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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]; D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, class invariants, programming by contract*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Languages, Reliability, Verification

Keywords

C#, Java, null references, type system, non-null types

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 [9] 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 [12] are a good example. Similarly, object-oriented languages such as Theta [11] and Moby [5] as well as functional languages such as ML [13] or Haskell [16] 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

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

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` contains 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 cannot actually 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 in 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 formalize this insight in the presence of unknown object extensions.
- Our proposal supports existing main-stream languages in their entirety. 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. But our type-system does rule out programs for which safety with respect to nullity cannot be proven.

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

- Statically checked interface documentation: Clients see when a method expects a non-null argument, and see 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 optmizations: 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 [6].
- 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 possibly-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 is 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 $S^+ <: T^+$ and $S^- <: T^-$, where $<:$ denotes the subtype relation. 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⁻ t = new T(...);   // allocate a non-null object
T⁺ n = t;            // this direction is always allowed
...
if (n != null) {
  t = n;             // in this context n has type T⁻
}
int x = t.f;         // type of t 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** $T(\dots)$ has type 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 — **this**.f 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 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 `name`. 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 guarantees that `A`'s constructor has 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.m(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. 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 solving this problem by introducing another family of types: for any reference type T, $T^{\text{raw}-}$ denotes the *partially initialized* objects of type 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 again contain a null value. This invariant is necessary to achieve soundness, for it is possi-

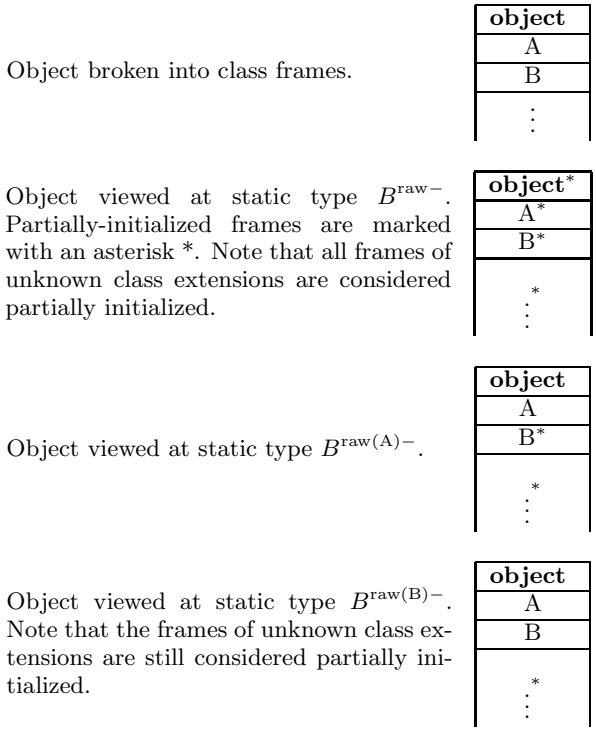

Figure 1: Illustration of class frames and raw types

ble 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 is 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 to a normal return include an assignment to every non-null field. 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** $T(\dots)$ has type T^{-}, not $T^{\text{raw}-}$. In effect, the "last" constructor takes care of casting the object being constructed from type $T^{\text{raw}-}$ 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 $T^{\text{raw}-}$, we can then distinguish an entire family of raw types of the form $T^{\text{raw(S)}-}$, 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, whereas 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.

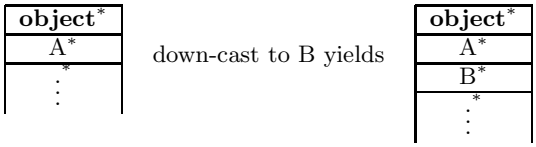

Figure 2: Illustration of down-cast from $A^{\text{raw}-}$ to $B^{\text{raw}-}$

If it is known that the static type B is equal to the dynamic type of an object, as is the case at a **new** B expression, then the type $B^{\text{raw(B)}-}$ is equal to B^{-}.

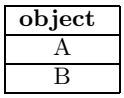

Figure 3: Implicit coercion from raw to non-raw if dynamic type is known

The inclusion of all possible class extensions in our raw types makes it easy to handle ordinary type down-casts from $A^{\text{raw}-}$ to $B^{\text{raw}-}$, 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 B, **this** has type $B^{\text{raw}-}$. After the call to the base class A constructor, the type of **this** is $B^{\text{raw(A)}-}$. At the end of the constructor of class B, the type of **this** is $B^{\text{raw(B)}-}$.

Thus the after the call to the B constructor in an expression **new** $B(\dots)$, the constructed object has type $B^{\text{raw(B)}-}$, that is, an object where all frames at or above class frame B are initialized. Since it is statically known that B is the dynamic type of the object—there are no subclass frames—the object is fully initialized, and thus the entire expression has type B^{-} (see Figure 3).

3.1 Subtyping of raw types

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

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

Since **this** is of type $T^{\text{raw}-}$ 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 $T^{\text{raw}-}$. 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 $T^{\text{raw}-}$, 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.

```
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.path = p;
  }

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

With these annotations, it 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 $A^{\text{raw}-}$, 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, given the raw type, 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. However, this alternative approach requires support from the runtime, since an extra bit per object is required.
As in the case of nullity, the behavior of the operators **is**, **as**, and **instanceof** is not affected and does not affect the rawness of the operands. These operators inspect only the named type of the object.

4. ARRAY TYPES

Both possibly-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 possibly-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-nullity 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 the construction of objects, there is a problem with the construction of arrays. In particular, there is a problem if the element type of the array is a non-null type. We propose that the allocation:

$$\textbf{new}\ T^-\ [n]$$

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

$$T^-\ []^{\text{raw}-}\ aTmp = \textbf{ new } T^-\ [n];$$
$$\ldots \quad \text{// initialize the elements of } aTmp$$
$$T^-\ []^-a = (T^-\ []^-)aTmp;$$

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 initialization perspective is that all structs in C# have a default constructor that initializes fields to their zero-equivalent values (*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 presentation, we distinguish structs for which the default constructor is not sufficient from other 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^{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^{raw}. There is no non-null type 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. Recall that if the field is of reference type T^-, then reading the field yields a possibly-null value of type T^+. In the additional case that the field is an istruct of type S, then reading the field yields a value of type S^{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-null 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 istructs, 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-null types yields the following subtype relations:

$$S <: \textbf{object}^- \qquad S <: S^{\text{raw}} \qquad S^{\text{raw}} <: \textbf{object}^{\text{raw}-}$$

6. OTHER LANGUAGE CONSTRUCTS

This section examines the impact of non-null types on langauge constructs we have not yet discussed.

6.1 Call-by-reference parameters

A further complication arises in languages like C# that support call-by-reference (*ref*) parameters. A formal ref parameter 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. Since these types are not identical, no type on the formal ref parameter will be identical to both. The problem also arises if field f contains an istruct.

One way to address this problem would be to introduce separate types for read and write accesses of ref parameters. However, to avoid further complicating matters, we simply 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 an istruct.

Note that there is no analogous complication with out parameters in C#. An out parameter is like a ref parameter, except the callee must assign to the parameter before returning, 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 of a static field $T.f$, the runtime tries to execute 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 reference pattern between static class constructors, or forward references within a class constructor to fields it hasn't yet initialized. 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 semantics 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 detect 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 $\mathcal{T}$ that is implicitly passed to all methods. By default, all methods expect these static objects to be in the fully initialized state, *i.e.*, as type $\mathcal{T}^-$ 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 expressed by giving the static object type $\mathcal{T}^{\text{raw}-}$. 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 initialization 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 annotated.

6.3 Generics

The next major 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 alternative 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^-.
Thus, 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 annotating fields, parameters, and results, we defined two parameter-less attributes, `[MayBeNull]` and `[Raw]`. The following table lists the correspondence between the types in our design and the C# syntax.

T^-	`T`
T^+	`[MayBeNull] T`
$T^{\text{raw}-}$	`[Raw] T`
$T^{\text{raw(S)}-}$	`[Raw(Upto=typeof(S))] T`

As the table shows, we chose the default for a reference to be a non-null type. We have found that 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 other languages that compile down to CIL (for example Visual Basic, and Managed C++).
In C# (and in the CIL), local variables in method bodies cannot be annotated with attributes. Instead, our checker infers the nullity and rawness type information using a simple flow-sensitive method-local (intra-procedural) data-flow analysis. The analysis is smart enough to refine the annotations in branches of tests against `null`. Programmers can thus use ordinary tests against null to refine a type from T^+ to T^-.
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 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.
Augmenting type declarations with nullity and rawness information allows our static checker to be completely modular. It analyzes each method body independently. At method calls, only the annotations on the called method signature are used to check the validity of the call.
Our checker implementation is also completely separate from the compiler and the runtime. We have not modified the CIL runtime in any way. All implicit null checks that the language imposes are still performed by the runtime during execution. Thus, a program that does not pass our checker can still be run, and the runtime will throw exceptions on null accesses.

7.1 Extensions

Our implementation makes use of a few 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 `[NotNull]`.

Initialized field precondition Within constructors, it is sometimes convenient to call an accessor that returns 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. Thus, if the field is intialized in the calling context, then such calls can be permitted. We added a simple refinement of the `[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 helper methods called from the constructor to initialize fields. For our checker to prove that all fields are initialized at the end of the constructor, it needs extra annotations on the helper method indicating which fields the method initializes. We added the following annotations:

```
[Inits]
[Inits("fieldnames")]
```

Both annotations on a parameter or receiver of type T imply that on entry to the method, the parameter has type $T^{\text{raw(S)}-}$, where S is the supertype of T. The first annotation additionally states that on exit, the parameter has type $T^{\text{raw(T)}-}$, *i.e.*, all fields declared in T have been initialized. 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-nullity 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 PC.
Perhaps surprisingly, we found that checking a simple property like non-nullity 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 checker 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:

```
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

```
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.

```
bool m(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.

```
// 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). In constructors of sealed classes after all fields are initialized, the checker knows that the object under construction 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.

	Total	MayBeNull	Raw	Annotated
Fields	922	38	0	2.6%
Parameters	2367	64	1	0.5%
Receivers	1581	-	1	$< 0.1\%$
Returns	1581	40	0	2.5%

In addition, the code contains 84 assertions of the form

```
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 the 226 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 our defaults are well chosen.

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 (Section 7.1), 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:

$$\textbf{this}.child == t \quad \&\& \quad t.parent == \textbf{this}$$

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 [18].

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

```
abstract object Visit(T arg1, object arg2);
```

was implemented in two incompatible contexts. One implementation expected non-null objects as `arg2` and produced non-null objects in return, whereas a second implementation accepted null as `arg2`, but would only produce null as a result, if `arg2` was actually null.
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 currently cannot handle is staged object initialization, where an object is only partially initialized by its constructor. 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 developing a generalization of our work on rawness to address this problem [4].

Properties with possibly-null values A C# *property* is a pair of methods, a *getter* and a *setter* 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, our checker cannot refine the type of a getter after its result has been tested against null. If the property is subsequently accessed again, the checker assumes again that it returns a possibly-null reference. To avoid false positives, we rewrote our code to bind 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

Because the largest complication of our type system and checker implementation relates to raw objects, it is instructive to consider a design alternative that would avoid such complications.
To avoid partially initialized objects, while keeping the flavor of constructors of C# and Java, 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 have 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.

This alternative design assumes that all objects can always be fully initialized at construction time, thereby simplifying the model, since no partially initialized objects ever arise. Although these simplifications would be desirable, they do not necessarily match up with programmer practice. We believe that instead, we need to generalize the idea of rawness, so that programmers can describe what state an object is in, even outside the dynamic scope of a constructor. This need arises from the use of staged initialization, which we think is a common and legitimate practice.

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 [8], 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 sound design.

The following category of related work has some form of non-null types and checking, but for languages without objects or inheritance subtyping—the main complications 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 [7] (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 [10] (a C-style language providing explicit region-based memory management), CCured [15] (a tool that compiles and type checks C programs under a safer type system), and Typed Assembly Language [14] (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 in object-oriented languages was mentioned in the introduction. None of these languages provides access to the object under construction, thereby avoiding the problem of having to deal with partially initialized objects [12, 11, 5].

Some null validations can be proven through other means, such as the presence of dominating accesses to the same object (see the Marmot paper [6]). Such techniques alone, however, cannot prove the kinds of invariants our system can establish. To obtain a modular analysis, the kinds of annotations we propose are needed.

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 Stata [19] and by Smith [17]. 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 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:

$$\begin{array}{ll} T^- <: T^+ & T^+ <: T^{\text{raw}+} \\ T^- <: T^{\text{raw}-} & T^{\text{raw}-} <: T^{\text{raw}+} \\ S^- <: T^- & S^{\text{raw}-} <: T^{\text{raw}-} \\ S^+ <: T^+ & S^{\text{raw}+} <: T^{\text{raw}+} \end{array}$$

and

$$\begin{array}{ll} X\,[]^- <: X\,[]^+ & X\,[]^+ <: X\,[]^{\text{raw}+} \\ X\,[]^- <: X\,[]^{\text{raw}-} & X\,[]^{\text{raw}-} <: X\,[]^{\text{raw}+} \\ X\,[]^- <: Y\,[]^- & X\,[]^{\text{raw}-} <: Y\,[]^{\text{raw}-} \\ X\,[]^+ <: Y\,[]^+ & X\,[]^{\text{raw}+} <: Y\,[]^{\text{raw}+} \end{array}$$

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 two 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 by throwing **this** (if the type of **this** is an exception type). This is dangerous because an exception handler may then expect to use the object as if it were fully initialized. 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

[1] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.

[2] ECMA. Standard ECMA-334: C# Language Specification, December 2002. Available on the web as `http://www.ecma-international.org/publications/files/ecma-st/Ecma-334.pdf`.

[3] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[4] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *Proceedings of the 1st International Workshop on Aliasing, Confinement and Ownership*, July 2003.

[5] Kathleen Fisher and John H. Reppy. The design of a class mechanism for Moby. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 34, number 5 in *SIGPLAN Notices*, pages 37–49. ACM, May 1999.

[6] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software-Practice and Experience*, 30(3), 2000.

[7] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, volume 31, number 5 in *SIGPLAN Notices*, pages 23–32. ACM, May 1996.

[8] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[10] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, June 2002.

[11] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual, preliminary version. Memo 88, Programming Methodology Group, MIT Laboratory for Computer Science, February 1995. Available on the web at `http://www.pmg.lcs.mit.edu/Theta.html`.

[12] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[14] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[15] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[16] Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 report, February 1999. Available on the web as `http://haskell.org/onlinereport`.

[17] Chris Smith. Java pointifications: Nullability constraints, June 2001. Available on the web as `http://cdsmith.twu.net/professional/java/pontifications/nonnull.html`.

[18] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the 14th European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer, March 2000.

[19] Raymie Stata. Improving the safety of Java, December 1995. Available on the web as `http://larch-www.lcs.mit.edu:8001/~raymie/Java/javachangessafety.html`.