# Generic Immutability and Nullity Types for an imperative object-oriented programming language with flexible initialization

James Elford

June 21, 2012

**Abstract**

We present a type system for parametric object mutability and reference nullity in an imperative object oriented language. We present a simple but powerful system for generic nullity constraints, and build on previous work to provide safe initialization of objects which is not bound to constructors. The system is expressive enough to handle initialization of cyclic immutable data structures, and to enforce their cyclic nature through the type system. We provide the crucial parts of soundness arguments (though the full proof is not yet complete). Our arguments are novel, in that they do not require auxiliary runtime constructs (ghost state) in order to express or demonstrate our desired properties.

# Contents

# Acknowledgements

I would like to thank my supervisor, Sophia Drossopoulou, for dedicating so much of her own time, for her original discussions, and for her patience, which have all been indispensable.

I would also like to thank anybody who spotted a typo or grammatical error while reading my work: though I have no doubt I will find one directly after submission.

## 1 Introduction

Historically, programmers have leveraged static analysis of their code to guarantee certain runtime properties of their programs. Statically-typed languages impose a type system which places restrictions and checks on which type of data can be accessed and used in which ways. In nearly all statically-typed languages, we use the type-system to guarantee that we pass correctly formed data into functions. In object-oriented languages, we also use it to guarantee that field-access and method-invocation on our objects are valid and – depending on the language – certain other statically-decidable properties, such as mutability.

Some statically-typed languages offer the programmer more guarantees than others: for example, C++ provides the notion of `const`, which prevents objects from being modified through certain access paths. The more recently developed D programming language provides the stronger construct of `Immutable` data – data which cannot be altered through any access path at all[12]. More recently still, the SPEC# language was developed, with – amongst other things – support for non-`null` reference types[3]. When we turn our attention to some of the most commonly used enterprise application programming languages, however, such guarantees are lacking: specifically, JAVA and C# lack support for non-`null` reference types, and while C# has support for compile-time constants via its `const` keyword, neither provide strong immutability guarantees for dynamically created objects.

While both immutability and non-`null`ity are desirable invariants for a type system to provide (for example, we can eliminate `null`-reference errors, or we can concurrently manipulate unchanging data structures without locking), there is currently no widely used language that has strong support for both immutability and nullity constraints. While there exist good solutions to extend JAVA-like languages with either (which we will discuss in Sections 2.2.3, 2.3 and 2.4), none of these systems addresses both topics together, and so none of them exploit the common overlap between the two: one of the most difficult concerns when formulating both immutability and `null`ity type systems is object initialization (the period before an object can possibly satisfy all of its invariants). In this project, we develop a type system that allows for both non-`null` reference types and immutability. In doing so, we create a type system that delivers a unified approach to the challenges of both, whilst delivering the expressivity of either.

Because both mutability and `null`ity concerns have implications for the initialization of objects, we believe it is helpful to treat them both together. We also wish to address parametric constraints: this has been successfully done already for immutability (c.f. [20]), but `null`ity constraints do not appear to have received the same treatment. Bringing these two object invariants into a unified system gives us an excellent opportunity to express them both in similar terms: we add parametric constraints to `null`ity types in the same way that it is common to treat mutability.

In systems that require objects to fulfil invariants that they cannot have when they are allocated (e.g. the non-`null`ity of all fields), the authors of type systems sometimes enforce those constraints at the end of a constructor (e.g.

[20], [18]). [21] introduced the concept of ownership to extend the initialization period of an object. Under the system presented there, it is entirely possible to initialize, for example, arbitrarily sized, cyclic, immutable structures. The drawback is that the creation of these structures is tied to the commitment point of some root node, their owner.

The main contributions of this dissertation are:

**Generic nullity** Genericity of `null`ity constraints is a novel construction, and we will see, when we discuss contributions in more detail later, that it offers expressivity not previously available.

**Flexible initialization** It is possible to create factories that produce cyclic data structures, or even to require that a data structure is initialized to be cyclic.

**A unified treatment of nullity *and* immutability** Under our system, initialization of `NotNull` fields must be (guaranteed) completed whilst an object's mutability constraints are not yet applied; there is no chance to create an object which still needs to have fields initialized, but is no longer mutable.

**Lightweight soundness arguments** We do not require any additional runtime constructs (erasable or otherwise) in order to express or show the properties of our system.

## 1.1 In this document

We begin by giving some background (Section 2) around the area of parametric types, immutability in existing languages, and existing systems for handling `null`ity. We discuss some of the more general aims of the project in Section 3, such as the need for modularity, expressivity, and so on. We describe in more detail some specific systems for mutability and `null`ity of the type we aim to achieve in Section 4, before proceeding with an informal description of the type system (Section 6), and finally a formalization of the type system and runtime semantics (Section 7) and of the properties that the system guarantees (Section 8). We discuss our contributions and possible further work in Sections 9 and 10. The impatient reader will find typing rules in Figures 39 and 40, and operational semantics in Figure 43. Subtyping rules are covered by Figure 37.

# 2 Background

The intention of this section is to provide a general introduction to the concepts involved: Parametric types, Immutability, and Nullity. The introduction to each will be more broad than we need for the purposes of our proposed type system – the intention is to provide background on the state of the field (for example, when we discuss parametric types, we will go into some detail on templates in C++, while for our type system, we will not be using features like metaprogramming). Where appropriate, practical examples of the way language features are used will be provided through code listings. Readers familiar with each each individual area are invited to skip ahead, and will not lose anything in doing so.

## 2.1 Parametric Types

Polymorphism is a powerful language construct, available in most modern languages. The idea is to allow us to use data of more than one type in the same way. It comes in a variety of flavours: static languages have long supported subtype polymorphism (a subtype is a *special kind of* the original type). Dynamic languages (such as PYTHON or JAVASCRIPT) use "duck-typing," which is a powerful form of polymorphism that does not require the consumer of an object to have any knowledge of its type[1]. This means that in PYTHON we could write code like:

```
def print_all(col):
        for i in col:
                print(repr(i))
```

The above defines a function that will accept any object, and for each item it contains (assuming that the object can meaningfully be said to *contain* anything), print out a textual representation of that item. This function will behave correctly for any object `col` over which the PYTHON interpreter can iterate. With flexibility, however, comes risk: there is no way to know, until we run the code, whether objects passed into the function *can* be iterated over. The function might work correctly in a production system for months before encountering such a case, but when it does, it might crash the whole program.

Duck typing is a powerful form of polymorphism, but for our purposes (guaranteeing *static* properties of a system, before runtime) we will not be making use of such flexibility. We will instead look to *parametric* polymorphism.

Parametric types, commonly known in object-oriented programming as "generics," allow us to leave part of a type signature for an object, method, or function undecided when we write it – preferring to allow the caller to decide. Parametric types do exactly what we would assume from the name: they allow us to specify *parameters* for our types (and, in doing so, allow us to program in a more *generic* fashion).

[1] "If it walks like a duck, quacks like a duck, and smells like a duck, . . . "

Since most readers will have some experience with generic programming, we will begin with a familiar example (written in Java), in Figure 1.

The listing gives a simple definition for a recursively defined `List` implementation. The thing to notice here, though, is the type declaration: we write `List<String>` to declare a list of `String`s. In doing so, we have parametrised the type of our list. Throughout the List's definition, we used a place-holder, `T`, but when we declare a usage of such a list, `T` is substituted with our choice of type. This means that we can write `l.get(i)` and know that it will return a `String`. This might not seem remarkable to begin with, but consider the situation if we could not specify that our `List` contains `String`s; every list would be list a `List<Object>`. In fact, this is exactly what we did, before `J2SE 5.0` (Java version 1.5). `l.get(...)` would return an object of type `Object`, and it would be up to the caller to know which concrete type they were expecting, and cast to it by hand:

```
Object entry = l.get(i);
// Let's just hope somebody didn't pass us a list containing
    Integers!
String str_entry = (String) entry;
```

This is the essence of parametric types in static languages. The following sections will look at two methods of implementing parametric typing: "Template Types," found in C++ and D, and "Generics," found in Java and C#. Either system can be used to produce, for example, convenient collection classes, but templates offer other features (e.g. "template metaprogramming") that are not present in generics.

### 2.1.1 Static Polymorphism through Templates

Templates deliver more power than we will require for our type system, but are interesting, because once their implementation is understood, parametric types are relatively simple, conceptually (although they can, in practice, still be difficult to use), but also extremely powerful. A templated class is so named because it provides a *template*, which must be completed during program compilation, with the appropriate "blanks" filled in. Every time we instantiate a templated class in C++, the compiler generates a *new* class, which contains all the appropriate substitutions. For example, consider the program in Figure 2.

When compiled, the two concrete usages of class `Box` become expanded, and the compiler writes new definitions to represent the two required instantiations (like in Figure 3).

These new definitions are known as "template specializations," can be thought of as entire separate classes – distinct from the template that generated them. The way that templated classes are expanded in C++ means that when a programmer instantiates a templated class with a concrete type, it is equivalent to having written out a whole class definition. This carries both advantages and disadvantages.

```java
public class List<T> {
    public static void main(String[] args) {
        List<String> l = new List<String>();
        l.add("a"); l.add("b"); l.add("c");

        for (int i=1; i<4; i++) {
            /* We know that l.get() will return a string, even
                though not all instances of List must contain
                strings: the type of l is parametrized when we
                write List<String> */
            String entry = l.get(i);
            System.out.println(
                Integer.toString(i) + "␣:␣" + entry);
        }
    }

    boolean hasItem;
    T item;
    List<T> next;

    public List() {
        this.hasItem = false;
    }

    private List<T> next() {
        if (this.next == null) {
            this.next = new List<T>();
        }
        return this.next;
    }

    public void add(T item) {
        if (this.hasItem) {
            this.next().add(item);
        } else {
            this.hasItem = true;
            this.item = item;
        }
    }

    public T get(int index) {
        if (index <= 1) {
            return this.item;
        } else {
            return this.next().get(index-1);
        }
    }
}

/* Output:
 * 1: a
 * 2: b
 * 3: c */
```

Figure 1: A simple recursive implementation of a List with a generic type parameter in Java

```
#include <iostream>

using namespace std;

template<typename T>
class Box {
    public:
        T item;
};

int main(){
    Box<int> a;
    a.item = 1;
    cout << a.item << endl;

    Box<const char*> b;
    b.item = "Hello, world";
    cout << b.item << endl;
}
```

Figure 2: Code showing a basic example of how templates can be used to create statically polymorphic code in C++

```

template<> // A concrete instantiation of the template
class Box<const char*> {
    public:
        const char* item;
};

template<>
class Box<int> {
    public:
        int item;
};
```

Figure 3: Code to illustrate how the templates from Figure 2 are expanded by the compiler to give concrete instantiations

**Disadvantages**

**Dependence on implementation** In order for the compiler to expand a call to a templated type into a concrete class, it will need to copy out large chunks of the template in full. For example, every time we want to store something in the `Box` container in the last listing, the compiler needs to have access to all the code needed to write the newly parametrized `Box` type. If the internal implementation details of the base `Box` class, change, then so do those of our concrete instantiation.

**Larger binaries** Every new concrete instance of the class must be stored as a new datatype in our compiled code (after being generated ahead of time).

**Debugging** Debugging errors in code that uses templates can be harder, because errors occur in code that the *compiler* has *generated*, rather than code that the user has written. Historically, error messages generated in such cases have been difficult for programmers to decipher.

**Verbosity** Prior to C++11, there was no type inference in C++. This meant that type signatures for objects had to be written out in full; this can become extremely verbose, and reduces code readability. This is also a problem in Java, where type inference is still very limited.

**Advantages**

**Structural Type Checking** Because every mention of class `T` in a class with `T` as a type parameter is replaced by a concrete class, we determine at compile-time whether a class has the required fields and methods. The implications of this will become clear when we discuss generics in Section 2.1.2.

**Metaprogramming** When the C++ compiler expands templates, it may in turn create further uses of the templates it has just expanded. To illustrate this, consider the recursive definition of a class that can be used to compute entries in the Fibonacci sequence, in Figure 4.

To instantiate `Fib<T>` with parameter `45`, the compiler must first expand the definitions of `Fib<44>` and `Fib<43>`, and so on. By providing `Fib<1>` and `Fib<0>` explicitly, we “specialise” the template (the compiler uses our specializations, rather than expanding its own), and provide a base-case for the recursion. Because all of this expansion occurs at compile-time, the overhead of actually calculating the Fibonacci sequence is shifted from run-time to compile-time; at run-time, looking up `Fib<45>::value` is no more costly than looking up any other compile-constant. On the other hand, `Fib<46>::value` will be unavailable at runtime.

The D language provides further meta-programming features, that we will not discuss here. Clearly, templates in C++ provide more functionality than if they were simply type arguments. Generics are a more limited concept, with

```cpp
#include <iostream>

template<int n>
class Fib {
    public:
    static const int value = Fib<n-1>::value +
                                     Fib<n-2>::value;
};

template<>
class Fib<1> {
    public:
    static const int value = 1;
};

template<>
class Fib<0> {
    public:
    static const int value = 1;
};


using namespace std;

int main() {
    const int i = Fib<45>::value;
    cout << i << endl;
}
```

Figure 4: Code demonstrating the use of template metaprogramming to calculate entries in the Fibonacci sequence

```
// Insite the list class, we replace T with Object
Object item;
...

        // The get method actually returns an Object, and the
        //  compiler inserts a cast to the correct type.
        String entry = (String) l.get(i);
```

Figure 5: Snippet to show type erasure in Java generics

their uses restricted purely to the parametrization of the types that include them (meta-programming and specialization, for example, are not available with generics).

### 2.1.2 Static Polymorphism through Generic Types

Generics come in many shapes and sizes, but we will discuss two commonly found implementations of generic types: those found in C#, and those found in Java. We will begin with Java, since the system is arguably simpler, but will focus on C#.

In both languages, generics are more limited than C++'s templates. They strictly specify a *type* parameter, and nothing more. We have already seen how they are used, c.f. Figure 1, but we have not discussed how they are implemented – which, in practice, has ramifications on their capabilities; we will see that generics in C# yield different results to generics in Java.

Historically, the Java programming language had no support for generics, until the release of JavaSE5.0. The way they are implemented now is relatively simple: the Java compiler performs *type erasure* [2] to create a new class that has the expected return types, method signatures, and field types, but is compatible with the Java 1.4 language.[2] The process is achieved by replacing the parametric type `T` in the generic class with `Object`, and then inserting appropriate casts to the required type (this is the process of *erasure*). The compiled byte-code[3] for calling the `get` method from Figure 1 can be thought of as identical to the byte-code resulting from the code in Figure 5 (when the class is instantiated with generic parameter `String`).

This simple translation allows type-safe parametrized types, without needing to modify the virtual machine (since the resulting types are compatible with the previous language specification, they would compile to byte-code compatible with the previous virtual machine). Note that, since here `item` has class `Object`, the parametrized class is unable to use the object in any way that it would not be able to use an `Object` – method calls or field lookups that could not be performed on an `Object` cannot be performed on `T`. This would be an extremely restrictive limitation – contrast with templates in C++, where we can perform

[2]Java 1.5 byte-code is *not* compatible with the Java 1.4 virtual machine; whilst generics compile to a compatible form, other 1.5 features do not (e.g. `enums`)

[3]Machine code for the Java virtual machine

```java
public class Printer<T extends Printer.IPrintable> {
    public static void main(String[] args) {
        new Printer<Page>().print(new Page());
    }

    public interface IPrintable {
        public void print();
    }

    public void print(T printable) {
        // We know that the object with parametrized type
        // T will have a .print() function, because it
        // must fulfil the interface IPrintable
        printable.print();
    }

    static class Page implements IPrintable {
        public void print() {
            System.out.println("Printing...");
        }

    }
}
```

Figure 6: We use interface restrictions to limit our parametric types to only be called with `T` satisfying certain constraints.

any method calls or field lookups that could correctly be performed on the type with which we actually instantiate the class. JAVA actually provides a mechanism to specify stronger requirements (and so make more assumptions) about `T`: we can require that the concrete type used in the type parameter fulfil certain *interfaces*. An *interface* specifies a set of interactions that must be available on an object of a given type:

```java
interface ILength {
        public int length();
}

class List implements ILength {
        ... // Any other implementation details of a List

        public int length() { // Required by the ILength interface
                ...
        }

}
```

Because `List` claims that it implements the methods in the `ILength` interface, we know (the compiler verifies) that it does in fact provide them. Programmers use these restrictions on parametrized types to allow us to have more sophisticated and flexible interaction with the objects for which the concrete type is not known when class is written – see Figure 6.

*Interfaces* in JAVA are also used in subtype polymorphism, but it won't be useful to cover that here.

C# generics are a more sophisticated construct. C#'s system of generics were introduced in [10] as an extension to the Common Language Runtime (CLR)[4]. Unlike in JAVA, C#'s generics involved changes to the underlying platform, to introduce parametric types as a first-class part of the type system. In fact, C#'s generics are a feature of the CLR, rather than of the language; they are introduced in [10] by making modifications to the CLR's intermediate language (which can be thought of in similar terms to the JVM's byte-code). The result is that the feature is *reified* into programs that use it; the generic typing information is a part of the compiled program, rather than a language construct that is converted away as in JAVA (where the parametric type information is replaced by casts). This means that the generic type information is available at runtime. [10] also made allowances for the use of value types in generics, whereas the implementation in JAVA did not – only reference types can be used in JAVA generics, because they must be castable to `Object` (which value types such as `int` or `long` cannot be).[5]

**Covariance** Covariance extends the notion of sub-typing to generic arguments and containers. To illustrate covariance in a familiar context, we will begin with array-covariance, as found in both C# and JAVA. Consider an array of `Integer`s. Since `Integer` sub-types `Object`, and arrays are covariant in JAVA, an array of `Integer`s will type-check whenever an array of `Objects` is required. Similarly, so will an array of `String`s, or anything else that sub-types `Object`. To illustrate the problem with this, the short code listing in Figure 7 will be most effective.

Whilst the program type-checks correctly in a JAVA compiler, it surely causes a run-time error when we attempt to assign a `String` into a list of `Integers`. The soundness of the JAVA type system is defeated by the covariance of array types. When Generics were introduced, stricter rules were applied: JAVA's generics do *not* allow covariance in the same way as arrays – see Figure 8.

In order to allow a type-safe implementation of covariance for generics, "wildcard" syntax was introduced in [19]. Wildcards allow us to express that, in some instances, we do not need to know about the concrete type with which a class is parametrized. Using a wildcard in place of a type parameter imposes certain restrictions (clearly, we cannot add elements to a list when we do not know what type of element it contains), but allows other freedoms: it is safe for us to ask the length of a list, regardless of the type of element it contains. Moreover, it is safe for us to *read* elements from the list; although we must concede that we know only that they sub-type `Object`. [19] distinguishes between the methods that it is safe for us to call, and those which are not. Informally, it does so by making the return-type of methods with parametrized type `T` as general as it

[4]The CLR is Microsoft's virtual machine, upon which the .NET infrastructure runs.

[5]JAVA and C# allow us to use value types as reference types by introducing the notion of *boxing* – which wraps the value type in a reference-type wrapper, which can be assigned to the heap.

```java
public class ArrayCovariance {
    public static void main(String[] args) {
        Dog[] dogArray = {Fido, Elly, Mutt};
        Cat[] catArray = {Felix, Whisker, Tiger};

        Object[] objArray1 = (Object[]) dogArray;
        Object[] objArray2 = (Object[]) catArray;

        for(int i=0; i<objArray1.length; ++i) {
            // We are attempting to put cats into our
            // array of dogs! This could cause chaos,
            // and the JVM's runtime checks will not
            // allow it. (A runtime error occurs, even
            // though the expression correctly type-checks
            // at compile time).
            objArray1[i] = objArray2[i];
        }
}
```

Figure 7: Arrays in Java are covariant. This is not type-safe.

```java
public class GenericCovariance {
    public static void main(String[] args) {
        ArrayList<Integer> intArray = new ArrayList<Integer>();
        intArray.add(1); intArray.add(2); // ...

        ArrayList<String> strArray = new ArrayList<String>();
        strArray.add("one"); strArray.add("two"); // ...

        // Compile error! Java knows that covariant generics are
            unsafe
        ... = (ArrayList<Object>) intArray;
        ... = (ArrayList<Object>) strArray;
    }
}
```

Figure 8: Generics are not covariant, to ensure type-safety

```
public class GenericCovarianceWithWildcards {
    public static void main(String[] args) {
        ArrayList<Integer> intArray = new ArrayList<Integer>();
        intArray.add(1); intArray.add(2); // ...

        ArrayList<String> strArray = new ArrayList<String>();
        strArray.add("one"); strArray.add("two"); // ...

        ArrayList<?> objArray1 = intArray;
        ArrayList<?> objArray2 = strArray;

        Object fromArray = objArray1.get(1);

        // Can guarantee that null sub-types whatever argument
        // the add method might be requiring in place of ?
        objArray1.add(null);

        // Compile error -- Cannot guarantee that 5 (Integer)
        // subtypes the concrete type of the ArrayList (which
        // is unknown):
        //
        // objArray1.add(5);

    }
}
```

Figure 9: Code showing the behaviour of unbounded wildcards in JAVA generics

can be (`Object`), while arguments of type `T` become as specific as they can be (by allowing only a single argument: `null` – which subtypes any possible type `T`). We can see an example of this in Figure 9.

Whilst it is useful to be able to inform the type-system (in a type-safe way) that we do not care for the concrete parametrization of a type, discarding it altogether is more restrictive than required. To that end, [19]'s system also allows us to place "bounds" on wildcards. These bounds allow for a more expressive covariance. If we write `<?  extends Foo>`, then the resulting type can only be sub-typed by classes that have a generic type parameter that subtypes `Foo`, as in Figure 10

The same restrictions apply as with un-bounded wildcards, but now we know that the most general type returned by in place of `?` will be `Mammal` (we still know nothing about how specific the underlying class is, so we are still limited to only passing *in* the `null` object in place of `?`s).

C# produces a similar effect via its `out` keyword when specifying a parametrized interface [17] (see Figure 11)

A type argument that is specified as covariant cannot be used as the argument to a method in C# – there can be no way to add to an `IEnumerable`.

Covariance will be relevant to our own system if we want to parametrize `null`ity and immutability constraints (one of our goals). When we discuss an

```
class Mammal { ... }

class Cat extends Mammal {...}

...

ArrayList<? extends Mammal> list = new ArrayList<Cat>();
...
/* We know anything in the list is a subtype of Mammal */
Mammal item = list.get(0);
list.add(null);

// Compile errors:
/* We don't know anything more specific */
Cat item = list.get(0);
/* We don't know the list doesn't want something more specific */
list.add(new Mammal());
```

Figure 10: A demonstration of how we can use wildcards in JAVA to achieve safe generic covariance

```
class List<T> { ... }
interface IEnumerable<out T> { /* get(...), size(), ... */ }

IEnumerable<object> objList = new List<string>();
```

Figure 11: Code showing C#'s equivalent of JAVA's wildcards

immutability extension to Java in Section 2.2.3, we will see that there are times when we need to take an upper bound of mutable and immutable (to express that we do not mind which of the two an object is). If we wish to treat mutability as a generic constraint, we will need to ensure that we provide a mechanism for covariance that is compatible with other generics in our system (if we offer any) – for example, through wildcards. This is the role played by `ReadOnly` in [6]. We will need to apply similar rules to `null`ity, since that will also be generic.

Now that we have some understanding of parametric types, we should consider immutability. Ultimately, we would like to be able to parametrize the immutability constraints of a class.

## 2.2 Immutability

In general, it is desirable for us to be able to talk about objects and know they will not (cannot) change. Immutability comes in more than one flavour; e.g. `const` in C++, `Readonly` in D, and Java's weaker `final` notion (which guarantees a form of reference-immutability). We will be concerned with strong immutability; the assertion (guaranteed by the type system) that an object can never be changed. Strong immutability is a very powerful concept, and is useful in several areas, for example:

**Concurrency** If we know an object's values cannot change, we can know that it is safe to access concurrently. Multithreaded operations on immutable objects reduce locking problems to trivial.

**Memory conservation** Modern virtual machines such as the JVM and CLR "intern" (store a single copy of, for the duration of an application's lifecycle) the immutable constructs that *are* a part of their respectively languages – specifically, `String`s, and in C# variables that are defined as `const`.

**Defensive programming** We can pass references of our immutable objects to other parts of our program (or even 3rd-party code), and know that our data will be left unmodified.

**Guarantee of properties** Once an immutable object is properly initialized, we know that it should never be able to leave such a state – this helps us to reason about the invariants of objects.

Immutability it not a new construct. The idea of having data in our program which does not (cannot) change at runtime is an attractive one, and has been available to us for some time. For example, in C, we might use `#DEFINE`s in our code, or simple String literals. These values are baked in at compile time, and give us a very basic form of immutability. What we want is a little more sophisticated; we would like to be able to talk about immutable objects that are initialized (and their properties are decided) at *runtime*. This, too, is a problem which various languages address in different ways.

**Functional programming languages** get immutability for free: because they do not maintain state (e.g. with variable assignment), mutation of a data structure impossible. This means that, when programming in a functional style, languages such as F# benefit with a minimum of effort from immutability constructs ([7]). Languages such as F# and OCaml, in practice, introduce non-functional aspects to allow the initialization of cyclic data-structures.

### 2.2.1 Immutability in C++

`C++` provides a keyword for the concept of Immutability, `const`. `C++`'s `const` is both powerful and flexible, allowing for different levels of immutability, according to the user's need. At its most basic, `const` guarantees that a variable cannot be changed (at least, not through the current alias). For example:

```
const int pi = 3;
pi = 4;                          // Compile error!
```

Assigning to a `const int` is forbidden by the type system at any point after the variable's declaration. Now recall that `C++` provides us with the more complicated construct of Objects:

```
class A {};
...
const A a();
a = A();                         // Compile error!
```

Again, if we try to assign to a variable that has been declared `const`, we get a compile error, just as we would expect. The language is sophisticated enough to go further:

```
class A {
    int a;

    public:
        void mutate() {
            ++a;
        }
};

...

const A a();
a.mutate(); // Compile error
```

This, also, is a compile error. `C++` recognises that the method `mutate` might alter the state of `a`, and prevents us from such a call at compile time. It would be impractical if we were never able to call any of the methods of an immutable object, so the language provides for that by allowing us to mark our methods as preserving object state (i.e. they are "pure" methods, depending only upon their input parameters, and without side effects). We call this "Object Immutability" (or, equivalently, "Value immutability," when not discussing Objects).

`C++` also provides us with another form of immutability: "Reference Immutability." Here, immutability refers to our handle on the object, rather than the object itself. To illustrate, the following is perfectly legal:

```
...
const A *a = new A();
a = new A();
```

Here, we have *object* but not *reference* immutability. As before, mutating the object itself is illegal (whilst as we this shows, mutating our reference to it is not):

```
...
const A *a = new A();
a->mutate();                        // Compile error!
```

Now, if we employ C++'s reference immutability, we can safeguard our reference (whilst allowing the object itself to mutate):

```
...
A *const a = new A();
a->mutate();                        // Legal
a = new A();                        // Compile error!

// Can also combine the two:
const A *const b = new A();
b->mutate();                        // Compile error
b = new A();                        // Compile error
```

By combining both kinds of immutability, C++ gives us powerful tools to create read-only structures for use in our programs. `const` works as expected in C++'s templates, and is one of the more complete systems for creating read-only data structures. The immutability that this language provides has two major limitations:

- C++ does not *really* provide immutable structures; only immutable aliases to those structures (see Fig. 12). We can, in general, access supposedly "immutable" structures through mutable references, and mutate them as normal. This means that C++ programs are not able to take advantage of some of the guarantees that a stronger system of immutability would grant us. For example, we cannot guarantee that when we pass a `const` reference to an object to a function, it does not re-alias that reference as mutable, and modify it.

- Initialization of complex structures is difficult, since fields marked `const` must be initialized before an object's constructor. For example, consider attempting to initialize a cyclic list with immutable (both value and reference) references from each node to the next and previous nodes. In practice, this is difficult to achieve whilst maintaining `const`-correctness. (Although since `Foo` subtypes `const Foo`, we can construct the data structure as normal, and then cast to a `const` version, to simulate the initialization of an immutable data structure).

A third limitation is that, in practice, the syntax for using `const` is clumsy and often confusing.

C++'s `const` allows us to express that, through a given alias to an object, a data structure cannot be changed. It does not allow us to express that the data structure cannot change through *any* alias, which we will see that systems for true immutability (such as Immutability Generic Java, in Section 2.3) do. Aliasing considerations aside, `const` does give us that an object's state cannot change; that is, any of its fields that are of non-pointer type also become immutable (we have deep object immutability).

### 2.2.2 Immutability in Java and C#

We should now consider Java and C#'s immutability constructs. Java provides us with the far less powerful keyword `final`. `final` denotes *reference* immutability. `final` is straightforward: a local variable declared `final` cannot be assigned to a different object, whilst a field declared `final` *must* be initialized before the end of the object's constructor, and cannot be re-assigned for the duration of the object's lifetime (see Figure 13). It offers us more flexibility than `const` does in C++ (for example, we have the entire body of a constructor to initialize our reference-immutable fields, rather than being required to do so before the constructor's body can begin), but in turn gives us a less powerful system (we have no way to create an object whose fields cannot be mutated – unless the same is true for *all* objects of this type because their fields have been marked `final`).

C# provides a similar construct by means of the `readonly` keyword. The two are almost identical, with the exception that Java's `final` allows for constructor-independent initialization of fields (i.e. fields may be initialized *before* the constructor, in which case they may not be modified within). C# also has the keyword `const`, which has a different meaning than in C++. In C#, it denotes a value which is known at compile-time, and must be statically resolved. For example, consider the following:

```
class C {
    public const string Field = ''literal''; // Legal
    public const string Mistake;             // Error
    ...
        // in a method:
        Field = "some other literal"         // Error
}
```

Whilst Java and C#'s mutability constructs appear less complete, we should note that they do, in one way, offer slightly more: it is difficult (though still possible [14], [11]) to force a `final` or `readonly` field to behave as a normal mutable field.

### 2.2.3 An extension to Java's immutability model

We will discuss an extension to the Java language that promises immutability. [6] introduces mutability constraints through annotations of `Rd`, `RdWr`, and `Any` –

```
#include <iostream>

using namespace std;

class A {
    public:
        const int i;
        A(const int i) : i(i) { }
};

int main() {
    const int i=1;

    // a should be immutable
    const A a = A(i);

    // get a pointer to the const int
    const int * j = &(a.i);

    // At this point, according to the type
    // system, i, a and the value at j are
    // all const (immutable)

    cout << "i:\t" << i << endl;      // 1
    cout << "a.i:\t" << a.i << endl; // 1
    cout << "j:\t" << *j << endl;    // 1

    // This would be a compile error, since
    // j is a pointer to a const int
    //*j = 2;

    // But we can cast away the const, and
    // mutate the "immutable" data
    *((int*) j) = 2;

    cout << "i:\t" << i << endl;      // 1

    // a's field has changed
    cout << "a.i:\t" << a.i << endl; // 2
    cout << "j:\t" << *j << endl;    // 2

}
```

Figure 12: Code listing demonstrating that C++'s immutability is only weakly enforced by the type system; there are no guarantees that the underlying data cannot be modified through some alternative, non-`const` alias.

```
class A {
    public int i;
    public void mutate() {
        ++i;
    }
}

class B {
    public final A a;      // Must be initialized B's constructor
    public B() {
        this.a = new A(); // Removing this line is a compile error
    }

    public static void main(String[] args) {
        final B b = new B();
        b.a = new A();     // Compile error; b.a is declared final
        b.a.mutate();      // Perfectly legal.
        b = new B();       // Compile error; b is declared final
    }
}
```

Figure 13: Code listing illustrating the proper use of JAVA's `final` keyword

to represent, respectively, objects that can be *read*, *read/written*, and their least upper bound (used when we don't know which of the two mutability constraints is applied to an object).

The important goal of [6] is that "*Well-typed programs never write to fields of Rd-objects.*" Such a constraint expresses exactly what we would like from a system of immutability; the state of an immutable object cannot change. Immutable object initialization is achieved by forbidding the escape of certain references – [6] uses "*stack local*" memory regions. A *stack local* region of memory cannot be referenced by other locations on the heap: all references to objects inside that region must be on the stack, and the region is said to be owned by the lowest (least recent) method on the call-stack that holds references to that region. The mutability constraints of all objects in that region can be set by that owning method (they must all be the same) – and start off as writable. When such a change occurs, the method owning the region must be on the top of the call-stack, so all references to objects inside the region are local variables of that method. The result is that we end up with a specific point in the program at which the mutability constraints on an object can be said to be *committed* – that is, initialization is finished, and mutability rules should be enforced. The model is similar to that of [4]'s delayed types (which are used for nullity constraints, and which we will come to later). Without introducing the new notation formally, we write:

```
// Require a new stack-local region on the heap
newtoken(n);

// f lives within our stack-local region
Foo f = new <Fresh(n)> Foo();
```

```
// We can freely mutate the fields of f
f.x = 4;

// Set objects within our stack-local region to read-only
commit Fresh(n) as Rd;

// Error -- f is read-only
f.x = 5;
```

Writing `new <Fresh(n)> Foo();` requires that the new instance of `Foo` be treated as uninitialized, and stored within the protected region[6] that we requested with the call to `newtoken(n)`.

The initialization of `f` is protected within the stack-local region of the heap, and cannot be referenced by any other part of the heap until it becomes committed. Once it has been committed, it can be stored in read-only fields, and it is safe to reference from other parts of the stack.

### 2.2.4 Parametric Immutability Constraints

**In C++** Mutability is a part of the type. This means that, when we come to handle mutability in parametric types, the `const` part of the type is expanded into a template in the same way as the rest of the type parameter. Figure 14 demonstrates the way that immutability expands into templates; the compile error when we try to assign to the `item` field of an existing node in the list comes because, when the list is instantiated with `T` as `const int`, then the field `item` has type `const int`, and cannot be modified after instantiation. The list itself can be mutated, but the data stored within it cannot. We cannot, however, parametrize the list in such a way that the structure cannot be altered, while the data can. Whilst writing `const List<int>` seems to achieve that effect, in fact it only makes the `List` part of the data structure immutable; it is still possible to mutate the structure of the underlying collection of `Nodes`, given a reference to the `sentinel`.

For interest, readers might wish to note that the following program will fail to compile (attempting to store immutable data in a `std::vector`):

```
int main() {
        std::vector<const int> immutableInts;
}
```

**In Java and C#** There is no support for the parametrization of mutability constraints within a class. The `final` and `readonly` keywords are not a part of the type, and are not allowed as arguments for generic parametrization.

[6]Protected in the sense that no references to any object inside it can appear below this point on the stack, or anywhere on the heap outside of this region

```
#include <iostream>
#include "list.hpp"

int main() {
    List<int> list;
    list.add(1); list.add(2); list.add(3);

    for(int i=0; i<3; ++i) {
        std::cout << i << ": " << list.get(i) << std::endl;
        list.set(i, i*i);
    }

    for (int i=0; i<3; ++i) {
        std::cout << i << ": " << list.get(i) << std::endl;
    }

    List<const int> list2;
    list2.add(1); list2.add(2); list2.add(3);

    //list2.set(1, 10); // Compile error!

    return 0;
}
```

Figure 14: Code listing demonstrating immutability constraints on parametrized classes (see Figure 15 for class definitions)

## 2.3 IGJ: Immutability Generic Java

[20] introduces a system of parametric immutability constraints on top of standard Java. The aim is to be able to express mutability constraints in a more flexible way than is available in, for instance, templated C++ code. In [20]'s system, mutability constraints are parametrized just as any other part of the type: all types gain an extra generic parameter, and this (their first parameter) dictates their mutability. The parameter can have one of three values (recall the discussion of [6]): `Mutable`, `Immutable`, and `Readonly` – which denote, respectively, mutable objects, immutable objects, and their common ancestor in the type hierarchy. An object can never concretely *be* `Readonly` – the annotation simply means that we don't know at the call site whether it is `Mutable` or `Immutable` (so we must treat it with the restrictions of both). The system presented by [20] is extremely annotation-heavy, but [21] (also by Potanin et al.) presents a system with similar capabilities and a more succinct syntax.

The following would be allowed under both [6] and [21] systems; they give us reference immutability on the fields of immutable objects, *not* deep object immutability (although this can certainly be achieved in [21] through parametric specification of the mutability of fields):

```
public static mutateNestedField(Main<Imm> immutable) {
        // Legal
        immutable.mutableField.mutate(in, some, way);
}
```

```
template <typename T>
class Node {
    T item;
    Node<T> *next;

    public:
        Node(T item) : item(item), next(NULL) { }
        void add(T item) {
            if (NULL == next) {
                next = new Node<T>(item);
            } else {
                next->add(item);
            }
        }

        T get(int index) {
            if (index == 0) { return item; }
            else { return next->get(--index); }
        }

        void set(int index, T newValue) {
            if (index == 0) { item = newValue; }
            else { next->set(--index, newValue); }
        }

};

template<typename T>
class List {
    Node<T> *sentinel;

    public:
        List() : sentinel(NULL) { }

        void add(T item) {
            if (sentinel == NULL) {
                sentinel = new Node<T>(item);
            } else {
                sentinel->add(item);
            }
        }

        T get(int index) const {
            return sentinel->get(index);
        }

        void set(int index, T item) const {
            sentinel->set(index, item);
        }
};
```

Figure 15: Templated classes used to demonstrate parametric immutability in Figure 14

Initialization of immutable data structures is difficult for obvious reasons: we will often need to write to immutable fields in this stage of an object's lifetime. To help us with this, IGJ allows a fourth mutability type: `AssignsFields`. This level of mutability is available inside constructors, and like `ReadOnly` cannot be assigned to an object directly. Objects in this state of mutability cannot be stored in fields; what this gives us is a level of immutability that can only occur in an object under construction – thus allowing for the initialization of immutable objects (whilst limiting it to the constructor). In [21], OIGJ makes the initialization of immutable objects more flexible through ownership.

To illustrate what this system gives over C++, consider that in C++, the only available mutability constraint (`const`) is coupled to the type parameter. Whilst we can construct a list of `const int`s, we cannot construct a list of type `List<Immutable, int>` – which would allow us to use the mutability parameter in other parts of the data structure, perhaps in parts that were not dependent on the `int` part of the parametrization.

Our next section is a discussion of nullity constraints – the other major area we wish to address.

## 2.4 Nullity

A striking feature about nearly every language with support for reference-types (Java, C#, C++, to name just a few) is the presence of a singleton value which transcends type boundaries: in the languages just mentioned, it is called `null`. A `null` reference can be used in place of any reference-type object (in Java, this means anything except a primitive), and C# has explicitly added support for `null`able value-type objects via `int?`, `string?`, `float?` and friends. Because `null` transcends type boundaries, we saw in Section 2.1.2 that it can be "safely" used any time an object of any type is required. The problem with this is that there is no way, through the type system, to specify that it is *not* safe to specify `null` in place of an object. This is the goal of nullity constraints – which languages such as Java, C# and C++ do not have. Efforts have already been made to address this situation through both theoretical work (which we discuss here), and through practical steps: the Kotlin ([9]) and Ceylon ([16]) languages (new languages which target the Java virtual machine) both have support for not-`null` fields of objects, and promise not to throw `NullPointerException`s (the familiar response from Java when a program attempts to dereference a `null` pointer) when objects are fully initialized. Both require initialization in an object's constructor (or a similar construct), and we will see when we discuss [18] that this approach is too limited for our purposes.

A `null`-pointer de-reference happens when a program attempts to access a field or method of the `null` object. Because `null` is considered a valid value by the type system for any object of any type, the only way for a programmer to ensure that an object passed into a function (or, indeed, any field of any object passed into a function) is not `null` is to explicitly check:

```
if (foo != null) {
    doSomethingWith(foo.someField);
}
```

Such checks are inconvenient, and in many practical cases considered unnecessary; calling code is generally expected to know in advance whether it is safe to pass `null`-references into the arguments of functions. The result of this “assumed prior knowledge” is that programmers make mistakes. With no way to express through the type system the expectations of calling code with regard to the `null`ity of a function’s arguments, we – as software engineers – often get it wrong. Even when we are conservative, and don’t directly pass `null`-references into functions that we call, the inability of the type system to express these constraints leaves us open to mistakes when it comes to the `null`ity of the objects reachable through our function’s arguments; consider:

```
public void myFunc(Foo foo) {
        foo.someField.someMethod();
}
```

In this case, the caller has to not only know that the argument to `myFunc` `foo` cannot be `null`, but also to know how to properly initialize an object of type `Foo`, such that none of its fields are left uninitialized.

Consider now, using an exclamation mark (`!`), as in [18] or [4], among others, after a type signature to indicate that for an object to type-check against that type, they must not be null:

```
class Foo {
        Bar! someField
}
...
public void myFunc(Foo! foo) {
        foo.someField.someMethod();
}
```

In this listing, the function `myFunc` is guaranteed not to cause a `null`-reference error; its argument, `foo`, cannot be `null`, and the field `foo.someField` can also not be `null`. It is thus safe to de-reference not only `foo`, but also `foo.someField`, and so we can call its fields and methods with impunity.

Strong efforts have been made to allow programmers to specify the non-`null`ity of types in Java-like languages already. Works such as [5], [18] (“Freedom Before Commitment” – FBC), and [4] (“Delayed Types”) decorate non-`null` fields to mark them as such (e.g. `Boo!` for a non-`null`able field of type `Boo`) – the main challenge of these systems lies, again, in initialization. But why is initialization difficult for non-null types?

The reason is that all fields of an object are – when a constructor is entered – initialized to `null`. This means that in a type-safe system for non-nullity, the type checker must be responsible for two things: ensuring that all non-`null`able fields have been initialized at some point, and ensuring that there is no attempt to access a non-nullable field in an unsafe way before that point. In practice, such schemes, like schemes for immutability, allow more flexible constraints on objects under initialization, whilst posing restrictions on how they can be used (e.g. an object currently under construction cannot be assigned to a field of an object for which initialization has finished in FBC), by assigning them a new type that reflects their uninitialized status (e.g. FBC labels objects in currently-executing constructors `[Free]`). Objects under initialization are not assumed to fulfil their invariants – so a non-`null` field of a `[Free]` object may in fact be `null`.

Whilst [18] presents the most concise approach (needing very few annotations, and resulting in a usable system for non-`null` fields), is it significantly less flexible than that of [4] (delayed types). FBC requires that all objects satisfy their non-`null`ity constraints by the end of their constructors (although they can be treated as possibly-uninitialized after that point). Whilst objects are allowed to escape their constructors as `[Free]` references, we cannot use auxilliary methods to initialize them, and we cannot continue their initialization after the end of their constructors. Initialization of cyclic data-structures is defeated by this scheme: it is impossible to assign a non-`null` back-reference to the final element in a cyclic list before the first constructor is over (see Figure 16 for an illustration of the problem, and Figure 17 for a compatible work-around). Whilst there is a work-around for a simple recursively-structured list (that is, a homogeneous data structure, where we can hoodwink the type system by using a reference to `this`, which is always available), it is perfectly conceivable that a programmer may need to construct mutually-referencing objects for which this trick cannot be applied (e.g. an object that needed a non-`null` reference to another type of object, with neither being fully constructed until it could store a non-`null` reference to the other – that is, a heterogeneous cyclic data structure).

[4] introduces more flexibility here by placing the point at which initialization must be complete *outside* the scope of the constructor – in a way very close to [6]. In effect, delaying the point of commitment until outside the constructor places the responsibility for initialization onto the caller, rather than the object itself. Methods are then decorated with information about which fields of an object they will guarantee to initialize.

The advantage of FBC is that it is clear, concise, and importantly modular; all initialization is the responsibility of the object being initialized, and not the calling code. The advantage of Delayed Types is that they are more flexible. Both systems are proven sound by their respective authors, but while FBC also has a sound type-checker implemented, Delayed Types has not. Both systems use the concept of "commitment" to mean a concrete time at which (deep) initialization of an object is finished, and from which point onwards we can safely rely on the invariants promised by the `null`ability of its fields' types.

An obvious omission from the non-`null` type systems mentioned so far is parametrized nullity constraints. Whilst both systems will allow them in the same sense that C++ templates allow parametrized mutability constraints, neither delivers the freedom that OIGJ ([21]) does for immutability – and nor does any system for `null`ity that we are aware of.

```
class Node {
    // In a cyclic list, we always need
    // a previous node
    Node! previous;

    // Similarly, we must always have a
    // previous node passed into the
    // constructor if we are part of a
    // cyclic list
    public Node(Node! previous) {
        this.previous = previous;
    }

    public static void main(String[] args) {
        // In the first instance, there can be
        // no previous node!
        Node! first =
                new Node(/* What do we put here?!*/);
    }
}
```

Figure 16: Code listing demonstrating the difficulty in initializing structures in which the non-null fields of objects are mutually dependent.

## 2.5 Areas of commonality

This concludes the discussion of immutability and nullity – ideas are about introducing further invariants to existing type systems in order to increase expressiveness and safety of programs. The shared challenge is that of object initialization: in the case of nullity, the challenge is to ensure that all fields of an object have been properly assigned to, while in mutability, the challenge is to allow the short-term mutability of otherwise immutable objects, without exposing their mutable forms to the wider world. We will see in section 4 that it is beneficial to be able to complete the initialization of an object's not-`null` fields outside of its constructor, but this pattern requires objects to remain mutable for that time also.

Current work on mutability seems to have embraced parametrization (c.f. [20], [21], [6]) while work on nullity seems, so far, to be limited to un-parametrized `null`ity constraints (e.g. [18], [5], [4]).

Greater flexibility has been achieved by generalizing the period of initialization to a "commitment point," which may by outside the scope of an object's constructor. [21] leverages ownership to provide an external point of commitment for objects, whilst [4] and [6] provide explicit commitment points within the code.

```java
class Node {
    Node! previous;

    // Initialize the first node
    public Node() {
        // FBC forces us to initialize this non-null
        // field , but we can have no "previous" field
        // when we initialize the first Node
        this.previous = this;

        // This is a trick; we do this to satisfy the
        // type system, not because it expresses
        // the intention of the program.
    }

    public Node(Node! previous) {
        this.previous = previous;
    }

    public static void main(String[] args) {
        // No "previous" node yet
        Node! root = new Node();
        Node! second = new Node(root);
        Node! third = new Node(second);

        // Complete the cycle
        root.previous = third;
    }
}
```

Figure 17: A work-around for initializing cyclic data-structures under [18]

# 3 Goals

We briefly cover the aims of this project in order to give a broad context to the more specific discussions of existing systems in the next section.

Our most important aim is to produce a system that supports mutability and `null`ity in a sound way, whilst allowing flexible initialization. Introducing a generic system of `null`ity is an important challenge, and ultimately we hope that the language and type system are as expressive as the other systems which tackle the same problems.

We should also consider the aims of the authors of other systems: we will consider the aims of [18] (Freedom Before Commitment, FBC), and [21] (Ownership Immutability Generic Java, OIGJ).

FBC was intended to be suitable for mainstream use, and consequently was designed with four main goals in mind:

- Modularity,
- Soundness,
- Expressiveness,
- Simplicity

The reasons for each should be clear – modularity means that we can type-check a class without concerning ourselves with how it is used by other code (e.g. sub-classes, or any other code that uses the class itself); soundness guarantees that any program that can be type-checked can execute without causing type errors (e.g. incorrect method calls); while expressiveness and simplicity speak for themselves, but are particularly important when designing a language (or language variant) for mainstream consumption: as programmers, we are often extremely reluctant to learn complicated new syntax, or use languages that don't allow us to express ideas that others might.

In [18], Summers et. al. note that FBC is in some ways less expressive than alternative non-null type systems, but gains simplicity as a consequence. They also performed an experimental evaluation, applying their type system to a large existing body of code ("*SSCBoogie*," a SPEC# verifier), having implemented a type-checker in a modified version of the SPEC# compiler. We will not attempt such an experimental approach, since implementing a type checker will be outside the scope of the project, but we will provide code examples to show that our system can express complex data structures and initialization patterns.

In [21], Potanin et. al. also implemented a type-checker, then annotated the collections from the JAVA standard collections. They note in their introduction that no refactoring of existing code was required after annotations were added, and conclude that their system is expressive enough for most purposes.

In presenting a type system that expresses (im)mutability and nullity information, along with flexible initialization, we will favour expressiveness over simplicity. The annotations required to write classes in the type system presented may be extremely verbose, but we prefer to focus on the problem of the ease with which these complex objects can be initialized and used. If there is opportunity, we aim to simplify the annotations through a series of defaults (e.g. objects are assumed committed unless the user states otherwise), but we see that as a nicety here, rather than as essential.

Whilst we will not consider *syntactic* simplicity, we should consider semantic simplicity (is the system conceptually easy to use?). While we will consider this in our evaluation, we should note: data-structures with parametric mutability and nullity constraints are fundamentally *not* simple ideas. With this in mind, it is quite expected (and acceptable) that a type system describing them is also not simple.

The above focuses on our requirements for the constructed type system from a practical point of view. We would also like for our formulation of the type system to provide a unified approach to the two separate concepts in a way that allows them to compliment each other with shared concepts (notably, the initialization phase of an object's lifetime). We aim to minimise the extra work required from the system in order to satisfy our goals (e.g. do we need to have separate ideas of when an object becomes initialized for `null`ity and mutability constraints? Do we need new run-time constructs?).

# 4 Existing systems in more detail

We should spend some time discussing some existing systems for nullity and immutability in more detail. In particular, we will look at the initialization section of an object's life-cycle; this section is perhaps the most complicated, since it is the time when invariants on an object must first be established.

## 4.1 The initialization problem

So far, we have noted that initialization is difficult, but we haven't explored *why*. Systems for both `null`ity and mutability spend some time on the problem, but the challenge for each is different. In the case of `null`ity, the problem is that, when an object is first allocated, it is likely *not to satisfy* its invariants. In the case of mutability, the problem is that if we *enforce* the invariants, it will become difficult to complete the object's initialization with respect to its other properties (e.g. `null`ity, although this also applies to properties expected of an object that are not a part of the type system).

In order to make the problem concrete, we should consider an example: we will define two classes, `Pet` and `Owner`. Each `Pet` will have a single, unique `Owner`, and `Owner` will be a special case of `Person` – one which has a `Pet`. To represent this specification in a real program, we will need to construct two mutually dependent objects. Traditionally we would write something like in Figure 18. Clearly there are times when such mutual dependencies occur in real programs, and the steps we could take to make the initialization appear cleaner often defeat good design practices (e.g. if we were to create the `Pet` in the `Owner`'s constructor, we would defeat dependency injection). Note that the initialization of these objects is not "safe;" the type system guarantees neither that all `Pet`s will end up with `Owner`s, nor that all `Owner`s will be given `Pet`s. Moreover, for the sake of example, we contrive to require "loyalty" from both our pets and our owners; a faithful dog will not desert its master, and a good master will not abandon his loyal companion. So we have two new requirements:

- That the `owner` and `pet` fields of `Pet` and `Owner` (respectively) are non-`null`.
- That they are also immutable.

In Figure 18, we did not initialize *either* of these fields within the body of each object's constructor. We can do a little better than that by mandating that a `Pet` be created after an `Owner`, and requiring that an `Owner` be passed into the `Pet`'s constructor. But how can we properly initialize the `Owner`? If our type system truly guarantees the mutability and `null`ity promises we would like it to at the end of a constructor, then we will certainly want it to give a type error in Figure 18 (we cannot satisfy the `null`ity requirements within the constructor, and we do not respect the mutability constraints outside it). Even if we make the modification we've just mentioned, we will be unable to satisfy the `null`ity and mutability requirements of *both* objects. Our options are limited:

- Weaken the type constraints on one of our fields. This is the simplest solution, but also the least desirable; our type system is rather impotent if we can only use it to guarantee mutability and `null`ity constraints in *some* places, but for the sake of initialization we must give up this privilege in others.
- Come up with some scheme for initialization that will allow us to satisfy our type system's requirements with regard to the invariants on an object, allowing that the constraints of the type system might be too rigid while the object is still under initialization.

## 4.2 What do we require from initialization?

We require two things from the initialization of an object:

- That when initialization is "finished," (the object is "committed") it satisfies all of its invariants. This includes both those that are part of the type system (e.g. `null`ity constraints), and also those required by the particular application (so the user should be able to perform arbitrary initialization).
- That it is *possible*, within the constraints of the type system, to fulfil (and check that we have fulfilled, at the pint of type-checking) all the invariants.

Often, we consider the initialization phase of an object to be only its constructor. This is because, in general, this is the only code that we can guarantee will be run by the client of our class. In C++, we actually require that some initialization is done before the constructor is entered; the initialization of `const` fields, for example. As we have seen though, the constructor alone is not enough:

- Sometimes it is not possible to perform all the initialization that an object requires from within its constructor. We generally rely on client code to know about this, and perform the required extra steps. Whatever system we design should be flexible enough to allow this pattern of initialization.
- Even when it is possible to perform all required initialization within the constructor, objects in real systems are required to undergo non-trivial procedures – the most obvious of which being method calls with the object under initialization as the receiver (or, equivalently, function calls with the object as an argument). We will refer to this as an object "escaping" its constructor.

So we require that we are able to use objects under initialization in a context that is not limited to the constructor – that is, we must be able to safely handle objects under initialization in other parts of the code, whether it be the code that creates the objects, or the code that is called from within their constructors.

```
class Person {
    ...
}

class Owner extends Person {
    // An owner is a special type of person,
    // with a pet.
    Pet pet;
}

class Pet {
    // All pets must have owners!
    Owner owner;
}

class Program {
    public static void main(String[] args) {
        // Create the objects -- they are not
        // yet ready to use, as they have not
        // been initialized
        Owner owner = new Owner();
        Pet pet = new Pet;

        // Finish initializing. One of these
        // (and only one) could have been
        // performed within a constructor.
        owner.pet = pet;
        pet.owner = owner;
        // We could comment out these field
        // assignments and the Java type
        // checker would not complain - the
        // fields would simply be left
        // uninitialized (to the peril of
        // anybody who uses the objects later
        // on in the program)
    }
}
```

Figure 18: Java code showing a common pattern for initializing objects that cannot be properly initialized within their own constructors because of mutual dependencies.

## 4.3 Approaches to initialization

**Freedom Before Commitment** (FBC, [18]) employs the simplest approach to initialization of the work we have mentioned. Object initialization finishes with the constructor – an object is said to be `[Free]` throughout its constructor, and while any of the arguments to its constructor are still `[Free]`. Otherwise it is said to be "`[Committed]`." Both `[Free]` and `[Committed]` objects subtype `[Unclassified]`, so that where it is not important which state is applicable, we need not restrict ourselves. These commitment annotations become a part of the type, and methods are required to specify commitment level (with defaults provided) along with the rest of their signature. Methods also gain one more requirement; they must specify the commitment level of their receivers. See Figure 19 for an example of how we might implement our pets/owners example from Figure 18.

Now that we can specify the level of initialization of our objects in FBC, we should discuss what it *means* for an object to be (un-)initialized in a type system of nullity. Whilst ostensibly simple (we have discussed already that the object may not yet satisfy its invariants), it places serious restrictions on what we can do with the object. Within FBC:

- We must treat all non-`null` fields of a `[Free]` object as possibly-`null` (but not `null`-assignable). This is to be expected; if an object has not finished initialization, then we might expect its not-`null` fields to be possibly `null`.
- We cannot store `[Free]` objects in the fields of `[Committed]` objects. If we were to store uninitialized objects in the fields of initialized ones, we would defeat the soundness of the type system; when we access the `[Committed]` object from elsewhere in the code, we would then be able to access the uninitialized fields of the stored `[Free]` object, through an alias that suggested they were initialized.
- In a `[Free]` object, we do not know the initialization state of the fields; even if they prove to be non-`null`, they may in turn also be `[Free]` – consider Figure 19, if we were to reference the `Pet`'s `owner` subsequently in the `Pet`'s constructor. They *may* of course be committed; we must say they are `[Unclassified]` and treat them with the restrictions of both.

**In summary,** FBC requires constructors to assign non-`null` values to all non-`null` fields. Objects are allowed to escape to methods whist `[Free]`, but are restricted to methods expecting possibly-`[Free]` arguments (or receivers, as appropriate). `[Free]` objects may only be assigned to the fields of other `[Free]` objects, and we must assume all their fields to be possibly-`null` and `[Unclassified]`. The challenge of initialization in FBC can be summarised into three core issues: ensuring all fields are properly initialized; allowing *some* objects to hold references to other uninitialized objects; and allowing the object to escape its constructor without compromising the type safety of the system as a whole.

```
class Person {
    ...
}

class Owner extends Person {
    Pet! pet;

    Owner() {
        // this escapes from the constructor
        // when it is passed as an argument to
        // a function (not yet initialized!)
        // which expects a [Free] owner.
        this.pet = new Pet(this);

        // this.pet must still be treated as
        // uncommitted, because it took a reference
        // to this in its constructor

        // Compile error! lick() requires a
        // [Committed] owner, but this is still
        // under construction -- it is still
        // [Free]
        this.pet.lick(this);
    }

    [Committed] void play() {
        // This is allowed, since both the owner
        // and the pet are committed.
        this.pet.lick(this);
    }
}

class Pet {
    Owner! owner;

    // Pet's constructor expects a [Free]
    // (uninitialized) owner to be passed in.
    Pet([Free] Owner owner) {
        this.owner = owner;

        // It was safe to accept an uninitialized
        // Owner, since we did not use the
        // non-nullness of any of its fields.
    }

    [Committed] void lick([Committed] Person person) {
        // A [Committed] Pet may lick a [Committed]
        // Person; the method expects all fields of
        // both this and person to be initialized.
        ...
    }
}
```

Figure 19: Code showing the specification of commitment levels in FBC code. `!` denotes a not-`null` field. Note that `Pet`'s constructor will not accept an initialized `Owner`; we could be more permissive here and use `[Unclassified]`.

**Ownership Immutability Generic Java** (OIGJ, [21]), as we have mentioned previously, uses ownership to help with initialization. The problem of initialization in OIGJ is harder than in FBC, because immutability places more powerful constraints on initialized objects than non-`null`ity does. In non-`null`ity, the only restriction on an initialized object is that we cannot set its fields to uninitialized values (we can freely manipulate them in any other way). With mutability constraints, we cannot change them at all. In essence: while FBC allows us to complete the initialization of the fields it constrains after the object is "cooked" (i.e. we can initialize them to *anything at all so long as it is not `null`*, and complete proper initialization outside of the constructor), a system for mutability constraints can offer no such freedom (by definition!). This is where ownership steps in; while FBC limits initialization to the constructor, OIGJ extends it; an object is considered to be under initialization until its *owner*'s constructor finishes. This allows us the flexibility to properly initialize objects even when we cannot do so in the constructor.

At first glance, OIGJ's seems the better system; but consider that, in terms of the guarantees it makes on the object after initialization finishes, it actually gives us very little. OIGJ does not mandate that *any* proper initialization occurs; which is entirely natural, since the system speaks to whether an object *can change*, rather than the proper initialization of the fields (or lack thereof).

The restrictions that OIGJ places on objects under initialization are in a similar vein to those that FBC uses. OIGJ does not place explicit emphasis on initialization state; it prefers to reason in terms of the properties that it describes (mutability). Mutability constraints come in the following forms: `Immut` (immutable), `Raw` (objects under construction; they are treated as mutable), `Mutable` (subtypes `Raw`: mutable objects that are not under construction). At the top of the type hierarchy is `ReadOnly`, used when we do not know the mutability of an object (and so must treat it with the restrictions of both, like `[Unclassified]`). Whilst both `Raw` and `[Free]` reflect that an object is under initialization, `[Free]` is *purely* a reflection on initialization state, where as `Raw` is a type parameter, and fits into the mutability type hierarchy (it is legitimate to expect fully initialized arguments which subtype `Raw`).

`Raw` objects cannot be stored in fields, like `[Free]` ones. An object with mutability constraint extending `Raw` (so, one that is either under initialization, or one that is `Mutable`) can be freely mutated. Once objects become cooked, their `Immutable` fields cease to be treated as `Raw` and become mutable.

For a comparison of `Raw` and `[Free]`, see Figures 20 and 21.

If we are to produce a system that unifies the specification of types with regard to `null`ity and mutability, we would like them to share a model for initialization. The problem is that `Raw` and `[Free]` make quite different guarantees to (and require quite different promises from) code that refers to objects with each type. The advantages of limiting initialization to within an object's own constructor are obvious; the system is simpler, and we can check in a modular fashion (i.e. without inspecting calling code) that we fulfil our promises (all non-`null` fields get properly initialized during the constructor). The advantage of allowing `Raw` objects to persist as long as their owners' constructors is

| Feature | `Raw` | `[Free]` |
|---|---|---|
| Can be assigned to fields of uninitialized objects | × | ✓ |
| Can be assigned to fields of initialized objects | × | × |
| Object is still in its constructor | ? | ? |
| Must establish invariants before cooking | × | ✓ |
| Initialization state of fields is known | ✓ | × |

Figure 20: Table comparing the guarantees made by and restrictions on `Raw` and `[Free]` types.

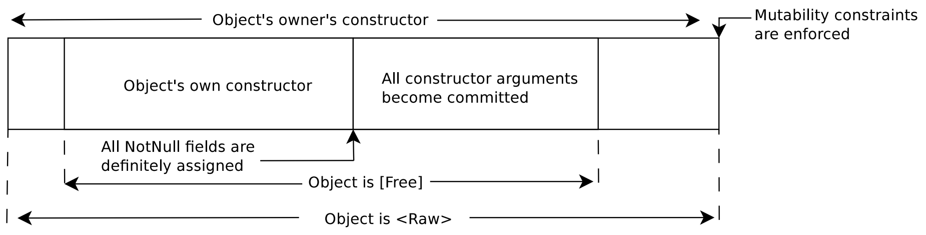
Object's owner's constructor
Mutability constraints are enforced
Object's own constructor
All constructor arguments become committed
All NotNull fields are definitely assigned
Object is [Free]
Object is <Raw>

Figure 21: Diagram showing the object's lifecycle in terms of: Owner's constructor, Object's constructor, `[Free]`, and `Raw` Initialization of `NotNull` fields in [18] must be completed by the end of the object's constructor, but it remains `[Free]` until all the constructor's arguments are committed. Mutability constraints in [21] are enforced from the end of the object's owner's constructor.

obvious: we can perform more complex initialization tasks that could not be performed within the constructor. Each system works well for its own constraints, but what happens if we extend non-`null` initialization to the same commitment point as `Raw`, or limit `Raw` to the object's constructor? In the former case, we lose the simplicity of finishing initialization before the object's constructor finishes (which seems a shame, but it not entirely prohibitive), but more importantly we lose the modularity. Unlike with mutability constraints (where the type system does not become somehow unsound if we fail to properly initialize a field), in the case of `null`ity constraints, *all fields must be initialized* – so if we extend the initialization period to a point outside of the constructor, we must be able to check that the combination of the calling code *and* the constructor, between them, properly initialize fields. The alternative, of limiting `Raw` to the end of the constructor, would prohibit us from many initialization patterns on immutable data structures; we would have to make an object with suitably complex initialization requirements mutable, and so give up many of the advantages of the type system.

## 4.4 Delay Types and initialization using stack-local regions

Both the systems we have just discussed involve keeping uninitialized objects inside protected regions: in the case of [18], this means that objects are initialized before they leave their constructors, while in the case of [21], this means they are initialized before they leave their owners' constructors. [6] and [4] generalise on this idea by talking explicitly about these protected uninitialized areas. [4] uses the notion of "*delay time*," confining uninitialized objects within a given lexical scope, and [6] names its protected regions "*stack local*." These systems differ from the previous ones by not binding these protected regions to e.g. a constructor, or an owner. In doing so, they create a more flexible, if more complicated system.

In Section 2.2.3 we talked about [6]'s stack-local regions. [4] also makes the presence (or otherwise) of an object in one of these protected regions a part of its type, but also allows other methods (to which the object may be escaped during its initialization) to take this part of the type as a parameter. This has the added advantage that, in [4]'s system, we can re-use the same protected region when we create more objects. Such a capability is not required in a system of mutability alone, since we are not required to initialized specific fields.

### 4.4.1 Replacing `[Free]` pets with delayed ones

It is informative to re-visit the example in Figure 19 from the perspective of [4]'s delayed types. This helps to motivate our decision in the next section to favour a model of initialization based on [4] and [6], rather than a constructor- or ownership-based model, and also illustrates explicit initialization regions in action. We have adapted the syntax presented originally in [4] for the sake of a comparison with other systems, to produce the code in Figure 22. We have omitted the declarations for methods on objects which are committed (which are not significantly different).

```
/* Declare the NonNull fields of the object */
Owner.pet : Pet!
Pet.owner : Owner!

/* Owner constructor expects an uninitialized Owner and Pet */
Owner.ctor[t, t > Now] (this : Owner!^t, pet : Pet!^t)
    /* Tell the type system which fields we initialize */
    { this.pet }
{
    this.pet = pet;

    /* pet.lick(this) would still be unavailable in this system, as
         pet is known to be uninitialized */
}

Pet.ctor[t, t > Now] (this : Pet!^t, owner : Owner!^t)
    { this.owner }
{
    this.owner = owner;
}
...
let owner : Owner!^Now =
    delay t
    in
        let o : Owner!^t = alloc Owner[t] // Allocate
        in
            let p : Pet!^t = alloc Pet [t] // Allocate
            in
                o.ctor(pet);    // Initialize Owner
                p.ctor(owner);  // Initialize Pet
                /* o.play() and p.lick() are still forbidden until
                    the end of the delay scope */

                o               // Result of the delay
in
    owner.play() // Owner and pet are committed and NotNull
```

Figure 22: We present code which illustrates that the example in Figure 19 can be better expressed in Delayed Types (notice that we can move the construction of the `Pet` to *outside* the constructor of `Owner`, without risking that either is left uninitialized by the programmer). Under the Delayed Types system, constructors are not implicitly called when object are allocated.

# 5 System Design

Having broadly covered some of the existing systems which tackle `null`ity and mutability, we are in a position to describe the properties of our system. When we must chose between expressivity and simplicity, we chose expressivity. This section introduces the parts of the system which are likely to be unfamiliar with most readers: we will discuss our models for initialization and genericity. Aside from these aspects, the concepts involved are largely familiar. We will go into more detail about the structure of types, the specifics of our initialization regions, and the type hierarchies for mutability and `null`ity parameters in the next section.

## 5.1 Initialization Regions

We have just seen in Figure 22 an example of using initialization regions to ensure the safe initialization of our objects. In reality, *all* the systems we have seen so far have the notion of an initialization region, but some are more explicit than others. The three models we have seen are:

**Constructor-based** The most restrictive model, which limits initialization to within an object's constructor. We have discussed already that this model is too limited for our purposes. [18] uses to constructor-based model, though in order to allow more complex initialization, it concedes that the guarantees (specifically, deep initialization) may not hold as soon as the constructor finishes; rather they will finish some time later, when its least committed argument becomes committed.

**Ownership-based** This model introduces more flexibility. Whilst in the case of [21] it was applied to the question of when we begin to enforce constraints, we have discussed the idea of adapting it to the question of by which point we require the programmer to satisfy guarantees.

**Explicit** This is the most flexible model, binding initialization to a region of the code, rather than a part of the object hierarchy. The code allocating the object can perform arbitrary initialization, and is only compelled to be finished by the time the method ends (that is, the allocating code decides the length of the initialization period).

It is useful to think of the explicit model in terms of an ownership model, but instead of assigning ownership of each object to another object, we assign ownership of an object to a region of the code. When we leave that region of the code, the initialization of all the objects it owns must be complete (and the restrictions are enforced).

**Our system** We use explicit initialization regions, like those found in [4] and [6]. There is a requirement on the code which allocates an object to ensure that the object becomes initialized before the end of the initialization region. Since the duty of object initialization is moved to the point at which the object is allocated, we follow the lead of [4] and do not use implicit constructor call. The commitment point of the object becomes a part of its type for the duration of the initialization region.

**Helper methods** It is desirable to be able to use auxiliary methods to aid in object initialization. To this end, a newly allocated (but as yet not-fully-initialized) object can be the target of method call. Methods are decorated with an effects list, reflecting which fields of its arguments the caller can expect to be initialized (this is just the system in [4]).

**Initialization regions for `nullity` *and* immutability** Recall Figure 21, which shows the gap between the end of initialization under [18] and the beginning of immutability under [21]. In a system with explicit commitment points not coupled to the allocation of a specific object (or ownership hierarchy), the allocator of the object may place that commitment point wherever is convenient; initialization may extend as long as desired. It is not clear that we would gain anything by allowing mutability constraints to be relaxed for any duration after the initialization of `NotNull` fields is complete, so we unify the commitment of objects with respect to both the end of their `NotNull` initialization, and their mutability constraints.

## 5.2 Generic `null`ity and Mutability

We have discussed already that we bring genericity in terms of mutability *and* `null`ity to our types. Generics are a familiar construct, and we see no reason to depart from convention in this regard: our parametric mutability closely resembles that of [20] and [21], and parametric `null`ity will be modelled on this system. In order to allow methods whose arguments are covariant in their generic parameters, we us a system like that of JAVA wild-cards to provide upper bounds for both mutability and `null`ity parameters of a type. We will give more detail in the next section, but it will be possible, for example, to write methods which are completely generic in terms of the mutability and `null`ity parameters of their arguments, for example in Figure 23. The mutability parameters themselves are almost exactly those from [21]: `Immutable` objects do not change, `Mutable` objects can, and we promise not to write to `ReadOnly` objects, but accept that somebody else might (that is, `ReadOnly` signifies that we can accept an object with any level of mutability). `null`ity constraints are, again, of a familiar form: a reference is either `Nullable` or `NotNull`, with a `NotNull` reference able to take the place of a `Nullable` one (subject to the restrictions that we will discuss in the next section).

The figure also reveals some other details of the system:

- The type of a method's receiver is a part of its signature. The implication of this decision is that we can limit the availability of certain functions by the parametrization of the receiver's type: for example, it is possible to

```
class C[i]<,n1> {
    f : C[i, n1]<,n1>

    /* The type of the first argument is given directly by the
        mutability and nullity constraints of the class */
    C[i, n1]<,n1> getF(C[i, NotNull]<,n1> this) {
        this.f // Just field lookup
    }

    /* Accept an instance of C which may or may not be Mutable, and
        whose f field may or may not be Nullable */
    void m (C[<? extends ReadOnly>, NotNull]<, <? extends Nullable>
        > this) {
        ...
        /* x would have type C[<? extends ReadOnly>, <? extends
            Nullable>]<, <? extends Nullable> > */
        x = this.getF()
        ...
    }
}
```

Figure 23: Figure demonstrating the constructs that allow generic covariance. We have omitted effects and initialization-state constraints from method signature, and assumed the presence of a type `void`.

specify that certain methods can only be called upon `Mutable` instances of the class (this is akin to the conditional method guards found in [8]).

- We separate the first mutability and `null`ity parameters from the others. This is because the first parameters represent the object which is pointed to by a particular references that holds this type: is the thing at the end of this reference mutable? It it guaranteed to be not-`null`? The other parameters are used within the body of the class to specify the types of its fields and signatures of the methods.
- Class definitions are not parametric in terms of their own `null`ity. We will cover the reasons for this in more detail in Section 7.8.4.

**Class genericity** We only chose to tackle genericity in the context of mutability and `null`ity parameters. We make no attempt to provide types which are generic in one or more class parameters. Such systems have been studied in detail, and good implementations exist (which we covered in some detail in Section 2).

## 5.3 Runtime model

We have mentioned already that the runtime model is simple, requiring no auxiliary constructs to specify our properties. We use a simple heap-based model, with objects, field-assignment, and runtime (class-based) method resolution. To that end, objects on the heap are just a class identifier, and a list of fields. Since method resolution is performed by looking up the runtime class of an object, and from there determining the appropriate method definition, an object is a unit of polymorphism (as well as statefulness): this is everything we need to encode booleans, integers and conditional branching flow constructs (the if-statement) – demonstrated by early systems for object calculus like [1].

## 5.4 The language

The source language itself is illustrated through extensive examples in section 9. Most of the examples there assume standard constructs like `if-then-else`, integers and booleans, but the language actually contains none of these; the full range of available expressions is shown in Figure 30. For the sake of illustration, we present an example code listing which assumes no additional constructs (except the definition of a class `Callable`, with a single method `call`) in Figure 24, and encodes booleans. The listing demonstrates class definition and the form of method signatures.

We are now ready to discuss the type system in more detail.

```
class Bool[i]<,> extends Object {
    {} // Do not promise to initialize any fields
    {} // Do not expect any parameters to be uninitialized
    /* Return an immutable Bool which may be null */
    Now -> Bool[Immutable, Nullable]<,>
        /* Method name */
        then(
            /* Expect an instance of Bool, which must be committed,
                 with the mutability given by the type of the
                receiver (NotNull) */
            Now -> Bool[i, NotNull]<,> this,
            /* Expect an instance of Callable, which must be
                committed, with any mutability, which is not Null
                */
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_true,
            /* As before */
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_false) {

            /* Just return null */
            null
    }
}

/* Extend Bool - this is a promise to implement its interface in
    full */
class True[i]<,> extends Bool {
    {} {} Now -> Bool[Immutable, Nullable]<,>
        then(
            Now -> Bool[i, NotNull]<,> this,
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_true,
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_false) {

            /* Do the action */
            callable_true.call();
            /* Return null */
            null
    }
}

class False[i]<,> extends Bool {
    {} {} Now -> Bool[Immutable, Nullable]<,>
        then(
            Now -> Bool[i, NotNull]<,> this,
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_true,
            Now -> Callable[<? extends ReadOnly>, NotNull]<,>
                callable_false) {

            callable_false.call();
            null
    }
}
```

Figure 24: Encoding of booleans. Assumes the presence of a class `Callable` which has the method `call`.

# 6 The Type System: Informally

This section presents an informal description of our type system. In Section 7, we will lay out the formal model. The system borrows heavily from the systems we have already mentioned

We use classes, objects, a call stack, a heap, field assignment, and methods. For simplicity, we will assume single inheritance and only virtual methods (we will see that method call is resolved by the runtime class of an object).

We will define initialization regions, based on delay types from [4]. We introduce the form of an expression's type, which will contain several pieces of information:

- A *Class Name*. This is familiar to all programmers of statically typed languages.
- An initialization region parameter, enabling us to:
  - Decide whether or not the object is under initialization – do its invariants necessarily hold?
  - Refer to the initialization region of the object. This means that we can create new objects (and assign them to fields) that are within the same initialization region.
- A primary mutability parameter, describing the mutability of the object with this type.
- A primary nullity parameter, describing the possibility that the reference is `null`.
- Zero or more further mutability and `null`ity parameters, which can be used in the body of the class, parametrizing the types of its fields and method arguments.
- A set of zero or more as-yet possibly-uninitialized fields

Whilst the `null`ity and mutability parameters serve an obvious purpose, we would like to give an intuition into our model for initialization regions. Readers familiar with [4] or [6] will find them nearly identical (up to a restriction on references between initialization regions), but they will be unfamiliar to most other readers.

## 6.1 Initialization Regions

The idea of an initialization region is to be able to assign new objects to the heap, and properly type them, without requiring that they yet be initialized. We say that an object has a commitment point given by the end of its initialization region in the code. As in [4], we will refer to this commitment point as a time, $t$. Objects whose commitment points have passed are guaranteed to be initialized, and we say they have commitment point `Now`. We further borrow from this system, and say that commitment points which have not yet passed (i.e. are in the future) are after `Now`: $t >$ `Now`. We should note that [4] models the ordering of time, while we (as in [6]) do not.

When we want to create a new initialization region $t$, we will write:

$$\texttt{delay } t\{e\}$$

with $e$ an expression that may refer to $t$. At the end of $e$, we say that objects allocated with the commitment point $t$ become committed; we substitute `Now` in place of their commitment point in their type. Within $e$, we say $t > \mathit{Now}$. We may create nested initialization regions within $t$, but since we do not place any ordering on time,

$$\texttt{delay } t\{\ \texttt{delay } t'\{e\}\}$$

is equivalent to

$$\texttt{delay } t'\{\ \texttt{delay } t\{e\}\}$$

We will see in Section 7.8.2 that we would not gain anything by placing an ordering on time.

Initialization regions cannot extend beyond the end of the method that creates them, and a reference to an object $o$ cannot be stored in the field of an object with commitment different to that of $o$. We enforce the following:

- Objects with commitment point $t =$ `Now` cannot reference objects still under initialization (those with commitment point $t' >$ `Now`).
- Objects with commitment point $t$ cannot reference objects with commitment point different to their own.
- If an initialization region is created within a method, then that method is at the top of the call stack at the end of that initialization region.

These restrictions on which objects can hold references to others dictate the nature of references between objects in the heap, which Figure 25 shows. If an initialization region $t_n$ is created at stack frame $\phi_n$, then it can only be referenced by stack frames higher up the call stack ($\phi_{>n}$).

[6] gives a similar discussion for the possibilities for references between object in various stack frames under their tokens-based system. That system does not allow the specification of the commitment points to be passed up the stack; the approach is simpler, but makes it impossible to complete the initialization of an object's fields via helper methods; there is no way for a method to tell whether an object is mutable, or simply uninitialized, so a conservative approach is taken, and methods cannot assign objects under initialization to fields, unless the initialization region for those objects is declared within the same method.

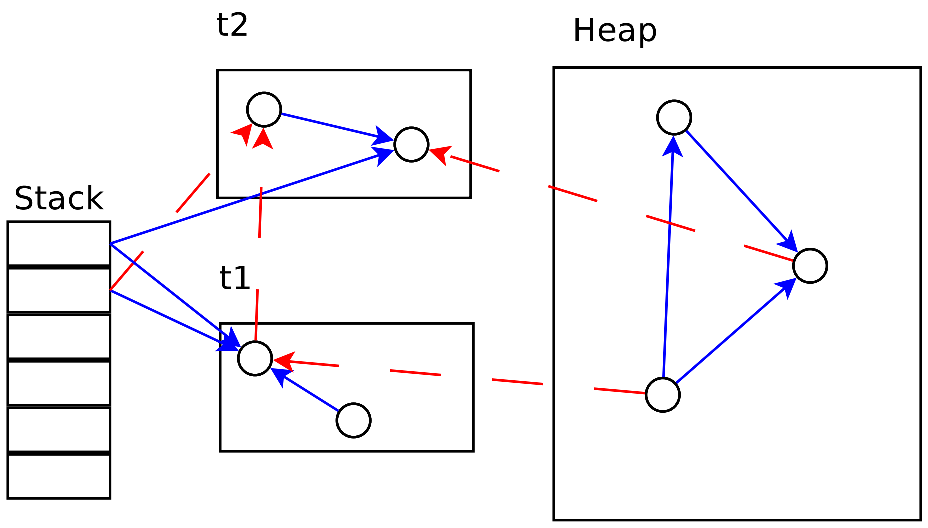

Figure 25: Figure showing the possibility of references to objects based on their commitment points times, and a method's place on the stack. Red (dashed) lines represent references which are not allowed and blue (solid) lines represent references which are permissible. $t1$ and $t2$ are protected initialization regions.

## 6.2 Types for immutability

We construct a type hierarchy for immutability. We need to be able to specify that an object is `Mutable`, `Immutable`, or under initialization. When we want to express that we do not care for the mutability of an object, we will say that it is $<?$ `extends ReadOnly` $>$. $<?$ `extends ReadOnly` $>$ will be an upper bound on `Mutable` and `Immutable` types, and will neither allow field-writes, nor imply immutability. `Mutable`will also subtype another mutability construct: $<?$ `extends Writable` $>$. This is the upper bound of `Mutable` and the mutability type of objects under initialization (which may be `Immutable` when they are committed). This type hierarchy is shown in Figure 26.

Both `Immutable` and `Mutable` references will correctly type when a `ReadOnly` reference is required, and `Writable` can be used to require either `Mutable` or objects under initialization. `Writable` and `ReadOnly`, however, will not be available as the concrete type of a field; both `Writable` and `ReadOnly` will serve *only* as bounds, as in [6] – we will write them only as $<?$ `extends ReadOnly` $>$ and $<?$ `extends Writable` $>$.

## 6.3 Types for `nullity`

Similarly to types for immutability, we require an upper bound on `nullity` for the case when we do not care for whether a reference is `Nullable` or not. In general, when constructing systems of `nullity`, authors simply allow `NotNull` reference

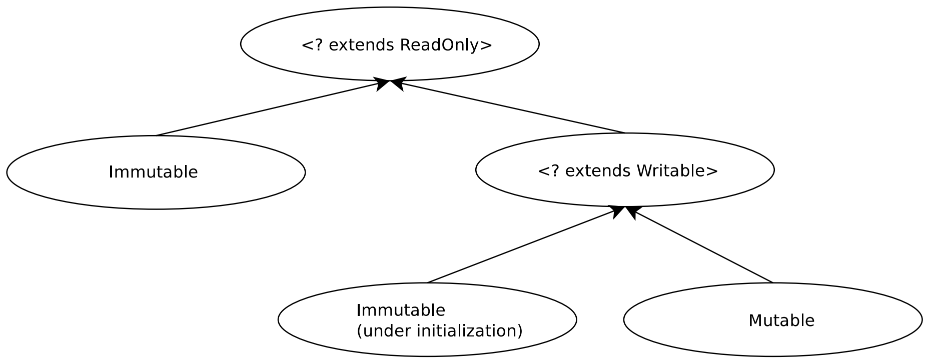

Figure 26: Figure showing the type hierarchy of immutability types.

to subtype `Nullable` ones. Whilst we also wish to have such a situation, we need to take more care in the case of generic parameters: we write a least upper bound for `Nullable` and `NotNull`:

$$\texttt{Nullable} \leq <?\ \texttt{extends Nullable}>$$

$$\texttt{NotNull} \leq <?\ \texttt{extends Nullable}>$$

We present a type hierarchy in Figure 27. Whilst we want to allow the user to use a `NotNull` reference in place of a `Nullable` one, we want to avoid the situation where a type with `NotNull` generic parameter subtypes a type with `Nullable` generic parameter: the generic parameter dictates the `null`ity of the object's fields, and we do not want to treat an object with `NotNull` fields in the same way as one with `Nullable` fields. The subtyping relationships are laid out in full by Figure 37 in Section 7.5.2.

One may not assign `null` to the `NotNull` fields of an object under initialization, but one also cannot guarantee them to be `NotNull`. The former is to allow us to check that all the `NotNull` fields of an object become initialized; we don't need to check that a helper method is not reseting all our `NotNull` fields to `null` when we pass in an object under initialization. This restriction is also found in other systems for `null`ity.

## 6.4 Field initialization as part of the type information

Readers familiar with type systems for `null`ity will have come across the notion of effects in the type system. The idea is that we must keep track of which fields of an object are (or conversely are not yet) initialized. In doing so, we can check that during the course of an object's initialization, all of its `NotNull` fields are assigned appropriate values. Some systems chose to add this construct to the typing environment, but we have added the possibly-uninitialized fields to the type itself. Types are written in the form

$$(\ldots)^{\{\overline{f}\}}$$

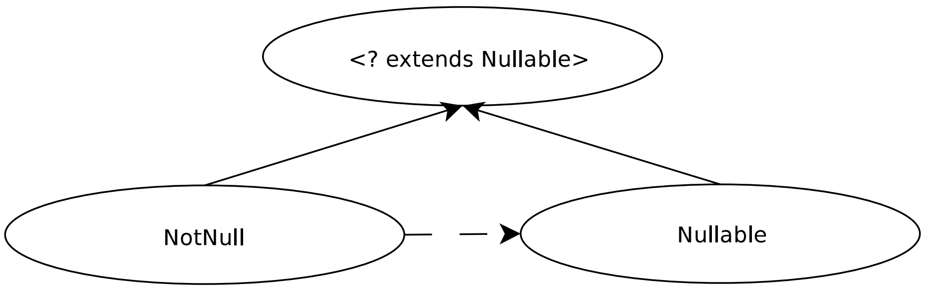

Figure 27: Figure showing the type hierarchy of nullity types. The dashed line reflects that whilst in general a type parametrized with `NotNull` in place of `Nullable` is not an acceptable substitute, when the `NotNull` reflects the `null`ity of the reference itself we allow the subtyping relationship.

to mean that the fields $\overline{f}$ are not yet guaranteed to be initialized. The outcome is that it is possible to write method signatures which expect objects to be in certain stages of initialization, and the programmer is allowed to immediately use the non-`null`ness of the fields to which they have just assigned (we can see an example in Figure 28, which omits mutability constraints and optional type parameters). For an example of how we can use the additional information in method signatures, see Figure 54 (which does include all the type annotations in the full system). By the time an object becomes committed, the programmer must ensure that its set of possibly-uninitialized-fields is just the empty set.

## 6.5 The type rules

The type rules presented in Figures 39 and 40 are mostly straightforward. The rules for variable lookup, object allocation, and possibly-null-dereference are of the familiar form. We include a judgement for type subsumption, and a less familiar rule for environment subsumption, which simplifies branching constructs: one environment is a sub-environment of another if it types all entries in the latter with a type that is more specific. Clearly all environments are sub-environments of the empty one.

Typing of method call is more complex than the others, and the rule is described in more detail in Section 7.6.1, but field assignment and lookup are also more complicated than in other systems.

**Field Lookup** Field lookup can be typed by either one of two separate judgements: (T-FieldLookupUninit) and (T-FieldLookup). There are three cases for field lookup:

**When an object is committed** This is the simplest case; we just check the class of the receiver, and find the class of the type directly from the class definition (modulo substitutions of the mutability and `null`ity parameters).

```
class C {
    f : C[NotNull]
    g : C[NotNull]
}
...
delay t {
    /* Allocate an object with class C and commitment point t to
        the variable z1, which holds a NotNull reference */
    alloc t -> C[NotNull] as z1;
    /* z1 : (t -> C[NotNull])^{f, g} */
    alloc t -> C[NotNull] as z2;
    /* z2 : (t -> C[NotNull])^{f, g} */

    /* z1 and z2 are now objects under initialization with
        committment point t */

    /* Type error: f is not initialized, so must be treated as
        possibly-null*/
    z1.f.g

    /* Type error: g is not initialized, so must be treated as
        possibly-null */
    z1.g.g

    z1.f = z2;
    /* z1 : (t -> C[NotNull])^{g} */

    /* Now legal: f is initialized, so we know it is NotNull*/
    z1.f.g

    /* Still a type error: g is still uninitialized */
    z1.g.g

    z1.g = z2;
    /* z1 : (t -> C[NotNull])^{} - we have finished initializing z1
        */

    /* Now legal: */
    z1.g.g

    /* Must also initialize z2, otherwise we have a type error */
    z2.f = z2;
    z2.g = z2;
    /* z2 : (t -> C[NotNull])^{} */
}
```

Figure 28: The programmer may use the knowledge of the already-initialized fields further down in the same method – omitting mutability and optional type parameters.

**When an object is not committed** There are two cases:

**The field is not listed as possibly-uninitialized** In this case, we type the field lookup exactly as in the previous case, with the exception that, since the object owning the field is uninitialized, we must assume that the referenced object is also uninitialized: in particular we assume the list of uninitialized fields for the object that is the result of field lookup to be all of its `NotNull` fields.

**The field is listed as possibly-uninitialized** In this case, we do exactly as in the previous case, but we also change the type of the field-lookup to be possibly-null. In this case we must assume that: the result of a `NotNull` field lookup might in-fact be `null`, and the result of any subsequent lookups on the result might also be `null`, even when they are declared as `NotNull`.

**Field Assignment** Field assignment is nearly exactly how we would expect, but warrants a paragraph of explanation. We check that the receiver is not `null`, that it is `Writable` (i.e. the type of the receiver is either `Mutable` or `Immutable` but under initialization), then that the value being assigned is a subtype of the expected type for the field. The part of the rule that is unusual is that we check that the expected field type is "*Grounded*." This entails checking that is not one of <? `extends ReadOnly` >, <? `extends Writable` >, or < ? `extends Nullable` >. This is exactly the check we have to make in any generic type system that supports covariant type parameters with wild cards: the idea is that we cannot assign to a field about which all we know is that it is e.g. `ReadOnly`; there might be other aliases to the object that type the field as `Immutable` or `Mutable`. We go into more detail about this in Section 7.8.3. In the case when we are assigning to a field of a variable ($y.f = \ldots$), then we also update the type information for that variable to reflect that the field is now initialized.

# 7 The Type System: Formal Description

In order to reason about the type system we create, we will require the language to do so. In this section we will formalize:

- The programming language itself,
- Typing statements,
- Runtime semantics,

## 7.1 Type statements

As we discussed in Section 6, a fully specified type must contain all the following information:

- Class,
- A commitment point, $t$,
- A primary mutability constraint,
- A primary nullity constraint,
- Zero or more further mutability and nullity constraints, which parametrize the definition of the underlying class,
- Zero or more possibly-uninitialized fields.

We will write types using the vocabulary in Figure 29. We write a fully-qualified type $\sigma$, contains information about the initialization state of an object. Informally, it should be read: "At time $\Theta$, an object with this type will have mutability $I_0$ and `nullity` $N_0$. The definition of the class is parametrized by $\mathbb{I}_0$, $\overline{\mathbb{I}}$, and $\overline{N}$, and the fields $\epsilon$ may be uninitialized."

In Section 7.3, we specify program definitions. A class definition specifies the fields and methods of a type, and is parametrized by the $\mathbb{I}$s and $\mathbb{N}$s in that type. The definition of fields here is straightforward (note that fields are also defined in terms of unqualified types; the fields of an object will have fully-qualified types dependent upon the fully-qualified type of that object – e.g. they will be assumed by the type system to have the same commitment point as the object itself, so commitment information isn't required here), but the definition of methods is a little more complex. Method definitions are in terms of fully qualified return and argument types (which allows us to specify that an object must be, e.g. immutable), but also a set of future times, $\Sigma$, which allows us to specify that some of the arguments to a function must by uncommitted, and an effects function, $\Psi$, which promises to the calling code which fields we will initialize.

As we will see when we specify the type system itself, these constraints govern both the arguments with which a method can be called, and also the times at which an object may be a receiver of a method (e.g. only instances of a list that is mutable will be able to have objects added to them after their commitment point).

## 7.2 Notation

Before we continue with the form of programs, and assorted definitions, we set some notation.

**Ranges** We write:

$$\overline{x}\big|_{j\ldots k} \equiv \{x_i\}_{i=j\ldots k}$$

and we will omit the indexes $i$ and $j$ in most cases, to get

$$\overline{x} \equiv \overline{x}\big|_{1\ldots n}$$

**Functions** We write:

$$\{x \mapsto a, y \mapsto b, z \mapsto c, \ldots\}$$

to mean a function which maps $x$ to $a$, $y$ to $b$, and so on.

We mix this notation with that for ranges, to get:

$$\{\overline{x \mapsto a}\}$$

which is equivalent to a function $f$ such that:

$$\forall i \in [1 \ldots n] : f(x_i) = a_i$$

**Substitutions** We will use the normal notation for substitutions:

$$C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > [\texttt{Immutable}/\mathbb{I}] = C[\texttt{Immutable}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >$$

And we freely combine it with the above notation to perform a set of substitutions:

$$C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > [\texttt{Mutable}/\overline{\mathbb{I}}] = C[\mathbb{I}, \mathbb{N}] < \overline{\texttt{Mutable}}, \overline{\mathbb{N}} >$$

$$C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > [\overline{\mathbb{I}/\texttt{Mutable}}\big|_{0\ldots 1}] = C[\texttt{Mutable}, \mathbb{N}] < \texttt{Mutable}, \overline{\mathbb{I}}\big|_{2\ldots n}, \overline{\mathbb{N}} >$$

We will also use this substitution notation with functions, to relabel elements in their domain:

$$\{\overline{x \mapsto z}\}[\overline{y/x}] = \{\overline{y \mapsto z}\}$$

| | |
|---|---|
| C, T $\in$ *ClassId* | Class Identifiers |
| $f, g, \in$ *Field Id* | Field Identifiers |
| $m \in$ Method Identifiers | |
| $\epsilon ::= \{\overline{f}\}$ | Uninitialized fields |
| $\texttt{this}, x \in$ Argument Identifiers | |
| $z \in$ Local Identifiers | |
| $y ::= x \mid z$ | Variables |
| $t \in$ *Time Variables* | Initialization Region Identifiers |
| $\chi ::= \texttt{Now}\mid \texttt{t}$ | Possible Times |
| $\theta \in$ Formal Time Parameters | |
| $\Theta ::= \chi \mid \theta$ | Time Parameters |
| $I ::= \texttt{Mutable} \mid \texttt{Immutable}$ | Mutability Modifiers |
| $i \in$ Formal Mutability Parameters | |
| $\mathbb{I} ::= I \mid i \mid <?\ \texttt{extends ReadOnly} >$ | |
| $\mid <?\ \texttt{extends Writable} >$ | Mutability Parameters |
| $N ::= \texttt{Nullable} \mid \texttt{NotNull}$ | `nullity` Modifiers |
| $n \in$ Formal Nullity Parameters | |
| $\mathbb{N} ::= N \mid n \mid <?\ \texttt{extends Nullable} >$ | Nullity Parameters |
| $\varsigma ::= C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >$ | Unqualified Types |
| $\sigma ::= (\Theta \rightarrow \varsigma)^{\epsilon}$ | Fully Qualified Types |
| $\Gamma ::= \{\overline{x \mapsto \sigma}\}$ | Typing Environment |
| $\Lambda ::= \{\overline{\Theta}\}$ | Time Environment |
| $\Sigma ::= \{\overline{\theta}\}$ | A set of future commitment points |
| $\Psi$ : Argument Identifiers $\rightarrow$ set of *Field Id*s | Effects list |
| $e ::=$ Expressions | |
| Field Def $::= f : \varsigma$ | Field Definition |
| Meth Def $::= m : \Psi, \Sigma, \sigma, (\sigma, \overline{\sigma})\{e\}$ | Method Definition |
| Qualified Method Signature $::= \Psi, \Sigma, \sigma, \Gamma$ | |
| Class Def $::=$ $\texttt{class}\ C[i] < \overline{i}, \overline{n} >$ | |
| $\texttt{extends}\ C'\{\overline{\text{Field Def}}, \overline{\text{Meth Def}}\}$ | Class Definition |
| Program $:==$ Class Def, Program $\mid e$ | Program definition |

Figure 29: Vocabulary of types.

## 7.3 Program definition

We define a program in Figure 29 as a set of declarations of the following form (followed by an expression):

$$\texttt{class } C[i_0] < \bar{i}, \overline{n} > \texttt{ extends } D\{\overline{f : \varsigma}, \overline{m : \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}}\}$$

This declares a class $(C)$, which has fields $\overline{f}$ of types $\overline{\varsigma}$, and methods $\overline{m}$ with the signatures $\overline{\Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}}$. The $\Sigma$'s will constrain the commitment points of a method's arguments, and the $\Psi$'s will denote the initialization performed by the method on its arguments. Notice that the definitions for field types and method signatures are parametric in terms of $i_0$, the mutability of the receiver, but not in terms of its `null`ity.

To refer to the declarations in the program, we define a function:

$$\Omega : \mathit{ClassId} \rightarrow \text{Class Definition}$$

We then extend $\Omega$ over field identifiers:

$$\Omega : \mathit{ClassId} \times \mathit{Field\ Id} \rightarrow \text{Parametrizable Types}$$

So with the program $\Omega$ with the single class declaration above, we would have mappings:

$$\Omega(C) = C[i_0] < \bar{i}, \overline{n} > \texttt{ extends } D\{\overline{f : \varsigma}, \overline{m : \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}}\}$$

$$\Omega(C, f_i) = \varsigma_i$$

Note that the definition of $\Omega(C)$ is parametrized by $i, n, \bar{i}, \overline{n}$. We extend $\Omega$ to include more specified types:

$$\Omega : \text{Unqualified Types} \times \mathit{Field\ Id} \rightarrow \text{Unqualified Types}$$

$$\Omega(C[\mathbb{I}] < \bar{\mathbb{I}}, \overline{\mathbb{N}} >, f) = \Omega(C, f)[\mathbb{I}/i, \mathbb{N}/n, \overline{\mathbb{I}/i}, \overline{\mathbb{N}/n}]$$

When we look up the type for a field in an unqualified type, we substitute the concrete nullity and mutability parameters of that type into the formal parameters specified by the class.

**Method signatures** We add a further extension to the function $\Omega$ ; We allow $\Omega$ to range over *ClassId*, Method Id pairs:

$$\Omega : \mathit{ClassId} \times \text{Method Id} \rightarrow \text{ Signatures}$$

Given the program $\Omega$ defined above, we define:

$$\Omega(C, m_i) = \Psi_i, \Sigma_i, \sigma_{i,ret}(\sigma_{i,rec}, \overline{\sigma_i})\{e_i\}$$

### 7.3.1 Auxiliary functions

To help us with definitions later, we will define the following auxiliary functions:

***body*** To easily retrieve a method body from a type and method identifier, we define the *body*:

$$\textit{body} : \textit{ClassId} \times \text{Method Id} \rightarrow \text{ Expression}$$

which maps to method bodies: when $\Omega(C, m_i)$ is defined as above, then we write

$$\textit{body}(C, m_i) = e_i$$

***Class*** As shorthand for accessing the class-name from a type we define a function:

$$\textit{Class} : \text{Unqualified types } \rightarrow \textit{ClassId}\text{s}$$

$$\textit{Class}(C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >) = C$$

We extend the function over fully-qualified types in the obvious way.

***Fields*** In order to talk about the set of fields that belong to an object of a given type, $C$, when

$$\Omega(C) = C[i, n] < \overline{i}, \overline{n} > \texttt{ extends } D\{\overline{f : \varsigma}, \overline{m : \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}}\}$$

then we say that

$$\textit{Fields}(C) = \{\overline{f}\}$$

and we make the natural extension to unqualified types, and fully-qualified types:

$$\textit{Fields}(C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >) = \{\overline{f : \varsigma[\mathbb{I}/i, \mathbb{N}/n, \overline{\mathbb{I}/i}, \overline{\mathbb{N}/n}]}\}$$

$$\textit{Fields}(\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >) = \{\overline{f : \varsigma[\mathbb{I}/i, \mathbb{N}/n, \overline{\mathbb{I}/i}, \overline{\mathbb{N}/n}]}\}$$

***notNullFields*** We then add a secondary function for extracting only the non-`null`fields of type:

$$\textit{notNullFields} : \text{Fully-qualified types } \rightarrow \text{set of } (\textit{Field Id}/\text{Fully-qualified type}) \text{ pairs}$$

$$\textit{notNullFields} : \text{Unqualified types } \rightarrow \text{set of } (\textit{Field Id}/\text{Unqualified type}) \text{ pairs}$$

$$\textit{notNullFields}(\sigma) = \{f : \sigma \text{ st. } f : \sigma \in \textit{Fields}(\sigma) \wedge \vdash \sigma \leq \texttt{NotNull}\}$$

$$\textit{notNullFields}(\varsigma) = \{f : \varsigma \text{ st. } f : \varsigma \in \textit{Fields}(\varsigma) \wedge \vdash \varsigma \leq \texttt{NotNull}\}$$

***time*** We define shorthand for taking the initialization point from a type by a function from fully-qualified types to times:

$$\textit{time} : \text{Fully Qualified Types} \rightarrow \text{Time Parameters}$$

$$\textit{time}((\Theta \rightarrow \varsigma)^{\epsilon}) = \Theta$$

***uninit*** For retrieving the possibly uninitialized fields from type information, we define *uninit*:

$$\mathit{uninit} : \text{Fully-qualified types} \rightarrow \text{set of } \mathit{Field\ Id}\text{s}$$

$$\mathit{uninit}((\Theta \rightarrow \varsigma)^{\epsilon}) = \epsilon$$

***possiblyNull*** We define the function *possiblyNull*, which replaces the `NotNull` modifier of uninitialized objects with <? `extends Nullable` > - this won't allow us to assign `null` references to `NotNull` fields, but it forces us to treat uninitialized fields as if they *might* be `null`.

$$\mathit{possiblyNull} : \text{Fully-qualified types} \rightarrow \text{Fully-qualified types}$$

$$\mathit{possiblyNull}((\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}) = (\Theta \rightarrow C[\mathbb{I}, <?\ \texttt{extends Nullable} >] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}$$

## 7.4 The programming language

With class, field, and method declarations for a program Ω defined as in Section 7.3, we can then define expressions in the programming language which evaluate to a result.

Program expressions are shown in Figure 30. We have the normal imperative ingredients: values (the final result of evaluating an expression), field assignment, field lookup, and sequences of expressions. We add an expression `delay` which creates a new initialization region. Method lookups are of a familiar form, with one exception: a method call is parametrized by the fully-qualified type of the receiver.

The de-referencing of possibly-`null` references is of a familiar form for non-`null` type systems; if the reference is not `null`, we evaluate $e_2$ as if y has a non-`null` concrete type. Finally, we allocate a new object to the heap by calling "`alloc , as` " which is equivalent to a parameter-less, body-less `new` call in Java (all fields are zero-initialized – since all types in our language are reference types, this means that they will be initialized to `null`). Any further initialization can be done through method calls, field assignment, and so on. There is then a proof burden on the calling code to ensure all non-`null` fields are initialized before the end of the initialization region *t*.

**Ensuring field initialization with effects** In order to ensure that all non-`null` fields are initialized, we must in some way keep track of which fields of an object under initialization we have assigned to. Since the non-`null` initialization of fields is a monotonic operation (that is, once they have become non-`null`, they will not revert to `null`), this contextual information could be used when later *accessing* those fields (that is, the user already knows the fields are non-`null`, because she made them that way further up the method – regardless of the fact that the object's initialization is not yet complete). With this idea in mind, it makes sense to carry the initialization information around as part of the

| | |
|---|---|
| $\iota \in \psi$ | (Addresses in the heap) |
| $v ::= \iota \mid$ `null` | (A value is an address or `null`) |
| $z \in$ Local variable identifiers | (Assigned by variable allocation) |
| $x \in$ Method Arguments | (Received by methods) |
| $y ::= x \mid z$ | |
| $p ::= y \mid p.f$ | (Paths) |
| $e ::= p \mid p.f := e \mid e; e$ | (Result, field assignment, lookup, concatenation) |
| $\mid$ `delay` $t\{e\}$ | (Initialization region) |
| $\mid < \bar{\mathbb{I}}, \bar{\mathbb{N}} > p.m(\overline{y})$ | (Method call) |
| $\mid$ `ifnull` $y$ `then` $e_1$ `else` $e_2$ | (Possibly-null dereference) |
| $\mid$ `alloc` $\sigma$ `as` $z$ | (Object allocation) |

Figure 30: Expression definitions

type. For this reason, we included the effects list as part of the fully-qualified type information of a variable, instead of as contextual information. We will write a fully-qualified type in the form:

$$(\Theta \to \varsigma)^{\epsilon}$$

with $\epsilon$ denoting the not-definitely-initialized fields of the object, and $\Theta$its initialization point (we will see full syntax for all type information below). Note that $\epsilon$ is just a set field whose initialization state we *do not know*. We will abbreviate

$$(\Theta \to \varsigma)^{\emptyset}$$

to

$$\Theta \to \varsigma$$

## 7.5 Typing Expressions

We will use a context ($\Gamma$) for the types of local variables (those that are not fields of objects, whose type information cannot be resolved through $\Omega$). An environment $\Gamma$ is just a map from variable names to fully-qualified types:

$$\Gamma(y) = \sigma$$

We will use a "time environment" ($\Lambda$), constituted of a set of time variables, which we will use to make judgements about the passing of time. We will discuss the time environment in more detail in Section 7.5.2.

There is a burden of proof on the type system to ensure that all the `NotNull` fields of all objects have been properly initialized, and we use the possibly-uninitialized fields list (which is part of the fully-qualified types given by the environment) to achieve that (contrast to [4], which uses a separate list of initialization effects).

We will write typing statements in the following form:
$\Omega, \Gamma, \Lambda \vdash e : \sigma, \Gamma'$
which can be read as: under environment given by $\Gamma$ and $\Lambda$, with program definitions as given in $\Omega$, the expression $e$ has fully qualified type $\sigma$. $\Gamma$ will be a set of (variable identifier/type) pairs. Typing an expression $e$ will also lead to a modified environment, $(\Gamma')$. The change of environment keeps track of newly declared variables, and will help us determine the how initialized newly assigned objects are. We then define a system of typing judgements in Figures 39 and 40 to type expressions.

Because the guarantees offered by the type system in terms of mutability are dependent upon initialization state, we use judgements about it in the form:
$\Lambda \vdash \sigma \leq \mathbb{I}$
where $\mathbb{I}$ is the declared mutability of a field. The judgement reads: With the commitment points in $\Lambda$ in the future, the type $\sigma$ has mutability subtyping $\mathbb{I}$.

### 7.5.1 Well-formedness of a type

We need to be able to check that the types in our programs make sense. For this, we will need to check that:

- They contain class-names which are within the program, and
- the Mutability and Nullity constraints are permitted

We make the judgement:

$$\frac{\Omega(C) = C[i_0] < \bar{i}\big|_{1\ldots n}, \overline{n}\big|_{1\ldots m} > \ \texttt{extends}\ D\{\ldots\}}{\mathit{TypeWF}_\Omega((\Theta \to C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}\big|_{1\ldots n}, \overline{\mathbb{N}}\big|_{1\ldots m} >)^\epsilon)}$$

We also need to be more specific than this: types allowed in method signatures are more general than types allowed in fields, for example, so we make the judgement:

$$\frac{\begin{array}{c}\sigma = (\Theta \to C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}\big|_{1\ldots n}, \overline{\mathbb{N}}\big|_{1\ldots m} >)^\epsilon \\ \mathit{TypeWF}_\Omega(\sigma) \\ \forall i \in \{0\ldots n\} : \mathit{Grounded}(\mathbb{I}) \\ \forall j \in \{0\ldots m\} : \mathit{Grounded}(\mathbb{N})\end{array}}{\mathit{FieldType}_\Omega(\sigma)}$$

Where *Grounded* is defined by:

$$\frac{}{\mathit{Grounded}(I)}$$

$$\frac{}{\mathit{Grounded}(i)}$$

$$\frac{}{\mathit{Grounded}(N)}$$

$$\frac{\Theta \in \Lambda}{\Lambda \vdash \Theta > \texttt{Now}}$$

$$\frac{}{\Lambda \vdash \texttt{Now} \leq \texttt{Now}}$$

Figure 31: Judgements about time

$$\frac{}{\vdash \mathbb{I} \leq \mathbb{I}} \text{ SM-Reflexive}$$

$$\frac{\vdash \mathbb{I} \leq \mathbb{I}' \qquad \vdash \mathbb{I}' \leq \mathbb{I}''}{\vdash \mathbb{I} \leq \mathbb{I}''} \text{ SM-Transitive}$$

$$\frac{}{\vdash \mathbb{I} \leq <?\ \texttt{extends ReadOnly} >} \text{ SM-AllReadonly}$$

$$\frac{}{\vdash \texttt{Mutable} \leq <?\ \texttt{extends Writable} >} \text{ SM-MutWritable}$$

Figure 32: Subtype relationships between mutability values

$$\frac{}{\mathit{Grounded}(n)}$$

In particular, $<?$ `extends ReadOnly` $>$, $<?$ `extends Writable` $>$, and $<$ $?$ `extends Nullable` $>$ are not *Grounded*: we will not allow them to be used for field assignment or field declaration.

### 7.5.2 Well-formedness of $\Gamma$, $\Lambda$

A well-formed context requires only that $\forall y \in \Gamma : \mathit{TypeWF}_{\Omega}(\Gamma(y))$, with $\Gamma(y)$ being a well-defined function into the space of types.

Formally, we define $\Gamma$ as a map:

$$\Gamma : \text{Variable Identifiers} \ \rightarrow \ \text{Types}$$

e.g. $\Gamma(y) = \sigma$.

A well-formed time-environment, $\Lambda$, is a set of delayed time variables, of the form:

$$\Lambda = \{\overline{\Theta}\}$$

A time-environment need only specify commitment points in the future; all others are either `Now`, or parametric and are treated as unknown. Note that there ss no requirement for an ordering on time. We define the judgements about time in Figure 31. It is convenient to write $t \leq \texttt{Now}$ when $t$ is committed.

We now have everything that is required to define the subtyping relationships in Figure 37 (which is augmented by the rules in Figures 32, 33, 34, 35).

$$\frac{\Lambda \vdash \sigma \leq \mathbb{I} \qquad \vdash \mathbb{I} \leq \mathbb{I}'}{\Lambda \vdash \sigma \leq \mathbb{I}'}\ \text{SM-TransitiveFullTypes}$$

$$\frac{\sigma = \texttt{Now} \rightarrow C[\texttt{Immutable}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >}{\Lambda \vdash \sigma \leq \texttt{Immutable}}\ \text{SM-CommittedImmutable}$$

$$\frac{\sigma = \Theta \rightarrow C[\texttt{Mutable}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >}{\Lambda \vdash \sigma \leq \texttt{Mutable}}\ \text{SM-MutableAlwaysMutable}$$

$$\frac{\begin{array}{c}\sigma = \Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > \\ \Lambda \vdash \Theta > \texttt{Now}\end{array}}{\Lambda \vdash \sigma \leq <?\ \texttt{extends Writable} >}\ \text{SM-UnCommittedCanMutate}$$

Figure 33: Rules for relating full types to mutability values

$$\frac{}{\vdash \mathbb{N} \leq \mathbb{N}}\ \text{SN-Reflexive}$$

$$\frac{\vdash \mathbb{N}_1 \leq \mathbb{N}_2 \qquad \vdash \mathbb{N}_2 \leq \mathbb{N}_3}{\vdash \mathbb{N}_1 \leq \mathbb{N}_3}\ \text{SN-Transitive}$$

$$\frac{}{\vdash \mathbb{N} \leq <?\ \texttt{extends Nullable} >}\ \text{SN-PossiblyNull}$$

Figure 34: Rules for subtyping of `null`ity constraints

$$\frac{\vdash \varsigma \leq \mathbb{N}}{\vdash (\Theta \rightarrow \varsigma)^{\epsilon} \leq \mathbb{N}}\ \text{SN-Timeless}$$

$$\frac{\varsigma = C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >}{\vdash \varsigma \leq \mathbb{N}}\ \text{SN-Shorthand}$$

$$\frac{\vdash \varsigma \leq \mathbb{N} \qquad \vdash \mathbb{N} \leq \mathbb{N}'}{\vdash \varsigma \leq \mathbb{N}'}\ \text{SN-TransitiveFullTypes}$$

Figure 35: Rules for relating fully-qualified types to `null`ity constraints. Note that time constraints play no part here, so we can make judgements about unqualified types.

$$\frac{}{\Omega \vdash C \leq C}\ \text{SC-Reflexive}$$

$$\frac{\Omega \vdash C \leq D \qquad \Omega \vdash D \leq E}{\Omega \vdash C \leq E}\ \text{SC-Transitive}$$

$$\frac{\Omega(C) = C[i] < \bar{i}, \bar{n} > \ \texttt{extends}\ D\{\ldots\}}{\Omega \vdash C \leq D}\ \text{SC-Extends}$$

Figure 36: Rules for class subsumption

## 7.6 The Type System

We define the typing rules for the system in Figures 39 and 40.

When typing a program (a single expression, $e$), we begin with an empty environment and time-environment.

Notice that we use a type subsumption rule, and also a similar rule for generalizing an environment (this is useful when we want to "merge" disparate environments resulting from the paths taken in a branch operation: specifically, possibly-null dereference). We define the relationship $\Gamma \subseteq \Gamma'$ in Figure 38.

### 7.6.1 Typing Method Calls

When they occur during a program, method calls are parametrized by their receiver, and all their arguments. The signature of a method is flexible, so that it can correctly handle, for example, the initialization of several non-committed objects. The bounds on the types objects that can be passed to a method are given in two ways: the types of the arguments, $\overline{\sigma}$ which may or may not be fully parametrizable (e.g. the programmer can specify that the arguments must be immutable); and a set of future times $\Sigma$(which indicate which of the arguments are still under initialization). When we discuss the well-formedness of a method definition, we will place some constraints on $\Sigma$ in order to specify what is a correctly-written program, but recall from Figure 29 that it is a set of formal time parameters ($\theta$).

We will wish to make judgements about whether or not a set of arguments is "correct" in terms of a method signature; we will need to make a judgement about whether or not the concrete argument types are compatible with the parametrizable method signature, and whether or not the commitment points on those parametrized types are appropriate for the time constraints, $\Sigma$, which are placed on the signature.

Further, we will need to take account of the initialization that a method performs on its uncommitted arguments. To do this, a method signature contains a mapping, $\Psi$, which specifies which fields it guarantees to initialize. We will need to update the type information of objects at the call-site with new initialization state. Formally, we define $\Psi$ as a map from argument identifiers to a set of field identifiers:

$$\frac{}{\Omega, \Lambda \vdash \sigma \leq \sigma}\ \text{S-Reflexive}$$

$$\frac{\Omega, \Lambda \vdash \sigma \leq \sigma' \qquad \Omega, \Lambda \vdash \sigma' \leq \sigma''}{\Omega, \Lambda \vdash \sigma \leq \sigma''}\ \text{S-Transitive}$$

$$\frac{\begin{array}{c}\sigma_1 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon_1} \\ \sigma_2 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon_2} \\ \epsilon_2 \subseteq \epsilon_1\end{array}}{\Omega, \Lambda \vdash \sigma_2 \leq \sigma_1}\ \text{S-Init}$$

$$\frac{}{\Omega, \Lambda \vdash (\texttt{Now} \rightarrow \varsigma)^{\epsilon} \leq (\texttt{Now} \rightarrow \varsigma)^{\emptyset}}\ \text{S-FullyInit}$$

$$\frac{\begin{array}{c}\sigma_1 = (\Theta \rightarrow C[\mathbb{I}'_0, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon} \\ \sigma_2 = (\Theta \rightarrow C[\mathbb{I}'_0, \mathbb{N}] < \overline{\mathbb{I}'}, \overline{\mathbb{N}} >)^{\epsilon} \\ \vdash \overline{\mathbb{I}' \leq \mathbb{I}}\big|_{0\ldots n}\end{array}}{\Omega, \Lambda \vdash \sigma_2 \leq \sigma_1}\ \text{S-Mut}$$

$$\frac{\begin{array}{c}\sigma_1 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}_0] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon} \\ \sigma_2 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}'_0] < \overline{\mathbb{I}}, \overline{\mathbb{N}'} >)^{\epsilon} \\ \vdash \overline{\mathbb{N}' \leq \mathbb{N}}\big|_{0\ldots n}\end{array}}{\Omega, \Lambda \vdash \sigma_2 \leq \sigma_1}\ \text{S-Null}$$

$$\frac{\begin{array}{c}\sigma_1 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon} \\ \sigma_2 = (\Theta \rightarrow C[\mathbb{I}, \texttt{NotNull}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}\end{array}}{\Omega, \Lambda \vdash \sigma_2 \leq \sigma_1}\ \text{S-NotNull}$$

$$\frac{\begin{array}{c}\sigma_1 = (\Theta \rightarrow D[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}\big|_{1\ldots i}, \overline{\mathbb{N}}\big|_{1\ldots j} >)^{\epsilon} \\ \sigma_2 = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}\big|_{1\ldots i'}, \overline{\mathbb{N}}\big|_{1\ldots j'} >)^{\epsilon} \\ \Omega(C) = C[i] < \overline{i}, \overline{n} > \ \texttt{extends}\ D\{\ldots\} \\ \Omega(D) = D[i] < \overline{i}\big|_{1\ldots i}, \overline{n}\big|_{1\ldots j} > \ \texttt{extends}\ E\{\ldots\} \\ i \leq i' \wedge j \leq j'\end{array}}{\Omega, \Lambda \vdash \sigma_2 \leq \sigma_1}\ \text{S-Class}$$

Figure 37: Full type subsumption relationship. Note that the program definitions are only required in the final judgement – this is where we account for (single) inheritance. This definition relies on the auxiliary rules defined in Figure 33, Figure 35, and Figure 36

$$\forall y \in \Gamma' : \Lambda \vdash \Gamma(y) \leq \Gamma'(y)$$
$$\equiv$$
$$\Lambda \vdash \Gamma \subseteq \Gamma'$$

Figure 38: Environment subsumption

$$\Psi : \text{Argument Identifier} \rightarrow \text{ set of } \textit{Field Id}\text{s}$$

These considerations make the typing rule for method call more complicated than the other typing rules, and it is be given separately (in Figure 41), with some explanation, as well as in-line with the other typing rules.

The rule itself can be read as follows:

- Find the fully-qualified type of the receiver and arguments, checking that the receiver is `NotNull`,
- From the receiver's type, find the method's signature (including return type and an environment with which our arguments must be compatible),
- Check that any arguments expected to be uninitialized are do indeed have commitment points in the future,
- The initialization information given by $\Psi$ is mapped into the local context, which each local variable's new initialization state being given by the union of all initialization promises made by the method signature for any of the arguments filled by that variable.

Recall that fully-qualified types are of the form:

$$(\Theta \rightarrow \varsigma)^{\epsilon}$$

We can think of this as a statement about an object with this type:

> "The object will be committed at time $\Theta$ to type $\varsigma$ . Until then, the fields $\epsilon$ are not guaranteed to be initialized."

With that in mind,we define a new syntax to help us with this rule:

$$((\Theta \rightarrow \varsigma)^{\epsilon})_{\epsilon'} \equiv (\Theta \rightarrow \varsigma)^{\epsilon \setminus \epsilon'}$$

which will be short-hand for taking account of newly initialized fields. We also define a further overload on $\Omega$ , taking a fully-qualified type, a method id, a set of mutability conditions, and a set of `null`ity conditions, and returning an effects map, the time constraints, the return type, and an environment associated with a method. The first three come (with some substitutions) from the method signature, and the environment is a more specialised version of the environment with which the method's body is type-checked:

$$\frac{\Omega, \Lambda \vdash \sigma_1 \leq \sigma_2 \qquad \Omega, \Gamma, \Lambda \vdash e : \sigma_1, \Gamma'}{\Omega, \Gamma, \Lambda \vdash e : \sigma_2, \Gamma'} \text{ (T-Subsum)}$$

$$\frac{\Omega, \Lambda \vdash \Gamma_1 \subseteq \Gamma_2 \qquad \Omega, \Gamma, \Lambda \vdash e : \sigma, \Gamma_1}{\Omega, \Gamma, \Lambda \vdash e : \sigma, \Gamma_2} \text{ (T-EnvSubsum)}$$

$$\frac{y \in \Gamma}{\Omega, \Gamma, \Lambda \vdash y : \Gamma(y), \Gamma} \text{ (T-env)}$$

$$\frac{\begin{array}{c} \Omega, \Gamma, \Lambda \vdash e : \sigma', \Gamma' \\ \Omega, \Gamma', \Lambda \vdash p : \sigma, \Gamma' \\ \sigma = (\Theta \rightarrow \varsigma)^{\epsilon} \\ \Lambda \vdash \sigma \leq \texttt{Writable} \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega(\varsigma, f) = \varsigma' \\ \sigma'' = (\Theta \rightarrow \varsigma')^{notNullFields(\varsigma')} \\ FieldType_{\Omega}(\sigma'') \\ \Omega, \Lambda \vdash \sigma' \leq \sigma'' \\ \Gamma'' = \Gamma'[p \mapsto \Gamma'(p)_{\{f\}}] \end{array}}{\Omega, \Gamma, \Lambda \vdash p.f := e : \sigma'', \Gamma''} \text{ (T-FieldAss)}$$

$$\frac{\begin{array}{c} \Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \sigma = (\Theta \rightarrow \varsigma)^{\epsilon} \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega(\varsigma, f) = \varsigma' \\ \Lambda \vdash \Theta > \texttt{Now} \\ f \in \epsilon \\ \sigma' = possiblyNull((\Theta \rightarrow \varsigma')^{notNullFields(\varsigma')}) \end{array}}{\Omega, \Gamma, \Lambda \vdash p.f : \sigma', \Gamma} \text{ (T-FieldLookupUninit)}$$

$$\frac{\begin{array}{c} \Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \sigma = (\Theta \rightarrow \varsigma)^{\epsilon} \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega(\varsigma, f) = \varsigma' \\ f \notin \epsilon \vee \Lambda \vdash \Theta = \texttt{Now} \\ \sigma' = \begin{cases} (\Theta \rightarrow \varsigma')^{\emptyset} \text{ if } \Theta = \texttt{Now} \\ (\Theta \rightarrow \varsigma')^{notNullFields(\varsigma')} \text{ otherwise} \end{cases} \end{array}}{\Omega, \Gamma, \Lambda \vdash p.f : \sigma', \Gamma} \text{ (T-FieldLookup)}$$

$$\frac{\Omega, \Gamma, \Lambda \vdash e_1 : \sigma_1, \Gamma' \qquad \Omega, \Gamma', \Lambda' \vdash e_2 : \sigma_2, \Gamma''}{\Omega, \Gamma, \Lambda \vdash e_1; e_2 : \sigma_2, \Gamma''} \text{ (T-Concat)}$$

Figure 39: Typing rules for programs pt.1

$$\frac{\begin{array}{c} t \notin \Lambda \\ \Lambda' = \{t\} \cup \Lambda \\ \Omega, \Gamma, \Lambda' \vdash e : \sigma, \Gamma' \\ \sigma = (\Theta \rightarrow \varsigma)^{\epsilon} \\ \forall y \in \Gamma' : \mathit{time}(\Gamma'(y)) = t \implies \mathit{uninit}(\Gamma'(y)) = \emptyset \\ \Gamma'' = \Gamma'[\texttt{Now}/t] \\ \sigma' = \begin{cases} (\texttt{Now} \rightarrow \varsigma)^{\emptyset} \text{ if } \Theta = t \\ \sigma \text{ otherwise} \end{cases} \end{array}}{\Omega, \Gamma, \Lambda \vdash \ \texttt{delay}\ \ t\{e\} : \sigma', \Gamma''} \text{(T-InitRegion)}$$

$$\frac{\begin{array}{c} C \in \Omega \\ \sigma = (\Theta \rightarrow C[\mathbb{I}, \texttt{NotNull}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon} \\ \Omega \vdash C[\mathbb{I}, \texttt{NotNull}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > \ \texttt{Well formed} \\ \epsilon = \mathit{notNullFields}(\sigma) \\ \Gamma' = \Gamma[z \mapsto \sigma] \\ \Lambda \vdash \Theta > \texttt{Now} \end{array}}{\Omega, \Gamma, \Lambda \vdash \texttt{alloc}\ \ \Theta \rightarrow C[\mathbb{I}, \texttt{NotNull}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} > \ \ \texttt{as}\ \ z : \sigma, \Gamma'} \text{(T-Allocate)}$$

$$\frac{\begin{array}{c} \Gamma(y) = \sigma \\ \sigma = (\Theta \rightarrow C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon} \\ \Omega, \Gamma, \Lambda \vdash e_1 : \sigma', \Gamma' \\ \Gamma_{notnull} = \Gamma[y \mapsto \sigma[\texttt{NotNull}/\mathbb{N}]] \\ \Omega, \Gamma_{notnull}, \Lambda \vdash e_2 : \sigma', \Gamma' \end{array}}{\Omega, \Gamma, \Lambda, \vdash \texttt{ifnull}\ \ y\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2 : \sigma', \Gamma'} \text{(T-NullDeref)}$$

$$\frac{\begin{array}{c} \Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega, \Gamma, \Lambda \vdash \overline{y : \sigma}, \Gamma \\ \Omega(\sigma, m, \overline{\mathbb{I}}, \overline{\mathbb{N}}) = \Psi, \Sigma, \sigma', \Gamma' \\ \Lambda \vdash \Gamma \subseteq \Gamma'[\overline{y/x}] \\ \Sigma[\overline{\mathit{time}(\sigma)/\theta}] \subseteq \Lambda \\ \overline{\epsilon_y = \bigcup_{y_j = y} \Psi(x_j)} \\ \Gamma' = \Gamma[\overline{y \mapsto \Gamma(y)_{\epsilon_y}}] \end{array}}{\Omega, \Gamma, \Lambda \vdash < \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y}) : \sigma', \Gamma'} \text{(T-MethCall)}$$

Figure 40: Typing rules for programs pt.2

$$\frac{\begin{array}{c}\Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega, \Gamma, \Lambda \vdash \overline{y : \sigma}, \Gamma \\ \Omega(\sigma, m, \overline{\mathbb{I}}, \overline{\mathbb{N}}) = \Psi, \Sigma, \sigma', \Gamma' \\ \Lambda \vdash \Gamma \subseteq \Gamma'[\overline{y/x}] \\ \Sigma[\overline{\mathit{time}(\sigma)/\theta}] \subseteq \Lambda \\ \overline{\epsilon_y = \bigcup_{y_j = y} \Psi(x_j)} \\ \Gamma' = \Gamma[\overline{y \mapsto \Gamma(y)_{\epsilon_y}}]\end{array}}{\Omega, \Gamma, \Lambda \vdash < \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y}) : \sigma', \Gamma'} \text{ (T-MethCall)}$$

Figure 41: Rule for typing Method Call

If

$$\Omega(D, m) = \Psi, \Sigma, \sigma_{\alpha+1}, (\sigma_0, \overline{\sigma}\big|_{1\ldots\alpha})$$

$$\sigma_0 = (\Theta \to C[\mathbb{I}_0, \mathbb{N}_0] < \overline{\mathbb{I}}\big|_{1\ldots i}, \overline{\mathbb{N}}\big|_{1\ldots j} >)^{\epsilon}$$

$$\sigma_0' = (\Theta' \to D[\mathbb{I}_0', \mathbb{N}_0'] < \overline{\mathbb{I}'}\big|_{1\ldots i'}, \overline{\mathbb{N}'}\big|_{1\ldots j'} >)^{\epsilon'}$$

$$\forall k \in 0 \ldots \alpha + 1 : \sigma_k^* = \sigma_k[\Theta'/\theta, \overline{\mathbb{I}'/i}\Big|_{0\ldots n}, \overline{\mathbb{N}'/n}\Big|_{0\ldots m}]$$

$$\Sigma \vdash \sigma_0' \leq \sigma_0^*$$

$$\Gamma' = \{\overline{x \mapsto \sigma^*}\}$$

Then

$$\Omega(\sigma_0', m, \overline{\mathbb{I}}\big|_{i'+1\ldots n}, \overline{\mathbb{N}}\big|_{j'+1\ldots m}) = \Psi, \Sigma, \sigma_{x+1}, \Gamma'$$

Finally, we give the judgement for method typing in Figure 41

### 7.6.2 Well-formed Programs

In order to talk about the correctness of our type system with respect to our runtime semantics, we will need to specify what qualifies as a correctly-written program. We will require that all class definitions are properly specified, which will in turn need us to check that all the fields and methods are properly specified.

**Well-formedness of a method** We will need to be able to type-check a method against its most general set of arguments. If the method type checks correctly in this case, then we will say that it is *well-formed.*

Each argument to a method has a type of the form

$$(\Theta \to C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}$$

with each $\mathbb{I}$, $\mathbb{N}$ being parametrizable. We need to be able to type-check the body of a method, given that we do not know the concrete types of each argument ahead of time. Since the type-system allows us to handle parametrizable types (i.e. types which contain formal parameters) without modification, this is not a problem. We do require, however, that any appearance of a parametrizable type within the body of the method is fully-specified by the method's arguments.

In order to type-check the method's body, we will need to construct an environment (and a time-environment). For a program with

$$\Omega(C,m) = \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}$$

We will construct a time-environment $\Lambda_{C,m}$ directly from the method signature, which specifies the commitment points which are in the future:

$$\Lambda_{C,m} = \Sigma$$

Recall that $\Sigma$ is a set of formal time parameters, which are the commitment points of zero or more of the method arguments.

We can construct an environment for type-checking, by writing:

$$\Gamma_{C,m}(\texttt{this}) = \sigma_{rec}$$

$$\overline{\Gamma_{C,m}(x) = \sigma}$$

Finally, we make the following judgement:

$$\frac{\begin{array}{c}\Omega(C,m) = \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\} \\ \mathit{TypeWF}_{\Omega}(\sigma_{rec}), \mathit{TypeWF}_{\Omega}(\sigma_{ret}), \mathit{TypeWF}_{\Omega}(\overline{\sigma}) \\ \Omega, \Gamma_{C,m}, \Lambda_{C,m} \vdash e : \sigma_{ret}, (\Gamma') \\ \forall i \in [1..n] : \Lambda_{C,m} \vdash \Gamma'(x_i) \leq \sigma_{i_{\Psi(x_i)}} \\ \Lambda_{C,m} \vdash \Gamma'(\texttt{this}) \leq \sigma_{rec_{\Psi(this)}} \\ \forall x \in \Gamma' \setminus \Gamma : \mathit{uninit}(\Gamma'(x)) = \emptyset\end{array}}{\Omega \vdash C.m \text{ well-formed}}$$

**Well-formed classes** The definition of a well-formed class is relatively straightforward. For a $C$ to be well-formed, we require that:

$$\Omega(C) = C[i,n] < \overline{i}, \overline{n} > \ \texttt{extends}\ D\{\overline{f : \varsigma}, \overline{m : \Psi, \Sigma, \sigma_{ret}(\sigma_{rec}, \overline{\sigma})\{e\}}\}$$

with

$$D \in \mathit{ClassId} \cup \{\texttt{Object}\}$$

$$D \neq \texttt{Object} \implies D \text{ well-formed}$$

and

$$\Omega \vdash \overline{C.m \text{ well-formed}}$$

i.e. the class must either extend some other well-formed class definition, or extend a special class "`Object`," and all the class's methods must be well-formed.

We further require:

| | |
|---|---|
| $\iota \in$ Addresses | |
| $v ::= \iota \mid \texttt{null}$ | Values |
| $f \in \textit{Field Id}$ | |
| $o ::= (C, \{f \mapsto v\})$ | Objects |
| $\phi$ : Variable Identifiers $\rightarrow$ Values | Stack Lookup |
| $\psi$ : Addresses $\rightarrow$ Objects | Heap Lookup |
| $\psi$ : Addresses $\times$ *Field Id*$\rightarrow$ Values | Field lookup |

Note that $\psi(\iota, f)$ is equivalent to $\psi(\iota) \downarrow_2 (f)$ – an object is a tuple of a class identifier and a map from field identifiers to values.

Figure 42: We define stacks, heaps, and values

- All methods appearing in $D$ must also appear in $C$, with *identical* method signature (but possibly different method body), and all fields appearing in $D$ must also appear in $C$, with the *identical* type. In this system, "extending" another class is a promise to implement its interface in full.
- $\forall f \in \textit{Fields}(C) : \textit{FieldType}(\Omega(C, f))$

**Well-formed programs** A well-formed program requires only two conditions:

- All classes declared in the program are well-formed according to the rules above
- There exists a class `Main` with a method `Main.main` accepting as receiver a mutable, non-`null` instance of `Main`. The class should have no further methods, no fields, and no extra formal parameters.

## 7.7 Runtime semantics

We define the stack, heap, and values in Figure 42. An Object is a tuple of a class identifier, and a number of field-identifier/value pairs. A value is either an address, or `null`. A heap is a function mapping addresses to objects, and for convenience, we extend the heap to map address/field-identifier pairs to values (i.e. we extend it with field lookup on objects).

A stack is a map from variable identifiers to values:

$$\phi : \text{ Variable Identifiers } \rightarrow \text{ Values}$$

We then define the run-time reduction semantics of our system in Figure 43, written in the form

$$\psi, \phi \vdash e \rightsquigarrow v, \psi', \phi'$$

with $v$, a value, the result of evaluating the expression $e$ on heap $\psi$ with stack $\phi$, and $\psi', \phi'$ the resulting heap and stack after that evaluation completes. A complete program is run by evaluating a single expression on an empty stack and heap (formally, the initial domain of each partial mapping is the empty set).

None of the runtime rules is complex enough to warrant special attention.

## 7.8 Design decisions

Before we go on to discuss proofs of the system's formal properties, we will briefly describe some of the design decisions made during the formulation of the type system just presented. The reader may freely skip ahead to the next section; this section is intended to provide a rationale for some of the features of the system, as well as document the process of designing it (in particular, some of the pitfalls we came across on the way).

**Full type information of fields** Since we are storing a list of definitely assigned fields on each type, we considered also including the full type information of the values assigned to those fields, so that if they were assigned further up in the same method, the user could use the more specific type information (rather than the declared type in the class declaration). This is attractive from the point of view of allowing the user to use all the information we can infer from the structure of the program, but adds more complexity, and may prove to be unsound in the context of multi-threading. (We might know what the concrete type of a field is when we assigned it, but perhaps it will be changed before we come to use that information further down in the same method).

### 7.8.1 Changing time environments

When we were formulating the type rules, it was intended that we would keep track of both commitment points that had not yet been reached (i.e. "which objects are not yet initialized?"), *and* commitment points which had passed (i.e. "which objects can be judged to be fully initialized?"). The idea was that we could make a judgement about whether the commitment point associated with any given object was in the past or not. Instead, we now just substitute the commitment point of a type with `Now` when it becomes committed (this is in line with [4]), so we need only keep track of those objects which are *not* committed.

Under the latter system, the only possibility to change the time environment is when we evaluate a sub-expression, within a `delay` construct (we evaluate the sub-expression with a new time-environment, but do not change the surrounding one). This means that we needn't keep track of a changing time environment. The new system is as expressive as the old system, since we only used the set of "passed times" to judge that an object was committed: now we just compare its commitment point to `Now`.

### 7.8.2 Subtyping between types with different commitment points

Originally, we intended to allow a subtyping relationship between types with different commitment points. This would allow the programmer to write, for

$$\frac{\phi(y) = v}{\psi, \phi \vdash y \rightsquigarrow v, \psi, \phi} \text{ (R-VAR)}$$

$$\frac{\begin{array}{cc} & \psi'(\iota) \downarrow_1 = C \\ \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi' & f \in \mathit{Fields}(C) \\ \psi', \phi' \vdash p \rightsquigarrow \iota, \psi', \phi' & \psi'' = \psi'[\iota \mapsto (C, \psi'(\iota) \downarrow_2 [f \mapsto v])] \end{array}}{\psi, \phi \vdash p.f := e \rightsquigarrow v, \psi'', \phi'} \text{ (R-FieldAss)}$$

$$\frac{\begin{array}{c} \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi \\ \psi(\iota, f) = v \end{array}}{\psi, \phi \vdash p.f \rightsquigarrow v, \psi, \phi} \text{ (R-FieldLookup)}$$

$$\frac{\begin{array}{c} \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi' \\ \psi', \phi' \vdash e' \rightsquigarrow v', \psi'', \phi'' \end{array}}{\psi, \phi \vdash e; e' \rightsquigarrow v', \psi'', \phi''} \text{ (R-Concat)}$$

$$\frac{\psi, \phi \vdash e \rightsquigarrow v, \psi', \phi'}{\psi, \phi \vdash \ \texttt{delay}\ t\{e\} \rightsquigarrow v, \psi', \phi'} \text{ (R-InitRegion)}$$

$$\frac{\begin{array}{cc} \iota \notin \psi & C := \mathit{Class}(\sigma) \\ y \notin \phi & \mathit{Fields}(C) = \{\overline{f}\} \\ \phi' = \phi[z \mapsto \iota] & \psi' = \psi[\iota \mapsto (C, \overline{f = \texttt{null}})] \end{array}}{\psi, \phi \vdash \texttt{alloc}\ \sigma\ \texttt{as}\ z \rightsquigarrow \iota, \psi', \phi'} \text{ (R-Allocate)}$$

$$\frac{\phi(y) = \texttt{null} \qquad \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi'}{\psi, \phi \vdash \texttt{ifnull}\ y\ \texttt{then}\ e\ \texttt{else}\ e' \rightsquigarrow v, \psi', \phi'} \text{ (R-NullDeref)}$$

$$\frac{\phi(y) = \iota \in \psi \qquad \psi, \phi \vdash e' \rightsquigarrow v, \psi', \phi'}{\psi, \phi \vdash \texttt{ifnull}\ y\ \texttt{then}\ e\ \texttt{else}\ e' \rightsquigarrow v, \psi', \phi'} \text{ (R-NotNullDeref)}$$

$$\frac{\begin{array}{c} \psi, \phi \vdash p \rightsquigarrow v \\ \psi(v) \downarrow_1 = C \\ \mathit{body}(C, m) = e \\ \psi, \phi\big|_{\overline{y}}[\texttt{this} \mapsto v][x/y] \vdash e \rightsquigarrow v', \psi', \phi' \end{array}}{\psi, \phi \vdash p.m(\overline{y}) \rightsquigarrow v', \psi', \phi} \text{ (R-MethCall)}$$

Figure 43: Runtime semantics for programs.

example:

```
delay t1 {
    alloc t1 -> C[Immutable, NotNull]<> as y;
    delay t2 {
        alloc t2 -> D[Immutable, NotNull]<> as x;
        /* Assigning an object with commitment point t2 to the
            field of one with commitment point t1 will cause a
            problem */
        y.field = x;
    }

    /* This has the same problem (this time we assign an object
        with commitment point Now to one with commitment point t1):
         */
    y.field = x;

    /* Allows (with D.mutate() being some method requiring a
        Writable instance of D): */
    y.field.mutate()
    /* which is clearly incorrect */
}
```

The problem with this is that when $y$ remains uncommitted at the end of delay scope $t_2$, its field, $y.field$ is also treated as uncommitted (i.e. our type system treats $y.field$ as `Writable`). In [4], this sort of field assignment is allowed: when we look up fields of an uninitialized object under that system, we know that the value in the field has commitment point *no later than* that of the object. In the context of `null`ity constraints alone, this is enough.

Because of the above, we do not allow types with different commitment points to have a subsumption relationship. Note that, when two objects are committed, we treat them both as having `Now` as their commitment point. The following is allowed:

```
delay t1 {
        alloc t1 -> C[Immutable, NotNull]<> as y;
        alloc t1 -> D[Immutable, NotNull]<> as x;
    /* This is safe, since both are committed at the same time */
        y.field = x;

    /* Safe since x is still uncommitted */
    y.field.mutate();
}

/* Not allowed by the type system, since y, and thus any field of y
    , is committed */
y.field.mutate()
```

### 7.8.3 Field Assignment

The type rules for field assignment go further than just checking that the value being assigned subtypes the type of the field; it checks that this field has '*Grounded*' type. This is because we need to avoid the situation where the

```
class C[i0]<i1,> extends object {
    f : D[i1, Nullable]<,>

    void m(
        /* We don't care for the committment points of these
            objects, so long as they are the same */
        T -> C[Mutable, NotNull]<<? extends ReadOnly>,> this,
        T -> D[<? extends ReadOnly>, Nullable]<,> x) {

        /* The type of x is exactly the type of this.f in this
            expression */
        this.f = x;
        /* The problem was that we allowed field assignment when we
             didn't concretely know the type required for this.f */
    }
}

...
delay t {
    alloc t -> C[Mutable, NotNull]<Immutable,> as z1;
    alloc t -> D[Mutable, NotNull]<,> as z2;
}
/* Fine to treat z1 as C[Mutable, NotNull]<<? extends ReadOnly>,>
   and to treat z2 as D[<? extends ReadOnly>, Nullable]<,> */
z1.m(z2);

/* We have just put a Mutable reference in z1's Immutable field! */
```

Figure 44: Figure showing that field assignment when the type of the field and value are not concrete is unsafe.

field and the value are both typed as `ReadOnly` or `Writable` in a method, but their types at the call-site are incompatible. For example see Figure 44 (which omits effects and method-initialization-promises and assuming the presence of a `void` type):

It is easy to construct a similar situation with fields that might be `Nullable` (see Figure 45).

The solution is that when we do not concretely know the mutability or `null`ity of a field (i.e. when it is the upper bound of some more concrete possibilities, such as $<?$ `extends ReadOnly` $>$, or $<?$ `extends Writable` $>$), we cannot assign to it. This is not surprising: covariance of generic arguments is also a problem in Java, and it gives rise to the `<? extends ...>` construct. We enforce that this kind of assignment is not allowed by the requirement $FieldType(\sigma'')$ in the rule (T-FieldAss).

### 7.8.4 Nullity of the receiver is not parametric

It would be unsafe to allow method definitions (or field types) to be parametric in the `null`ity of the receiver. The easiest way to illustrate this is through an example: in Figure 46, we show a situation where we allow such parametric

```
class C[i0]<,n1> extends object {
    g : D[i0, n1]<,>

    void m(C[i0, NotNull]<,<? extends Nullable>> this,
                D[i0, <? extends Nullable>]<,> x) {
        /* The type of x is exactly the type of this.g in this
            expression */
        this.g = x
        /* The problem is as before: we allowed field assinment
            when we didn't concretely know the type required for
            this.g */
    }
}

...
delay t {
    alloc C[Mutable, NotNull]<,NotNull> as z1;
    alloc D[Mutable, NotNull]<,> as z2;
    /* Must initialize z1.g since it is a NotNull field */
    z1.g = z2;
};

/* null is acceptable in place of D[Mutable, <? extends Nullable
    >]<,> */
z1.m(null);
/* z1.g is now Null! */
```

Figure 45: Code listing showing that field assignment when concrete type is not known is also unsafe in the case of `null`ity.

```
/* The definition of the class is parametric in its own nullity */
class C[n0] extends object {
    /* The type of f is parametric in how we see the nullity of the
         receiver */
    f : C[n0]

    void
    /* Formal time parameter T is in the future */
    {T}
    /* this is NotNull, but x might be null */
    m(T -> C[NotNull] this,
        T -> C[Nullable] x) {

        ifnull x then
            ...
        else:
            /* Now we see x as C[NotNull] and x.f as C[NotNull] */
            this.f = x.f

        this
    }
}

...
delay t {
    alloc t -> C[NotNull] as z0;
    z0.f = z0; // z0 is now considered initialized
    alloc t -> C[NotNull] as z1;
    z0.f(z1); // Sets z0.f to null!
    z1.f = z1; // z1 is now initialized
}
/* z0.f == null, even though z0 is considered initialized! */
```

Figure 46: Code demonstrating that allowing fields to be parametric in their receiver's `null`ity is not safe

definitions (omitting annotations for mutability and optional type parameters).

# 8 Soundness

We wish to be able to specify the goodness of a program, in terms of how it operates on a Heap and Stack. We wish to express that, given that a program has been typed by the rules in Figures 39 and 40, it will transform a heap that is in a "good state" into another heap, also with a "good state," and will, eventually, evaluate to some result (a value).

We will show in this section that the combination of a well-typed expression and a well-formed program will result in a complete evaluation to a result. We will show that a well-typed expression has the following properties:

**Progress** Every program should evaluate to some result, or contain an infinite loop (e.g. in the case of recursive function call). We will not prove this property.

**Well-formedness** We define well-formedness in Section 47. Briefly, we require that object on the heap have the fields we expect them to have, with the `null`ity we expect them to have.

**Mutability** We define the Mutability properties in Section 8.4. In short: the user cannot write to the fields of objects which are types as `Immutable`.

The rest of this section is concerned with showing that these properties hold in the system which we have thus far laid out. We will begin by describing an assumption that regards the way expressions are typed (our *generation lemma*), then move on to the body of the arguments. In order to formally express the properties above, we first need to define a notion of *consistency* between types (Section 8.1). We are then able to specify what it means for the execution environment to be well-formed with respect to associated type information (that is, what it means for the system to be in a "good state") in Section 8.2. From there, we go on to show that this good state is preserved by the execution of the program (Section 8.3). This preservation property, along with the properties of a well-formed stack and heap, are exactly what is required to guarantee that lookup of a value which the type system expects to be not-`null` will in fact not be `null`. Finally, we discuss what it means for an object to be immutable in Section 8.4, and formalize the guarantees that our system gives with regard to mutability.

**Generation Lemma** Throughout this section, proofs will revolve around the structure of derivations for types and runtime reductions as defined in Figures 39, 40 and 43. We will implicitly assume that these derivations are determined by the structure of the expressions to which they apply: in particular we will assume that, for example, when the type for an expression was derived, the conditions for the associated derivation rule were satisfied. Such a lemma is common in the literature, and is normally referred to as a Generation Lemma (or Inversion Lemma, as in [13])

**Lemma 8.1.** Generation Lemma *The most important case is field assignment; we will assume, for example, that the following holds:*

$$\Omega, \Gamma, \Lambda \vdash p.f = e : \sigma', \Gamma'' \implies \begin{cases} \exists \sigma' \ st. \ \Omega, \Gamma, \Lambda \vdash e : \sigma', \Gamma' \\ \exists \sigma \ st. \ \Omega, \Gamma', \Lambda \vdash p : \sigma, \Gamma' \\ \Lambda \vdash \sigma \leq <? \ \textit{\textbf{extends Writable}}> \\ \vdash \sigma \leq \textit{\textbf{NotNull}} \\ \sigma = (\Theta \to \varsigma)^{\epsilon} \\ \Omega(\varsigma f) = \varsigma' \\ \sigma'\prime = (\Theta \to \varsigma')^{notNullFields(\varsigma')} \\ FieldType_{\Omega}(\sigma'') \\ \Omega, \Lambda \vdash \sigma' \leq \sigma'' \\ \Gamma'' = \Gamma'[p \mapsto \Gamma'(p)_{\{f\}}] \end{cases}$$

*We assume similarly that expressions in any other form have been typed with the appropriate typing rule.*

*Proof.* We assume here that the lemma holds. □

## 8.1 Consistency

Consistency is a property about types: we will use to describe two different types being in some way compatible. The motivation for formalizing consistency is to allow us to reason about the different ways in which we can reach an address on the heap; we will argue that if there are two different ways of accessing the same object, then they impose similar restrictions. Clearly this is a desirable property from the point of view of arguing that the guarantees of the system really hold: for example, we would like a formal way to describe that an object cannot be reached via paths that describe it as both `Mutable` and `Immutable` at the same time, or paths that give differing types to its fields.

Consistency between *types* is not a property of a heap (or stack); consistency between *paths* will be – paths are a series of field lookups in a heap and stack. When we talk about consistency of paths, we really mean the consistency between the types given to those paths under a certain environment (and time-environment). It is impossible to talk about the consistency of paths outside the context of their type information, but it is quite possible to talk about the consistency of types without them necessarily being applied to any expression.

We start by formalizing what it means for two types to be compatible. The following definition is the intuition: two types are compatible if one subtypes the other. But as we will see, it is too strong a condition for us (that is to say: it does not hold in our system).

**Definition 8.1.** *Type compatibility (intuition)*

$$\frac{\Omega, \Lambda \vdash \sigma \leq \sigma' \vee \Omega, \Lambda \vdash \sigma' \leq \sigma}{\Omega, \Lambda \vdash \sigma \approx \sigma'}$$

Whilst this property is both symmetric and reflexive, it is not necessarily transitive. Consider:

$$\Omega, \Lambda \vdash (t \to \varsigma)^{\{b\}} \leq (t \to \varsigma)^{\{a,b\}}$$

$$\Omega, \Lambda \vdash (t \to \varsigma)^{\{b\}} \leq (t \to \varsigma)^{\{b,c\}}$$

so

$$(t \to \varsigma)^{\{b\}} \approx (t \to \varsigma)^{\{a,b\}}$$

and

$$(t \to \varsigma)^{\{b\}} \approx (t \to \varsigma)^{\{b,c\}}$$

but

$$(t \to \varsigma)^{\{a,b\}} \not\approx (t \to \varsigma)^{\{b,c\}}$$

In particular, this initial, strong condition is not an equivalence relation. Not only that, but it does not hold for types in our system which we *do* want to be able to consider to be compatible. Consider:

$$(\texttt{Now} \to C[<?\ \texttt{extends ReadOnly}>, \texttt{NotNull}]<,>)$$

$$\not\approx$$

$$(\texttt{Now} \to C[\texttt{Mutable}, <?\ \texttt{extends Nullable}>]<,>)$$

Now, it is perfectly reasonable to apply either of these types to an object which has the type:

$$(\texttt{Now} \to C[\texttt{Mutable}, \texttt{NotNull}]<,>)$$

What makes it reasonable to use either of the above types instead of the more specified one is that it is a subtype of them both, while neither one is a subtype of the other. The condition is too strong to apply to these types which we wish to consider compatible.

Now we give a new definition for type compatibility (which we call "consistency"), specifying that two types are consistent if and only if they have some common subtype:

**Definition 8.2.** *Type consistency*

$$\frac{\exists \sigma''\ st.\ \Omega, \Lambda \vdash \sigma'' \leq \sigma' \wedge \sigma'' \leq \sigma}{\Omega, \Lambda \vdash \sigma \sim \sigma'}$$

As before, it's clear that this property is both symmetric and reflexive. In general, this property is not transitive, but when considering different views on fields, we can show transitivity. The following lemma gives us an equivalence result for the types that can be given to a field, given that one of those types allows field assignment.

First let $\mathbb{S}$ be the set of maps from unqualified types to unqualified types by the action of substitution of formal parameters, e.g.:

$$s \in \mathbb{S} : s(C[i,n] < \bar{i}, \overline{n} >) = C[\texttt{Immutable}, \texttt{NotNull}] < \bar{\mathbb{I}}, \overline{\mathbb{N}} >$$

$$s' \in \mathbb{S} : s'(C'[i,n] <,>) = C'[\texttt{Mutable}, \texttt{Nullable}] <,>$$

We want to talk about functions in $\mathbb{S}$ because it allows us to characterize types which are associated with a field of an object. If

$$\Omega, \Gamma, \Lambda \vdash p.f : (\Theta \to \varsigma)^{\epsilon}$$

then

$$\exists s \in \mathbb{S}, C \in \Omega, \text{ st. } \sigma = (\Theta \to s(\Omega(C,f)))^{\epsilon}$$

(substitution on a lookup $\Omega(C,f)$ is exactly how we determine the type of a field).

**Lemma 8.2.** *Consistency is a transitive relationship through types which can be assigned to fields of objects with a given class.*

$$\left.\begin{array}{r} \Omega, \Lambda \vdash \sigma \sim \sigma' \\ \Omega, \Lambda \vdash \sigma' \sim \sigma'' \\ FieldType_{\Omega}(\sigma') \\ s, s' \in \mathbb{S} \\ E \in \Omega \\ \sigma = s(\Omega(E,f)) \\ \wedge\ \sigma' = s'(\Omega(E,f)) \end{array}\right\} \implies \Omega, \Lambda \vdash \sigma \sim \sigma''$$

*Proof.* Start by writing:

$$\sigma \equiv (\Theta \to C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}$$

$$\sigma' \equiv (\Theta \to C'[\mathbb{I}', \mathbb{N}'] < \overline{\mathbb{I}'}, \overline{\mathbb{N}'} >)^{\epsilon'}$$

$$\sigma'' \equiv (\Theta \to C''[\mathbb{I}'', \mathbb{N}''] < \overline{\mathbb{I}''}, \overline{\mathbb{N}''} >)^{\epsilon''}$$

Note that $\sigma \sim \sigma' \implies \mathit{time}(\sigma) = \mathit{time}(\sigma')$ and for the same reason $\mathit{time}(\sigma') = \mathit{time}(\sigma'')$ Then let:

$$\sigma_a = (\Theta \to D[\mathbb{I}_a, \mathbb{N}_a] < \overline{\mathbb{I}_a}, \overline{\mathbb{N}_a} >)^{\epsilon_a}$$

$$\sigma_b = (\Theta \to D'[\mathbb{I}_b, \mathbb{N}_b] < \overline{\mathbb{I}_b}, \overline{\mathbb{N}_b} >)^{\epsilon_b}$$

with

$$\Omega, \Lambda \vdash \sigma_a \leq \sigma \wedge \sigma_a \leq \sigma'$$

$$\Omega, \Lambda \vdash \sigma_b \leq \sigma' \wedge \sigma_b \leq \sigma''$$

(we know such types exist; this is the definition of the $\sim$ relationship)

Notice that we have single inheritance with respect to classes, so we know that either $\Omega \vdash D \leq D'$, or vice versa. Whilst the number of mutability and `null`ity parameters may be different across each of the types, these are dictated by the classes, with the highest number of parameters being at the bottom of the hierarchy. In particular, we know that since $\sigma_a$ and $\sigma_b$ both subtype $\sigma'$, the first $n$ mutability parameters and the first $m$ `null`ity parameters are the same (where $n$ and $m$ are the number of mutability and `null`ity parameters expected by class $C'$ respectively), with the possible exception of the first `null`ity parameter. Since $\exists E$, a class which specifies the form of both $\sigma$ and $\sigma'$ up to the number of mutability and `null`ity parameters, we know that they both have the same number, so if $\sigma_a$ and $\sigma_b$ have a differing set of mutability and `null`ity parameters, we can set all mutability parameters after position $n$ and all `null`ity parameters after position $m$ to be just those from $\sigma_b$ (yielding a new type, $\sigma'_a$, which satisfied all the subtype relationships that we had for $\sigma_a$); the resultant types will still subtype $\sigma$ and $\sigma'$. If there are more parameters in $\sigma''$ then we must pick the parameters after $n$ and $m$ to match with $\sigma''$ (which is safe for the same reason). We will see that if the first `null`ity parameter is not the same for both $\sigma_a$ and $\sigma_b$ it is safe to use `NotNull`.

We know (from inspection of the subtyping rules) that:

$$\mathbb{I}_a \leq \mathbb{I}'$$

$$\mathbb{I}_b \leq \mathbb{I}'$$

$$\mathbb{N}_a \leq \mathbb{N}'$$

$$\mathbb{N}_b \leq \mathbb{N}'$$

$$\overline{\mathbb{I}_a \leq \mathbb{I}'}$$

$$\overline{\mathbb{I}_b \leq \mathbb{I}'}$$

$$\overline{\mathbb{N}_a \leq \mathbb{N}'}$$

$$\overline{\mathbb{N}_b \leq \mathbb{N}'}$$

And since we know $\mathit{FieldType}(\sigma')$, we know that these relationships are in fact equality relationships (by inspection of the possibilities for the $\mathbb{I}$s and $\mathbb{N}$s) for all but the first `null`ity parameter. So:

$$\mathbb{I}_a = \mathbb{I}' = \mathbb{I}_b$$

$$\overline{\mathbb{I}_a = \mathbb{I}' = \mathbb{I}_b}$$

$$\overline{\mathbb{N}_a = \mathbb{N}' = \mathbb{N}_b}$$

The only chance for the first `null`ity parameter to be different is if one of the $\sigma^{\cdots}$ have `NotNull` as the first parameter, but the others have `Nullable`. In that case, notice that substituting `NotNull` for the first `null`ity parameter of $\sigma_a$ or $\sigma_b$ is does not violate the subtyping relationship (so, just re-write $\sigma_a$ or $\sigma_b$ with the `NotNull` parameter if necessary).

Without loss of generality, say $\Omega \vdash D' \leq D$. Then

$$\Omega, \Lambda \vdash \sigma_b \leq \sigma'_a$$

and so

$$\Omega, \Lambda \vdash \sigma_b \leq \sigma$$

which is exactly what we need for the statement:

$$\Omega, \Lambda \vdash \sigma \sim \sigma$$

□

Notice that the conditions in Lemma 8.2 hold whenever $\sigma$ and $\sigma''$ are the result of field lookup, and when $\sigma'$ permits field assignment (see rule (T-FieldAss)).

**Lemma 8.3.** *Basic Consistency:*

$$\left.\begin{array}{l}\Omega, \Gamma, \Lambda \vdash e : \sigma, \Gamma' \\ \Omega, \Gamma, \Lambda \vdash e : \sigma', \Gamma''\end{array}\right\} \quad \Longrightarrow \Omega, \Lambda \vdash \sigma \sim \sigma'$$

*Proof.* By induction on the structure of type derivations. In particular, by inspection of the last rule applied in the derivation. The last rule applied is uniquely determined (up to subsumption) by the structure of $e$, so the last rule applied in the derivation of each type $\sigma$ and $\sigma'$ will be the same; it suffices to show that the types derived from any one particular judgement obey this rule. In the case where one of the types $\sigma$, $\sigma'$ is derived with (T-Subsum) as the final step, the result is clear.

**(T-env)** Then $e = y$ and by the well-formedness of $\Gamma$ there is no possibility for variation: $\sigma \sim \sigma'$.

**(T-FieldAss)** $e$ is $p.f = e'$. The type of the expression is determined completely by the type of $p$. If $p$ is typed with both $\sigma$ and $\sigma'$, then we assume (induction hypothesis) that $\sigma \sim \sigma'$. Then $\exists \sigma''$ which is a subtype of both. The rule (T-FieldAss) does not care for the uninitialized fields of the type of $p$, So write:

$$\sigma = (\Theta \to C[\mathbb{I}, \mathbb{N}] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)^{\epsilon}$$

$$\sigma' = (\Theta \to C'[\mathbb{I}', \mathbb{N}'] < \overline{\mathbb{I}'}, \overline{\mathbb{N}'} >)^{\epsilon}$$

$$\sigma'' = (\Theta \to C''[\mathbb{I}'', \mathbb{N}''] < \overline{\mathbb{I}''}, \overline{\mathbb{N}''} >)^{\epsilon}$$

Note that whilst the number of mutability and `null`ity parameters in $\sigma, \sigma', \sigma''$ may be different, single inheritance and the rules for well-formed classes guarantee that there is some common supertype, and the field-type lookup will only require some fixed number of these parameters, which is identical in each case.

To determine the type of the expression, we check $\Omega(\varsigma, f)$ or $\Omega(\varsigma', f)$.

**Claim** $\Omega(\varsigma'', f)$ is a subtype to both.

**Proof** The result of $\Omega(\varsigma'', f)$ has only one chance for variation: in the $\mathbb{I}''$s or $\mathbb{N}''$. Since $\sigma''$ is a subtype of $\sigma$ and $\sigma'$, we know that

$$\overline{\mathbb{I}'' \leq \mathbb{I}}$$

$$\overline{\mathbb{I}'' \leq \mathbb{I}'}$$

$$\overline{\mathbb{N}'' \leq \mathbb{N}}$$

$$\overline{\mathbb{N}'' \leq \mathbb{N}'}$$

so we have everything we need to apply (S-Mut) and (S-Null), which gives the result.

**Now,** since we have found a subtype of both $\Omega(\varsigma, f)$ and $\Omega(\varsigma', f)$, we satisfy the relationship $\sim$ (for the whole expression).

**(T-FieldLookupUninit, T-FieldLookup)** The argument here is identical to field assignment: in either case, we determine the type for the field lookup uniquely from the type of the path to the receiver.

**(T-Concat)** By our induction hypothesis, and noting that the derivation for the type of the full expression is one rule-application larger than the derivation for a sub-expression (but has the same result).

**(T-InitRegion)** As with concatenation, by our induction hypothesis, noting that the derivation for this rule is one rule-application larger than that of the sub-expression, from which we take the type directly.

**(T-Allocate)** The result comes directly from the definition of the rule, which has type determined directly by the expression.

**(T-NullDeref)** As before, the type for the whole expression comes directly from the type of a sub-expression with a shorter derivation.

**(T-MethCall)** $e$ is $< \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y})$. The type of the expression is completely determined by the type of $p$, the parameters $\overline{\mathbb{I}}$, $\overline{\mathbb{N}}$, and the method name. The only chance for variation in the type of the call method is in the type of $p$. Assume that the expression as a whole is typed with $\sigma_e$ and $\sigma'_e$. Then we must be able to type $p$ as both $\sigma$ and $\sigma'$, such that the expression as a whole is typed as $\sigma_e$ and $\sigma'_e$. From our induction hypothesis, $\sigma \sim \sigma'$. Let $\sigma''$ be the common subtype of $\sigma$ and $\sigma'$. Then

**Claim** The return type for $\Omega(\sigma'', m, < \overline{\mathbb{I}}, \overline{\mathbb{N}} >)$ subtypes both $\sigma_e$ and $\sigma'_e$ (i.e. it subtypes the return types yielded by looking up the method signature for $\sigma$ or $\sigma'$).

**Proof** First writing

$$\sigma = C[\mathbb{I}_0, \mathbb{N}_0] < \overline{\mathbb{I}}, \overline{\mathbb{N}} >$$

$$\sigma' = C'[\mathbb{I}'_0, \mathbb{N}'_0] < \overline{\mathbb{I}'}, \overline{\mathbb{N}'} >$$

$$\sigma'' = C''[\mathbb{I}''_0, \mathbb{N}''_0] < \overline{\mathbb{I}''}, \overline{\mathbb{N}''} >$$

Note: Inheritance rules allow only for single inheritance. In particular, the method signature for $m$ is defined in some class $D$ which is a super class of all of $C, C', C''$, and takes only a fixed number of parameters. This means that whilst $\sigma, \sigma', \sigma''$ might all have a different number of mutability and `null`ity parameters, only the first $n$ are relevant, with $n$ determined by the signature in $D$.

Now, since $\sigma''$ subtypes $\sigma$ and $\sigma'$, a brief inspection of the subtyping rules reveals that

$$\overline{\mathbb{I}'' \leq \mathbb{I}}$$

$$\overline{\mathbb{I}'' \leq \mathbb{I}'}$$

$$\overline{\mathbb{N}'' \leq \mathbb{N}}$$

$$\overline{\mathbb{N}'' \leq \mathbb{N}'}$$

The return type for the method is identical in every case, apart from the parameters specified by the $\mathbb{I}''$s, $\mathbb{N}''$s (and their counterparts). This is exactly what we need to apply the subtyping rules (S-Mut) and (S-Null).

**Now,** we have found a common subtype between the two alternative return types for the expression as a whole, which is exactly what we need for the $\sim$ relationship.

**So,** any two types for an expression have a common subtype. □

We have not proved a principal type property for our type system. We may have such a property, but we make no claim about that here.

We would like any two paths to the same variable to have types that are consistent. This will be a property of a well-formed stack and heap, and we would like to maintain it.

## 8.2 Well-formedness of $\psi$ and $\phi$

The well-formedness of the $\psi$ and $\phi$ will be properties that speak to the way the heap and the stack are constructed. The result will be that we will be able to judge that objects really do contain the fields they say they do (including their `null` ity parameters).

In simple terms, we want to be able to express:

- An object on the heap has the same fields that are expected from the type of its reference when the program is typed. Specifically:

- Anything that is typed in the environment (i.e. all the locals that the type system expects) is in the stack.
- Anything that is typed as non-null by Γ , Λ is not only in the stack, but also evaluates to a `NotNull` address in the heap.
- Any address on the heap that can be reached by evaluating one of the values on the stack *agrees* with the type given to it by the environment.

**Agreement** between an address and a type requires that:

- The address has runtime class consistent with the class associated with that type.
- Each of the fields that is non-`null`able in the class is not `null`.
- Each of the fields with its type given by the class of the original address.

**Consistent paths** We would also like to have a situation where any two paths through the stack and heap to the same address have consistent types. Notice that we do not store any type information in the heap beyond class; whilst we can check whether an address agrees with a type's `null`ity constraints by checking the values of its fields, there is no way to say that an address in some sense "agrees" with mutability constraints from a type. We will note, however, that in an *empty* stack and heap (that is, the initial stack and heap), there are no paths, so the condition that all paths to the same address are consistent is trivially satisfied.

The formal rules for the well-typedness of the stack and heap with regard to an environment and set of time constraints are laid out in Figure 47 (recall the auxiliary functions *Fields* and *possiblyNull* defined in Section 7.3.1).

**Lemma 8.4.** *Agreement with a subtype implies agreement with a supertype:*

$$\Omega, \psi, \Lambda \vdash v \lhd \sigma \Omega, \Lambda \vdash \sigma \leq \sigma' \} \implies \Omega, \psi, \Lambda \vdash v \lhd \sigma'$$

*Proof.* We know that

$$\mathit{Fields}(\sigma) \subseteq \mathit{Fields}(\sigma')$$

and in particular that the types of the fields in $\sigma$ which are also in $\sigma'$ are exactly the same (from the definition of a well-formed class). So we have all the conditions we require for agreement with $\sigma'$. □

**Lemma 8.5.** *Paths agree with their types*

$$\left.\begin{array}{r} \Omega, \Gamma, \Lambda \vdash p : \sigma \\ \Omega, \Gamma, \Lambda \vdash \psi, \phi \ \texttt{Well formed} \\ \Omega, \psi, \phi \vdash p \rightsquigarrow v, \psi, \phi \end{array}\right\} \implies \Omega, \phi, \Lambda \vdash v \lhd \sigma$$

$$\frac{\nvdash \sigma \leq \texttt{NotNull}}{\Omega, \psi, \Lambda \vdash \texttt{null} \lhd \sigma} \text{ (WF-Null)}$$

$$\frac{\begin{array}{c} \sigma = (\Theta \rightarrow \varsigma)^{\epsilon} \\ \Omega \vdash \psi(v) \downarrow_1 \leq \mathit{Class}(\sigma) \\ \mathit{Fields}(\sigma) = \{\overline{f} : \overline{\sigma}\} \\ \Omega, \psi, \Lambda \vdash \psi(v, \overline{f}) \lhd \overline{\sigma'} \\ \text{where} \begin{cases} \sigma'_i = \mathit{possiblyNull}(\sigma_i) \text{ if } \Lambda \vdash \Theta > \texttt{Now} \wedge f_i \notin \epsilon \\ \sigma'_i = \sigma_i \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases} \end{array}}{\Omega, \psi, \Lambda \vdash v \lhd (\Theta \rightarrow \varsigma)^{\epsilon}} \text{ (WF-Fields)}$$

$$\frac{\begin{array}{c} \forall y \in \Gamma : \Omega, \psi, \Lambda \vdash \phi(y) \lhd \Gamma(y) \\ y \in \phi \implies y \in \Gamma \\ \left.\begin{array}{r} \Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \Omega, \Gamma, \Lambda \vdash p' : \sigma', \Gamma \\ \Omega, \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi \\ \Omega, \psi, \phi \vdash p' \rightsquigarrow \iota, \psi, \phi \end{array}\right\} \implies \Lambda \vdash \sigma \sim \sigma' \end{array}}{\Omega, \Gamma, \Lambda \vdash \psi, \phi \texttt{ Well formed}}$$

Figure 47: A well-formed stack and heap with regard to an environment pair

*Proof.* In the case where $p$ is just a variable lookup, the result is just the definition of well-formedness. Otherwise, we can write $p$ as $p'.f$, and the proof is by arithmetic induction on the length of the path. First we define:

$$length(y) = 1$$

$$length(p.f) = 1 + length(p)$$

Now assume $\forall p : length(p) < n \implies$ the result. Then the type of $p.f$ with length $n$ is given by one of the field lookup rules. We know that the address given by $p$ agrees with $p$'s type (say, $\sigma'$), so if the type of $p.f$ is $\sigma$ then $\sigma$ is given directly by lookup up the field $f$ of $\sigma'$ in the program (subsumption is permitted when typing $p.f$, but Lemma 8.4 covers this case). Agreement with the result of this lookup is exactly the requirement that well-formedness places on the field $f$ for the address given by $p$. We don't need to check the fields of $p.f$ since they are given by $p$'s agreement with $\sigma'$ (notice the definition of agreement is recursive). $\square$

The following two corollaries come directly from the lemma and the definitions.

**Corollary 8.1.**

$$\left.\begin{array}{r} \Omega, \Gamma, \Lambda \vdash p : \sigma \\ \vdash \sigma \leq \texttt{NotNull} \\ \Omega, \psi, \phi \vdash p \rightsquigarrow v \\ \Omega, \Gamma, \Lambda \vdash \psi, \phi \ \texttt{Well formed} \end{array}\right\} \implies v \neq \texttt{null}$$

**Corollary 8.2.**

$$\left.\begin{array}{r} \Omega, \Gamma, \Lambda \vdash p : \sigma \\ \Omega, \psi, \phi \vdash p \rightsquigarrow \iota \\ \iota \neq \texttt{null} \\ \Omega, \Gamma, \Lambda \vdash \psi, \phi \ \texttt{Well formed} \end{array}\right\} \implies \psi(\iota) \downarrow_1 \leq \text{ClassId}(\sigma)$$

## 8.3 Preservation of well-formed $\psi$ and $\phi$

For this section, we will work with a more limited version of the language. The capabilities will be equivalent, but there will be less flexibility with regard to the expressions we allow. We limit method call and field assignment, as in Figure 48. Notice that field assignment is limited to using local variables for the value assigned, and that the result of method call is always stored in a local variable (in the stack). The new type and reduction rules augment those in Figures 40 and 43, but the system is more limited because it is not possible to write all of the previously available expressions.

$$e : p \mid p.f = z \mid \ldots \mid z =< \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y})$$

$$\frac{\begin{array}{c}\Omega, \Gamma, \Lambda \vdash < \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y}) : \sigma, \Gamma' \\ z \notin \Gamma' \\ \Gamma'' = \Gamma'[z \mapsto \sigma]\end{array}}{\Omega, \Gamma, \Lambda \vdash z =< \overline{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\overline{y}) : \sigma, \Gamma''} \text{(T-MethCallLim)}$$

$$\frac{\Omega, \psi, \phi \vdash p.m(\overline{y}) \rightsquigarrow v, \psi', \phi}{\Omega, \psi, \phi \vdash z = p.m(\overline{y}) \rightsquigarrow v, \psi', \phi[z \mapsto v]} \text{(R-MethCallLim)}$$

Figure 48: Field assignment and method call are limited to allow us to reason about them more easily

**Theorem 8.1.** *Types and well-formed heaps and stacks are preserved by execution:*

$$\left.\begin{array}{l}\Omega, \Gamma, \Lambda \vdash \psi, \phi \ \texttt{Well formed} \\ \Omega, \Gamma, \Lambda \vdash e : \sigma, \Gamma' \\ \Omega, \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi'\end{array}\right\} \implies \left\{\begin{array}{r}\Omega, \Gamma', \Lambda \vdash \psi', \phi' \ \texttt{Well formed} \\ \Omega, \Lambda \vdash v \lhd \sigma\end{array}\right.$$

Before we can prove the theorem, we will need some intermediate lemmas:

**Lemma 8.6.** *If we restrict our view on a typing environment and make the corresponding restriction on the stack, then we maintain the well-formedness property:*

$$\left.\begin{array}{l}\Omega, \Gamma, \Lambda \vdash \psi, \phi \ \texttt{Well formed} \\ \overline{y} \in \Gamma \\ \overline{y} \in \phi \\ \Gamma' = \Gamma\big|_{\overline{y}} \\ \phi' = \phi\big|_{\overline{y}}\end{array}\right\} \implies \Omega, \Gamma', \Lambda \vdash \psi, \phi', \ \texttt{Well formed}$$

*Proof.* We require that:

- All items in the new stack are in the new environment,
- All items in the new environment are in the stack, and agree with the type given by the environment.

Since the two are restricted to the same domain, and (by their well-formedness) they were both originally over the same domain, we clearly have the first requirement, and most of the second requirement: we only need to demostrate that agreement is preserved.

Pick $y \in \Gamma'$. Then $y \in \Gamma$, and so (by well-formedness) $\phi(y) \lhd \Gamma(y)$. But also $y \in \phi'$ and $\phi'(y) = \phi(y)$, and $\Gamma'(y) = \Gamma(y)$ so $\phi'(y) \lhd \Gamma'(y)$, which is all we need. □

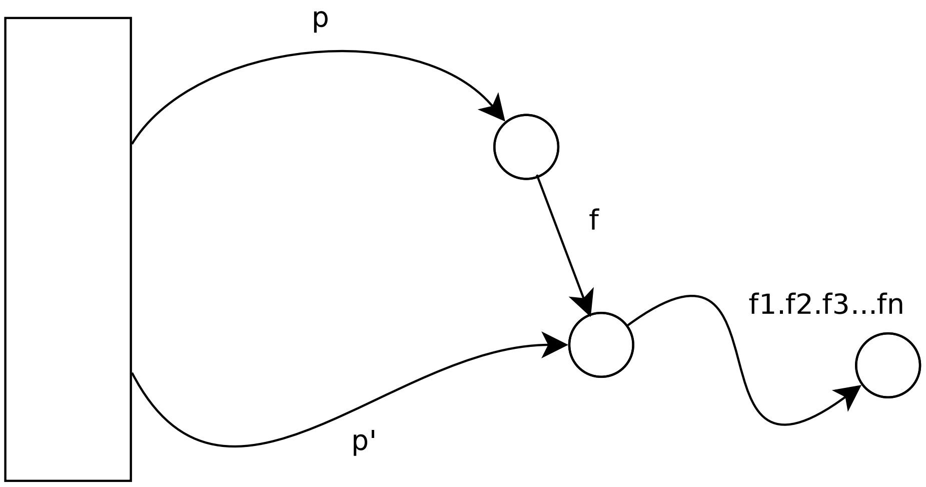

Figure 49: Two paths through the same object

Lemma 8.6 will be required for the proof (not given here) that method-call preserves the well-formedness of the stack.

**Lemma 8.7.**

$$\left.\begin{aligned}&\Omega,\Gamma,\Lambda\vdash\psi,\phi\ \texttt{Well formed}\\&\phi(y)=\iota\in\psi\\&\Gamma(y)=(\Theta\rightarrow C[\mathbb{I}_0,\mathbb{N}_0]<\overline{\mathbb{I}},\overline{\mathbb{N}}>)^{\epsilon}\\&\Gamma'=\Gamma[y\mapsto(\Theta\rightarrow C[\mathbb{I}_0,\texttt{NotNull}]<\overline{\mathbb{I}},\overline{\mathbb{N}}>)^{\epsilon}]\end{aligned}\right\}\implies\Omega,\Gamma',\Lambda\vdash\psi,\phi\ \texttt{Well formed}$$

*Proof.* Since the types of the fields of $y$ are not parametric in $\mathbb{N}_0$, and since the rule (WF-Null) did not apply to $y$ under the environment $\Gamma$, the conditions for agreement of $y$ are exactly the same. Agreement of any other variable is not affected. □

**Lemma 8.8.** *Subtyping is preserved through field lookup:*

$$\left.\begin{aligned}\Omega,\Lambda\vdash\sigma\leq\sigma'\\\sigma=(\Theta\rightarrow\varsigma)^{\epsilon}\\\sigma'=(\Theta\rightarrow\varsigma')^{\epsilon'}\end{aligned}\right\}\implies\Omega(\varsigma,f)\leq\Omega(\varsigma',f)$$

*Proof.* Since $\sigma\leq\sigma'$, then $\exists C\in\Omega,s,s'\in\mathbb{S}$ such that $\Omega(\varsigma,f)=s(\Omega(C,f))$, $\Omega(\varsigma',f)=s'(\Omega(C,f))$, and also we know that if $m$ and $n$ are the numbers of mutability and `null`ity constraints in $\sigma'$, then $\vdash\overline{\mathbb{I}\leq\mathbb{I}'}\big|_{0\ldots m}$ and $\vdash\overline{\mathbb{N}\leq\mathbb{N}'}\big|_{0\ldots n}$, where $I_i$ is the $i$th mutability constraint in $\sigma$, $I'_i$ is the $i$th mutability constraint in $\sigma'$, and so on. Now note that $s(\Omega(C,f))$ and $s'(\Omega(C,f))$ are determined by no more than $m$ mutability constraints and $n$ `null`ity constraints from the types $\sigma$ and $\sigma'$ (the number of constraints used is determined by the class $C$, which has no more formal constraints than $\sigma'$ has parameters). In particular, $\Omega(\varsigma, f) \leq \Omega(\varsigma', f)$. $\square$

**Notation** We will write $p.\overline{f}$ for $p.f_1.f_2.\ldots.f_n$.

**Lemma 8.9.** *When the types of two paths are consistent, that consistency is preserved by successive field lookup (e.g. Figure 49):*

$$\left.\begin{array}{c} \Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \Omega, \Gamma, \Lambda \vdash p' : \sigma', \Gamma \\ \Omega, \Lambda \vdash \sigma \sim \sigma' \\ \Omega, \Gamma, \Lambda \vdash p.\overline{f}\big|_{1\ldots n} : \sigma'' \\ \Omega, \Gamma, \Lambda \vdash p'.\overline{f}\big|_{1\ldots n} : \sigma''' \end{array}\right\} \implies \Omega, \Lambda \vdash \sigma'' \sim \sigma'''$$

*Proof.* By arithmetic induction on $n$.

**Induction Hypothesis:**

$$\left.\begin{array}{c} \Omega, \Gamma, \Lambda \vdash p.\overline{f}\big|_{1\ldots n-1} : \sigma''_{-1}, \Gamma \\ \Omega, \Gamma, \Lambda \vdash p'.\overline{f}\big|_{1\ldots n-1} : \sigma'''_{-1}, \Gamma \end{array}\right\} \implies \Omega, \Lambda \vdash \sigma''_{-1} \sim \sigma'''_{-1}$$

We want to show that $\sigma'' \sim \sigma'''$. We know from the consistency of the types of $p$ and $p'$ that the delay times are the same for all the types we consider here. Now write:

$$\sigma''_{-1} = (\Theta \to \varsigma)^{\epsilon}$$
$$\sigma'''_{-1} = (\Theta \to \varsigma')^{\epsilon'}$$

**Inductive step:** Define:

$$\sigma''''_{-1} = (\Theta \to \varsigma'')^{\epsilon''}$$

such that:

$$\Omega, \Lambda \vdash \sigma''''_{-1} \leq \sigma''_{-1} \wedge \sigma''''_{-1} \leq \sigma''_{-1}$$

(such a type exists, since $\sigma''_{-1} \sim \sigma'''_{-1}$).

Then $\sigma''$ is given by $\Omega(\varsigma, f)$, and $\sigma'''$ is given by $\Omega(\varsigma', f)$ – up to subsumption, the time parameters (which are common between the types), and the initialization states of the types $\sigma^{\cdots}_{-1}$. The differing initialization states are irrelevant, since we only track initialization states in the environment, $\Gamma$ , for paths of length 1; otherwise we assume all `NotNull` fields are as-yet possibly-uninitialized (if the commitment point is in the future; otherwise we ignore them entirely). Then writing:

$$\sigma'''' = (\Theta \to \Omega(\varsigma'', f))^{notNullFields(\Omega(\varsigma''))}$$

Such a type is well-formed; $\sigma''''_{-1}$ must have the field $f$ since it subtypes two other types which do, and the form of the type is determined (up to substitution of mutability and `null`ity parameters) entirely by the *class* (and is common to all three).

Note also that (along with Lemma 8.8):

$$\Omega, \Lambda \vdash \sigma'''' \leq \sigma'' \wedge \sigma'''' \leq \sigma'''$$

Which is exactly what we need for $\Omega, \Lambda \vdash \sigma'' \sim \sigma'''$.

So, by induction on $n$, we have that the consistency of the whole paths comes directly from the consistency of the sub-paths. □

**Lemma 8.10.** *Fields of a path with grounded type also have grounded type:*

$$\left.\begin{array}{r}\Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ FieldType_{\Omega}(\sigma) \\ \Omega, \Gamma, \Lambda \vdash p.\overline{f} : \sigma'\end{array}\right\} \implies FieldType_{\Omega}(\sigma')$$

*Proof.* By induction on $length(p.\overline{f}) - length(p)$ (which is just $length(\overline{f})$). First assume the difference is 0. Then there is nothing to show. Now assume the difference is $n+1$. Then:

$$\Omega, \Gamma, \Lambda \vdash p.\overline{f}\big|_{1 \ldots n} : \sigma''$$

Now, we have a path $q$ such that:

$$\Omega, \Gamma, \Lambda \vdash q : \sigma''$$

$$FieldType_{\Omega}(\sigma'')$$

So

$$\Omega, \Gamma, \Lambda \vdash q.f : \sigma'$$

and this is just the single-step case, so it's clear that

$$FieldType_{\Omega}(\sigma'') \implies FieldType_{\Omega}(\sigma')$$

□

*Proof.* Proof of Theorem 8.1. Proof will be by induction on the structure of the expression.

$y$ No change is made to the heap, stack, environment, or time environment. Well-formedness is preserved.

$p.f$ Again, no change is made to the heap, stack, environment, or time environment. Well-formedness is preserved.

$p.f = z$ We assume that:

$$\Omega, \Gamma, \Lambda \vdash p.f = z : \sigma'', \Gamma'$$

via the rule (T-FieldAss), and that:

$$\Omega, \Gamma, \Lambda \vdash z : \sigma', \Gamma$$

$$\Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma$$
$$\sigma = (\Theta \rightarrow \varsigma)^{\epsilon}$$
$$\sigma'' = (\Theta \rightarrow \Omega(\varsigma, f))^{notNullFields(\Omega(\varsigma,f))}$$
$$FieldType(\sigma'')$$
$$\Lambda \vdash \sigma' \leq \sigma''$$
$$\Gamma' = \Gamma[p \mapsto \Gamma(p)_f]$$

We know that the expression is reduced by the rule (R-FieldAss):

$$\Omega, \psi, \phi \vdash z \rightsquigarrow v, \psi, \phi$$

$$\Omega, \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi$$
$$\Omega, \psi, \phi \vdash p.f = z \rightsquigarrow v, \psi', \phi$$
$$\psi' = \psi[\iota \mapsto \psi(\iota)[f \mapsto v]]$$

The result of evaluating the expression, $v$, agrees with the type of the expression, since it agrees with the type of $z$ and the type of $z$ is a subtype of the expression as a whole. Now we need to show that the agreement of $\iota$ is not violated. Note that $\Omega, \psi, \Lambda \vdash \iota \lhd \sigma$ The only change is in the field $f$, so we only need to check that $v \lhd \Omega(\varsigma, f)$: since $\sigma$ is *Grounded* and field definitions do not change between subtypes, this is sufficient (any other path to $\iota$ must be typed in a way that is consistent with $\sigma$, so if the agreement of the stack relies on $\iota \lhd \sigma_x \neq \sigma$ then $\Omega(\mathit{ClassId}(\sigma_x), f)$ gives rise to no more specific requirements on $\psi(\iota, f)$ than does $\sigma$). Since $\Omega, \Lambda \vdash v \lhd \sigma' = \Gamma(z)$ (by the well-formedness of $\psi, \phi$), and $\sigma' \leq \sigma''$, so $v \lhd \sigma''$, which is enough.

So what remains to be shown is that any new paths also satisfy our consistency conditions. We need only consider *new* paths – in particular, those that pass through the address given by $v$. If $v = \texttt{null}$, then there is nothing to show. Now assume $v = \iota'$. So the only possibility for a change is paths through $\iota$, through field $f$. Now, let:

$$\Omega, \psi', \phi \vdash q \rightsquigarrow \iota, \psi', \phi$$

(we know $\psi'(\iota, f) = \iota'$; this is the result of the field assignment). Then let

$$\Omega, \psi', \phi \vdash q.f.\overline{f} \rightsquigarrow \iota'', \psi', \phi$$

$$\Omega, \psi', \phi \vdash q' \rightsquigarrow \iota'', \psi', \phi$$
$$\Omega, \Gamma' \vdash q : \sigma_q$$

$$\Omega, \Gamma' \vdash q.f : \sigma_{q,f}$$

$$\Omega, \Gamma' \vdash q.f.\overline{f} : \sigma_a$$

$$\Omega, \Gamma' \vdash q' : \sigma_b$$

Then we need to show that:

$$\Omega, \Lambda \vdash \sigma_a \sim \sigma_b$$

If we can show this, then by the generality of $q$ and $q'$, we will have shown that any new paths created by our modification to the heap (new paths are $q.f.\overline{f}$) are consistent with any other paths in the heap evaluating to the same address. Since these are the only paths we need to consider, this is sufficient. We will do this by showing that both paths ($q.f.\overline{f}$, $q'$) are consistent with $z.\overline{f}$, and since $\mathit{FieldType}_\Omega(\sigma'')$, will will know that the two paths are consistent with each other (by Lemma 8.2).

**Claim** $\Omega, \Lambda \vdash \sigma_{q,f} \sim \sigma'$

**Proof** Assume there are no cycles in $q$ or $p$. If there are, then we can just use a shorter version of the paths with the cycles omitted. Specifically, $p$ and $q$ do not rely on $\psi(\iota, f)$ (or indeed $\psi'(\iota, f)$). Then:

$$\Omega, \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi$$

$$\Omega, \psi, \phi \vdash q \rightsquigarrow \iota, \psi, \phi$$

Also:

$$\Omega, \Gamma, \Lambda \vdash q : \sigma'_q, \Gamma$$

$$\Omega, \Lambda \vdash \sigma_q \leq \sigma'_q$$

(the only change to $\Gamma$ makes the types of paths through $\iota$ more specific).

By the well-formedness of $\psi, \phi$ with respect to $\Gamma, \Lambda$, we know that $\Omega, \Lambda \vdash \sigma \sim \sigma'_q$. By Lemma 8.9, the type of $q.f$ is consistent with the type of $p.f$. By inspection of the formulation of $\sigma''$, and the generation lemma together with the rule (T-FieldLookup), we know that the type of $p.f$ is exactly $\sigma''$. So:

$$\Omega, \Lambda \vdash \sigma_{q,f} \sim \sigma''$$

$$\Omega, \Lambda \vdash \sigma' \leq \sigma''$$

$$\therefore \Omega, \Lambda \vdash \sigma' \sim \sigma''$$

$$\mathit{FieldType}_\Omega(\sigma'')$$

$$\therefore \Omega, \Lambda \vdash \sigma_{q,f} \sim \sigma'$$

That is, the type of $q.f$ is consistent with the type of $z$. Also note that since $\sigma'$ is a subtype of a field-assignable-type, so $\mathit{FieldType}_\Omega(\sigma')$. So we have $\Omega, \Lambda \vdash \sigma_{q,f} \sim \sigma'$

**Claim** $\Omega, \Lambda \vdash \sigma_a \sim \sigma_b$

**Proof** First assume that the evaluation of $q'$ does not pass through $\iota$. If it does, then $\sigma_b \sim \sigma'''$ by the argument above. Assuming it does not, it's clear that $\Omega, \psi', \phi \vdash q' \rightsquigarrow \iota'' \implies \Omega, \psi, \phi \vdash q' \rightsquigarrow \iota''$. By the consistency of paths in the well-formed $\psi$ and $\phi$ with respect to $\Gamma$, $z.\overline{f}$ has type compatible with $q'$:

$$\Omega, \Gamma, \Lambda \vdash z.\overline{f} : \sigma''', \Gamma$$

$$\Omega, \Lambda \vdash \sigma''' \sim \sigma_b$$

By Lemma 8.10, $\mathit{FieldType}_\Omega(\sigma''')$, and by Lemma 8.9, $\sigma_a \sim \sigma'''$, so Lemma 8.2 gives us:

$$\Omega, \Lambda \vdash \sigma_a \sim \sigma_b$$

$e; e'$ Preservation of well-formedness comes directly from the preservation of well-formedness in the sub-expressions. We will assume this from our induction hypothesis. The value that is the result of reducing the expression is just the result of reducing $e'$, and the type of the expression also types $e'$, so we also assume that the resultant value agrees with the type for the expression as a whole.

`delay` $t\{e\}$ The heap, stack, and environment are unaltered. We introduce a new time variable, $t$. The expression is typed by the rule (T-InitRegion), which requires that $t$ is not present in the existing time-environment. Since commitment points become `Now` when they leave their initialization region (that is, any time they are not in the time-environment), any non-`Now`, non-delayed times must be time parameters ($\theta$s), and so by the form of expressions they are not $t$. So, we have not affected the type information for any path, and the well-formedness of the heap and stack are maintained, and we can use our induction hypothesis that $e$ results in a well-formed heap and stack. Note also that the rule (T-InitRegion) – which must be used in the typing of the expression, by the generation lemma – requires any new addresses to be fully initialized; in particular, their fields must be assigned values agreeing with their types. Since any values assigned to the fields of an object with commitment point $t$ must also have commitment point $t$, they must also be newly allocated addresses in this scope, and in particular they are also required to be fully initialized. So we not only have agreement at the end of $e$ for any newly allocated addresses, but also their non-`null` fields are guaranteed to be initialized, and so we can set their commitment points to `Now`, and the well-formedness of the stack and heap are maintained for the whole expression. Any new paths are created within the expression $e$, so if they reduce to the same address, then their types are consistent (and their commitment points are the same, so any substitution for `Now` that we perform when leaving the initialization scope applies to them both, and does not affect their consistency).

`alloc` $\sigma$ `as` $z$ The type rule (T-Allocate) adds one new variable to the environment, and the runtime rule (R-Allocate) adds one new variable to the stack (the same new variable). We need to show that the new address (added to the heap) agrees with the type of the new variable (that is: $\phi'(z) \lhd \Gamma'(z)$). The runtime rule gives us that the runtime class of the new address agrees with $\sigma$ , and that it has the correct fields, but they are all initialized to `null`. For agreement, all the fields must agree with their types. Note that the commitment point of the type is required to be in the future by (T-Allocate), and that all the `NotNull` fields are noted as uninitialized in the resultant environment (i.e. $\mathit{uninit}(\Gamma'(z)) = \mathit{notNullFields}(\sigma)$): so agreement just requires that all the fields have well-defined values (that they are all `null` is fine). The only new paths are the path $z$, and Lemma 8.5 gives us that any such path is guaranteed to be consistent with any other such path. The value resulting from this reduction is just the newly allocated address, and we have already argued that it agrees with the expression's type.

`ifnull` $y$ `then` $e$ `else` $e'$ By assumption, noting that Lemma 8.7 guarantees the well-formedness of the stack and heap when evaluating the sub-expression $e'$ (and that in $e$ they are unchanged). A violation of well-formed heap and stack for the expression as a whole requires such a violation in either $e$ or $e'$, and we will assume such a violation does not occur.

$< \bar{\mathbb{I}}, \overline{\mathbb{N}} > p.m(\bar{(y)})$ We will not give a formal proof for this case. An argument that should serve as the basis of the proof is as follows: The reduction rule (R-MethCall) leaves the stack unchanged, but allows transformations of the heap within a new (more restricted) stack. We need to show that transformations within the new stack cannot affect the well-formedness of the old one. By Lemma 8.6, the method body is evaluated in a well-formed heap and stack (allowing for re-labelling), and by assumption (and the well-typedness of the method body), the resultant heap and stack are well-formed (with respect to the more limited environment and time-environment). The challenge is to show that the *original* stack is still well-formed with the new heap.

- First consider the parts of the heap that were directly accessible to the new stack (that is, the arguments to the function). The arguments themselves do not change (we do not change any of the values on the stack) – only their fields do. Note that any path through the arguments to the method (the parts of the call-site stack accessible in the method body) is also a path in the call-site stack. Any changes to the fields of addresses accessible through those paths are equivalent to changes in the call-site stack and heap; in particular, any operation that is permitted in the method body is also a well-typed expression under the environment and stack at the call-site (up to re-labelling). So, the possibilities for changes to the heap within the method body are *more limited* than the changes to the heap permitted at the call-site, and (along with our induction hypothesis) the well-formedness of the stack and heap are preserved.

- Now consider any addresses which were not accessible through the call-site stack. When we leave the method body, the only possibility for these addresses to be accessible is through the fields of the arguments (or any other objects in the call-site stack), which is what we dealt with in the previous case.

The well-formedness of the stack and heap are preserved.

So any well-typed expression (apart from the case of method call, which we have given only the outline of an argument for) preserves the well-formedness of the heap and stack with respect to the environment and program. The case of method call should be tackled in future work. □

## 8.4 Mutability

We wish to express that, given that a variable is declared to be `Immutable`, it cannot change during the course of a program's execution (that is, the evaluation of a well-typed expression).

**Definition 8.3.** *An address $\iota$ is* `Immutable` *if there is a path $p$ such that:*

$$\begin{aligned} \Omega, \Gamma, \Lambda &\vdash p : \sigma, \Gamma \\ \Omega, \Lambda &\vdash \sigma \leq \mathtt{Immutable} \\ \Omega, \psi, \Gamma &\vdash p \rightsquigarrow \iota, \psi, \phi \end{aligned}$$

First, we wish to guarantee that if an address is seen as `Immutable` via any path through the heap, then *no* path through the heap allows it to be treated as `Writable`. This condition distinguishes the notion of true immutability from more limited notions, such as `C++`'s `const`; `const` is a guarantee to the caller about how the *local* scope will treat an object. `Immutable` is a guarantee to the local scope about how an object will be treated globally.

**Theorem 8.2.**

$$\left.\begin{aligned} \Omega, \Gamma, \Lambda \vdash \psi, \phi \ \mathtt{Well\ formed} \\ \Omega, \Gamma, \Lambda \vdash p : \sigma \\ \Omega, \psi, \phi \vdash p \rightsquigarrow \iota \\ \Omega, \Gamma, \Lambda \vdash p' : \sigma' \\ \Omega, \psi, \phi \vdash p' \rightsquigarrow \iota \\ \Lambda \vdash \sigma \leq \mathtt{Immutable} \end{aligned}\right\} \implies \Omega, \Lambda \not\vdash \sigma' \leq <? \ \mathtt{extends\ Writable} >$$

*Proof.* The proof is direct from the definition of a well-formed heap and stack, which gives us that:

$$\Omega, \Lambda \vdash \sigma \sim \sigma'$$

```
class D[i]<,> {
}

class C[i]<,> {
    f : D[Immutable, NotNull]<,>

    Now -> C[i, Nullable]<,> {x0 -> f} {T} m0(
            (T -> C[<? extends ReadOnly>, NotNull])^{f} x0) {
        alloc T -> D[Immutable, NotNull]<,> as z;
        x0.f = z;
        null;
    }

    Now -> C[i, Nullable]<,> {} {} m1(
            Now -> C[<? extends Writable>, NotNull]<,> x0) {
        delay t {
            alloc (t -> D[Immutable, NotNull]<,>) as z;
        }
        x0.f = z;
        null;
    }
}

...
delay t {
    alloc C[Mutable, NotNull]<,> as z0;
    z0.m0()
    /* z0.f is an immutable object, l  */
}

z0.m1()
/* z0.f is a new immutable object; there
    is no remaining path to l */
```

Figure 50: Immutable paths are not necessarily maintained

By inspection of the subtyping rules, there is no $\sigma''$ such that

$$\Omega, \vdash\vdash \sigma'' \leq <? \texttt{ extends Writable} > \wedge \sigma'' \leq \texttt{Immutable}$$

So $\Omega, \Lambda \vdash \sigma' \leq <? \texttt{ extends Writable} >$ is a contradiction. □

Inspection of the typing rules (specifically, (T-FieldAss)) will reveal what guarantees this gives us: namely, that field assignment on an object that is typed `Immutable` is impossible through any path.

We would like to know that, given that an object is seen as `Immutable` at some stage in our program, it will remain that way. We cannot guarantee that if we have an immutable path to an address, then we will have one for the remainder of the program's execution; it is easy to construct a counter example; see Figure 50.

Note that in Figure 50, there is no way to construct a new (writable) path to the address `l`. We formalize this property (without proof):

**Theorem 8.3.**

$$\left.\begin{array}{r}\iota \in \psi \Omega, \Gamma, \Lambda \vdash e : \sigma', \Gamma', \Lambda' \\ \Omega, \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi' \\ \Omega, \Gamma', \Lambda \vdash p' : \sigma', \Gamma' \\ \Omega, \psi', \phi' \vdash p' \rightsquigarrow \iota, \psi, \phi \\ \Omega, \Lambda \vdash \sigma' \leq <?\ \textit{extends Writable} > \end{array}\right\} \Longrightarrow$$

$$\left\{\begin{array}{l}\Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \Omega, \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi \\ \quad \Omega, \Lambda \vdash \sigma \leq <?\ \textit{extends Writable} > \\ \qquad \textit{for some path } p \end{array}\right.$$

The argument for the correctness of this theorem is that the only way to create new paths to an address already on the heap is through field assignment or method call (more specifically, field assignment within the new scope created by method call). In either case, the type of the assignee must be *Grounded*. Lemma 8.10 grants us that we will have a *Grounded* view on the address to which we are about to create a `Writable` path; for a new `Writable` path to be created, we must already have a path which is either `Immutable` but not yet committed, or `Mutable` – in either case, we already have a `Writable` path.

**The promise of Immutability** The two theorems above are important properties of the system, but we should formalize the guarantees this gives us at a higher level. The following Theorem comes directly from Theorems 8.3 and 8.2, along with the fact that the only chance for field assignment is an expression of the form $p.f = z$. In plain English, the theorem is a statement that, given that we have an immutable view on an address, its fields will not change throughout the life of the program.

**Theorem 8.4.**

$$\left.\begin{array}{r}\Omega, \Gamma, \Lambda \vdash p : \sigma, \Gamma \\ \Omega, \Lambda \vdash \sigma \leq \textit{Immutable} \\ \Omega, \psi, \phi \vdash p \rightsquigarrow \iota, \psi, \phi \\ \Omega, \Gamma, \Lambda \vdash e : \sigma', \Gamma' \\ \Omega, \psi, \phi \vdash e \rightsquigarrow v, \psi', \phi' \end{array}\right\} \Longrightarrow \psi'(\iota) = \psi(\iota)$$

*Proof.* From theorem 8.2, we know that there are no `Writable` paths to $\iota$ in $\psi, \phi$ (under the original environment). From theorem 8.3 we know that the only way for there to be a `Writable` path to an address already in the heap is if one already exists. In particular, no new `Writable` paths can be created to $\iota$. The only chance to change $\psi(\iota)$ is through field assignment. The type rule for field assignment requires a `Writable` path to the address of which the field is being assigned. Since there are no `Writable` paths to $\iota$, there is no possibility to assign to its fields. □

This final theorem is exactly what we require from a type system for mutability.

# 9 Contributions

There are four main contributions, which we will detail in the following section:

**Generic Nullity** The system approaches `null`ity constraints in a novel way: a whole *type* is parametric in `null`ity, not just a single reference or field. We give a little more discussion below in the hope that it will inform any future work on parametric `null`ity constraints.

**Flexible initialization** We achieve a great degree of flexibility in initialization which is as free as systems of `null`ity, whilst allowing us the constraints offered by systems of mutability.

**A unified treatment `null`ity *and* immutability** These two concepts share a common challenge, and we present a unified system which caters for both.

**Lightweight Soundness** We do not need to add any extra constructs to our idea of runtime semantics in order to demonstrate that our system's properties hold.

## 9.1 Generic Nullity

We achieve a greater degree of flexibility than existing systems, in the sense that we allow a distinction between instances of a class which can have `null`-able fields, and those which cannot (whilst allowing them to be treated generically where safe). This type of flexibility allows us to construct generically-`Nullable` collections, as in Figure 52 (which has a usage example in Figure 53). Such genericity is not possible in other systems for `null`ity.

Under the system presented here, it is possible to create a linked data structure which is *guaranteed* to be cyclic after initialization. This is difficult to express through the model of genericity presented by [18], which simply allows `null`ity conditions to be a part of a generic type, but not as parameters in their own right: compare

```
List<Item!>!
```

to

```
List[NotNull]<NotNull>
```

In the former case (which is proposed by [18]), we specify a `List` which contains `Item!`s. In the latter (which is our model), we specify a `List` which contains `NotNull` items (though in our system there is no possibility to specify the class of the items contained).

We chose not tackle class-based genericity, but instead separate the class from the other type information in a way not possible in other systems for `null`ity. Figure 52 provides an example of a single linked list class which can be instantiated to contain either non-`null` or possibly-`null` references (this is

```
class C[i0]<,n1> {
    next : C[i0, n1]<,n1>
}
...
delay t {
    alloc t -> C[Immutable, NotNull]<,NotNull> as cyclic;
    /* If we fail to initialize cyclic.next, it
        is a type error */
    cyclic.next = cyclic;

    alloc t -> C[Immutable, NotNull]<,Nullable> as nonCyclic;
    /* There is no requirement to initialize
        nonCyclic.next */
}

/* cyclic is a 1-cycle. */
/* cyclic.next.next.next.next == cyclic */
/* It is impossible to construct an instance of C[Immutable,
    NotNull]<,NotNull> that does not have this property (i.e. it
    must be cyclic).

/* nonCyclic.next is just null */
/* We see that the cyclic nature is determined by the specific
    instantiation of the class. */
```

Figure 51: Example of leveraging generic `null`ity constraints to enforce a cyclic structure

easy to specify in any `null`ity type system with support for generics). Figure 51 gives an example of leveraging the `null`ity parameter to guarantee the cyclic property of a structure: this is hard to specify in previous systems for `null`ity, which do not treat it as a generic parameter in its own right.

**Considerations with generic `nullity`** It is clearly sensible and desirable to be able to assign a `NotNull` reference in place of a `Nullable` one, so in general we say that a `NotNull` reference subtypes a `Nullable`one. But an object with `NotNull` fields does *not* subtype one with `Nullable` fields: we saw in Section 7.8.4 this was unsafe. So whilst `C[I, NotNull]` will do in place of `C[I, Nullable]`, `C[I, Nullable]<,NotNull>` will not do in place of `C[I,Nullable]<,Nullable>`. There is no obvious analogue in the case of mutability constraints, but this is because the case of `C[I, NotNull]` subtyping `C[I, Nullable]` is a special one – the first `null`ity parameter in a type describes the particular *reference* (or path) that we are typing, rather than the object that lies at the other end of it. We have treated the first `null`ity parameter as a generic parameter just like all the others, but it really is different, in the sense that the others describe the object, whereas the first describes the reference.

**The upper bound of `null` and `NotNull`** Whilst we cannot use a reference like $C[\ldots] <$ `NotNull` $>$ in place of $C[\ldots] <$ `Nullable` $>$, we would still like to be able to handle these two different possibilities in a generic way. In other systems which do not treat `null`ity as a type parameter, the upper bound of `Nullable` and `NotNull` is just `Nullable`. Since this is not possible here, we instead write <? `extends Nullable` > where we wish to say: "We do not care the `null`ity of this reference; we will treat it with all the restrictions of either."

## 9.2 Flexible initialization patterns

We retain a great degree of flexibility in initialization, whilst providing the safety and guarantees traditionally offered by systems of both `null`ity and mutability. We allow expressively typed, generic factories to be created (such as the one in Figure 55, with a usage example in Figure 56), that can initialize cyclic, immutable, non-`null` structures in a natural way. The not-`null` fields of objects can be initialized by helper methods (as in [4], and unlike [18]). One of the challenges of a system which combines mutability constraints and `null`ity constraints is ensuring that non-`null` fields are initialized before the object becomes immutable, and our system overcomes that difficulty neatly and naturally. [21] allows for factories and the initialization of cyclic structures through ownership; we achieve this aim without the extra concept, but by instead associating the initialization of an object with a section of the method body in which it is created, as in [4] and [6]. We include more detail in the type about initialization state than other systems for `null`ity (although not as much as all systems for initialization: see Masked Types in [15]), and a possible application of this is in the design of fluent interfaces for initialization, such as in Figure 54. In standard `Java`, we *can* achieve this affect by casting through otherwise-useless "micro-interfaces."

## 9.3 Exploration of the crossover between `null`ity and Mutability

We gain a better understanding of combining mutability with `null`ity, and in particular with regard to the way we initialize objects. The overlap between these systems is in two main areas:

**The initialization region itself** We unify initialization: in particular, an object is either initialized with respect to *both* mutability *and* `null`ity, or neither. This is exactly the intuition.

**The way we treat objects with different initialization states** We can be less relaxed about allowing objects with different initialization states hold references to each other than in systems for `null`ity alone. In this regard, we have all the constraints of systems for mutability. In particular, we cannot initialize data objects with fields having differing commitment points to the objects themselves, which previous systems of `null`ity have been able to do.

```
class Item[i0]<,> {
    Item[i0, Nullable]<,> {} {} mutate(
        Item[Mutable, NotNull]<,> this) {
        ...
        null
    }
}

class List[i1]<i2,n1> {
    next : List[Mutable, Nullable]<i2, n1>
    item : Item[i2, n1]<,>

    List[Mutable, Nullable]<i2, n1> {} {} getNext(
        List[Mutable, NotNull]<i2, n1> this) {

        this.next;
    }

    List[i1]<i2, n1> {} {} add(
        List[Mutable, NotNull]<i2, n1> this,
        Item[i2, n1]<,> i) {

        z0 = this.getNext();
        ifnull z0 then
            this.item = i;
            delay t {
                alloc t -> List[Mutable, NotNull] as z1
            };
            this.next = z1
        else
            this.next.add(i);

        this
    }

    Item[i2, n1]<,> {} {} get(
        List[<? extends ReadOnly>, NotNull]<i2, n1> this,
        int index) {

        if (index == 0)
            this.item
        else
            z = this.getNext()
            ifnull z then
                throw new IndexOutOfBoundsException();
            else
                z.get(index - 1)
    }
}
...
```

Figure 52: A linked list with generic `null`ity. Assumes the presence of integers with the standard operations, standard booleans, an if-statement, and exceptions, and omits time annotations. We present a usage example in Figure 53

```

delay t {
    // A list of Immutable, NotNull entries
    alloc t -> List[Mutable, NotNull]<Immutable, NotNull> as list1;
    // A list of mutable, possibly-null entries
    alloc t -> List[Mutable, NotNull]<Mutable, Nullable> as list2;
}

list1.get(4); // null; list index out of range

delay t {
    alloc t -> Item[Immutable, NotNull]<,> as item1;
    alloc t -> Item[Mutable, Nullable]<,> as item2;
}

/* Okay */
list1.add(item1);

z1 = list1.get(0); // z1 : Item[Immutable, NotNull]
/* No need to test if z1 is null */
/* z1.mutate() would be a type error
    because z1 is Immutable */

/* Type Errors: */
list1.add(item2); // Cannot accept Mutables
list1.add(null);  // Cannot accept null
list2.add(item1); // Cannot accept Immutables

/* Okay */
list2.add(item2);
list2.add(null);  // Can accept null

z2 = list2.get(0); // z2 : Item[Mutable, Nullable]
z3 = list2.get(1); // z3 : Item[Mutable, Nullable]
ifnull z2 then
    /* List can contain null */
else
    /* z2 : Item[Mutable, NotNull] */
    /* legal */
    z2.mutate();

ifnull z3 then
    /* List can contain null */
else
    ...
```

Figure 53: Code demonstrating the use of differently parametrized generic list example from Figure 52

```
class Rectangle[i] extends object {
    height : Integer[Immutable, NotNull]<,>
    width : Integer[Immutable, NotNull]<,>

    T -> Rectangle[i, NotNull]
    { this -> height }
    {T}
    withHeight(
        (T -> Rectangle[i, NotNull]<,>)^{height, width} this,
        (T -> Integer[Immutable, NotNull]<,>)^{} height) {

        this.height = height;
        this
    }

    T -> Rectangle[i, NotNull]
    { this -> width }
    {T}
    andWidth(
        /* Require that the height has been initialized */
        (T -> Rectangle[i, NotNull]<,>)^{width} this,
        (T -> Integer[Immutable, Nullable]<,>)^{} width) {

        ifnull width then
            /* We know height has been initialized */
            this.width = height;
        else:
            this.width = width;

        this
    }
}
...
delay t {
    alloc t -> Rectangle[Immutable, NotNull] as z;
    /* Assume we can initialize height to some integer */
    alloc t -> Integer[Immutable, NotNull] as height;
    /* z.andWidth() is a type error because it requires height
        to be initialized */
    z.withHeight(height)
    /* failure to initialize width is a type error */
    /* We can now call z.andWidth() because height is
        initialized */
    z.andWidth()
}
```

Figure 54: A demonstration of statically-enforced fluent interfaces for initialization. Assumes the presence of integers.

```
class Wheel[i]<,> {
    next : Wheel[i, NotNull]<,>
    prev : Wheel[i, NotNull]<,>
}

class WheelFactory[i]<,> {
    Now -> Wheel[I, NotNull]<,> {} {}
        makeWheel((Now -> WheelFactory[i, NotNull])^{} this,
                  int size) {

            delay t {
                alloc t -> Wheel[Immutable, NotNull] as wheel;

                /* ‘init’ promises to initialize ‘current.next’ and
                     ‘root.prev’. Both are ‘wheel’, so it will
                    initialize all our NotNull fields. */

                this.init(wheel, size, wheel);
            };

            wheel
        }

    T -> Wheel[i, NotNull]<,>
        /* Promise to initialize ‘current.next’ and ‘root.prev’.
            This makes recursive initialization possible. */
        {current -> next, root -> prev}
        /* Require time parameter T to be in the future; the Wheel
            is under initialization.  */
        {T}
        init(
             /* Committed objects must be fully initialized! */
             (Now -> WheelFactory[i, NotNull])^{} this,
             /* Wheel which can be completely uninitialized */
             (T -> Wheel[i, NotNull]<,>)^{prev, next} current,
             int size,
             /* Similarly, ‘root’ can be uninitialized */
             (T -> Wheel[i, NotNull]<,>)^{prev, next} root) {

        if (size == 1)
            /* Fulfil our initialization obligations */
            current.next = root;
            root.prev = current;
        else
            /* Parametric time constraints allow us to assign a new
                 Wheel with the same commitment point as ‘current’
                and ‘root’ */
            alloc T -> Wheel[i, NotNull]<,> as z;

            /* Fulfil our initialization obligations */
            current.next = z;
            z.prev = current;
            /* If we fail to initialize ‘z’ the method will not
                type-check */
            this.init(z, size-1, root);

        root
    }
}
...
```

Figure 55: A cyclic, non-`null`, Immutable linked structure with no distinguishable root node. Assumes the presence of integers, booleans, and an if statement (with the normal behaviour).

```
delay t {
    alloc t -> WheelFactory[Immutable, NotNull]<,> as factory;
}

wheel = <Immutable,> factory.makeWheel(3);

/* wheel is now a cyclic immutable structure. */
/* The following is guaranteed by the type system to be non-null */
wheel.next.next.next.next.next.next.next.next.next

/* It it impossible to initialize a ‘Wheel’ for which
    ‘wheel.next.next.next...next.next’
   is not guaranteed non-null */

/* wheel.next is an indistinguishable cyclic immutable structure;
    there is no root object which must own the others */

/* We can also make mutable Wheels: */
wheel2 = <Mutable,> factory.makeWheel(4);
wheel3 = <Mutable,> factory.makeWheel(4);
wheel2.next.next = wheel3.next.next;
wheel2.next.next.next.prev = wheel2.next.next;
wheel3.next.next = wheel2.prev.prev;
wheel3.next.next.next.prev = wheel3.next.next;
/* Now we have a figure of eight; a strict ownership tree would not
     allow the nodes to be shared between wheels in this way. */
```

Figure 56: Listing demonstrating the usage of a `WheelFactory` from Figure 55

**Initialization Region** We began this document by noticing that the two concerns of `null`ity and mutability have an obvious point of cross-over: the challenge of initialization. In either case, the guarantees and constraints of the type system must be relaxed while an object is initialized. It is not enough just to notice the commonality; we must ensure the initialization period of an object with regard to `null`ity is compatible with its initialization period with regard to mutability. By making these initialization periods one and the same, we both create a simpler system (an object is either under initialization or not), and ensure that the two concerns (of initializing non-`null` fields, and avoiding mutation of immutable objects) do not interfere with one another: it would be impossible to initialize the non-`null` fields of an object after it became immutable.

**Objects with differing levels of initialization** In systems for `null`ity, including [18] and [4], objects which will be initialized at different times are allowed to hold references to one another: the condition is that for one object to be stored as a field in the another, the former must be initialized *no later than* the latter. If that is the case, it is always safe to treat the fields of an object as being *as initialized as* the object itself. The reason this is safe is that, in `null`ity systems, it is always safe to treat an object as being less initialized than it actually is. In mutability systems, on the other hand, this is not the case: if we treat an immutable object as being less initialized than it is, then we might see it as mutable when it is not. This is exactly what we want to avoid in systems of mutability. For this reason, we must give up some of the flexibility possible in systems of `null`ity with regard to which objects we can store in the fields of others, in order to meet the requirements of a system of mutability. We saw an example in Section 7.8.2 of allowing such a situation.

# 10 Further Work

Whilst we have presented a full and detailed picture of the treatment of generic mutability and `null`ity types, there are several avenues for further work. Perhaps most important is completing the proofs of language properties.

## 10.1 Proofs of language properties

**Maintaining a well-formed heap through method call** We have given an argument, without proof, that the well-formedness of a heap and stack is maintained through method call. Before the system can be considered complete, it is necessary to prove this claim.

**Generation Lemma** As we noted at the beginning of Section 8.1, we implicitly assume a generation lemma, which allows us to assume that the typing of expressions is carried out by rules exactly determined by the structure of the expression; in particular, we assume that we do not have to take account of arbitrarily many subsumption steps. Whilst we do not think that such a lemma will be difficult to prove, for the sake of time and concision, we have left it out. It would be preferable to return to the proofs and make these assumptions explicit throughout, then prove that they hold.

**Maintaining immutability** We have shown in Theorem 8.2 that when a path is typed as `Immutable`, we can be sure that no other part of the program types it as `Writable`. We know that the fields of an object that we believe to be `Immutable` cannot be written to. We did not prove Theorem 8.3, i.e. that this property (that the address is not typed as `Writable`) is maintained throughout the life of the program. Theorem 8.2 enables us to make better judgements as programmers than in systems which do not speak to mutability: for example, we need not worry that we can accidentally changed an `Immutable` object via an alias. Theorem 8.3 property would have allowed us to make optimizations in a runtime environment, or in the context of multithreading; for example, we could allocate `Immutable` objects in read-only memory, and if we could guarantee that subsequent execution did not affect the mutability of the object, we could avoid the need to lock on `Immutable` objects that are shared between threads (without this theorem, we could not guarantee that another thread does not reach a point in the program where the object was no longer seen as `Immutable`). We presented an outline of the argument for a proof, but did not formalize it. This is an important next step.

## 10.2 Expected features

**Class-parametric types** Types are not parametric in their classes. There is no possibility to have a `List<Hat>` instead of a `List<Item>`, for example. Further, there is no casting construct in this language, so it would be impractical to simple use a `List` of `Objects`, and then cast to the appropriate more specific type. Whilst it is practical to use languages without types that are parametric in their class, it would make the language more usable (and we should note that the other type systems to which we compare ours *do* allow parametric classes).

**Immutable fields** We currently allow only for object-level immutability: it is possible to specify that an object as a whole is immutable, or that an object pointed to by a field is immutable, but not that we have a (single) immutable field. It would be desirable to have a construct like `Java`'s `final`, to indicate that a field cannot be changed, even though the object as a whole (or the object referenced by the field) might be `Mutable`.

**Local (temporary) variables** The only opportunity to assign new variables to the stack is currently as the result of object allocation or method call. Obviously it makes sense to allow local variables to be stored as the result of field lookup as well; this would allow us to remove the method `List.getNext()` from Figure 52

## 10.3 Creating a usable language

**Implementation** There is no implementation of either our type checker, or an interpreter for runtime expressions. It would be a reasonable extension to our work to create such an implementation, and there are no obvious barriers to doing so (a proof of Theorem 8.3 and the soundness of method call aside).

**Shorthand and defaults** As it currently exists, the language is extremely verbose. Even when viewed in the context of other comparatively heavy-weight type systems, such as OIGJ, the type annotations required for our system are laborious to write. This is, perhaps, to be expected; we introduce three major new concepts to a traditional object-oriented programming language: Mutability, `null`ity, and initialization time. We also require the programmer to specify uninitialized fields of objects expected as arguments to methods.

In [18], the authors give a sensible set of defaults for their `null`ity constraints on references. It would be good to investigate whether it was sensible to provide a similar set of defaults, so that, for example:

```
C[<? extends ReadOnly>, <? extends Nullable>]<,>
```

could become simply:

```
C?
```

with `?` meaning that an object was possibly-`null` and `ReadOnly` being the default mutability. Since the class $C$ does not expect any extra parameters, it should not be necessary to specify an empty list. It is more than possible that we could construct a much more succinct system by common-sense defaults such as these. Likely candidates include:

- Uncommitted method parameters should implicitly expect all not-`null` fields to be uninitialized. In Figure 55 we specify them explicitly.
- Committed method parameters should implicitly expect all not-`null` fields to be initialized. Whilst the type hierarchy treats committed objects in this way, the user must still specify the (empty or otherwise) list of possibly-uninitialized fields.
- Initialization of arguments should be committed or unknown by default. Where the initialization state of an argument does not matter, the user should not have to specify. Either committed of unknown might make a sensible default.
- The receiver of a method is always `NotNull`. There should be no need to specify.

**Case studies** Whilst we have shown through extensive example a range of possibilities for initialization patters under our system, it would be informative to attempt to port "in-the-wild" programs into the language; are there situations which require initialization patters that we are unable to type check? [18] and [21] both annotated large bodies of existing code in order to show the expressivity of their systems, and for the sake of comparison we could do similarly.

## 10.4 Language extensions

**Standard imperative constructs** In Figure 55 we noted the assumption of integers, booleans, and an if-statement. Additionally, looping constructs are universal in imperative languages. We know that it is possible to encode all these constructs in a polymorphic object-oriented language (for example, we encoded booleans in Figure 24), but most programmers expect them as a first-class part of the language.

**Standard constructs in higher-level languages** We do not support implicit constructor call, nor do we support common features such as method overloading or contravariant method signatures in subtypes. Such capabilities would probably be uncomplicated to implement, and would add flexibility to the language, but we did not consider them important to the idea of `null`ity or mutabililty types.

**Threading** The system as a whole was considered in the context of threading; one of the major advantages to immutability in a type system is to eliminate the need for locking on shared data-structures. We would like to investigate the practicalities of adding basic multi-threading constructs to the system. There should be no new challenges presented by our system, with one consideration: objects which are uncommitted should not be shared between threads. We would also require a proof of Theorem 8.3, that new `Writable` paths are not created once an object is seen to be `Immutable`; otherwise we run the risk that one thread creates a new `Writable` path whilst another is still in a scope that considers the object to be `Immutable`.

**Static method resolution** At the end of Section 10.3 we noted that the receiver to a method-call is always `NotNull`. This is true because the run-time semantics resolve method-call by the runtime class of the method's receiver. In compiled languages like `C++` or `C#`, we use the type information (that we found when we type-checked the program) to resolve method call ahead-of-time for certain methods, and we require runtime information only for methods which are declared `virtual`. This is also true of `Java` – but in `Java`, *all* methods are virtual, unless declared otherwise. If we followed a static strategy for method resolution like these languages, we could allow the receiver of a non-`virtual` method to be `null` without any problem resolving method lookup, since it would be decided when we type-checked the program.

## Conclusion

We have given a formal system for treatment of mutability and `null`ity constraints. We unify the two concepts in a natural way, creating a type system that is expressive and intuitive. The main novel concept here is the treatment of `null`ity constraints in a generic way, and we hope this will form the basis of future work on the topic of `null`ity (in particular, the subtleties noted in Section 7.8 will warrant consideration by authors considering parametric `null`ity). We treated the initialization problem on the basis of previous work ([6], [4]), and provided concrete examples of the power and flexibility of the system presented.

## References

[1] Martn Abadi and Luca Cardelli. A theory of primitive objects. In Masami Hagiya and JohnC. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer Berlin Heidelberg, 1994.

[2] Gilad Bracha. Generics in the java programming language. java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[3] Microsoft Corporation. Specsharp. https://research.microsoft.com/en-us/projects/specsharp/, Dec 2011.

[4] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350, 2007.

[5] Manuel Fhndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language, 2003.

[6] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a java-like language. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 347–362, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Jon Harrop. *F# for Scientists.* Wiley-Interscience, New York, NY, USA, 2008.

[8] Shan Shan Huang, David Zook, and Yannis Smaragdakis. cj: enhancing java with safe type conditions. In Brian M. Barry and Oege de Moor, editors, *AOSD*, volume 208 of *ACM International Conference Proceeding Series*, pages 185–198. ACM, 2007.

[9] JetBrains. Kotlin. http://confluence.jetbrains.net/display/Kotlin/Null-safety, May 2012.

[10] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *PLDI*, pages 1–12, 2001.

[11] Philippe Leybaert. Can I change a private readonly field in C# using reflection? http://stackoverflow.com/a/934942.

[12] Digital Mars. Const and immutable. http://www.d-programming-language.org/const3.html, Dec 2011.

[13] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, London, England, 2002.

[14] polygenelubricants. change private static final field using java reflection. http://stackoverflow.com/a/3301720.

[15] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. *SIGPLAN Not.*, 44(1):53–65, January 2009.

[16] Inc Red Hat. Ceylon. http://ceylon-lang.org/, May 2012.

[17] Alexandra Rusina. Covariance and contravariance faq. https://blogs.msdn.com/b/csharpfaq/archive/2010/02/16/covariance-and-contravariance-faq.aspx?Redirected=true, Febuary 2010.

[18] Alexander J. Summers and Peter Müller. Freedom before commitment: a lightweight type system for object initialisation. In *OOPSLA*, pages 1013–1032, 2011.

[19] Mads Torgersen, Christian Plesner Hansen, and Erik Ernst. Adding wildcards to the java programming language. In *Journal of Object Technology*, pages 1289–1296. ACM Press, 2004.

[20] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, and Michael D. Ernst. Object and reference immutability using java generics. In *In ESEC/FSE*, pages 75–84. ACM Press, 2007.

[21] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2010)*, pages 598–617, Revo, NV, USA, October 19–21, 2010.