Pluggable Type Systems

Gilad Bracha

October 27, 2004

Abstract

Traditionally, statically typed programming languages incorporate a built-in static type system. This system is typically *mandatory*: every legal program must successfully typecheck according to the rules of the type system. In contrast, *optional type systems* are neither syntactically nor semantically required, and have no effect on the dynamic semantics of the language. This in turn enables *pluggable type systems* [Bra03, Bra04], allowing multiple type systems to be used simultaneously and/or sequentially for various semantic analyses. We argue that pluggable types can provide most of the advantages of mandatory type systems without most of the drawbacks.

1 Mandatory Typing Considered Harmful

Mainstream programming languages incorporate *mandatory type systems.*

These type systems are widely perceived to enhance software reliability and security, by mechanically proving program properties. Paradoxically, such mandatory type systems inhibit software reliability and security, because they contribute to system complexity and brittleness.

The advantages of mandatory typing are widely recognized:

- Types provide a form of machine-checkable documentation.
- Types provide a conceptual framework for the programmer, that is extremely useful for program design, maintenance and understanding.
- Types support early error detection.
- Languages with mandatory types can be optimized based on type information, yielding significant performance advantages.

The disadvantages of mandatory typing are not as universally acknowledged. Types constrain the expressiveness of the language. While all major programming languages are Turing-complete, and so may be considered "equally expressive", the fact is that a mandatory type system greatly reduces the number of legal programs that can be expressed in a language.

Even more importantly, mandatory type systems make systems brittle. Once a mandatory type system is in place, the temptation to rely upon it is irresistable. It becomes a basis for optimizations and for security guarantees that fail ungracefully if the underlying assumptions of the type system do not hold. If the type system fails, system behavior is completely undefined.

Of course, we all know that such type systems should not fail. A good type system will be formally proven to be sound and complete. However, real systems tend to be too complex to formalize. Formalizations make simplifying assumptions. These assumptions tend to be make the formal model an inaccurate reflection of reality, and invalidate any reasoning based on the formal model. In addition, implementations tend to have bugs, so even the most carefully vetted type system may fail in practice.

2 Optional Type Systems

I believe we can have our cake and eat (most of) it too. Let's examine the issues in turn.

The performance disadvantage of object-oriented languages with mandatory types is greatly overstated. Research has shown that in most (though not all) applications highly optimized method dispatch (the dominant operation in object oriented programming) in a dynamically typed language is competitive with the table based dispatch used in statically typed languages. The importance of performance also overstated.

This is not universally true - especially not in numeric applications with large data vectors or matrices. Nevertheless, across a wide range of uses, we can forego the optimizations that rely on mandatory typing.

The other advantages of static types can be had without suffering the disadvantages of mandatory typing, through the use of an *optional type system.*

An optional type system is one that:

1. has no effect on the run-time semantics of the programming language, and
2. does not mandate type annotations in the syntax.

The former point is much more significant than the latter. It is in fact a very stringent requirement.

Note that an optional type system is the only way to extend a dynamically typed language with a static type system in a compatible way. The Strongtalk [BBG$^+$, BG93, Bra96] type system is an example of such an optional type system.

To appreciate how demanding the requirement that the dynamic semantics be independent of the type system is, consider some of the common constructs that are excluded by it:

1. Public fields. If you don't know in advance what type of object you are accessing, you cannot access its fields directly. You must look them up dynamically, essentially performing a method call.

2. Class based encapsulation. Code like

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

   is unacceptable, since there is no way to know when access to encapsulated members is allowed without a (prohibitively costly) dynamic check. Instead, object-based encapsulation, as in Smalltalk or Self, is used.

3. Static type based overloading. You cannot include both of the following method declarations in the same class, since one cannot distinguish between calls to one or the other.

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

The discipline of an optional type system prevents the language designer from introducing such constructs. This is a benefit, not a limitation. The above features are all things one is better off without anyway. The software engineering disadvantages of public fields or overloading should be well understood, and are beyond the scope of this paper. Class based encapsulation is very popular, but inherently less secure than object based encapsulation. While class based encapsulation relies critically on complex and fragile typecheckers and verifiers, object based encapsulation can be enforced by context-free grammar alone. Typecheckers are tricky and often buggy; parsers are one of the few areas of software design that are truly well understood. Those who ignore this when security concerns are critical risk nasty surprises. The austere discipline of optional typing forces more minimalist and simple language design, to the benefit of the system as a whole.

Optional typing has benefits for language evolution as well. Language design is known to be difficult, largely because every feature tends to effect every other feature, and so the complexity of the language grows exponentially with the number of features.

A useful type system will necessarily depend on the language being typed. Typically, languages introduce a dependency in the opposite direction: the semantics depend on the type system (e.g., casts, overloading, accessibility). By eliminating this latter dependency, we can make our languages more modular.

One result of such modularity is that the type system can evolve more quickly than the language. Programs that were untypeable in the past can be typechecked now, but are guaranteed to run the same way they did before.

Finally, note that the most common theoretical models of programming languages use optional typing. The evaluation rules for typed λ calculi do not depend on their type rules. These calculi almost always share the same evaluation rules - those of the untyped λ calculus. The type system serves only to reject certain programs whose derivations might (or might not) "fail".

3 Toward Pluggable Types

Once our runtime is independent of the type system, we can choose to treat type systems as plug-ins. We can have zero, one or many type systems, suited to differing purposes, all at the same time.

There are static type systems that deal with aliasing, ownership, with information flow, as well as traditional types systems. Indeed, a very wide range of static analyses can be cast as type systems.

It isn't practical to add all these analyzers into a language, which is why most of them languish as exotic research projects. However, if the language provides a general framework for plugging in type systems, the situation can change. Programmers might benefit from a host of analyses that integrate cleanly into their code.

This is easier said than done. How do we integrate multiple type systems into a language? This area needs research. One approach would leverage off of the growing support for user-defined annotations (often referred to as *metadata*) in programmming languages. Both C# and Java support some form of annotations added onto the program's abstract syntax tree. In this approach, one takes the view that types are just a kind of metadata. Tools can then choose which metadata to display and how to display and use it.

Challenges remain with this approach however. Mainstream languages do not allow metadata to be conveniently associated with every node of the AST. Many users are still reluctant to part with ASCII based editors, in which case the syntactic clutter of varying annotations can rapidly become overwhelming. Issues of namespace and version management need to be handled well. In short, it's a long way to a proven, robust and usable system that supports pluggable types, but the potential is clear.

4 Type Inference is to Type Checking as Type Checking is to Execution

Type inference is commonly cited as a key feature in "soft" type systems that try to bridge the gap between statically typed and dynamically typed languages.

I believe this notion is in fact a crucial misconception. Type inference should be optional, just like type checking. By mandating type inference as a requirement on a type system, designers of soft type systems fall into the very trap they seek to escape.

A mandatory type inference scheme restricts the expressiveness of the type system in much the same way as a mandatory type checker restricts the expressiveness of the executable language.

Making the design of the type system dependent on the type inference mechanism is gratuitous and makes the entire language design more brittle and complex - just as making the executable semantics dependent on the type checker does.

A classic example is the Hindley-Milner system used in ML. The use of inference leads to a type system that cannot support polymorphic recursion.

A better engineering approach is to implement type inference as a separate tool, available in the IDE. Programmers who find entering type annotations tiresome can invoke an inferencer on demand.

Once the rest of the system is not dependent on the inferencer, a wider range of inferencers can be implemented. The inferencer need not be sound - it can use heuristics that can fail on occasion. For example, it can infer types based on variable names. Multiple inferencers, utilizing different techniques, can coexist and be chosen by the developer as circumstances and utility dictate - in other words, we can have pluggable type inference.

5 Related Work

Over the years, there have been many efforts to combine the advantages of static and dynamic typing. This list is by no means complete. Common Lisp had a concept of optional typing, which meant something significantly different - type annotations could optionally be used to guide optimizations. These annotations were syntactically optional, but could affect program semantics. In particular, the annotations were not checked, and if they were wrong, programs could suffer unpredictable behavior. Soft typing [CF91] provided an optional type system for scheme, coupled with a type inference system that was not truly optional with respect to the type checker. Marlow and Wadler introduced a soft type system for Erlang [MW97]. Cecil [Lit03, Lit98] explored ideas similar to Strongtalk. An optional type system has been proposed for Javascript [AD02]. Finally, Meijer and Drayton [MD04] in this very workshop have explored this problem, though their approach remains based on mandatory type systems.

6 Conclusions

The dichotomy between statically typed and dynamically typed languages is false and counterproductive. The real dichotomy is between mandatory and optional type systems.

A new synthesis that combines most of the advantages of both static and dynamic typing is both possible and useful, based on the notions of optional and pluggable type systems. In summary:

1. Mandatory typing causes significant engineering problems
2. Mandatory typing actually undermines security
3. Types should be optional: runtime semantics must not depend on the static type system
4. Type systems should be pluggable: multiple type system for different needs

References

[AD02] Christopher Anderson and Sophia Drossopoulou. Babyj - from object based to class based programming via types, 2002.

[BBG+] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Strongtalk website. http://www.cs.ucsb.edu/projects/strongtalk/.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, September 1993.

[Bra96] Gilad Bracha. The Strongtalk type system for Smalltalk, September 1996. OOPSLA Workshop on Extending the Smalltalk Language.

[Bra03] Gilad Bracha. Pluggable types, March 2003. Colloquium at Aarhus University. Slides available at http://bracha.org/pluggable-types.pdf.

[Bra04] Gilad Bracha. Toward secure systems programming languages, March 2004. Keynote address. Slides available at http://www.acm.org/conferences/sac/sac2004/keynote_speakers.htm.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[Lit98] Vassily Litvinov. Contraint-based polymorphism in Cecil: towards a practical and static type system. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–411. ACM Press, 1998.

[Lit03] Vassily Litvinov. *Constraint-Bounded Polymorphism: an Expressive and Practical Type System for Object-Oriented Languages*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 2003.

[MD04] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, October 2004. Position paper at Revival of Dynamic Languages workshop at OOPSLA 2004.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, June 1997.