Programming languages shouldn't and needn't be Turing complete

GABRIEL PICKARD

Any algorithmic problem faced by application programmers in the wild[1] can in principle be solved using a Turing incomplete programming language. [Rice 1953] suggests that Turing completeness bears a heavy price, fundamentally limiting the ability of automatic assistants to help programmers be more productive, create less bugs and write safer software.

Nevertheless, no mainstream general purpose programming language is Turing incomplete, with the arguable exception of *SQL*. We identify historical causes for this discrepancy and argue that the current moment offers better conditions for introducing Turing incompleteness.

We present an approach to eliminating Turing completeness during the language design process, with several examples suitable languages targeting application development.

Designers of modern practical programming languages should strongly consider evading Turing completeness and enabling better verification, security, automated testing and distributed computing.

1 CEDING POWER TO GAIN CONTROL

1.1 Turing completeness considered harmful

The history of programming language design[2] has been a history of removing powers which were considered harmful (cf. [Dijkstra 1966]) and replacing them with tamer, more manageable and automated constructs, including:

(1) Direct access to CPU registers and instructions → higher level compilers
(2) `GOTO` → structured programming
(3) Manual memory management → garbage collection (and borrow checkers)
(4) Arbitrary access → information hiding
(5) Side-effects and mutability → pure functions and persistent data structures

While not all application domains benefit from all the replacement steps above[3], it appears as if removing some expressive power can often enable a new kind of programming, even a new kind of application altogether. Unix was possible because of the *C* compiler. Photoshop could hardly have been built on the basis of `GOTO`. The *JavaScript* ecosystem would not have grown without garbage collection.

We shall argue that Turing incompleteness may represent a shift of at least similar magnitude towards less bug-ridden, more manageable software, perhaps enabling new kinds of applications. We shall furthermore argue that this gain in terms of control would come at low-to-no cost in terms of programming language expressivity for many application domains.

1.2 The promise in circumventing Rice's theorem

[Rice 1953] implies that *any* static analysis of Turing complete languages is either very limited or fundamentally incomplete. Conversely, since Rice's proof relies on a reduction to the halting problem, we can hope that well-constructed Turing incomplete languages will support much more in-depth static analysis, such as:

(1) Runtime complexity in memory or time consumption

[1] besides interpreting a Turing complete language and other degenerate cases
[2] with regards to verification and human factors
[3] e.g. there likely will always be demand for manual memory management in some areas

Author's address: Gabriel Pickard, gpickard@uni-osnabrueck.de.

(2) Safety conditions to rule out malware or unauthorized access to data
(3) Correctness of complex program specification beyond mere type safety
(4) Automatically generated code from such specification

These capabilities hold a promise to enable a wide variety of new and improved applications:

(1) Better collaborative sandboxing and control of plugins and app store submissions
(2) Predictable and safe "serverless" cloud functions
(3) Safer and correct distributed computing, including verified *Byzantine fault tolerance*[4].
(4) Safer blockchain smart contracts and static gas billing
(5) Interactive coding environments with rich inferred feedback
(6) Better monitoring and runtime analysis due to simpler trace structure

1.3 Turing complete by default

While there are notable exceptions in query languages (*SQL*, *Datalog*) and proof assistants, Turing completeness has been the default choice for most general purpose programming languages. It is difficult to determine whether the dominance of any technology is due to merit or happenstance and path-dependence. We do know that Turing machines as a concept preceded physical computers. The earlier generations of programming languages preceded the field of strongly normalizing λ calculi and mainly had Turing machines and the closely related *von Neuman* architecture to draw on as theoretical inspiration.

There are many *generative* (cf. [Felleisen et al. 2018]) fundamental algorithms, particularly in numerical computing and data structure manipulation such as sorting [5] for which the most efficient implementation[6] uses unbounded loops (and in-place mutation). Unconstrained loops, recursion and mutability imply Turing completeness (in the absence of advanced types and proof techniques such as loop invariants), hence most early programming languages were Turing complete.

Implementing these kinds of fundamental algorithms efficiently was an important area of focus for the first decades of software engineering history. More recently though, higher-level languages with features such as memory management and persistent data structures have been gaining popularity, sacrificing efficiency for safety and usability. This shift has been made possible by pervasive use of libraries covering numerical computing and data structure manipulation: Individual software engineers often need not concern themselves with the implementation details of e.g. hash maps, sorting or prime factorization.

This shift in the experience of software engineering from small, efficient and "algorithmic" programming to larger and conceptually simpler application development bodes well for the project of making practical Turing incomplete languages: While we propose Turing incomplete language in order to support automated reasoning about correctness and other characteristics of programs, we also contend that *requiring* an understanding of advanced type systems and proof procedures. As such, building up languages based on *structural*, as opposed to *generative* loops *generative* (cf. [Felleisen et al. 2018]) appears promising.

In fact, we conducted a review of 10 source code repositories on GitHub, selected from the first two pages of results for the search terms *"app"* and *"webserver"*, without finding a single procedure that could not be expressed using the simple Turing incomplete constructs we shall introduce in the following.

[4] [Castro et al. 1999]
[5] quicksort, Newton-Raphson, Euclid's algorithm, generational garbage collection to name a few
[6] on a Turing or von Neuman machine

2 HOW TO STAY TOTAL

2.1 Guaranteed termination

If one forces a language to terminate for all inputs, one automatically achieves Turing incompleteness. This is both a desirable and fairly straightforward approach, if not necessarily easy.

There are two perspectives on ensuring termination:

(a) Take existing control structures such as arbitrary loops or recursion and constrain them via types[7], so as to only accept terminating programs.
(b) Remove and replace loops and recursion, as well as any other constructs that might interact with the replacements in order to create accidental Turing completeness.

In the following we shall focus on the latter perspective, since we are seeking to make terminating languages accessible beyond academia. Currently available type systems of total functional languages require in-depth, arcane knowledge which may not be palatable to the mainstream. This is not to say that future formalisms won't be more user friendly. Potential avenues of inquiry may be:

- An updated variant of ACL2[8]
- A meta-inference system based on integrated logic programming capabilities, perhaps somewhat inspired by [Byrd et al. 2012]
- Dependently typed programming with a heavy emphasis on solid defaults, tailored to application programming, perhaps even paired with a generative machine learning system for constructing proof tactics.

2.2 Replacing loops

In order to replace loops and recursion it behooves us to understand the kinds of tasks that programmers apply them to in our target field. Here is a list of common use cases:

- Iterating over elements in a collection, creating a new collection (perhaps filtering elements)
- Iterating through the integers, accessing elements in an existing list and creating a new data structure
- Recursively descending into a tree-like data structure and aggregating a result on the way back up
- Performing recursive arithmetic
- Re-applying an optimization / search-space expansion function until some condition holds

Of the above, all but the last can readily be replaced by a combination of `reduce` and integer ranges.[9] Luckily, most applications never require exploring a search space, hence such functionality could be elided or relegated to an advanced feature, requiring additional user-constructed proofs, in accordance with the maxim *"make easy things easy, and hard things possible"*.

Constructs that could lead to infinite loops must be removed in concert, so as avoid *Turing traps*: to make sure that the presence of one set of features (e.g. higher order functions and

[7] or other meta-inference mechanisms
[8] [Kaufmann and Moore 1996]
[9] Some recursive arithmetic may be difficult to capture this way, though it then is likely of either academic interest or very performance sensitive, in which case other methods apply anyway.

mutable variables) does not undermine the salutary effect of replacing another (e.g. arbitrary recursion → `reduce`-based collection processing).

2.2.1 Functional. A workable set of choices for a functional language include;

- First class functions
- Immutable names and variables, with acyclic data structures
- No recursion (arguably a variant of acyclic data structures)
- `map`,`reduce` and `filter` (with some modifications for handling tree structures)
- List comprehensions and ranges / lazy data structures

Alternatively, a functional programming language may lean on type checking and some scheme of *well founded recursion* (e.g. [Abel and Pientka 2013]).

2.2.2 Imperative. An imperative language could be made Turing incomplete by choosing:

- Mutability
- No first class functions
- No unconstrained `for` or `while` loops
- A modified, `reduce`-style version of `foreach` with a cycle checker
- List comprehensions and ranges / lazy data structures

2.3 Making a pleasant reduce

> "[reduce] is actually the one I've always hated most, because, apart from a few examples involving + or ∗, almost every time I see a `reduce()` call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the `reduce()` is supposed to do." – *Guido van Rossum*[10]

We shall demonstrate the work necessary for user friendly Turing incompleteness in a *Python*-style programming language. Consider the following modification of the `for` operator in python, to accommodate a form of `reduce`:

```
def factorial(n):
    for i in range(n) with state = 1:
        state = state * i

    return state
```

There also can be affordances for processing nested data structures:

```
def flatten(nested_list):
    for item nested in nested_list with result = []:
        result.append(item)

    return result
```

Consider the alternative `unpack` keyword introduced in the following. It enables processing nested data structures with one loop, pushing the parent collection onto the stack and inserting the child collection into the processing flow:

[10] [van Rossum 2005]

```
    def flatten(nested_list):
        for item in nested_list with result = []:
            if type(item) == list:
                unpack;
            else:
                result.append(item)

        return result
```

Finally we present syntax for recursive traversal of trees. If the tree is represented as a nested list, one could could perform traversal in the following way, using the `nested` keyword:

```
def traverse(nested_list):
    for direction of item nested in nested_list with result = "":
        if type(item) == list:
            if direction is descending:
                result += "( "
            else:
                result += " )"
        else:
            result += item
    return result
```

Given a dedicated tree data structure, the `traverse` operator takes two sub-blocks: `down` and `up` for recursive descent and ascent respectively.[11] Note the additional syntax involved in gathering up all the child states in the `up` phase:

```
    def max_height(nested_dict):
        traverse nested_dict with h = 0:
            down k, item:
                print(k, item)
                h += 1
            up k, item for hs:
                h = max(hs.values())

        return h
```

These examples should cover most structural use cases. If other needs turn up, more syntax might be added. A Turing incomplete language may also choose to offer an "escape hatch" to an underlying Turing complete system, particularly useful for implementing efficient data structures.

[11] Integration with `break` and `unpack` keywords (not shown in examples) should be relatively self-explanatory.

3 CONCLUSION

We demonstrated a path to designing terminating languages for the pragmatic programmer, while avoiding Turing traps and maintaining usability. We argued that such avenues of inquiry have been unduly overlooked in the history of computer science and software engineering. Fundamental theorems indicate significant potential for more robust software built with smarter developer tools. We predict that the necessary rigor and clarity alone will benefit programming languages designed to terminate.

3.1 Future work

There are many details to be explored in the choice of language paradigm and design of Turing incomplete control structures. We do not yet know which application domains might benefit the most and how much of a lift enhanced meta-reasoning and static analysis will provide over Turing complete systems. It may very well be that many analysis techniques also work on subsets of existing languages, though it still seems a worthwhile endeavor to check. As language designers we know there is a great difference between equivalence in theory and the particulars in practice.

Existing total languages such as *Agda* and *Coq* already exploit the rich information they have about a program at edit time in an interactive user experience. Transferring such functionality to areas like web programming would be very worthwhile. For that purpose it appears crucial to design a rich system of type statements or some other meta-language palatable to mainstream engineers.

Most importantly, we want to encourage language designers to aim high in automating and assisting the software development process.

REFERENCES

Andreas M Abel and Brigitte Pientka. 2013. Wellfounded recursion with copatterns: A unified approach to termination and productivity. *ACM SIGPLAN Notices* 48, 9 (2013), 185–196.

William E Byrd, Eric Holk, and Daniel P Friedman. 2012. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. 8–29.

Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

EW Dijkstra. 1966. GOTO considered harmful. *Comm. of the ACM* 11, 3 (1966).

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to design programs: an introduction to programming and computing*. MIT Press.

Matt Kaufmann and J Strother Moore. 1996. ACL2: An industrial strength version of Nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*. IEEE, 23–34.

Stephan Kepser. 2002. *A proof of the Turing-completeness of XSLT and XQuery*. Technical Report. Technical report SFB 441, Eberhard Karls Universitat Tubingen.

H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

Guido van Rossum. 2005. The fate of reduce() in Python 3000. https://www.artima.com/weblogs/viewpost.jsp?thread=98196. Accessed: 2020-09-18.