Pluggable Type Systems

Gilad Bracha

The Paradox of Type Systems

- Type systems help reliability and security by mechanically proving program properties
- Type systems hurt reliability and security by making things complex and brittle

Mandatory Typing

Well known advantages:

- Machine-checkable documentation
- Types provide conceptual framework
- Early error detection
- Performance advantages

Mandatory Typing

Disadvantages:

- Brittleness/Rigidity
- Lack of expressive power

Mandatory Typing

Disadvantages:

- Brittleness/Rigidity
- Lack of expressive power

Brittleness of Mandatory Typing

- Security/Robustness - as strong as the type system/the weakest link
- Persistence/Serialization
- Type systems for VM and language collide

Brittleness of Mandatory Typing

- Security/Robustness - as strong as the type system/the weakest link
- Persistence/Serialization
- Type systems for VM and language collide

How Mandatory Typing Undermines Security

- Once a mandatory type system is in place, people rely on it for:
  - Optimization
  - Security Guarantees
- If type system fails, behavior is completely undefined

Example: Class Loaders

Class loading becomes incredibly subtle (cf. Liang and Bracha, OOPSLA 98)

- Class loaders define name spaces for types
- JVM has nominal type system
- Inconsistent namespaces mean inconsistent types
- Failure to detect inconsistencies across class loaders leads to catastrophic failure (forgeable pointers, privacy violations etc.)

Example: Class Loaders

class A { C getC() { return new B().getC();}}

class B { C getC() { return new C();}}

- *A* and *B* defined in different, but overlapping namespaces N1 and N2. N1 and N2 agree on *B* but differ on *C*.
- One version of *C* may have a pointer as its first field, the other an int; or one may have a private field and the other may have a public one.
- Attacker may create suitable versions to suit their needs

Example: Class Loaders

Class loading based type spoofing never caused a real security breach, because other security layers protect against unauthorized class loader definition.

One may not always be so lucky.

How Mandatory Typing Undermines Security

Wait, type systems shouldn't fail! A good type system will be formally proven to be sound and complete

- Real systems tend to be too complex to formalize
  - Formalizations make simplifying assumptions
  - These assumptions tend to be wrong
- Implementations tend to have bugs

How Mandatory Typing Undermines Security

- Type Systems are subtle and hard
- ***Relying*** on them is dangerous

Brittleness of Mandatory Typing

- Security/Robustness - as strong as the type system/the weakest link
- Persistence/Serialization
- Type systems for VM and language collide

Persistence and Typing

Consider Serialization in mainstream languages

- Nominal typing forces serialization to separate objects from their behavior
- Versioning problems galore
- Exposes class internals, compiler implementation details

Persistence and Typing

Consider Serialization in mainstream languages

- Nominal typing forces serialization to separate objects from their behavior
- Versioning problems galore
- Exposes class internals, compiler implementation details

Nominal Typing Separates Objects from their Classes

- When serializing an object one might naturally serialize its class as well
- This guarantees that data and behavior match
- Class can change over time, but clients are ok as long as public API is preserved

Nominal Typing Separates Objects from their Classes

```
class Point { // initial version
   private int x, y;
   public int getX() { return x;}
   public int getY() {return y;}
   }
```

Nominal Typing Separates Objects from their Classes

```
class Point { // new version
  private double rho, theta;
  public int getX() { return cos(rho, theta);}
  public int getY() { return sin(rho, theta);}
  }
```

Nominal Typing Separates Objects from their Classes

- New version of point differs in format, size
- Should not be a problem for clients - public API unchanged
- Deserialization can create distinct classes named Point
- Works with dynamic or structural typing
- But ...

Nominal Typing Separates Objects from their Classes

- Nominal typing cannot tolerate two classes named Point!
- "Solution":
  - Serialize object together with the name of its class
  - Deserialization binds object to class of stored name

Persistence and Typing

Consider Serialization in mainstream languages

- Nominal typing forces serialization to separate objects from their behavior
- Versioning problems galore
- Exposes class internals, compiler implementation details

Persistence and Typing

- Persistence works well with structural typing; nominal typing does not
- Nominal typing suited to practical languages; structural typing problematic
- Mandatory typing forces a choice between two suboptimal options

Persistence and Typing

- Persistence bugs can undermine type system
- Undermining a mandatory type system leads to catastrophic failure

Brittleness of Mandatory Typing

- Security/Robustness - as strong as the type system/the weakest link
- Persistence/Serialization
- Type systems for VM and language collide

Type Systems Collide

Run-time and compile-time type systems may be misaligned

- Cases where Java source code will not verify
- Definite assignment rules clash with verifier inference algorithm
- Weird cases with try-finally, boolean expressions

Having our Cake and Eating it too

- Performance disadvantage is greatly overstated
- Importance of performance also overstated
- Other advantages of static types can be had without the downside
- Enter Pluggable, Optional Type Systems

Having our Cake and Eating it too

- Performance disadvantage is greatly overstated
- Importance of performance also overstated
- Other advantages of static types can be had without the downside
- Enter Pluggable, Optional Type Systems

Optional Typing

- How do I define optional typing
- Concrete example:Strongtalk
- Principled arguments for optional typing

Optional Typing

- How do I define optional typing
- Concrete example:Strongtalk
- Principled arguments for optional typing

Optional Type Systems

- Run-time semantics are independent of type system
- Type annotations are optional

Optional Type Systems

- Run-time semantics are independent of type system
- Type annotations are optional

Common Constructs Precluded by Optional Typing

- Public fields
- Class based encapsulation, i.e.

```
class C {
    private int secret;
    public int expose(C c) { return c.secret;}
}
```

- Type based overloading

```
draw(Cowboy c) ....
draw(Shape s) ....
```

Optional Typing

- How do I define optional typing
- Concrete example:Strongtalk
- Principled arguments for optional typing

Strongtalk

- An optional type system for Smalltalk
- Fastest Smalltalk ever, but does not rely on types for performance
- Very good fit for object oriented languages

Optional Typing

- How do I define optional typing
- Concrete example:Strongtalk
- Principled arguments for optional typing

Theoretical Justification

Closely related to theory of programming languages:
Formal calculi use pluggable typing all the time, e.g. :

- Evaluation rules of lambda calculus need not change to accommodate type system
- Type system only determines which programs are guaranteed not to "fail"

Language Evolution

Traditional type systems introduce bidirectional dependency:

- Type system depends on executable language
- Semantics of executable language depend on type system (e.g., casts, overloading, accessibility)

Language Evolution

Optional typing breaks dependency of executable language on type system

- Type system can evolve faster than language
- Programs that were untypeable in the past can be typechecked now, but run the same

Type Inference

- Type inference relates to type system as type system relates to executable language
- Inference naturally depends on type system but type system should not depend on type inference
- Counterexample : Hindley-Milner restricts polymorphic recursion

Type Inference

- Type inference has caused us a lot of grief in the JVM
- Verifier complexity -> security bugs, maintenance headaches, performance overhead

Having our Cake and Eating it too

- Performance disadvantage is greatly overstated
- Importance of performance also overstated
- Other advantages of static types can be had without the downside
- Enter Pluggable, Optional Type Systems

From Optional to Pluggable

We want various static analyses to coexist

- Traditional types, ownership types, tracing information flow

Make it easy to experiment with new tools.

How to integrate into the language?

Metadata

- Allows programmers to add user-defined annotations to ASTs
- Popularized by C#; Being added to Java

Types, Syntax & Metadata

- Types are just one kind of metadata
- Tools can choose which metadata to display
- Require ability to add metadata to every node of AST; Java and C# fall short
- Metadata might self-identify and choose its own syntax; is this a good idea?

Related Work

Variants of this idea have been around for quite a while, but not quite the same

- Optional Types in Common Lisp
- Soft Typing in Scheme (Cartwright/Fagan)
- Type system for Erlang (Marlow/Wadler)
- Cecil (Chambers/Litvinov)
- BabyJ type system for JavaScript (Anderson, Giannini)

Conclusions

- Mandatory typing causes significant engineering problems
- Mandatory typing actually undermines security
- The deeper in the system one requires types, the more acute the problems
- Types should be optional: runtime semantics must not depend on static type system
- Type systems should be pluggable: multiple type systems for different needs