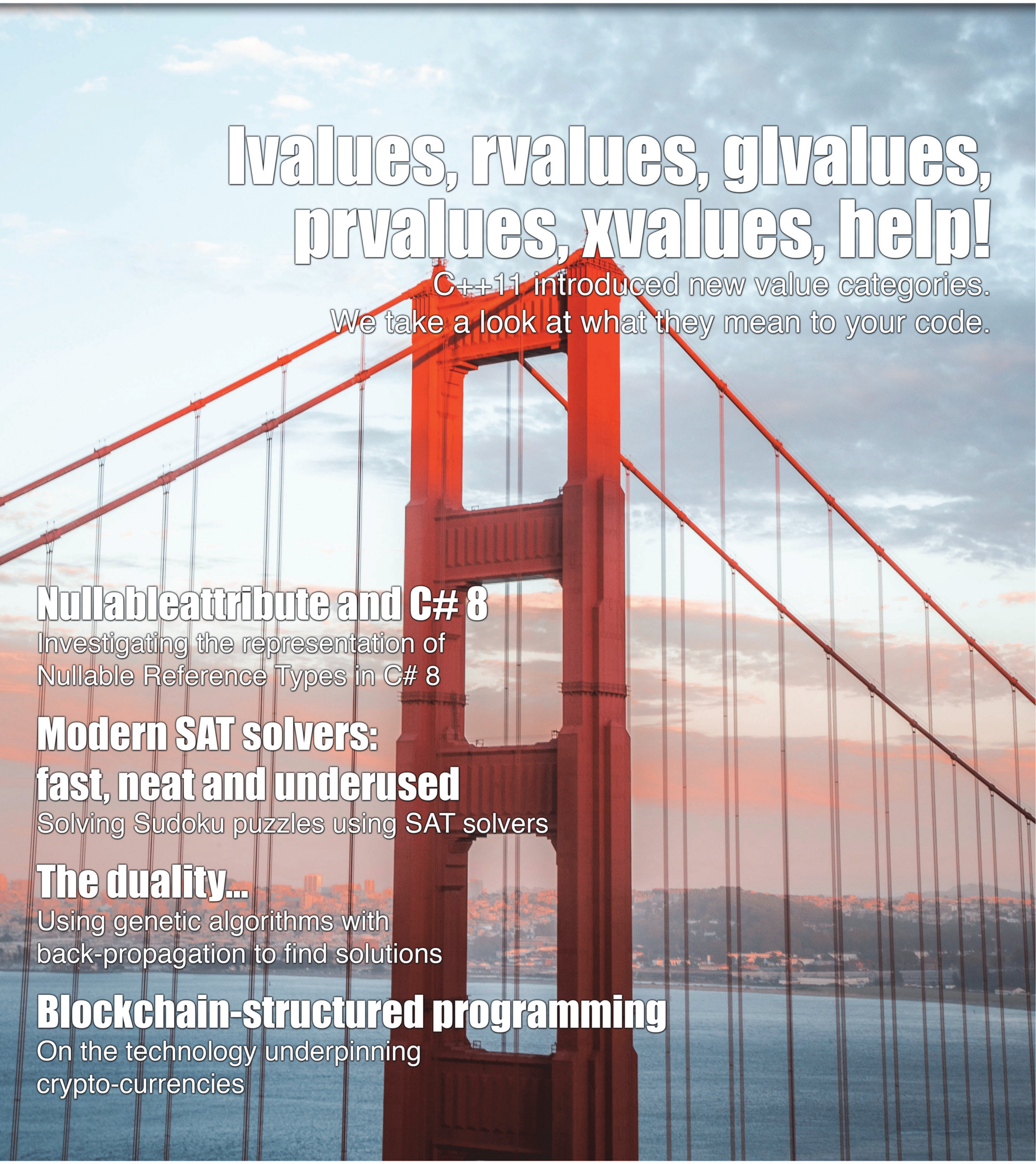
overload 150

APRIL 2019 £4.50



lvalues, rvalues, glvalues, prvalues, xvalues, help!

C++11 introduced new value categories. We take a look at what they mean to your code.

Nullableattribute and C# 8

Investigating the representation of Nullable Reference Types in C# 8

Modern SAT solvers: fast, neat and underused

Solving Sudoku puzzles using SAT solvers

The duality...

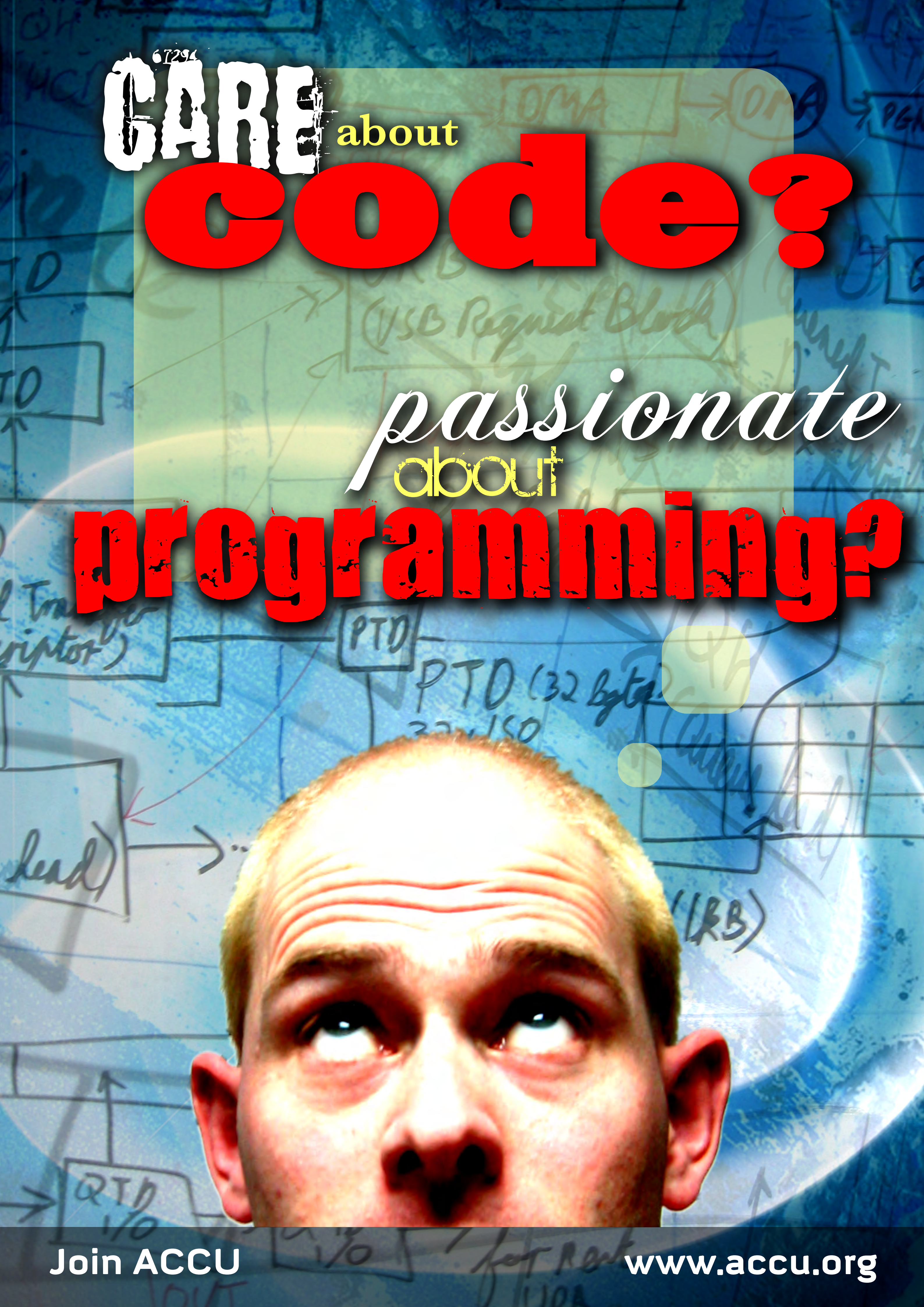
Using genetic algorithms with back-propagation to find solutions

Blockchain-structured programming

On the technology underpinning crypto-currencies

A magazine of ACCU

ISSN: 1354-3172


CARE about
code?
passionate
about
programming?
Join ACCU
www.accu.org

OVERLOAD 150

April 2019

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Andy Balaam
andybalaam@artificialworlds.net

Balog Pal
pasa@lib.h

Ben Curry
b.d.curry@gmail.com

Paul Johnson
paulf.johnson@gmail.com

Klitos Kyriacou
klitos.kyriacou@gmail.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Philipp Schwaha
<philipp@schwaha.net>

Anthony Williams
anthony@justsoftwaresolutions.co.uk

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 151should be submitted by 1st May 2019 and those for Overload 152 by 1st July 2019.

Overload is a publication of the ACCU

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

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.

Copyrights and Trade Marks

This means war!

Careless use of language can upset people. Frances Buontempo asks what we can do to make the developer experience better.

Being the 150th edition of *Overload*, an editorial was called for. However, I have not managed to think for long enough to write one. Instead, we have waged war on our house, in an attempt at a long overdue spring clean. Why do I say 'waged war'? It was the first phrase that sprang to mind, and it did feel like a battle. We can now find things in the cupboard under the stairs. Previously, I was afraid to open the door, for fear of being buried under a pile of goodness knows what. Injuries were sustained, well I broke a nail. However, it was worth the fight. We did well.

Could I describe the tidy up in less military terms? Certainly. We had a tidy up. That doesn't sound as attention grabbing, click-baity, or exciting. Is this the only reason we use clichéd phrases for titles, talks or even code? Niels Dekker's lightning talk at *C++ On Sea*, 'noexcept considered harmful???', ended up on reddit [Dekker19a], accused of being a 'snowclone headline'. I personally loved the talk, and am curious to see the discussion continue. Does `noexcept` slow your code down? Niels has asked an interesting question, in a considered manner. I personally think this was exactly the right title for his talk. He's shared resources on his github page if you want to investigate further [Dekker19b]. The meme title, 'X considered harmful' has a long history, and was originally click-bait. Dijkstra's short letter to the *Communication of the ACM* in 1968 was given the title by the editor, presumably because that sounds more punchy than the suggested 'A Case Against the Goto Statement' [Wikipedia-1]. I say 'punchy', which sounds a tad violent. Why do we use the terms we use? And why are so many of them violent or warlike in nature?

Much of the history of computing is based in military research. Wars and rumours of wars can produce magic money trees, or at least extra sources of funding, for research and development into new technology. Bletchley Park became a hub of industrious innovation during the Second World War. I suspect these roots lead us to use military phrases, often without realising. Why do we have variables or functions named `foo` and `bar` in our code? Kate Gregory's 'Oh the humanity!' keynote at *C++ On Sea* [Gregory19] reminded us the original spelling was 'fubar', meaning something like fouled up beyond all recognition. At what point this became 'foo' instead of 'fu' is not clear. 'Snafu' is another related term. Situation normal, all fouled up. In fact, it seems the American military created a series of cartoons, entitled *Private Snafu*, [Wikipedia-2] to train service personnel about security, safety and protocol. There's a suggestion that these terms were partly introduced to ridicule military acronyms [Wikipedia-3]. You can certainly hide some horror behind TLAs (three letter acronyms), and encourage people to use them without thinking. "Ours is not to reason why, ours is but to do and die," as the saying goes. Not literally true for most programmers. Literally true for the light brigade, whom Tennyson was writing about in his poem. Sometimes we use whole words rather than TLAs. Kill script, anyone? Deadline? To be fair, there is only sketchy evidence this originally referred to shooting prisoners who tried to cross boundaries, or even imaginary lines [Online Etymology Dictionary]. So much violence.

Despite software teams tending to use military terms, we are not in a war room. I watched a talk by Portia Tung a while ago, called 'Enterprise Gardening' [Tung12]. She discussed the use of bullying, dehumanising terminology, with roots in warfare. Her main point was metaphor gives shapes to our thoughts, language and behaviour. "It's important we do gardening instead of pick fights." Allowing plants, or people to grow and thrive is a much better aim than fighting a battle. Plants need light, water and food. So do people. Kate's keynote touched on similar points, for example, changing a variable name from `errorMessage` to `helpMessage` might give you a clearer perspective on what to write in it. What happens in your head if you say 'user help' instead of PEBKAC or 'user error'? RTFM seems harsh. Though I do remind myself to do this sometimes, and make sure I'm reading the correct version of a manual. How many times have I tried something in Python while reading v2.x instead of 3.x, or ElasticSearch, though I've lost track of ES version numbers, since they move so fast. I have used the phrase RTFM on many occasions. In fact, one of my favourite books is called *RTFM: Red Team Field Manual*. [Clark14]. There is a blue team field manual too. It has lots of clearly written scripts, for use in cyber-security CTF (Capture The Flag) training sessions [CTF]. The blue team defends the 'flag' – system, network or similar – while the red team tries to take over. Now there's a thing. Cyber-security is absolutely full of military jargon. And computer gaming. Offensive? Humorous? To be considered harmful? I love the book because it has a glider on the cover and is small. Many books are far too big. This fits in my handbag easily. Five stars.

Military terms do have a place. I was surprised to find some people didn't know the origins of 'foobar'. I don't remember how I discovered its background. I do have a tendency to speak up when I don't understand something, so probably asked what on earth someone was on about when they first used the phrase at me. If you don't know something, speak up. Or look it up. Finding the origins of meaning of words, AKA etymology, is informative. And helps me concentrate on how to spell words from time to time. I wonder what the etymology of etymology is? Will I break the universe if I look that up? Nope. All still there. The internet mentioned something about PIE. I digress. Anyway, many programmers' language involves war stories, fire-fighting and Foo, Bar as well as Baz.

This warfare and fire-fighting talk isn't always negative though. I recently saw a conversation on twitter between parents with fire-fighters in their

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.

family. They were discussing protocols for handing over a small baby. It went something like, "I'm handing you the baby" to which the response "I have the baby" was required before the first person would let go. This ensures a vulnerable person, or small baby, is held securely. Protocols, pre-flight/commit/release checklists have a place. They make sure you are communicating clearly, and paying attention in high pressure situations. Using a protocol or procedure to keep things running smoothly is a good thing.

Broadening out the discussion, the problem with waging war in a software context is the potential collateral damage. People can get hurt. Maybe no-one dies, but feelings get hurt. Shouting and swearing happens. Blame gets attributed, while problems don't get fixed. Svn blame or praise; that is the question. People have been talking about and measuring user experience, Ux, for a long time and its value is well recognised. I've recently noticed developer experience, Dx, being talked about too. What is your experience as a developer like? What does this involve? Team structure? Ways of working? Your office? Your desk and keyboard? Perhaps that's more DI; developer interface. Your toolchain? How confident you are when you commit code? How code reviews are conducted? Is the rebuild button right next to the build button in your IDE? If you use Visual Studio and have ever hit the wrong button wasting hours, consider adding to the Developer Community 'Option to confirm rebuild or clean all' [VS Dev Community]. You are not alone.

I can't find a precise definition for Dx, though many blog posts talk more about the toolchain than the overall developer experience. For me, Dx includes the team you work with. Kind, supportive team members count as much as, if not more than, stable build tools, quick-running tests and a comfortable chair. That said, you can still make your life easier if you observe what is slowing you down, or making you annoyed. If you change one line of code and two projects build, ask why. Find out and fix it. If you come to a project with an old FORTRAN code base and a word document showing how to build it, create a make file instead. Typing 'make' is much simpler than having to re-read a large document and copy/paste commands, which end up with non-Unicode characters in. If the Wiki instructions are out of date, update them.

A podcast about Dx [Boak] talks about developers being people too. Ux came about to make sure people have a good experience using software. Well, probably to ensure buy-in and customer loyalty, if you are a cynic. If you download something on your phone and can't get it working in 30 seconds, you will probably give up and find a different one. How many dev tools have you got working in a matter of seconds? How long does the on-boarding process take for a new team member? How do you know when they are on-board? This podcast hinted at many developers not liking quick start wizard tools, preferring to figure out complicated stuff by themselves, thereby gaining some pride. I'm not sure how generally true that is. Some people do tend towards building up their secret knowledge, becoming the team wizard or warlock for specific problems. Some people are better at sharing the knowledge and leaving simple-to-use tools for others.

If you are a developer on a live product, is it too easy to break things? Do you have root permissions? How easy is it to follow the logging from your system? Do errors leave you confused? Are most of your errors 'normal'? Are warnings actually critical? What about compiler errors? To quote a recent Valentine's tweet:

> Roses are red
> Violets are blue
> initializing argument 1 of std::_Rb_tree_iterator<_Val, _Ref, _Ptr>::_Rb_tree_iterator(std::_Rb_tree_node<_Val>*)[with _Val= std::pair<const int, double>, _Ref= std::pair<const int, double>&, _Ptr = std::pair<const int, double>*]
> On line 22

Does a failing test give you a clear message? If you see "Expected True, actual false" that might not be much help. Instead of adding a breakpoint to find out the values, consider including the values in the assert, so you can see immediately what happened. Wage war against the things that cause you grief, not people. Instead of fixing the blame, fix the problems. What can you do you make your own Dx better?

Many developers I know put up with the process being harder than it has to be. Perhaps there is an element of feeling like a tenth Dan wizard master if you figure out how to do something complicated, which might leave you tempted to keep things as they are. Other times, I suspect fear stops people changing things. I've frequently heard people saying, "It only takes a couple of minutes." That sounds reasonable, but if you can automate that down to a few seconds, it's worth considering. All the seconds add up. Chris Oldwood wrote about keeping things clean and tidy in a recent edition of the magazine [Oldwood18]. He mentioned lashing out, and generally unpleasant behaviour. He even mentioned the Second World War. Perhaps Chris' piece encouraged me to tidy up our house a bit, and start musing on warfare, though trying to blame him for my continued failure to write an editorial might be taking it too far. If you want to see a proper editorial in *Overload*, remember anyone is welcome to try their hand as a guest editor. Get in touch. Remember, *Overload* is available online so is read by many non-ACCU members, which is fine. If you'd rather do something a little less public, you can also volunteer to be guest editor for *CVu* instead. Get in touch.

So, war – what is it good for? Absolutely nothing, apart from technical innovations, and thinking about clear ways of communicating. Pick your battles. Be kind. Be aware of the language you are using, and its history and real meaning. If you can't follow all of the strange in-house acronyms when you start a new gig, or panic while learning something new as soon as 'foo' or 'widget' gets mentioned, take a breath. You are not alone. Let's see what we can do to ensure we all have a great developer experience.

We have met the enemy and he is us
~ Pogo Possum [Wikipedia-4]

References

[Boak] Steve Boak, David Dollar and Justin Baker 'Don't Make Me Code: Ep. #5, Developers Are People Too', available at: https://soundcloud.com/heavybit/dont-make-me-code-ep-5-developers-are-people-too

[Clark14] Ben Clark (2014) *RTFM: Red Team Field Manual* CreateSpace Independent Publishing Platform; 1.0 edition (11 Feb. 2014)

[CTF] https://ctf.hacker101.com/ (for example)

[Dekker19a] Niels Dekker (2019) 'noexcept considered harmful???', presented at *Cpp On Sea* 2019, available at: https://www.reddit.com/r/cpp/comments/apg0yk/noexcept_considered_harmfull_benchmark_and

[Dekker19b] Niels Dekker (2019), resources on github: https://github.com/N-Dekker/noexcept_benchmark

[Gregory19] Kate Gregory (2019) 'Oh the humanity!', presented at *Cpp On Sea 2019*, available at: https://cpponsea.uk/sessions/keynote-oh-the-humanity.html, and search on YouTube for CppOnSea

[Oldwood18] Chris Oldwood (2018) 'Afterwood', *Overload* 148, Dec 2018, available at: https://accu.org/index.php/journals/2584

[Online Etymology Dictionary] https://www.etymonline.com/word/deadline

[Tung12] Portia Tung (2012) 'Enterprise Gardening', available at: https://www.slideshare.net/portiatung/enterprise-gardening

[VS Dev Community] https://developercommunity.visualstudio.com/content/idea/432348/option-to-confirm-rebuild-or-clean-all.html

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

[Wikipedia-2] https://en.wikipedia.org/wiki/Private_Snafu

[Wikipedia-3] https://en.wikipedia.org/wiki/SNAFU

[Wikipedia-4] Pogo (comic strip): https://en.wikipedia.org/wiki/Pogo_(comic_strip)#%22We_have_met_the_enemy_and_he_is_us.%22

NullableAttribute and C# 8

C# 8 will bring many new features. Jon Skeet investigates the metadata representation of Nullable Reference Types.

Background: Noda Time and C# 8

C# 8 is nearly here. At least, it's close enough to being 'here' that there are preview builds of Visual Studio 2019 available that support it. Unsurprisingly, I've been playing with it quite a bit.

In particular, I've been porting the Noda Time source code [Skeet-1] to use the new C# 8 features. The master branch of the repo is currently the code for Noda Time 3.0, which won't be shipping (as a GA release) until after C# 8 and Visual Studio 2019 have fully shipped, so it's a safe environment in which to experiment.

While it's possible that I'll use other C# 8 features in the future, the two C# 8 features that impact Noda Time most are *nullable reference types* and *switch expressions*. Both sets of changes are merged into master now, but the pull requests are still available so you can see just the changes:

- PR 1240: Support nullable reference types [Skeet-2]
- PR 1264: Use switch expressions [Skeet-3]

The switch expressions PR is much simpler than the nullable reference types one. It's entirely an implementation detail… although admittedly one that confused docfx, requiring a few of those switch expressions to be backed out or moved in a later PR.

Nullable reference types are a much, much bigger deal. They affect the public API, so they need to be treated much more carefully, and the changes end up being spread far wide throughout the codebase. That's why the switch expression PR is a single commit, whereas nullable reference types is split into 14 commits – mostly broken up by project.

Reviewing the public API of a nullable reference type change

So I'm now in a situation where I've got nullable reference type support in Noda Time. Anyone consuming the 3.0 build (and there's an alpha available for experimentation purposes [NodaTime]) from C# 8 will benefit from the extra information that can now be expressed about parameters and return values. Great!

But how can I be confident in the changes to the API? My process for making the change in the first place was to enable nullable reference types and see what warnings were created. That's a great starting point, but it doesn't necessarily catch everything. In particular, although I *started* with the main project (the one that creates NodaTime.dll), I found that I needed to make more changes later on, as I modified other projects.

Just because your code compiles without any warnings with nullable reference types enabled doesn't mean it's 'correct' in terms of the API you want to expose.

Jon Skeet is a Staff Software Engineer at Google, working on making Google Cloud Platform rock for C# developers. He's a big C# nerd, enjoying studying the details of language evolution. He is @jonskeet on Twitter, and his email address is on his Stack Overflow profile.

```
// Allowing null input, producing nullable output
public static string? Identity(string? input)
  => input;


// Preventing null input, producing non-nullable
// output
public static string Identity(string input)
{
  // Convenience method for nullity checking.
  Preconditions.CheckNotNull(input,
    nameof(input));
  return input;
}
```

Listing 1

For example, consider this method:

```
public static string Identity(string input)
  => input;
```

That's entirely valid C# 7 code, and doesn't require any changes to compile, warning-free, in C# 8 with nullable reference types enabled. But it may not be what you actually want to expose. I'd argue that it should look like *one* of the options in Listing 1.

If you were completely diligent when writing tests for the code before C# 8, it should be obvious which is required – because you'd presumably have something like:

```
[Test]
public void Identity_AcceptsNull()
{
  Assert.IsNull(Identity(null));
}
```

That test would have produced a warning in C# 8, and would have suggested that the null-permissive API is the one you wanted. But maybe you forgot to write that test. Maybe the test you *would* have written was one that would have shown up a need to put that precondition in. It's entirely possible that you write much more comprehensive tests than I do, but I suspect most of us have *some* code that isn't explicitly tested in terms of its null handling.

The important part take-away here is that even code that hasn't changed in appearance can change *meaning* in C# 8… so you really need to review any public APIs. How do you do that? Well, you could review the *entire*

Nullable reference types

C# 8 introduces nullable reference types, which complement reference types the same way nullable value types complement value types. You declare a variable to be a nullable reference type by appending a `?` to the type. For example, `string?` represents a nullable string. You can use these new types to more clearly express your design intent: some variables must always have a value, others may be missing a value. [Microsoft]

It would be oh-so-simple if each parameter or return type could just be nullable or non-nullable. But life gets more complicated than that, with both generics and arrays

```
public class Test
{
#nullable enable
  public void X(string input) {}

  public void Y(string? input) {}
#nullable restore

#nullable disable
  public void Z(string input) {}
#nullable restore
}
```

Listing 2

public API surface you're exposing, of course. For many libraries that would be the simplest approach to take, as a 'belt and braces' attitude to review. For Noda Time that's less appropriate, as so much of the API only deals in value types. While a full API review would no doubt be useful in itself, I just don't have the time to do it right now.

Instead, what I want to review is any API element which is impacted by the C# 8 change – even if the code itself *hasn't* changed. Fortunately, that's relatively easy to do.

Enter NullableAttribute

The C# 8 compiler applies a new attribute to every API element which is affected by nullability. As an example of what I mean by this, consider the code in Listing 2, which uses the **#nullable** directive to control the nullable context of the code.

The C# 8 compiler creates an internal **NullableAttribute** class within the assembly (which I assume it wouldn't if we were targeting a framework that already includes such an attribute) and applies the attribute anywhere it's relevant. So the code in Listing 2 compiles to the same IL as this:

```
using System.Runtime.CompilerServices;
public class Test
{
  public void X([Nullable((byte) 1)]
    string input) {}
  public void Y([Nullable((byte) 2)]
    string input) {}
  public void Z(string input) {}}
}
```

Note how the parameter for **Z** doesn't have the attribute at all, because that code is still *oblivious* to nullable reference types. But both **X** and **Y** have the attribute applied to their parameters – just with different arguments to describe the nullability. 1 is used for not-null; 2 is used for nullable.

That makes it relatively easy to write a tool to display every part of a library's API that relates to nullable reference types – just find all the members that refer to **NullableAttribute**, and filter down to public and protected members.

It's slightly annoying that **NullableAttribute** doesn't have any properties; code to analyze an assembly needs to find the appropriate **CustomAttributeData** and examine the constructor arguments. It's awkward, but not insurmountable.

I've started doing exactly that in the Noda Time repository, and got it to the state where it's fine for Noda Time's API review. It's a bit quick and dirty at the moment. It doesn't show protected members, or setter-only properties, or handle arrays, and there are probably other things I've forgotten about. I intend to improve the code over time and probably move it to my Demo Code repository at some point, but I didn't want to wait until then to write about **NullableAttribute**.

But hey, I'm all done, right? I've explained how **NullableAttribute** works, so what's left? Well, it's not *quite* as simple as I've shown so far.

NullableAttribute in more complex scenarios

It would be oh-so-simple if each parameter or return type could just be nullable or non-nullable. But life gets more complicated than that, with both generics and arrays. Consider a method called **GetNames()** returning a list of strings. All of these are valid:

```
// Return value is non-null, and elements aren't null
List<string> GetNames()

// Return value is non-null, but elements may be null
List<string?> GetNames()

// Return value may be null, but elements aren't null
List<string>? GetNames()

// Return value may be null, and elements may be null
List<string?>? GetNames()
```

So how are those represented in IL? Well, **NullableAttribute** has one constructor accepting a single **byte** for simple situations, but another one accepting **byte[]** for more complex ones like this. Of course, **List<string>** is still *relatively* simple – it's just a single top-level generic type with a single type argument. For a more complex example, imagine **Dictionary<List<string?>, string[]?>**. (A non-nullable reference to a dictionary where each key is a not-null list of nullable strings, and each value is a possibly-null array of non-nullable elements. Ouch.)

The layout of **NullableAttribute** in these cases can be thought of in terms of a pre-order traversal of a tree representing the type, where generic type arguments and array element types are leaves in the tree. The above example could be thought of as the tree in Figure 1.

The pre-order traversal of that tree gives us these values:

- Not null (dictionary)
- Not null (list)
- Nullable (string)

> **When all the elements in the tree are 'not null' or all elements in the tree are 'nullable', the compiler simply uses the constructor with the single-byte parameter instead**

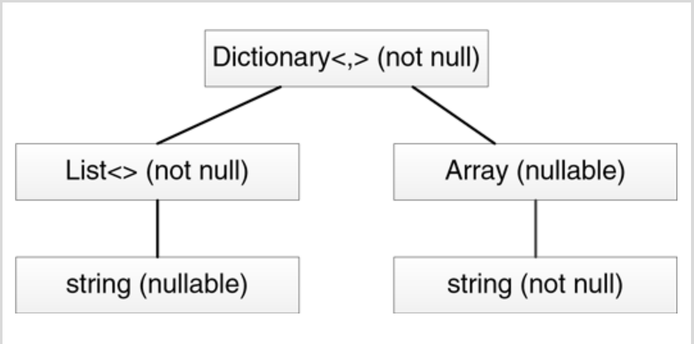


Figure 1

- Nullable (array)
- Not null (string)

So a parameter declared with that type would be decorated like this:

```
[Nullable(new byte[] { 1, 1, 2, 2, 1 })]
```

But wait, there's more!

NullableAttribute in simultaneously-complex-and-simple scenarios

The compiler has one more trick up its sleeve. When all the elements in the tree are 'not null' or all elements in the tree are 'nullable', it simply uses the constructor with the single-byte parameter instead. So `Dictionary<List<string>, string[]>` would be decorated with `Nullable[(byte) 1]` and `Dictionary<List<string?>?, string?[]?>?` would be decorated with `Nullable[(byte) 2]`.

(Admittedly, `Dictionary<,>` doesn't permit null keys anyway, but that's an implementation detail.)

Conclusion

The C# 8 feature of nullable reference types is a really complicated one. I don't think we've seen anything like this since async/await. This article has just touched on one interesting implementation detail. I'm sure there'll be more on nullability over the next few months… ■

References

[Microsoft] Background information: https://devblogs.microsoft.com/dotnet/nullable-reference-types-in-csharp/

[NodaTime] Alpha build: https://www.nuget.org/packages/NodaTime/3.0.0-alpha01

[Skeet-1] https://github.com/nodatime/nodatime

[Skeet-2] PR1240: Support nullable reference types, available at: https://github.com/nodatime/nodatime/pull/1240

[Skeet-3] PR 1264: Use switch expressions, available at: https://github.com/nodatime/nodatime/pull/1264

This article was first published on Jon Skeet's coding blog on 10 February 2019 at https://codeblog.jonskeet.uk/2019/02/10/nullableattribute-and-c-8/

lvalues, rvalues, glvalues, prvalues, xvalues, help!

C++11 introduced new value categories. Anders Schau Knatten explains his way of thinking about them.

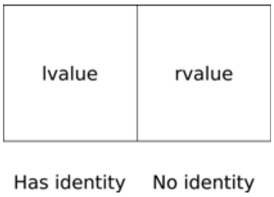
Did you used to have some sort of intuition for what 'lvalue' and 'rvalue' mean? Are you confused about glvalues, xvalues and prvalues, and worry that lvalues and rvalues might also have changed? This article aims to help you develop a basic intuition for all five of them.

First, a warning: This article does not give a complete definition of the five value categories. Instead, I give a basic outline, which I hope will help to have in the back of your mind the next time you need to look up the actual details of one of them.

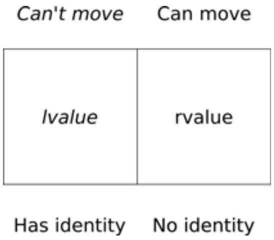
Back before C++11, there were two value categories, lvalue and rvalue. The basic intuition was that lvalues were things with identities, such as variables, and rvalues were expressions evaluating to temporaries (with no identity). Consider these definitions:

```
  Widget w;
  Widget getWidget();
```

If we now use the expression `w` anywhere, it evaluates to the object `w`, which has identity. If we use the expression `getWidget()`, it evaluates to a temporary return value with no identity. Let's visualise it like the diagram on the right.



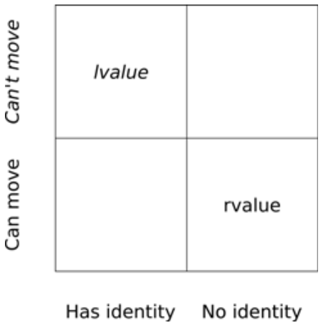
However, along came rvalue references and move semantics. On the surface, the old lvalue/rvalue distinction seems sufficient: Never move from lvalues (people might still be using them), feel free to move from rvalues (they're just temporary anyway). Let's add movability to our diagram.



Why did I put 'Can't move' and 'lvalue' in italics in that diagram? It turns out that you might want to move from certain lvalues! For instance, if you have a variable you won't be using anymore, you can `std::move()` it to cast it to an rvalue reference. A function can also return an rvalue reference to an object with identity.

So as it turns out, whether something has identity, and whether something can be moved from, are *orthogonal properties*! We'll solve the problem of moving from lvalues soon, but first, let's just change our diagram to reflect our new orthogonal view of the world.

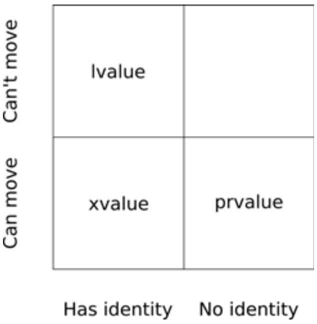
Clearly, there's a name missing in the lower left corner here. (We can ignore the top right corner, temporaries which can't be moved from is not a useful concept.)



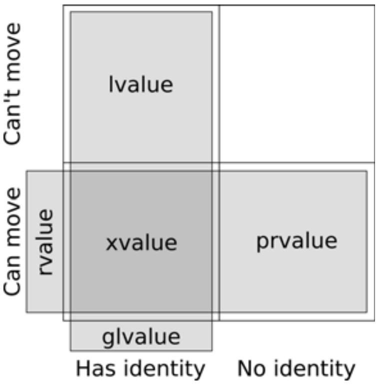
C++11 introduces a new value category 'xvalue', for lvalues which can be moved from. It might help to think of 'xvalue' as 'eXpiring lvalue', since they're probably about to end their lifetime and be moved from (for instance a function returning an rvalue reference).

This article was first published on Anders Schau Knatten's *C++ on a Friday* blog on 9 March 2019 at https://blog.knatten.org/2018/03/09/lvalues-rvalues-glvalues-prvalues-xvalues-help/

In addition, what was formerly called 'rvalue' was renamed to 'prvalue', meaning 'pure rvalue'. These are the three basic value categories, shown in the diagram on the right.



But we're not quite there yet, what's a 'glvalue', and what does 'rvalue' mean these days? It turns out that we've already explained these concepts! We just haven't given them proper names yet.



A glvalue, or 'generalized lvalue', covers exactly the 'has identity' property, ignoring movability. An rvalue covers exactly the 'can move' property, ignoring identity. And that's it! You now know all the five value categories.

If you want to go into further detail about this topic, cppreference has a very good article [cppreference]. ■

Reference

[cppreference] Value categories: https://en.cppreference.com/w/cpp/language/value_category

Anders Schau Knatten makes robot eyes at Zivid, where he strives for simplicity, stability and expressive code. He's also the author of CppQuiz and @AffectiveCpp, which strive for none of the above. Anders can be contacted at anders@knatten.org

Modern SAT solvers: fast, neat and underused (part 1 of N)

SAT solvers can quickly find solutions to Boolean Logic problems. Martin Hořeňovský demonstrates how this can be used to solve arbitrary Sudoku puzzles.

Before I started doing research for Intelligent Data Analysis (IDA) group at FEE CTU, I saw SAT solvers as academically interesting but didn't think that they have many practical uses outside of other academic applications. After spending ~1.5 years working with them, I have to say that modern SAT solvers are fast, neat and criminally underused by the industry.

Introduction

Boolean satisfiability problem (SAT) is the problem of deciding whether a formula in boolean logic is satisfiable. A formula is *satisfiable* when at least one interpretation (an assignment of `true` and `false` values to logical variables) leads to the formula evaluating to `true`. If no such interpretation exists, the formula is *unsatisfiable*.

What makes SAT interesting is that a variant of it was the first problem to be proven NP-complete, which roughly means that a lot of other problems can be translated into SAT in reasonable[1] time, and the solution to this translated problem can be converted back into a solution for the original problem.

As an example, the often-talked-about dependency management problem is also NP-Complete and thus translates into SAT[2,3], and SAT could be translated into dependency manager. The problem our group worked on, generating key and lock cuttings based on user-provided lock-chart and manufacturer-specified geometry, is also NP-complete.

To keep this article reasonably short, we will leave master-key systems for another article and instead use Sudoku for practical examples.

Using SAT solvers

These days, SAT almost always refers to CNF-SAT[4], a boolean satisfaction problem for formulas in conjunctive normal form (CNF) [Wikipedia-1]. This means that the entire formula is a conjunction (AND) of clauses, with each clause being a disjunction (OR) of literals. Some examples:

- $(A \vee B) \wedge (B \vee C)$
- $(A \vee B) \wedge C$
- $A \vee B$
- $A \wedge C$

There are two ways to pass a formula to a SAT solver: by using a semi-standard file format known as DIMACS, or by using the SAT solver as a library. In real-world applications, I prefer using SAT solver as a library (e.g. MiniSat for C++ [MiniSAT]), but the DIMACS format lets you prototype your application quickly, and quickly test the performance characteristics of different solvers on your problem.

Martin Hořeňovský is currently a researcher at Locksley.CZ, where he works on solving large master-key systems. In his (rare) free time, he also maintains Catch2, a popular C++ unit testing framework, and he used to teach a course on modern C++ at Czech Technical University in Prague. He can be reached at martin.horenovsky@gmail.com

DIMACS format

DIMACS is a line oriented format, consisting of 3 different basic types of lines.

1. A comment line. Any line that starts with `c` is comment line.
2. A summary line. This line contains information about the kind and size of the problem within the file. A summary line starts with `p`, continues with the kind of the problem (in most cases, `cnf`), the number of variables and the number of clauses within this problem. Some DIMACS parsers expect this line to be the first non-comment line, but some parsers can handle the file without it.
3. A clause line. A clause line consists of space-separated numbers, ending with a `0`. Each non-zero number denotes a literal, with negative numbers being negative literals of that variable, and `0` being the terminator of a line.

As an example, this formula

$$(A \vee B \vee C) \wedge (\neg A \vee B \vee C) \wedge (A \vee \neg B \vee C) \wedge (A \vee B \vee \neg C)$$

would be converted into DIMACS as

```
c An example formula
c
p cnf 3 4
1 2 3 0
-1 2 3 0
1 -2 3 0
1 2 -3 0
```

Minisat's C++ interface

MiniSat is a fairly simple and performant SAT solver that also provides a nice C++ interface and we maintain a modernised fork with CMake integration. The C++ interface to MiniSat uses 3 basic vocabulary types:

- `Minisat::Solver` – Implementation of the core solver and its algorithms.
- `Minisat::Var` – Representation of a variable.

[1] This means polynomial, because when it comes to complexity theory, algorithms with polynomial complexity are generally considered tractable, no matter how high the exponent in the polynomial is, and algorithms with exponential complexity are considered intractable.

[2] At least as long as we assume that:
- To install a package, all its dependencies must be installed
- A package can list specific versions of other packages as dependencies
- Dependency sets of each version of a package can be different
- Only one version of a package can be installed

[3] In fact, various dependency managers in the wild already use SAT solvers, such as Fedora's DNF, Eclipse's plugin manager, FreeBSD's pkg, Debian's apt (optionally), and others.

[4] There are some extensions like XOR-SAT, which lets you natively encode XOR clauses, but these are relatively rare and only used in specialist domain, e.g. cryptanalysis.

> Very few problems are naturally expressed as a logical formula in the CNF format

- `Minisat::Lit` – Representation of a concrete (positive or negative) literal of a variable.

The difference between a variable and a literal is that the literal is a concrete 'evaluation' of a variable inside a clause. As an example, formula $(A \lor B \lor \neg C) \land (\neg A \lor \neg B)$ contains 3 variables, A, B and C, but it contains 5 literals, A, ¬A, B, ¬B and ¬C.

MiniSat's interface also uses one utility type: `Minisat::vec<T>`, a container similar to `std::vector`, that is used to pass clauses to the solver.

The example in Listing 1 uses MiniSat's C++ API to solve the same clause as we used in the DIMACS example.

```
// main.cpp:
#include <minisat/core/Solver.h>
#include <iostream>
int main() {
  using Minisat::mkLit;
  using Minisat::lbool;

  Minisat::Solver solver;
  // Create variables
  auto A = solver.newVar();
  auto B = solver.newVar();
  auto C = solver.newVar();
  // Create the clauses
  solver.addClause( mkLit(A), mkLit(B), mkLit(C));
  solver.addClause(~mkLit(A), mkLit(B), mkLit(C));
  solver.addClause( mkLit(A), ~mkLit(B),
    mkLit(C));
  solver.addClause( mkLit(A), mkLit(B),
    ~mkLit(C));
  // Check for solution and retrieve model if found
  auto sat = solver.solve();
  if (sat) {
    std::clog << "SAT\n"
              << "Model found:\n";
    std::clog << "A := "
              << (solver.modelValue(A) == l_True)
              << '\n';
    std::clog << "B := "
              << (solver.modelValue(B) == l_True)
              << '\n';
    std::clog << "C := "
              << (solver.modelValue(C) == l_True)
              << '\n';
  } else {
    std::clog << "UNSAT\n";
    return 1;
  }
}
```

Listing 1

```
cmake_minimum_required (VERSION 3.5)
project (minisat-example LANGUAGES CXX)

set(CMAKE_CXX_EXTENSIONS OFF)

find_package(MiniSat 2.2 REQUIRED)

add_executable(minisat-example
  main.cpp
)
target_link_libraries(minisat-example
  MiniSat::libminisat)
```

Listing 2

Because all of our clauses have length ≤ 3, we can get away with just using utility overloads that MiniSat provides, and don't need to use `Minisat::vec` for the clauses.

We will also need to build the binary. Assuming you have installed our fork of MiniSat (either from GitHub [MiniSAT] or from vcpkg [vcpkg]), it provides proper CMake integration and writing the `CMakeLists.txt` is trivial (see Listing 2).

Building the example and running it should[5] give you this output:

```
SAT
Model found:
A := 0
B := 1
C := 1
```

Conversion to CNF

Very few problems are naturally expressed as a logical formula in the CNF format, which means that after formulating a problem as a SAT, we often need to convert it into CNF. The most basic approach is to create an equivalent formula using De-Morgan laws, distributive law and the fact that two negations cancel out. This approach has two advantages: one, it is simple and obviously correct. Two, it does not introduce new variables. However, it has one significant disadvantage: some formulas lead to exponentially large CNF conversion.

The other approach is to create an equisatisfiable[6] CNF formula, but we won't be covering that in this article.

Some common equivalencies are in Table 1, overleaf.

[5] When using the library interface of MiniSat, it defaults to being entirely deterministic. This means that if you are using the same version of MiniSat, the result will always be the same, even though there are different models.

[6] Two formulas, f1 and f2, are equisatisfiable when f1 being satisfied means that f2 is also satisfied and vice versa.

you don't have to remember these identities, but knowing at least some of them is much faster than deriving them from the truth tables every time

Original clause	Equivalent clause
$\neg\neg\alpha$	α
$\alpha \Rightarrow \beta$	$\neg\alpha \vee \beta$
$\neg(\alpha \wedge \beta)$	$\neg\alpha \vee \neg\beta$
$\neg(\neg\alpha \wedge \neg\beta)$	$\alpha \vee \beta$
$(\alpha \wedge \beta) \vee \gamma$	$(\alpha \vee \gamma) \wedge (\beta \vee \gamma)$
$\alpha \Leftrightarrow \beta$	$(\alpha \Rightarrow \beta) \wedge (\alpha \Leftarrow \beta)$

Table 1

Obviously, you don't have to remember these identities, but knowing at least some of them (implication) is much faster than deriving them from the truth tables every time.

Solving Sudoku using SAT

With this background, we can now look at how we could use a real-world problem, such as Sudoku, using a SAT solver. First, we will go over the rules of Sudoku [Wikipedia-2] and how they can be translated into (CNF-)SAT. Then we will go over implementing this converter in C++ and benchmarking the results.

Quick overview of Sudoku

Sudoku is a puzzle where you need to place numbers 1–9 into a 9×9 grid consisting of 9 3×3 boxes[7], following these rules:

1. Each row contains all of the numbers 1–9
2. Each column contains all of the numbers 1–9
3. Each of the 3×3 boxes contains all of the numbers 1–9

We can also restate these rules as:

1. No row contains duplicate numbers
2. No column contains duplicate numbers
3. No 3×3 box contains duplicate numbers

Because these rules alone wouldn't make for a good puzzle, some of the positions are pre-filled by the puzzle setter, and a proper Sudoku puzzle should have only one possible solution.

Translating the rules

The first step in translating a problem to SAT is to decide what should be modelled via variables, and what should be modelled via clauses over these variables. With Sudoku, the natural thing to do is to model positions as variables, but in SAT, each variable can only have 2 values: `true` and `false`. This means we cannot just assign each position a variable, instead we have to assign each combination of position *and* value a variable. We will denote such variable as $x^v_{r,c}$. If variable $x^v_{r,c}$ is set to `true`, then the number in *r*-th row and *c*-th column is *v*.

Using this notation, let's translate the Sudoku rules from the previous section into SAT.

Rule 1 (No row contains duplicate numbers)

$$\forall (r,v) \in (rows \times values) : \text{exactly-one}(x^v_{r,0}, x^v_{r,1}, \ldots, x^v_{r,8})$$

In plain words, for each row and each value, we want exactly one column in that row to have that value. We do that by using a helper called exactly-one, that generates a set of clauses that ensure that exactly one of the passed-in literals evaluate to `true`.

We will see how to define exactly-one later. First, we will translate the other Sudoku rules into these pseudo-boolean formulas.

Rule 2 (No column contains duplicate numbers)

$$\forall (c,v) \in (columns \times values) : \text{exactly-one}(x^v_{0,c}, x^v_{1,c}, \ldots, x^v_{8,c})$$

This works analogically with Rule 1, in that for each column and each value, we want exactly one row to have that value.

Rule 3 (None of the 3x3 boxes contain duplicate numbers)

This rule works exactly the same way as the first two: for each box and each value, we want exactly one position in the box to have that value.

$$\begin{aligned}&\forall (box, value) \in (boxes \times values)\\ &\quad : \text{exactly-one}(\text{literals-in-box}(box, value))\end{aligned}$$

Even though it seems to be enough at first glance, these 3 rules are in fact *not* enough to properly specify Sudoku. This is because a solution like this one:

	0	1	2	3	4	5	6	7	8
0	x	.	.	.	.	.	.	.	.
1	.	.	.	x	.	.	.	.	.
2	.	.	.	.	.	.	x	.	.
3	.	x	.	.	.	.	.	.	.
4	.	.	.	.	x	.	.	.	.
5	.	.	.	.	.	.	.	x	.
6	.	.	x	.	.	.	.	.	.
7	.	.	.	.	.	x	.	.	.
8	.	.	.	.	.	.	.	.	x

where **x** denotes a position where all variables are set to `true` and `.` denotes a position where no variables are set to `true`, is valid according to the rules as given to the SAT solver.

[7] There is also a notion of generalised sudoku, where you have to fill in numbers 1-N in NxN grid according to the same rules. It is proven to be NP-complete.

There is no way to encode numeric constraints natively in boolean logic, but often you can decompose these constraints into simpler terms and encode these

Unstated assumptions

When translating problems into SAT, be very careful not to rely on unstated assumptions. While an assumption might seem common sense to a human, SAT solvers (and computers in general) do not operate on common sense, and will happily find a solution that is valid according to your specification of the problem but does not make sense in the context of human expectations.

This is because we operate with an unstated assumption, that each position can contain only one number. This makes perfect sense to a human, but the SAT solver does not understand the meaning of the variables, it only sees clauses it was given.

We can fix this simply by adding one more rule.

Rule 4 (Each position contains exactly one number)

$$\forall (r,c) \in (rows \times columns) : \text{exactly-one}(x^1_{r,c}, x^2_{r,c}, \ldots x^9_{r,c})$$

With this rule in place, we have fully translated the rules of Sudoku into SAT and can use a SAT solver to help us solve sudoku instances. But before we do that, we need to define the helper our description of Sudoku relies on.

Exactly-one helper

There is no way to encode numeric constraints natively in boolean logic, but often you can decompose these constraints into simpler terms and encode these. Many research papers have been written about the efficient encoding of specific constraints and other gadgets, but in this article, we only need to deal with the most common and one of the simplest constraints possible: exactly one of this set of literals has to evaluate to true. Everyone who works with SAT often can write this constraint from memory, but we will derive it from first principles because it shows how more complex constraints can be constructed.

The first step is to decompose the constraint $x == n$ into two parts: $x \geq n$ and $x \leq n$, or for our specific case, $x \geq 1$ and $x \leq 1$, or, translated into the world of SAT, at least 1 literal has to evaluate to **`true`**, and no more than 1 literal can evaluate to **`true`**. Forcing *at least one* literal to be true is easy, just place all of them into one large disjunction:

$$\bigvee_{lit \in Literals} lit$$

Forcing *at most* one literal to be true seems harder, but with a slight restating of the logic, it also becomes quite easy. At most one literal is true when there is *no pair of literals where both of the literals are true at the same time*.

$$\neg \bigvee_{i\in 1..n, j\in 1..n, i\neq j} lit_i \wedge lit_j$$

This set of clauses says exactly that, but it has one problem: it is not in CNF. To convert them into CNF, we have to use some of the identities in the previous section on converting formulas to CNF. Specifically, the fact that negating a disjunction leads to a conjunction of negations, and negating a conjunction leads to a disjunction of negations. Using these, we get the following CNF formula:

$$\bigvee_{i\in 1..n, j\in 1..n, i\neq j} \neg lit_i \vee \neg lit_j$$

We can also use the fact that both conjunction and disjunction are commutative (there is no difference between $x \wedge y$ and $y \wedge x$) to halve the number of clauses we create, as we only need to consider literal pairs where $i < j$.

Now that we know how to limit the number of 'true' literals to both at least 1 and at most 1, limiting the number of 'true' literals to *exactly* 1 is trivial; just apply both constraints at the same time via conjunction.

C++ implementation

Now that we know how to describe Sudoku as a set of boolean clauses in CNF, we can implement a C++ code that uses this knowledge to solve arbitrary Sudoku. For brevity, this post will only contain relevant excerpts, but you can find the entire resulting code on GitHub[8] [Sudoku].

The first thing we need to solve is addressing variables, specifically converting a (row, column, value) triple into a specific value that represents it in the SAT solver. Because Sudoku is highly regular, we can get away with linearizing the three dimensions into one, and get the number of variable corresponding to $x^v_{r,c}$ as $r \times 9 \times 9 + c \times 9 + v$. We can also use the fact that **`Minisat::Var`** is just a plain **`int`** numbered from 0 to avoid storing the variables at all because we can always compute the corresponding variable on-demand:

```
Minisat::Var toVar(int row, int column,
  int value)
{
  return row * columns * values
  + column * values + value;
}
```

Now that we can quickly retrieve the SAT variable from a triplet of (row, column, value), but before we can use the variables, they need to be allocated inside the SAT solver:

```
void Solver::init_variables() {
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < columns; ++c) {
      for (int v = 0; v < values; ++v) {
        static_cast<void>(solver.newVar());
      }
    }
  }
}
```

With the variables allocated, we can start converting the SAT version of Sudoku rules into C++ code.

- **Rule 1** (No row contains duplicate numbers) is shown in Listing 3.

[8] The real code differs in places, especially in that it is coded much more defensively and contains more validity checking in the form of assertions.

Because only one of the variables for any given position can be set to true, the value corresponding to that specific variable is the value of the given position

```
for (int row = 0; row < rows; ++row) {
  for (int value = 0; value < values; ++value) {
    Minisat::vec<Minisat::Lit> literals;
    for (int column = 0; column < columns;
         ++column) {
      literals.push(Minisat::mkLit(toVar(row,
         column, value)));
    }
    exactly_one_true(literals);
  }
}
```

Listing 3

- **Rule 2** (No column contains duplicate numbers) is shown in Listing 4.
- **Rule 3** (None of the 3x3 boxes contain duplicate numbers)

  This rule results in the most complex code, as it requires two iterations – one to iterate over all of the boxes and one to collect variables inside each box. However, the resulting code is still fairly trivial (see Listing 5).
- **Rule 4** (Each position contains exactly one number) is shown in Listing 6.

We also need to define the **`exactly_one_true`** helper (Listing 7).

With these snippets, we have defined a model of Sudoku as SAT. There are still 2 pieces of the solver missing: a method to specify values in the pre-filled positions of the board and a method that extracts the found solution to the puzzle.

Fixing the values in specific positions is easy, we can just add a unary clause for each specified position (see Listing 8).

Because the only way to satisfy a unary clause is to set the appropriate variable to the polarity of the contained literal, this forces the specific position to always contain the desired value.

```
for (int column = 0; column < columns; ++column) {
  for (int value = 0; value < values; ++value) {
    Minisat::vec<Minisat::Lit> literals;
    for (int row = 0; row < rows; ++row) {
      literals.push(Minisat::mkLit(toVar(row,
        column, value)));
    }
    exactly_one_true(literals);
  }
}
```

Listing 4

```
for (int r = 0; r < 9; r += 3) {
  for (int c = 0; c < 9; c += 3) {
    for (int value = 0; value < values; ++value) {
      Minisat::vec<Minisat::Lit> literals;
      for (int rr = 0; rr < 3; ++rr) {
        for (int cc = 0; cc < 3; ++cc) {
          literals.push(Minisat::mkLit(toVar
            (r + rr, c + cc, value)));
        }
      }
      exactly_one_true(literals);
    }
  }
}
```

Listing 5

To retrieve a solution, we need to be able to determine a position's value. Because only one of the variables for any given position can be set to true, the value corresponding to that specific variable is the value of the given position (see Listing 9).

With the solver finished, we can go on to benchmarking its performance.

```
for (int r = 0; r < 9; r += 3) {
  for (int c = 0; c < 9; c += 3) {
    for (int value = 0; value < values; ++value) {
      Minisat::vec<Minisat::Lit> literals;
      for (int rr = 0; rr < 3; ++rr) {
        for (int cc = 0; cc < 3; ++cc) {
          literals.push(Minisat::mkLit(toVar(
            r + rr, c + cc, value)));
        }
      }
      exactly_one_true(literals);
    }
  }
}
for (int row = 0; row < rows; ++row) {
  for (int column = 0; column < columns; ++column)
  {
    Minisat::vec<Minisat::Lit> literals;
    for (int value = 0; value < values; ++value) {
      literals.push(Minisat::mkLit(toVar(
        row, column, value)));
    }
    exactly_one_true(literals);
  }
}
```

Listing 6

```
void Solver::exactly_one_true(
    Minisat::vec<Minisat::Lit> const& literals) {
  solver.addClause(literals);
  for (size_t i = 0; i < literals.size(); ++i) {
    for (size_t j = i + 1; j < literals.size();
      ++j){
      solver.addClause(~literals[i],
        ~literals[j]);
    }
  }
}
```

Listing 7

```
bool Solver::apply_board(board const& b) {
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < columns; ++col) {
      auto value = b[row][col];
      if (value != 0) {
        solver.addClause(Minisat::mkLit(toVar(row,
          col, value - 1)));
      }
    }
  }
  return ret;
}
```

Listing 8

Benchmarks

As far as I could tell from a cursory search, there are no standard test suites for benchmarking Sudoku solvers. I decided to follow Peter Norvig's blog post [Norvig] about his own Sudoku solver and use a set of 95 hard Sudokus [Sudoku-data]for measuring the performance of my solver.

The measurements were done on PC with factory-clocked i5-6600K CPU @ 3.5 GHz, the code was compiled using g++ under Windows Subsystem for Linux, and each input was run 10 times. After that, I took the mean of the results for each problem and put all of them into a boxplot. Since I am a proponent of LTO builds, I also compiled the whole thing, including MiniSat, with LTO enabled, and then benchmarked the binary.

Figure 1 shows the results.

As you can see, the LTO build performed somewhat better, but not significantly so. What is interesting, is that the number of outliers above the box, and the relative lengths of the whiskers, suggest that the underlying distribution of solver's running time over all of the inputs is heavy-tailed. This means that the longest-running inputs will need significantly longer to be solved than the others, and it is a common attribute of solvers for NP-complete problems. This is because a single wrong decision during the search for a solution can lengthen the total runtime significantly.

```
board Solver::get_solution() const {
  board b(rows, std::vector<int>(columns));
  for (int row = 0; row < rows; ++row) {
    for (int col = 0; col < columns; ++col) {
      for (int val = 0; val < values; ++val) {
        if (solver.modelValue(toVar(row, col,
            val)).isTrue()) {
          b[row][col] = val + 1;
          break;
        }
      }
    }
  }
  return b;
}
```

Listing 9

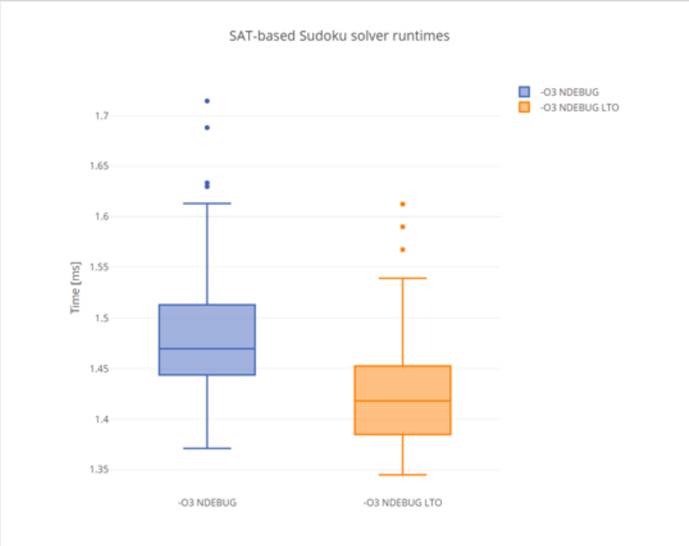


Figure 1

There is one more question to answer, namely, how does this performance compare with high-performance Sudoku-specialized solvers? I picked two, ZSolver [Zhouyundong12] and fsss2 [Dobrichev], and tried running them on the same set of problems. Not too surprisingly, they both outperformed our SAT-based solver badly. The sort of 'converting' solver we wrote will always be slower than a well-tuned specialised solver, but they do have some advantages that can make them desirable. As an example, I have no prior domain-specific knowledge about solving Sudokus, but I was able to write the SAT-based Sudoku solver in less than 2 hours. It is also much more readable and extendable[9].

That is all for part 1, but I have much more I want to say about SAT solvers, so you can expect more posts on my blog both about using them, and about their internals and the theory behind why are they so fast. ■

References

[Dobrichev] fsss2: https://github.com/dobrichev/fsss2

[MiniSAT] Production-ready MiniSAT: https://github.com/master-keying/minisat/

[Norvig] Peter Norvig 'Solving Every Soduko Puzzle', available at: http://norvig.com/sudoku.html

[Sudoku] Sudoku example code: https://github.com/horenmar/sudoku-example

[Sudoku-data] https://raw.githubusercontent.com/horenmar/sudoku-example/master/inputs/benchmark/top95.txt

[vcpkg] Installation of MiniSAT: https://github.com/Microsoft/vcpkg

[Wikipedia-1] 'Conjunctive normal form': https://en.wikipedia.org/wiki/Conjunctive_normal_form

[Wikipedia-2] 'Sudoku': https://en.wikipedia.org/wiki/Sudoku

[Zhouyundong12] ZSolver: http://forum.enjoysudoku.com/3-77us-solver-2-8g-cpu-testcase-17sodoku-t30470-90.html#p216748

[9] Try reading through the code of the linked solvers and imagine extending them to work with 18x18 boards. With our solver, it is just a matter of changing 3 constants on top of a file from 9 to 18.

This article was first published on Martin Hořeňovský's blog, *The Coding Nest*, on 3 August 2018 at https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/

The Duality…

Genetic algorithms can find solutions that other algorithms might miss. Anders Modén discusses their use in conjunction with back-propagation for finding better solutions.

Back then…

Many years ago, I was working on a numerical problem. I needed to solve an optimization problem in my work and I had a lot of trouble finding a good solution. I tried a lot of algorithms like the Gauss-Newton [Wikipedia-1] equation solver for least-square problems and I tried the Levenberg-Marquardt algorithm (LM) [Wikipedia-2] to improve the search for local optima.

All the time, my solution solver got stuck in some local optima. It was like being blind and walking in a rocky terrain, trying to find the lowest point just by touching the ground around you with a stick and guessing whether the terrain went up or down.

My intuition told me that the problem was related to the inability to look further away for low ground at the same time as looking for slopes nearby.

So I started to look for other solutions. I tried random searches and similar methods that use some sort of random trial, but in a problem with very large dimensionality, the random searches tried never found any ‘good’ random values.

I needed some kind of method that had a high probability of searching in places close to ‘old’ good solutions. If I found a local min going downwards I should continue to explore that solution. But at the same time, I needed to look far away in places I had never looked before at some random distance away. Not infinitely far away but at a normal distributed distance away. A normal distance has a mean value but can jump very far away [Wikipedia-3].

Evolution

In my search for a suitable algorithm, I was eventually inspired by the evolution principles in Genetic Algorithms (GA) [Wikipedia-4]. GA are a set of well-established methods used in the Genetic Evolution of Software and have been around since 1980. Initially used in simple tree-based combinations of logical operations, GA have the ability to define a large set of individual solutions. In this case, each solution is represented a local position in the rocky landscape with parameters in a non-linear polynomial equation. By building new solutions in new generations based on crossover and mutations, GA provided an iterative way both to explore local minima and to find new candidates further away. By setting up a normal distributed random function, I got a good balance between looking near and looking far away [Springer19].

Anders Modén is a Swedish inventor working with software development at Saab Dynamics, which is a Swedish defence company building fighter aircraft and defence equipment. In his spare time he likes solving math problems and he plays jazz in a number of bands. He is also a 3D programmer and has written the Gizmo3D game engine (www.gizmosdk.se). He also develops the Cortex SDK in his private company, ToolTech Software.

Cost function (or, how we reward our equation)

In order to solve a non-linear equation, we need to establish a criterion of how large an error a particular solution has. We usually identify this with different metrics like total least-squares using different norms. A simple least-square norm will do in this case [Wikipedia-5]. By looking at the error (cost function) [Wikipedia-6] in a large dimensional space, we get this rocky landscape. The LM method uses the gradient of this landscape to go along the slope to find a better solution. Furthermore, all the methods in back-propagation are also based on this stochastic gradient descent principle. Basically, it’s just a Newton-Raphson solver.

Solution

I wrote an algorithm to solve non-linear optimization problems using GA and I had a lot of fun just looking at the landscape of solutions where you could follow the progress of the algorithm. In the start there was just a large bunch of individual solutions randomly distributed in the numerical landscape.

In a while, some solutions were found that actually were better than the average and other solutions started to gather around these solutions. As the survival of the fittest kicked in, only the best solutions were saved. These clusters of solutions walked down the path to the lowest local point. At the same time, solutions started to be found further away. Sometimes they died out but sometimes they found new local optima. It was like following tribes of native inhabitants that settled, lived and died out.

Eventually the algorithm found better solutions than the LM algorithm. It wasn’t fast but it worked and solved my problem. When the job was done, I also forgot about it…

Almost present time

In 2013 I was impressed by the amazing results of AI and the start of deep learning. The technique behind convolution networks was so simple but very elegant. I felt I had to explore this domain and quite soon I recognized the behavior of the optimization problem. It was the same as my old non-linear optimization problems. I saw that the steepest gradient descent locally was just the same gradient descent as in LM and the rocky landscape was there. Lots of local minima, large plateaus of areas with little descent and the very large dimensionality of the solutions made my decision obvious. I needed to try to solve this by evolution…

I started to design a system that allowed me to try both traditional back-propagation as well as genetic evolution. I also wanted to build a system that allowed not only the optimization parameters to evolve but also the topology of the network. The structure of DNA inspired me as a key to both parameter values and to topology. By having an ordered set of tokens representing both parameters and connections, I could define an entire network by just using different tokens in a specific order.

The birth of Cortex API

I decided to build a software ecosystem both to verify my thoughts and to create a platform for experiments in genetic programming combined with

A simulation is basically a loop running the genetic reproduction laws on a large number of successive generations

neural networks. I named the software ecosystem Cortex, inspired by the neurons in our brain [Modén16a]. The requirements for the system were to be able to use both back-propagation and genetic evolution. I also wanted the system to allow any interconnectivity between neurons as our brain does and not only forward feed as most systems today do. The design should be based on DNA-like genetic elements.

Genetic element classes

In my design of Cortex, there are two classes of genetic elements (the smallest type of building block). The first class is the topology element. It defines the actual algorithm to be used in evaluation as a large connectivity graph. It is typically a chain of low level execution elements which we call the 'topology DNA'. It is described as a fully generic network topology in high-level terms but can also be regarded as a sequence of DNA-based functions from input to output. These elements represent the connections between axons and dendrites in the human brain or in other animals.

The second element class is the activation or parameter class. It's a set of variables that defines the activation and control of the first class network topology. It is also seen as a chain of 'parameter DNA', which is more or less the actual state of the brain. It represents the chemical levels that trigger the different synapses in the brain.

Toplogy DNA

The topology DNA is defined in my Cortex engine as a set of low-level instructions which all execute in a very defined sequence just like a stack-based state machine. The instructions are constructed so they can be randomly generated, communicate through registers and have local memory. The topology DNA is executed by a virtual machine just like a JVM, and a JIT compiler in the backend part can also accelerate it [Modén18].

Parameter DNA

The parameter DNA is a set of register values used by the topology DNA for memory access and parameter control. Different algorithms in a population can have the same topology but individual parameter DNA. Maybe you can regard the topology DNA as the definition of the 'creature' and the parameter DNA is its personal skill or personality.

Genetic laws

The reproduction of a new generation of DNA is based on three basic genetic rules that affect parameter DNA

1. **Crossover**

   Two parents' DNA are combined in a number of ways. Some will be just new random outcomes but some will eventually inherit the good parts from both parents. Crossover can occur in one or multiple split points of the DNA chains. The dimensionality in a correctly constructed crossover is a subspace aligned with the gradients.

2. **Mutation**

   A DNA position can be altered to a completely new DNA value. Typically created by chemical processes or radiation in real life. The dimensionality of this is high unless you limit the number of possible mutations in a DNA.

3. **Biased crossover or breeding**

   This is a new term defined by the Cortex project but it is a very strong function. It uses two parents in a standard crossover but the results are biased towards a parent DNA value but not necessarily the same one. A crossover selects the same values from either parent. A breeding can be a linear combination with both positive and negative factors. An interpretation of the genetic law could be that it represents the 'environmental' effect (education, breeding or life experience) as you grow and it is not used in common GA as they are purely defined by genetic inheritance. This law allows a child to have genomes inherited as a function of parent's genome. The dimensionality of this linear combination is still a subspace with a bounding volume aligned with the gradient as a convex hull which is extremely important.

There is also one genetic rule for topology DNA

1. **Connectivity changes**

   The topology DNA is defined by instructions. The topology can be though of as an infinite number of neurons where all neurons are interconnected but with all weight factors set to zero. The Cortex Assembly Language simulates these neurons and their connectivity. By adding or removing instructions, you can simulate different topologies. Just like the other genetic laws, there are random additions of new routes between neurons. When the weights in the parameter topology are close to zero, the connections are removed.

These 4 rules are used to create new individuals in a new generation. The parameter DNA changes each new generation while the topology connectivity change seldom occur.

Simulation

A simulation is basically a loop running the genetic reproduction laws on a large number of successive generations. You start out with a start population that is a random set of individuals with random-generated DNA. The random values must have a normal distribution.

The number of individuals in the first generation represents the 'survivor of the fittest' and each new generation of new individuals will compete against this set. If their evaluated fitness function scores a value which is better that the worst fitness value in the first generation, the new individual will get into the 'survivors set' and the worst individual will die.

If the simulation is run in multiple instances on a distributed network, they all have different sets but the new candidates will be distributed if they manage to get in on the survivors list and then they will be able to get into other instances' 'survivor sets'. Eventually the sets will converge to a common set of individuals. Notice that this part doesn't need any closed

> This sudden emergence of a new group is really the strength of the evolution principle

loop. Only the best DNA are broadcast and this is a huge benefit compared to back-propagation as you can scale this up without any limit.

Fitness function

In order to successfully rank the solutions in each generation, you need a global fitness function that can be evaluated for each individual. One cost isn't enough as you are in this large multidimensional-parameter landscape. Instead you need a distribution of costs that can be compared. If your input/output contains a large set of input values and expected outputs and they follow the same statistical distribution, you can evaluate a statistical cost function using a large number of representative samples from a subset. The subset will follow the same distribution and can therefore be compared using different metrics like Frobenius norm, L2 norm or cost functions like SoftMax in back-propagation.

The Cortex engine combines the same cost functions used in back-propagation with the sampled cost function for the GA, which makes them use the same input/output and the same distribution of data.

Flow

In the beginning of a simulation, you may notice a high degree of chaos. Not that many individuals score really well, but after a while you will see a number of groups emerge. They represent a local max on the high-degree function surface. The mathematical solution is a very high-degree nonlinear function with multiple local min/max. In a traditional solver, you interpolate the gradients in the function to step towards the best local max value and this is seen as a successive improvement of the fittest in each local group generation by generation. BUT you will also eventually see new groups started far away from existing groups within completely new local min/max. A traditional LM solver will then not be able to find in its gradient search state, and many times this is also true for a deep neural network in its back-propagation when the problem is very flat with lots of local min/max, since this is a problem for the stochastic gradient search too.

This sudden emergence of a new group is really the strength of the evolution principle. A new 'feature' can be so dominant that new individuals now replace all offspring that were previously part of the 'survivor' groups. Some old strong genomes are often preserved for a number of generations but if they don't lead to new strong individuals, they will be drained and disappear.

By having larger populations, the old strong genomes will survive longer and there is a higher probability that an old strong genome will combine with a future very new strong genome.

Building DNA

Let's start with building a fixed topology network and select a very basic model.

To compare with other existing neural network APIs, we select a basic [input-4-4-output] network (see Figure 1, overleaf).

The code to realize this network in Cortex is shown in Listing 1.

The cortex layers are now defined and the connections (synapses) are connected using the forward feed pattern. All neurons between each layer are fully connected to each other.

Generating the code

We will now tell the network to generate an assembly a like program from the network that can be used to evaluate the network. First, we compile it…

```
// Compile it and clear all internal data,
// possibly optimize it
// cortex->compile(TRUE,optimize);
```

And then we set some random values in the DNA parameters…

```
// Provide some random start values
cortex->getContext()->
  randomParameters(1.0f/(neurons));
```

We now have generated an assembly-like program that in the future could possibly be run by a dedicated HW. We can take a look at the 'disassembler' of this program to understand the content.

The disassembly looks like Listing 2. The first section contains information about instructions and parameters. There are 33 value states. One for each node and one for each synapse. There is one input register and 33 parameter registers. These correlate to the node's different internal attributes. Compare with bias and weights in a normal network.

```
// Create a brain
ctxCortexPtr cortex=new ctxCortex;

// define input output as derived classes of
// my input and output
ctxCortexInput *input=new MyCortexInput();
ctxCortexOutput *output=new MyCortexOutput();

// Create an output layer that has the right size
// of output
output->addLayerNeurons(output->getResultSize());

// Add input/output to brain
cortex->addInput(input);

// Add input to brain
cortex->addOutput(output);

// create internal layers. two layers with
// 4 neurons in each
cortex->addNeuronLayers(2,4);

// Connect all neurons feed forward
cortex->connectNeurons
  (CTX_NETWORK_TYPE_FEED_FORWARD);
```

Listing 1

All mutations and crossovers and breeding are performed on the parameter DNA

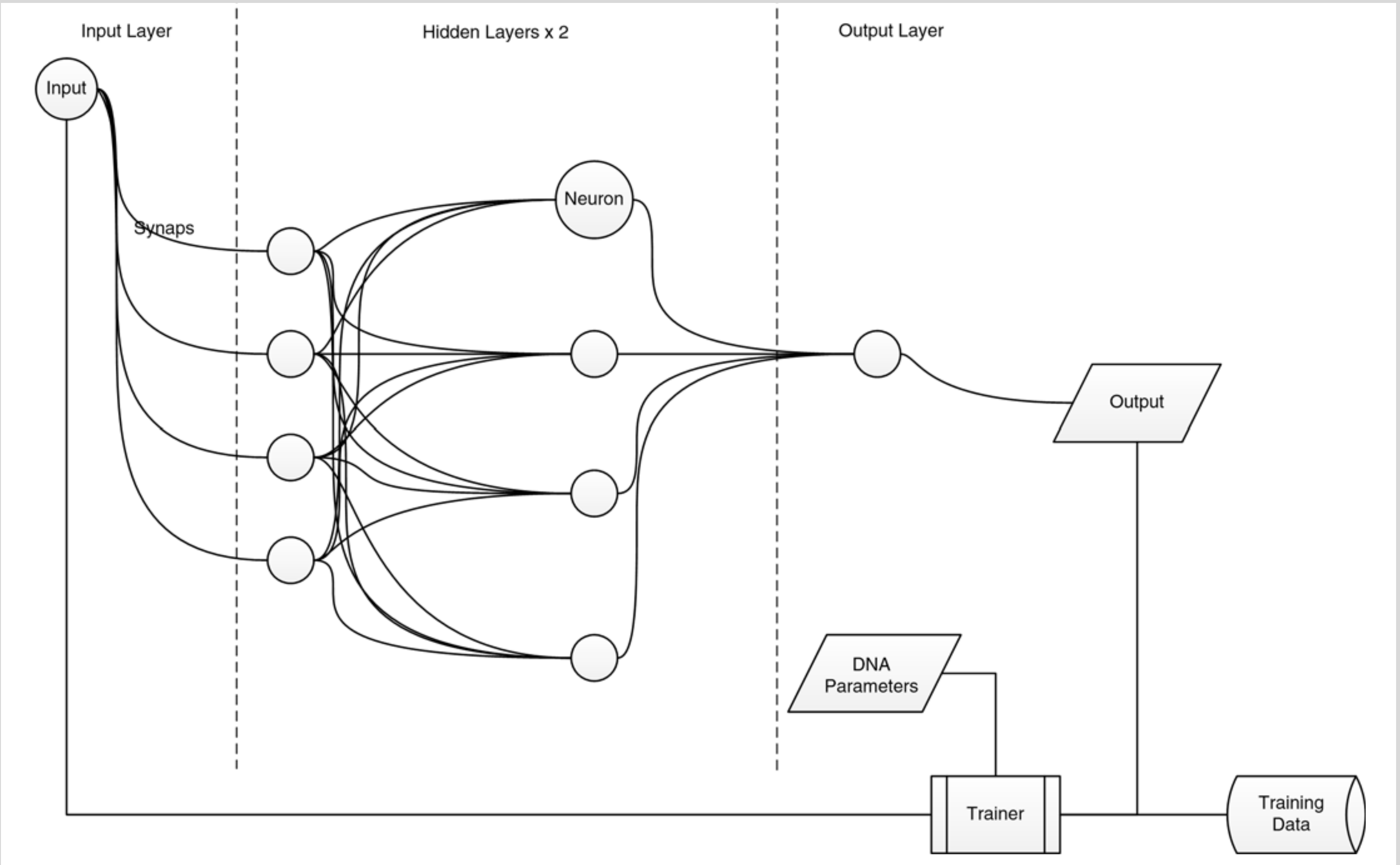


Figure 1

As you see above, the topology of synapses and neurons is translated into reading values from specific registers, doing some functions on the values and storing the value result in a new register.

Now this is an important statement. If we had an infinitely large network where all nodes were interconnected, we could model all networks just by the parameter values and set all weights to zero where we don't have any synapses. This would result in an infinitely huge parameter DNA with just some sparse values and the rest being zeros. Instead, we divide the DNA into parameter DNA that has non-zero values most of the time and use a topology based on random interconnections modelled by the CAL instructions.

These two DNA parts are then kind-of exchangeable, where you can look at the parameters only or parameters and instructions together. This forms an important criterion in the evolving of the network. All mutations and crossovers and breeding are performed on the parameter DNA. During a simulation, the parameter DNA are evaluated and updated a lot. Then, every now and then, a network update occurs in an individual. A synapse is added with weight 0. This gives the same result as all the other cost-

```
-- Compiled Cortex Program --
Instructions:311
ValueStates:33
Parameters:33
LatencyStates:0
SignalSources:1
InputSignals:1

------------ SubRoutines ---------------
 Threads:1
----------------------------------------

---- Thread:0 ----
 Offset:0
 Length:310
 Pass:0
```

Listing 2

...a parameter that represents a weight that stays at zero for several generations could be exchanged for a removed synapse...

```
---- Commands ----
RCL Input (0)       { Recall input register SP: +1 }
MULT Param (0)      { multiply with param register }
TEST Drop Value (0) { Test drop value register }
STO Value (0)       { Store value register SP: -1 }
RCL Value (0)       { Recall value register SP: +1 }
ADD Param (1)       { add param register }
NFU (7)             { Neuron function }
STO Value (1)       { Store value register SP: -1 }
RCL Value (1)       { Recall value register SP: +1 }
MULT Param (2)      { multiply with param register }
TEST Drop Value (2) { Test drop value register }
STO Value (2)       { Store value register SP: -1 }
RCL Value (2)       { Recall value register SP: +1 }
RCL Input (0)       { Recall input register SP: +1 }
MULT Param (3)      { multiply with param register }
TEST Drop Value (3) { Test drop value register }
STO Value (3)       { Store value register SP: -1 }
RCL Value (3)       { Recall value register SP: +1 }
ADD Param (4)       { add param register }
NFU (7)             { Neuron function }
STO Value (4)       { Store value register SP: -1 }
RCL Value (4)       { Recall value register SP: +1 }
MULT Param (5)      { multiply with param register }
TEST Drop Value (5) { Test drop value register }
.
. (CUT, see [Modén16b])
.
ADD Param (30)      { add param register }
NFU (7)             { Neuron function }
STO Value (30)      { Store value register SP: -1 }
RCL Value (30)      { Recall value register SP: +1 }
MULT Param (31)     { multiply with param register }
TEST Drop Value (31){ Test drop value register }
STO Value (31)      { Store value register SP: -1 }
CL Value (31)       { Recall value register SP: +1 }
SUM Stack (4)       { Sum stack values SP: -3 }
ADD Param (32)      { add param register }
NFU (7)             { Neuron function }
STO Value (32)      { Store value register SP: -1 }
RCL Value (32)      { Recall value register SP: +1 }
DROP                { drop stack value SP: -1 }
RCL Value (32)      { Recall value register SP: +1 }
DROP                { drop stack value SP: -1 }
---------------------------------------
 Passes:1
 MinLen:310
 MaxLen:310
 TotLen:310
 AvgLen:310
 Pass:0  Threads:1  Len:310  AvgLen:310
```

Listing 2 (cont'd)

function evaluations but now we have a new parameter to play with. In the same fashion, a parameter that represents a weight that stays at zero for several generations could be exchanged for a removed synapse but in my simulations, I have chosen to let them stay so my network keeps growing but with more zeros.

Duality between GA and Neural Network Gradient Descent

As stated before, the strength of GA is in finding a new set of survivors far away from the current solutions. This is hard using a traditional LM but the LM is very good at incrementing the last steps to the best local solution; that is also true for the back-propagation method for Neural Networks using stochastic gradient descent. The GA takes a long time to find the 'optimal local min/max'. It will find new solutions but they rely on random changes and not as targeted as the LM is.

So the duality exists between them. The GA will find new start points and the back-propagation of Neural Networks will find local min/max points. Cortex implements a hybrid mechanism that jumps between the two search modes.

Duality example

To exemplify the strength of the duality between Genetic Algorithms (GA) and Neural Network (NN) back-propagation a simple example is used [Modén16c].

In this example we use a simple target function like sin(*x*). We want the neural network when given an input [0,2PI] to generate a sin(*x*) curve. A sin(*x*) curve is easy for us humans to recognize and we know that it can be described as a Taylor series of higher dimensions

The network knows nothing about the actual function but only the target value output for each *x*. We can choose any other function or non-continuous transfer function. It doesn't matter. We are just interested in how a neuron network will simulate this and that it should be easy for us to look at it and understand it.

We start with a simple neural network with 20 neurons in each layer and just one mid layer. Listing 3 (overleaf) is the code to set up our start topology. We do no topology evolution in this example.

The input layer is a single neuron and the output is also a single neuron. We use the ELU (exponential linear unit) activation function and we train in batches of 10 samples in each batch. In total, we sample the sin function with 1000 steps, which gives us 100 iterations then for each epoch.

We initialize the neurons with zero mean and standard deviation of 1/20 (the number of neurons in each layer) and use a back-propagation step of 0.1. This is the result of the back-propagation…

Let me explain the charts (see Figure 2, overleaf)…

The most important chart is the 'Error Magnitude' chart (top right). It shows the error of the cost function for each iteration. This function should decrease as an indication of that we are learning. In this case we use a magnitude cost function L2 norm so it basically the sum of all squared

The strength of GA is in finding a new set of survivors far away from the current solutions

```
// Create a brain
ctxCortexPtr cortex=new ctxCortex;

// define input output as derived classes of
// my input and output
ctxCortexInput *input=new MyCortexInput();
ctxCortexOutput *output=new MyCortexOutput();

// Create an output layer that has the right size
// of output
output->addLayerNeurons(output->getResultSize());

// Add input/output to brain
cortex->addInput(input);

// Add input to brain
cortex->addOutput(output);

// create internal layers. one layer with
// 20 neurons in each
cortex->addNeuronLayers(1,20);
```

Listing 3

errors in the estimated output function(x) compared to sin(x). When the cost function is zero, we have the correct output.

The next important chart is the 'delta-err' chart (bottom centre). It shows the error for each sample as a function of (x) compared to the sin(x) target function. Optimally this will be zero for all values when the trained function is near or equal the target function sin(x). As we define the function with 1000 values for x, we want the graph to show delta error = 0 for each x.

And the 'value' chart (bottom right) actually shows the estimated output function, which should be sin(x) for each x. Right now we can see that the output is starting to get the shape of a straight line with some bends in the ends. As we just defined it using one layer, we have typically a network of parallel neurons that handles local segments of the transfer function.

The 'delta' chart (bottom left) shows the error propagated through the network for each iteration to the last node, which is the input node. Look at the delta values in back-propagation [Wikipedia-7]. One delta for each connection but only the 5 first deltas are drawn. In a shallow network, it is quite easy to propagate delta to the last node so the magnitude are 'rather' high in this case. That means that the last layer is updated with 'training' information. If a network starts to learn, the deltas will increase.

After a small number of iterations (20–100) we can see that the target function is looking like a sin(x) curve. (See Figure 3, opposite.)

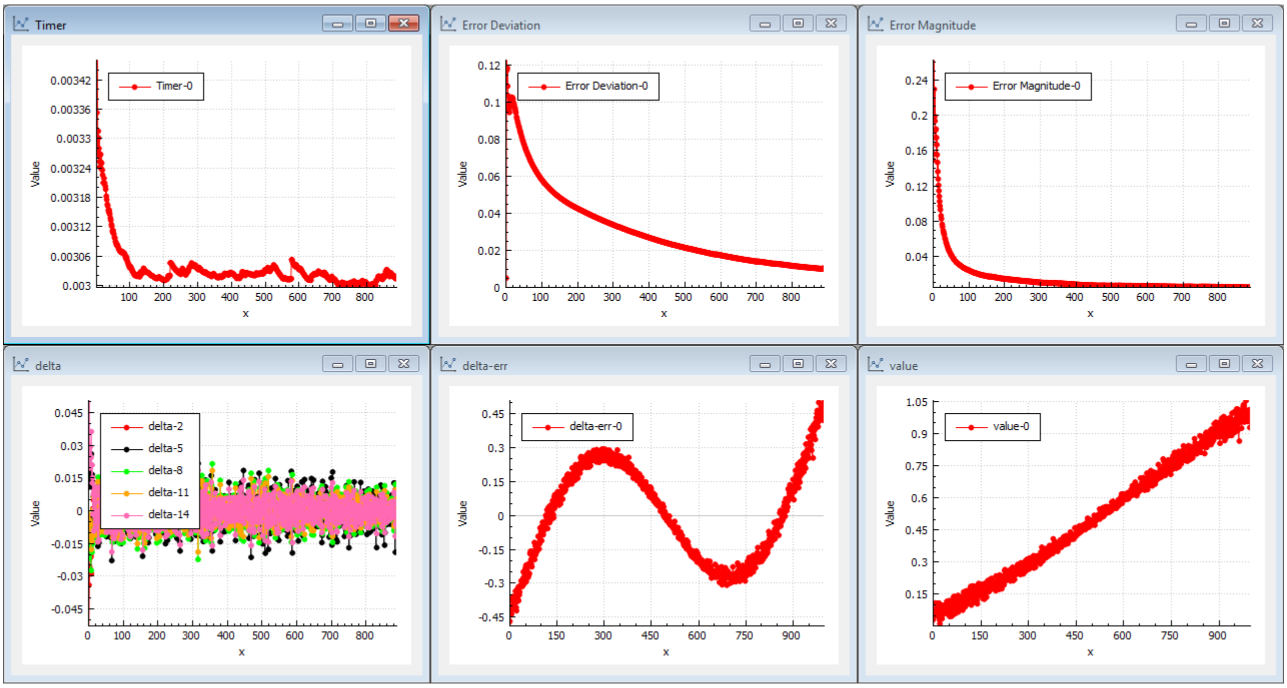


Figure 2

In **genetic training**, the probabilities of finding solutions are **not randomly equally distributed**.

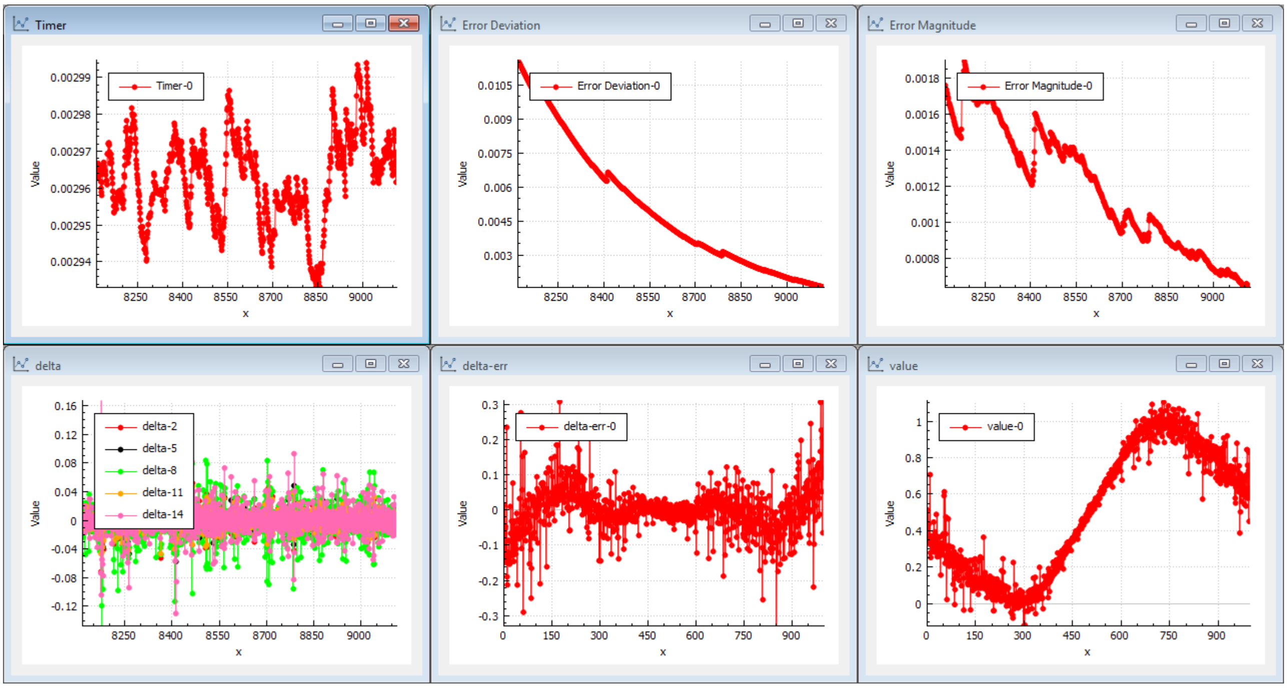


Figure 3

The delta error of the output is rippling around 0 but with pretty large values (1/100–1/10). We can draw the conclusion the network is learning. The cost error magnitude is continuously decreasing. As we have 20 neurons in parallel in the mid layer, we can continue to iterate to get better precision with smaller back-propagation steps. To make the neural network more sensitive to deep layers (making it less likely to get stuck in local optima), we just use a simple momentum as the gradient function. RMS or Adam would improve performance.

This first step shows that the back-propagation mechanism is very fast at finding solutions in the NN that converge to a sin function. This is done in seconds. The conclusion is that this feature is trained very easily in a shallow network using simple batch gradient descent.

But let's see what happens in a deep network – or, at least, a deeper network – when we increase the layers and neurons. We now use 6 layers instead.

After a large number of iterations, the output is still just random (see Figure 4). The output doesn't resemble the sin(x) target function. The delta error shows that the output is pretty much a noisy dc level, which makes the error a noisy sin(x). The deltas propagated through the network are now very low because each hidden layer kind-of reduces the energy in the back-propagation. The cost magnitude isn't really decreasing and the system isn't learning much at the beginning.

After a while, the training finally is able to kick some data through the network. Figure 5 shows the information after 14700 iterations.

The error magnitudes start to drop and the deltas propagated through the network start to increase. If we had selected a ReLU or a tanh activation function, it would have taken even longer to reach this state.

A deep network has a larger number of parameters and a larger 'equation' with many more minimums and saddle points, and therefore the gradient search mechanism – even if it is boosted with speed and momentum and other smart features – will take longer to iterate to a proper solution. Gradient selection methods like Adam and RMSProp are really good but they still struggle in deeper networks.

Now let's introduce genetic training. In genetic training, the probabilities of finding solutions are not randomly equally distributed. There is a higher probability of finding solutions where the previous generations succeeded, which is the fundamental property of evolution. 'The survival of the fittest'.

Figure 6 on page 22 shows what genetic training can look like.

Genetic training uses evolution to find new candidates in the population. In the beginning, the first populations just contain garbage but very quickly (in this case in seconds) even if there are 6 layers and 20 neurons in each layer, the genetic algorithm finds a bit better candidate. The genetic algorithm doesn't care about delta levels, so the innermost layers can instantaneously be updated by the genetic evolution.

The solutions found first have a value curve far from a sine and the delta-err is still a sine. But pretty quickly, the value curve starts showing sine-like shapes.

The genetic algorithm doesn't care about delta levels, so the innermost layers can instantaneously be updated by the genetic evolution.

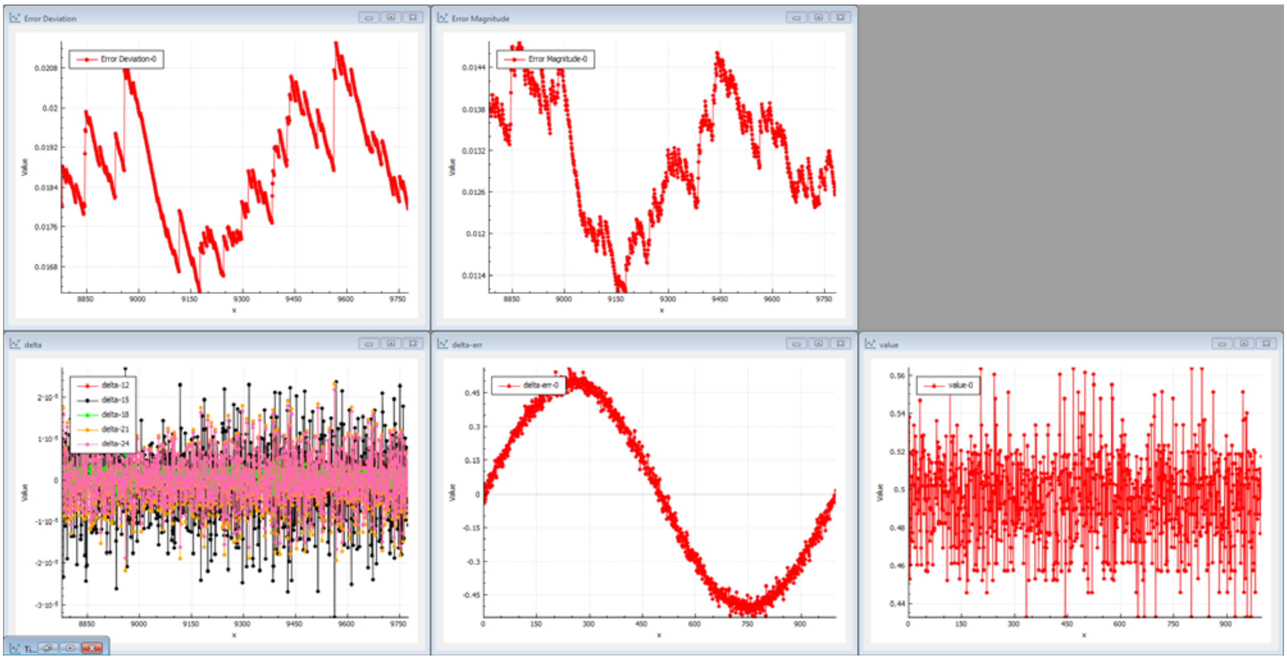


Figure 4

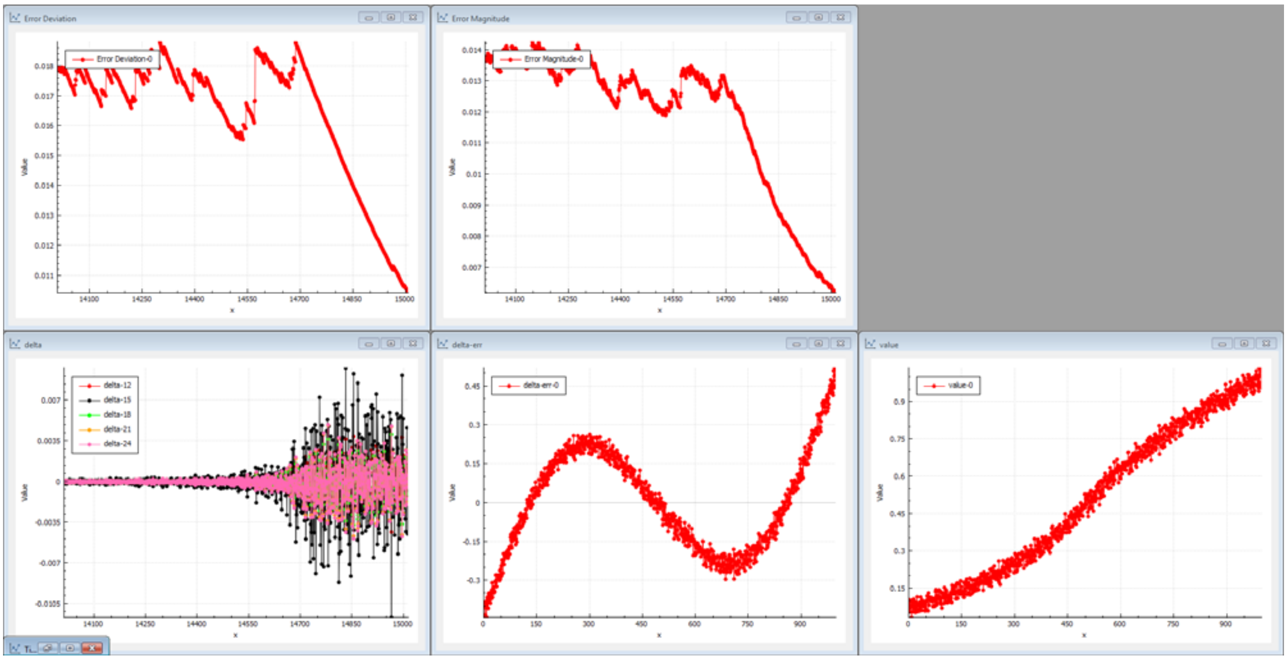


Figure 5

The solutions found first have a value curve far from a sine and the delta-err is still a sine

In this case (Figure 7) the genetic solver actually finds a better solution in a shorter time than the back-propagation mechanism.

The genetic solution will be a good one but perhaps not the very best we can find. It finds ‘global’ good solutions very quickly but the fine-grained tuned solutions will take a longer time to find (see Figure 8 on next page).

But the NN back-propagation, as noted before, was very efficient to iterate when a good solution was found. Can we use this to improve our genetic result?

We put the result from the genetic evolution into the NN back-propagation and we get Figure 9 (on the next page).

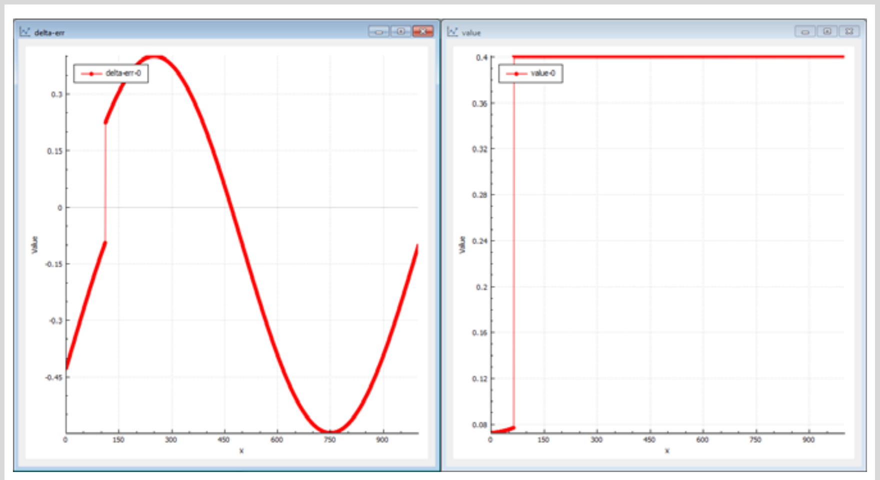


Figure 6

Instantly we can see that the NN back-propagation picks up a very good solution from the beginning. In this case we find a better solution than we previously found using only traditional back-propagation. The deltas are large from the start and the magnitude of error drops immediately. This shows that the solution found by the genetic evolution was a very good candidate and that the NN back-propagation is capable of iterating this solution to a better solution immediately.

This clearly shows the duality in one direction. The other direction is more trivial. You can feed better solutions into the GA using results from the NN and therefore improve the fittest solutions.

This result then defined the duality (state machine) between NN and GA.

I do believe the strong mechanisms of GA and the capability to run almost infinitely large parallel simulation either in the cloud or in Quantum Computers in the future will evolve the techniques of using GA.

Thanx for reading■

Further reading

- Genetic Programming and Evolvable Machines, ISSN: 1389-2576 https://link.springer.com/journal/10710
- Link to Sebastian Ruder’s excellent pages on various gradient optimization methods http://ruder.io/optimizing-gradient-descent/
- Link to article about CAL. The Cortex Assembler Language: https://www.linkedin.com/pulse/cortex-assembler-language-20-anders-modén/
- Article about Sussex GA Robotic SAGA framework http://users.sussex.ac.uk/~inmanh/MonteVerita.pdf
- Full source for CAL execution example http://tooltech-software.com/CorTeX/execution_example.pdf
- The Cortex Engine https://www.linkedin.com/pulse/cortex-genetic-engine-anders-modén/
- Deep Genetic Training www.tooltech-software.com/CorTeX/Deep_Genetic_Training.pdf

References

[Modén16a] Modén, A. (2016) CorTeX Genetic Engine, *LinkedIn*, 14 April 2016: https://www.linkedin.com/pulse/cortex-genetic-engine-anders-modén/

[Modén16b] Modén, A. (2016) CAL Execution Example, *ToolTech Software*, 1 June 2016: http://tooltech-software.com/CorTeX/execution_example.pdf

[Modén16c] Modén, A. (2016) Deep Genetic Training, *ToolTech Software*, 1 June 2016: www.tooltech-software.com/CorTeX/Deep_Genetic_Training.pdf

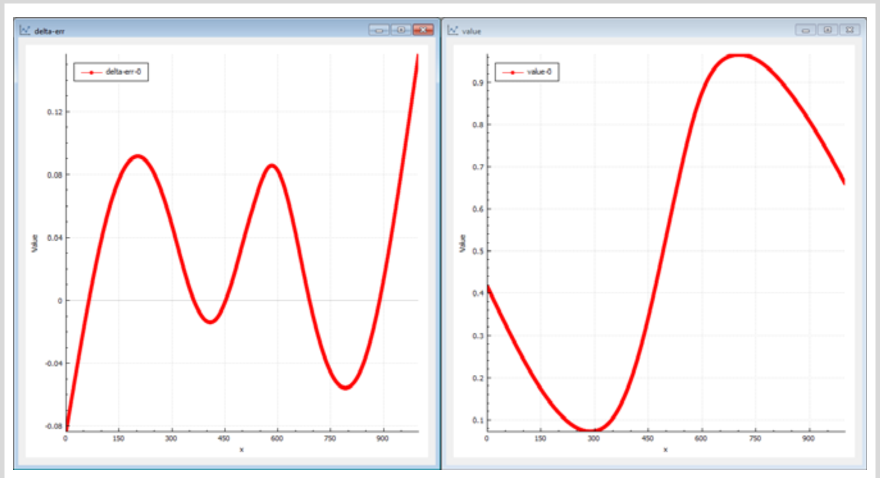


Figure 7

> **The strong mechanisms of GA and the capability to run almost infinitely large parallel simulation ... will evolve the techniques of using GA**

[Modén18] Modén, A. (2018) Cortex Assembler Language 2.0, *LinkedIn*, 10 July 2018: https://www.linkedin.com/pulse/cortex-assembler-language-20-anders-modén/

[Springer19] (2019) ‘Genetic Programming and Evolvable Machines’ on Springer Link: https://link.springer.com/journal/10710

[Wikipedia-1] Gauss-Newton algorithm: https://en.wikipedia.org/w/index.php?title=Gauss%E2%80%93Newton_algorithm&oldid=886266631

[Wikipedia-2] Levenberg-Marquardt algorithm: https://en.wikipedia.org/w/index.php?title=Levenberg%E2%80%93Marquardt_algorithm&oldid=888015378

[Wikipedia-3] Normal distribution: https://en.wikipedia.org/w/index.php?title=Normal_distribution&oldid=887884220

[Wikipedia-4] Genetic algorithm: https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=887643125

[Wikipedia-5] Lp space: https://en.wikipedia.org/w/index.php?title=Lp_space&oldid=887448344

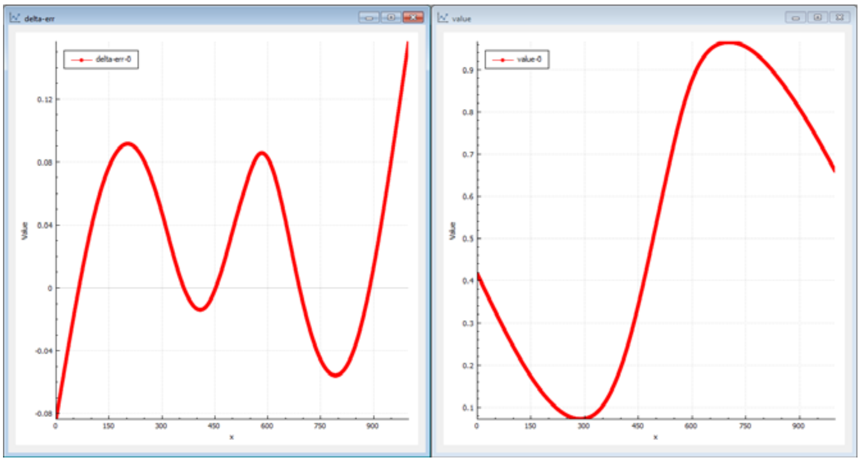


Figure 8

[Wikipedia-6] Artificial neural network: https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=887630632

[Wikipedia-7] Backpropagation: https://en.wikipedia.org/w/index.php?title=Backpropagation&oldid=886645221

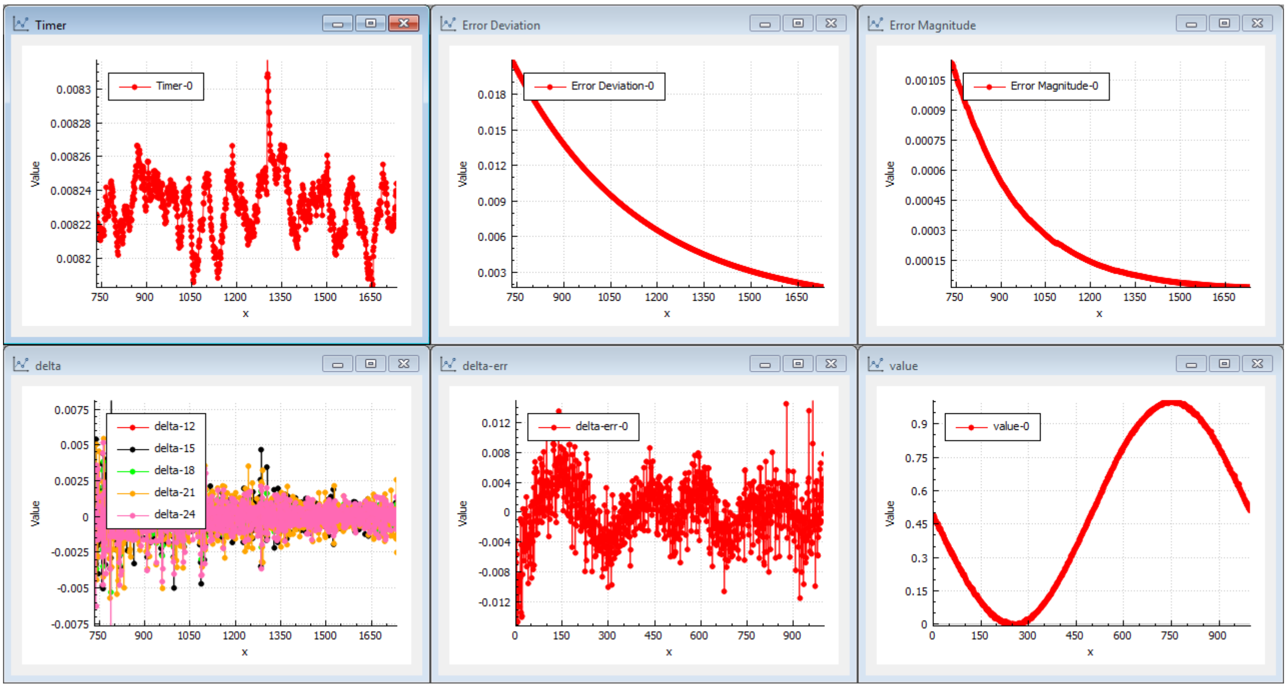


Figure 9

Blockchain-Structured Programming

Coins are a wedge-shaped piece used for some purpose. Teedy Deigh shares her angle on the technology underpinning cryptocurrencies.

Whenever a new programming paradigm is hailed it is more often of a blend of existing paradigms than being genuinely novel. For example, what do you get when cross reactive programming (Rx) with functional programming? Functional reactive programming (FRP), which is essentially a prescription without side effects.

Or acclaimed new paradigms are simply existing paradigms with a twist. For example, what do you get when you take procedural programming and eliminate side effects? Functional programming.

Sometimes such a twist may be a cynical corruption. For example, what do you get when take a pragmatic approach to functional programming, such as allowing some side effects? Pragmatic functional programming (a.k.a. procedural programming).

And so it is that cynicism and corruption bring us to the new kid on the block. What do you get when you cross the elegant and reasoned hierarchies of block-structured programming with the clarity, fairness and accountability of unregulated capitalism? Blockchain-structured programming.

In one sense, this blended paradigm represents a new mix, yet in another it can claim a long history. For example, pyramid schemes date back to ancient Egyptian times. The privateering (a.k.a. piracy, theft, government-backed enterprise) that kicked off Europe's imperial age popularised pieces of eight as a currency. In modern computing terms, the division of a byte into pieces of eight results in bits, hence Bitcoin. Another cryptocurrency, Ether, is named for the insubstantial and fictitious medium that, in the 19th century, was believed to propagate light.

Naming offers many insights: Litecoin is named for its lack of financial weight; PotCoin is dope; and Namecoin has yet to be named – as developers know, naming is hard. As for *blockchain* itself, although the term clearly describes a list structure, it also seems to double up as the seed word in a word association game, to which the most obvious response would be *flushplunger*.

In traditional computer science, algorithms and data structures are described in terms of performance costs. Insertion into a conventional linked list has a cost of O(1), for instance. A similar theme underpins blockchains, but the measures are more generous. Insert a record into a list using a cryptographic hash and you can be talking a cost of 1 MWh. Indeed, the costs are so severe that instead of using IDEs (Integrated Development Environments), blockchain-structured programmers favour DIED (Doing Insertions with Environmental Destruction).

The energy consumption of Bitcoin, for example, is on a par with that of a nation like Switzerland. But unlike Switzerland, which uses its utilities to support a well-regulated banking system, CERN and chocolate, Bitcoin has so far eluded utility. Often touted as a currency, it behaves more like a speculative asset – mostly, people speculating as to whether or not it's actually an asset. The specific mechanism in Bitcoin that has proven so costly is its proof-of-work algorithm (POW, see also *Prisoner of War*). Proving that Bitcoin works has taken more time and energy than expected – unlike rigorous, reasoned and environmentally neutral high-energy physics experiments, it's not going well.

Data structures in the blockchain world have many superficial similarities to more conventional data structures. For example, instead of lists having a *head*, they have a *blockhead*. The same word substitution can also apply to those who head blockchain companies. Classic singly linked lists, such as found in Lisp, are often created using **`cons`**. This is the same in blockchain-structured programming.

There appear to be many operators in blockchain-structured programming, but many seem to have dubious reputations – cryptocurrencies have proven surprisingly popular in the kinds of international trade frowned upon by less entrepreneurial individuals hindered by decency and morals.

Perhaps one of the most popular operators is the **`exchange`** operator. Where **`exchange`** and **`swap`** are similar in conventional programming, in blockchain-structured programming an exchange involves gambling with speculative assets cryptographically, whereas the swaps market involves gambling with speculative assets without the assistance of cryptography. Exchange operators are therefore defined on Bitcoin-set as opposed to bit-set structures. These are often password-protected by a charismatic founder (or *head*, see above) who may drop dead at any moment – cryptonite that adds a certain frisson to chaining one's finances to blocks. In such cases the underlying architecture of a distributed, tolerant and anonymous ledger model is considered ironic.

As with any programming paradigm in the 21st century, blockchain-structured programming addresses distribution and concurrency. Blockchains are inherently distributed and their transactions make much more sense if you drop ACID. Concurrency is typically addressed through smart contracts. As with most other things labelled *smart*, they're not, and still accommodate many classes of error that programmers have come to expect – even demand! – from their concurrency models, such as race conditions and re-entrancy problems. Being able to have a race fits well with the competitive ethos of blockchain-structured programming.

We can conclude that underpinning many cryptocurrencies is a simple design pattern: the Blockchain of Irresponsibility. Blockchain-structured programmers are quick to counter that the inefficiencies and lack of accountability inherent in many cryptocurrencies and blockchain models should not be used to judge all blockchain applications. They contend that blockchain is a good solution, albeit one in need of a good problem. ■

Teedy Deigh Devoted as she is to her Fiat Uno, Teedy Deigh is not as much a fan of fiat currency, preferring instead to keep her investments spread across Amazon vouchers, Linden dollars, the gold of Azeroth and other ersatz mattresses.

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

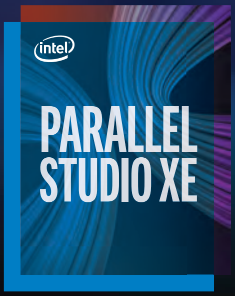
You can now purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Design: Pete Goodliffe

CODE
MAXIMIZED


intel
PARALLEL
STUDIO XE