rust is not about memory safety

01 june, 2024

most of rust discussions nowadays revolve around memory safety, and how it is safer than C / C++ / zig / go / whatever language is being trashed on twitter that day. while yes, that is true - not that the bar for most of these is particularly high - what I think is the main point of the language is always glossed over: correctness. when one tries to criticize any of the aforementioned languages, one is answered with the following argument:

> *your program segfaults? skill issue*

but i'd like to make the counter-argument that, no, this has nothing to do with skill issue.

formal language theory

the first thing one learns when they're studying formal languages (the field that studies grammars, state automata, etc) is that the rules that describe a certain grammar must match **exactly** the ones that you want to include in your language. this means that there's a bidirectional relationship between the grammar you describe (which directly define the automata that parses that language) and the words[1] that it parses (which are related to the semantics of the program, how it executes).

from it, it can be inferred that the grammar **must not** allow in the language any words that does not have defined semantics, and in the opposite direction, that the language should not specify semantics to any program that cannot be parsed by the rules of the grammar. both of these are required in order to make study of this grammar <-> language partnership fun, pleasing, and most importantly sound.

going beyond, formal language theory also gives you the knowledge that the execution of any program can be given as a set of grammar rules in an abstract machine (the most famous one being a turing machine). in the same way you can define a set of grammar rules to parse parenthesized arithmetic expressions using a stack automaton, you can define a set of grammar rules to model the execution of a C program, that, albeit super complex, can be modeled as a turing machine. this usually gets the name of C abstract machine, and is the basis for formal specification of behavior in the language.

and no, i'm not talking about modeling a C parser as a state machine (which probably is easier than most languages, if you ignore pre-processor stuff). i'm talking about modeling C **execution** as a language being parsed. drawing a parallel, when parsing parenthesized expressions, you pop things in and out of the stack to represent "balancedness", and in the same way, when "parsecuting" C code, you must write to memory, represent side effects, represent type casts and pointer conversions and everything **as part of the language**.

in the same way that you'd hope that a parenthesized arithmetic expression parser would recognize that `(1 + 2) + 3)` is an invalid expression, you'd expect that the C compiler would correctly verify that the following series of tokens is not a *well behaved* program:

```
int foo(int * myptr) {
  *myptr = 5;
}
foo(NULL);
```

i say *well behaved* because i can't say *invalid*. it is in fact defined by the spec that when you dereference a `NULL` pointer the result is *undefined behavior*. and this is C's achilles heel: instead of outright banning programs like the one above (which i'd argue is the correct approach), it will happily compile and give you garbage output.

framing it this way really exposes the fragility of C, because undefined behavior has to always be taken into account. and, by the nature of it, there is no way to represent it other than as a black box, such that, if your code ever encounters it, then literally all you can say is that **the whole result of the program** is undefined - that is, it can be anything. you cannot show properties, nor say what will happen once your program enters this state, as the C specification literally does not define it. it may come to a halt, write garbage to the screen or completely delete half of the files of your program, and there's no way to predict what will come out of it, by definition. in the lucky case, it will segfault while executing and you'll be extremely pissed off, but that is not at all guaranteed. this is akin to having a float expression with some deep term being `NaN`, in that it eventually must evaluate to `NaN` and you can't draw any conclusions about the result of the expression (other that it isn't a number).

language designers and compiler developers are by no means dumb, and yes, they know much, much more than me about these problems. undefined behavior exists exactly because there must be parts of your code that your compiler **must** assume that aren't possible, so that it can correctly compile. for example, let's say that you inadvertently try to dereference a pointer that you have no knowledge about. the C compiler simply does not have enough information to know if it is `NULL`, if it is still pointing to valid memory, or if the memory has been initialized, so it's approach is to simply emit code **as if** it was a valid, initialized, non-null pointer.

it is essential to realize that this is an **assumption**, and in almost most cases, the compiler does not care whether or not it was actually still valid, so writing to it may have a myriad of effects of different effects (none of which are the compiler's concerns). sometimes, your system might correctly intercept a read/write from invalid/null memory and raise you a signal, but that is not guaranteed.

and there are a huge number of tools to aid in finding undefined behavior in a code base, it's just that

1. they are not by any means standards of C development (not in spec and not in standard compilers) and
2. they are fallible and will always let some undefined programs slip by.

runtime exceptions are not the solution

most languages try to handle this by introducing some sort of runtime exception system, which i think is a terrible idea. while this is much, much safer than what C does, it still makes reasoning about the code extremely hard by completely obliterating locality of reason. your indexing operation may still be out of bounds, and while this now has defined outcomes, it is one of the possible outcomes of your program (whether you like it or not), and you must handle it. and, of course, no one handles all of them, for it is humanely impossible to do it in most languages because:

1. it is hard to know when an operation can raise an exception, and under which conditions.
2. even if documented, it is never enforced that all exceptions must be gracefully handled, so some random function in a dependency of a dependency may raise an error from an unexpected corner case and you must deal with it.

this is a symptom of virtually all modern languages, and none of them have any good answers to it. java mandates that you report in your function type signature the errors that it may raise (which is a rare java W), but it does let you write code with unchecked exceptions that won't signal a compile error if ignored, which eventually will crash your minecraft game. python, ruby, php and most other languages (even haskell made this mistake) do not even attempt to signal when a function might raise an exception.

javascript somehow manages to be even worse, by having horrible implicit-by-default type casts, having undefined AND null, using strings as UTF-16, using floats as standard numbers, implicitly inserting semicolons, and honestly the list could go on forever.

the root of all these problems is, quite literally, the same: that your compiler (or interpreter) lets into your program execution states that you didn't anticipate for. one of the best of examples of the opposite, surprisingly enough, is regex matchers. while i concede that their syntax can be extremely confusing, they have the best property of software: if they compile, they work exactly as intended - which i henceforth will call **correctness**. this is because regular languages' properties and their state automata have been studied to extreme depths, and it is entirely possible to write a regex implementation that is **correct** (in the same way as above), going as far as providing formal verifications of that [2].

from this definition of **correctness** we can also derive a semantically useful definition for the word bug: an unexpected outcome for the program, that shouldn't be allowed in the language. of course java behavior might be defined for all inputs (for the most part, i'm sure there are might be problems here and there) but just because one possible outcome of program is `NullPointerException` doesn't mean that it is **expected**, making it, by my definition, a bug.

make invalid states unrepresentable

what the regex example makes clear is that the key to correctness is to make your language tight enough to have defined and **desired** output for all possible inputs. this is not to say that it won't raise errors; much to the contrary, it must have parser errors saying that some strings aren't valid regexes. instead, it means that all errors are **predictable**, and **well defined** (in some sense).

you, as the programmer, are then in charge of ensuring that the resulting regex program actually solves the problem you have at hand. want to match 3 words of 2 digit numbers followed by a capital letter? great, they can do that. want to match balanced parenthesized expressions? sadly, regex is incapable of ever solving that, because that language is not regular, so no matter how hard you try it will never solve it.

in a way, there's a beauty in how C sidesteps this: it defines one of the possible program outputs as being *undefined*, and it is on the programmers behalf to tightly ensure that the program has 0 paths to *undefined behavior*. in fact, it is probably one of the most well specified languages, which is what makes it suitable for writing formally verifiable programs [3].

the main strength of rust, and where it differs from all mainstream languages, is that it has a very clear focus on program **correctness**. the raison d'être of the borrow checker is statically assuring that all references are pointing to valid memory, such that it is literally impossible for any borrow be null or to point to some freed memory (modulus implementation errors of course). this completely rules out this possibility of bugs from the language we're trying to "parse". remember the C excerpt from above, where i said that the compiler should rule out the program as invalid? well, it is literally impossible to write that sort of program in rust, because one cannot construct a `NULL` reference.

not only that, but rust languages features makes it so, so much easier to write **correct** software: sum types (tagged unions), `Option` instead of `NULL` (which in and of itself is amazing), `Result` for errors (making obligatory to handle all possible branches your program can take), a strong and powerful static type system, and ditching inheritance and classes in favor of traits.

note that i never ever talked about memory safety. even in a world where C wasn't in fact full of memory vulnerabilities, rust would still be miles better, because it statically assures you that the **meaning of your program is tightly reproduced by the code you've written**. it is, by design, more correct than C, and the only way a problem can possibly happen is by side stepping rust static checks by using `unsafe`.

it is just a happy coincidence that this leads to a language that isn't garbage collected, that is relatively lean, fast, easy to embed, has good ergonomics and that enables you to write asynchronous and multi-threaded programs. these properties are awesome to boost rust to a very well regarded status between developers, but aren't at all related to languages that enable you to build reliable, correct software. out of curiosity, i'd happily defend the case that coq is also one of these languages, and it absolutely does not hold any of these properties.

software engineering as a craft

finally, i think this relates to how i personally model the software development job as a whole. it starts by having some problem you think you can use computers to solve, and then follow 3 clearly stratified steps:

1. define how one might solve the problem. this usually means splitting it into several possible cases and treating each and every one of them separately.
2. define an abstract machine that executes the very same steps, **and making sure that it tightly adheres to your plan**
3. implement the very same machine in a language, **making sure that your implementation adheres tightly to your abstract machine**

the part that programmers usually get paid millions of dollars for is the step **1 -> 2**, which is by far the hardest and that requires the most creativity and craftsmanship. what usually makes people say that software is in decline is that we don't learn the value of executing step **3** properly. this leads to sloppy, half baked software that crashes when X Y Z happens, and we've just come to terms with software being so brittle.

it is not by chance that Yang et al. could only find measly 9 bugs after 6 CPU years of fuzzing in compcert, a formally verified c compiler (written in coq), where as in gcc and clang, they found and reported more than 300. all these 9 bugs where in the unverified front end of the compiler (the parser), and there were literally 0 middle end (compiler passes and AST translations) bugs found, which is unheard of. this is not by chance, they've spent many years writing proofs that all of their passes are correct, safe, and preserve the meaning of the original program.

i really think software developers should strive for that kind of resilience, which i believe can only be achieved through properly valuing **correctness** . i don't think it is reasonable to expect that all software be built using coq and proving every little bit of it (due to business constraints) but i think that rust is a good enough language to start taking things more seriously.

---

1. formally they are defined as a sequence of tokens in certain alphabet that the automata closures over. normally we think of "words" as the whole program that we're parsing. ↩

2. the excellent software foundations book explains thoroughly how one might formally write one possible regex matcher, and prove that the implementation is correct ↩

3. through the use of external tools like coq's verifiable C series ↩

#rust #correctness

leonardo [dot] ribeiro [dot] santiago [at] gmail [dot] com |