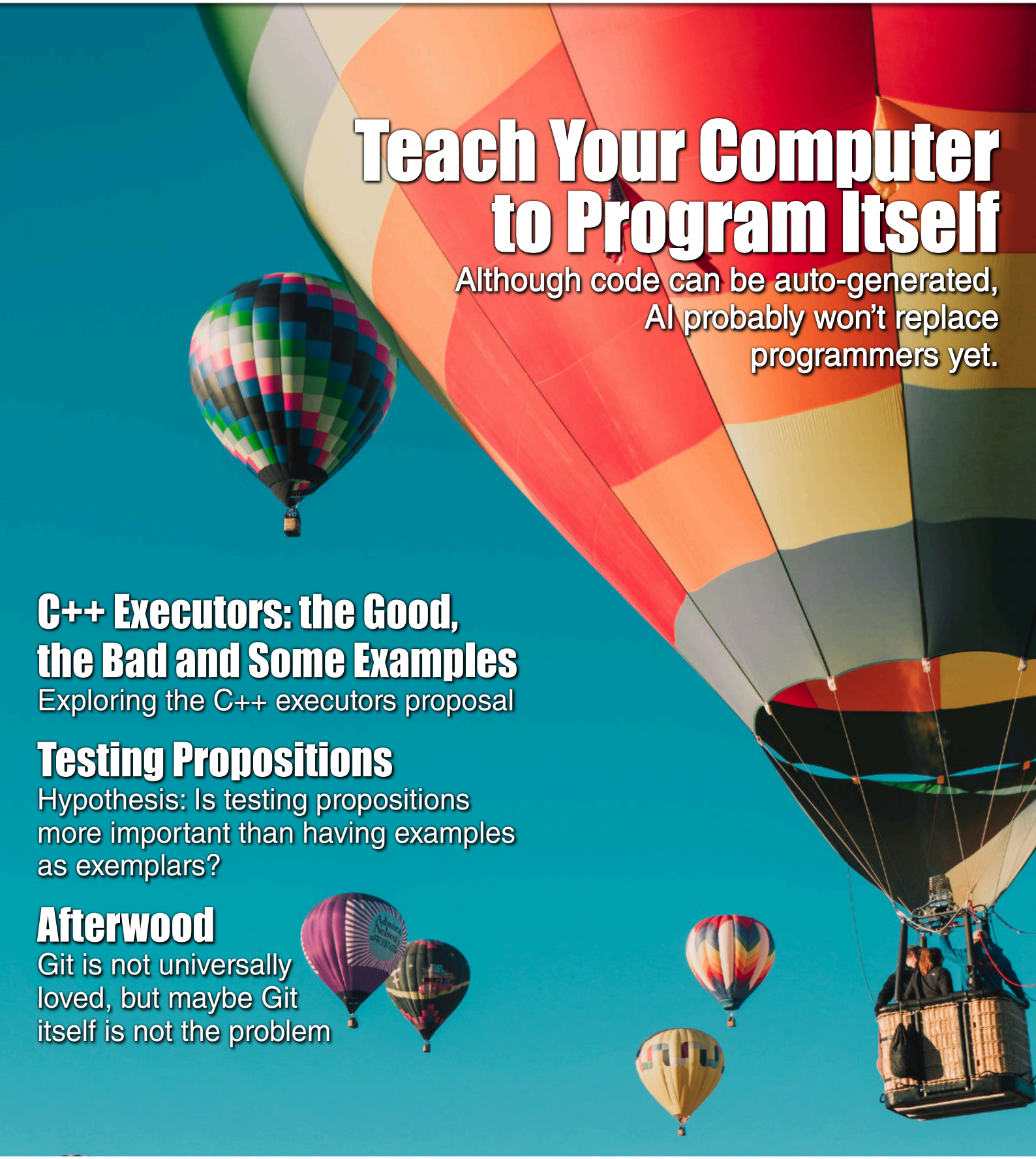
overload 164
AUGUST 2021 £4.50



Teach Your Computer to Program Itself

Although code can be auto-generated, AI probably won't replace programmers yet.

C++ Executors: the Good, the Bad and Some Examples

Exploring the C++ executors proposal

Testing Propositions

Hypothesis: Is testing propositions more important than having examples as exemplars?

Afterwood

Git is not universally loved, but maybe Git itself is not the problem

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 164

August 2021

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Ian Dooley,
on Unsplash.

Copy deadlines

All articles intended for publication in Overload 165 should be submitted by 1st September 2021 and those for Overload 166 by 1st November 2021.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copyrights and Trade Marks

Too Much Typing

Programmers spend an inordinate amount of time typing. Frances Buontempo wonders whether this can be curtailed.

Last time I wrote about using labels and typecasting people. I still don't know what label I'd choose for myself and I don't know what to write about for an editorial either. Besides anything else, my typing has taken a nosedive recently. I really must practise with gtypist or similar. Remote pairing sessions seem to mean I get flustered and can't remember where the letters or symbols are. I suspect some of the trouble comes from listening, looking at someone, typing and thinking simultaneously. Some days it's easier to multitask than others.

Remote pairing or ensemble sessions can take on various forms. Some IDEs allow you to all connect to one session, which avoids the problem of having to push broken code for another team member to pick it up. You could copy code into a chat, but many chat app do very nasty things to whitespace, line endings and the like. I'm using Teams at work currently, which has a propensity to make code appear yellow on a green background when pasted into a chat, aside from messing with the contents. Looking at it makes me feel green, perhaps in sympathy. As an alternative to typing, we can resort to a call. We can even sketch a diagram. I prefer pen and paper or whiteboards, but there are many online collaboration tools. I always then spend far too long trying to work out how to move, focus or add text boxes, all the while muttering "I hate GUIs". The thing about a piece of paper is it has a fixed size and clear edges, which affects what you choose to draw. Constraint can be liberating. A software tool may allow extension in all directions for a very long way. Finding anything afterwards can be a challenge. The old saying goes "A picture is worth a thousand words", but if that picture has over a thousand labels and comments, not to mention arrows and other mystic symbols, words may well have been more succinct. Less is more and drawing clear diagrams is a skill.

Documenting things well is also a skill. I recently noticed when our business analysts ask for documentation they mean write down what was decided, so people know afterwards. As a programmer, when I hear the word documentation, I tend to groan. I have tried to use so many APIs, libraries and the like with pages and pages of instructions. Various things go wrong. Sometimes I am reading the wrong version for the tool I am using. Many cutting edge tools change at a huge pace, so it's easy to waste time trying something that either no longer works or will only work after an upgrade, breaking many other things in the process. Other times, scroll bars are involved. You ever had that thing where you open a man page, start reading, page down, keep reading, notice something interesting you didn't know, try it out, forget what you were trying to do, then remember, scroll further, try to find an example, give up, look on stack overflow, desperate for an example to copy and twiddle with, finally finding one using a totally different tool that you then install? And so it continues. Maybe it's just me? That was, of course, far too many words to illustrate an example we are probably all familiar with. Too many words, people. Think back to documentation you have written. Was it useful for anyone? Have you ever re-read something you've previously written? I usually spot typos or similar when it's too late, however that's not what I'm talking about. Writing can tend to fall into one of two camps – either almost no details, so not much use, or every possible detail but no high level "How to start" or similar. If you have trouble documenting code, or writing blogs or articles, try to notice writing you like and ask yourself why. What's good about it? How is it presented? Are the sections helpful and easy to navigate? This will make your writing better.

Many companies use some kind of wiki to keep their documentation. This is sensible, but trying to find useful information afterwards can be a challenge. It can feel like walking into someone's room, only to find it full of hand-written notebooks. The instructions for baking the cake you were dreaming of may be in there somewhere, but frankly, it might be easy to throw it all out and start over. Wikis and the like need curating. If you don't have a tidy up once in a while, the rot can set in. Most people tend to avoid deleting pages that are no longer relevant, just in case they prove to be important historical documents. Often they can be of interest, but are a distraction if you are trying to find out how to do something. If your system supports showing historical pages and views of pages, maybe trust it as you would trust version control? In fact, how often do we leave, or at least see, commented out code in a code base? If it's not used, delete it. I guess it can take some skill to find code in version control history that no longer exists, especially if the file it was in has gone too, but it's not impossible. And, it might be possible to write code to do what's required if you can't find an old version. Less clutter can make it easier to think.

I am reliably informed Rich Hickey, the inventor of Clojure once said, "Programming is not about typing, it's about thinking." On the face of it this sounds true, yet sometimes we need to try out code to see what it does. No amount of thinking will catch all the edge cases etc. In my career, I have often found myself in a meeting discussing a bug or feature and how to approach it, while people draw on white boards, speculate and the like. Frequently, we go nowhere near the code, guess, misremember and then when we come to write or fix the code, realise many further issues have not been taken into account. Don't be shy of opening up the code while people talk and doing science. See what actually happens. It's the only way. No amount of writing a wish list in a Jira will "Make It So".

Talking of Clojure, as I'm sure you know this dynamic language is a Java-based dialect of Lisp. Dynamic languages offer what is referred to as duck typing, in that you can duck typing in the types, and furthermore "If it

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

quacks like a duck, it is a duck". Wikipedia tells me that though the Clojure "type system is entirely dynamic, recent efforts have also sought the implementation of gradual typing". [Wikipedia-1]. I understand dynamic typing: the type checks happen at run time, though it can be possible to flush out problems with a linter without waiting until something blows up. This can be odd to get used to if you come from a static typing background. I had not come across the term gradual typing before, so had to read a lot of the internet. My executive summary is it's some kind of mixed model, like C#, although compiled and statically type checked, since the introduction of a dynamic type some things may blow up at runtime instead of compile time. Other gradually typed languages are available, including J, Objective-C and Typescript. I've had a run in with Typescript recently. Initially it felt like too much typing, pun intended, though one or two ideas, such as being able to coerce some JSON into an interface actually saved some physical typing. I still have mixed feelings.

Whenever I've spent time using a dynamic language, allowing me to type `x = 10` and the like, switching to C++ or similar and having to add an `int` in as well seems unnecessary. I do eventually get over myself, but the paradigm shift takes a while. Occasionally, I find myself in what seems like the worst of both worlds. In particular, I am referring to Python's type hints. These provide optional annotation for code. There were introduced by PEP484 [Python]. Initially, I was curious and tried them out. The PEP states

> While these annotations are available at runtime through the usual `annotations` attribute, no type checking happens at runtime. Instead, the proposal assumes the existence of a separate off-line type checker which users can run over their source code voluntarily. Essentially, such a type checker acts as a very powerful linter.

It mentions mypy, an optional type checker, as the inspiration. The type hints felt like a vast amount of extra effort for not much benefit. Many people often tell me it helps their IDE offer suggestions for auto-complete and similar. That's as may be, but I tend to use Vim for Python, though I understand not everyone does. I have heard tale of type hints being used on large code bases, and the main thing I recall was the type hints finding places where a function would return `None` because a snakey mess of `if`/`else` statements missed a `return`. Such an error can be picked up by a linter, so I didn't come away inspired. Since type hints are not all or nothing, you can add them in places you feel might be beneficial. I'm still not convinced of their value, but feel free to have a play and report back. There are plenty of details online, but Real Python [RealPython] offer a good overview.

In what I consider a similar vein, Stepanov and McJones added a template constraint, requires to their code listings in their *Elements of Programming* book [Stepanov09]. The C++ community was starting to talk about concepts at the time so it made sense. The start of the book told us the requires clause could be an "expression built up from constant values, concrete types, formal parameters, applications of type attributes and type functions, equality on values and types, concepts, and logical connectives." They thereby described properties of a type. If you turn to Appendix B, you will find

```
#define requires(…)
```

which is somewhat underwhelming. I do see the benefit in what became concepts in C++. At very least, they can make error messages clearer. However, C++ being C++ means they can end up getting quite complicated. Trying to explain why you sometimes need to say `requires requires` is a challenge [StackOverflow].

Even though C++ is statically type checked, you can achieve what feels a bit like duck typing with templates, though things will go wrong at compile time rather than run time. Freedom to take any type and call a method on it, without having to build up a class hierarchy or other boilerplate code can be liberating. This allows us to pass iterators into an algorithm and do something with each element, regardless of the data structure the iterators came from. You could suggest C#'s LINQ has a similar feel, but Stepanov's genius idea was to separate the data structures and algorithms. C++ also allows use of any, variant, optional and similar. Claiming something is of type any feels like an oxymoron, but these can be very useful.

I started by talking about the physical act of typing, and then strayed into the world of types of objects in code. I could stray into category and type theory, but will spare you that. Let's think about typing lots of code, or perhaps worse, attempting to grok a long function. Sometimes, we see a very long select statement or snakey mess of `if`s and `else`s. I personally believe it is usually possible to collapse these down a bit to something that's easier to follow. Even pulling out of few statements into their own function can leave less detail to look at in one place, though they have moved elsewhere. Sometimes swapping a for loop for an algorithm makes the code more succinct. Opinions can be divided at this point. Sometimes code is obviously messy, sometimes it is quite neat, but sometimes it is 'too cute'. You spend longer figuring out what it does when it's a one liner than you would have done with a `for` loop. Try out the cute code, by all means, but check you still understand it the next day.

Managing to be succinct and clear is difficult, but worth a try. Lengthy rambles take too long to read and I'm now running out of space, so shall end with a Pascal quote: "I would have written a shorter letter, but I did not have the time." [Wikipedia-2] ■

References

[Python] Python: https://www.python.org/dev/peps/pep-0484/

[RealPython] Real Python: https://realpython.com/python-type-checking

[StackOverflow] For an example, see https://stackoverflow.com/questions/54200988/why-do-we-require-requires-requires

[Stepanov09] Alexander Stepanov and Paul McJones (2009) *Elements of Programming* ISBN 9780321635372, Addison Wesley. Available from: http://elementsofprogramming.com/

[Wikipedia-1] Clojure: https://en.wikipedia.org/wiki/Clojure

[Wikipedia-2] Pascal (Letter XVI), according to https://en.wikipedia.org/wiki/Lettres_provinciales

C++ Executors: the Good, the Bad, and Some Examples

Executors aim to provide a uniform interface for work creation. Lucian Radu Teodorescu explores the C++ executors proposal.

One of the most anticipated features in the upcoming C++ standard is the support for executors. Through executors, we hope to get a higher level set of abstractions for doing concurrent/parallel work. If everything goes well, the executors will land in C++23.

This article aims at providing a critical perspective on the proposed additions to the C++ standard. We specifically look at the proposal entitled P0443R14: A Unified Executors Proposal for C++ [P0443R14], but we touch on a few connected proposals as well.

The article provides a brief tour of the proposal with a couple of examples, and then jumps to a critical analysis of various points of the proposal. This critical analysis tries to bring forward the strong points of the proposal, as well as the weak points. The hope is that by the end of the article the reader will have a better understanding of the proposal, and of its pros and cons.

A brief tour of the proposal

P0443R14 has some sort of internal unity, but at a more careful reading one can divide the proposal in two main parts:

- support for executors
- support for senders and receivers

The executors part doesn't need to be coupled with senders and receivers, while senders and receivers can be theoretically based on slightly different executor semantics. Furthermore, conceptually, they solve different problems. Moreover, the paper itself makes the distinction between these two parts. Thus, it makes sense for us to treat them separately as well.

The libunifex library [libunifex] is a prototype implementation for the proposal, containing almost everything from the proposal, and much more. The authors of the library were also contributors to the proposal. My own Concore library [concore] also has support for the main concepts in the proposal.

Executors support

An executor is a work execution interface [P0443R14]; it allows users to execute generic work. The following code shows a simple usage of an executor:

```
executor auto ex = ...;
execute(ex, []{ cout << "Hello, executors!\n"; });
```

If we have an executor object (i.e., matching the **`executor`** concept), then we can just execute work on it. The work is some form of an invokable entity. We have decoupled the work from the context in which it is executed.

That's it! Things are that simple!

The P0443R14 paper describes an executor that can be obtained from a **`static_thread_pool`** object, proposed to be added to the standard library. As the name suggests, this would add support for thread pools. The users can then create thread pools and pass work to be executed on these pools. Here is one simple example that will execute work on one of the threads inside the thread pool:

```
std::static_thread_pool pool(4);
execution::execute(pool.executor(),
  []{ cout << "pool work\n"; }
```

The executor concept and the thread pool by themselves are directly usable for building concurrent applications. But, more importantly, it is easy to create other executors. The paper exemplifies how one could write an **`inline_executor`** that just executes the work on the current thread, similarly to calling a function. To define a new executor, the user must provide an **`execute`** method that takes work (technically this is a customisation-point-object that can take other forms too, but we'll try to provide a simplified view) and a way to compare the executors for equality.

To showcase how easy it is to define an executor, we'll just copy the definition of **`inline_executor`** given by P0443R14 here:

```
struct inline_executor {
  // define execute
  template<class F>
  void execute(F&& f) const noexcept {
    std::invoke(std::forward<F>(f));
  }
  // enable comparisons
  auto operator<=>(const inline_executor&)
    const = default;
};
```

Piece of cake!

An executor can have multiple properties attached to it. For example, an executor can be blocking to callers (like **`inline_executor`**) or non-blocking (like the executor from **`static_thread_pool`**). Another example would be the property that indicates the type of allocator that the executors use.

The support for properties is actually introduced by paper P1393R0: A General Property Customization Mechanism [P1393R0] (and the executors paper builds on it). At this point, there is no consensus on whether this would move forward or not. But, even if this proposal doesn't move forward, executors are still usable.

Senders and receivers

Here, things get a bit more complicated. The paper defines the following (major) concepts:

- **`sender`**: work that has not been scheduled for execution yet, to which one must add a continuation (a receiver) and then "launch" or enqueue for execution [P0443R14]
- **`receiver`**: is a callback object to receive the results from a sender object
- **`scheduler`**: a factory of single-shot senders [P0443R14]

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

One way to think of receivers is to consider them as generalised callbacks; they are called to announce the completion (successful or not) of another operation

- **operation_state**: the state of an asynchronous operation, the result of connecting a sender with a receiver

Besides this, the paper also defines a few customisation points (CPOs):

- **set_value**: applied to a **receiver** object, it is called by the sender to notify the receiver about the completion of the work, with some resulting values; part of the semantic of being a **receiver**.
- **set_done**: applied to a **receiver** object, it is called whenever the work in the sender has been cancelled; part of the semantic of being a **receiver**.
- **set_error**: applied to a **receiver** object, it is called whenever there was an error with the work in the sender; part of the semantic of being a **receiver**.
- **connect**: applied to a **sender** object and passing in a **receiver**, it is called to connect the two objects, resulting in an object compatible with **operation_state**; part of the semantic of being a **sender**.
- **start**: applied to an **operation_state** object, it is called to logically start the operation (as resulting from connecting the sender and the receiver); part of the semantic of being an **operation_state**.
- **submit**: applied to a **sender** object, it is called to combine it with a receiver object and immediately start (at least logically) the resulting operation.
- **schedule**: applied to a **scheduler** object, it is called to return a 'single-shot' sender (this will call the receiver with no value); part of the semantics of being a **scheduler**.

Customisation point objects are generalisations over functions. The standard provides some free functions for them, but allows (and sometime requires) the user to customise their behaviour. The key point that one should distinguish between two variants of functions with the same name: one that the framework provides and one that the user needs to provide.

Let's take a simple example. To define a receiver, the user is required to define a **set_done** method/function. The framework also defines an **execution::set_done** function that can be called by the senders. Something like the following:

```
struct my_sender {
...
execution::set_done(recv);
...
};
```

Yes, things can be a bit confusing, but I think that with enough exposure people will get used to this. It's similar to the **std::begin()** function versus the **begin()** method defined in containers like **std::vector**.

Using this, one can write code that represents asynchronous computations as chains between senders and receivers. Listing 1 presents an example.

At the beginning of the listing, there are two types that model the **receiver** concept; they can be used to be notified about the completion of some asynchronous operation. **set_value()** is called when the previous computation is successful, **set_done()** is called when the operation was cancelled, and **set_error()** is called whenever there was an error in the previous computation. One way to think of receivers is to consider them as generalised callbacks; they are called to announce the completion (successful or not) of another operation.

```
struct my_recv0 {
  void set_value() { cout << "impulse\n"; }
  void set_done() noexcept {}
  void set_error(exception_ptr) noexcept {}
};
template <typename T>
struct my_recv {
  void set_value(T val) { cout << val << endl; }
  void set_done() noexcept {}
  void set_error(exception_ptr) noexcept {}
};
static_thread_pool pool{3};
auto sched = pool.scheduler();
// single-shot sender
auto sndr1 = schedule(sched);

auto op_state = connect(sndr1, my_recv0{});
start(op_state); // prints "impuslse"

// computation with P1897R3 abstractions
auto f = [](int x) { return 3.141592 * x; };
auto print = [](double x) { cout << x; };
auto sndr = just(2)
          | on(sched)
          | transform(f);
// prints the result 2*3.141592 asynchronously
submit(move(sndr), my_recv<double>{});
```

Listing 1

Schedulers are objects that can generate 'single-shot' senders. These single-shot senders are just sending *impulses* to receivers downstream, without sending any information to them. This is why we can connect such a sender (**sndr1** with a receiver that doesn't take any value).

A sender can be bound to a receiver only once, so they can be considered short-lived: they only live for one computation to go through them. This is why it is important for the framework to allow easy creation of senders.

The example in Listing 1 shows how a sender (**sndr1**) can be connected to a receiver (in our case, an object of type **my_recv0**). The connection between a sender and a receiver is captured by an object that models the **operation_state** concept. This concept corresponds to an asynchronous operation (i.e., tasks). Senders by their own, and receiver by their own, cannot be considered tasks. The only thing that one can do with an **operation_state** object is to start it. This is done by calling the **start()** customisation-point-object.

In the last few lines of Listing 1, we present how a computation can be represented using the sender algorithms introduced by [P1897R3]. A

Just like we use serializers to handle some types of restrictions, we can generalise them and use executors for solving all types of restrictions

sender algorithm is a function that returns a sender (or, something that, when combined with a sender, returns another sender). In our case, `just(2)` returns a sender that will just push the value `2` to its receiver. The `on(sched)` returns an object that when combined with the previous sender generates a sender that runs the previous computation on the given scheduler. In our case, we are indicating that everything needs to be executed on our thread pool. Finally, `transform(f)`, when combined with the previous sender, will return a sender that will execute the given function to transform the received input.

Putting all these together, we obtain a sender that will multiply `2` with `3.141592` on our thread pool.

In the last, line, instead of connecting it to a receiver and calling `start` on the resulting operation state, we call `submit`. This is a shorter version of the above.

This was a quick summary of the important parts of the proposal. More examples and discussions can be found in other resources on the Internet. Two presentations that explain executors and senders/receivers in more detail can be found at [Hollman19] and [Niebler19].

A critique of the executors proposal

We will organise this section into a series of smaller inquiries into various aspects of the proposals. We will label each of these inquiries with *Good* and *Bad*. This labelling scheme is a bit too polarising, but I'm trying to convey a summary of the inquiry. Especially the *Bad* label is maybe too harsh. This label must definitely NOT apply transitively to the people behind this proposal; they worked really hard to create this proposal; this thing is not an easy endeavour (see [Hollman19] for some more insights into the saga of executors).

Good: Support for executors

I can't find words to express how good it is to have executors as a C++ core abstraction. It allows one to design proper concurrent algorithms, while separating them from the actual concurrency primitives.

The same application can have multiple concurrency abstractions, and we can design algorithms or flows that work with any of them. For example, one might have one or more thread pools as described by this proposal, or can have executors from third-party libraries (Intel oneAPI, Grand Central Dispatch, HPX, Concore, etc.). Moreover, users can write their own executors, with reasonable low effort. For example, in my C++Now 2021 presentation [Teodorescu21b] I've showcased how one can build a *priority serializer* (structure that allows executing one task at a time, but takes the tasks in priority order) – the implementation was under 100 lines of code.

The reader might know that I often talk about serializers as concurrent abstractions that help in writing better concurrent code, simulating locks behaviour while avoiding the pitfalls of the locks (see [Teodorescu21a], [Teodorescu20a] or [Teodorescu20b]). Serializers are also executors. Generalising, we can introduce concurrent abstractions as executors.

Moreover, the executors can be easily composed to provide more powerful abstractions. For example, a serializer (which is an executor) can be parameterised with one (or even two) executors, which specifies the actual mechanism to execute the tasks.

If there is one thing that the readers remember from this article, I hope it is that executors are a good addition to the standard.

Good: Every concurrent problem can be specified using only executors

As argued in [Teodorescu20a], every concurrent problem can be expressed as a set of tasks and restrictions/dependencies between them. I'll not try to give all the formal details here, but we can prove that the restrictions/dependencies of the tasks can be represented using different executors. Just like we use serializers to handle some types of restrictions, we can generalise them and use executors for solving all types of restrictions.

To achieve this, we can add various labels (with or without additional information) to the tasks, and we define rules that infer the restrictions/dependencies based on these labels. For example, dependencies can be encoded with a particular label that contains some ordering number. Restrictions like those found in serializers can be implemented with labels that mark that certain tasks are mutually exclusive.

For each type (or better, for each instance) of a label, we can create an executor that will encode the restrictions represented by that label. This way, for each type of restriction that we have, we will have an executor to encode it.

If we have all these, then one just needs to compose the executors in the proper way to ensure the safety of the application. The composition might pose some problems in terms of performance, but these performance problems can always be solved by specialising the executors.

Executors are fundamental building blocks for writing concurrent programs.

Good: Proposal provides a way to encode computations

Between a sender and a receiver pair, we can encode all types of computations. As an `operation_state` object can act like a task, and as we can represent all computations with tasks, it means that we can represent all types of computations with senders and receivers.

Beyond this, the proposal seems to encourage the expression of computations as chains of computations, in which values are passed from a source (an initial sender) to a final receiver. This indicates a tendency towards functional style expression of computations. This sounds good.

Bad: Proposal seems to restrict how computations can be expressed

The above point can be turned around as a negative. It feels awkward in a predominant-imperative programming language to allow expression of concurrency in a functional style. Functional style sounds good to me, but there are a lot of C++ programmers that dislike functional style.

Probably my biggest complaint here is that where to place computations is confusing: in the senders or in the receivers. And the naming here doesn't help at all. Let's say that one wants an asynchronous action to dump to disk

the state of some object. How should one design this? The problem of dumping some state to disk doesn't properly map to the sender and receiver concept; there is nothing to send or to receive.

There is no way one can directly put this computation into an `operation_state` object, so one needs to choose between a sender and a receiver. If we were to look where the proposal puts most of the computation, we would end up with the idea that the dumping code needs to be placed inside a sender; the proposal also states "a sender represents work [that has not been scheduled for execution yet]" [P0443R14], so this seems like the reasonable thing to do. But writing custom senders is hard (see below); moreover, we need to bind to it a dummy receiver for no purpose.

An easier alternative is to put the computation in the receiver. But that goes against the idea of "a receiver is simply callback" that is used by the proposal to describe receivers. Creating a receiver that dumps the data to disk is relatively easy. But, in addition to that, one needs to also connect the receiver with a single-shot sender and lunch the work to the execution. The presence of the sender should not be needed.

One can easily solve the same problem with executors.

Bad: The concepts introduced are too complex

Executors are simple: you have an executor object, and you provide work to it. This can be easily taught to C++ programmers.

On the other hand, teaching senders and receivers is much harder. Not only there are more concepts to teach, but also the distinction between them is unclear.

Here are a few points that can generate confusion:

- `scheduler`s seem to be an unneeded addition; we can represent the same semantics with just executors and senders
- `operation_state` objects seem more natural (as they correspond to tasks), but there is no way for the user to directly write such objects
- if `operation_state` objects cannot be controlled by the user, then maybe they shouldn't be exposed to the user
- `submit` seems to be a nice simplification of `connect` and `start`, but having both variants adds confusion
- considering that both `submit` and the pair `connect` and `start` can be customised by the user, one can end up in cases in which `submit` is not equivalent with `connect`/`start`; this means that we have ambiguous semantics

Looking closely at the proposal, we find circular dependencies between the proposed concepts and customisation-point-objects (via wrapper objects). For example, `connect` CPO can be defined via the `as-operation` wrapper in terms of `execute` CPO. But then, `execute` is defined in terms of `submit` CPO, which is also defined in terms of `connect` (via the `submit-state` wrapper object). Circular dependencies are a *design smell*.

The point is that it's really hard to grasp senders and receivers.

Good: Receivers have full completion notifications

Many threading libraries are more concerned about ensuring the execution of asynchronous work, and don't consider error cases much. The way that senders and receivers are conceived, if the user puts the work in the sender, there will be a notification that indicates in which way the work was completed: successfully, with an error, or it was cancelled.

Good error handling is always desired.

Good: Easy to write receivers

As shown in Listing 1, it's relatively easy to write receivers. If one provides three methods – `set_value`, `set_done` and `set_error` – then one has defined a receiver. There are multiple ways in which the receiver can be defined, and there might be different types of receivers, but the idea remains simple.

If users only need to write receivers, then using senders and receivers would probably be an easy endeavour.

Bad: Hard to write senders

On the other hand, it's hard to write even simple senders. For a sender, one needs to write a `connect` method/function (or maybe `submit`?). This gets a receiver and has to generate an `operation_state` object, which should be compatible with the `start` CPO. On top of these, templates, tag types, move semantics and exception specifications will make this much harder.

But this is not the complete picture. The proposal encourages composition of senders; thus one should write sender algorithms instead of simple senders. That is, algorithms that take one sender and return another sender. If one sender receives a signal from another sender, it should do its corresponding work and notify the next receiver. This means that the user also needs to write a receiver for the previous sender, and ensure that the flow is connected from the previous sender to the next receiver.

Listing 2 provides an example of a sender algorithm that just produces a value when invoked by the previous sender. I doubt that the reader would consider this easy to write, even with this relatively simple case we are covering.

Bad: Hard to extend the senders/receivers framework

As mentioned above, the proposal envisions extensibility through sender algorithms, similar to the one discussed above. This directly implies that the framework is hard to extend.

This can be compared with executors, which are relatively easy to extend. It's not hard to create a class that models the `executor` concept.

Bad: No monadic bind

One solution to the extensibility problem was a monadic bind. That is, provide a way in which one can create new senders by providing a function with a certain signature. Although monads are sometimes considered hard to grasp, they are proven to be useful for extending the operations on certain structures.

It is worth noting that the senders framework has the potential to use monads as follows. First, `msender<T>` encodes the concept of a sender that connects to receivers that take object `T` as input. For the *type converter* part of the monad, it is easy to find a transformation from an object `T` to `msender<T>` – this is actually provided by the sender `just()` defined by [P1897R3]. The missing operation is something that would have a signature like:

```
template <typename T1, typename T2>
msender<T2> bind(const msender<T1>& x,
  function<msender<T2>(T1)> f);
```

The reader should note that [P1897R3] provides a relatively similar function (imperfect translation):

```
template <typename T1, typename T2>
msender<T2> transform(const msender<T1>& x,
  function<T2(T1)> f);
```

But this is not the same. In the first case the received function object is of type `T1` → `msender<T2>`, while in the second case has the kind `T1` → `T2` One cannot provide the same extensibility with `transform` as with the `bind` function. If the `bind` function corresponds to monoids, the `tranform` function corresponds to *functors* (using terminology from category theory). Of the two, monads are good at composition.

Bad: Cannot express streams with senders/receivers

One might think that senders and receivers are good at representing data streams (i.e., reactive processing, push model). In such a model, one would have sources of events (or values). Then, one can attach various transformations on top of these sources to create channels that transform the input events/values so that they can be properly processed.

The chains of transformations can also be created with senders and receivers, but unfortunately such a channel can only propagate one value

```
// Receiver of void, and sender of int
template <typename S>
struct value_sender {
  int val_;
  S base_sender_;

  value_sender(int val, S&& base_sender)
    : val_(val)
    , base_sender_((S &&) base_sender) {}
  template <template <typename...> class Tuple,
    template <typename...> class Variant>
  using value_types = Variant<Tuple<int>>;
  template <template <typename...> class Variant>
  using error_types = Variant<std::exception_ptr>;
  static constexpr bool sends_done = true;

  template <typename R>
  struct op_state {
    struct void_receiver {
      int val_;
      R final_receiver_;

      void_receiver(int val, R&& final_receiver)
        : val_(val)
        , final_receiver_((R &&) final_receiver)
          {}
      void set_value() noexcept {
        execution::set_value((R &&)
        final_receiver_, val_); }
      void set_done() noexcept {
        execution::set_done((R &&)
        final_receiver_); }
      void set_error(std::exception_ptr e)
        noexcept { execution::set_error((R &&)
        final_receiver_, e); }
    };
    typename detail::connect_result_t<S,
      void_receiver> kickoff_op_;
    op_state(int val, R&& recv, S base_sender)
      : kickoff_op_(execution::connect((S &&)
        base_sender, void_receiver{
          val, (R &&) recv})) {}
    void start() noexcept {
      execution::start(kickoff_op_); }
  };
  template <typename R>
  op_state<R> connect(R&& recv) {
    return {val_, (R &&) recv
      , (S &&) base_sender_};
  }
};
```

Listing 2

through it. For each value that needs to be propagated, a new channel of transformation needs to be created.

This is unfortunate as data stream programming can be an efficient way of solving some concurrent problems.

Bad: Too much templatised code

I'm not going to dwell too much on this topic. The proposal advocates highly templatised code, which will result in increased compilation time for all the code that uses it. If most C++ software were to use this as the fundamental basis for concurrency, then the overall compilation time for all the programs would increase considerably.

The reader should note that, for proper concurrency, the executors would have to move work packages between threads. This implies that at some point there needs to be some type-erasure. It's a pity that this type-erasure is not at a higher level.

Conclusions

This article tried to provide a critique of the executors proposal for the upcoming C++ standard. As with a lot of such critiques in our field, we cannot be fully objective. Software engineering is based on compromises, and the tendency to choose one alternative over another makes us more subjective than we would want to be. But, even if we can't achieve objectivity, such a critique can serve to highlight some nuances of the critiqued object. I have tried to be as objective as I can in this article, but, perhaps, my biases found their way through. However, it is my hope that the reader will find some help in all this endeavour.

As it is already mentioned in the proposal, P0443R14 has two main parts: one that introduce executors, and one that introduce senders and receivers.

Overall, I find the executors part to be a needed addition to the C++ language, Moreover, it's simple to use and very extensible.

For the senders and receivers part of the proposal, I have formulated some objections. They are more complex than they need to be and not as extensible. Probably the best way forward for the standard committee is to split the proposal in two parts and consider them separately for inclusion in the C++ standard.

Looking at the overall proposal, the simple presence of executors makes it worthwhile. C++ can move towards using higher level abstractions for concurrency, abstractions that need executors as their fundamentals. For example, the parallel algorithms would greatly benefit from executors. ■

References

[concore] Lucian Radu Teodorescu, Concore library, https://github.com/lucteo/concore

[Hollman19] Daisy Hollman, 'The Ongoing Saga of ISO-C++ Executors', *C++Now 2019*, https://www.youtube.com/watch?v=iYMfYdO0_OU

[libunifex], Facebook, libunifex library. https://github.com/facebookexperimental/libunifex

[Niebler19] Eric Niebler, Daisy Hollman, 'A Unifying Abstraction for Async in C++', *CppCon 2019*, https://www.youtube.com/watch?v=tF-Nz4aRWAM

[P0443R14] Jared Hoberock et al., 'P0443R14: A Unified Executors Proposal for C++', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html

[P1393R0] David Hollman et al., 'P1393R0: A General Property Customization Mechanism', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html

[P1897R3], Lee Howes, 'P1897R3: Towards C++23 executors: A proposal for an initial set of algorithms', http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1897r3.html

[Teodorescu21a], Lucian Radu Teodorescu, 'Threads Considered Harmful', https://www.youtube.com/watch?v=_T1XjxXNSCs

[Teodorescu21b], Lucian Radu Teodorescu, 'Designing Concurrent C++ Applications', https://www.youtube.com/watch?v=nGqE48_p6s4

[Teodorescu20a] Lucian Radu Teodorescu, 'The Global Lockdown of Locks', *Overload* 158, August 2020, available from https://accu.org/journals/overload/28/158/teodorescu/

[Teodorescu20b] Lucian Radu Teodorescu, 'Concurrency Design Patterns', *Overload* 159, October 2020 available from https://accu.org/journals/overload/28/159/teodorescu/

Testing Propositions

Is testing propositions more important than having examples as exemplars? Russel Winder considers this hypothesis.

With the rise of test-driven development (TDD) in the 1990s as part of the eXtreme Programming (XP) movement, the role of example-based testing became fixed into the culture of software development[1]. The original idea was to drive development of software products based on examples of usage of the product by end users. To support this Kent Beck and others at the centre of the XP community created test frameworks. They called them unit test frameworks presumably because they were being used to test the units of code that were being constructed. This all seemed to work very well for the people who had been in on the start of this new way of developing software. But then XP became well-known and fashionable: programmers other than the original cabal began to claim they were doing XP and its tool TDD. Some of them even bought the books written by Kent Beck and others at the centre of the XP community. Some of them even read said books.

Labels are very important. The test frameworks were labelled unit test frameworks. As all programmers know, units are functions, procedures, subroutines, classes, modules: the units of compilation. (Interpreted languages have much the same structure despite not being compiled per se.) Unit tests are thus about testing the units, and the tools for this are unit test frameworks. Somewhere along the line, connection between these tests and the end user scenarios got lost. Testing became an introvert thing. The whole notion of functional testing and 'end to end' testing seemed to get lost because the label for the frameworks were 'unit test'.

After a period of frustration with the lack of connection between end user scenarios and tests, some people developed the idea of acceptance testing so as to create frameworks and workflows. (Acceptance testing has been an integral part of most engineering disciplines for centuries; it took software development a while to regenerate the ideas.) FitNesse [FitNesse] and Robot [Robot] are examples of the sort of framework that came out of this period.

However, the distance between acceptance testing and unit testing was still a yawning chasm[2]. Then we get a new entrant into the game, behaviour-driven development (BDD). This was an attempt by Dan North and others to recreate the way of using tests during development. The TDD of XP had lost its meaning to far too many programmers, so the testing frameworks for BDD were called JBehave, Cucumber, etc. and had no concept of unit even remotely associated with them.

Now whilst BDD reasserted the need for programmers and software developers to be aware of end user scenarios and at least pretend to care about user experience whilst implementing systems, we ended up with even more layers of tests and test frameworks.

And then came QuickCheck [QuickCheck], and the world of test was really shaken up: the term 'property-based testing' became a thing.

QuickCheck [Hackage] first appeared in work by John Hughes and others in the early 2000s. It started life in the Haskell [Haskell] community but has during the 2010s spread rapidly into the milieus of any programming language that even remotely cares about having good tests.

Example required

Waffling on textually is all very well, but what we really need is code; examples are what exemplify the points, exemplars are what we need. At this point it seems entirely appropriate to make some reuse, which, as is sadly traditional in software development, is achieved by cut and paste. So I have cut and paste[3] the following from a previous article for *Overload* [Winder16]:

> For this we need some code that needs testing: code that is small enough to fit on the pages of this august journal, but which highlights some critical features of the test frameworks.
>
> We need an example that requires testing, but that gets out of the way of the testing code because it is so trivial.
>
> We need factorial.
>
> Factorial is a classic example usually of the imperative vs. functional way of programming, and so is beloved of teachers of first year undergraduate programming courses. I like this example though because it allows investigating techniques of testing, and allows comparison of test frameworks.
>
> Factorial is usually presented via the recurrence relation:
>
> $$f_0 = 1$$
> $$f_n = nf_{n-1}$$

This is a great example, not so much for showing software development or algorithms, but for showing testing[4], and the frameworks provided by each programming language.

Given the Haskell heritage of property-based testing, it seems only right, and proper, to use Haskell for the first example. (It is assumed that GHC 7.10 or later (or equivalent) is being used.)

Russel Winder Ex-theoretical physicist, ex-UNIX system programmer, ex-academic. Now an ex-independent consultant, ex-analyst, ex-author, ex-expert witness and ex-trainer. Was interested in all things parallel and concurrent. And build. Was actively involved with Groovy, GPars, GroovyFX, SCons, and Gant. Also Gradle, Ceylon, Kotlin, D and bit of Rust. And lots of Python especially Python-CSP.

1. Into the culture of cultured developers, anyway.
2. Yes there is integration testing and system testing as well as unit testing and acceptance testing, and all this has been around in software, in principle at least, for decades, but only acceptance testing and unit testing had frameworks to support them. OK, technically FitNesse is an integration testing framework, but that wasn't how it was being used, and not how it is now advertised and used.
3. Without the footnotes, so if you want those you'll have to check the original. We should note though that unlike that article of this august journal, this is an August august journal issue, so very august.
4. OK so in this case this is unit testing, but we are creating APIs which are just units so unit testing is acceptance testing for all intents and purposes.

it seems to be idiomatic to have the type signature ... as a check that the function implementation is consistent with the stated signature

```
module Factorial(iterative, naïveRecursive,
  tailRecursive) where
exceptionErrorMessage = "Factorial not defined for
negative integers."

iterative :: Integer -> Integer
iterative n
    | n < 0 = error exceptionErrorMessage
    | otherwise = product [1..n]

naïveRecursive :: Integer -> Integer
naïveRecursive n
    | n < 0 = error exceptionErrorMessage
    | n == 0 = 1
    | otherwise = n * naïveRecursive (n - 1)

tailRecursive :: Integer -> Integer
tailRecursive n
    | n < 0 = error exceptionErrorMessage
    | otherwise = iteration n 1
    where
      iteration 0 result = result
      iteration i result = iteration (i - 1)
        (result * i)
```

Listing 1

Haskell implementation...

There are many algorithms for realizing the Factorial function: iterative, naïve recursive, and tail recursive are the most obvious. So as we see in Listing 1 we have three realizations of the Factorial function. Each of the functions starts with a type signature followed by the implementation. The type signature is arguably redundant since the compiler deduces all types. However, it seems to be idiomatic to have the type signature, not only as documentation, but also as a check that the function implementation is consistent with the stated signature. Note that in Haskell there are no function call parentheses – parentheses are used to ensure correct evaluation of expressions as positional arguments to function calls. It is also important to note that in Haskell functions are always curried: a function of two parameters is actually a function of one parameter that returns a function of one parameter. Why do this? It makes it really easy to partially evaluate functions to create other functions. The code of Listing 1 doesn't make use of this, but we will be using this feature shortly.

The `iterative` and `naïveRecursive` implementations are just matches with an expression: each match starts with a `|` and is an expression of Boolean value then a `=` followed by the result expression to evaluate for that match expression. Matches are tried in order and `otherwise` is the 'catch all' "Boolean" that always succeeds; it should, of course, be the last in the sequence. The `error` function raises an exception to be handled elsewhere. The `tailRecursive` function has a match and also a 'where clause' which defines the function `iteration` by pattern matching on the parameters. The 'where clause' definitions are scoped to the function of definition[5,6].

...and example-based test

Kent Beck style TDD started in Smalltalk with sUnit[7] and then transferred to Java with JUnit[8]. A (thankfully fading) tradition seems to have grown that the first test framework in any language is constructed in the JUnit3 architecture – even if this architecture is entirely unsuitable, and indeed not idiomatic, for the programming language. Haskell seem to have neatly side-stepped the problem from the outset since although the name is HUnit [HUnit] as required by the tradition, the architecture is nothing at all like JUnit3. Trying to create the JUnit3 architecture in Haskell would have been hard and definitely not idiomatic, HUnit is definitely idiomatic Haskell.

Listing 2 shows the beginnings of a test using a table driven (aka data driven) approach. It seems silly to have to write a new function for each test case, hence the use of a table (`positiveData`) to hold the inputs and outputs and create all the tests with a generator (`testPositive`, a function of two parameters, the function to test and a string unique to the function so as to identify it). The function `test` takes a list argument with all the tests, here the list is being constructed with a list comprehension: the bit before the `|` is the value to calculate in each case (a fairly arcane expression, but lets not get too het up about it) and the expression after is the 'loop' that drives the creation of the different values, in this case create a list entry for each pair in the table. Then we have a sequence (thanks to the `do` expression[9]) of three calls to `runTestTT` (a function of one parameter) which actually runs all the tests.

Of course, anyone saying to themselves "but he hasn't tested negative values for the arguments of the Factorial functions", you are not being silly; you are being far from silly, very sensible in fact. I am avoiding this aspect of the testing here simply to avoid some Haskell code complexity[10] that adds nothing to the flow in this article. If I had used Python or Java

5. If you need a tutorial introduction to the Haskell programming language then http://learnyouahaskell.com/ and http://book.realworldhaskell.org/ are recommended.
6. If you work with the JVM and want to use Haskell, there is Frege; see http://www.frege-lang.org or https://github.com/Frege/frege Frege is a realization of Haskell on the JVM that allows a few extensions to Haskell so as to work harmoniously with the Java Platform.
7. The name really does give the game away that the framework was for unit testing.
8. Initially called JUnit, then when JUnit4 came out JUnit was renamed JUnit3 as by then it was at major version 3. Now of course we have JUnit5.
9. Yes it's a monad. Apparently monads are difficult to understand, and when you do understand them, they are impossible to explain. This is perhaps an indicator of why there are so many tutorials about monads on the Web.
10. Involving Monads. Did I mention about how once you understand monads, you cannot explain them?

The proposition of proposition-based testing is to make propositions about the code and then use random selection of values from the domain to check the propositions are not invalid

```
module Main where

import Test.HUnit

import Factorial

positiveData = [
 (0, 1),
 (1, 1),
 (2, 2),
 (3, 6),
 (4, 24),
 (5, 120),
 (6, 720),
 (7, 5040),
 (8, 40320),
 (9, 362880),
 (10, 3628800),
 (11, 39916800),
 (12, 479001600),
 (13, 6227020800),
 (14, 87178291200),
 (20, 2432902008176640000),
 (30, 265252859812191058636308480000000),
 (40,
815915283247897734345611269596115894272000000000)
]

testPositive function comment =
    test [comment ++ " " ++ show i ~: "" ~:
    expected ~=? function i |
    (i, expected) <- positiveData]

main = do
 runTestTT (testPositive Factorial.iterative
   "Iterative")
 runTestTT (testPositive Factorial.naïveRecursive
   "Naïve Recursive")
 runTestTT (testPositive Factorial.tailRecursive
   "Tail Recursive")
```

Listing 2

(or, indeed, almost any language other than Haskell) we would not have this issue. For those wishing to see the detail of a full test please see my Factorial repository on GitHub [Winder].

And the proposition is...

The code of Listing 2 nicely shows that what we are doing is selecting values from the domain of the function and ensuring the result of executing the function is the correct value from the image of the function[11]. This is really rather an important thing to do but are we doing it effectively?

Clearly to *prove* the implementation is correct we have to execute the code under test with every possible value of the domain. Given there are roughly 2^{64} (about 18,446,744,073,709,551,616) possible values to test on a 64-bit machine, we will almost certainly decide to give up immediately, or at least within just a few femtoseconds. The test code as shown in Listing 2 is sampling the domain in an attempt to give us confidence that our implementation is not wrong. Have we done that here? Are we satisfied? Possibly yes, but could we do more quickly and easily?

The proposition of proposition-based testing is to make propositions about the code and then use random selection of values from the domain to check the propositions are not invalid. In this case of testing the Factorial function, what are the propositions? Factorial is defined by a recurrence relation comprising two rules. These rules describe the property of the function that is Factorial with respect to the domain, the non-negative integers. If we encode the recurrence relation as a predicate (a Boolean valued function) we have a representation of the property that can be tested by random selection of non-negative integers.

Listing 3 shows a QuickCheck test of Factorial. The function `f_p` is the predicate representing the property being tested. It is a function of two parameters, a function to test and a value to test, with the result being whether the recurrence relation that defines Factorial is true for that value and that function: the predicate is an assertion of the property that any function claiming to implement the Factorial function must satisfy. Why is this not being used directly, but instead `factorial_property` is the predicate being tested by the calls to `quickCheck`? It is all about types and the fact that values are automatically generated for us based on the domain of the property being tested. `f_p` is a predicate dealing with `Integer`, the domain of the functions being tested, values of which can be negative. Factorial is undefined for negative values[12]. So the predicate called by `quickCheck`, `factorial_property`, is defined with `Natural` as the domain, i.e. for non-negative integers[13]. So when we execute `quickCheck` on the function under test, it is non-negative integer values that are generated: The predicate never needs to deal with negative values, it tests just the Factorial proposition and worries not about handling the exceptions that the implementations raise on being given a negative argument. Should we test for negative arguments and that an exception is generated? Probably. Did I mention ignoring this for now?

Earlier I mentioned currying and partial evaluation. In Listing 3, we are seeing this in action. The argument to each `quickCheck` call is an expression that partially evaluates `factorial_property`, binding a

11. Pages such as https://en.wikipedia.org/wiki/Domain_of_a_function and https://en.wikipedia.org/wiki/Image_(mathematics) may be handy if you are unused to the terminology used here.
12. And also non-integral types, do not forget this in real testing.
13. If you are thinking we should be setting up a property to check that all negative integers result in an error, you are thinking on the right lines.

Shrinking is such a boon ... that it is now seen as essential for any property-based testing framework

```
module Main where

import Numeric.Natural
import Test.QuickCheck

import Factorial

f_p :: (Integer -> Integer) -> Integer -> Bool
f_p f n
    | n == 0 = f n == 1
    | otherwise = f n == n * f (n - 1)

factorial_property :: (Integer -> Integer) ->
  Natural -> Bool
factorial_property f n = f_p f (fromIntegral n)

main :: IO()
main = do
  quickCheck (factorial_property iterative)
  quickCheck (factorial_property naïveRecursive)
  quickCheck (factorial_property tailRecursive)
```

Listing 3

particular implementation of `Factorial`, and returning a function that takes only a `Natural` value. This sort of partial evaluation is a typical and idiomatic technique of functional programming, and increasingly any language that supports functions as first class entities.

By default QuickCheck selects 100 values from the domain, so Listing 3 is actually 300 tests. In the case we have here there are no fails, all 300 tests pass. Somewhat splendidly, if there is a failure of a proposition, QuickCheck sets about 'shrinking' which means searching for the smallest value in the domain for which the proposition fails to hold. Many people are implementing some form of proposition testing in many languages. Any not having shrinking are generally seen as being not production ready. Shrinking is such a boon to taking the results of the tests and deducing (or more usually inferring) the cause of the problem, that it is now seen as essential for any property-based testing framework.

Figure 1 shows the result of running the two test programs: first the HUnit example based testing – 18 hand picked tests for each of the three implementations; and second the QuickCheck property-based testing – 100 tests for each case, all passing so no need for shrinking.

But who uses Haskell?

Well, quite a lot of people. However, one of the major goals of Haskell is to 'Avoid success at all costs'[14]. The point here is not un-sensible. Haskell is a language for exploring and extending ideas and principles of functional programming. The Haskell committee therefore needs to avoid having to worry about backward compatibility. This puts it a bit at odds with many commercial and industrial operations who feel that, once written, a line of code should compile (if that is appropriate) and execute exactly the same for all time without any change. Clearly this can be achieved easily in any language by never upgrading the toolchain. However, the organizations that demand code works for all time usually demand that toolchains are regularly updated. (Otherwise the language is considered dead and unusable. There is irony in here somewhere I believe.) There is no pleasing some people. Successful languages in the sense of having many users clearly have to deal with backward compatibility. Haskell doesn't. Thus Haskell, whilst being a very important language, doesn't really have much market traction.

Frege makes an entry

Frege [Frege] though is actually likely to get more traction than Haskell. Despite the potential for having to update codebases, using 'Haskell on the JVM' is an excellent way of creating JVM-based systems. And because the JVM is a polyglot platform, bits of systems can be in Java, Frege, Kotlin [Kotlin], Ceylon [Ceylon], Scala [Scala], Apache Groovy [Groovy], etc. For anyone out there using the Java Platform, I can strongly recommend at least trying Frege. To give you a taste, look at Listing 4, which shows three Frege implementations of the Factorial function, and that Frege really is Haskell. The tests (see Listing 5) are slightly different from the Haskell ones not because the languages are different but because the context is: instead of creating a standalone executable as happens with Haskell, Frege create a JVM class to be managed by a test runner. So instead of a `main` function calling the test executor, we just declare property instances for running using the `property` function, and assume the test runner will do the right thing when invoked. The three examples here show a different way of constraining the test domain to non-negative integers than we saw with Haskell. Function composition (`.` operator, must have spaces either side to distinguish it from member selection) of the property function (using partial evaluation) with a test data generator (`NonNegative.getNonNegative`; dot as selector not function composition) shows how easy all this can be. Instead of just using the default generator (which would be `Integer` for this property function `factorial_property`, we are providing an explicit generator so as to condition the values from the domain that get generated.

```
$ ./factorial_test_hunit
Cases: 18  Tried: 18  Errors: 0  Failures: 0
Cases: 18  Tried: 18  Errors: 0  Failures: 0
Cases: 18  Tried: 18  Errors: 0  Failures: 0

$ ./factorial_test_quickcheck
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Figure 1

14. A phrase initially spoken by Simon Peyton Jones a number of years ago that caught on in the Haskell community.

it behoves us to consider the proposition of proposition testing in one or more languages that have already gained real traction

```
module Factorial where

exceptionErrorMessage = "Factorial not defined for
negative integers."

iterative :: Integer -> Integer
iterative n
    | n < 0 = error exceptionErrorMessage
    | otherwise = product [1..n]

naïveRecursive :: Integer -> Integer
naïveRecursive n
    | n < 0 = error exceptionErrorMessage
    | n == 0 = 1
    | otherwise = n * naïveRecursive (n - 1)

tailRecursive :: Integer -> Integer
tailRecursive n
    | n < 0 = error exceptionErrorMessage
    | otherwise = iteration n  1
    where
      iteration 0 result = result
      iteration i result = iteration (i - 1)
        (result * i)
```

Listing 4

```
module Factorial_Test where

import Test.QuickCheck(quickCheck, property)
import Test.QuickCheckModifiers(NonNegative)

import Factorial(iterative, naïveRecursive,
  tailRecursive)

factorial_property :: (Integer -> Integer)
  -> Integer -> Bool
factorial_property f n
    | n == 0 = f n == 1
    | otherwise = f n == n * f (n - 1)

factorial_iterative_property =
  property ((factorial_property iterative)
  . NonNegative.getNonNegative)
factorial_naïveRecursive_property =
  property ((factorial_property naïveRecursive)
  . NonNegative.getNonNegative)
factorial_tailRecursive_property =
  property ((factorial_property tailRecursive)
  . NonNegative.getNonNegative)
```

Listing 5

```
Factorial_Test.factorial_tailRecursive_property:
  +++ OK, passed 100 tests.
Factorial_Test.factorial_iterative_property:
  +++ OK, passed 100 tests.
Factorial_Test.factorial_naïveRecursive_property:
  +++ OK, passed 100 tests.
Properties passed: 3, failed: 0
```

Figure 2

The result of executing the Frege QuickCheck property-based tests are seen in Figure 2. As with the Haskell, 100 samples for each test with no fails and so no shrinking.

But…

With Haskell trying not to have a user base reliant on backward compatibility, and Frege not yet having quite enough traction as yet to be deemed popular, it behoves us to consider the proposition of proposition testing in one or more languages that have already gained real traction.

First off let us consider…Python.

Let's hypothesize Python

Python [Python_1] has been around since the late 1980s and early 1990s. During the 2000s it rapidly gained in popularity. And then there was the 'Python 2 / Python 3 Schism'.[15] After Python 3.3 was released, there were no excuses for staying with Python 2. (Well, except two – and I leave it as an exercise for the reader to ascertain what these two genuine reasons are for not immediately moving your Python 2 code to Python 3.) For myself, I use Python 3.5 because Python now has function signature type checking [Python_2][16].

Listing 6 shows four implementations of the Factorial function. Note that the function signatures are advisory not strong type checking. Using the MyPy [MyPy] program the types will be checked, but on execution it is just standard Python as people have known for decades.

I suspect the Python code here is sufficiently straightforward that almost all programmers[17] will be able to deduce or infer any meanings that are not immediately clear in the code. But a few comments to help: the `range` function generates a range 'from up to but not including'. The `if` expression is of the form:

15. We will leave any form of description and commentary on the schism to historians. As Python programmers, we use Python 3 and get on with programming.
16. This isn't actually correct: Python allows function signatures as of 3.5 but doesn't check them. You have to have to have a separate parser-type-checker such as MyPy. This is annoying, Python should be doing the checking.
17. We will resist the temptation to make some facetious, and likely offensive, comment about some programmers who use only one programming language and refuse to look at any others. "Resistance is futile." Seven of Nine.

not only are we testing non-negative and negative integers, we also test other forms of error that are possible in Python

```
<true-value> if <boolean-expression>
  else <false-value>
```

The nested function iterate in `tail_recursive` is scoped to the else block.

But are these implementations 'correct'? To test them let's use PyTest [Pytest]. The test framework that comes as standard with Python (unittest, aka PyUnit) could do the job, but PyTest is just better[18]. PyTest provides an excellent base for testing but it does not have property-based testing.

```
from functools import reduce
from operator import mul

def _validate(x: int) -> None:
  if not isinstance(x, int):
    raise TypeError('Argument must be an
integer.')
    if x < 0:
      raise ValueError('Argument must be a
non-negative integer.')

def iterative(x: int) ->int:
  _validate(x)
  if x < 2:
    return 1
  total = 1
  for i in range(2, x + 1):
    total *= i
  return total

def recursive(x: int) -> int:
  _validate(x)
  return 1 if x < 2 else x * recursive(x - 1)

def tail_recursive(x: int) -> int:
  _validate(x)
  if x < 2:
    return 1
  else:
    def iterate(i: int, result: int=1):
      return result if i < 2 else iterate(i - 1,
        result * i)
    return iterate(x)

def using_reduce(x: int) -> int:
    _validate(x)
    return 1 if x < 2 else reduce(mul,
      range(2, x + 1))
```

Listing 6

For this we will use Hypothesis [Hypothesis] (which can be used with PyUnit as easily as with PyTest, but PyTest is just better).

Listing 7 shows a fairly comprehensive test – not only are we testing non-negative and negative integers, we also test other forms of error that are possible in Python. Tests are functions with the first four characters of the name being t, e, s, t. Very JUnit3, and yet these are module-level

```
from pytest import mark, raises

from hypothesis import given
from hypothesis.strategies import (integers,
  floats, text)

from factorial import (iterative, recursive,
  tail_recursive, using_reduce)

algorithms = (iterative, using_reduce, recursive,
  tail_recursive)

@mark.parametrize('a', algorithms)
@given(integers(min_value=0, max_value=900))
def test_with_non_negative_integer (a, x):
  assert a(x) == (1 if x == 0 else x * a(x - 1))

@mark.parametrize('a', algorithms)
@given(integers(max_value=-1))
def test_negative_integer_causes_ValueError(a, x):
  with raises(ValueError):
    a(x)

@mark.parametrize('a', algorithms)
@given(floats())
def test_float_causes_TypeError(a, x):
  with raises(TypeError):
    a(x)

@mark.parametrize('a', algorithms)
def test_none_causes_TypeError(a):
  with raises(TypeError):
    a(None)

@mark.parametrize('a', algorithms)
@given(text())
def test_string_causes_TypeError(a, x):
    with raises(TypeError):
        a(x)

if __name__ == '__main__':
  from pytest import main
  main()
```

Listing 7

18. For reasons that may, or may not, become apparent in this article, but relate to PyUnit following JUnit3 architecture – remember the fading tradition – and PyTest being Pythonic.

automated data generation is at the heart of property-based testing

```
============================= test session starts ==============================
platform linux -- Python 3.5.1, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /home/users/russel/Docs/Papers/ACCU/Draft/TestingPropositions/SourceCode/Python, inifile:
plugins: hypothesis-3.4.0, cov-2.2.1
collected 20 items

test_factorial.py ....................

========================== 20 passed in 2.32 seconds ===========================
```

Figure 3

functions. There are no classes or inheritance in sight: that would be the PyUnit way. The PyTest way is to dispense with the classes as necessary infrastructure, swapping them for needing some infrastructure to be imported in some way or other. (This is all handled behind the scenes when pytest.main executes.) PyTest is in so many ways more Pythonic[19] than PyUnit.

PyTest has the `@mark.parametrize` decorator that rewrites your code so as to have one test per item of data in an iterable. In all the cases here, it is being used to generate tests for each algorithm[20].

The `@given` decorator, which comes from Hypothesis, does not rewrite functions to create new test functions. Instead it generates code to run the function it decorates with a number (the default is 100) of randomly chosen values using the generator given as argument to the decorator, recording the results to report back. This automated data generation is at the heart of property-based testing, and Hypothesis, via the supporting functions such as `integers`, `floats`, and `text` (for generating integers, floats, and string respectively), does this very well. Notice how it is so easy to generate just negative integers or just non-negative integers. Also note the use of the 'with statement'[21] and the `raises` function for testing that code does, in fact, raise an exception.

All the test functions have a parameter `a` that gets bound by the action of the `@mark.parametrize` decorator, and a parameter `x` that gets bound by the action of the `@given` decorator. This is all very different from the partial evaluation used in Haskell and Frege: different language features lead to different idioms to achieve the same goal. What is Pythonic is not Haskellic/Fregic, and vice versa. At least not necessarily.

The `pytest.main` function, when executed, causes all the decorators to undertake their work and executes the result. The output from an execution will look very much as in Figure 3. You may find when you try this that the last line is green.[22]

19. See http://docs.python-guide.org/en/latest/writing/style/
20. There are ways of parameterizing tests in PyUnit (aka unittest), but it is left as an exercise for the reader to look for these. PyTest and `@pytest.mark.parametrize` are the way this author chooses to do parameterized tests in Python.
21. Context managers and the 'with statement' are Python's way of doing RAII (resource acquisition is initialization, https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization) amongst other great things.

Doing the C++ thing

There are many other example languages we could present here to show the almost complete coverage of property-based testing in the world: Kotlin [Kotlin], Ceylon [Ceylon], Scala [Scala], Apache Groovy [Groovy], Rust [Rust], D [D], Go [Go],… However, given this is an August[23] ACCU journal and, historically at least, ACCU members have had a strong interest in C++, we should perhaps look at C++. Clearly people could just use Haskell and QuickCheck to test their C++ code, but let's be realistic here, that isn't going to happen[24]. So what about QuickCheck in C++? There are a number of implementations, for example CppQuickCheck [QuickCheck_2] and QuickCheck++ [QuickCheck_3]. I am, though, going to use RapidCheck [RapidCheck] here because it seems like the most sophisticated and simplest to use of the ones I have looked at to date[25].

There is one thing we have to note straight away: Factorial values are big[26]. Factorial of 30 is a number bigger than can be stored in a 64-bit integer. So all the implementations of Factorial used in books and first year student exercises are a bit of a farce because they are shown using hardware integers: the implementations work for arguments [0..20] and then things get worrisome. "But this is true for all languages and we didn't raise this issue for Haskell, Frege and Python." you say. Well for Haskell (and Frege, since Frege is just Haskell on the JVM) the `Int` type is a hardware number but `Integer`, the type used in the Haskell and Frege code, is an integer type the values of which can be effectively arbitrary size. There is a limit, but then in the end even the universe is finite[27]. What about Python? The Python[28] `int` type uses hardware when it can or an unbounded (albeit finite[27]) integer when it cannot. What about C++? Well

22. Whilst this is an August august journal (and so very august), it is monochrome. So you will have to imagine the greenness of the test output. Either that or actually try the code out for yourself and observe the greenness first hand.
23. Or should that be august. Well actually it has to be both.
24. Not least because Haskell's avowed aim is never to be successful.
25. Also it uses Catch [Catch] for its tests.
26. Factorials are big like space is big, think big in *Hitchhiker's Guide to the Galaxy* terms: "Space," it says, "is big. Really big. You just won't believe how vastly, hugely, mindbogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space. Listen…?" https://en.wikiquote.org/wiki/The_Hitchhiker%27s_Guide_to_the_Galaxy

```
#include <gmpxx.h>

namespace Factorial {

mpz_class iterative(mpz_class const n);
mpz_class iterative(long const n);
mpz_class reductive(mpz_class const n);
mpz_class reductive(long const n);
mpz_class naive_recursive(mpz_class const n);
mpz_class naive_recursive(long const n);
mpz_class tail_recursive(mpz_class const n);
mpz_class tail_recursive(long const n);

} // namespace Factorial
```

Listing 8

the language and standard library have only hardware-based types, which could be taken as rather restricting. GNU has however conveniently created a C library for unbounded (albeit finite[27]) integers, and it has a rather splendid C++ binding [GNU].

So using the GMP C++ API, we can construct implementations of the Factorial function that are not restricted to arguments in the range [0..20] but are more generally useful. Listing 8 shows the functions being exported by the **`Factorial`** namespace. We could dispense with the **`long`** overloads, but it seems more programmer friendly to offer them.

Listing 9 presents the implementations. I suspect that unless you already know C++ (this code is C++14) you have already moved on. So any form of explanatory note is effectively useless here.[29] We will note though that there is a class defined in there as well as implementations of the Factorial function.

```
#include "factorial.hpp"

#include <functional>
#include <iterator>
#include <numeric>

namespace Factorial {
static void validate(mpz_class const n) {
  if (n < 0) {
    throw std::invalid_argument("Parameter must be
a non-negative integer."); }
}
auto const one = mpz_class(1);
auto const two = mpz_class(2);

mpz_class iterative(mpz_class const n) {
  validate(n);
  mpz_class total {1};
  for (unsigned int i = 2; i <= n; ++i) {
    total *= i; }
  return total;
}
mpz_class iterative(long const n) {
  return iterative(mpz_class(n)); }
```

Listing 9

```
class mpz_class_iterator:
  std::iterator<std::input_iterator_tag,
  mpz_class> {
  private:
    mpz_class value;
  public:
    mpz_class_iterator(mpz_class const v) :
      value(v) { }
    mpz_class_iterator& operator++() {
      value += 1; return *this; }
    mpz_class_iterator operator++(int) {
      mpz_class_iterator tmp {
        *this}; ++*this; return tmp; }
      bool operator==(mpz_class_iterator const &
        other) const {
        return value == other.value; }
      bool operator!=(mpz_class_iterator const &
        other) const {
        return value != other.value; }
    mpz_class operator*() const { return  value; }
    mpz_class const * operator->() const {
      return &value; }
};

mpz_class reductive(mpz_class const n) {
  validate(n);
  return (n < 2)
  ? one
  : std::accumulate(mpz_class_iterator(two),
    mpz_class_iterator(n + 1), one,
      std::multiplies<>());}
mpz_class reductive(long const n) {
  return reductive(mpz_class(n)); }
mpz_class naive_recursive(mpz_class const n) {
  validate(n);
  return (n < 2) ? one :
    n * naive_recursive(n - 1);
}

mpz_class naive_recursive(long const n) {
  return naive_recursive(mpz_class(n)); }

static mpz_class tail_recursive_iterate
  (mpz_class const n, mpz_class const result) {
  return (n < 2) ? result :
    tail_recursive_iterate(n - 1, result * n);
}

mpz_class tail_recursive(mpz_class const n) {
  validate(n);
  return (n < 2) ? one : tail_recursive_iterate(n,
    one);
}
mpz_class tail_recursive(long const n) {
  return tail_recursive(mpz_class(n)); }
} // namespace Factorial
```

Listing 9 (cont'd)

Listing 10 presents the RapidCheck-based test code for the Factorial functions. There is a vector of function pointers[30] so that we can easily iterate over the different implementations. Within the loop we have a sequence of the propositions. Each check has a descriptive string and a lambda function. The type of variables to the lambda function will cause (by default 100) values of that type to be created and the lambda executed for each of them. You can have any number of parameters – zero has been chosen here, which might seem a bit strange at first, but think generating random integers. Some of them are negative and some non-negative and

27. Space may be big (see above) but the universe (space being the same thing as the universe as far as we know) is finite – assuming the current theories are correct.
28. Python 3 anyway. Python 2 has effectively the same behaviour, but with more types. It is left as an exercise for the reader whether to worry about this.
29. There was some thought of introducing the acronym RTFC (read the fine code), but this temptation was resisted. "Resistance is futile." Seven of Nine.
30. Well, actually pairs, with the first being the function pointer and the second being a descriptive string.

```
#include "rapidcheck.h"

#include <string>
#include <utility>

#include "factorial.hpp"

std::vector<std::pair<mpz_class (*)(long const),
std::string>> const algorithms {
  {Factorial::iterative, "iterative"},
  {Factorial::reductive, "reductive"},
  {Factorial::naive_recursive, "naïve recursive"},
  {Factorial::tail_recursive, "tail recursive"}
};

int main() {
  for (auto && a: algorithms) {
    auto f = a.first;

    rc::check(a.second + " applied to non-negative
      integer argument obeys the recurrence
      relation.", [f]() {
      auto i = *rc::gen::inRange(0, 900);
      RC_ASSERT(f(i) == ((i == 0) ? mpz_class(1) :
        i * f(i - 1)));
    });

    rc::check(a.second + " applied to negative
       integer raises an exception.", [f]() {
      auto i = *rc::gen::inRange(-100000, -1);
      RC_ASSERT_THROWS_AS(f(i),
        std::invalid_argument);
    });
  }
  return 0;
}
```

Listing 10

we have to be careful to separate these cases as the propositions are so very different. Also some of the calculation for non-negative integers will result in big values. The factorial of a big number is stonkingly big. Evaluation will take a while… a long while… a very long while… so long we will have read *War and Peace*… a large number of times. So we restrict the integers of the domain sample by using an explicit generator. In this case for the non-negative integers we sample from [0..900]. For the negative integers we sample from a wider range as there should only ever be a very rapid exception raised, there should never actually be a calculation.

So that is the Factorial functions themselves tested. I trust you agree that what we have here is a very quick, easy, and providing good coverage test. But, you ask, what about that class? Should we test the class? An interesting question. Many would say "No" because it is internal stuff, not exposed as part of the API. This works for me: why test anything that is not observable from outside. Others will say "Yes" mostly because it cannot hurt. For this article I say "Yes" because it provides another example of proposition-based testing. We do not test any examples, we test only properties of the class and its member functions. See Listing 11. By testing the properties, we are getting as close to proving the implementation not wrong as it is possible to get in an easily maintainable way. QED.

And to prove that point, see Figure 4, which shows the Factorial tests and class test executed. So many useful (passing) tests, so little effort.

The message

Example-based testing of a sample from the domain tells us we are calculating the correct value(s). Proposition-based testing tells us that our code realizes the relationships that should exist between different values

```
#include "rapidcheck.h"
#include "factorial.cpp"

int main() {
  using namespace Factorial;

  rc::check("value of operator delivers the right
    value", [](int i) {
      RC_ASSERT(*mpz_class_iterator{i} == i);
    });

  rc::check("pointer operator delivers the right
    value", [](int i) {
      RC_ASSERT(mpz_class_iterator{i}->get_si()
        == i);
    });

  rc::check("equality is value not identity.",
    [](int i) {
      RC_ASSERT(mpz_class_iterator{i}
        == mpz_class_iterator{i});
    });

  rc::check("inequality is value not identity.",
    [](int i, int j) {
      RC_PRE(j != 0);
      RC_ASSERT(mpz_class_iterator{i}
        != mpz_class_iterator{i + j});
    });

  rc::check("preincrement does in fact increment",
    [](int i) {
      RC_ASSERT(++mpz_class_iterator{i}
        == mpz_class_iterator{i + 1});
    });

  rc::check("postincrement does in fact
    increment", [](int i) {
      RC_ASSERT(mpz_class_iterator{i}++
        == mpz_class_iterator{i});
    });

  rc::check("value of preincrement returns correct
    value",  [](int i) {
      RC_ASSERT(*++mpz_class_iterator{i}
        == i + 1);
  });

  rc::check("value of postincrement returns
    correct value",  [](int i) {
      RC_ASSERT(*mpz_class_iterator{i}++ == i);
  });
}
```

Listing 11

from the domain. They actually tell us slightly different things and so arguably good tests do both, not one or the other. However if we have chosen the properties to test correctly then zero, one, or two examples are likely to be sufficient to 'prove' the code not incorrect. Hypothesis, for example, provides an `@example` decorator for adding those few examples. For other frameworks in other languages we can just add one or two example-based tests to the property-based tests.

But, some will say, don't (example-based) tests provide examples of use? Well yes, sort of. I suggest that these examples of use should be in the documentation, that users should not have to descend to reading the tests. So for me property-based testing (with as few examples as needed) is the future of testing. Examples and exemplars should be in the documentation. You do write documentation, don't you…

```
$ ./test_factorial
Using configuration: seed=10731500115167123548

- iterative applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- iterative applied to negative integer raises an
exception.
OK, passed 100 tests

- reductive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- reductive applied to negative integer raises an
exception.
OK, passed 100 tests

- naïve recursive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- naïve recursive applied to negative integer
raises an exception.
OK, passed 100 tests

- tail recursive applied to non-negative integer
argument obeys the recurrence relation.
OK, passed 100 tests

- tail recursive applied to negative integer
raises an exception.
OK, passed 100 tests

$ ./test_mpz_class_iterator
Using configuration: seed=9168594634932513587

- value of operator delivers the right value
OK, passed 100 tests

- pointer operator delivers the right value
OK, passed 100 tests

- equality is value not identity.
OK, passed 100 tests

- inequality is value not identity.
OK, passed 100 tests

- preincrement does in fact increment
OK, passed 100 tests
```

Figure 4

An apology

Having just ranted about documentation, you may think I am being hypocritical since the code presented here has no comments. A priori, code without comments, at least documentation comments[31], is a Bad Thing™ – all code should be properly documentation commented. All the code in the GitHub repository that holds the originals from which the code presented here were extracted is. So if you want to see the properly commented versions, feel free to visit https://github.com/russel/Factorial. If you find any improperly commented code, please feel free to nudge me about it and I will fix it post haste[32].

31. Debating the usefulness or otherwise of non-documentation comments is left as an exercise for the readership.
32. And request Doctor Who or someone to perform appropriate time travel with the corrections so that the situation has never been the case.

Acknowledgements

Thanks to Fran Buontempo for being the editor of this august[33] journal, especially this August august journal[34], and letting me submit a wee bit late.

Thanks to Jonathan Wakely for not laughing too much when I showed him the original C++ code, and for making suggestions that made the code far more sensible.

Thanks to the unnamed reviewers who pointed out some infelicities of presentation as well as syntax. Almost all the syntactic changes have been made – I disagreed with a few. Hopefully the changes made to the content has fully addressed the presentation issues that were raised.

Thanks to all those people working on programming languages and test frameworks, and especially for those working on property-based testing features, without whom this article would have been a very great deal shorter. ■

References

[Catch] https://github.com/philsquared/Catch

[Ceylon] http://ceylon-lang.org/

[D] http://dlang.org/

[FitNesse] http://www.fitnesse.org/

[Frege] http://www.frege-lang.org or https://github.com/Frege/frege

[GNU] https://gmplib.org/, https://gmplib.org/manual/C_002b_002b-Interface-General.html

[Go] https://golang.org/

[Groovy] http://www.groovy-lang.org/

[Hackage] https://hackage.haskell.org/package/QuickCheck

[Haskell] https://www.haskell.org/

[HUnit] https://github.com/hspec/HUnit

[Hypothesis] http://hypothesis.works/, https://hypothesis.readthedocs.io/en/latest/, https://github.com/HypothesisWorks/hypothesis-python

[Kotlin] http://kotlinlang.org/

[MyPy] http://www.mypy-lang.org/

[Pytest] http://pytest.org/latest/

[Python_1] https://www.python.org/

[Python_2] https://www.python.org/dev/peps/pep-3107/, https://www.python.org/dev/peps/pep-0484/

[QuickCheck] https://en.wikipedia.org/wiki/QuickCheck

[QuickCheck_2] https://github.com/grogers0/CppQuickCheck

[QuickCheck_3] http://software.legiasoft.com/quickcheck/

[RapidCheck] https://github.com/emil-e/rapidcheck

[Robot] http://robotframework.org/

[Rust] https://www.rust-lang.org/

[Scala] http://www.scala-lang.org/

[Winder] The full Haskell example can be found at https://github.com/russel/Factorial/tree/master/Haskell.

[Winder16] *Overload*, 24(131):26–32, February 2016. There are PDF (http://accu.org/var/uploads/journals/Overload131.pdf#page=27) or HTML (http://accu.org/index.php/journals/2203) versions available.

Russel was a long-standing ACCU member who passed away earlier this year. We miss him and have reprinted this article in his memory, which was first published in *Overload* 134, August 2016.

33. And, indeed, August.
34. "This joke is getting silly, stop this joke immediately." The Colonel.

Teach Your Computer to Program Itself

Can AI replace programmers? Frances Buontempo demonstrates how to autogenerate code and why we may not be replaceable yet.

Programming is difficult. Since the dawn of computers, various attempts have been made to make programmers' lives easier. Initially, programming involved low level work, which we would struggle to recognize as a language in any way, shape or form. In order to ease the situation, automatic programming was introduced. This may sound as though programmers could be automated away; however, at the time this meant high level languages. Biermann [Biermann76] begins his introduction by saying:

> Ever since the early days of electronic computers, man has been using his ingenuity to save himself from the chores of programming. Hand coding in machine language gave way to assembly language programming, which was replaced for most users by compiled languages.

He goes on to ask how much could be automatically generated, for example could input/output specification allow automatic generation of the code in-between? This question raises its head from time to time.

In this article, I will illustrate one approach, mentioning the difficulties involved and considering if it can become widespread. Before we begin, bear in mind Biermann was writing at a time when high level programming languages were cutting edge. Prior to these, programmers coded numerical operations using absolute addresses, hexadecimal programming if you will. The introduction of assembly languages was the start of the high level language evolution. The next step removed programmers further from the machine, introducing computers that programmed themselves, in some sense. "Interpreters, assemblers, compilers, and generators—programs designed to operate on or produce other programs, that is, automatic programming" according to Mildred Koss [Wikipedia]. It could be argued that we no longer program computers. Instead, we give high level instructions, and they write the code for us. Computers still need our input in this model though. Though automatic programming is what we might now simply call programming, another approach to programming, alluded to by Biermann's mention of code generation from specification, known as Program Synthesis has received attention again recently. In particular Microsoft have been writing about the subject. According to Microsoft [Microsoft17a]:

> Program synthesis is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification. Since the inception of AI in the 1950s, this problem has been considered the holy grail of Computer Science.

Their paper [Microsoft17b] offers a literature review covering common program synthesis techniques and potential future work in the field. One topic they mention is Genetic Programming. My article will provide you with an introduction to this technique, using the time honoured Fizz Buzz problem.

According to the c2 wiki [c2], Fizz Buzz is an interview question, "designed to help filter out the 99.5% of programming job candidates who can't seem to program their way out of a wet paper bag."

The problem is as follows:

> Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Genetic programming (GP) starts from a high-level statement of 'what needs to be done' and automatically creates a computer program to solve the problem. In this sense, is it identical to other synthesis techniques. GP falls into a broader area of stochastic search, meaning the search is partly random. In theory, a completely randomly generated program may solve the problem at hand, but genetic programing uses the specification, or in simpler language, tests, to guide the search. GP works in a similar manner to a genetic algorithm. I previously gave a step by step guide to coding your way out of a paper bag, regardless of whether or not it was wet, in *Overload* [Buontempo13], and my book [Buontempo19] goes into more detail.

A basic genetic algorithm finds variables that solve a problem, either exactly or approximately. In practice this means a set of tests must be designed up front, also known as a fitness or cost function to ascertain whether the problem has been solved. A potential solution is assessed by running the tests with the chosen variables and using the outcome as a score. This could be number of tests passed, a cost which could be in dollars to be minimised or a 'fitness' to be maximized, such as money saved. The potential applications are broad, but the general idea of a test providing a numerical score to be maximized or minimized means a genetic algorithm can also be seen as an optimization technique. Genetic algorithms are inspired by the idea of Darwinian evolution, forming a subset of evolutionary algorithms. They use selection and mutation to drive the population towards greater fitness, inspired by the idea of the survival of the fittest.

The algorithm starts with a collection of fixed length lists populated at random. The values can be numbers, Booleans or characters, to investigate problems requiring numerical solutions, various constraint-based problems, hardware design and much more besides. They can even be strings, such as place names, allowing a trip itinerary to be discovered. The problem at hand will steer the size of the list. Some problems can be encoded in various ways, but many have one obvious encoding. If you want to discover two numbers, your list will comprise two numbers. The number of solutions, or population size, can be anything. Starting small and seeing what happens often works, since this will progress quickly, but may get stuck on bad solutions. The fixed length lists are often referred to as genotypes, alluding to genes representing a chromosome. The solutions can be assessed, or tested, and the best solution reported. The average, best, worst and standard deviation of scores can also help decide how well the

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

At each step, the population can be completely or partially replaced by creating or 'breeding' new lists from 'better' solutions or 'parents'

search is progressing. As the algorithm continues, the score should improve. At its simplest the population will remain fixed size over the duration of the search, though variations allow it to grow or shrink. At each step, the population can be completely or partially replaced by creating or 'breeding' new lists from 'better' solutions or 'parents'. The parent solutions are taken from the population, using one of many selection algorithms. A tournament selection is the simplest to implement and understand. A fixed number are randomly selected from the population and the two with the optimal score are selected. This fixed number is another decision that must be made. It could vary over the life of the algorithm.

Once two parents are selected, they 'breed', referred to as 'crossover' or recombination. For a list of two variables, this could be achieved by splitting the variables in half, passing one value (or gene) from each parent to the new offspring (Figure 1)

For more than two variables, there are many other options: splitting at a random point, splitting in various places and interleaving the genes, alternating, and so on. Regardless of the crossover strategy, this alludes to a population gene pool with gene crossover forming new offspring. Survival of the fittest, or fit enough, ensures better adapted genes survive.

Crossover will explore the search space of potential solutions, though may miss some. In order to avoid getting stuck in a rut, mutation is also used, again alluding to natural selection. A random percentage of the new solutions are mutated. This percentage is yet another variable, or hyper-parameter to choose. In effect, this means flipping a Boolean value, incrementing, decrementing or scaling a number, or replacing a value with an entirely new random value. The combination of selection, pushing solutions to get better, along with crossover and mutation encourages exploration of the search space. This is useful if a brute force solution would take too long, since it explores fruitful looking parts of the space

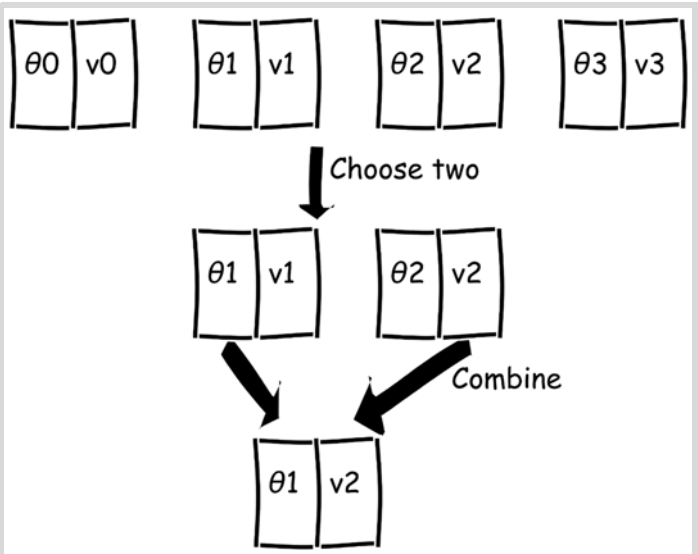


Figure 1

without needing to try everything. The algorithm is not guaranteed to work, but even if it cannot solve your problem the evolved attempts at a solution will reveal details about the shape of the search and solution space.

It should be noted that crossover and mutation can generate invalid solutions. A trip itinerary could end up with the same place twice while leaving somewhere out. A numerical problem may end up with negative numbers which are inappropriate. Such unviable solutions could be killed immediately, however the testing, or fitness step will also weed them out, if designed appropriately. This is one of many decisions to be made.

Once the population has been fully or partially replaced, using crossover and mutation, the algorithm's progress is reported, again by best, along with any other statistics required. The breed → test → report cycle is repeated either for a pre-specified number of times or until a good enough solution is found. Recall that several hyper-parameters had to be chosen – rates of mutation, how to select parents, how big a population size, how to perform crossover. A genetic algorithm may be run several times with different hyper-parameters before a solution is found.

With one change, from solutions as lists to solutions as trees, we can switch from a genetic algorithm to genetic programming. The tree can represent a program, either as an expression or an abstract syntax tree. GP also uses crossover, mutation and testing to explore the search space. Aside from hyper-parameter choice, which is admittedly difficult, a suite of tests to pass means GP can generate code for us. I shall take the liberty of borrowing Kevlin Henney's Fizz Buzz test suite. They comprise eight tests, which are necessary and sufficient for Fizz Buzz, as follows:

1. Every result is 'Fizz', 'Buzz' 'FizzBuzz' or a decimal string
2. Every decimal result corresponds to its ordinal position
3. Every third result contains 'Fizz'
4. Every fifth result contains 'Buzz'
5. Every fifteenth result is 'FizzBizz'
6. The ordinal position of every 'Fizz' result is divisible by 3
7. The ordinal position of every 'Buzz' is divisible by 5
8. The ordinal position of 'FizzBuzz' is divisible by 15

These form our fitness function, giving a maximum score of eight. A score of minus one will be awarded to programs which are not viable (for example, invalid syntax or a run time error).

In what follows, I shall use the Distributed Evolutionary Algorithms in Python (DEAP) framework [DEAP-1]. Other frameworks are available. First, a Toolbox object is required. This will hold all the parts we need.

```
toolbox = base.Toolbox()
```

Of note, DEAP does not use an abstract syntax tree directly, instead what are described as primitive operators are required, stored in a 'primitive set'.

```
primitive_set = gp.PrimitiveSet("MAIN", 1)
```

This requires some extra boilerplate to add primitives (operators and functions) to the set, such as `operator.add` and similar. In addition you can add terminals (leaf nodes in a tree), such as 3 or 'Fizz'. You can also add what are termed ephemeral constants, which could take a lambda that returns a random number. This allows constants to be generated per solution.

Not all expressions randomly generated are valid. Invalid trees can be dropped immediately

```
primitive_set.addPrimitive(operator.and_, 2)
primitive_set.addPrimitive(operator.or_, 2)
primitive_set.addPrimitive(if_then_else, 3)
primitive_set.addPrimitive(mod3, 1)
primitive_set.addPrimitive(mod5, 1)
primitive_set.addPrimitive(mod15, 1)
primitive_set.addTerminal("Buzz")
primitive_set.addTerminal("Fizz")
primitive_set.addTerminal("FizzBuzz")
```

Listing 1

The full code listing is available in a gist [Buontempo], but Listing 1 contains a few examples I used.

The mysterious `if_then_else` is defined as follows:

```
def if_then_else(x,y,z):
  if x:
    return y
  else:
    return z
```

There may be a neater way to get DEAP to use this construct, but it will do.

The modXXX functions are again a bit of a hack, for example:

```
def mod3(x):
  return operator.mod(x, 3) == 0
```

A primitive takes a callable and the number of parameters, while a terminal, or leaf node, takes a value only. I did not manage to get the GP to find the values 3, 5 or 15 which was disappointing. The gist includes a version to generate code finding if a number is odd or even, using new primitives and 'constants':

```
primitive_set.addPrimitive(operator.mod, 2)
primitive_set.addEphemeralConstant("rand101",
  lambda: random.randint(-1,1))
primitive_set.addEphemeralConstant("randints",
  lambda: random.randint(0,10))
```

This did work, so in theory I suspect the GP could discover more of the parts needed for Fizz Buzz with more experimentation. For this article, we shall cheat and give the GP some help.

It is worth naming the arguments so the generated code looks less ugly:

```
primitive_set.renameArguments(ARG0='x')
```

Once the primitive set is populated, DEAP can form an expression:

expr = gp.genFull(primitive_set, min_=1, max_=3)

that can then be converted to tree:

```
tree = PrimitiveTree(expr)
```

Printing the randomly generated tree gives

```
mod3('Fizz')
```

You can 'compile' this to see what happens as follows:

```
example = toolbox.compile(tree)
```

This returns a lambda, which you can then call. The astute amongst you may notice this will lead to a run time error. Not all expressions randomly generated are valid. Invalid trees can be dropped immediately.

In order to run the algorithm, we tell DEAP how to assess the fitness of individual trees

```
creator.create("FitnessMax", base.Fitness,
  weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree,
  fitness=creator.FitnessMax)
```

This is very stringly typed, but allows much flexibility. The weights for the fitness are set to one, meaning the number of tests passed, from our eight tests, is multiplied by one. Setting the weight to `-1` would seek out a minimum rather than a maximum. The primitive tree type is "specifically formatted for optimization of genetic programming operations" according to the documentation [DEAP-2]. In theory you could extend this or make a new type. A strongly typed version is available. We will stick to the `PrimitiveTree` type for this example.

Using the toolbox we created earlier, we register expressions, individuals, populations and a compile function (see Listing 2).

The half and half for the expressions means half the time expressions are generated with grow (between the maximum and minimum height) and the other half with full (maximum height). Letting expressions get too big takes time and may not give much benefit. In theory you could add `if true` inside `if true` many, many times. Without baking preferences for acceptable source code, this will have no impact on the fitness of a solution, but take time to grow, so the maximum tree size parameters can help.

I initially tried 5 and the trees improved but then stuck below 100% fitness, then started getting worse. As with genetic algorithms, and almost all machine learning, you need to experiment with the parameters.

We need to register a few more things then we're ready to go. We need a fitness function (Listing 3, on the next page).

Yes, the fitness function returns a tuple, required by the framework, so that is not a spare comma at the end. The `tests_passed` function takes the function generated by GP and runs it over some numbers, conventionally 1 to 100 for Fizz Buzz, though I have chosen to start at zero. `safe_run` is used to catch and penalize errors.

```
toolbox.register("expr", gp.genHalfAndHalf,
  primitive_set=primitive_set, min_=1, max_=2)
toolbox.register("individual", tools.initIterate,
  creator.Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat,
  list, toolbox.individual)
toolbox.register("compile", gp.compile,
  primitive_set=primitive_set)
```

Listing 2

```
def tests_passed (func, points):
  passed = 0
  def safe_run(func, x):
    try:
      return func(x)
    except:
      return -1
  results = [safe_run(func, x) for x in points]
  # not listed for brevity,
  # but passed += 1 for each of the eight tests
  # assessed on the results list just formed
  return passed
def fitness(individual, points):
  # Transform the tree expression in a callable
  # function
  func = toolbox.compile(expr=individual)
  return tests_passed(func, points),
```

Listing 3

We then register this fitness function and set up more options and parameters (see Listing 4).

I also added some statistics to the toolbox to track the algorithms progress. This used DEAP's multi-statistics tool (again, full details in the gist)

```
mstats = tools.MultiStatistics(…)
```

These are reported at each step in the loop. To keep track of the best tree use the hall of fame:

```
hof = tools.HallOfFame(1)
```

Recall that the GP uses randomness to find solutions, so I clamped the seed for repeatable experiments, `random.seed(318)`

We finally decide a population size, how often to crossover and mutate, and how many generations to run this for. It's worth trying a small population first, 10 or so, and a few generations to see what happens, then increase these if things do seem to improve over the generations:

```
pop = toolbox.population(n=4000)
pCrossover = 0.75
pMutation = 0.5
nGen = 75
pop, log = algorithms.eaSimple(pop, toolbox,
  pCrossover, pMutation, nGen, stats=mstats,
  halloffame=hof, verbose=True)
```

After running for several minutes we obtain an expression that generates Fizz Buzz for us (Listing 5). The best tree has 81 nodes, with the expression shown in Listing 6. (See also Figure 2, overleaf.)

The generated expression is very unpleasant, but it works. In my defence, I recently came across a report into 'Modified Condition Decision Coverage' to "assist with the assessment of the adequacy of the requirements-based testing process". [US-FAA]. On page 177, the authors cite an expression with 76 conditionals found in Ada source code for aeroplane black boxes (Listing 7).

So, humans can write complicated code too. Somewhere in the middle of the figure you may see the conditional

```
if FizzBuzz or FizzBuzz: then FizzBuzz
```

```
toolbox.register("evaluate",
  fitness points=range(101))
toolbox.register("select", tools.selTournament,
  tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0,
  max_=2)
toolbox.register("mutate", gp.mutUniform,
  expr=toolbox.expr_mut, pset=primitive_set)
```

Listing 4

```
['FizzBuzz', 1, 2, 'Fizz', 4, 'Buzz', 'Fizz', 7,
8, 'Fizz', 'Buzz', 11, 'Fizz', 13, 14, 'FizzBuzz',
16, 17, 'Fizz', 19, 'Buzz', 'Fizz', 22, 23,
'Fizz', 'Buzz', 26, 'Fizz', 28, 29, 'FizzBuzz',
31, 32, 'Fizz', 34, 'Buzz', 'Fizz', 37, 38,
'Fizz', 'Buzz', 41, 'Fizz', 43, 44, 'FizzBuzz',
46, 47, 'Fizz', 49, 'Buzz', 'Fizz', 52, 53,
'Fizz', 'Buzz', 56, 'Fizz', 58, 59, 'FizzBuzz',
61, 62, 'Fizz', 64, 'Buzz', 'Fizz', 67, 68,
'Fizz', 'Buzz', 71, 'Fizz', 73, 74, 'FizzBuzz',
76, 77, 'Fizz', 79, 'Buzz', 'Fizz', 82, 83,
'Fizz', 'Buzz', 86, 'Fizz', 88, 89, 'FizzBuzz',
91, 92, 'Fizz', 94, 'Buzz', 'Fizz', 97, 98,
'Fizz', 'Buzz']
```

Listing 5

```
if_then_else(mod15(if_then_else(if_then_else(mul(
x, 'FizzBuzz'), 'Fizz', 'Buzz'), x,
if_then_else('Buzz', 'FizzBuzz', mod3(x)))),
'FizzBuzz',
if_then_else(both(if_then_else(if_then_else(mod15
(x), either('FizzBuzz', 'FizzBuzz'), 'FizzBuzz'),
if_then_else('FizzBuzz', mod15(mod5(x)), 'Buzz'),
'Buzz'), if_then_else('Fizz', 'Buzz',
if_then_else('FizzBuzz',
if_then_else(if_then_else('Buzz',
if_then_else(if_then_else(mod3(x), x,
'FizzBuzz'), if_then_else(x, x, either('Buzz',
'Buzz')), x), 'Fizz'), 'Fizz', x),
if_then_else(either(if_then_else(x, x, mod3(x)),
'FizzBuzz'), 'Fizz', 'Fizz')))),
if_then_else(mod15(x), either('FizzBuzz',
either('Buzz', x)), if_then_else(mod3(x), 'Fizz',
x)), 'Buzz'))
```

Listing 6

It is possible to prune a tree, however if the aim of program synthesis is to remove programmer from the process, no one need ever see the implementation.

It took considerably longer to get my computer to generate this code than it would have taken to write this by hand. Any machine learning algorithm needs several runs and many attempts to tune parameters. Tools are gradually coming to the fore to track the training process. For example, Feature stores to keep cleansed data for training are becoming popular and DevMLOps is becoming a common phrase. MLFLow [MLFLow] offers a way to track training and parameter choice. Furthermore automatic parameter tuning is being investigated, for example AWS' automatic model tuning in SageMaker [Amazon].

Was this GP Fizz Buzz exercise worth the time spent? The exercise was interesting, and I encourage you to try it out. If you can get a version working without cheating, so that your computer discovers the magic

```
Bv or (Ev /= El) or Bv2 or Bv3 or Bv4 or Bv5 or
Bv6 or Bv7 or Bv8 or Bv9 or Bv10 or Bv11 or Bv12
or Bv13 or Bv14 or Bv15 or Bv16 or Bv17 or Bv18 or
Bv19 or Bv20 or Bv21 or Bv22 or Bv23 or Bv24 or
Bv25 or Bv26 or Bv27 or Bv28 or Bv29 or Bv30 or
Bv31 or Bv32 or Bv33 or Bv34 or Bv35 or Bv36 or
Bv37 or Bv38 or Bv39 or Bv40 or Bv41 or Bv42 or
Bv43 or Bv44 or Bv45 or Bv46 or Bv47 or Bv48 or
Bv49 or Bv50 or Bv51 or (Ev2 = El2) or ((Ev3 =
El2) and (Sav /= Sac)) or Bv52 or Bv53 or Bv54 or
Bv55 or Bv56 or Bv57 or Bv58 or Bv59 or Bv60 or
Bv61 or Bv62 or Bv63 or Bv64 or Bv65 or Ev4 /= El3
or Ev5 = El4 or Ev6 = El4 or Ev7 = El4 or Ev8 =
El4 or Ev9 = El4 or Ev10 = El4
```

Listing 7

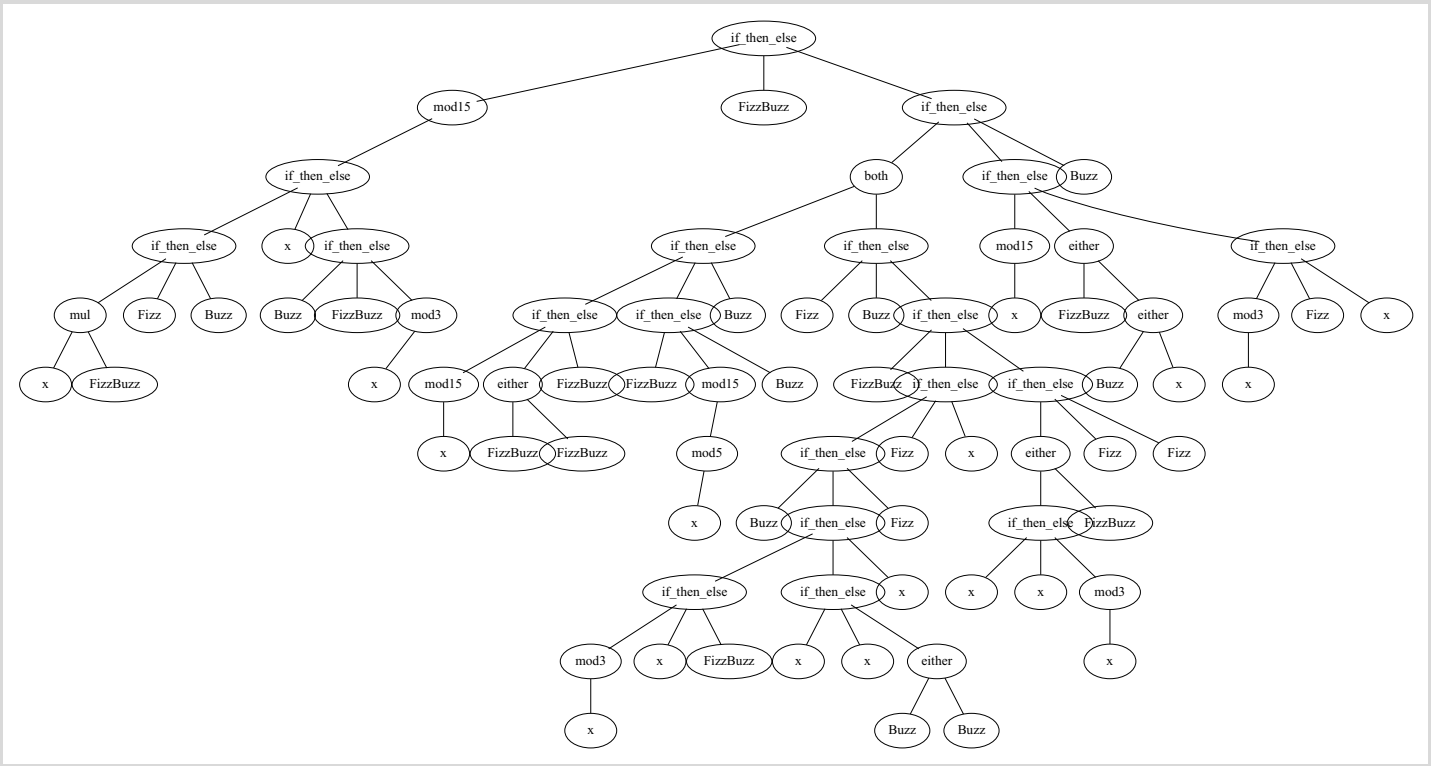


Figure 2

numbers 3 and 5, do write in. Having an understanding of the implementation and details behind buzz words such as Artificial Intelligence is always useful.

I mentioned a Microsoft paper at the start. Program synthesis seems to be an area of research that has recently become active again. They mention a Python Programming Puzzles project "which can be used to teach and evaluate an AI's programming proficiency" [Microsoft-1]. Fizz Buzz isn't here, yet. For further background on genetic programming, go to Genetic Programming Inc [GP] and the Royal Society Paper, 'Program synthesis: challenges and opportunities' [David17] gives a thorough overview . The ACM SIGPLAN published a blog giving an overview of the state of the art in 2019 [Bornholt19]. This includes Excel's 'Flash Fill' [Microsoft-2], which offers suggestions to fill cells as you type based on patterns it finds. This uses an inductive approach, which you could describe as extrapolating from examples, rather than the pre-specified criteria tests approach used in this article. Program synthesis has been used to automatically derive compiler optimizations. In particular, Souper can help identify missing peephole optimizations in LLVM's midend optimizers [Souper]. Even if computers never fully program themselves, program synthesis can be and is used for practical applications. Watch this space. ■

References

[Amazon] 'How Hyperparamter Tuning Works', availble at: https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-how-it-works.html

[Biermann76] Biermann (1976) 'Approaches to automatic programming' in *Advances in Computers* Vol 15, p1–63, available at: https://www.sciencedirect.com/science/article/pii/S0065245808605197

[Bornholt19] James Bornholt (2019) 'Program Synthesis in 2019', available at https://blog.sigplan.org/2019/07/31/program-synthesis-in-2019/

[Buontempo] GP Fizz Buzz: https://gist.github.com/doctorlove/be4ebe4929855f69861c13b57dbcf3aa

[Buontempo13] Frances Buontempo (2013) 'How to Program Your Way Out of a Paper Bag Using Genetic Algorithms' in *Overload* 118, available from https://accu.org/journals/overload/21/118/overload118.pdf#page=8

[Buontempo19] Frances Buontempo (2019) *Genetic Algorithms and Machine Learning for Programmers*, The Pragmatic Bookshelf, ISBN 9781680506204. See https://pragprog.com/titles/fbmach/genetic-algorithms-and-machine-learning-for-programmers/

[c2] 'Fizz Buzz Test' on the c2 wiki: https://wiki.c2.com/?FizzBuzzTest

[David17] Cristina David and Daniel Kroening (2017) 'Program synthesis: challenges and opportunities', The Royal Society September 2017, available at: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0403

[DEAP-1] DEAP documentation: https://deap.readthedocs.io/en/master/

[DEAP-2] Genetic Programming: https://deap.readthedocs.io/en/master/api/gp.html

[GP] Genetic http://www.genetic-programming.com/

[Microsoft-1] https://github.com/microsoft/PythonProgrammingPuzzles

[Microsoft-2] Excel's 'Flash Fill': https://support.microsoft.com/en-us/office/using-flash-fill-in-excel-3f9bcf1e-db93-4890-94a0-1578341f73f7

[Microsoft17a] 'Program Synthesis' abstract (2017) https://www.microsoft.com/en-us/research/publication/program-synthesis/

[Microsoft17b] 'Program Synthesis' paper (2017) https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/program_synthesis_now.pdf

[MLFLow] https://mlflow.org/

[Souper] https://github.com/google/souper

[US-FAA] Federal Aviation Administration (2001) 'An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion', U.S. Department of Transportation, available from: http://www.tc.faa.gov/its/worldpac/techrpt/ar01-18.pdf

[Wikipedia] Mildred Koss, quoted by Wikipedia: https://en.wikipedia.org/wiki/Automatic_programming

Afterwood

Git is not universally loved. Chris Oldwood suggests that Git itself is not the problem.

Git comes in for a lot of stick for being a complicated tool that's hard to learn, and they're right, git is a complicated tool. But it's a tool designed to solve a difficult problem – many disparate people collaborating on a single product in a totally decentralized fashion. However, many of us don't need to work that way, so why are we using the tool in a way that makes our lives more difficult?

KISS

For my entire professional programming career, which now spans over 25 years, and my personal endeavours, I have used a version control tool (VCS) to manage the source code. In that time, for the most part, I have worked in a trunk-based development fashion [Oldwood14a]. That means all development goes on in one integration branch and the general philosophy for *every* commit is 'always be ready to ship' [c2]. (This does not preclude the use of private branches for spikes and/or release branches for hotfix engineering when absolutely needed. #NoAbsolutes.) As you might guess features toggles [Oldwood13] (in many different guises) play a significant part in achieving that.

A consequence of this simplistic way of working [Oldwood16] is that my development cycle, and therefore my use of git, boils down to these few steps:

- clone
- *edit* / *build* / *test*
- diff
- add / commit
- pull
- push

There may occasionally be a short inner loop where a merge conflict shows up during the pull (integration) phase which causes me to go through the edit / diff / commit cycle again, but by-and-large conflicts are rare due to close collaboration and very short change cycles. Ultimately though, from the gazillions of commands that git supports, I *mostly* use just those 6. As you can probably guess, despite using git for nearly 7 years, I actually know very little about it (command wise). (I pondered including 'log' in the list for when doing a spot of software archaeology [Oldwood14b] but that is becoming much rarer these days. I also only use 'fetch' when I have to work with feature branches.)

Isolation

Where I see people getting into trouble and subsequently venting their anger is when branches are involved. This is not a problem that is specific to git though, you see this crop up with any VCS that supports branches whether it be ClearCase, Perforce, Subversion, etc. Hence, *the tool is not the problem, the workflow is*. And that commonly stems from a delivery process mandated by the organization, meaning that ultimately the issue is one of an organizational nature, not the tooling *per se*.

An organisation which seeks to reduce risk by isolating work (and by extension its people) onto branches is increasing the delay in feedback thereby paradoxically increasing the risk of integration, or so-called 'merge debt' [Oldwood14a]. A natural side-effect of making it harder to push through changes is that people will start batching up work in an attempt to boost 'efficiency'. The trick is to go in the opposite direction and break things down into smaller units of work that are easier to produce and quicker to improve. Balancing production code changes with a solid investment in test coverage and automation reduces that risk further along with collaboration boosting techniques like pair and mob programming. [Oldwood18]

Less is more

Instead of enforcing a complicated workflow and employing complex tools in the hope that we can remain in control of our process we should instead seek to keep the workflow simple so that our tools remain easy to use. Git was written to solve a problem most teams don't have as they neither have the volume of distributed people or complexity of product to deal with. Organisations that do have complex codebases cannot expect to dig themselves out of their hole simply by introducing a more powerful version control tool, it will only increase the cost of delay while bringing a false sense of security as programmers work in the dark for longer. ■

References

[c2] 'Always Be Ready To Ship', available from https://wiki.c2.com/?AlwaysBeReadyToShip

[Oldwood13] Chris Oldwood 'Codebase Stability With Feature Toggles', published to blog 17 October 2013, available from https://chrisoldwood.blogspot.com/2013/10/codebase-stability-with-feature-toggles.html

[Oldwood14a] Chris Oldwood (2014) 'Branching Strategies', *Overload* 121 published June 2014, available from https://accu.org/journals/overload/22/121/oldwood_1920/

[Oldwood14b] Chris Oldwood (2014) 'In The Toolbox – Software Archaeology', published in *CVu* and available from http://www.chrisoldwood.com/articles/in-the-toolbox-software-archaeology.html

[Oldwood16] Chris Oldwood (2016) 'In The Toolbox – Commit Checklist' published in *CVu* and available from http://www.chrisoldwood.com/articles/in-the-toolbox-commit-checklist.html

[Oldwood18] Chris Oldwood (2018) 'To Mob, Pair or Fly Solo', published in *CVu* 30.5 (November 2018) and available from http://www.chrisoldwood.com/articles/to-mob-pair-or-fly-solo.html

This article was first published by Chris on his blog in December 2019.

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the lounge below his bedroom. With no Godmanchester duck race to commentate on this year, he's been even more easily distracted by messages to gort@cix.co.uk or @chrisoldwood

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

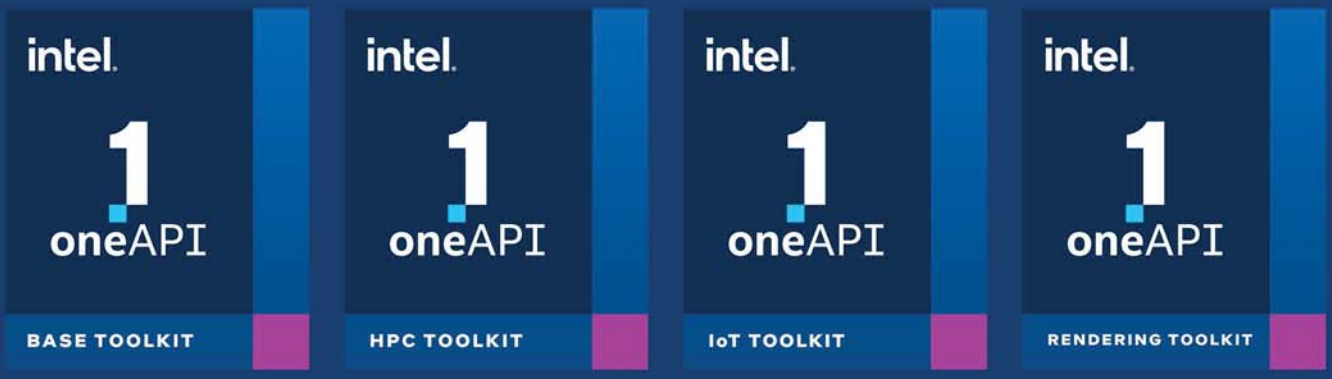
PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel software

oneAPI: New Era of Accelerated Computing

oneAPI

Take the open, productive path to accelerate cross-architecture computing using Intel® oneAPI Toolkits.



Developers, take advantage of oneAPI's unified, standards-based, cross-architecture programming model that sets you free to develop applications for your choice of architectures. Get full hardware performance using a complete set of proven tools without the limits of proprietary language lock-in.

Learn more: www.qbssoftware.com