Table of Contents

What To Know Before Debating Type Systems

Recently, it was brought up on Proggit that Chris Smith's "What to Know Before Debating Type Systems" was no longer online. This is a really great article, and in an effort to make sure it survives, I've grabbed the archive.org cache and am 'reprinting' it here. If you're into programming languages, read this and level up!

EDIT: Chris has placed it back online. You can find it here.

Without further ado, "What to Know Before Debating Type Systems":

What To Know Before Debating Type Systems

I would be willing to place a bet that most computer programmers have, on multiple occasions, expressed an opinion about the desirability of certain kinds of type systems in programming languages. Contrary to popular conception, that's a great thing! Programmers who care about their tools are the same programmers who care about their work, so I hope the debate rages on.

There are a few common misconceptions, though, that confuse these discussions. This article runs through those I've encountered that obscure the most important parts of the debate. My goal is to build on a shared understanding of some of the basic issues, and help people get to the interesting parts more quickly.

Classifying Type Systems

Type systems are commonly classified by several words, of which the most common are "static," "dynamic," "strong," and "weak." In this section, I address the more common kinds of classification. Some are useful, and some are not.

Strong and Weak Typing

Probably the most common way type systems are classified is "strong" or "weak." This is unfortunate, since these words have nearly no meaning at all. It is, to a limited extent, possible to compare two languages with very similar type systems, and designate one as having the strong**er** of those two systems. Beyond that, the words mean nothing at all.

Therefore: I give the following general definitions for strong and weak typing, at least when used as absolutes:

Strong typing: A type system that I like and feel comfortable with

Weak typing: A type system that worries me, or makes me feel uncomfortable

What about when the phrase is used in a more limited sense? Then strong typing, depending on the speaker or author, may mean anything on the spectrum from "static" to "sound," both of which are defined below.

Static and Dynamic Types

This is very nearly the *only* common classification of type systems that has real meaning. As a matter of fact, it's significance is frequently under- estimated. I realize that may sound ridiculous; but this theme will recur throughout this article. Dynamic and static type systems are two *completely* different things, whose goals happen to partially overlap.

A static type system is a mechanism by which a compiler examines source code and assigns labels (called "types") to pieces of the syntax, and then uses them to infer something about the program's behavior. A dynamic type system is a mechanism by which a compiler generates code to keep track of the sort of data (coincidentally, also called its "type") used by the program. The use of the same word "type" in each of these two systems is, of course, not really entirely coincidental; yet it is best understood as having a sort of weak historical significance. Great confusion results from trying to find a world view in which "type" really means the same thing in both systems. It doesn't. The better way to approach the issue is to recognize that:

Much of the time, programmers are trying to solve the same problem with static and dynamic types.

Nevertheless, static types are not limited to problems solved by dynamic types.

Nor are dynamic types limited to problems that can be solved with static types.

At their core, these two techniques are not the same thing at all.

Observing the second of these four simple facts is a popular pass-time in some circles. Consider this set of presentation notes, with a rather complicated "the type system found my infinite loop" comment. From a theoretical perspective, preventing infinite loops is in a very deep sense the most basic possible thing you can do with static types! The simply-typed lambda calculus, on which all other type systems are based, proves that programs terminate in a finite amount of time. Indeed, the more interesting question is how to usefully extend the type system to be able to describe programs that don't terminate! Finding infinite loops, though, is not in the class of things most people associate with "types," so it's surprising. It is, indeed, provably impossible with dynamic types (that's called the halting problem; you've probably heard of it!). But it's nothing special for static types. Why? Because they are an entirely different thing from dynamic types.

The dichotomy between static and dynamic types is somewhat misleading. Most languages, even when they claim to be dynamically typed, have some static typing features. As far as I'm aware, *all* languages have some dynamic typing features. However, most languages can be characterized as choosing one or the other. Why? Because of the first of the four facts listed above: many of the problems solved by these features overlap, so building in strong versions of both provides little benefit, and significant cost.

Other Distinctions

There are many other ways to classify type systems. These are less common, but here are some of the more interesting ones:

Sound types. A sound type system is one that provides some kind of guarantee. It is a well-defined concept relating to static type systems, and has proof techniques and all those bells and whistles. Many modern type systems are sound; but older languages like C often do not have sound type systems by design; their type systems are just designed to give warnings for common errors. The concept of a sound type system can be imperfectly generalized to dynamic type systems as well, but the exact definition there may vary with usage.

Explicit/Implicit Types. When these terms are used, they refer to the extent to which a compiler will reason about the static types of parts of a program. All programming languages have some form of reasoning about types. Some have more than others. ML and Haskell have implicit types, in that no (or very few, depending on the language and extensions in use) type declarations are needed. Java and Ada have very explicit types, and one is constantly declaring the types of things. All of the above have (relatively, compared to C and C++, for example) strong static type systems.

The Lambda Cube. Various distinctions between static type systems are summarized with an abstraction called the "lambda cube." Its definition is beyond the scope of this article, but it basically looks at whether the system provides certain features: parametric types, dependent types, or type operators. Look here for more information.

Structural/Nominal Types. This distinction is generally applied to static types with subtyping. Structural typing means a type is assumed whenever it is possible to validly assume it. For example, a record with fields called x, y, and z might be automatically considered a subtype of one with fields x and y. With nominal typing, there would be no such assumed relationship unless it were declared somewhere.

Duck Typing. This is a word that's become popular recently. It refers to the dynamic type analogue of structural typing. It means that rather than checking a tag to see whether a value has the correct general type to be used in some way, the runtime system merely checks that it supports all of the operations performed on it. Those operations may be implemented differently by different types.

This is but a small sample, but this section is too long already.

Fallacies About Static and Dynamic Types

Many programmers approach the question of whether they prefer static or dynamic types by comparing some languages they know that use both techniques. This is a reasonable approach to most questions of preference. The problem, in this case, is that most programmers have limited experience, and haven't tried a lot of languages. For context, here, six or seven doesn't count as "a lot." On top of that, it requires more than a cursory glance to really see the benefit of these two very different styles of programming. Two interesting consequences of this are:

Many programmers have used very poor statically typed languages.

Many programmers have used dynamically typed languages very poorly.

This section, then, brings up some of the consequences of this limited experience: things many people assume about static or dynamic typing that just ain't so.

Fallacy: Static types imply type declarations

The thing most obvious about the type systems of Java, C, C++, Pascal, and many other widely-used "industry" languages is not that they are statically typed, but that they are explicitly typed. In other words, they require lots of type declarations. (In the world of less explicitly typed languages, where these declarations are optional, they

are often called "type annotations" instead. You may find me using that word.) This gets on a lot of people's nerves, and programmers often turn away from statically typed languages for this reason.

This has nothing to do with static types. The first statically typed languages were explicitly typed by necessity. However, type inference algorithms - techniques for looking at source code with no type declarations at all, and deciding what the types of its variables are - have existed for many years now. The ML language, which uses it, is among the older languages around today. Haskell, which improves on it, is now about 15 years old. Even C# is now adopting the idea, which will raise a lot of eyebrows (and undoubtedly give rise to claims of its being "weakly typed" -- see definition above). If one does not like type declarations, one is better off describing that accurately as not liking explicit types, rather than static types.

(This is not to say that type declarations are always bad; but in my experience, there are few situations in which I've wished to see them required. Type inference is generally a big win.)

Fallacy: Dynamically typed languages are weakly typed

The statement made at the beginning of this thread was that many programmers have used dynamically typed languages poorly. In particular, a lot of programmers coming from C often treat dynamically typed languages in a manner similar to what made sense for C prior to ANSI function prototypes. Specifically, this means adding lots of comments, long variable names, and so forth to obssessively track the "type" information of variables and functions.

Doing this prevents a programmer from realizing the benefits of dynamic typing. It's like buying a new car, but refusing to drive any faster than a bicycle. The car is horrible; you can't get up the mountain trails, and it requires gasoline on top of everything else. Indeed, a car is a pretty lousy excuse for a bicycle! Similarly, dynamically typed languages are pretty lousy excuses for statically typed languages.

The trick is to compare dynamically typed languages when used in ways that fit in with their design and goals. Dynamically typed languages have all sorts of mechanisms to fail immediately and clearly if there is a runtime error, with diagnostics that show you exactly how it happened. If you program with the same level of paranoia appropriate to C - where a simple bug may cause a day of debugging - you will find that it's tough, and you won't be actually using your tools.

(As a side comment, and certainly a more controversial one, the converse is equally true; it doesn't make sense to do the same kinds of exhaustive unit testing in Haskell as you'd do in Ruby or Smalltalk. It's a waste of time. It's interesting to note that the whole TDD movement comes from people who are working in dynamically typed languages... I'm not saying that unit testing is a bad idea with static types; only that it's entirely appropriate to scale it back a little.)

Fallacy: Static types imply upfront design or waterfall methods

Some statically typed languages are also designed to enforce someone's idea of a good development process. Specifically, they often require or encourage that you specify the whole interface to something in one place, and then go write the code. This can be annoying if one is writing code that evolves over time or trying out ideas. It sometimes means changing things in several different places in order to make one tweak. The worst form of this I'm aware of (though done mainly for pragmatic reasons rather than ideological ones) is C and C++ header files. Pascal has similar aims, and requires that all variables for a procedure or function be declared in one section at the top. Though few other languages enforce this separation in quite the same way or make it so hard to avoid, many do encourage it.

It is absolutely true that these language restrictions can get in the way of software development practices that are rapidly gaining acceptance, including agile methodologies. It's also true that they have nothing to do with static typing. There is nothing in the core ideas of static type systems that has anything to do with separating interface from implementation, declaring all variables in advance, or any of these other organizational restrictions. They are sometimes carry-overs from times when it was considered normal for programmers to cater to the needs of their compilers. They are sometimes ideologically based decisions. They are not static types.

If one doesn't want a language deciding how they should go about designing their code, it would be clearer to say so. Expressing this as a dislike for static typing confuses the issue.

This fallacy is often stated in different terms: "I like to do exploratory programming" is the popular phrase. The idea is that since everyone knows statically typed languages make you do everything up front, they aren't as good for trying out some code and seeing what it's like. Common tools for exploratory programming include the REPL (read-eval-print loop), which is basically an interpreter that accepts statements in the language a line at a time, evaluates them, and tells you the result. These tools are quite useful, and they exist for many languages, both statically and dynamically typed. They don't exist (or at least are not widely used) for Java, C, or C++, which perpetuates the unfortunate myth that they only work in dynamically typed languages. There may be advantages

for dynamic typing in exploratory programming (in fact, there certainly are *some* advantages, anyway), but it's up to someone to explain what they are, rather than just to imply the lack of appropriate tools or language organization.

Fallacy: Dynamically typed languages provide no way to find bugs

A common argument leveled at dynamically typed languages is that failures will occur for the customer, rather than the developer. The problem with this argument is that it very rarely occurs in reality, so it's not very convincing. Programs written in dynamically typed languages don't have far higher defect rates than programs written in languages like C++ and Java.

One can debate the reasons for this, and there are good arguments to be had there. One reason is that the average skill level of programmers who know Ruby is higher than those who know Java, for example. One reason is that C++ and Java have relatively poor static type systems. Another reason, though, is testing. As mentioned in the aside above, the whole unit testing movement basically came out of dynamically typed languages. It has some definite disadvantages over the guarantees provided by static types, but it also has some advantages; static type systems can't check nearly as many properties of code as testing can. Ignoring this fact when talking to someone who really knows Ruby will basically get you ignored in turn.

Fallacy: Static types imply longer code

This fallacy is closely associated with the one above about type declarations. Type declarations are the reason many people associated static types with a lot of code. However, there's another side to this. Static types often allow one to write much more concise code!

This may seem like a surprising claim, but there's a good reason. Types carry information, and that information can be used to resolve things later on and prevent programmers from needing to write duplicate code. This doesn't show up often in simple examples, but a really excellent case is found in the Haskell standard library's `Data.Map` module. This module implements a balanced binary search tree, and it contains a function whose type signature looks like this:

> *lookup :: (Monad m, Ord k) => k -> Map k a -> m a*

This is a magical function. It says that I can look something up in a `Map` and get back the result. Simple enough, but here's the trick: what do I do if the result isn't there? Common answers might include returning a special "nothing" value, or aborting the current computation and going to an error handler, or even terminating the whole program. The function above does any of the above! Here's how I compare the result against a special nothing value:

> *case (lookup bobBarker employees) of Nothing -> hire bobBarker Just salary -> pay bobBarker salary*

How does Haskell know that I want to choose the option of getting back `Nothing` when the value doesn't exist, rather than raising some other kind of error? It's because I wrote code afterward to compare the result against `Nothing`! If I had written code that didn't immediately handle the problem but was called from somewhere that handled errors three levels up the stack, then `lookup` would have failed that way instead, and I'd be able to write seven or eight consecutive lookup statements and compute something with the results without having to check for `Nothing` all the time. This completely dodges the very serious "exception versus return value" debate in handling failures in many other languages. This debate has no answer. Return values are great if you want to check them now; exceptions are great if you want to handle them several levels up. This code simply goes along with whatever you write the code to do.

The details of this example are specific to Haskell, but similar examples can be constructed in many statically typed languages. There is no evidence that code in ML or Haskell is any longer than equivalent code in Python or Ruby. This is a good thing to remember before stating, as if it were obviously true, that statically typed languages require more code. It's not obvious, and I doubt if it's true.

Benefits of Static Types

My experience is that the biggest problems in the static/dynamic typing debate occur in failing to understand the issues and potential of static types. The next two sections, then, are devoted to explaining this position in detail. This section works upward from the pragmatic perspective, while the next develops it into its full form.

There are a number of commonly cited advantages for static typing. I am going to list them in order from *least* to *most* significant. (This helps the general structure of working up to the important stuff.)

Performance

Performance is the gigantic red herring of all type system debates. The knowledge of the compiler in a statically typed language can be used in a number of ways, and improving performance is one of them. It's one of the least important, though, and one of the least interesting.

For most computing environments, performance is the problem of two decades ago. Last decade's problem was already different, and this decades problems are at least 20 years advanced beyond performance being the main driver of technology decisions. We have new problems, and performance is not the place to waste time.

(On the other hand, there are a few environments where performance still matters. Languages in use there are rarely dynamically typed, but I'm not interested enough in them to care much. If you do, maybe this is your corner of the type system debate.)

Documentation

If, indeed, performance is irrelevant, what does one look to next? One answer is documentation. Documentation is an important aspect of software, and static typing can help.

Why? Because documentation isn't just about comments. It's about everything that helps people understand software. Static type systems build ideas that help explain a system and what it does. The capture information about the inputs and outputs of various functions and modules. This is exactly the set of information needed in documentation. Clearly, if all of this information is written in comments, there is a pretty good chance it will eventually become out of date. If this information is written in identifier names, it will be nearly impossible to fit it all in. It turns out that type information is a very nice place to keep this information.

That's the boring view. As everyone knows, though, it's better to have self- documenting code than code that needs a lot of comments (even if it has them!). Conveniently enough, most languages with interesting static type systems have type inference, which is directly analogous to self-documenting code. Information about the correct way to use a piece of code is extracted from the code itself (i.e., it's self-documenting), but then verified and presented in a convenient format. It's documentation that doesn't need to be maintained or even written, but is available on demand even without reading the source code.

Tools and Analysis

Things get way more interesting than documentation, though. Documentation is writing for human beings, who are actually pretty good at understanding code anyway. It's great that the static type system can help, but it doesn't do anything fundamentally new.

Fundamentally new things happen when type systems help computer programs to understand code. Perhaps I need to explain myself here. After all, a wise man (Martin Fowler, IIRC) one said:

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*

I don't disagree with Martin Fowler, but we have different definitions of *understand* in mind. Getting a computer to follow code step by step is easy. Getting a computer to analyze it and answer more complex questions about it is a different thing entirely, and it is very hard.

We often want our development tools to understand code. This is a big deal. I'll turn back to Martin Fowler, who points this out as well.

Correctness

Ultimately, though, the justification for static typing has to come back to writing correct code. Correctness, of course, is just the program doing "what you want."

This is a really tough problem; perhaps the toughest of all. The theory of computation has a result called Rice's Theorem, which essentially says this: Given an arbitrary program written in a general purpose programming language, it is impossible to write a computer program that determines anything about the program's output. If I'm teaching an intro to programming class and assign my students to write "hello world", I can't program a grader to determine if they did so or not. There will be some programs for which the answer is easy; if the program never makes any I/O calls, then the answer is no. If the program consists of a single print statement, it's easy to check if the answer is yes. However, there will be some complicated programs for which my grader can never figure out the answer. (A minor but important technical detail: one can't run the program and wait for it to finish, because the program might never finish!) This is true of any statement about programs, including some more interesting ones like "does this program ever finish?" or "does this program violate my security rules?"

Given that we can't actually check the correctness of a program, there are two approaches that help us make approximations:

Testing: establishes upper bounds on correctness

Proof: establishes lower bounds on correctness

Of course, we care far more about lower bounds than upper bounds. The problem with proofs, though, is the same as the problem with documentation. Proving correctness is easy only somewhat insanely difficult when you have a static body of code to prove things about. When the code is being maintained by three programmers and changing seven times per day, maintaining the correctness proofs falls behind. Static typing here plays exactly the same role as it does with documentation. If (and this is a big if) you can get your proofs of correctness to follow a certain form that can be reproduced by machine, the computer itself can be the prover, and let you know if the change you just made breaks the proof of correctness. The "certain form" is called structural induction (over the syntax of the code), and the prover is called a type checker.

An important point here is that static typing does not preclude proving correctness in the traditional way, nor testing the program. It is a technique to handle those cases in which testing might be guaranteed to succeed so they don't need testing; and similarly, to provide a basis from which the effort of manual proof can be saved for those truly challenging areas in which it is necessary.

Dynamic Typing Returns

Certainly dynamic typing has answers to this. Dynamically typed languages can sometimes perform rather well (see Dylan), sometimes have great tools (see Smalltalk), and I'm sure they occasionally have good documentation as well, though the hunt for an example is too much for me right now. These are not knock-down arguments for static typing, but they are worth being aware of.

The correctness case is particularly enlightening. Just as static types strengthened our proofs of correctness by making them easier and automatic, dynamic typing improves testing by making it easier and more effective. It simply makes the code fail more spectacularly. I find it amusing when novice programmers believe their main job is preventing programs from crashing. I imagine this spectacular failure argument wouldn't be so appealing to such a programmer. More experienced programmers realize that correct code is great, code that crashes could use improvement, but incorrect code that doesn't crash is a horrible nightmare.

It is through testing, then, that dynamically typed languages establish correctness. Recall that testing establishes only upper bounds on correctness. (Dijkstra said it best: "Program testing can be used to show the presence of bugs, but never to show their absence.") The hope is that if one tries hard enough and still fails to show the presence of bugs, then their absence becomes more likely. If one can't seem to prove any better upper bound, then perhaps the correctness really is 100%. Indeed, there is probably at some correlation in that direction.

What is a Type?

This is as good a point as any to step back and ask the fundamental question: what is a type? I've already mentioned that I think there are two answers. One answer is for static types, and the other is for dynamic types. I am considering the question for static types.

It is dangerous to answer this question too quickly. It is dangerous because we risk excluding some things as types, and missing their "type" nature because we never look for it. Indeed, the definition of a type that I will eventually give is extremely broad.

Problems with Common Definitions

One common saying, quoted often in an attempt to reconcile static and dynamic typing, goes something like this: Statically typed languages assign types to variables, while dynamically typed languages assign types to values. Of course, this doesn't actually define types, but it is already clearly and obviously wrong. One could fix it, to some extent, by saying "statically typed languages assign types to expressions, ..." Even so, the implication that these types are fundamentally the same thing as the dynamic version is quite misleading.

What is a type, then? When a typical programmer is asked that question, they may have several answers. Perhaps a type is just a set of possible values. Perhaps it is a set of operations (a very structural-type-ish view, to be sure). There could be arguments in favor of each of these. One might make a list: integers, real numbers, dates, times, and strings, and so on. Ultimately, though, the problem is that these are all symptoms rather than definitions. Why is a type a set of values? It's because one of the things we want to prove about our program is that it doesn't calculate the square roots of a string. Why is a type a set of operations? It's because one of the things we want to know is whether our program attempts to perform an impossible operation.

Let's take a look at another thing we often want to know: does our web application stick data from the client into SQL queries without escaping special characters first? If this is what we want to know, then these becomes types. This article by Tom Moertel builds this on top of Haskell's type system. So far, it looks like a valid definition of "type" is as follows: something we want to know.

A Type System

Clearly that's not a satisfactory definition of a type. There are plenty of things we want to know that types can't tell us. We want to know whether our program is correct, but I already said that types provide conservative lower bounds on correctness, and don't eliminate the need for testing or manual proof. What makes a type a type, then? The other missing component is that a type is part of a type system.

Benjamin Pierce's book Types and Programming Languages is far ans away the best place to read up on the nitty gritty details of static type systems, at least if you are academically inclined. I'll quote his definition.

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

This is a complex definition, but the key ideas are as follows:

syntactic method .. by classifying phrases: A type system is necessarily tied to the syntax of the language. It is a set of rules for working bottom up from small to large phrases of the language until you reach the result.

proving the absence of certain program behaviors: This is the goal. There is no list of "certain" behaviors, though. The word just means that for any specific type system, there will be a list of things that it proves. What it proves is left wide open. (Later on in the text: "... each type system comes with a definition of the behaviors it aims to prevent.")

tractable: This just means that the type system finishes in a reasonable period of time. Without wanting to put words in anyone's mouth, I think it's safe to say most people would agree that it's a mistake to include this in the definition of a type system. Some languages even have undecidable type systems. Nevertheless, it is certainly a common goal; one doesn't expect the compiler to take two years to type-check a program, even if the program will run for two years.

The remainder of the definition is mainly unimportant. The "kinds of values they compute" is basically meaningless unless we know what kinds we might choose from, and the answer is any kind at all.

An example looks something like that. Given the expression `5 + 3`, a type checker may look at 5 and infer that it's an integer. It may look at 3 and infer it's an integer. It may then look at the + operator, and know that when + is applied to two integers, the result is an integer. Thus it's proven the absence of program behaviors (such as adding an integer to a string) by working up from the basic elements of program syntax.

Examples of Unusual Type Systems

That was a pretty boring example, and one that plays right into a trap: thinking of "type" as meaning the same thing it does in a dynamic type system. Here are some more interesting problems being solved with static types.

http://wiki.di.uminho.pt/twiki/pub/Personal/Xana/WebHome/report.pdf. Uses types to ensure that the correct kinds of data are gotten out of a relational database. Via the type system, the compiler ends up understanding how to work with concepts like functional dependencies and normal forms, and can statically prove levels of normalization.

http://www.cs.bu.edu/~hwxi/academic/papers/pldi98.pdf. Uses an extension to ML's type system to prove that arrays are never accessed out of bounds. This is an unusually hard problem to solve without making the languages that solve it unusable, but it's a popular one to work on.

http://www.cis.upenn.edu/~stevez/papers/LZ06a.pdf. This is great. This example uses Haskell's type system to let someone define a security policy for a Haskell program, in Haskell, and then proves that the program properly implements that security. If a programmer gets security wrong, the compiler will complain rather than opening up a potential security bug in the system.

http://www.brics.dk/RS/01/16/BRICS-RS-01-16.pdf. Just in case you thought type systems only solved easy problems, this bit of Haskell gets the type system to prove two central theorems about the simply typed lambda calculus, a branch of computation theory!

The point of these examples is to point out that type systems can solve all sorts of programming problems. For each of these type systems, concepts of types are created that represent the ideas needed to accomplish this particular task with the type system. Some problems solved by static type systems look nothing like the intuitive idea of a type. A buggy security check isn't normally considered a type error, but only because not many people use languages with type systems that solve that problem.

To reiterate the point above, it's important to understand how limiting it is to insist, as many people do, that the dynamic typing definition of a "type" is applied to static typing as well. One would miss the chance to solve several real-world problems mentioned above.

The True Meaning of Type

So what is a type? The only true definition is this: a type is a label used by a type system to prove some property of the program's behavior. If the type checker can assign types to the whole program, then it succeeds in its proof; otherwise it fails and points out why it failed. This is a definition, then, but it doesn't tell us anything of fundamental importance. Some further exploration leads us to insight about the fundamental trade-offs involved in using a static type checker.

If you were looking at things the right way, your ears may have perked up a few sections back, when I said that Rice's Theorem says we can't determine anything about the output of a program. Static type systems prove properties of code, but it almost appears that Rice's Theorem means we can't prove anything of interest with a computer. If true, that would be an ironclad argument against static type systems. Of course, it's not true. However, it is very nearly true. What Rice's Theorem says is that we can't determine anything. (Often the word "decide" is used; they mean the same thing here.) It didn't say we can't prove anything. It's an important distinction!

What this distinction means is that a static type system is a conservative estimate. If it accepts a program, then we know the program has the properties proven by that type checker. If it fails... then we don't know anything. Possibly the program doesn't have that property, or possibly the type checker just doesn't know how to prove it. Furthermore, there is an ironclad mathematical proof that a type checker of any interest at all is *always* conservative. Building a type checker that doesn't reject any correct programs isn't just difficult; it's impossible.

That, then, is the trade-off. We get assurance that the program is correct (in the properties checked by this type checker), but in turn we must reject some interesting programs. To continue the pattern, this is the diametric opposite of testing. With testing, we are assured that we'll never fail a correct program. The trade-off is that for any program with an infinite number of possible inputs (in other words, any interesting program), a test suite may still accept programs that are not correct - even in just those properties that are tested.

Framing the Interesting Debate

That last paragraph summarizes the interesting part of the debate between static and dynamic typing. The battleground on which this is fought out is framed by eight questions, four for each side:

1. For what interesting properties of programs can we build static type systems?
2. How close can we bring those type systems to the unattainable ideal of never rejecting a correct program?
3. How easy can it be made to program in a language with such a static type system?
4. What is the cost associated with accidentally rejecting a correct computer program?
5. For what interesting properties of programs can we build test suites via dynamic typing?
6. How close can we bring those test suites to the unattainable ideal of never accepting a broken program?
7. How easy can it be made to write and execute test suites for programs?
8. What is the cost associated with accidentally accepting an incorrect computer program?

If you knew the answer to those eight questions, you could tell us all, once and for all, where and how we ought to use static and dynamic typing for our programming tasks.