1 Programming Languages

Many developers want to feel in control and gravitate to so-called powerful languages. Powerful languages seem to be synonymous with languages that don't get in the way of the developer. The concept of a powerful programming language is a weird one. In essence, every programming language should be powerful — assuming it's Turing complete. This is a deceptive view of programming language though and has been called the Turing tar-pit [5]. Everything is possible in a Turing tar-pit, but nothing of interest is easy. A powerful languages is thus a language where things of interest are at least perceived to be easy. As we'll discuss in the following sections; this perception is often deceptive.

There are developers on the other side of the spectrum however, who have a more stoic approach to programming. These people value reliability above all else. More reliable code is in turn easier to reason about and easier to collaborate on with other people. These languages often have a sound theoretical foundation, such as the lambda calculus. But unfortunately, unless you intend on using those languages for mathematical work, they can get stuck in the Turing tar-pit.

Both sides raise valid points, so which is the correct way to program? Recent development seem to point towards finding a middle ground. Many languages that could be considered to be on the powerful side of the spectrum are moving towards the center of the spectrum. C# has been under heavy influence of Haskell [10], Scala emerged as a more strict alternative to Java and in turn influenced Java's development. New languages have been created in search of the sweet spot of the spectrum. Erlang was created at Ericsson to make their telecommunications backend more fault-tolerant [7]. Go was made by Google to be an efficient programming language, that's easy to work with and that has reliable concurrency [6]. Rust was created by Mozilla because they needed a more reliable alternative to C to power their new Servo rendering engine which makes extensive use of concurrency [8]. Many of these new languages come with considerable limitations but are undoubtedly powerful — they're all created to serve a real and non-trivial goal.

Some languages are left behind in this recent trend towards reliability however. They are stuck with what they do because of backwards compatibility. This doesn't mean it's impossible to write reliable code in the languages however. Static analysis tools have been around ever since developers wanted more insurances about their code. One class of programming languages that is notoriously hard to analyze statically are the dynamic languages. Put optimistically, these languages are just too powerful.

In the following sections we'll illustrate why powerful languages aren't always in the best interest of a developer. After that we'll look at some *best practices* of programming and how some languages enforce them. Finally we'll look at what static analyzers are able to achieve for certain programming languages.

1.1 Powerful Languages

A lot of programmers see a programming language as just another tool that will dance to their whims and do whatever they want it to. This is realistic to some extent but there is more to it than that. Let's regard programming languages as companies and developers as their customers; in some cases this analogy is even entirely correct. Nobody in their right mind would expect the company to act in the customer's interests. Companies act in their own best interests, which just so happens to coincide with the customers' interests from time to time.

This analogy isn't meant as a sociological critique, it's meant to shine light on an important aspect of practical programming languages. Companies need advertisements and publicity to fuel their existence, and so do programming languages. One effective selling point for a programming language is how powerful they can make a developer feel. Feelings have found their way into something that was originally purely logical.

This analogy goes further: programming languages have their own priorities — aside from what developers actually want. The C programming language is a great example of this. Depending on who you ask, the best thing about C is either its efficiency or its simplicity. Both come at a cost. The cost of C's simplicity is well-known: C does very little for the developer. This makes the programmer feel in control, which many perceive as a good thing.

The efficiency also comes at a cost, and it's a one many people forget about. C's *raison d'être* isn't

making developers feel good about themselves, it's generating efficient code. It was created to be a thin abstraction over various assembly languages that were limiting software development at the time [9]. This has lead to the existence of so-called undefined behaviors in the language. Entire papers have been written on this subject [11] [1] [2]. But one striking thing is how recent a lot of these papers are. If the language is over 40 years old, why are people still writing about it? The problem is that the product has stopped working as advertised. Or rather, it's doing what the fine print has always said it could do.

As Bjarne Stroustrup once famously said: "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off". While the C vs C++ debate is a whole different topic, this quote is very relevant because of the context he gave it on his own website [4]. He says that what he said about C++ can be said of most powerful languages, While C++ made some things easier, it also brought with it a whole new class of problems. The problems might be rarer, but that also makes discovering them all that much harder.

Undefined behaviors are a prime example of these rare problems that are hard for people to discover. One of the papers that deals with this topic gives a very good example of this [1]. The following piece of code was part of PostreSQL:

```
if (arg2 == 0)
    ereport(ERROR,
      errcode(ERRCODE_DIVISION_BY_ZERO),
      errmsg("division by zero"));

/* No overflow is possible */
PG_RETURN_INT32((int32) arg1 / arg2);
```

The `ereport` function never returns, it does some logging before calling `exit`. In the mind of the developer this prevents the division by zero on the next line. Functions in C are always expected to return, so far so good. But recent advances in compiler optimizations have broken the above piece of code. Division by zero in undefined in C, so the compiler is under no particular obligation whenever `arg2` equals to 0. As a result, the conditional above the division gets flagged as dead code, and is removed entirely.

One could argue that this makes C a strict language. But I'd say the opposite is true, nothing is being enforced. C let's you do whatever you want as a developer, after which the compiler does what it wants. There's little interaction between the compiler and the developer.

Another set of languages that make developers feel powerful are dynamic languages.

…

1.2 Self-Imposed Strictness

Something that plagues our field of work — whether academic of professional — is that developers seem to overestimate their own grasp of a programming language. Why this is the case is probably a subject of psychology instead of computer science.

Douglas Crockford postulates that humans can approach a problem from two ways: either using their head or their gut [3]. The head is obviously the most reliable way but it's also the slowest way. Following your gut on the other hand is fast but error-prone. Ideally we a professional developer would only follow his head but realistically, the developers are under time constraints as well. So despite knowing better, we fall back on our gut. Our gut is also notoriously bad at following good programming practices. We all know we should document our code, that we should use clear variable names, avoid certain anti-patterns, … But we're all sinners in this regard. And to cope with the guilt we make up excuses: "we know what we're doing" or "nobody else is going to read this anyway'.

And herein lies the danger, we trivialize certain bad practices for the sake of productivity.

2 Implementation

2.0.1 Observation

Every execution branch is either taken, or it isn't. Figuring out which is the case is well-known to be uncomputable, for the simple reason that the branch condition can be arbitrarily hard to evaluate. This implies that in some cases, we can't decide whether or not a given branch gets taken. The best we can do in these conditions is conclude that the branch *might* get taken. In the following section, we'll denote this possibility with `Maybe`, in line with Python's `True` and `False`.

2.0.2 Observation

In some cases, we do need a definitive answer. Consider the following examples

```
if current is None:
  current = datetime.now()
```

```
if current is not None:
  print('{}-{}'.format(current.year,
                       current.month))
```

The above examples gives a pretty good indication that in some cases, we really need an exact example. This is particularly important for any sort of input validation. The first example is a common pattern for filling in optional arguments, while the second one is just good practice in general. Other examples include checking the length of a certain collections and type-checks.

2.0.3 Boolean Expressions with Certainty

There aren't a lot of boolean expressions which we can evaluate with certainty. Luckily enough, the ones that we can do are mostly the ones of interest. The `is` operator for example should compare the addresses of two objects and return `True` if and only if they're equal. So internally, we can mimic this behavior – and answer with certainty and under which conditions, the objects have the same address in our own analysis. The other possibilities are harder to do, and the best we can realistically return is `Maybe`.

The `==` operator is a special case. If the `__eq__` method was implemented correctly, this should at the very least return `True` if the two objects being compared are the same – as with `is`. The analyzer in its current state does not properly support analysis of operator overloading, so it will assume that `__eq__` does indeed have a sane implementation. The analysis of the `==` operator in this case becomes the same as the analysis of the `is` operator. Likewise for the `<=` and `>=` operators.

The `and` and `or` operators are quite obvious. `and` will return `True` if both sides are true with certainty, `False` if either side is false with certainty, and `Maybe` in any other case. The `or` operator is analogous.

2.0.4 Definition: containment

A path A is contained in another path B if every node of path A occurs in the same way as it does in path B

2.1 Path exclusion

When executing an execution branch, we should have information about why that specific branch is being executed. If that information includes for example that we are sure that `x` is not `None`, we should disregard any mapping that says the opposite. And even better, we can exclude any mapping that would occur under the same contradictory conditions – even if those mappings don't have an explicit connection to `x`. For example in the following trivial example:

```
if cond1:
  y = None
  z = None

if y is not None:
  print(z.attribute)
```

In the positive branch of the first condition, there's a point where both `y` and `z` become `None`. After evaluating the second branching condition, we can be absolutely sure that the positive branch of the second branch will not be taken if the positive branch of the taken has been taken. In effect, this means that the mapping for `z` where it receives the value `None`in the first branch is of no use while evaluating `z.attribute`.

The exclusion of certain mappings is what we'll conveniently call *path exclusion*. We can give this term a more formal representation as well.

Assume that resolving an identifier x resulted in a set of mappings M. Every mapping $m \in M$ is of the form (p, a), where a is the address to which x can point, and p is the execution path that's required to get this mapping from x to a.

Call R the set of restrictions; the set of every execution pth that is of no concern while evaluating. If there exists a path r in R for a given mapping (p, a), for which it holds that p is contained within r, we can exclude the mapping from the current evaluation.

References

[1] Wang, X. et al. 2012. Undefined behavior: What happened to my code? *Proceedings of the asia-pacific workshop on systems* (New York, NY, USA, 2012), 9:1–9:7.

[2] A guide to undefined behavior in c and c++. http://blog.regehr.org/archives/213.

[3] Crockford on javascript - section 8: Programming style & your brain. https://www.youtube.com/watch?v=taaEzHI9xyY.

[4] Did you really say that? http://www.stroustrup.com/bs_faq.html#really-say-that.

[5] Epigrams on programming. http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

[6] Golang faq. https://golang.org/doc/faq.

[7] History of erlang. https://www.erlang.org/course/history.

[8] Rust faq. https://www.rust-lang.org/en-US/faq.html.

[9] The development of the c language. https://www.bell-labs.com/usr/dmr/www/chist.html.

[10] The marvels of monads. http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

[11] What every c programmer should know about undefined behavior. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.