Non-null types in an object-oriented language

Manuel Fähndrich and K. Rustan M. Leino

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

Abstract. Non-null types can detect certain null-related errors in object-oriented programs earlier and avoid other such errors altogether. This paper gives a proposal for retrofitting a language like C# or Java with non-null types. It addresses the complications that arise in constructors, where non-null fields may not yet have been initialized.

0 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 $\mathbf{null}$. When designing a program, programmers need to consider whether or not a value may be null, and often need to handle null differently than proper objects. Since such handling can be error prone, one may hope for language tools that inspect programs to enforce the programming discipline the programmer intended and to point 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 like C# [10] and Java [5] provide only one object type per declared class, and null is a value of every such object type. In this paper, we propose providing more than one type for every declared class, so that expressions that may yield null can be identified.

Type systems where one can distinguish special values like null from proper values exist. The tagged unions in the object-centered language CLU [9] are a good example, because CLU requires their use in order to introduce null in the first place. For any type T, one can declare a " T or null" type NT as follows:

$$NT = \mathbf{oneof}[\ None\colon \mathbf{null} \quad Value\colon T\ ]$$

where " $None$ " and " $Value$ " are user-defined tags, and where $\mathbf{null}$, in CLU, denotes a singleton type. This offers programmers a choice in using the *maybe-null type* NT or the *non-null type* T.

To construct an object in CLU, one provides an initial value for each of the object's fields (which, in CLU, are fields of the record that represents the object's abstract type). This ensures that every field has a value of its type before the newly constructed object is accessible by the program. The same guarantee is provided in the object-oriented languages Theta [8] and Moby [2], whose **make** statement and maker routines, respectively, allow such construction in the presence of subclassing. Functional languages such as ML [11] or Haskell [12] similarly provide access only to fully initialized data records. References in these languages are by default non-null and an explicit tagged union, akin to the one in CLU, is used to express possibly absent references.

We may wish that all object-oriented languages would provide features such as these for constructing objects, but not all do: the mainstream language Java gives access to the object being constructed (by **this**) *before* all field initializers have run.[0] In C# classes, field initializers are evaluated before **this** can be accessed, but C# value types disallow field initializers and provide an unavoidable default constructor. Being type-safe languages, C# and Java do ensure that fields have zero-equivalent values of their type before an object being constructed can be accessed, but that is not good enough for non-null types.

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 a limited way. Thus, our proposal accommodates many modern programming styles.

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

– Better interface documentation: Clients know when a method expects a non-null argument, and know when it promises a non-null return value.
– Better pin-pointed 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.
– Declared invariants: Object invariants such as fields holding non-null references can be declared and statically checked.
– Fewer runtime-checks: 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. The programmer controls when runtime checks are inserted.
– Fewer unexpected null reference exceptions: For example, the C# 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 1 introduces non-null types. Section 2 discusses the crux of the paper: how to establish object invariants. Sections 3 and 4 extend the proposal to array types and to C# value types. Finally, section 5 examines the impact on methods with call-by-reference parameters.

[0] Through a minor change in the language, Java could have avoided this problem [7].

1 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 will 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 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 superinterface 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 typecast. For example:

```
$T^-\ t = \textbf{new}\ T(\ldots);$     // allocate a non-null object
$T^+\ n = t;$                          // this direction is always allowed
$\ldots$
$t = (T^-)n;$                          // this direction requires a typecast to $T^-$
$\textbf{int}\ x = t.f;$               // the type of $t$ in $t.f$ must be a non-null type
```

For now, we require the typecasts from T^+ to T^- to be provided explicitly; we'll return to this issue later. Note that we have now removed null reference exceptions from the language, since all null violations now instead show up (earlier in the program) as typecast errors.

As the code snippet above shows, an application of $\textbf{new}\ T(\ldots)$ has type T^-, since the object constructed is always non-null.

Finally, 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*. In the context of our new types, we propose removing the “that is not null” part of this definition. That is, if e is null, then we propose that e **is** T^+ return true and that e **is** T^- return false. Furthermore, in C#, the expression e **as** T returns e if e **is** T, and returns null otherwise. Thus, 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. The construction of these compound values complicates the story a good deal. Let's look at object construction first, then array construction, and finally at value type construction.

2 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 — $\mathbf{this}.f$ may not have been initialized yet, where $\mathbf{this}$ denotes the object being constructed. So, a use of the value $\mathbf{this}.f$ 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 $\mathbf{this}$ during construction, the problem is not limited to cases where field f is accessed through the special keyword $\mathbf{this}$; if $\mathbf{this}$ 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^-.

We propose to solve this problem by introducing another family of types: for any class T (not interface), $T^{\mathrm{raw}-}$ denotes the *partially initialized* objects of class T or subclass thereof. More precisely, for any class T, $T^{\mathrm{raw}-}$ 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 $C^{\mathrm{raw}-}$. However, we require that expressions *assigned* to $c.f$ be of type T^-, even in the case that c is of type $C^{\mathrm{raw}-}$.

The above restrictions 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 maintain soundness, for it is possible to have two references to the same object o, one via x typed $S^{\mathrm{raw}-}$, the other via y typed S^-. The former could have been captured during the construction of the object, the latter after the construction was complete. 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 $y.f$ were T^-.

With the restrictions in place, objects evolve monotonically towards full initialization. This innovation will enable 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 to. That is, we require that every path through a constructor to a normal return include an assignment to every non-null field (modulo 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 $\mathbf{new}$, all of its non-null fields have non-null values. Hence, for any class T, $\mathbf{new}\ T(\ldots)$ has type T^-, not $T^{\mathrm{raw}-}$. In effect, the "last" constructor takes care of casting the object being constructed from type $T^{\mathrm{raw}-}$ to type T^-.

As one would expect, if S is declared to be a subclass of T, then $S^{\mathrm{raw}-}$ is a subtype of $T^{\mathrm{raw}-}$. Also, for any T, T^- is a subtype of $T^{\mathrm{raw}-}$.

For completeness, we also introduce a maybe-null type for partially initialized objects, written $T^{\mathrm{raw}+}$. If S is declared to be a subclass of T, then $S^{\mathrm{raw}+}$ is a subtype of $T^{\mathrm{raw}+}$. Furthermore, for any T, $T^{\mathrm{raw}-}$ is a subtype of $T^{\mathrm{raw}+}$, and T^{+} is a subtype of $T^{\mathrm{raw}+}$.

Since $\mathbf{this}$ is of type $T^{\mathrm{raw}-}$ in a constructor, any assignments of $\mathbf{this}$ to other variables can be done only if the other variable is of the appropriate type, namely a supertype of $T^{\mathrm{raw}-}$. For example, if $\mathbf{this}$ 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 when $\mathbf{this}$ is passed as a parameter to a method by virtue of that method being invoked on $\mathbf{this}$), 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 $T^{\mathrm{raw}-}$, then the method's formal receiver parameter must be of an appropriate partially-initialized type.

We allow typecasts of an expression of a partially-initialized type to a fully-initialized type. We propose that such a cast succeed if and only if the construction of the object has been completed, as measured by the "last" constructor having finished and the object having been returned by the $\mathbf{new}$ expression that prompted its construction. A straightforward implementation of this would use an extra bit per object, setting the bit when the construction of the object has been completed.[1]

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

3 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 co-variant array types in C# and Java work as expected in the presence of these new types.

But, 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:

$$\mathbf{new}\ T^{-}\,[n]$$

[1] An alternative would be to dynamically check that every non-null field has been initialized, which can be done by comparing those fields with null. Doing so, however, would not be able to guarantee the completion of other tasks performed in constructors which conceptually make the object be fully initialized.

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 $T^- \,[\,]^-$ (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 this would thus have the form:

$$\begin{array}{l} T^- \,[\,]^{\text{raw}-}\ aTmp = \ \mathbf{new}\ T^- \,[n]; \\ \ldots \qquad \text{// initialize the elements of } aTmp \\ T^- \,[\,]^- a = (T^- \,[\,]^-) aTmp; \end{array}$$

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.

4 Value types

The C# language supports value types via *struct* declarations. Structs are data records similar to objects, 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.

All structs in C# have a default constructor that initializes fields to their zero-equivalent default value. This default constructor cannot be overwritten.

We want to allow structs with invariants, that is, structs for which the default constructor is not sufficient because, for example, a field is declared as non-null, or a field is itself a struct with an invariant. To make the presentation easier, we differentiate structs with invariants from structs without invariants and call the former *istructs*.

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 S^- or possibly-null type S^+ for a struct S, since a struct is not a reference.

Since fields can now be non-null references or istructs, we have to extend our rule for accessing fields of partially initialized istructs and objects. 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^{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.[2]

The subtype relation on value types in C# only includes a boxing conversion between a value type S and the class root $\mathbf{object}$. Adapting this relation to our raw and non-null types yields the following subtype relations:

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

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

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 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 maybe-null types,

[2] In footnote 1, we argued that an object should be considered fully initialized when its "last" constructor has finished, not when all its non-null fields have been assigned to. That design requires an extra bit per object. For structs, however, it is not obvious that a bit could be added, because a struct value may be packed inside a legacy data structure. Thus, we have settled for the design that an istruct is fully initialized when all its non-null and istruct fields have been initialized.

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:

$T^- <: T^+$	$T^+ <: T^{\text{raw}+}$	$X[\,]^- <: X[\,]^+$	$X[\,]^+ <: X[\,]^{\text{raw}+}$
$T^- <: T^{\text{raw}-}$	$T^{\text{raw}-} <: T^{\text{raw}+}$	$X[\,]^- <: X[\,]^{\text{raw}-}$	$X[\,]^{\text{raw}-} <: X[\,]^{\text{raw}+}$
$S^- <: T^-$	$S^{\text{raw}-} <: T^{\text{raw}-}$	$X[\,]^- <: Y[\,]^-$	$X[\,]^{\text{raw}-} <: Y[\,]^{\text{raw}-}$
$S^+ <: T^+$	$S^{\text{raw}+} <: T^{\text{raw}+}$	$X[\,]^+ <: Y[\,]^+$	$X[\,]^{\text{raw}+} <: Y[\,]^{\text{raw}+}$

So far, we've said that narrowing typecasts, which may fail at runtime, require an explicit typecast expression in the program. That's how we would have liked to design these language features, possibly in conjunction with some static analysis that uses more than types to reduce the number of explicit typecasts needed (*cf.* [6]). However, in order to address the concern of backward compatibility, we can imagine that the narrowing typecasts that arise just because of our new taxonomy would be inserted implicitly at the points where they are needed.

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 [4] (a static checking tool whose annotation language provides a **non-null** modifier for variable declarations), Vault [0] (a language that aims to make programs safer by preventing certain kinds of resource-management programming errors), MrSpidey [3] (a tool that searches Scheme programs for car-of-an-atom errors, the functional-programming equivalent of null dereference errors), and LCLint [1] (a tool for checking various properties of C programs that also provides null and non-null annotations on references). Even with implicit typecasts, we believe our proposal would be a step toward this goal.

Because field initializers in C# are evaluated before **this** can be accessed, one can entertain a reduced proposal for adding non-null types to C#. In the reduced proposal, a class that declares a field of a non-null type must also provide an initializer for the field, a struct is not allowed to declare fields with non-null types, and an array is not allowed to have a non-null element type. Although less expressive and less flexible than our main proposal, the added benefit of this reduced proposal is that partially-initialized types are not 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 [14] and by Smith [13]. 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.

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, 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.

References

0. 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.
1. 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.
2. 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.
3. 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)*, pages 23–32. ACM SIGPLAN Notices 31(5), May 1996.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. To appear in the proceedings of PLDI'02, 2002.
5. James Gosling, Bill Joy, and Guy Steele. *The Java$^{\text{TM}}$ Language Specification*. Addison-Wesley, 1996.
6. K. Rustan M. Leino. Applications of extended static checking. In Patrick Cousot, editor, *Static Analysis: 8th International Symposium, SAS 2001*, volume 2126 of *Lecture Notes in Computer Science*, pages 185–193. Springer, July 2001.
7. K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center, January 1997.
8. 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`.
9. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

10. Microsoft. *Microsoft C# language specifications*, beta contents edition, 2001.
11. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
12. 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. On the web as `http://haskell.org/onlinereport`, February 1999.
13. Chris Smith. Java pointifications: Nullability constraints. On the web as `http://cdsmith.twu.net/professional/java/pontifications/nonnull.html`, June 2001.
14. Raymie Stata. Improving the safety of Java. On the web as `http://larch-www.lcs.mit.edu:8001/~raymie/Java/javachangessafety.html`, December 1995.