The Programming Languages Enthusiast

BY MICHAEL HICKS | AUGUST 5, 2014 · 7:15 AM

What is type safety?

In response to my previous post defining memory safety (for C), one commenter suggested it would be nice to have a post explaining type safety. Type safety is pretty well understood, but it's still not something you can easily pin down. In particular, when someone says, "Java is a type-safe language," what do they mean, exactly? Are all type-safe languages "the same" in some way? What is type safety getting you, for particular languages, and in general?

In fact, what type safety means depends on language type system's definition. In the simplest case, type safety ensures that program behaviors are well defined. More generally, as I discuss in this post, a language's type system can be a powerful tool for reasoning about the correctness and security of its programs, and as such the development of novel type systems is a rich area of research.

Basic Type Safety

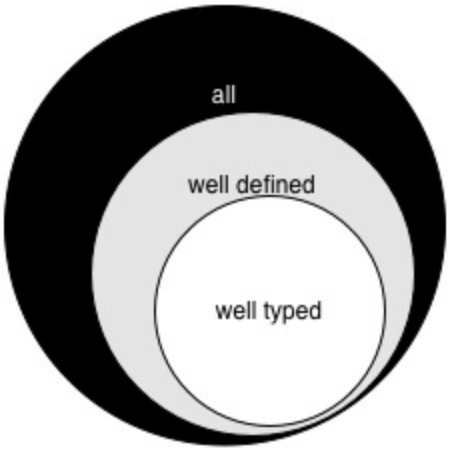
An intuitive notion of type safety is pithily summarized by the phrase, "***Well typed*** *programs* ***cannot go wrong***." This phrase was coined by Robin Milner in his 1978 paper, A Theory of Type Polymorphism in Programming. Let's deconstruct this phrase and define its parts, considering the second part first.

Going wrong

Programming languages are defined by their *syntax* — what programs you're allowed to write down — and *semantics* — what those programs mean. A problem all languages face is that there are many programs that are syntactically valid but are semantically problematic. A classic English example is Chomsky's "Colorless green ideals sleep furiously"—perfectly syntactically correct, but meaningless. A example in the OCaml programming language is **1 + "foo"**; according to the language's semantics, such a program has no meaning. Another example is **{ char buf[4]; buf[4] = 'x' }** in C: the write to index 4 is outside the declared bounds of the buffer, and the language specification deems this action to be undefined, i.e., meaningless. If we were to run such meaningless programs, we could say that they would *go wrong*.

Well typed ⟹ Cannot go wrong

In a type-safe language, the language's *type system* is a particular way of ensuring that only the "right" (non-wrong) programs go through. In particular, we say a program (or program phase) is *well typed* if the type system deems it acceptable, and type safety ensures that well typed programs never go wrong: they will have a (well defined) meaning. The following picture visualizes what's going on.



In a type safe language, well typed programs are a subset of the well defined programs, which are themselves a subset of all possible (syntactically valid) programs.

Which languages are type safe?

Let's now consider whether some popular languages are type safe. We will see that for different languages, type safety might mean different things.

C and C++: *not* type safe. C's standard type system does not rule out programs that the standard (and common practice) considers meaningless, e.g., programs that write off the end of a buffer.[ref] C is also not memory safe; in effect, the undefined behaviors that memory safety rules out are a subset of the undefined behaviors ruled out by type safety.[/ref] So, for C, well typed programs can go wrong. C++ is (morally) a superset of C, and so it inherits C's lack of type safety.

Java, C#: type safe (probably). While it is quite difficult to ascertain whether a full-blown language implementation is type safe (e.g., an early version of Java generics was buggy), formalizations of smaller, hopefully-representative languages (like Featherweight Java) are type safe.[ref] While full-scale languages rarely receive the same attention as smaller *core calculi* that represent them, in 1997, Standard ML (SML) was formally specified — both its semantics and type system — and proven to be type safe. Later work mechanized the metatheory of SML, adding further assurance to this result. This was landmark work to prove that a full-scale language was indeed type safe.[/ref] Interestingly, type safety hinges on the fact that behaviors that C's semantics deem as undefined, these languages give meaning to. Most notably, a program in C that accesses an array out of bounds has no meaning, but in Java and C# it does: this program will throw an *ArrayBoundsException*.

Python, Ruby: **type safe** (arguably). Python or Ruby are often referred to as *dynamically typed languages*, which throw exceptions to signal type errors occurring during execution: Just like Java throws an *ArrayBoundsException* on an array overflow at run-time, Ruby will throw an exception if you try to add an integer and a string. In both cases, this behavior is prescribed by the language semantics, and therefore the programs are well defined. In fact, the language semantics gives meaning to all programs, so the *well defined* and *all* circles of our diagram coincide. As such, we can think of these languages as type safe according to the *null* type system,[ref]Sometimes languages like Python and Ruby are characterized as *uni-typed*, meaning that for the purposes of type checking, all objects in the language have one type that accepts all operations, but these operations may fail at run-time. Programmers may not think of the languages this way, though.[/ref] which accepts all programs, none of which go wrong (making all three circles coincide). Hence: type safety.

This conclusion might seem strange. In Java, if program **o.m()** is deemed well typed, then type safety ensures that **o** is an object that has a no-argument method **m**, and so the call will always succeed. In Ruby, the same program **o.m()** is *always* deemed well typed by Ruby's (null) type system, but when we run it, we have no guarantee that **o** defines the method **m**, and as such either the call will succeed, or it will result in an exception.

In short, type safety does not mean one thing. What it ensures depends on the language semantics, which implicitly defines wrong behavior. In Java, calling a nonexistent method is wrong. In Ruby, it is not: doing so will simply produce an exception.[ref]You could imagine building an alternative type system to rule out some of Ruby's run-time errors; indeed that's what the Diamondback Ruby project tries to do.[/ref]

Beyond the Basics

Generic type safety is useful: without it, we have no guarantee that the programs we run are properly defined, in which case they could do essentially anything. C/C++'s allowance of undefined behavior is the source of a myriad security exploits, from stack smashing to format string attacks. Such exploits are not possible in type safe languages.

On the other hand, as our discussion above about Ruby and Java illustrates, not all type systems are created equal: some can ensure properties that others cannot. As such, we should not just ask whether a language is type safe; we should ask what type safety actually buys you. We'll finish out this post with a few examples of what type systems can do, drawing from a rich landscape of ongoing research.

Narrowing the gap

The *well typed* and *well defined* circles in our diagram do not match up; in the gap between them are programs that are well defined but the type system nevertheless rejects. As an example, most type systems will reject the following program:

```
1  if (p) x = 5;
2    else x = "hello";
3  if (p) return x+5;
4    else return strlen(x);
```

This program will always return an integer, but the type system may reject it because the variable **x** is used as both an **int** and a **string**. Drawing an analogy with static analysis, as discussed previously in the context of the Heartbleed bug, type systems are *sound* but *incomplete*. This incompleteness is a source of frustration for programmers, (and may be one reason driving them to use dynamic languages like Python and Ruby). One remedy is to design type systems that narrow the gap, accepting more programs.

As an example if this process in action, Java's type system was extended in version 1.5 with the notion of *generics*. Whereas in Java 1.4 you might need to use a cast to convince the type system to accept a program, in Java 1.5 this cast might not be needed. As another example, consider the lambda calculus, the basis of functional programming languages. The simple type system for the lambda calculus accepts strictly fewer programs than Milner's polymorphic type system, which accepts fewer programs than a type system that supports Rank-2 (or higher) polymorphism. Designing type systems that are expressive (more complete) *and* usable is a rich area of research.

Enforcing invariants

Typical languages have types like **int** and **string**. Type safety will ensure that a program expression that claims to be an **int** actually is: it will evaluate to -1, 2, 47, etc. at run-time. But we don't need to stop with **int**: A type system can support far richer types and thereby express more interesting properties about program expressions.

For example, there has been much recent interest in the research community in *refinement types*, which refine the set of a type's possible values using logical formulae. The type **{v: int | 0 <= v}** refines the type **int** with formula **0 <= v**, and in effect defines the type of non-negative integers. Refinement types allow programmers to express data structure invariants in the types of those data structures, and due to type safety be assured that those invariants will always hold. Refinement type systems have been developed for Haskell and F# (called Liquid Haskell and F7, respectively), among other languages.

As another example, we can use a type system to ensure data race freedom by enforcing the invariant that a shared variable is only ever accessed by a thread that holds the lock that guards that variable. The type of a shared variable describes the locks that protect it. Types for safe locking was first proposed by Abadi and Flanagan, and implementations have been developed for Java and C (Locksmith).

There are many other examples, with type systems designed to restrict the use of tainted data, to prevent the release of private information, to ensure object usage follows a strict protocol called *type state* (e.g., so that an object is not used after it is freed), and more.

Type abstraction and information hiding

Many programming languages aim to allow programmers to enforce *data abstraction* (sometimes called *information hiding*). These languages permit writing abstractions, like classes or modules or functions, that keep their internals hidden from the client code that uses them. Doing so leads to more robust and maintainable programs, because the internals can be changed without affecting the client code.

Type systems can play a crucial role in enforcing abstraction. For example, with the aid of a well designed type system, we can prove representation independence, which says that programs only depend on the behavior of an abstraction, not on the way it is implemented. Type-enforced abstraction can also be an enabler of free theorems, as Wadler elegantly showed. The late John Reynolds did groundbreaking work on types and abstraction, most notably described in his 1983 paper, Types, Abstraction, and Parametric Polymorphism. In this paper he famously states "*type structure is a syntactic discipline for maintaining levels of abstraction,*" positing that types have a *fundamental* role in building maintainable systems.

Parting thought

Type safety is an important property. At the least, a type safe language guarantees its programs are well defined. This guarantee is necessary for reasoning about what programs might do, which is particularly important when security is a concern. But type systems can do more, forming a foundation for reasoning about programs, ensuring that they enforce invariants, and maintain abstractions. As software is becoming ever more pervasive and more complicated, type systems are an important tool for ensuring the systems we rely on are trustworthy and secure.

If you are interested in learning more about type systems, I'd recommend starting with Benjamin Pierce's books, Types and Programming Languages, and Advanced Topics in Types and Programming Languages.

Thanks to Matthew Hammer, Jeff Foster, and David Van Horn for comments and suggestions on drafts of this post.

40 Responses to *What is type safety?*

Derek Dreyer
August 5, 2014 at 7:31 am

Nice post, Mike. One correction, though: you write that "we should not just ask whether a language is type safe; we should ask what type safety actually buys you." You then list some stronger properties of type systems, like representation independence. However, "type safety" as it is commonly known — and as you have defined it here, in terms of well-typed programs having well-defined behavior — does not imply representation independence. Certain type systems do enjoy representation independence, but this does not *follow* from (syntactic) type safety: it's an independent, and strictly stronger, property. This was a key point of my PLMW talk this year: http://www.mpi-sws.org/~dreyer/talks/plmw2014-talk.pdf

Reply

Michael Hicks
August 5, 2014 at 7:34 am

Derek, thanks for the clarification. What you say is exactly what I meant: type systems can be a foundation for proving stronger properties like representation independence. Thanks for the link to your talk; it looks really helpful!

Reply

Derek Dreyer

August 5, 2014 at 8:00 am

I guess part of the point of my clarification, which is really amplifying a theme of your post, is that “type safety” is not a clear way of differentiating between languages. Languages with very weak type systems (especially reflection mechanisms) can be perfectly “type-safe”. This doesn’t mean type safety isn’t useful from a software engineering perspective, but I think we overemphasize it at the expense of more important concepts like data abstraction, which is rarely formalized precisely in an undergrad PL course. Part of the reason that we have traditionally done so is because type safety is much easier to state and prove than representation independence is, but it’s also much less informative.

Reply

Michael Hicks

August 5, 2014 at 8:32 am

Good call. In addition to the talk you linked already, can you suggest good materials for teaching these concepts, e.g., to advanced undergrads or junior grad students?

Reply

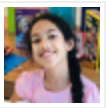
Derek Dreyer

August 5, 2014 at 8:41 am

Not really. 😕 I taught myself about representation independence by reading various papers, like Pitts’s chapter in ATTAPL, but most of these papers are not so easy to digest. I have plans to write a book/course notes about this material, but I have not made much progress on it yet. The students at the Oregon PL summer school asked me the same thing, and I will prepare at least a list of useful papers to read, with some commentary on them, but there’s no really good text for this stuff that I know of…yet.

Part of the problem is that the interesting properties like representation independence are deep properties, and the techniques necessary to scale proofs of those properties to realistic languages combining OO, FP, concurrency, etc. were only developed in the last couple years. And they are still in active development, so it’s a bit of a moving target. But the book would probably stop short of the most recent developments anyway.

Avik Chaudhuri



August 5, 2014 at 8:02 am

To me, types are really all about capturing invariants, and everything else follows from there. When designing a type system, it is useful to think what the meaning of each type is, so that programs with that type can be assigned that meaning, and that meaning is preserved as they execute. That meaning could be something entirely trivial, or something very deep: the condition that it be invariant is the only thing that constrains it, but it forces the language semantics to be aligned with it. Another related phenomenon is that the exercise of designing a type system often feels much like constructing a proof; just like you may need a stronger lemma to prove a theorem, so you may need some complicated “intermediate” types (capturing “intermediate” invariants) to give a consistent meaning to everything in your program.

Reply

Robert Harper

August 5, 2014 at 9:03 am

There is no independent concept of “type safety” that a language either enjoys or does not enjoy. Type safety is an internal coherence property stating that the statics and the dynamics fit together. It is always true by definition, and therefore never interesting. “True by definition” means, if it weren’t true, you’d change the definition. A language L is minimally sensible iff it enjoys type safety in the sense of L. It makes no sense to ask whether L is “type safe” (or “memory safe”) in some independent sense that L’ might enjoy but L might not.

For example, C is perfectly type safe. It’s semantics is a mapping from 2^64 64-bit words to 2^64 64-bit words. It should be perfectly possible to call rnd(), cast the result as a word pointer, write to it, and read it back to get the same value. Unix never implemented the C dynamics properly, so we get absurdities like “Bus Error” that literally have no meaning whatsoever in terms of C code. Or we have talk about “stacks”, which do not exist in C code, but rather are an artifact of its Unix-implemented dynamics. As long as stacks are represented by the semantics as regions of memory, there are no such things as “stack overflows” or the like. Were stacks an abstract part of the dynamics, then such concepts would make sense, but C does not do this. The one exception to safety for C is the idea of casting as a function pointer, which requires a commitment to the representation of programs in memory. In reality no one implements or cares about the von Neumann model that we supposedly use, so it would make perfect sense to define a semantics for C that separated program from memory (and perhaps stack from memory).

The problem with C is the pretension of an abstract type system for what is in reality a concrete language. BCPL was much more honest about this, and vastly superior for this reason. People impose statics and dynamics on C that really don’t apply, and find that when they do, the “design” they come up with is incoherent, which is to say not type safe. If you want “security” properties of languages, then what you’re saying is that you want an abstract language, not a concrete one, that abstracts from memory representations and so forth, and that can support theorems that state that desirable behavior is ensured. One can prove such theorems about concrete languages, but the “desirable behaviors” are rather pathetic, because there is no abstraction.

It is not a matter of opinion whether an untyped language is unityped, it is a matter of fact. Quite a lot of people think if you swing a tethered ball about your head in a circular motion and then release it, it will continue to curve. So what?

It is absurd to attribute type refinement to, say, Liquid Haskell (whatever the merits of that fine work may be). This form of typing goes back to Brouwer (whose conception of the semantics of constructive mathematics has nothing to do with what is currently popularized as the “propositions-as-types principle”.) Brouwer’s conception is much closer to Kleene’s realizability, which was a first attempt to formulate it precisely, and to its development in full as a theory of truth in the NuPRL type system, which most emphatically is not based on the overplayed formal correspondence between programs and proofs. Freeman and Pfenning, in the late 80’s, developed a syntactic formulation, called type refinements, that was further developed by Davies 20 years ago. The syntactic formulations are a pale approximation to truth, but are useful in practice. Davies and I recently released a preliminary paper on applying type refinements (as realizability) to dynamic dispatch so that one may verify, say, the absence of “not understood” errors. (See my web page for a link.)

I fully agree with Derek’s emphasis on properties such as parametricity, which are behavioral properties of types expressed in a realizability framework, are far more important for the programmer than is type safety. A safe (i.e., internally coherent) static type discipline induces, via the theory of logical relations, a canonical behavioral type system that provides some of these sorts of properties “for free”, but of course only if the language in question is properly defined in the first place. In particular, the “types” have to make sense from the point of view of logic and category theory, and cannot be arbitrary concoctions written down with horizontal lines, and, moreover, they must cohere with the dynamics according to Gentzen’s principles (from the 1930’s).

All of these ideas are explained much more fully in my Practical Foundations for Programming Languages, in case anyone is interested in learning more.

Reply

Michael Hicks

August 5, 2014 at 9:14 am

Bob, thanks for the excellent thoughts and historical perspective, and apologies for not knowing some of this stuff beforehand. For those interested, here's a link to Bob's book.

One comment about C: when I am referring to the language's semantics, I am referring to the C standard, which specifically states that certain programs are undefined. The type system, also defined by the standard, does not rule out these programs, so it is not safe with respect to the standard's semantics. Of course, you can reasonably argue (and have done so!) that the standard is not defining the "true" semantics of C, and that an alternative semantics with a lower level interpretation is more meaningful, and safe. But as a practical matter, programmers are often taught C according to its high-level semantics (at the level of the standard), and pointing out that the language is not type-safe with respect to this semantics, while other languages are, is useful. At least colloquially, I've commonly heard the statement that C is not type-safe, and here I am attempting to explain why that statement is (arguably) true.

Reply

anon_coward

August 5, 2014 at 1:00 pm

> It is not a matter of opinion whether an untyped language is unityped, it is a matter of fact.

With all due respect, stating that something is true does not make it true, regardless of your stature and contributions to the field.

Reply

Robert Harper

August 5, 2014 at 1:40 pm

I rely on theorems, not stature or contribution. It is all fully documented in PFPL. I've not seen anything resembling a counterexample to what I have proved and shown there, or, for that matter, to Scott's theorem dating back to the '70's.

Reply

BobHarper

August 8, 2014 at 12:39 pm

You are missing the point. Your results are not being questioned.

What is being questioned is why your definitions and understanding, useful though they may be, take precedence over other people's.

Reply

Sandro Magi

August 8, 2014 at 9:33 am

I'm not sure classifying C as type safe is coherent. A critical aspect of type safety in my mind is that a language ought to protect its own abstractions. The fact that violating the constraints of these abstractions is "undefined" in C, means this principle is violated.

Reply

Nat R

March 8, 2015 at 3:28 pm

This description of the real C type system is perfect, and it's very evident when c is made to run on top of other languages, such as Crossbridge/FlasCC where the unitype is a AS3 ByteArray and Emscripten where Javascript typed or untyped arrays can be used.

Reply

Christopher

August 5, 2014 at 11:14 am

All I could think about while reading this article: https://www.destroyallsoftware.com/talks/useing-youre-types-good

You make good points though. Type safety is hard and doesn't just mean one thing.

Reply

J. Ian Johnson

August 7, 2014 at 1:31 pm

Your theorems rely on definitions that not everyone agrees on. That is where the stature comment really matters: because of who you are, your definition is the right one, the only one that makes sense, and why anyone would think otherwise is either a complete mystery or evidence of their ignorance. Are we, PL researchers, in the business of telling people what they mean? Linguists would say this is presumptuous, arbitrary prescriptivism. Meanings evolve as contexts and ideas change. You can say, "with my definition of programming language, X, what you call an untyped language is really is a typed language with one type." You then get people like Sam Tobin-Hochstadt who would disagree with your definition.
I do appreciate the bulk of your work, but to say your opinions are provocative is an understatement. I'm reminded of a joke about the scientists lost in a drifting hot-air balloon, wondering where they are. The mathematician says, "we're in a balloon." He's completely right, but it doesn't help the matter at hand. You can be technically right and still completely miss the point. There's yet another joke that comes to mind: "Knock knock."
"Who's there?"
"Interrupting tech bro"
"Interru.."
"ACTUALLY!"

Reply

Robert Harper

August 7, 2014 at 1:33 pm

To reiterate, I deal in facts and proofs. Show me a counterexample, or show me another theorem, otherwise there is no science to be done.

Reply

anon_coward

August 8, 2014 at 12:47 pm

> there is no science to be done.

That's exactly the point — when you come up with and use your definitions, you're "doing science". When you attempt to force others to use your definitions, you're *not* "doing science", despite your repeated claims along the lines of "I deal in facts and proofs".

Since the "doing science" aspect of your work isn't the issue here, counterexamples and theorems are not relevant to the discussion.

Reply

J. Ian Johnson

August 7, 2014 at 1:48 pm

A counterexample is John Reynolds' qualitative definition of type system that Sam points to in his article.
"Type structure is a syntactic discipline for enforcing levels of abstraction."
You can interpret it in such a way that a language with no syntactic discipline for enforcing abstraction has no type structure. It has no type system. An example is x86 assembly. I philosophically disagree with describing one level, the program text itself, an abstraction.
The question of fact is a philosophical one as well. If you want to conflate fact with theorem, then you still have to import contentious definitions. If not, and fact is a more metaphysical ideal of "Truth," whatever that may be, then we go back to proof by authority as anon first objected to.

Reply

Robert Harper

August 7, 2014 at 3:48 pm

I've shown in PFPL (building on Dana Scott's results) that dynamic typing is a mode of use of static typing, and does, therefore, enforce levels of abstraction, by Reynolds's dictum. Therefore the statement you quote is not correct.

Reply

J. Ian Johnson

August 7, 2014 at 4:24 pm

Which levels? I already rejected your notion that a singleton set of "levels of abstraction" is an abstraction on philosophical grounds, to play devil's advocate. Do you argue that if you can embed a language X into another language Y, that X is really /just/ a restricted or trivial use of Y? If that is the case, do you also consider lazy languages strict? If not, why? An embedding need not preserve expressiveness, either. If I can compile language X to language Y, is X /just/ a use of Y? Is SML just a restricted use of x86?
The theoretical PL/category-theoretic "just" to describe concepts in terms of others as a way to dismiss the concrete idea is obnoxious and misses the point of being concrete for the sake of human consumption.
Your definitions aren't everyone's. Don't go around the internet shouting that people are wrong on this topic. If you must shout, shout that people should adopt your definitions because you believe that they offer greater insight into the operations of programming languages. In

many ways I agree with a lot of what you say because I know the context in which you say them. I disagree with the way you say them because unstated assumptions and personal taste make you sound dogmatic and vitriolic.

Reply

Robert Harper

August 7, 2014 at 4:42 pm

The justification is given fully in PFPL; I know of no error, or alternative account that refutes it. I am sorry that this fact induces such a vitriolic and dogmatic response; it baffles me that there is such opposition to a very simple, but very significant imo, fact. As a technical coda, the embedding is fully abstract, so the comparisons you suggest do not pertain to the point at hand. That this is so relies on parametricity, which, as Derek suggested, is of fundamental importance in both theory and practice.

Reply

J. Ian Johnson

August 7, 2014 at 5:06 pm

In reality, I don't oppose to the truth of the statement. I oppose the attitude that comes with it. I'm trying to comprehend this opinion that "since X compiles fully abstractly into language Y, it is a needless restriction to use language X." to quote PFPL, "every dynamic language is inherently a static language in which we confine ourselves to a (needlessly) restricted type discipline to ensure safety."
You have made statements such as "I can write Scheme in SML and still have types elsewhere, but I don't know why I'd restrict myself to just one type." One, this isn't true, because SML doesn't have a feature like Racket's #lang that actually allows you to use Scheme syntax and expand into all the fold/unfolds that SML would require. Also, macros. You have to write a big ergonomically unwise mess. Two, adhering to a static typing discipline that draws several distinctions of expressions leads to code duplication, contortion, and headaches when light testing afterwards is all you need.
I don't know about your tendon health, but I need the restriction to one type because I can't afford to keep writing fold/unfold everywhere in my programs. I also need standard libraries to not require adherence to rigid types so that the typing requirements don't bleed into my own code.
This isn't because I'm not "taking my vitamins" like you've previously described adhering to a typing discipline. This is because I'd rather not justify the correctness of my EDSL implementation strategy to an adversarial type checker, which is deserving of PhDs, when I can be reasonably confident in its correctness with a few tests.

Reply

Robert Harper

August 7, 2014 at 5:12 pm

Well, good luck with that; testing never beats verification. Funnily enough, your own emphasis on syntax extension is exactly how one would avoid the syntactic difficulties you complain about. In any case my point is about semantics, which is what matters first of all.

BTW, I don't recall using vitamins as a metaphor in this context; I'd appreciate a reference for that comment.

And regarding DSL's, it would be nice to have a definition of what is a domain to be specific to, and why one would think that being "domain-specific" is a benefit rather than a drawback. But that's another conversation; I've written about this topic on my blog in case you may be interested.

Reply

Stephen Kell

August 14, 2014 at 9:33 am

Hi Mike. Thanks (again) for tackling this question… I enjoyed reading. Sorry I'm late to the party. I just want to pick a few nits.

Firstly, the statement towards the end:

"A type safe language guarantees its programs are well defined. This guarantee is necessary for reasoning about what programs might do"

… is not true. It is turning an existential into a universal. It would be true if you said "what *all* programs [in the language] might do". Some C programs are well-defined for all inputs. Others are well-defined for some inputs. We can certainly reason about all these cases.

Secondly, I'd contend that your definition, although it is a fine and consistent one, is more general than what most programmers understand by "type safety". For example, if we made division by zero a case of undefined behaviour, most programmers would not suddenly consider it a "type error" to divide by zero, and consequently would not consider the feasibility of this error to be a question of "type safety".

Personally I find it helps to think first about "type correctness" (a property of executions, also easily extended to programs), and then "type safety" by extension of that (the property that all programs are type-correct for all inputs). But perhaps I should save elaborating on that for a (hypothetical) blog post of my own.

Also, I'd argue that "data abstraction" and "information hiding" are not the same thing: information hiding is a mechanism that can be used to enforce data abstraction, but it is not data abstraction itself. The act of data abstraction is at least somewhat valuable even if we do nothing to enforce it, or if we enforce it by other means (e.g. dynamically). I have an essay relating to all this appearing at Onward! later this year. (it's not quite camera-ready yet, but soon will be. I'll send you a copy if you're interested.)

Reply

Pingback: Interview with Avik Chaudhuri - The PL Enthusiast

HaskellHead

March 3, 2015 at 8:53 pm

> For example, if we made division by zero a case of undefined behaviour, most
> programmers would not suddenly consider it a "type error" to divide by zero,
> and consequently would not consider the feasibility of this error to be a
> question of "type safety".
But you can easily eliminate a "divide by zero" error with proper type safety.

Consider Haskell:

```
data Maybe a = Nothing | Just a

safeDivide :: (Num a, Num b, Num c) => a -> b -> Maybe c
safeDivide _ 0 = Nothing
safeDivide dividend divisor = Just (dividend / divisor)
```

Reply

Paritosh Kulkarni

March 22, 2015 at 2:51 am

What is relation between Type Safety and memory Safety
Bobs post is confusing and says me that mem safe language may or may not be type safe

Reply

Michael Hicks

March 22, 2015 at 2:08 pm

Memory Safety is a prerequisite for Type Safety, in the way I have set things up in this post. In particular, if a language is not memory safe, then that fact can be used to circumvent type safety. E.g., you can write outside the bounds of a buffer to other memory, and write ill-typed values to that memory, e.g., you can write an integer 5 past the end of an integer array, where the 5 is stored in memory that is supposed to be a pointer.

Reply

Robert Harper

March 22, 2015 at 2:32 pm

Memory safety is a marketing term for type safety.

Reply

Pingback: Knowledge is Power | Software Security Ideas Ahead of Their Time

Pingback: SecDev: Bringing Security Innovation Into Design & Development

Pingback: Teaching Programming Languages (part 2) - The PL Enthusiast

Pingback: Software Security is a Programming Languages Issue | MySafePick

Pingback: What is Type-safe? – Bloofer Blog

Pingback: Software Security is a Programming Languages Issue | SIGPLAN Blog

Pingback: Teaching Programming Languages (part 2) | essay studess

Pingback: How difficult is it to write correct smart contracts? Depends on your tools!

Pingback: CodeNewbie > Seekalgo

Pingback: Is Python type safe? - Tutorial Guruji

Pingback: Is Python type safe? – Row Coding