Contents

1 Introduction 2
   1.1 Advantages of type systems ................................................. 2
   1.2 Problems with types ......................................................... 2
   1.3 Optional type systems ....................................................... 3

2 JavaCOP 4
   2.1 Basic rules ........................................................................ 5
   2.2 Subtyping .......................................................................... 5
   2.3 Predicates ......................................................................... 6
   2.4 Additional syntax ............................................................... 6

3 Checker framework 7
   3.1 Nullness ........................................................................... 8
   3.2 Interning ........................................................................... 8
   3.3 Locks ............................................................................... 8
   3.4 Mutability ....................................................................... 9

4 TypePlug 9

5 Soft typing 10

6 Conclusion 11

References 11
1 Introduction

1.1 Advantages of type systems

Most of the currently available programming languages include some sort of type system (as an example of those that do not, consider Scheme or assembly). The language evolution proceeded in the direction of reducing the complexity of maintaining complex systems and types provide well-known benefits advancing this goal.

1. Compile-type error detection. Despite fundamental theoretical restrictions on static analysis, a great deal can be inferred from the source code of the program without actually executing it. Relatively simple to detect are common errors: typos in writing identifiers, passing invalid arguments to function an operators (e.g. trying to compute the sum of a Boolean value and a string) and other mistakes which pass the syntax checker but are discarded by the semantics. When encountered, they might just crash the program (possibly with the loss of data), mess with external resources (dangling database connections, opened file descriptors, locks), or, in the worst case, continue execution in the compromised state, leading to cascading failure. Thus early prevention of them makes systems more reliable and development process faster.

2. Performance advantages. Knowing the types of data used can help compiler to produce faster code. At the basic level, dynamically typing values is always slower, as all of them must be tagged with the type information at run-time, wrapped in some kind of generic object. The static type system provides guarantees at compile time about what kinds of values can a given identifier represent. This allows for more efficient memory usage (no need to hold type tags and other run-time-related overhead) and faster code (no overhead, e.g. need to check argument types before each function application; hardware may provide some support for specific cases etc). Even in dynamically typed languages similar ideas can be applied, for example, in recent work on typed arrays in JavaScript.

3. Software engineering. Untyped code can quickly become hard to follow and maintain, due to some information available only in run-time. Types provide conceptual framework for problem code, a way of structuring data and logic. Modularity, while not necessarily requiring rigid type system, benefits from its availability greatly, for example, the common code-to-interface idiom of separating module interface and its implementation is naturally based on subtyping. Type annotations can also serve as an executable (meaning checked by compiler) documentation. Most generally, types can encode constraints (with expressiveness dependent on complexity of type system) on data, e.g. for preventing certain kinds security holes (by having different types for sanitized and raw strings etc).

1.2 Problems with types

The Church-Turing thesis states that any computable function can be computed using a Turing machine or equivalent formalisms like mu-recursive functions and lambda calculus. Type theory development proceeds by building increasingly more sophisticated systems over lambda-calculus. Each TS, however, rejects some of the actually correct programs, narrowing the set of valid programs. For example, simple typing enforces strong normalization property - if the
term can be assigned a simple type, it must be reducible to its normal form - meaning that infinite loops are not possible. But non-termination is required for Turing-completeness, so by creating a type system for language we made it much less expressive in computational sense. This is indeed a corner case - real-life typing is not so restrictive, however any useful TS still accepts only a subset of legal programs.

Bracha[2] argues that in the presence of a strong type system programmers tend to over-rely on it, which can lead to unfortunate consequences. Though systems are usually proven to be sound, those proofs tend to rely on assumptions (otherwise the task of formalization is too complicated), and assumption can be discovered to be wrong (or the implementation is erroneous, as type checking in a modern language is a non-trivial task). Types are now used not only trivially, but to provide security and robustness guarantees, and when those assumptions fail, the operation of the whole system is compromised and behavior undefined.

There are some cases where mandatory typing causes problems. For example, consider the problem of object persistence. In nominal type system type equality and subtyping relations are defined explicitly - two structures are considered different by default even if they have the same data and behavior. Therefore the role of the class and its actual code are separated. Now suppose the implementation of class A was changed, while the interface preserved. The client code can still use A as before. However, if an object of type A was serialized before the change and needs to be restored in the new version, problems begin, as the nominal type system cannot support two classes with the same qualified name. In the structural systems, where the equality and subtyping are defined solely based on supported fields and operation, such problem does not arise, as the code using A relies only on its interface and will accept any structure supporting it (this idea is widely known as “duck typing”). Still, structural systems are not unconditionally better than nominal - latter are much more useful in practical programming. So, inclusion of the type system forces the choice between two sub-par alternatives.

1.3 Optional type systems

The main idea is that we can have all of typing advantages (except optimization) without having to put up with outline problems. This is achieved by replacing mandatory, built-in typing with optional, or pluggable, one. The optional types system satisfies two criteria:

1. It does not require any syntax support. The code should not change to provide type information, type checker has to operate on an untyped source. However, annotations can still be used to guide the system, similarly to the situation with type inference in modern languages, where explicitly stating types in not compulsory, but serves mostly as a documentation.

2. More importantly, the language semantics does not depend on the type system in use. This means that the result of the program execution is always the same, regardless whether the setting is purely untyped or includes a sophisticated type framework. Optional TS can only reject a program, not influence its meaning. This requirement is much more demanding than the previous one, as it forbids the use of quite a lot of idioms common in language design, for example:

(a) Direct access to public fields. When compiler has a complete information about an object’s class, accessing its public variables is essentially referencing the memory
at a certain offset. However, without type info access to a field requires a dynamic lookup, as the object accessed might be of the different type altogether. So, public field get transformed to public methods. The consequence of this is a minor decrease in performance.

(b) Static overloading (ad-hoc polymorphism). Which of the overloaded functions is call depends entirely on the type of passed arguments and on subtyping relation. Thus, to execute a function, the run-time must again have a type information available.

(c) All sorts of casts between types.

According to Bracha, such restrictions are still beneficial, as they promote the cleaner language design. Static typing is used to reduce complexity of the code, but an advanced type system is complex by itself, especially since completely orthogonal features tend to require a serious work to integrate into each other (for example, subtyping and universal quantification are easy additions to primitive typing, but, when used both in the same system, require much more complex typechecker with a serious theory behind it).

Traditionally, language and its type system are interdependent. With optional TS, the dependency is unidirectional, so the language becomes more modular, as the semantics and typing can be developed separately at a different pace. The simplest example of this is lambda-calculus. While type systems with different properties are developed, the semantics of untyped LC stays the same. In such conditions, type systems can be viewed as plugins, each performing some sort of static analysis, not necessarily in the realm of tradition TS. Currently many of such projects are language extensions, but if an appropriate integration support is provided, a programmer can just combine optional systems to cleanly perform a set of checks needed.

The ideas of Bracha were not developed to the full extent, especially consider the controversial call to dropping mandatory typing altogether. This report describes some of the projects and ideas that implement or support the pragmatic side of optional TS - their pluggability.

2 JavaCOP

JavaCOP[1, 5] (Constraints on programs) is a framework for building custom type systems for Java. It is implemented as an extension of OpenJDK compiler. Designers specify rules of the new type system, which then get translated and executed during compilation as an additional pass. In the later versions (developed since the framework was first described in 2006), data flow analysis and testing means were added.

JavaCOP employs domain-specific declarative language to express the logic of the new system. Because JavaCOP is embedded into the compiler, it has access to its internal structures, most importantly the abstract syntax tree of a module (framework, as the compiler itself, operates on module-by-module basis, not on the program as a whole - this ensures scalability, but makes analyzing certain cross-class interactions impossible). The AST is a structured representation of the program, in contains all semantically relevant elements of the source code (which can be though of as a flattened AST), suitable for analyzing and interpreting the program. JavaCOP represents the syntax tree using the hierarchy of classes corresponding to different elements of the language, e.g JCAssign for assignment of JCIf for conditional statement. Each node of a tree provides a set of methods for getting information about it. As the JavaCOP pass is performed after the Java typechecker, not only structure and context
information, but also inferred type of the node is a available, which is relied upon by many kinds of static analyzers.

2.1 Basic rules

Custom type system is described by the set of rules, which constraint AST nodes based on the available data. The basic rule structure can be seen in this code fragment:

```java
class checkNonNull(Assign a) {
    where(requiresNonNull(a.lhs)) {
        require(definitelyNonNull(a.rhs)) {
            error(a, "Possible null assignment to @NonNull");
        }
    }
}
```

This rule is a part of a typical testing system for PTS frameworks: allowing to declare a reference in the code as non-null, meaning it should always reference a real value. It begins with the keyword rule, name and so-called joinpoint, the type of AST node on which the rule operates. In this case, the node is an assignment, with access to the left-hand and right-hand sides (a.lhs and a.rhs respectively). During the JavaCOP pass, this rule will be called on every assignment encountered during the depth-first traversal of the AST. After the header, the body follows, which holds the actual constrains. In this case, two additional helper predicates are used: requiresNonNull(x) checks whether the node x has @NonNull annotation attached (in this particular type system programmer uses Java annotations to declare additional qualifiers on a variable), definitelyNonNull(x) fails if it is possible for x to be a null reference. where construct filters out unrelated nodes, applying constrains in its body only to those nodes, on which requiresNonNull succeeds. If it does, according to the require construct the system performs the definitelyNonNull check and throws an error if it fails. Any Boolean expression can be used in require. Body of a rule can contain multiple constrains, which are viewed in a conjunction.

2.2 Subtyping

All object-oriented languages are based on the notion of a subtype (superclass - child class relation), when the object of a subtype can be safely used when the object of a supertype is expected: e.g in method invocations, assignments and return types. This behavior is known in OOP terminology as polymorphism. Optional type systems may want to extend or constrain the subtype relation to specify the way how classes interact under new rules, both within the optionally typed program, and with outside packages (for example, can constrained class be a subtype of standard API class?). JavaCOP supports user-defined subtyping with another kind of rule. The previous code can be generalized to

```java
class checkNonNull(node <<= sym) {
    where(requiresNonNull(sym)) {
        require(definitelyNonNull(node)) {
            error(a, "Possible null assignment to @NonNull");
        }
    }
}
```
which differs only in the joinpoint, which now expresses that the rule in checked not on specific node types, but when the node node is cast to type sym (possibly the same as the type of node), i.e. where the subtyping is involved: for example, in

```java
@NotNull Double a = 5;
...
a = calculate();
```

`checkNonNull` will be called with `node` being the node for the call of `calculate` method, and `sym` being the `Double` class.

### 2.3 Predicates

As mentioned above, additional predicates can be defined as a code abstraction for rule definitions, used similarly to the functions in general programming languages, using the `declare` keyword:

```java
declare requiresNonNull(JCTree t) {
    require (t.holdsSymbol && t.getSymbol().hasAnnotation("NonNull"));
}
```

There can be multiple definitions for the same predicate. In this case, those work as a disjunction: as long as one of the bodies is satisfied, the predicate succeeds. The predicate `definitelyNotNull`, for example, can be defined by such node type case distinction:

```java
/*@ If the node is annotated @NotNull, it cannot be null */
declare definitelyNotNull(JCTree t) {
    require (requiresNonNull(t));
}

/*@ If the value assigned in non-null, the whole assignment expression is non-null */
declare definitelyNotNull(JCAssign a) {
    require (definitelyNotNull(a.rhs));
}

/*@ JCNewClass is a new operator result, which is always an object, never a null reference */
declare definitelyNonNull(JCNewClass n) {
    return true;
}
```

### 2.4 Additional syntax

Even though previous examples show that a language used is very clear, it still would not have much overhead when written in Java. However, there are powerful syntactic features available. AST analysis is often case-based, where the program is a switch-statement checking the structure of the tree. This can lead to an increasing amount of boilerplate and maintenance
complexity stemming from it. Therefore the language includes primitives for easy type checking and pattern matching. The expression $v \leftarrow e$ is true if $e$ has the same type as $v$, in which case $v$ is bound to the typecast value of $e$.

```java
require(JCIdent recv, String name;
    recv <- dref.selected && fname <- dref.name)
{}
```

This rule requires that the expression is of the form $x.f$, where $x$ is an identifier (JCIdent). If this condition holds, $x$ node is let-bound to $recv$ and $fname$ to $fname$.

Pattern matching allows to easily deconstruct values based on their structure. A pattern match is an expression $e \Rightarrow [pat]$, where $e$ is a node. The match succeeds if $e$ is of form specified by $pat$. Patterns are written as Java code, which must have the same structure as $e$:

```java
where(Tree typ, String name;
    decl => [@NonNull typ name (...)]
)
```

This pattern checks whether $decl$ represents a method annotated with @NonNull and, if so, binds its return type to $typ$ and name to $name$.

In many cases, we do not know the exact structure of the tree, but need to find some of the subnodes of a tree. To this end, JavaCOP includes `forall` and `exists` constructs. The first one has the same semantics as in Java, the second is more interesting and can be exemplified by the code

```java
declare callsAssertMethod(JCMethodDecl m) {
    exists(JCMethodInvocation a : m) {
        require(a.meth.owner.fullName.equals("org.junit.Assert"));
    }
}
```

which iterates over all method invocations in the body of a method declaration $m$, binding them to $a$.

## 3 Checker framework

A similar project to JavaCOP is the Checker framework, though there is difference in goals: JavaCOP target use base is type system creators, while Checker advertises itself mostly as an extensible set of ready-made pluggable type system refinements. Each of them provides a set of checks, which rely on Java annotations specified by the programmer. With those in place, TS can be enabled as a compiler annotation processor plugin (not an extension, as in JavaCOP case). The strong point of Checker is its integration with the existing toolchain, including IDEs.

Each checker consists of three part. The compiler interface implementation responds to predefined annotation processor hooks and report errors through compiler’s own error messaging mechanism. Next part is the visitor class, which traverses the AST of the module, provided by the Java Tree API and performs the type checking. Type factory, the last part, provides annotation data to the visitor; this abstraction is needed, for example, when the annotation can be inferred from the context or when additional flow analysis is needed. Developers specify type systems as Java code (though greatly simplified, as much of the boilerplate code can be avoided), not on the high declarative level as in JavaCOP, so the rest of this subsection will deal with some examples of included PTS.
3.1 Nullness

Nullness checker adds a single new qualifier to the reference types, with the same semantics as the aforementioned @NonNull-based PTS, dealing with the prevention of null pointer exceptions. It operates more proactively than its JavaCOP counterpart, issuing warnings not only when the @NonNull-declared value might become null, but also on every place where possibly null variable or field is dereferenced:

```java
Object possiblyNullObj;
@NonNull Object nnObj;

possiblyNullObj.toString(); // warning: possible null pointer exception
nnObj = possiblyNullObj; // error: assignment invalidates @NonNull qualifier
```

The checker also includes some inference in form of the static flow control analysis. For example, if the reference is checked to be null in this block it can be considered non-null before any invalidating statement. This allows to omit many of the obvious annotations, lessening the checking overhead on the developer.

3.2 Interning

Interning is a design pattern where there exists only one object for each value of the type. For example, two Integer objects may represent the same value 53, but as they are in different memory places, they are considered different; when interning is used, however, there can only be one Integer object with value 53. This way the object reference equality == can be used instead of more expensive equals method call.

The functionality of the Interning checker is to control that == operator is used only in proper places:

1. When the objects compared are instances of interned classes. As always, this property is declared using an annotation. Interned object are intended to be immutable.

2. When the control flow analysis reports that the objects of some class are created a bounded number of times, each one with a different state, such that they are pairwise distinct accordingly to possibly overridden equals method.

3. In special cases, when the interning can’t be applied to all objects of the class, but works in the local scope. For example, search for an object in an array of distinct-state objects may use plain equality.

3.3 Locks

In the Lock system, programmer can assign a lock to variable, with the meaning that this variable can only be accessed when the corresponding lock is acquired. While this cannot prevent all kinds of concurrency errors, it still eliminates a whole class of common ones. There are two annotations supporting this:

1. @GuardedBy assigns the lock to a field. This can be an explicit Lock instance or built-in monitor lock.
2. @Holding is used on a method to indicate that it can be invoked only when the caller is holding the specified lock.

```java
@GuardedBy("MyClass.myLock") Object x = ...;

x.toString(); // error: lock is not held

synchronized(Myclass.myLock) {
    x.toString(); // ok
}
```

3.4 Mutability

IGJ and Java type systems provide qualifiers that specify mutability constraints, for example (in IGJ case):

1. @Immutable: referenced object cannot be modified.
2. @Mutable: an opposite one.
3. @ReadOnly: object cannot be modified using this reference. It is still possible that is is aliases by some mutable reference.
4. @Assignable: annotated field can be modified regardless of the class/object immutability, for example, when it is used as cache.
5. @AssignsFields: permits only assignment to field, intended for use in constructor helper methods.

As in the interning system, some constrains can be inferred automatically (though they will mostly default to @Mutable).

4 TypePlug

TypePlug[4] is a pluggable TS framework for Squeak dialect of Smalltalk. Similarly to JavaCOP and Checker types are specified using annotations. However, TypePlug deals differently with the problem of interacting with outside code: only annotated code is typechecked. This limits the strength of guarantees the system provides, but is much more practical than other approaches. Still, like in JavaCOP, there is a support for external annotations and inference (not complete, just enough to stop error propagation).

A type system is specified by first defining the types and then providing methods for mapping Smalltalk objects to them. Each node in the AST can be annotated with a PTS-specific type information. This is different from JavaCOP and Checker, where only some nodes can be annotated - TypePlug allows to annotate any part of the tree. When such annotation is encountered, it is compared to a type the node is mapped to. As the Java frameworks, TypePlug requires programmer to specify subtype relation between defined types, which is
used in the same cases: return types, argument passing and so on. Also, to support type-checking algorithm, unification method is required, which creates a union of two types. The semantic of this operation is arbitrary, but it should be some sort of common supertype.

One type, Top, is shared by all pluggable systems, which is predefined to be a supertype of any other type. If a value has the type Top, it could actually have any type, so no information is known about it. Unless overridden, every unannotated node has the Top type. This is used to support checking partially annotated code.

As an example, the primitive variant of non-null system contains following rules:

\[
\begin{align*}
typeForLiteral: & \quad aValue \\
& \quad aValue ~\text{if}~\text{NotNil}: \quad [self ~\text{singleType}] \\
& \quad \text{ifNil}: \quad [self ~\text{topType}] \\

typeForSelfInClass: & \quad aClass \\
& \quad (UndefinedObject \quad \text{includesBehavior}: \quad aClass) \\
& \quad \text{ifTrue}: \quad [self ~\text{topType}] \\
& \quad \text{ifFalse}: \quad [self ~\text{singleType}] \\
\end{align*}
\]

The first method maps a literal to its type. There are only two type, singleType for non-null values and topType for Top. The code checks if the literal aValue is nil. The second method returns the type of a self reference. If self is used inside UndefinedObject class or its superclasses (UndefinedObject is a class of nil value), its type is Top, otherwise singleType.

Despite Smalltalk being completely dynamic language, it still may contain type information, for example, about class of an object. While the environment does not track this data in any sort of structured system, it may be of use to pluggable TS developers. Because of this, TypePlug includes a basic type system, which is always enabled and contains known type data for AST nodes. This shows the applicability of the framework - in the case studies the system is expanded into a fully-functional object-oriented one, supporting many features of contemporary OO languages, e.g. generics.

5 Soft typing

Soft typing[3] is proposed generalization of static and dynamic type systems that tries to combine advantages of both. Like in static typing, a checker is run before the program execution. Its role is firstly to try to check the type correctness where it is possible. After that, if the analyzer is uncertain about some parts of the code, it doesn't just abort execution, like in static case, but inserts dynamic type checks at relevant points. Then the modified program is executed. This approach has the benefits of both disciplines:

1. Like in dynamic languages, no syntactically program is rejected, when the type system is not sure if it is erroneous or not.

2. As in static TS, some errors will be detected before execution, providing guarantees about the code.

Even though run-time checks are inserted, they do not increase the execution time or modify the semantics. On the contrary, the program is optimized in one direction. Despite type system being dynamic, languages still need to check the types of the objects. For example, when accessing field \(x\) of object \(obj\), interpreter need to consult the dictionary of available
methods of `Object`. In softly typed language many of those checks can be dropped based on the result of static analysis. Thus, there is actually a speedup - another included advantage of static typing.

While the idea is clear, there are many obstacles to practical implementation. Authors outline two criteria for a soft type system:

1. Minimal text principle: TS should not rely on programmer-provided annotations.
2. Minimal failure principle: TS must be rich enough to be able to assign a type to any of the commonly used idioms.

While the first problem can be dealt with using an inference engine, the second dictates the design of the type system, which is expected to be more powerful than, for example, the one used in ML. First, there are many cases, where expression can be correctly evaluated to objects of different types:

```python
def pred(n):
    if n < 1:
        return None
    else:
        return n - 1
```

Here, `x` can either be a number or a `None` value. Static type system would reject the definition of `pred`, but it may be correct semantically. To assign a type to such an expression, we need union types. If the value is of type $T_1 + T_2$, it belongs to one of $T_1, T_2$. This is slightly different from the commonly used algebraic data types, which are also unions, but additionally add a tag to each value. The function `pred` has the type `Num -> Num + None`.

Another issue is recursive definitions. A classic example is function `self = λx.x(x)`. It is a valid term (identity function: `id = id(id)`), but is impossibly to type in most of the static type systems. The way to address the problem used by authors is adding a fixed point operator `fix` to the type definitions. Using it, `self` can be given type `fix.t.t -> t`

### 6 Conclusion

The current state of the art does not include any research projects actually rigidly conforming to Bracha’s initial proposal. However, much work has been done on providing support for developing custom pluggable type systems. JavaCOP provides a declarative way to specify AST constraints, which was tested by building proof-of-concept systems. Checker placed an emphasis not on providing a framework, but building practically useful static type systems, only a small subset of which is described in this review. Both of those tools operate on Java code, which already has a full-featured static TS. Other two sections deal with dynamic languages. TypePlug provides a toolset similar to Java frameworks and illustrates a difference between implementing pluggable TS on top of static and dynamic systems. Of all three, TypePlug is the closest to the PTS ideals, though it still requires annotations. The last section, while not directly describing the work in PTS field, is related, because it illustrates how difficult it is to preserve a dynamic semantics when adding a static analyzer.
References


