Declaring and checking non-null types in an object-oriented language

Manuel Fähndrich K. Rustan M. Leino

Microsoft Research One Microsoft Way Redmond, WA 98052, USA {maf,leino}@microsoft.com

ABSTRACT
Distinguishing non-null references from possibly null references at the type level can detect null-related errors in object-oriented programs at compile-time. This paper gives a proposal for retrofitting a language such as C# or Java with non-null types. It addresses the central complications that arise in constructors, where declared non-null fields may not yet have been initialized, but the partially constructed object is already accessible. The paper reports experience with an implementation for annotating and checking null-related properties in C# programs.

1. INTRODUCTION
Vital to any imperative object-oriented programming language is the ability to distinguish proper objects from some special null object or null objects, commonly provided by the language as the constant null. When designing a program, programmers need to consider whether a value may be null, and often need to handle null differently than proper objects. Since such handling can be error prone, it is preferable that a compiler or tool enforces the programming discipline the programmer intended and points out places where the code may have errors.

Perhaps the clearest and most direct way for a language to accommodate such tools is to type expressions according to whether or not they may yield null. However, the type systems of mainstream object-oriented languages such as C# [2] and Java [8] provide only one object type per declared class, and null is a value of every such object type. In this paper, we propose splitting reference types into possibly-null and non-null types, so that expressions that may yield null can be identified.

Type systems in which one can distinguish special values such as null from proper values exist. The tagged unions in the object-centered language CLU [11] are a good example. Similarly, object-oriented languages such as Theta [10] and Moby [4] as well as functional languages such as ML [12] or Haskell [15] make distinctions between possibly null and non-null references in their type systems. In these languages, the declaration of a field of non-null type provides the following three-part contract:

- The construction of an object or record must initialize the field with a proper non-null value
- Every read access of the field yields a non-null value
- Every update to the field requires a non-null value

The non-nullity of such a field therefore becomes an object invariant that is enforced statically (at compile-time) by the type system.

All the above mentioned languages use a simple mechanism to establish this object invariant:

An object/record under construction cannot be accessed until fully constructed.

This programming restriction makes it fairly simple to prove the three-part contract above.

Unfortunately, the mainstream languages C# and Java give access to the object being constructed (through this) while construction is ongoing. This extra flexibility makes reasoning about proper initialization of objects much harder, both for the programmer and for an automatic tool, such as the type system we are proposing in this paper.

Being type-safe languages, C# and Java do ensure that fields have zero-equivalent values of their type (null for references) before an object being constructed can be accessed, but for fields declared as containing a non-null reference, this null-initialization is not sufficient, since the field is not properly initialized. In this paper, we use the adjectives partially initialized or raw for objects containing non-null declared fields that may be uninitialized, and our type system distinguishes raw objects from fully initialized objects.

Example The following C# code illustrates the problem of dealing with partially initialized objects. Class A con-
tains a field name of type string that is annotated as being non-null. (In our examples, we use annotations of the form [NotNull] to annotate types with null-related attributes.)

class A {
    [NotNull]
    string name;

    public A([NotNull] string s) {
        this.name = s;
        this.m(55);
    }

    virtual void m(int x) { ... }
}

The constructor for A correctly initializes field name with a non-null string that it obtains as a parameter. It then proceeds to call the virtual method m on the object being constructed.

Although this code may look correct—after all, class A properly initializes its field—the correctness of this code can actually not be guaranteed. To see why, consider the following code (possibly declared in a separate module):

class B : A {
    [NotNull]
    string path;

    public B([NotNull] string p, [NotNull] string s) {
        base(s)
        this.path = p;
    }

    override void m(int x) {
        ... this.path ...
    }
}

Class B extends class A with another field path that is also declared as being non-null. The constructor of B correctly calls the base class constructor of A, and then initializes its own field path.

The problem with the code is that during the base call to A’s constructor, the virtual method B.m may be invoked. At this time, field path of the object under construction has not yet been initialized. Thus accesses of this.path in method B.m may yield a possibly-null value, even though the field has been declared as being non-null.

In this paper, we propose a way to retrofit a language like C# or Java with non-null types. The proposal does allow access to the object being constructed before it has been completely constructed, but only ways that can be statically checked for soundness. Thus, our proposal accommodates many modern programming styles.

The contributions of the paper are as follows:

- We give the first sound technical solution to deal with explicit object initialization in the presence of inheritance subtyping and where access to the object under construction is allowed. The main insight we are using is that initialization is monotonically evolving from uninitialized to initialized, and we capture this aspect for objects in the presence of unknown object extensions.
- Our proposal is backward compatible with existing main-stream languages in that it does not require changes to the language semantics or runtime implementations. Any program accepted by our type system is also a valid C# (resp. Java) program with unchanged behavior.

The advantages of adding non-null types to a language like C# or Java include:

- Statically checked interface documentation: Clients know when a method expects a non-null argument, and know when it promises a non-null return value. Implementations can rely on the non-nullity of declared parameters and are held to promises of non-null results.
- Statically checked object invariants: Object invariants such as fields holding non-null references can be declared and statically checked.
- More precise error detection: The error of using null when a program’s design expects a non-null value is detected at the program point where the error is committed, which often comes before the program point where an object dereference operation uses the value.
- Performance optimizations: Given a reference of a non-null type, dereference operations and throw statements can proceed without the normal null check, thus providing a possible runtime advantage in some cases. If the runtime supports non-null types, the programmer can limit where runtime checks are inserted by judicious use of non-null types. The freedom from such runtime checks also enables effective compiler optimizations, in part due to fewer possible exception paths [5].
- Fewer unexpected null reference exceptions: The C# language reference document lists 8 cases when a NullReferenceException can be thrown. Given a non-null type in those contexts, the compiler guarantees that such operations do not throw null exceptions.
- Basis for other checkers: The use of non-null types facilitates the task of writing other program checking tools for the language by eliminating a large source of false warnings.

The rest of the paper is organized as follows: Section 2 introduces non-null types. Section 3 deals with the crux of the paper: how to establish object invariants. Sections 4 and 5 extend the proposal to array types and to C# value types. Section 6 examines the impact of our design on methods with call-by-reference parameters, on static class fields, and on generics. Section 7 describes a checker we have implemented and Section 8 discusses our experience in using the checker on a non-trivial program. Section 9 describes design alternatives, Section 10 discusses related work, and Section 11 concludes.

2. NON-NULL TYPES

For every declared class or interface T, we propose the addition of a distinguished reference type T+ for non-null references (proper objects) of type T. To avoid confusion, we write T+ (rather than just T) for types including the
null value. That is, C# and Java currently provide just the maybe-null type $T^+$, not the non-null type $T^-$. (Here and throughout, our notation is used to describe concepts, not to propose language syntax.)

Where the language currently requires an expression of a reference type $T^+$ and stipulates that a null reference exception be thrown at runtime if the expression evaluates to null, we instead require that the expression be of type $T^-$. For example, our field dereference operator “.” takes an expression of a non-null type as its left-hand argument (and a field name as its right-hand argument).

The types $T^+$ and $T^-$ can be used whenever a type is expected. For example, formal parameters and method results can be declared to be of type $T^-$. Both C# and Java have definite assignment rules for local variables. Uninitialized local variables do not evaluate to null, but instead cannot be read until after they have been assigned. Thus, local variables with non-null types are supported nicely in these languages, since the eventual initializations of such variables are forced to assign non-null references.

As one would expect, if $S$ is declared to be a subclass of $T$ or $T$ is a super-interface of $S$, then $S^+$ is a subtype of $T^+$ and $S^-$ is a subtype of $T^-$. Furthermore, for any $T$, $T^-$ is a subtype of $T^+$. Therefore, an expression of type $T^-$ can freely be assigned to a variable of type $T^+$, but to go the other way (to narrow the type) requires a test. For example:

$$T^- \overset{\text{t = new T(...)}}{\rightarrow} // allocate a non-null object$$
$$T^+ n = t; \quad // this direction is always allowed$$

\[\text{if}(n! = \text{null}) \{\text{int } x = t.f;\quad // type of } t \text{ must be non-null}\]

Note that we have now removed null reference exceptions from the language, since all null violations now instead show up statically as type errors.

As the code snippet above shows, an application of new $T(...) \rightarrow T^-$, since the object constructed is always non-null.

For an expression $e$ and a type $T$, the expressions $e$ is $T$, in C# and $e$ instanceof $T$ in Java return true if $e$ evaluates to an object of type $T$ that is not null. We do not need these expressions to be extended to $e$ is $T^+$ or $e$ is $T^-$, since tests against null can already be written in these languages directly.

Furthermore, in C#, the expression $e$ as $T$ returns $e$ if $e$ is $T$, and returns null otherwise. Again, no change to the language is needed since, under our proposal, there is no difference between the expressions $e$ as $T^+$ and $e$ as $T^-$, and both expressions have type $T^+$.

This would be the entire story, except for the existence of compound values, namely the data records of objects, the elements of arrays, and the fields of value types. As our example in the introduction shows, the construction of these compound values complicates the story a good deal. Let’s look at object construction first, then at array construction, and finally at value type construction.

### 3. CONSTRUCTION OF OBJECTS

A field (instance variable) $f$ in a class $C$ may be declared with a non-null type $T^-$. Consequently, one expects an expression $c.f$ to yield a non-null value (where $c$ is of type $C^-$). But during the construction of a $C$ object—that is, during the execution of the constructor of $C$ and the constructors of the superclasses of $C$

\[\text{this}, f \text{ may not have been initialized yet, where this denotes the object being constructed. So, a use of the value this.f may yield null, despite the fact that } f \text{ is declared to be of the non-null type } T^-! \]

Because C# and Java do not limit the use of this during construction, the problem is not limited to cases where field $f$ is accessed through the special keyword this; if this is passed as a parameter $x$ to another routine, for example, then $x.f$ in the callee may also yield null despite the fact that $f$ is declared of type $T^-$. Before we propose a type-based solution to this problem, let us examine where and why the example in the introduction is faulty. Consider the object under construction (this) within $A$’s constructor, just after the initialization of field $n$. At this point, we know that the fields declared in class $A$ are properly initialized, and the fields of all super classes of $A$ are properly initialized (because the language semantics guarantee that $A$’s constructor must have called the base constructor as well). What we don’t know at this point is that the fields of any potential subclasses of $A$ are properly initialized. From a type system perspective, the type of this at the method call this.n(55) is not really an object of type $A$ just yet, that is, it cannot be used in every context where an $A$ object is expected. The one context where it cannot be used yet is in virtual calls, because virtual calls may implicitly reveal the state of subclasses that have not been initialized yet.

We propose to solve this problem by introducing another family of types: for any class $T$ (not interface), $T^{\text{raw}}$ denotes the partially initialized objects of class $T$ or subclass thereof. More precisely, for any class $T$, $T^{\text{raw}}$ denotes a value of the same structure as a value of type $T^-$, except that any field of the former may yield null, even if the field is declared with a non-null type. That is, if $f$ is a field of type $T^-$ in a class $C$, then the expression $c.f$ may evaluate to null if $c$ is of type $C^{\text{raw}}$. However, we require that expressions assigned to $c.f$ be of type $T^-$, even in the case where $c$ is of type $C^{\text{raw}}$.

The restrictions above guarantee that an object, once fully initialized, never becomes uninitialized again; in other words, once a $T^-$ field of an object is initialized to a non-null reference, the field will never contain a null value. This invariant is necessary to achieve soundness, for it is possible to have two references to the same object $o$, one via $x$ typed $C^{\text{raw}}$, the other via $y$ typed $C^-$. The former may have been captured during the construction of the object, the latter after the construction has completed. If we were allowed to assign a null value to $x.f$, then a subsequent read of $y.f$ would result in null, even if the declared type of $f$ were $T^-$. With the restrictions in place, objects evolve monotonically towards full initialization. This innovation enables us to keep the overhead of checking field initializations to something manageable.

We require that, by the end of every constructor of class $C$ (including the default constructor, if any), every non-null field declared directly in class $C$ has been assigned. That is, we require that every path through a constructor
Object broken into class frames.

Object of type $B^{raw-}$. Partially-initialized frames are marked with an asterix *. Note that all frames of unknown class extensions are considered partially-initialized.

Object of type $B^{raw(A)-}$.

Object of type $B^{raw(B)-}$. Note that the frames of unknown class extensions are still considered partially initialized.

Figure 1: Illustration of class frames and raw types

to a normal return include an assignment to every non-null field (except fields with field initializers). We refer to the definite assignment rules of C# and Java for the details of the definition of “every path”. Our rule means that by the time the newly constructed object is returned to the caller of new, all of its non-null fields have non-null values. Hence, for any class $T$, new $T(...)$ has type $T^-$, not $T^{raw-}$. In effect, the “last” constructor takes care of casting the object being constructed from type $T^{raw-}$ to type $T^-$. More technically, we break an object into a stack of class frames, where each class frame represents the fields introduced by the declarations of a particular class (see Figure 1 top). Thus, the object in our example of dynamic type $B$ has 3 class frames, one for class $B$, one for class $A$, and one for the root class object. For each type $T^{raw-}$, we can then distinguish an entire family of raw types of the form $T^{raw(B)}^-$, where type $S$ is a supertype of $T$. The extra type $S$ marks the lowest class frame that is properly initialized. Thus, every class frame at or above type $S$ is properly initialized, where as frames strictly below $S$ are not yet known to be initialized. Figure 1 illustrates these cases for various raw types of statically known class $B$.

The inclusion of all possible class extensions in our raw types makes it easy to handle ordinary type down-casts from $A^{raw-}$ to $B^{raw-}$, as illustrated in Figure 2. With this refinement in hand, we can precisely state the type of an object returned by a particular constructor. On entry to a constructor of class $T$, the this has type $T^{raw-}$. After the call to the base class $B$ constructor, the type of this is $T^{raw(B)-}$. At the end of the constructor, the type of this is $T^{raw(T)-}$.

Thus the result of an expression new $T(...)$ has type $T^{raw(T)-}$, that is, an object where all frames at or above class frame $T$ are initialized. Since we know that $T$ is the dynamic type of the object—there are no subclass frames—the object is fully initialized, and thus has type $T^-$ (see Figure 3).

3.1 Subtyping of raw types

As one would expect, if $S$ is declared to be a subclass of $T$, then $S^{raw-}$ is a subtype of $T^{raw-}$. Furthermore, if $S$ is a subtype of $R$, then $T^{raw(S)-}$ is a subtype of $T^{raw(R)-}$, the latter being less initialized than the former, similarly, $T^{raw(R)-}$ is a subtype of $T^{raw-}$, where the latter is the maximal partially initialized type in the family $T$. Also, for any $T$, $T^-$ is a subtype of $T^{raw-}$.

For completeness, we also introduce a maybe-null type for partially initialized objects, written $T^{raw+}$. If $S$ is declared to be a subclass of $T$, then $S^{raw+}$ is a subtype of $T^{raw+}$. Furthermore, for any $T$, $T^{raw-}$ is a subtype of $T^{raw+}$, and $T^-$ is a subtype of $T^{raw+}$. In practice, we don’t expect such types to be necessary.

Since this is of type $T^{raw-}$ in a constructor, any assignments of this to other variables can be done only if the other variable is of the appropriate type, namely a supertype of $T^{raw-}$. For example, if this is passed as a parameter, then the corresponding formal parameter must be a partially-initialized type. There’s no explicit place in C# and Java to give the type of the receiver parameter (for the case where this is passed as a parameter to a method by virtue of that method being invoked on this), but we can imagine adding one (perhaps by declaring an instance method with some special keyword). If a method is invoked on an object of type $T^{raw-}$, then the method’s formal receiver parameter must be of an appropriate partially-initialized type.

3.2 Correcting our example

Let us revisit our example from the introduction. As we have identified in our discussion, reads of field path in method B.m may return null, because the this object is not yet fully constructed. All that is needed to handle this example is to state explicitly in the signature of methods A.m and B.m, that the receiver this is partially-initialized.
We use the annotation \[Raw\] on the method to mark a method as callable on a raw object, that is, on objects of type $T^\text{raw}$, where $T$ is the type of this. Here is the corrected code.

```csharp
class A {
    [NotNull]
    string name;

    public A([NotNull] string s) {
        this.name = s;
        this.m(55);
    }

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

class B : A {
    [NotNull]
    string path;

    public B([NotNull] string p, [NotNull] string s) : base(s) {
        this.s = p;
    }

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

With these annotations, is now possible to verify that the code is consistent with our type rules for non-null types. At the call to method $m$ in $A$'s constructor, the type of this is $A^\text{raw(A)}$. The expected type of method $m$ is $A^\text{raw(A)}$, which is a supertype of $A^\text{raw(A)}$. Therefore, the call is valid.

Conversely, in the method body of $B.m$, we know this has type $B^\text{raw}$. Thus, any read accesses to this.path may yield null, and the method code must correctly handle this value. If method $B.m$ also accesses field this.name, then it would also have to expect null, unless we strengthen the type of this in the signature of method $m$ to $A^\text{raw(A)}$. In that case, method $B.m$ can rely on the fact that class frame $A$ of the object is properly initialized and thus field name is non-null.

### 3.3 Casts between raw and non-raw types

There may be situations where a programmer knows that an object is fully initialized, even though the type system cannot prove it. For this situation, we allow typecasts of expressions from a partially-initialized type to a fully-initialized type. We propose that such a cast succeed by checking each non-null declared field of the runtime type for proper initialization. This check can be implemented in C# and Java using reflection. Alternatively, completion of construction could be measured by the “last” constructor having finished and the object having been returned by the new expression that prompted its construction. Such an approach however requires support from the runtime since an extra bit per object is required.

The behavior of the operators is, as, and instanceof is now straightforwardly defined to take into consideration partially-initialized types.

### 4. ARRAY TYPES

Both maybe-null and non-null types are allowed as the element type of an array type. In addition, the array type itself (which is a reference type in both C# and Java) may be either a maybe-null type or a non-null type. We thus have the following types for any reference type $T$:

- $T^−[\]−$ non-null array of non-null elements
- $T^+ [\]−$ non-null array of possibly-null elements
- $T^− [\]−$ possibly-null array of non-null elements
- $T^+[\]−$ possibly-null array of possibly-null elements

The covariant array types in C# and Java work as expected in the presence of these new types, provided the runtime check on element assignment takes the non-nullness of element types into account. This aspect requires runtime support. For a design without runtime support, the covariant subtyping of array elements with respect to non-null can be disallowed.

As with object construction, there is a problem with the construction of an array. In particular, there is a problem if the element type of the array is a non-null type. We propose that the allocation:

```csharp
new T− [n]
```

where $n$ is an expression that gives the size of the array to be allocated, return an array of type $T^−[\]raw−$.

Analogous to the fields of a partially-initialized object, reading the elements from a partially-initialized array may yield null, and expressions assigned to the elements of a partially-initialized array must be non-null. However, unlike classes and fields, there is for an array no program point that corresponds to the end of a constructor, by which time the construction of the array is supposed to have been completed. Furthermore, a simple definite assignment rule won’t work to ensure that all array elements are assigned. Therefore, we instead let the programmer cast the array of type $T^−[\]raw−$ to an array type $T^−[\]−$ (or $T^−[\]−$) when the programmer claims to have assigned all elements of the array. The typecast performs a check that all the array elements have been initialized, that is, that they are non-null. A typical program fragment for array initialization thus has the form:

```csharp
T−[\]raw− aTmp = new T− [n];
...
// initialize the elements of aTmp
T−[\]− a = (T−[\]−)aTmp;
```

To require this check may seem expensive, but note that the cost of the program’s initializing each array element is likely to exceed the cost of the typecast expression’s checking that the array elements are indeed initialized.

### 5. VALUE TYPES

The C# language supports value types via struct declarations. Structs are data records similar to classes, but they are manipulated as values rather than as references to the data record. Structs are declared similarly to objects, with fields and methods. Struct constructors initialize the fields of a struct.
What distinguishes structs from objects from an initial-
ization perspective is that all structs in C# have a default
constructor that initializes fields to their zero-equivalent val-
es (i.e., null for reference fields). This default constructor
cannot be overwritten.

This poses a problem, since we want to allow structs with
non-null declared fields, that is, structs for which the default
constructor does not establish the invariant of the struct,
because it does not initialize such fields. To ease the pre-
sentation, we distinguish structs for which the default con-
structor is not sufficient from normal structs and call the
former istructs (since they have an invariant). A struct is
an istruct if it has a non-null declared field, or contains an
istruct.

We model a partially initialized istruct analogously to a
partially initialized object by giving it a raw type \( S^{\text{raw}} \).
A constructor for a struct \( S \) produces a value of type \( S \),
except the default constructor of an istruct, which produces
a value of type \( S^{\text{raw}} \). There is no non-null \( S^+ \) or possibly-
null type \( S^* \) for a struct \( S \), since a struct is not a reference.
Since fields can be istructs, we have to extend our rule
for accessing such fields in partially initialized objects. As
before, if the field is of reference type \( T^- \), then reading the
field yields a possibly-null value of type \( T^+ \). If the field is
an istruct \( S \), then reading the field yields a value of type
\( S^{\text{raw}} \), since the struct itself may not be properly initialized.
Assignments to the field however require a value of type \( S \).

Arrays of istructs are handled similarly to arrays of non-
ull references. An allocation of an istruct array produces a
partially initialized array of istructs, of type \( S^{\text{raw}} \). After
the array has been initialized to proper structs, an explicit
cast is needed to obtain type \( S \). This cast involves a
number of non-null checks per element to determine that it
is a properly initialized istruct of type \( S \).

The subtype relation on value types in C# only includes
a boxing conversion between a value type \( S \) and the class
root object. Adapting this relation to our raw and non-
ull types yields the following subtype relations, where \(<:\) denotes the subtype relation:
\[
S <: \text{object}^- \quad S <: S^{\text{raw}} \quad S^{\text{raw}} <: \text{object}^{\text{raw}}^-
\]

6. OTHER LANGUAGE CONSTRUCTS

6.1 Call-by-reference parameters

A further complication arises in languages like C# that
support call-by-reference (ref) parameters. A formal ref pa-
rameter represents the same storage location as the actual
parameter to which it is bound. A ref parameter can be read
and assigned to by the callee, and these operations have the
same effect as if they had been performed directly on the
actual parameter. As with any parameter whose value can
be read by the callee, the type of the formal ref parameter
must be a supertype of the type of the actual parameter.
Since a ref parameter can also be assigned to by the callee,
the type of the formal must also be a subtype of the type
of the actual. That is, for ref parameters, the types of the
formal and actual must be identical.

The problem is that, for a class \( C \) with a field \( f \) of type
\( T^- \), if \( c \) is of type \( C^{\text{raw}}^- \), then \( c.f \) has type \( T^+ \) in a
read context and type \( T^- \) in a write context. The problem
also arises if \( f \) has type \( S^{\text{raw}} \) for a struct \( S \). Therefore,
we disallow an expression of the form \( c.f \) from being used
as an actual ref parameter if \( c \) is of a partially-initialized
type and \( f \) is a field of type \( T^- \) or \( S^{\text{raw}} \).

Note that there is no analogous complication with out pa-
rameters in C#. An out parameter is like a ref parameter,
except the callee must assign to the parameter before return-
ing, and if the callee reads the parameter it must first have
assigned to it. Because of the second of these stipulations,
any value the callee reads from the parameter is indeed of
the parameter’s declared type.

6.2 Static class fields

The runtime semantics of C# and Java guarantee that
static fields are null initialized. Furthermore, prior to the
first access to a static field \( T.f \), the runtime tries to exe-
cute the static class constructor for class \( T \). This algorithm
guarantees that static field initializers are executed before
their first access, except in the presence of cycles in the ref-
ence pattern between static class constructors, or forward
references within a class constructor to fields it hasn’t ini-
tialized yet. When a static field of a class is accessed whose
initializer is already running, the runtime simply obtains the
current value of the field, which may be null.

Although we could handle the entire initialization semi-
tics conservatively in our type system, it is not practical to
do so, since it wouldn’t be able to take advantage of the
common non-cyclic case. We therefore assume that cyclic
dependencies between static initializers of multiple classes
are symptomatic of a design problem and should be found
by other means. Testing is usually a reasonable way to de-
tect these problems, since the complexity of static initializers
is typically low.

We thus assume that any static field of a different class
accessed from a static constructor is already initialized. We
are left with the problem of forward references within a static
constructor to fields that are initialized later. For each class
\( T \), we treat its static fields as belonging to a special static
object of type \( T \) that is implicitly passed to all methods.
By default, all methods expect these static objects in the
fully initialized state, i.e., as \( T^- \) for each class \( T \),
thereby relying on the initialization of all static fields.

During execution of the static constructor for class \( T \), we
assume that \( T \)’s static fields are uninitialized, which is ex-
pressed by giving the static object type \( T^{\text{raw}}^- \). Each static
constructor is responsible for initializing the static fields of
its class.

To call a method from the static constructor of class \( T \),
the called method must be specially annotated as handling
the raw static state of class \( T \). In other words, such a
method must not rely on the static fields of class \( T \) being
initialized. However, the method can rely on the initializa-
tion of static fields of other classes.

In practice, we think these restrictions are reasonable. If
methods are called from static constructors, they must in
general be aware that not all fields are initialized yet. The
most common methods called from static constructors are
instance constructors. Thus some of these must be anno-
tated.

6.3 Generics

An upcoming release of C# will add support for generic
types and methods to the language. Similar additions
are planned for the Java language. This section briefly explores
the impact of generics on our proposal.
Our distinctions between possibly-null and non-null values at the type level are orthogonal to the generics proposals in the sense that type abstraction in those proposals abstracts only over the underlying class/interface type, but does not abstract over the nullity of references of such types. Therefore, if $T$ is a bound generic type in the context of some class or method, it will be possible to form types $T^-$ and $T^+$. Instantiations of $T$ will only be other types $S$ without nullity modifiers.

The opposite approach, where type abstraction also abstracts over nullity is problematic, since it leads to situations where we need to give meaning to types of the form $T^+$ in contexts where we instantiate $T$ with $S^-$. The addition of type genericity will not automatically provide genericity over nullity. Such genericity is orthogonal and has to be added independently.

7. IMPLEMENTATION

To evaluate our design, we implemented a non-null checker for C#. This section describes our implementation and simplifying assumptions.

We augment type declarations in C# programs using a language feature, called custom attributes. Custom attributes are structured comments that persist into the compiled object code. A custom attribute consists of a name, plus zero or more positional and named parameters, whose values are limited to compile-time constants of a few basic types.

For our purposes, we defined two attributes without arguments named $\text{[MayBeNull]}$ and $\text{[Raw]}$ for annotating fields, parameters, and results. The following table lists the correspondence between the types in our design and the C# syntax.

$$
\begin{align*}
T^- & \rightarrow T \\
T^+ & \rightarrow [\text{[MayBeNull]}] T \\
T^{\text{raw}} & \rightarrow [\text{[Raw]}] T \\
T^{\text{raw}(S)^-} & \rightarrow [\text{[Raw]}(\text{Upto=typeof}(S))] T
\end{align*}
$$

As the table shows, we chose the default for a reference to be a non-null type. This choice requires fewer annotations than making the possibly-null case the default. It would be a simple matter to allow alternative class-wide or module-wide defaults.

Since attributes are preserved in the CIL (common intermediate language) bytecode produced by the C# compiler, we decided to implement our checker at the CIL level, rather than at the C# language level. This approach offers several advantages: 1) no source code parsing and semantic disambiguation is necessary, 2) only a small and well-defined set of instructions needs to be handled, and 3) the same checker works for any language that compiles down to CIL (for example Visual Basic).

Local variables in method bodies cannot be annotated with attributes. Our checker infers the annotations using a simple flow-sensitive intra-procedural data-flow analysis. The analysis is smart enough to refine the annotations in branches of tests against null. As a result, no explicit cast from $T^+$ to $T^-$ is provided. Programmers must instead test a possibly-null value before using it.

The typecast from $T^{\text{raw}}$ to $T^-$ in our design is implemented by a special method

```
static void AssertInit([Raw] object rawobj);
```

that uses reflection to dynamically check that all non-null declared fields are indeed non-null. The checker recognizes calls to this method and treats the argument as initialized in the continuation.

Our checker does not yet implement the full design described in this paper. The reason for the omissions is simply to keep the implementation effort small rather than technical difficulties. The differences are as follows:

- No support for non-null array elements.
- No support for annotations to make methods callable from static constructors.

Furthermore, the checker assumes programs are free of synchronization errors. For example, after a possibly-null field has been tested against null, a subsequent read of the field (without intervening assignments to the field or method calls) is assumed still to be non-null, even though another thread could potentially update the field to null.

Finally, our current implementation does not examine exceptional control flow paths, that is, handler blocks are not checked, but we believe the current design can handle them without additional techniques.

Since our current checker implementation is separate from the compiler and the runtime, all implicit null checks that the language imposes are still performed by the runtime during execution. Thus we any remaining warnings the checker issues will be handled automatically at runtime. Our primary motivation is not improvements in performance, but improvements in code quality.

7.1 Extensions

Our implementation makes use of a couple of small extensions that we have not described so far.

Strengthened return type Overriding methods may want to strengthen the result type from possibly-null to non-null. This is sound, and we allow such cases via another attribute $\text{[NonNull]}$.

Initialized field precondition Within constructors, there may be calls to accessors that return some aspect of the object under construction. By default, the checker flags such calls as errors, since the receiver is still raw. These accessors, however, typically read only one field and in instances we’ve seen, the fields being read were previously initialized. We added a simple refinement of the $\text{[Raw]}$ annotation of the form

```
[Raw(except='''fieldnames''')]  
```

that can be used to annotate such accessors. It states that the object is raw, except for the given initialized fields.

Helper initializers Some classes use a helper method called from the constructor to initialize all fields. For our checker to prove that at the end of the constructor, all fields are initialized, it needs an extra annotation on the helper method indicating that it acts like a constructor and initializes the fields of the current class. We added the following annotations to express these cases:

```
[Inits]  
[Inits(''fieldnames'')] 
```

7
The first annotation on a parameter or receiver of type $T$ states that on exit, the parameter has type $T^{\text{raw}(T)}$. The second annotation states that only the listed fields have been initialized.

8. EXPERIENCE

We have experimented with our checker on one of our own C# programs of roughly 20KLOC. The checker was able to validate non-nullness for 8000 individual places in the code, where, according to the .NET CIL semantics, a null-check is performed. The checker takes approximately 10 seconds to run on a 1.8GHz P4 laptop.

Perhaps surprisingly, we found that checking a simple property like non-nullness can point out higher-level design issues in the code. We describe the kinds of errors detected in our code base and the shortcomings of the current implementation.

8.1 Errors

Many non-null errors are simple failures to handle all possible cases in the program. Here, we focus on more subtle bugs we discovered.

Vacuous initialization We found several instances of the following statement in constructors:

```csharp
this.foo = foo;
```

where the right-hand side foo was intended to denote a parameter of the constructor. It turned out, however, that there was no such parameter and what looked like a field initialization was in fact a dummy assignment of the form

```csharp
this.foo = this.foo;
```

Use of wrong local The operator as is used to test the dynamic type of an object. It returns the first argument if its dynamic type is compatible with the tested type, otherwise it returns null.

```csharp
bool as(Q other) {
    T that = other as T;
    if (other == null) return false;
    if (this.bar != that.bar) ...
}
```

The code above intends to return false if other is not of type T. Unfortunately, the first test compares other against null instead of comparing that against null. The checker discovers the problem at the access of that.bar, since that may be null.

Use of as rather than downcast The checker assumes that $y$ in the code below can be null.

```csharp
// x has dynamic type T, but static type is Q
T y = x as T;
... y ...
```

As long as the unchecked invariant is true, the code looks fine, but if the invariant becomes false (because of code changes), the error gets caught later than desired as a stray null reference. It is better to use a downcast $y = (T)x$, because it will dynamically detect the error earlier and also keep the static type checker from issuing an error.

Field declared too high in class hierarchy We found two instances in our code where a field was declared in an abstract base class, but only some of the subclasses actually initialized and used the field. Making the field possibly-null in this scenario is undesirable since it caters to the non-using subclasses at the expense of the users of the field. It is better to move the field declaration to a derived base class (possibly inserted), so that the field does not appear in the subclasses that don’t use it. After this transformation, we detected some subclasses that did not initialize the field, but still accessed it later!

Sloppy inheritance Occasionally, inheritance is used purely for subtyping purposes, without the desire to inherit implementation. Such situations call for the use of interfaces. But in situations where the type to be subtyped is not an interface, inheritance is still used. This approach usually leads to ugly code using null to initialize base class fields. The approach may be viable if the subclass can correctly reimplement all methods of the base class. But that is not possible if the base class has public fields or non-virtual methods. In our code, null checking pointed out one such case in which an interface rather than a class type should be used as the common supertype of two implementations.

Non-instance method The checker marked several calls to instance methods from within constructors as not expecting the receiver object in the raw state. It turned out that these methods could be made static, since they didn’t access the receiver object.

Non-sealed class We found a couple of classes that trigger all of their behavior from calling the constructor, i.e., they compute some result during construction and cache it. The constructed object is then used to access the computed result only. Our checker marks method calls on this within the constructor as errors, since the receiver is still raw, but the called methods are not declared to expect a raw receiver. We fixed this case by making these classes sealed (or final in Java). Constructors of sealed classes know that after all fields are initialized, the object is no longer raw, since there cannot be any subclass fields.

8.2 Annotations

To give an impression of the density of annotations, the following table lists the number and kind of annotations on fields, parameters, receivers, and return types.

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>MayBeNull</th>
<th>Raw</th>
<th>Annotated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields</td>
<td>922</td>
<td>38</td>
<td>0</td>
<td>2.6%</td>
</tr>
<tr>
<td>Parameters</td>
<td>2367</td>
<td>64</td>
<td>1</td>
<td>0.5%</td>
</tr>
<tr>
<td>Receivers</td>
<td>1581</td>
<td>0</td>
<td>1</td>
<td>&lt; 0.1%</td>
</tr>
<tr>
<td>Returns</td>
<td>1581</td>
<td>40</td>
<td>0</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

In addition, the code contains 84 assertions of the form

```csharp
Debug.Assert(x != null);
```

where it was not possible to express an invariant using our current annotations. We used a single cast from Raw to a non-raw type using our AssertInit dynamic check.

With those annotations in place, the checker reports 40 spurious warnings due to our incomplete handling of static field initializers and arrays with null elements.

These numbers show that the annotation burden is very small. This stems in a large part from the fact that locals do not need to be annotated.
8.3 Shortcomings

Our experiment also revealed several shortcomings in our checker, most of which will require extensions to our annotation language.

Field precondition Some methods expect a possibly-null field to be non-null where in fact every calling context does establish this precondition. Our annotations are currently not rich enough to express this precondition. This case differs from the initialized field precondition case described in our extensions, in that non-null fields of raw objects never revert back to null, which makes checking easier.

Field postcondition Some methods establish that a possibly-null field is in fact not-null, and callers rely on it immediately after the call. Again, we need extra annotations to express this case. A similar case arises through predicate methods that test if a field is non-null and return a boolean. The context testing the result then deduces that the field is non-null.

Parent-child cycle There were two instances in our code base where a constructor creates a cycle between this and some object t that it creates and stores in a field child of this. The constructor passes this down to the constructor of the child object t which in turn stores it as a pointer to its parent. That is, the code establishes:

\[\text{this.child} == t \land \text{t.parent} == \text{this}\]

Our annotations do not allow us to establish this invariant. We believe a specialized set of annotations for the parent and child fields can be devised to capture this scenario. This problem has given rise to other solutions in the past [17].

Lack of polymorphism over nullity There was one case in our code base where a method of type

\[\text{object Visit(T arg1, object arg2)};\]

was declared and used in two incompatible contexts, namely one context passing in non-null objects as arg2 and expecting non-null objects in return, and one where the argument could be null. The context in which null was passed actually did not need that extra parameter. However, we think such cases are rare.

Unfortunately, the addition of generics to C# will not solve this issue, since nullity annotations are orthogonal to generics, and generic types cannot be instantiated with nullity information (see Section 6.3).

Staged initialization An idiom our approach cannot currently handle is staged object initialization. An object is only partially initialized at construction time. Later, some method is called that further initializes part of the object and from there on, the newly initialized fields never become null again. We are still working on ways to capture this scenario.

Properties with possibly-null values A C# property is a pair of methods, a getter and a setter method for reading and writing some aspect of an object. In the C# syntax, getter and setter calls look exactly like field reads and writes. Unlike field accesses however, the checker cannot refine the type of a getter after its result has been tested against null. If the property is accessed again subsequently, the checker assumes again that it returns a possibly-null reference. To avoid false positives in our checker, we bound the getter result to a local for both the test against null and the subsequent use.

Other invariants Some objects have more complicated invariants that cannot be expressed with our annotations. For example, an object may have two possibly-null fields, but at every moment, at most one of them is null.

9. DESIGN ALTERNATIVES

The main complication of our design is the handling of partially initialized objects. If we could redesign C# and Java, or influence the design of a new OO-language, we would certainly propose to eliminate access to partially initialized objects. Constructors should be split into three sections:

1. A prelude that must initialize all fields of the current class frame but without access to this. It is important though that this section has access to constructor parameters. For example, the field initializers in C# implicitly form such a prelude, but since they have no access to the constructor parameters, that feature is not frequently applicable.

2. A call to the base class constructor. At this point, the current class frame is fully initialized (and so are all class frames of subtypes).

3. A constructor body that has full access to this and where this is fully initialized.

For structs, the same design as above can be applied and unlike the current .NET CIL design, it must allow definition of the default constructor.

To properly initialize arrays, a syntactic form such, as an array comprehension, would serve the purpose.

10. RELATED WORK

The goal of our proposal is to introduce another degree of rigor into programming languages, a mechanism by which programmers can state their design decisions and get help from a static checker to identify places in the source code where the program does not live up to the intended design. This is similar to the goals of, for example, ESC/Java [7], a static checking tool whose annotation language provides a non-null modifier for variable declarations. Our proposal differs from ESC/Java in that object invariants in ESC/Java are not enforced under all circumstances, whereas we have aimed for a design that is sound.

The following category of related work has some form of non-null types and checking, but for non-object languages without inheritance subtyping—the main complication addressed by our proposal: LCLint [3] (a tool for checking various properties of C programs that also provides null and non-null annotations on references, but no soundness guarantees and no object invariants), MrSpidey [6] (a tool that analyzes Scheme programs for type errors, including null access), Vault [1] (a C-style language aiming at making low-level programming safer by providing typestate checking, including null reference checking), Cyclone [9] (a C-style
language providing explicit region-based memory management), CCured [14] (a tool that compiles and type checks C programs under a safer type system), and Typed Assembly Language [13] (a target language for typed compilation, which uses the idea of monotonic initialization via initialization flags, but not in the presence of inheritance subtyping).

Null-related work on object-oriented languages was mentioned in the introduction. None of these languages provides access to the object under construction, and therefore avoid the problem of having to deal with partially initialized objects [11, 10, 4].

Some null validations can be proven through other means, such as the presence of dominating accesses to the same object (see the Marmot paper [5]). Such a technique alone however cannot prove the kinds of invariants our system can establish. To obtain a modular analysis, the amount of annotations would not be significantly reduced compared to our proposal.

The thought of introducing non-null types in a language like Java certainly isn’t new. For example, at least two other proposals can be found on the web, by Staeta [18] and by Smith [16]. As both of these proposals suggest, non-null types are natural and can be valuable. However, neither proposal even mentions the more difficult problem of constructing objects with non-null components, let alone suggests a solution to the problem.

11. CONCLUSION

In summary, to retrofit an object-oriented language like C# or Java to have non-null types, we propose breaking the reference types into four families of types, by introducing a taxonomy along the following two axes: non-null types versus possibly-null types, and partially-initialized types versus fully-initialized types. Let $<$: denote the subtype relation, let $S$ and $T$ be any classes, or interfaces (where defined), where $T$ is a superclass or superinterface of $S$, and let $X$ and $Y$ be any types such that $X < Y$. Then the following relations hold:

$$
T^- < T^+ \quad T^+ < T'^{raw+} \\
T^- < T'^{raw-} \quad T'^{raw-} < T'^{raw+} \\
S^- < T^- \quad S'^{raw-} < T'^{raw-} \\
S^+ < T^+ \quad S'^{raw+} < T'^{raw+}
$$

and

$$
X[-]^- < X[+]^+ \quad X[+]^- < X[+]^{raw+} \\
X[-]^- < X[+]^{raw-} \quad X[+]^{raw-} < X[+]^{raw+} \\
X[-]^- < Y[-]^+ \quad X[+]^{raw-} < Y[+]^{raw-} \\
X[+]^- < Y[+]^+ \quad X[+]^{raw+} < Y[+]^{raw+}
$$

Our experience with an implementation of our proposal for C# has been positive in that it eliminated null-reference problems and unearthed a number of design level problems.

We end by sketching how our partially-initialized types may help with other problems related to initialization in C# and Java. First, for any readonly (in C#) or final (in Java) field $f$, after the allocation of an object $x$ and before the assignment to $x.f$, reading $x.f$ will return a zero-equivalent value. This may lead to unexpected behavior in a program, especially if $x.f$ is read in a method that is called from a constructor rather than in the constructor itself. Under our proposal, if $x$ is of a fully-initialized type, then $x.f$ is guaranteed to have its final value. Second, a constructor in C# and Java may ”leak” the object this being constructed before it is fully constructed, for example by assigning this to a global variable or a globally reachable field, or by throwing this (if the type of this is an exception type). This is dangerous because other parts of the program may then expect to use the object as if it were fully initialized. Because this is of a partially-initialized type, any such assignment under our proposal can be done only to variables and fields with partially-initialized types; these types will then moderate the expectations of other parts of the program. And under our proposal, the argument to throw must have type Exception (in C#) or Throwable (in Java), thus preventing partially initialized exceptions from being thrown.

Perhaps our partially-initialized types can help in establishing and maintaining object invariants more generally.

12. REFERENCES


[10] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay...


