

Pluggable checking and inferencing of non-null types for Java

Torbjörn Ekman, Computing Laboratory, Oxford University, United Kingdom
Görel Hedin, Department of Computer Science, Lund University, Sweden

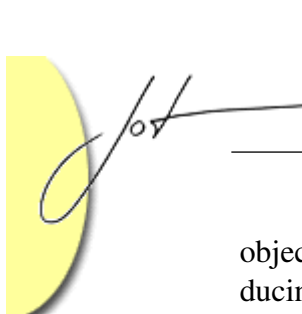
We have implemented a non-null type checker for Java and a new non-null inferencing algorithm for analyzing legacy code. The tools are modular extensions to the JastAdd extensible Java compiler, illustrating how pluggable type systems can be achieved. The resulting implementation is compact, less than 230 lines of code for the non-null checker and 460 for the inferencer. Non-null checking is a local analysis with little effect on compilation time. The inferencing algorithm is a whole-program analysis, yet it runs sufficiently fast for practical use, less than 10 seconds for 100.000 lines of code. We ran our inferencer on a large part of the JDK library, and could detect that around 70% of the dereferences, and around 24% of the method return values, were guaranteed to be non-null.

1 INTRODUCTION

Static typing allows for early detection of certain classes of errors and allows developers to document important aspects of their intent in the form of type signatures. Extending a given type system is often desirable, as language developers discover new useful ways of stating additional static properties of programs. There are also drawbacks in that different extensions might be desirable for different applications, and the language becomes more complex. To overcome these drawbacks, Bracha has suggested the notion of *pluggable types* where optional type systems can be plugged into a compiler [Bra04]. Bracha suggests the use of meta data, such as Java annotations, for expressing the new constructs, and that the annotations should have no effect on the dynamic semantics of the program.

In this paper, we show how Bracha-style pluggable type systems can be implemented in a compact and modular way, by extending the JastAdd Extensible Java Compiler [EH, EH07], using ReCRAGs (Rewritable Circular Reference Attributed Grammars) [EH04, MH03]. We illustrate our technique by the development of a pluggable non-null type checker. Furthermore, we suggest an algorithm to infer non-null types in legacy code, also implemented as a pluggable extension to the base compiler. The approach promises to allow similar type extensions at reasonable performance and implementation costs.

Non-null types are as available in some newer object-oriented languages like Spec# [BLS04] and Nice [Bon] as well as the next generation of Eiffel [Mey05]. The idea is to let the compiler detect that certain expressions will never have the value null. These expressions can be safely dereferenced without any risk of leading to null pointer exceptions at runtime. To help the compiler in this analysis, the source code can be annotated using modifiers on reference declarations. Fähndrich and Leino [FL03] showed how an



object-oriented language such as Java or C# can be extended with non-null types, introducing the notion of *raw* types to handle the intricacies of partially initialized objects. Extending Java to support non-null types using annotations is also suggested as part of a Java Specification Request (JSR-305).

When starting to use non-null annotations, a problem is that legacy code is not annotated. Experiments indicate that non-null references are much more common than possibly-null references [FL03, CJ07], and that non-null should therefore be the default for reference declarations. However, for legacy code, the reverse default rule must be used. The annotated new code must always assume null values from legacy code, and add extra checks against null when using such code in order to be safe from possible run-time null pointer exceptions.

We suggest an improved approach by inferring annotations in legacy code, e.g., the JDK, to get a safe conservative approximation of which references in the legacy code are always non-null. These inferred annotations can then be used by the explicitly annotated code to be able to safely use much of the legacy code without extra guarding null checks. The inference analysis over the legacy code is a whole-program analysis since it is necessary to take inheritance and overriding into account in order to obtain good approximations that are useful. I.e., a safe but uninteresting approximation would be to infer that all references in the library code are possibly null.

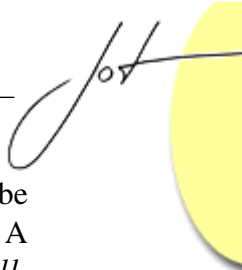
The rest of this paper is structured as follows. Sections 2 and 3 present background to non-null types and the JastAdd system used for the implementation. Section 4 defines a base language which is extended with non-null type checking in Section 5 and non-null inferring in Section 6. The approach is evaluated in Section 7 and related work is discussed in Section 8. Section 9 concludes the paper.

2 NON-NULL TYPES BACKGROUND

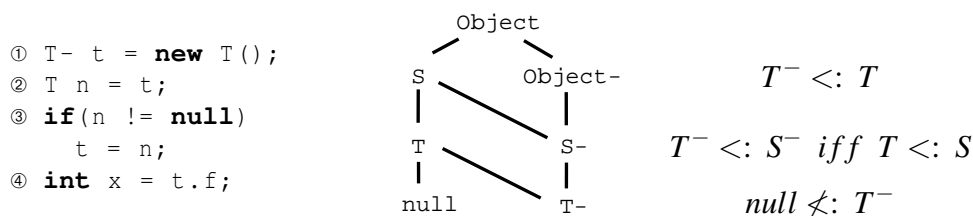
Our non-null types extension is based on the work on non-null types by Fähndrich and Leino [FL03]. Their approach differs from earlier work on non-null types by taking inheritance and object initialization into account.

The purpose of non-null types is to add the possibility to distinguish *non-null* references from *possibly-null* references in the type system. This enables the compiler to statically detect null-related errors at compile-time. The programmer already needs to consider whether a value may be null or not, and the special handling of null values is error prone. It is clearer to make this design explicit in the code, and making use of these type invariants in order to write simpler and safer code. In particular, conditionals guarding against null can safely be removed for non-null references, and the compiler can give warnings or errors if there are missing null guards for expressions that are possibly null.

Fähndrich and Leino split reference types into possibly-null and non-null types. The subtype relation is extended to include these new types. Consider the example to the left in the figure below. ① A newly created instance is clearly non-null and thus allowed



to be assigned to a non-null typed variable. ② A possibly-null typed variable may be assigned an expression typed by its non-null counterpart, i.e., T^- is a subtype of T . ③ A conditional statement that checks a possibly-null local variable or parameter against *null*, automatically casts it into its non-null counterpart, as long as it is not assigned a possibly-null value in that branch. ④ A qualified variable or method-name must be qualified by a value of a non-null type, otherwise a possible null pointer violation is reported.



The type hierarchy can be modeled by the diagram in the middle where a type T is a subtype of another type S if there is a path in the upward direction from T to S . The subtype relation is extended to include three new rules relating non-null types to possibly null types. These rules are shown on the right hand side in the figure above. The typing rules for Java state that all reference types are supertypes of the null type, and the third rule excludes this relation for non-null types.

The combination of non-null instance fields and object initialization complicates matters. Consider the listings in Figure 1. The left hand is actually equivalent to the right hand side from a code generation point of view. There is an implicit call to the super constructor and fields are not initialized until after that call is completed. The constructor in `A` is thus reached before `b` has been initialized and the virtual call to `print()` reaches the implementation in `B` that uses the `b` field prior to its initialization. The problem is that a partially initialized object is referenced by `this` within a constructor. A type based solution to this problem is to extend the type system with reference types for partially initialized objects. These types are called *raw* and when reading fields in raw objects we expect them to be possibly-null regardless of their annotations. Instance methods that are dispatched on a raw object, e.g., called from a constructor, can be annotated as being *raw* which implies that the type of `this` is raw in the method body. A thorough description of raw types is given in [FL03].

3 JASTADD BACKGROUND

The JastAdd compiler construction system combines object-orientation with declarative attributes and context-dependent rewrites to allow highly modular specifications [EH]. Inter-type declarations [KHH⁺01] allow for modular extension and declarative attributes enable composition of specifications. Behavior may be specified declaratively using the Rewritable Circular Reference Attributed Grammars (ReCRAGs) formalism [HM03, EH04, MH03] or imperatively using Java code.

```

class A {
    String a = "a";
    A() {
        print();
    }
    void print() {
        System.out.println(a);
    }
}
class B extends A {
    String b = "b";
    B() { }
    void print() {
        System.out.println(a+b);
    }
}

class A extends Object {
    String a;
    A() {
        super(); this.a = "a"; this.print();
    }
    void print() {
        System.out.println(this.a);
    }
}
class B extends A {
    String b;
    B() { super(); this.b = "b"; }
    void print() {
        System.out.println(this.a + this.b);
    }
}

```

Figure 1: Access to fields in partially initialized objects. The generated code is identical for both examples and the message "anull" is printed if B is instantiated.

Object oriented abstract grammars

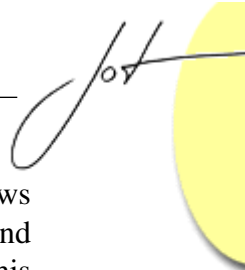
An abstract grammar models a class hierarchy from which classes are generated that are used as node types in the abstract syntax tree (AST). Consider the grammar in Figure 2 in Section 4. A class is generated for each production in the grammar, e.g., `TypeDecl`, `Expr`, and may inherit another production by adding a colon followed by the super production, e.g., `ClassDecl : TypeDecl`.

The right hand side of a production is a list of elements. The default name of an element is the same as its type unless it is explicitly named by prefixing the element with a name and a colon. E.g., the `TypeDecl` has an element named `Extends` which is of type `TypeName`. Elements enclosed in angle brackets are values, e.g., `<Name:String>` in `TypeDecl`, while other elements are tree nodes, e.g., `Extends:TypeName` and `BodyDecl` in `TypeDecl`. An element may be suffixed by a star to specify a list of zero or more elements, e.g., `BodyDecl*` in `TypeDecl`. Optional elements are enclosed in square brackets.

The system generates a constructor and accessor methods for value and tree elements. An accessor method is prefixed by `get`, e.g., `TypeName getExtends()`. List elements can be queried for the number of elements in the list using `int getNumBodyDecl()`. Elements are selected using an index to specify the appropriate child through `getBodyDecl(int index)`.

Declarative attributes

Attribute grammars [Knu68] have proven useful when describing context-sensitive information, and specifies behavior through attributes whose values are defined by equations. They integrate well with the object-oriented programming paradigm when extended with *reference attributes*, allowing an attribute to be a reference to another tree node [Hed00].



A *synthesized* attribute is similar to a virtual method without side-effects which allows for efficient evaluation using caching. Consider the grammar in Figure 2 in Section 4 and the task to determine whether an `Expr` node accesses a field named *name* or not. This can be implemented through the following synthesized attribute. Notice that the equation for `FieldName` overrides the default equation for its superclass `Expr`. Equations may be specified as either a side-effect free block returning the value or a single expression.

```
syn boolean Expr.isFieldName(String name);
eq Expr.isFieldName(String name) { return false; }
eq FieldName.isFieldName(String name) = getName().equals(name);
```

An *inherited* attribute propagates context downwards the AST. Consider the task to determine the enclosing `TypeDecl` for a `BodyDecl` node. A `TypeDecl` defines the enclosing context for all its `BodyDecls` through the following inherited attribute. Equations for inherited attributes are broadcast to an entire subtree to eliminate the need for copy rules. This subtree is explicitly selected using a child accessor, `getBodyDecl(int index)` in this case. The equation should thus be read as: define the value for the `enclosingTypeDecl()` attribute in the entire subtree whose root is the node returned by `getBodyDecl(int index)` in a `TypeDecl` node. The index can be used in the equation to define individual values for children in a list. The sample equation states that the value is **this**, i.e., a reference to the type declaration node defining the equation.

```
inh TypeDecl BodyDecl.enclosingTypeDecl();
eq TypeDecl.getBodyDecl(int index).enclosingTypeDecl() = this;
```

A common way to extend functionality in a modular fashion is to define new node types for new language constructs and to provide equations that override existing behavior in a superclass. This feature is extensively used in our compiler, but there is sometimes the need to refine an existing equation. This is similar to overriding but the new behavior affects the same class instead of a subclass. The following equation refines the behavior of `FieldName.isFieldName(String name)` previously defined in a module called `Base`. This new equation will replace the old definition. The old definition can still be accessed within the new equation by prefixing its name with the module in which it is defined, i.e., `Base`.

```
refine Base eq FieldName.isFieldName(String name) =
  name.equals("?") ? true : Base.FieldName.isFieldName(name);
```

Circular attributes

Circular Reference Attributed Grammars [MH03] allow iterative fixed-point computations to be expressed directly using recursive equations. Cyclic dependencies are allowed as long as there is a fixed-point that can be computed with a finite number of iterations. This is for example guaranteed if the value domain forms a finite height lattice and the functions are monotonic. Consider the code snippet below that detects cyclic class hierarchies. A `ClassDecl` optionally extends another class through explicit naming and that superclass is reached through the `type()` attribute in the extends clause. We define a boolean attribute `cyclicHierarchy()` which is true if, for instance, a class `A` **extends** `B`

and B **extends** A . In this situation the attribute is indeed circular since it depends on itself. We therefore need to declare the attribute **circular** and give a bottom value, in this case **true**, to be used in the fixed-point computation. This computation is guaranteed to terminate since the value domain is a boolean lattice and the equation has bottom value true and a meet operation that reaches top for a single false element.

```
ast ClassDecl : TypeDecl ::= <Name:String> [Extends:TypeName]
                                   BodyDecl* ImplicitTypeDecl:TypeDecl*;
syn boolean ClassDecl.cyclicHierachy() circular [true] =
    hasExtends() && getExtends().type().cyclicHierachy();
```

4 JAVADEMOTYPES BASE LANGUAGE

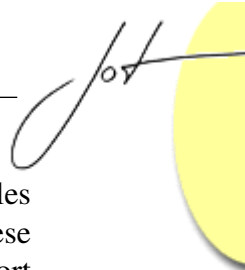
For demonstration purposes we define a small core language, JavaDemoTypes, which is then extended with non-null and raw types. The declarative specification language allows us to modularize our grammars freely, and the implementation can therefore share the same modularization as this presentation. JavaDemoTypes is implemented as separate modules for name binding, type checking, and an abstract grammar. There are three separate modules extending the base language with non-null types, raw types, and implicit casts through explicit null-checks. The language captures the essence of our implementation techniques while being small enough to fit in the paper. The same techniques are used to extend the JastAdd extensible Java compiler [EH07] and all implementations and extensions can be downloaded from the JastAdd web site [EH].

The main design principle is to represent the entire compilation process as an attributed AST. All language construct relations are represented as bindings between nodes in the tree. For instance, a type of an expression is represented by a reference to a type declaration which can then be used to look up members in name analysis, and to compute the subtype relation during type checking. There is thus no need for separate symbol tables, and this allows the extension mechanisms that apply to ASTs and attributes to be applied to the compilation data structures too. Code generation is then a one pass traversal of the tree which emits bytecode, reading attributes to gather non-local information.

The base JavaDemoTypes language contains class declarations with member fields, object instantiation, the **null** literal, and access to the current object through **this**. This allows us to write a very scaled-down version of Java and the language is then gradually extended with new concepts as they are introduced through language extensions.

Language structure

The AST structure is defined by the grammar in Figure 2. A type declaration inherits a single named type and has a list of body declarations. Each type also contains a list of implicit type declarations, e.g., arrays and non-null types based on that ground type. These implicit declarations are dynamically created when needed and added to the AST since all concepts must be explicit in the AST.



The only body declarations in the base language are member fields, but later examples add constructors and method declarations. Fields are initialized where declared and these assignments are checked using the subtype relation to ensure type safety. Basic support for names include type names, field names, and qualified names. Qualified names are used for nested types and member fields. Fields can also be qualified by `this`. A class instance expression is used to create an object and there is an explicit `null` literal. A singleton `NullDecl` is used to represent the type of the `null` literal.

```

ast abstract TypeDecl ::= <Name:String> Extends:TypeName
                          BodyDecl* ImplicitTypeDecl:TypeDecl*;
ast ClassDecl : TypeDecl;
ast NullDecl : TypeDecl;
ast BodyDecl ::= <Modifiers:String>;
ast FieldDecl : BodyDecl ::= TypeName <Name:String> Expr;
ast abstract Expr;
ast QualName : Expr ::= Left:Expr Right:Expr;
ast TypeName : Expr ::= <Name:String>;
ast FieldName : Expr ::= <Name:String>;
ast This : Expr;
ast ClassInstanceExpr : Expr ::= TypeName;
ast NullLiteral : Expr;

```

Figure 2: Abstract grammar for the JavaDemoTypes base language.

Name binding and type system API

For brevity, we only include the implementation of the base system that is concerned with types. The full implementation, including name analysis, is available at [EH]. Figure 3 shows the API for relevant parts of the base code that are not included in this paper. ① Each expression has a type which is represented by a reference to a type declaration. The following attributes deal with names: ② Each `FieldName` is bound to a `FieldDecl`, and ③ some names are qualified by an expression in which case there is a reference to that expression ④. Each `Expr` is enclosed in a `TypeDecl` which can be found through ⑤ and member fields of types can be bound using ⑥.

```

① syn TypeDecl Expr.type(); // each Expr has a type
② syn FieldDecl FieldName.decl() // names are bound to declarations
③ inh boolean Expr.isQualified(); // an Expr can be qualified ...
④ inh Expr Expr.qualifier(); // ... and then has a qualifier
⑤ inh TypeDecl Expr.enclosingType(); // an Expr has an enclosing type
⑥ syn FieldDecl TypeDecl.memberField(String n); // member field binding

```

Figure 3: The base language API that is used by the type extension.

Subtype computation for error checking

The *subtype relation* is used when typechecking object-oriented programs, and subsumption allows a subtype S of T to be used where a type T is expected. A typical example is assignment where an expression is assigned to a variable and the type of that expression must be a subtype of the declared variable type. Error checking in `JavaDemoTypes` is performed by a single traversal of the AST that collects errors and presents them to the user. Figure 4 shows sample error checking for `FieldDecl` where the initialization is type checked ① using the *type* and *subtype* attributes. In the full Java compiler the error method adds location information to the error message, e.g., filename and line number.

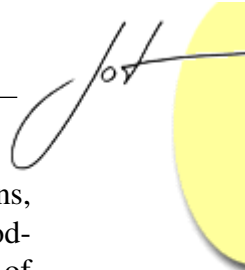
```
public void FieldDecl.errorCheck() {
①  if(!getExpr().type().subtype(getTypeName().type()))
    error("Field_" + getName() + "_assignment_error");
}
```

Figure 4: Type checking field initialization.

Extensible subtype tests

The key to enable modular type system extensions is the possibility to add new kinds of types and to extend the subtype relation to handle these new types. The previous typechecking example uses the boolean attribute *subtype* to determine whether two types are in the subtype relation or not. We use a straight-forward implementation of the subtype test for $S <: T$ that searches the direct supertypes of S transitively for T to determine if S and T are in a subtype relation. Since all type declarations are represented by nodes in the AST we can simply transitively follow the reference attribute that binds a type to its direct supertype. This works well in practice due to the automatic caching of previous subtype tests. New kinds of types can then be supported by adding node types to the AST representing these types. The type binding rules need to be modified to use these new types where appropriate, and the subtype attribute needs to be augmented to support these new kinds of types.

To make the subtype relation extensible we need an implementation that allows for modular specification of different rules for different combinations of types. These requirements are very similar to binary methods where the target method is selected based on both the receiver and its argument. Since Java only supports dispatch on the receiver we use the *double dispatch* pattern [Ing86] to emulate a binary method by first dispatching on the receiver and then performing a second dispatch on the argument. Figure 5 shows the implementation of the subtype relation for the base language using double dispatch. The first invocation, *subtype* ①, reduces the polymorphic receiver to a monomorphic one by the type dispatch inherent in method invocations. The target method, e.g., ②, reduces the polymorphic argument into a monomorphic one by a second dispatch on that argument. Notice that this second dispatch reverses the relation from *subtype* to *supertype* while also selecting a specific supertype computation based on the run-time receiver type,



e.g., ③ or ④. The traditional use of double dispatch does not allow modular extensions, but when combined with inter-type declarations the implementation can be done in a modular fashion. This enables us to provide specific equations for an arbitrary combination of type kinds in the subtype relation.

```

// double dispatch pattern to implement binary methods
① syn boolean TypeDecl.subtype(TypeDecl type);
eq ClassDecl.subtype(TypeDecl type) = type.superClassDecl(this);
② eq NullDecl.subtype(TypeDecl type) = type.superNullDecl(this);

// the subtype relation is reflexive and transitive
③ syn boolean TypeDecl.superClassDecl(ClassDecl type)
  = this == type || type.superclass().subtype(this);
// all types are supertypes of NullDecl ...
④ syn boolean TypeDecl.superNullDecl(NullDecl type) = true;

```

Figure 5: Base language subtype relation computation.

5 NON-NULL TYPES EXTENSION

This section presents a modular non-null type extension to JavaDemoTypes based on the type system by Fähndrich and Leino [FL03] introduced in Section 2. We add a non-null type declaration and extend the subtype relation to cope with this new kind of type. In Section 2, we used the suffix ‘-’ to indicate non-null types. In our implementation, we will instead use the Java annotation `@NonNull`, suggested in JSR-305, to annotate declarations that should not be assigned a null value. The extension is divided into the following tasks: add a representation of non-null types, extend the subtype relation to handle non-null types, refine type binding rules, and detect possible null pointer violations. Finally, we show how to extend the language to handle partially initialized objects using raw types.

Non-null type representation

Each type is represented by a type declaration node in the AST, e.g., a class declaration node is used to represent its corresponding type. Since non-null types are not declared explicitly we add them to the AST on demand when being used. Figure 6 shows attributes that bind a possibly-null declaration to its non-null counterpart ① and vice versa ②. The first time a non-null type is requested we build a `NonNullDecl` node that is attached to the AST as a child to its possibly-null counterpart. The `NonNullDecl` also delegate all member lookups to the possibly-null type ③.

```

ast NonNullDecl : TypeDecl ::= <Name:String> Extends:TypeName
                                BodyDecl* ImplicitTypeDecl:TypeDecl*;
    // link possibly-null type and the non-null counterpart
① inh TypeDecl NonNullDecl.possiblyNull();
    eq TypeDecl.getImplicitTypeDecl().possiblyNull() = this;
② syn TypeDecl TypeDecl.nonNull() {
    TypeDecl typeDecl = new NonNullDecl("@NonNull_" + getName(),
    new TypeName(getExtends().getName()), new List(), new List()
    );
    return addImplicitTypeDecl(typeDecl);
    }
    // delegate member field lookup to possibly-null type
③ eq NonNullDecl.memberField(String name) =
    possiblyNull().memberField(name);

```

Figure 6: Compute the non-null counterpart of a possibly-null type and vice versa.

Extend subtype relation to non-null types

The subtype extension to handle non-null types is straightforward when using double dispatch and reference attributes. Figure 7 adds the new subtype rules for non-null types. The equation for `superNullDecl(NullDecl type)` ① is for instance overridden for `NonNullDecl` to be false, effectively removing the `(NullDecl, NonNullDecl)` pair from the subtype relation.

```

    // subtype rules
eq NonNullDecl.subtype(TypeDecl type) = type.superNonNullDecl(this);
syn boolean TypeDecl.superNonNullDecl(NonNullDecl t) = false;

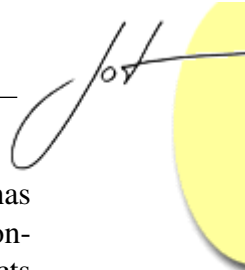
    // S- is a subtype of a T if S is a subtype of T
eq ClassDecl.superNonNullDecl(NonNullDecl type)
    = type.possiblyNull().subtype(this);
    // S- is a subtype of a T- if S is a subtype of T
eq NonNullDecl.superNonNullDecl(NonNullDecl type)
    = type.possiblyNull().subtype(possiblyNull());
    // a possibly-null type is not a subtype of a non-null
eq NonNullDecl.superClassDecl(ClassDecl type) = false;
    // null is not a subtype to a non-null type
① eq NonNullDecl.superNullDecl(NullDecl type) = false;

```

Figure 7: Modular extension of the subtype relation to include non-null types.

Refine type binding and error checking

With the new subtype relation in place, and attributes to bind a possibly-null type to its non-null counterpart, we need to refine a few type equations to use these new types and add checks for possible null-pointer violations. Figure 8 shows the refined equations for



both `FieldDecl` and `ClassInstanceExpr`. The type of a field is refined to be non-null if it has a `@NonNull` modifier ①. The base type is found through the `TypeName` binding and its non-null counterpart is referred using the `nonNull()` attribute. All newly instantiated objects are guaranteed to be non-null and therefore bound to a non-null type ②. We use these refined types to prevent possible null-pointer violations by detecting field reads that may dereference a null-pointer reference ③. If the qualifier to a field name is of a possibly-null type, there might be a null pointer violation at run-time and we instead raise a compile-time error. The extended subtype attribute ensures that a field declared non-null can not be assigned a possibly-null typed value.

```

// detect @NonNull modifier
① refine Base eq FieldDecl.type() = modifier("@NonNull") ?
  getTypeName().type().nonNull() : getTypeName().type();
// instantiated objects are non-null
② refine Base eq ClassInstanceExpr.type()
  = Base.ClassInstanceExpr.type().nonNull();
// detect attempt to dereference possibly-null types
refine BaseErrorCheck void FieldName.errorCheck() {
  BaseErrorCheck.FieldName.errorCheck();
③ if(isQualified() && qualifier().type().maybeNull())
  error("Qualifier_may_be_null");
}
syn boolean TypeDecl.maybeNull() = true;
eq NonNullDecl.maybeNull() = false;

```

Figure 8: Refine type binding and error checking when using non-null types.

Raw types

The extensions done so far assume that all fields have been assigned non-null values before they are accessed. However, as illustrated by Figure 1 in Section 2 this can not be guaranteed when allowing virtual dispatch in constructors. The problem is that `this` may reference a partially initialized object within a constructor in which the fields may not yet have been assigned. Fähndrich and Leino therefore introduce raw types that may hold references to such objects, in which all member fields are assumed to be possibly null [FL03]. Raw objects originate as `this` in constructors but they may escape constructors by being the receiver of a method invocation, an argument to a method invocation, or the right hand side in an assignment. Fields and parameters may therefore be annotated as being raw to allow such assignments, and methods annotated as raw may be invoked on partially initialized objects. We introduce raw type declarations (`ast RawDecl : TypeDecl`) and extend the subtype relation for raw types (using the same implementation technique as for non-null types) with the following rules: $A <: A^{raw}$ and $B^{raw} <: A^{raw}$ iff $B <: A$.

Both possibly-null and non-null types are thus subtypes of the corresponding raw type. This also means that raw types are not subtypes of `Object` but rather the `Objectraw` type. The `print()` method, in Figure 1 in Section 2, would have to be declared `@Raw` or

else the call from the constructor in A would result in a type error since A^{raw} is not a subtype of A . If we had dereferenced b in that context there would be a possible null pointer violation and we would have to guard that operation with an explicit comparison $b \neq \text{null}$. Figure 9 adds node types for constructors and methods to support programs where partially initialized objects are exposed.

```

ast ConstrDecl : BodyDecl ::= <Name:String> Block;
ast MethodDecl : BodyDecl ::= TypeName <Name:String> ParamDecl* Block;
ast ParamDecl ::= <Modifiers:String> TypeName <Name:String>;
ast ParamName : Expr ::= <Name:String>;
ast MethodInv : Expr ::= <Name:String> Arg:Expr*;
ast AssignExpr : Expr ::= LValue:Expr Expr;
    
```

Figure 9: Extend the base language with constructors and methods.

Detect partially initialized objects

Raw types force the actual type of `this` to change in certain contexts, e.g., within constructors and methods annotated as being raw. Figure 10 refines the implementation for the type of `this` in these contexts using the inherited attribute `thisType()` which defines the actual type for a particular context ①. Equation ② sets the type to be raw in the constructor body, and equation ③ takes annotations on methods into account. The Java language specification prescribes that fields in the same class can not be used in an initialization of another field unless they are declared before that field. This actually allows us to consider the type of `this` in the initialization of a field not to be raw ④. We finally need to add an additional check to verify that the receiver of a method call is a subtype of the declared method receiver ⑤.

```

refine Base eq This.type() = thisType();
① inh TypeDecl This.thisType();
② eq ConstrDecl.getBlock().thisType() = enclosingType().raw();
③ eq MethodDecl.getBlock().thisType() = modifier("@Raw") ?
    enclosingType().raw() : enclosingType().nonNull();
④ eq FieldDecl.getExpr().thisType() = enclosingType().nonNull();
refine BaseErrorCheck void MethodInv.errorCheck() {
    BaseErrorCheck.MethodInv.errorCheck();
    TypeDecl qualifierType = isQualified() ?
        qualifier().type() : thisType();
    if (!qualifierType.subtype(decl().methodThisType()))
⑤    error("Qualifier_not_compatible");
}
    
```

Figure 10: Compute the rawness state of `this` in a certain context.



Discussion

We have implemented a full non-null type extension to Java 1.4 using the same techniques as described in the previous sections. The main differences are that we need to extend error checking, e.g., overriding of methods with the new rules, and dealing with interfaces. Interfaces are added as a new kind of node type and equations are provided for that node type in the same way as the equations in Figure 5. Interfaces also affect the subtype relation in that each type may have several direct supertypes. The implementation checks multiple supertypes, but is otherwise very similar to the one for JavaDemoTypes.

We have used the same techniques to implement parameterized types in Java 5. The subtype relation is extended similar to the non-null extension, but dynamically built types do not delegate lookups to their base types. Instead we build method and field signatures where the use of type parameters have been replaced by type arguments for that particular parameterization. This allows the name binding framework to provide bindings to methods and fields that reflect the instantiated type parameters. The standard type checking can then be re-used even for type checking member access in parameterized types.

6 AUTOMATIC NON-NULL TYPE INFERENCE

The extended type system relies on annotations to separate possibly-null from non-null references. This section presents a technique to infer such annotations automatically in library code, e.g., the JDK. This allows users to start using non-null types in new code without having to guard all uses of libraries with null-checks. It also enables the user to gradually refactor legacy code to annotated code, moving the barrier between annotated and non-annotated code. The approach can also be used to detect candidates for possible null pointer errors, i.e., dereferencing references that are inferred to be possibly null.

Infer the non-null type property

The purpose of the type inference is to add as many non-null annotations as possible without violating the extended subtype rules. For each language construct that may be annotated we compute whether it is safe to replace its type T by its non-null counterpart T^- or not. It is considered safe if all subtype checks where the type is used still pass after the replacement. Consider the following code snippet for an element in a linked list:

```
class Element {
    Element next, pred;
    void remove() { pred.next = next; }
}
```

The assignment of the field `next` depends directly on itself and the computation of whether the possibly-null type can be replaced with a non-null type is thus circular. This naturally leads to iterative fix-point computations that are conveniently implemented using circular

attributes. The computation is guaranteed to terminate since the value domain of the non-null property forms a boolean lattice and the equation has bottom value `true` and a meet operation that reaches top for a single `false` element.

Figure 11 shows how to infer the non-null property for parameters and fields. Parameters are considered non-null unless they are assigned a possibly-null value ①. Fields are somewhat more complicated in that they must also be definitely assigned at the end of every constructor ②. However, this property needs to be computed by the Java compiler for blank final fields anyway, and is reused as is. Both parameters and fields depend on the non-null property of the expression values that they are assigned to and we therefore compute the non-null property of expressions as well. Newly created objects are always non-null ③ similar to equation ② in Figure 8. `FieldNames` need the rawness property to be computed to allow for safe refinement to non-null types ④. `ParamNames` that are not refined to non-null may still be considered non-null when guarded by a null comparison ⑤.

```

syn boolean ParamDecl.isNonNull() circular [true] {
  for(Iterator iter = assignedValues(); iter.hasNext(); )
  ①  if(!((Expr)iter.next()).isNonNull()) return false;
  return true;
}
syn boolean FieldDecl.isNonNull() circular [true] {
  for(Iterator iter=assignedValues(); iter.hasNext(); )
    if(!((Expr)iter.next()).isNonNull()) return false;
  for(Iterator iter = enclosingType().constructors(); iter.hasNext(); )
  ②  if(!((ConstrDecl)iter.next()).definitelyAssigns(this))
    return false;
  return true;
}
syn boolean Expr.isNonNull() circular [true];
③ eq ClassInstanceExpr.isNonNull() = true;
④ eq FieldName.isNonNull() =
  decl().isNonNull() && !qualifier().isRaw();
⑤ eq ParamName.isNonNull() =
  decl().isNonNull() || guardedByNullCheck(getName());

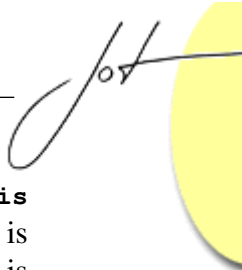
```

Figure 11: Infer the non-null property.

The legacy code differs from newly developed client code in that we allow possibly-null pointers to be dereferenced. There may thus be null-pointer violations in legacy code, but this is necessary to enable us to compile it without having to add manual annotations. However, the inferred non-null annotations are always safe and we can therefore use the legacy code without introducing new null-pointer violations in client code.

Infer raw types for partially initialized objects

The non-null inference checks whether a field is read from a partially initialized object, in which case the result must always be considered possibly null. We therefore need to compute the rawness property for expressions including `this`. Figure 12 shows the



implementation of these properties. A method is raw if any receiver is raw ① while **this** is raw in raw methods ② and in constructors ③. The computation for `thisTypeRaw()` is very similar to the implementation in Figure 10 and differs in that a boolean property is computed rather than an actual type, and also in the possibility of circularities.

```

syn boolean MethodDecl.isRaw() circular [false] {
  for(Iterator iter = receiverExprs(); iter.hasNext(); )
  ①   if((Expr)iter.next()).isRaw() return true;
  return false;
}
syn boolean Expr.isRaw() circular [false];
eq This.isRaw() = thisTypeRaw();
inh boolean This.thisTypeRaw() circular [false];
② eq MethodDecl.getBlock().thisTypeRaw() = isRaw();
③ eq ConstructorDecl.getBlock().thisTypeRaw() = true;

```

Figure 12: Infer the raw property.

7 EVALUATION

We have presented JavaDemoTypes and shown how to extend its type system with non-null types, raw types, and inference of these properties in legacy code. The same techniques are used in the JastAdd Extensible Java Compiler and we compare these implementations to validate that the techniques scale up to full languages. Figure 13 shows the module sizes for the non-null extension for JavaDemoTypes as well as Java. The Java extension includes additional checks that ensure that modifiers are only used in valid contexts, and also includes support for more fine-grained type rawness. Even when this functionality is included, the full extension is only 56% larger than the extension for JavaDemoTypes. As a comparison, the size of the complete Java compiler is close to 14.000 lines of code. The extension is thus only a small fraction of the full compiler.

Module	Demo	Java	Responsibility
NonNull	43	72	Add non-null type representation, extend subtype relation, detect non-null modifiers, and detect unsafe dereferences.
Raw	73	99	Add AST representation of raw types. Extend subtype relation. Include raw modifiers in type computations.
Guard	26	30	Check if a possibly-null variable is guarded by explicit comparison to null.
Check	n/a	20	Check for valid modifiers. Not included in JavaDemoTypes.
Total	142	221	

Figure 13: The size for each module in the non-null type extension for JavaDemoTypes and the corresponding size for the JastAdd Extensible Java Compiler. All sizes are number of lines of code excluding comments and whitespace.

We do a similar comparison for the type inference implementation in Figure 14. While still being fairly compact (< 3.3 % of the full compiler size), the refinement algorithm for full Java is roughly ten times larger than for JavaDemoTypes. The reason is that the type inference extension is not as deeply integrated in the compiler as the non-null type extension, but merely adds additional computations on top of existing analyses. The base compiler converts and propagates types according to the operations in an expression tree. The type extension reuses that code for new types as well, e.g., non-null types and raw types. When we infer the non-null property, on the other hand, we need to implement that propagation in all language constructs in the base Java compiler.

Module	Demo	Java	Responsibility
NonNull	19	112	Infer the non-null property.
Raw	14	134	Infer the raw property.
Guard	8	45	Detect variables guarded by explicit null check.
Output	n/a	30	Include inferred properties in pretty printer.
Def-Use	n/a	113	Bind definitions to uses, e.g., assignments.
Flatten	n/a	24	Conservatively merge methods with the same signature.
Total	41	458	

Figure 14: The size for each module in the type inference extension for JavaDemoTypes and the corresponding size for the JastAdd Extensible Java Compiler. All sizes are number of lines of code excluding comments and whitespace.

Inference performance

We have evaluated the type inference algorithm using a substantial part of the JDK standard class library as a benchmark and measured the inferred non-null property in that body of code. Roughly 100.000 source lines from the following packages in JDK 1.4.2 were included in the analysis: *java.lang*, *java.util*, *java.io*.

Our primary use of the inference algorithm is to automatically detect non-nullness properties of the APIs in legacy code. When we annotate new code we would otherwise have to assume all legacy code to return possibly-null types. We are thus mostly interested in the non-nullness property for return values. The *% Non-null return* column in Figure 15 shows the percentage of reference typed return values that the inference algorithm concludes are non null. A substantial part of the return values can thus be considered non null when used by annotated code instead of assumed possibly null.

However, additional insights can be gained from the analysis. We can use the inferred types of expressions to find possible null-pointer violations in the legacy code. We thus partition the code into blocks that are safe from null-pointer violations and blocks where null-pointer violations may occur. The *% Safe dereferences* column in Figure 15 shows the percentage of locations in the code where the language prescribes a possible null pointer exception, but where the inference algorithm inferred the non-nullness property. The *# Safe dereferences* column shows the number of dereferenced non-null references.



	Non-null return		Safe dereferences	
	%	#	%	#
Non-null + raw	24 %	259	71 %	8580
Non-null	24 %	252	69 %	8354

Figure 15: The result from non-null inference in terms of successful type refinements

Raw types allow for safe non-null declaration of fields that are accessed in partially initialized objects. Without raw types these declarations would have to be considered possibly-null. To see if we benefit from the analysis we compute the same property with raw types disabled and consider such fields possibly-null. The first line shows the results with raw types enabled, while the second line shows the result without raw types. The total number of non-null return values is slightly increased when using raw types. The total benefit may seem minor, but the increased cost of inferring raw types is very small.

Inference speed

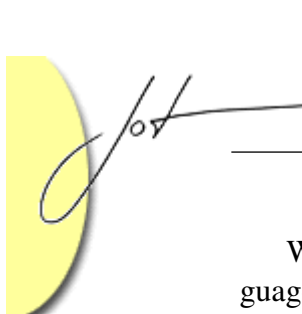
To evaluate the speed of our inference algorithm we compare the execution times for our inference implementation, the base compiler, and the standard javac compiler for reference. Figure 16 shows the execution times for our benchmark as described above. We included the time for javac as a reference to show that the JastAdd design to structure the compiler is reasonably fast while still providing excellent support for modularity and extensibility. The generated JastAdd based compiler is roughly three times slower than javac. The cost for checking non-null types is negligible, but the additional cost of inferring compared to normal compilation is roughly 50%. We conclude that the execution speed for the analysis is reasonably high even for fairly large programs.

	Non-Null + Raw	Non-Null	Base Compiler	javac
time	9.1 s	9.0 s	6.4 s	2.5 s

Figure 16: Execution speed with and without various type inference computations

8 RELATED WORK

This paper shows how the type system in a Java compiler can be extended in a modular fashion using JastAdd. Polyglot is an extensible Java front-end based on imperative programming using visitor patterns and relies on explicit passes to schedule computations [NCM03]. It has been extended with various type related concepts, e.g., parametrized types [BLM97] and predicate dispatch [Mil04]. The JastAdd system, in contrast, is based on declarative extensions through attribute grammars. To our knowledge, there is no implementation of non-null types in Polyglot.



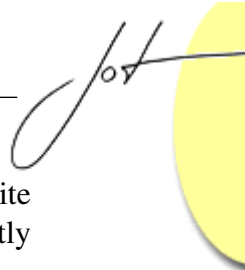
We base our type extension on the work on non-null types for object-oriented languages, as presented by Fähndrich and Leino [FL03]. Their approach differs from earlier work on non-null types by taking inheritance and object initialization into account and they did a prototype implementation for C#. We have implemented their type system as a modular extension for Java and extend their approach by adding a simple but effective inference algorithm in order to handle legacy code that does not have non-null annotations.

JavaCOP is a framework for adding pluggable type systems to Java [ANMM06]. The authors present an impressive number of type systems in this framework, including non-null types. However, it is not clear how they deal with partially initialized objects. The Annotation Processing Tool (apt) for Java is a set of reflective APIs to process program annotations. The system provides no API to method bodies and can thus not be used as is to implement a non-null type system. There are several static analysis systems that detect possible null pointer violations in Java code. FindBugs analyzes class files annotated with non-null annotations [HSP05]. A major difference is that our compiler extension may use any computation in the base compiler while these systems use domain-specific languages where only a restricted part of the compiler infrastructure is accessible. The Java Modeling Language (JML) is a specification language used to annotate implementations with behavioral interfaces [BCC⁺05]. It supports a non-null type system, where non-null is the default. This is a sensible choice as shown by Chalin and James that came to the conclusion that more than 2/3 of all references are non-null through a series of case studies [CJ07]. ESC/Java2 is a static checker that can be used to detect null-pointer violations [FLL⁺02]. It uses theorem provers to infer non-null properties, which potentially is a more exact approach than using a type system, but also much more expensive.

Type qualifiers [FFA99] allow types to be refined by adding a qualifier to a type name, e.g., non-null. This is a simple but highly useful form of subtyping. A framework is presented that extends the type rules in C to propagate the qualifiers through a program. There are publicly available systems with extensible type qualifiers without any automatic soundness proofs with implementations for C (CQUAL) and Java (JQUAL). Semantic Type Qualifiers [CMM05] provide an impressive framework and language to specify and automatically prove soundness of extended types by refining types using type qualifiers. An automatic theorem prover is used to prove soundness of various properties including non-null types for C. Our implementation is larger than the corresponding type qualifier definitions, but we can take other properties than propagation of types into account when refining types, e.g., definite assignment, null comparisons, etc.

9 CONCLUSIONS AND FUTURE WORK

We have shown how the JastAdd extensible Java compiler can be extended with non-null types. A type inference implementation that automatically infers non-null types in legacy code has also been presented, detecting around 24% of method returns and 70% of dereferences as non-null in JDK library code. Our modular implementation constitutes a strong case for using ReCRAGs and JastAdd as an infrastructure for pluggable type systems. The resulting implementation, including both the non-null checker and the non-null



inferencer, is less than 700 lines of JastAdd code, and is available at the JastAdd website [EH]. While being a whole-program analysis, the inferencer performance is sufficiently fast for practical use, running in less than 10 seconds on 100.000 lines of JDK code.

Future work includes some refinements of the implementation. Raw types can be split in more fine-grained types where objects are only raw up to a certain class. Constructors in ancestral superclasses are guaranteed to have completed, making sure that non-null fields in those inherited parts of the object are properly initialized. Currently, *raw upto* is used in the non-null checker, but not yet in the non-null inferencer. The implementation will also be extended to support elements in arrays as described by Fähndrich and Leino [FL03]. References to arrays are handled in the same way as other references but the elements of an array are always possibly null in our implementation.

References

- [ANMM06] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of OOPSLA'06*, pages 57–74. ACM Press, 2006.
- [BCC⁺05] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.
- [BLM97] J. A. Bank, B. Liskov, and A. C. Myers. Parameterized Types and Java. In *Proceedings of POPL'97*, pages 132–145. ACM Press, 1997.
- [BLS04] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.
- [Bon] D. Bonniot. Why programs written in Nice have less bugs. <http://nice.sourceforge.net/safety.html>.
- [Bra04] G. Bracha. Pluggable Type Systems. In *OOPSLA'04 workshop on revival of dynamic languages*, 2004.
- [CJ07] P. Chalin and P. James. Non-null References by Default in Java: Alleviating the Nullity Annotation Burden. In *Proceedings of ECOOP'07*, LNCS. Springer, 2007.
- [CMM05] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proceedings of PLDI'05*, pages 85–95. ACM Press, 2005.
- [EH] T. Ekman and G. Hedin. The JastAdd compiler compiler system. <http://jastadd.cs.lth.se>.
- [EH04] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP'04*, volume 3086 of *LNCS*. Springer, 2004.

- [EH07] T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of OOPSLA'07*, 2007.
- [FFA99] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of PLDI'99*, pages 192–203. ACM Press, 1999.
- [FL03] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of OOPSLA'03*, pages 302–312. ACM Press, 2003.
- [FLL⁺02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI '02*, pages 234–245. ACM Press, 2002.
- [Hed00] G. Hedin. Reference attribute grammars. In *Informatica (Slovenia)*, 24(3), 2000.
- [HM03] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.
- [HSP05] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proceedings of PASTE '05*, pages 13–19. ACM Press, 2005.
- [Ing86] D. H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of OOPSLA'86*, pages 347–349, 1986.
- [KHH⁺01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, volume 2072 of *LNCS*, pages 327–355. Springer, 2001.
- [Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).
- [Mey05] B. Meyer. Attached types and their application to three open problems of object-oriented programming. In *Proceedings of ECOOP'05*, volume 3586 of *LNCS*. Springer, 2005.
- [MH03] E. Magnusson and G. Hedin. Circular reference attributed grammars - their evaluation and applications. *ENTCS*, 82(3), 2003.
- [Mil04] T. Millstein. Practical Predicate Dispatch. In *Proceedings of OOPSLA'04*, pages 345–364. ACM Press, 2004.
- [NCM03] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC'03*, volume 2622 of *LNCS*, pages 138–152. Springer, 2003.



ABOUT THE AUTHORS



Torbjörn Ekman is a Research Fellow at the Computing Laboratory at Oxford University, United Kingdom. He received a PhD from Lund University in 2006. His research interests include extensible compilers, scriptable refactorings, domain-specific languages, and aspect oriented programming. He can be reached at torbjorn@comlab.ox.ac.uk



Görel Hedin is an Associate Professor of computer science at Lund University, Sweden. She received a PhD from Lund University in 1992. Her research interests include object-oriented languages and systems, compilation technology, domain-specific languages, and agile methodologies. She has served on the program committees of many international workshops and conferences including ECOOP, CC, and LDTA. She can be reached at gorel@cs.lth.se.