed
```

Definition 7.10 (Lazy Initialization)**.** Constructing new objects with all of their data and fields may take time, and cause long waits at the start of an application. The idea here is to initialize fields *when they are first used*. This means that many fields may have `null`-types and thus every method that uses a particular field must check that it is indeed not `null` when it tries to use it. If a field is `null` the method is responsible for initializing it. This is a design choice, not a language construct. Consider we define a class aptly named `Demo` in typical fashion.

```
class Demo {
    private Vector! data;

    Demo () {
        this.data = new Vector();
    }
```

```
    public Vector! getData() {
        return this.data.clone();
    }
}
```

However if we choose to do lazy initialization our `Demo` will look more like:

```
class Demo {
    private Vector? data;

    Demo () {
        // does nothing
    }

    public Vector! getData() {
        Vector? d = this.data;
        if (d = null) {
            d = new Vector();
            this.data = d;
        }
        return d.clone();
    }
}
```

Definition 7.11 (Array Initialization)**.** It is difficult for a static analyzer to check in a decidable way if all non-`null` members of an array field are properly initialized in a constructor. Array initialization is typically accomplished with loops, but static analyzers have difficulties reasoning about loops. To ensure that all non-`null` members of an array are properly initialized one could:

- Assign the field directly rather than each member.
  `String![]! arr = {``Hey'', ``Baby''}`
- Have language support for putting default values into the array (Eiffel).
- Use a run time check at the end of the constructor.

```
free String![]! arr = new String![arrLength];
for (int i; i < arrLength; i++) {
     arr[i] = i % 2 ? "Even" : "Odd";
}
NonNullType.AssertInitialized(arr);
this.arr = arr;
```

Note that the type rule for array assignment forces right hand side to be committed.

Note that we do not allow overloads from differences in construction types. Checkout spec sharp.

Remark. *Some of the pitfalls of initialization may be avoided if the call to the super constructor takes place at end of sub constructor.*

7.3 Initialization of Global Data

Why are we still here? Just to suffer?

Kazuhira Miller

Definition 7.12 (Design goals for initializing global data)**.** We would like our solution to initialize global data to satisfy the following criteria.

1. **Effectiveness:** We would like to ensure that global data is initialized *before its first access*. For instance we would like ostensibly non-`null` to actually not be `null` when we use it?

2. **Clarity:** The semantics of initialization are simple to reason about.

3. **Laziness:** Global data is initialized *lazily* to reduce boot-up time.

Since we are in an OOP can you guess which of the above will *not* generally hold?

Definition 7.13 (Global variables and init methods)**.** Initialization of global variables could be done by *explicit* calls to `init` methods. These `init` methods must be called in directly or indirectly from `main`. If a global variable in module A depends upon a global variable from module B, `main` needs to somehow initialize B's global variables first. Thus it requires (the implementors of) `main` to know the dependency structure of imported modules. How did we do?

1. **Effectiveness, Strike:** Initialization order needs to be set *manually* by the programmer, and programmers are of course infallible.

2. **Clarity, Strike:** Need to know module dependencies requires non-modular reasoning and violates *information hiding*.

3. **Laziness, Meh:** Possible but also needs to be coded manually.

Two strikes (one strike is enough to be bad here)!

Example 7.7 (C++ Initializers)**.** This is more of a variation of 7.13 than its own conceptual solution. Here initializers are executed not by but *before* `main`. No explicit calls to these initializers by the programmer are needed. The order these are executed depends upon *the order they appear in the source code*. Thus the order must again be determined by the programmer. In C++ the programmer needs to declare modules in right order. How did we do this time?

1. **Effectiveness, Strike:** Initialization order needs to be set *manually* by the programmer, and programmers are of course infallible.

2. **Clarity, Strike:** Need to know module dependencies requires non-modular reasoning and violates *information hiding*.

3. **Laziness, Strike:** No support for this.

Three strikes and this attempt is out!

Definition 7.14 (Static fields and initializers (in Java))**.** `static` fields store references to global data. `static` initializers are executed by the system *immediately before a class is used at run time*. Thus initialization happens *lazily*. The run time systems manages dependencies between modules. However it is possible (in Java) for `static` initializers and fields in different classes to be *mutually dependent*. As you could expect, this could easily lead to `null`-pointer exceptions. Because `static` initializers may have arbitrary side effects reasoning about them is *non-modular*. To understand what value a `static` field will have requires knowing the order initializers are run, which requires executing the program. How did we do this time?

1. **Effectiveness, Strike:** `null`-pointer exceptions are incredibly likely.

2. **Clarity, Strike:** Need to know module dependencies requires non-modular reasoning and violates *information hiding*. Furthermore one needs to effectively execute a program to understand the order initializers will be executed, what values `static` fields will have, etc.

3. **Laziness, Yes:** Done at the last possible moment.

Two strikes! Laziness is just a wet blanket here. Basically, in Java `static` initializers calling `static` stuff from other class causes problems.

Example 7.8 (Scala Singletons)**.** Scala provides language support for singletons[1]. But because these are defined by translation to Java they are glorified examples of 7.14.

Definition 7.15 (Once Methods (Eiffel))**.** `Once` methods are executed exactly twice. Just kidding, they're only executed only once. The `Result` of the first call is cached and used for all subsequent calls. Thus for all subsequent calls the arguments are ignored. For recursive methods the current value of `Result` is used. How does this do?

1. **Effectiveness, Strike:** Reading initialized fields is possible.
2. **Clarity, Strike:** Need to know when `Once` methods were first called and with what values, may be cluttered by concurrency.
3. **Laziness, Yes:** Done at the last possible moment.

Two strikes! But what were we expecting from Eiffel?

Sadly there is no solution (that uses global data) that properly ensures (in particular non-`null`) global data is initialized before it is first used. This is especially complicated by mutual dependencies between modules.

8 Reflection

When calling `f.get(o)`, o may not have field f, or f may not be accessible, etc. Thus we need run time checks because Java cannot Statically guarantee that reflection method calls are okay. `setAccessible` can be disabled by security manager. `setAccessible` doesn't change class, just allows one to get a method of any visibility. Java cannot change established class code at run time, but Python can!

[1] A class with a single instance