human readable magazine

SEPTEMBER 2019

000

Jonathan Boccara · Rob Landlay · Andy Kitchen · Omar Shehata · Michael Kohl · Michele Caini

How to Avoid Template Type Deduction in C++

Template type deduction is an awesome feature of C++. Except when it gets in your way.

An explanation of counterfactual fairness

A new notion of fairness for ML classifiers proposed by researchers at DeepMind.

The Plight of Open Source

How a one-line contribution took 3 months to get released (and still broke everything anyway)

Contributors

Editor in Chief

PANAGIOTIS PEIKIDIS
Mostly known as Pek, he has been programming since he was in high school. After a series of terrible programming language choices he decided to start a magazine about programming in the hopes that he'll finally make the right choice.

Authors

ANDY KITCHEN
Andy is a startup CTO, loves all things AI, Machine Learning and Computer Science. Never wants to stop learning and building things.

ROB LANDLEY
He's currently working to turn Android into a development environment capable of building itself under itself. Video of him throwing liquid nitrogen into a swimming pool has been viewed on Youtube 11 million times.

JONATHAN BOCCARA
Jonathan Boccara is a C++ software engineering lead, blogger and writer focusing on how to make code expressive. His blog is Fluent C++.

MICHAEL KOHL
Michael's love affair with Ruby started around 2003. He also enjoys writing and speaking about the language and co-organizes Bangkok.rb and RubyConf Thailand.

MICHELE CAINI
Michele is fond of two things: C++ and gaming. When he isn't spending his time attending conferences, he blogs about coding and works on his popular open source game engine EnTT.

OMAR SHEHATA
Omar is a graphics programmer at Cesium. In a past life, he made flash games on Newgrounds.

PHIL PEARL
After a successful career in the oil-field service business all over the World in field and management positions, I transitioned to computer software. I'm currently working on finshing my second children's book.

EZE ONUKWUBE
Eze is a writer and software engineer who is curious about people, processes and the stubborn charm of life. He is currently cultivating an obsessive interest in learning, algorithms, and intelligence – both artificial and genuine.

Illustrators

LAURA SUMMERS
Laura is a multi-disciplinary designer working as a startup consultant and strategist, and the human behind fairxiv.org. She speaks, writes and runs workshops at the intersection of design and technology.

ZARA MAGUMYAN
Zara is an illustrator and graphic designer based in Yerevan, Armenia. Currently she specializes in illustration, user interface development, and digital products.

FAUZY LUKMAN
Fauzy Lukman is a multidisciplinary designer based in Indonesia who work on graphic, branding, illustration and also UI for Web & Mobile app.

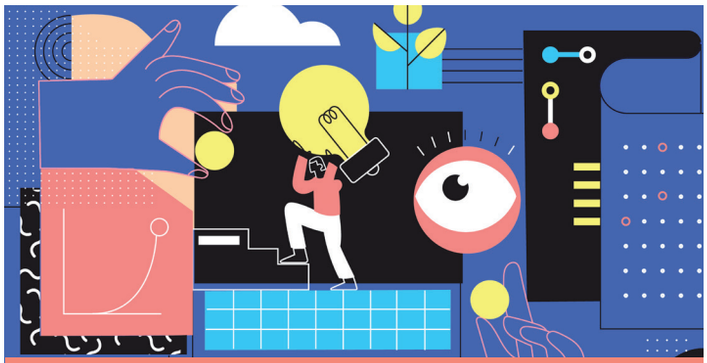
LEANDRO LASSMAR
Leandro Lassmar is an illustrator living in Minas Gerais, Brazil. He worked in animation studios, currently works for magazines, books and advertising.

SERGEY KONOTOPCEV Illustrator, **ADAOBI OBI TULTON** Copyeditor, **YANNIS SPANOUDIS** Magazine Designer

Contents

Contact

WEB www.humanreadablemag.com
EMAIL hello@humanreadablemag.com



EDITORIAL

But why?

By **Panagiotis Peikidis**
Illustration by **Zara Magumyan**

- *A magazine?*
- *Yes, a magazine.*
- *But.. why?*
- .....

You probably don't know this about me, but I have a history of working in industries that have a high risk of failure. My first job as a professional software engineer was in video game development. After seven years of crunch, I took a break and took over a failing restaurant business... in New York. And now, after three years of seeing my wallet fluctuate between bursting from too much cash to bursting with moths, I want to start a magazine.

Putting aside my oblivious career choices that all but guarantee I will never have a steady sleep schedule, at first glance, you would think the common thread here is risky industries. But I challenge you to read between the lines. There's a good reason why I made those choices.

No, not that!

In all three of those cases, there was something missing from my life, and I wanted to be the one to create it. I don't think I need to explain the video game industry, but if you play video games and do programming, chances are you have at least 20 side projects that never went past a working prototype. Basically, I wanted a video game I'd like to play. For the restaurant business, I wanted a cafe that I'd want to hang out in, just like all the cafes I used to hang out in as a university student back in Greece. And so, for the same reason, I wanted a magazine that I'd like to read.

I may not be a 10x engineer, far from it, but I have been geeking out about programming ever since I was 14, way before I owned a computer. I went from learning GWBasic in my local PC shop, to building terribly written PHP websites that only I had access to, to publishing my first open-source software in Visual Basic 6, to trying to wrap my head around OOP in Java, to professionally writing Unity games in C#, to building a tool for my newsletter in Ruby on Rails, to now learning about SPA in Vue.js. And those are just the highlights. There's also Prolog, Pascal, C++, Processing, Flutter, ActionScript, mIRC scripting language, Lua, Python, and the list goes on. Point being, I like learning about new technologies. Trying to always pick the right tool for the job and not shoehorning the one thing I know into everything I want to create is part of the fun for me.

And to do this, I read a lot. Not necessarily books, which I have a tiny collection of, but online resources like blogs, news sites, tutorials, and so on. I slowly but surely built an RSS library of about a thousand sources over the last few years. But there's one thing that has been bothering me for years: consistency.

It is difficult to find a single source with the types of articles you personally like. You can find many authors, but they are scattered throughout the web. Thankfully, RSS solves many of these problems, but not entirely. Some authors don't even support RSS, some only include an excerpt, et cetera, et cetera. Basically, what I've come to realize is that in a day and age where there's an infinite amount of content to shift through, you need to find a publication whose editorial strategy aligns with what you are looking for.

And since I didn't find that, I decided to do what most people without a budget or even an idea of how to run such a thing do: create it myself.

This magazine, the Human Readable Magazine, is what I believe should exist out there. It's entirely based on what I like to read, and my hope is that it will resonate with you.

Just like I learn, the magazine doesn't focus on any one particular language or domain. There are articles about C++, JavaScript, and Ruby; and about security, monads, and neural networks. Just like I prefer, the articles are designed to be read at your leisure—no need to follow along in a tutorial. You can read it at home, at work, on your commute—heck, you can even read it in the bathroom. And finally, just like I love, the articles are deeply technical. Except for the occasional column and humor piece, the main body of the magazine is about technical and actionable knowledge. Educating is one of my principles as an editor.

But don't just take my word for it. This issue, which I spent the last few months putting together with authors and illustrators I have been following, is here to showcase what to expect from our magazine now and in the future.

But before I leave you, I must thank the people that gave me the push to actually start this project: our 6,000+ newsletter subscribers and especially our Patreon backers, who have been supporting my work for over a year now. You see, this idea has been floating around in my head for over three years. My first attempt to start a magazine was cut short when one of the authors I contacted asked a very apt question: Who the hell are you and where is your audience? I may be paraphrasing here, but you get the point. And fair enough, I had no audience. But how do you get an audience if you have no content?

"Why don't you start a newsletter?" suggested another author.

"A newsletter?" I thought to myself, baffled. "Do people even read newsletters?" I asked myself in a patronizing way. And to my surprise, I replied back, "Why not?"

And so Morning Cup of Coding was born a little over a year ago. The idea was simple: collect the types of articles you'd like the magazine to have, add a summary to explain why each article is worth somebody's time, and then wait and see if there are people out there looking for the same.

And wait I did. Turns out, people are interested in these articles.

So now we're back to where I began, this time with an audience and still no idea of how to run this thing. But I'll be damned if I didn't try publishing it this time. ✦ hello@humanreadablemag.com

human readable MAGAZINE

CAUSERIE

How to Bake a Deep Network

By **Andy Kitchen**
Illustration by **Fauzy Lukman**

Like many people from my generation, I remember returning to our small apartment to my mother slaving away over a hot GPU. If you're like me, you probably get nostalgic over the smell of hot PCB and the ping of notifications as Mum excitedly chatted to her friends over IRC.

I'm going to teach you a great recipe for a deep network, just like my mother used to make. You'll be a hit in user groups, at science fairs, and at potlucks if you whip out one of these classics.

A lot of people these days get their models premade from cloud or SaaS and that's OK (hey, we all get busy sometimes), but you can make a classic model at home, and isn't that more satisfying?
You will need:

- 50,000 data examples (with labels)
- 1 GPU, ~2 Terra-flops
- 1 deep learning framework

The key to a great neural network is good ingredients. That's why I always use high-quality, hand-labeled data. Call me a traditionalist, but as my mother would say: "A great model needs great inputs! Simple as that!"

Start by splitting your data into validation, training, and test. A lot of people ask me how to split it up. Traditionally it depends on the school and region you're from, but go with a 60/20/20 split and you should be fine!

Now normalize your data. This is a small step but a lot of people forget this nowadays. Just like salting an eggplant, you want to get that nasty, high-variance bitterness out and ensure a nice, smooth zero mean. Mm, mm, that's deep learning!

Layers, oh boy. Back when I was in university there was one kind of layer we all used and it was fine. Nowadays, kids have their Inceptions, ResNet, SqueezeNet, YOLOs...it's hard to keep up! I like a classic multilayer perceptron (MLP) myself, nothing more fancy than matrix multiplication and sigmoids. Stack five to seven layers with sigmoid activation, but feel free to use ReLu if you want something a bit more spicy.

Bake for 20–40 epochs at a high learning rate, then reduce learning rate and let the model simmer for another 20–40 epochs. Checking validation performance regularly. Add regularization to taste.

Now invite four friends over and build a web interface or API, and you're halfway to being a startup. I leave you, dear reader, with some parting advice from my mother's wisdom:

> *"Son, wash behind your ears and always make off-site backups."*

Bonne epoch! ✦ hello@humanreadablemag.com

OUR COLUMNISTS

Frogging code

By **Rob Landley**
Illustration by **Fauzy Lukman**

In the knitting community, "frogging" means pulling yarn back out to unravel rows of stitches when you spot a problem and need to redo a section. It's a lovely word, and programming should adopt it. Yesterday I frogged multiple functions when my prototype hit a snag and I had to change the design. I had the functions working, but they did the wrong thing, so they had to go.

My favorite quote from Ken Thompson, the creator of Unix, is "One of my most productive days was throwing away 1000 lines of code." Linus Torvalds says a patch that "removes more lines than it adds...is always nice to see."[1] Deleting existing, working code often counts as progress: you've learned by doing and come up with a better way.

Corporate management treats code as an investment, attaching a dollar value to each line and horrified when you get rid of any. The fallacy here is called "managing what you measure": if you don't understand what your employees do very well, but know how much they earn and how much they weigh, you might decide one employee is a good bargain at $80 per pound and another earning $100 per pound should either take a pay cut or gain weight.

Joel Spolsky made a good case against frogging code in his article "Things You Should Never Do Part I,"[2] highlighting the value of implicit knowledge in the existing codebase from all the real-world use cases it's adapted to handle, a testing load that can't easily be replicated in the lab, influencing a design that's never fully documented. This is the same sort of good advice for conventional proprietary software development as Brooks' Law[3], part of which says larger teams are less efficient because productivity increases linearly but communication costs increase exponentially.

But open-source development solves both problems by treating code submissions the way print magazines treat their "slush pile" of unsolicited contributions. Editors read through and reject 99% of them, bouncing a few entries back for revision ("I can't use this, but if you tried changing it this way I'd give it a second look..."), stitch the best few together into something vaguely coherent, and publish the next issue. This drastically improves scalability by replacing coordination with a discard/retry loop, and relies on Linus' Law ("given enough eyeballs, all bugs are shallow") to incorporate end-user feedback into the project's design loop. It's not just bug reporting; in a good open-source project, what the software should do and how it should do it is driven by the needs of the user base, often via the patches they submit to drive the project in a given direction.

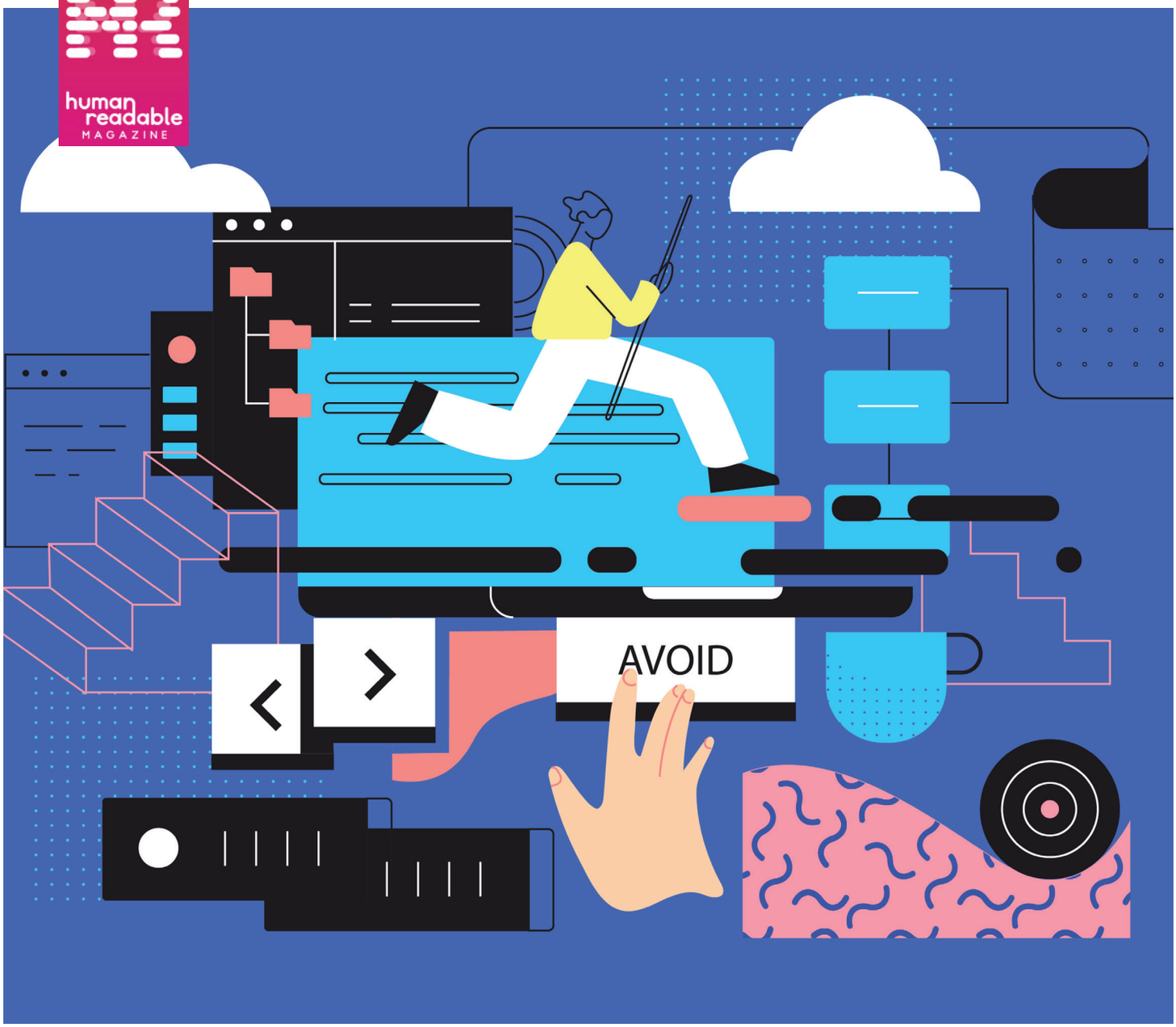
Frogging code, and lots of it, is fundamental to open source. When Netscape released its source code to the world to create Mozilla, its unwillingness to delete code led to the resignation of the project's maintainer[4]. On its way to becoming successful, Mozilla forked off the Galleon project (throwing away 90% of the code) and then forked off Firefox (throwing away 90% more). A proprietary project becoming an open-source project had a lot of moulting to do.

So don't worry about "creating negative value" when you delete code. Cry "Ribbit," and let slip the frogs of war. ✦

hello@humanreadablemag.com

ARTICLE LINKS

1. https://hrm.link/1000linesofcode
2. https://hrm.link/NeverDoPartI
3. https://hrm.link/BrooksLaw
4. https://hrm.link/maintainer-resignation



LANGUAGE FEATURES

How to Avoid Template Type Deduction in C++

Article by **Jonathan Boccara**
Main illustration by **Zara Magumyan**

TEMPLATE TYPE DEDUCTION IS AN AWESOME FEATURE OF C++. EXCEPT WHEN IT GETS IN YOUR WAY BY NOT DOING WHAT YOU WANT.BASED ON AN EXAMPLE OF A FUNCTION THAT CHANGES THE KEYS OF THE ELEMENTS OF AN STD::MAP, WE SEE HOW TO PREVENT TEMPLATE TYPE DEDUCTION TO KICK IN FOR SELECTED FUNCTION PARAMETERS.

MOTIVATING EXAMPLE: CHANGING THE KEYS OF A MAP OR SET

Here is how to change a key of the elements of a `std::map`:

```
auto myMap = std::map<std::string, int>{ {“one”, 1}, {“two”, 2}, {“three”, 3} };

auto node = myMap.extract(“two”);
if (!node.empty())
{
    node.key() = “dos”;
    myMap.insert(std::move(node));
}
```

This relies on the C++17 extract method of `std::map`.
And `std::set` has a similar way to perform the operation:

```
auto mySet = std::set<std::string>{“one”, “two”, “three”};

auto node = mySet.extract(“two”);
if(!node.empty())
{
    node.value() = “dos”;
    mySet.insert(std::move(node));
}
```

But you don’t want to write all this in your code. It would be nicer to have a function that encapsulates the mechanics of taking a node off the container and putting it back in, like this:

```
auto myMap = std::map<std::string, int>{ {“one”, 1}, {“two”, 2}, {“three”, 3} };
replace_key(myMap, “two”, “dos”);

auto mySet = std::set<std::string>{“one”, “two”, “three”};
replace_key(mySet, “two”, «dos”);
```

At first sight, this looks really simple because `std::map` and `std::set` seem to have the same code to extract a node and to put it back in. So we could have a generic function that takes a template parameter type for the container and be done with it.

Except they don’t have the same code. They almost have the same code, except for the method to change the value of the extracted node. If you look at line 6 in both of the above snippets, you’ll see that the one for `std::map` uses `node.key()` and the one for `std::set` uses `node.value()`. This difference can be surprising, especially since the values of sets are often themselves called “keys” as in `key_type`, which is an alias for `value_type` in `std::set`.

So we need to have two implementations for `replace_key`.

One way out of this issue is to call the function replace_value for `std::set`. Then we’d have two independent functions with two implementations. Until we decide to implement `replace_value` for `std::vectors` too, and we’re back to the need of having two overloads of a function that has different code for different containers.

In any case, I think it is interesting to see how to overload the function for several containers, because it is not trivial. In particular, it shows us how to deactivate the template type deduction of selected function parameters.

The Problem of Template Type Deduction

What’s all that fuss with overloading anyway? Let’s just write the overloads like this:

```
template<typename Key, typename Value>
void replace_key(std::map<Key, Value>& container, // this is the overload for
   maps
                 const Key& oldKey,
                 const Key& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
    {
        node.key() = newKey;
        container.insert(std::move(node));
    }
}

template<typename Key>
void replace_key(std::set<Key>& container, // this is the overload for sets
                 const Key& oldKey,
                 const Key& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
    {
        node.value() = newKey;
        container.insert(std::move(node));
    }
}
```

First off, once we know Key we can deduce type_identity_t<Key>: it is Key, by definition. But what makes it useful is that the deduction doesn't work the other way around: if the compiler knows Key, it doesn't try to deduce type_identity_t<Key>.

Let's now test them with the following calling code, for `std::map` to start:

```
auto myMap = std::map<std::string, int>{ {"one", 1}, {"two", 2}, {"three", 3} };
replace_key(myMap, "two", «dos");
```

What we get is a compilation error:

```
main.cpp: In function 'int main()':
main.cpp:35:32: error: no matching function for call to 'replace_key(std::map<std
  ::__cxx11::basic_string<char>, int>&, const char [4], const char [4])'
  replace_key(myMap, "two", "dos");
                             ^
main.cpp:7:6: note: candidate: 'template<class Key, class Value> void replace_
  key(std::map<Key, Value>&, const Key&, const Key&)'
 void replace_key(std::map<Key, Value>& container,
      ^~~~~~~~~~~
main.cpp:7:6: note:   template argument deduction/substitution failed:
main.cpp:35:32: note:   deduced conflicting types for parameter 'const Key'
  ('std::__cxx11::basic_string<char>' and 'char [4]')
 replace_key(myMap, "two", "dos");
                             ^
main.cpp:20:6: note: candidate: 'template<class Key> void replace_key(std::
  set<Key>&, const Key&, const Key&)'
 void replace_key(std::set<Key>& container,
      ^~~~~~~~~~~
main.cpp:20:6: note:   template argument deduction/substitution failed:
main.cpp:35:32: note:   'std::map<std::__cxx11::basic_string<char>, int>' is not
  derived from 'std::set<Key>'
 replace_key(myMap, "two", "dos");
                             ^
```

The problem is that "two" and "dos" are of type `char const *`, which is implicitly convertible to `std::string`. But there is no implicit conversion in the context of template type deduction (like we saw when handling multiple types with `std::max`). So `Key` is deduced as being of type `char const *`.

On the other hand, Container is deduced to be of `type std::map<std::string, int>` because this is the type of `myMap`. Because of this inconsistency in the deduced types of `Key`, the template generation fails, so the compiler doesn't find a `replace_key` function.

Note that we wouldn't have encountered this problem if we didn't rely on an implicit conversion—for example, if we had a `std::map<int, int>` or even a `std::map<int, std::string>`. But our generic function has to work with all types.

One way out is to work around the implicit conversion:

```
auto myMap = std::map<std::string, int>{ {"one", 1}, {"two", 2}, {"three", 3} };
replace_key(myMap, std::string("two"), std::string(«dos"));
```

Yuck! This doesn't look natural at all. Can't we do better?

Using type_identity

C++20 will define a new template type in the standard library: `std::type_identity`. This is the identity function in template metaprogramming. Or put another way, this is the template type defined by

```
template<typename T>
struct type_identity
{
    using type = T;
};
```

It comes with C++20's standard library, but it was implementable since C++98 (by using a `typedef` instead of `using`).

Like all metaprogramming functions, `std::type_identity` comes with its `_t` counterpart, to avoid the noisy `typename` and `::type` in calling code:

```
template< class T >
using type_identity_t = typename type_identity<T>::type;
```

We can use type_identity to prevent the deduction of some template parameters (thanks to Walletfox for showing me this technique!):

```
template<typename Key, typename Value>
void replace_key(std::map<Key, Value>& container,
                 const type_identity_t<Key>& oldKey,
                 const type_identity_t<Key>& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
```

ARTICLE LINKS

1. https://hrm.link/intention-templates

```
    {
        node.key() = newKey;
        container.insert(std::move(node));
    }
}
```

What does this change?

First off, once we know `Key` we can deduce `type_identity_t<Key>`: it is `Key`, by definition. But what makes it useful is that the deduction doesn't work the other way around: if the compiler knows `Key`, it doesn't try to deduce `type_identity_t<Key>`.

Indeed, there could be template specializations of `type_identity` that define type as something different. We could even have several specializations of type_identity that define type as the same thing. So you can't deduce `type_identity<Key>` if you only know `Key`.

As a result, `type_identity_t<Key>` doesn't take part in the deduction of template parameters in replace_key. Only Container does, and this is just enough for us because it allows us to deduce Key and Value.
This technique also works with `std::set`:

```
template<typename Key>
void replace_key(std::set<Key>& container,
                 const type_identity_t<Key>& oldKey,
                 const type_identity_t<Key>& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
    {
        node.value() = newKey;
        container.insert(std::move(node));
    }
}
```

A CURIOUS NAME

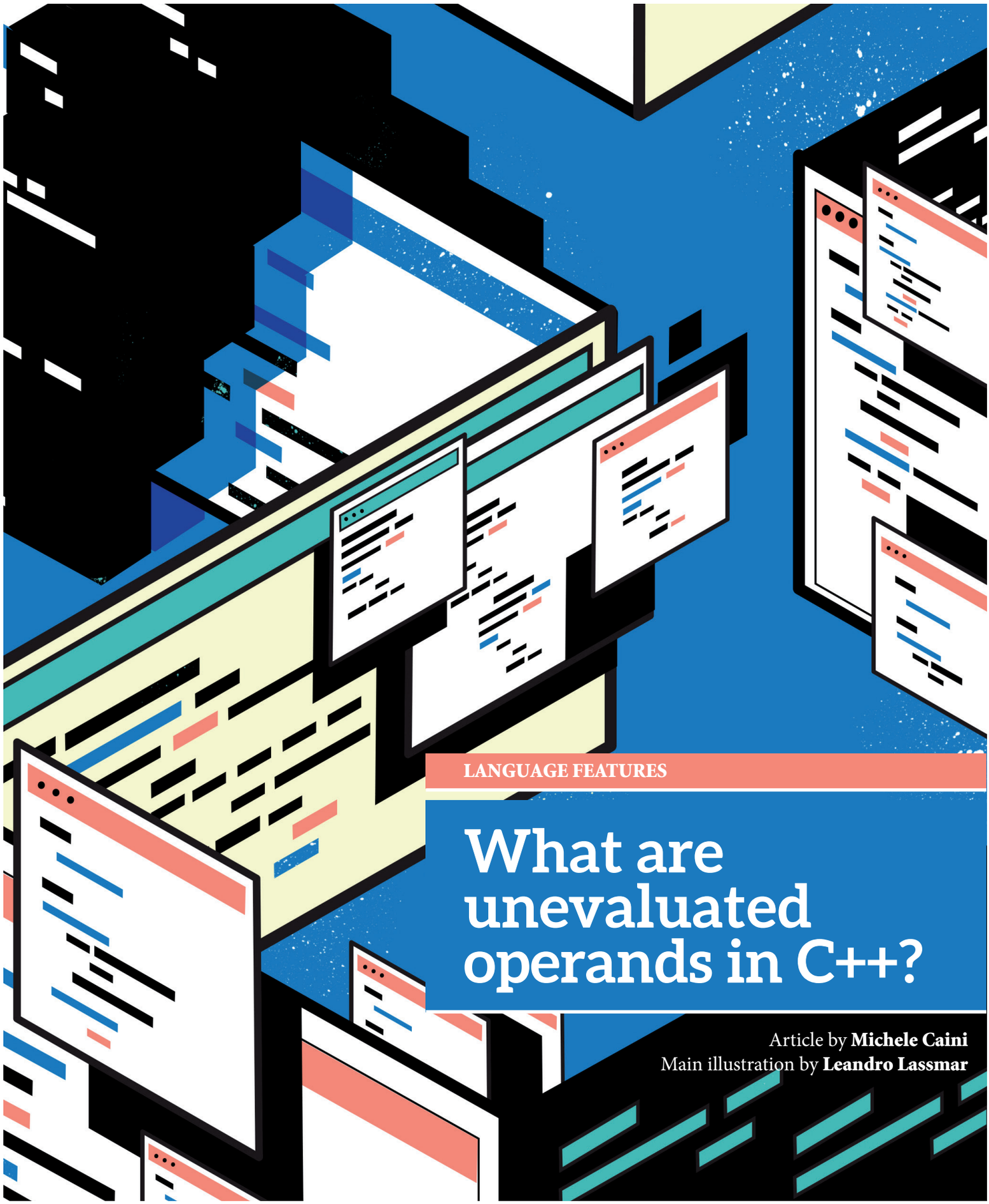
What we use type_identity for is to prevent type deduction. But this is not clear in the naming of our interface. To make our intention clearer in templates[1], we could use an alias for `type_identity_t`:

```
template< class T >
using not_deduced = type_identity_t<T>;

template<typename Key, typename Value>
void replace_key(std::map<Key, Value>& container,
                 const not_deduced<Key>& oldKey,
                 const not_deduced<Key>& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
    {
        node.key() = newKey;
        container.insert(std::move(node));
    }
}

template<typename Key>
void replace_key(std::set<Key>& container,
                 const not_deduced<Key>& oldKey,
                 const not_deduced<Key>& newKey)
{
    auto node = container.extract(oldKey);
    if(!node.empty())
    {
        node.value() = newKey;
        container.insert(std::move(node));
    }
}
```

There are advantages and drawbacks to doing this. `not_deduced` is not a standard name, so a reader of your code won't know it. It has to be clear enough for them to understand it means that `Key` won't be used for type deduction.
On the other hand, if they don't know the technique of using `type_identity` to avoid type deduction, the code with `type_identity_t` in the interface will raise more than one eyebrow.
If anything, it will help you later remember why you didn't just use Key in the interface.
What's your opinion? Should we use an alias for `type_identity_t` in this context? If so, what alias would you think make the intention clearer?
You can let me know in the comments section as well as let me know any other piece of feedback you have on this post or anything else on Fluent C++. ✦ hello@humanreadablemag.com



LANGUAGE FEATURES

What are unevaluated operands in C++?

Article by **Michele Caini**
Main illustration by **Leandro Lassmar**

YOU KNOW, THE C++ STANDARD IS AMAZING SOMETIMES. IT CONTAINS HIDDEN GIFTS FOR THE MOST CAREFUL READERS. ONE OF THESE GIFTS IS BURIED IN THE DEFINITION OF UNEVALUATED OPERANDS, BOTH SUCCINCT AND IMPORTANT:

> *In some contexts, unevaluated operands appear [...].*
> *An unevaluated operand is not evaluated.*

Let's try to dissect it and understand what the standard offers us by means of this often-underrated tool.

Introduction

To be honest, I've cheated a little. The full quote contains another statement that should make it clearer what an *unevaluated operand* is:

> *An unevaluated operand is considered a full-expression.*

In other words, the *unevaluated operands* are the operands of some operators of the language. They are expressions in all respects, but such that they are never evaluated. The reason is because those operators are there just to query the compile-time properties of their operands.

If this still doesn't seem interesting to you, note that not being evaluated means not giving rise to side effects. Right now, the term SFINAE is probably showing up in your thoughts along with many other fancy things, and unevaluated operands are getting more and more interesting.

What are these operators then? Up to C++17, there are four operators the operands of which are unevaluated: `typeof`, `sizeof`, `decltype`, and `noexcept`. C++20 will add a few other operators like them, but we have to wait a little longer for that.

So, why are these operators so special and what can we do with them?

decltype

If you've ever worked in modern C++, it's likely that you used `decltype` at least once. This is probably the most used operator when doing SFINAE.

To sum up and to avoid speaking *standardese* too much, its goal is to inspect the declared type of an element or of an expression. Let's take a look at an example of use:

```
template<typename T>
auto inspect(int, T &&item) -> decltype(item.func(), void()) { /* ... */ }

template<typename T>
void inspect(char, T &&item) { /* ... */ }

template<typename T>
void inspect(T &&item) { inspect(0, std::forward<T>(item)); }

// ...

inspect(std::string{«example"});
```

Here we are exploiting the *tag dispatching* idiom to literally *select* the right function to execute. As you can see, `decltype` is used to probe a compile-time feature of `item`, or better yet, of the type T that has a member function named `func`. The best part is that `func` isn't actually executed in this context because **(remember!)** the whole expression is an unevaluated operand of `decltype`.

In other words, this trick can be used to favor an overload when a given type has a member function named func. In all other cases, the fallback is executed. A bit of templates, the deduction rules, and our beloved SFINAE do the rest.

noexcept

So far, so good. `decltype` is used in many codebases and you've probably already seen enough examples of it.
What about `noexcept` instead? Can we do something similar with it? Actually, yes. As an example, consider the case in which we want to provide two different implementations of the same function: the former throws exceptions in case of errors; the latter doesn't make use of exceptions and returns error codes instead. The way we decide what function to use is by probing a given member and its `noexcept`-ness from the type we receive:

```
template<typename T>
auto inspect(int, T &&item) -> std::enable_if_t<noexcept(item.func())> {
    // ...
    throw;
}

template<typename T>
int inspect(char, T &&item) {
    // ...
    return 0;
}

template<typename T>
auto inspect(T &&item) { return inspect(0, std::forward<T>(item)); }
```

Because of how `std::enable_if_t` works, the first function is selected only if `T::func` has the `noexcept` qualifier, and in this case the return type is `void`. Otherwise, the second function is picked up and the return type is `int`; that is our error code.

Again, remember that the operands of the `noexcept` operator are unevaluated and therefore the function call `item.func()` is only taken in consideration to probe its compile-time feature, and we have no actual side effects at runtime when entering the function.

sizeof

`sizeof` isn't as good as the two operators above to do SFINAE. However, one can imagine some interesting uses for it, in particular when it comes to working with something like the small buffer optimization. In this case, we can exploit the properties of this operator to provide different implementations of a class template when the size of the type we use to specialize it fits that of a `void *`:

```
template<typename, typename = std::bool_constant<true>>
struct can_sbo { /* ... */ };

template<typename T>
struct can_sbo<T, std::bool_constant<sizeof(T) <= sizeof(void *)>> { /* ... */ };
```

I used `sizeof(T)` in the example, but we aren't constrained to it. In fact, we can use any expression we want. As an example `sizeof(T::member)`. Of course, it won't be evaluated.

typeid

Let's go further and see what can offer us `typeid`. As you know, it's purpose is to return information about types, nothing less and nothing more.
Unfortunately, this operator isn't very *SFINAE-friendly* and it's not worth it to show an example, although one can perhaps build something ad hoc with it.

THE CHOICE TRICK

We have seen how some operators whose operands are not evaluated can be useful in some cases. Now it's time to see one of them in action in a real-world case that I've faced more than once.
In particular, have you ever worked with templates and found yourself wanting to execute a function if the type has a given property, another function if it has a different property, or a third function as a fallback? Quite common indeed.
The hard way is something that looks like the following:

```
template<typename T>
std::enable_if_t<has_h<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<has_g<T> and !has_h<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_g<T> and !has_h<T> and has_f<T>>
invoke() { /* ... */ }

template<typename T>
std::enable_if_t<!has_f<T> and !has_g<T> and !has_h<T>>
invoke() { /* ... */ }
```

where `has_FUNC` is the typical detection idiom:

```
template<typename T, typename = void>
struct has_f: std::false_type {};

template<typename T>
struct has_f<T, std::void_t<decltype(std::declval<T>().f())>>
: std::true_type {};
```

Pretty annoying indeed, and the conditions become more and more complex in order to avoid ambiguities every time we want to add a switch to our cascade.
Fortunately, C++17 introduced `if constexpr` that clears this a bit:

```
template<typename T>
void invoke() {
  if constexpr(has_h<T>) {
    /* ... */
  } else if constexpr(has_g<T>) {
    /* ... */
  } else if constexpr(has_f<T>) {
    /* ... */
  } else {
    /* ... */
  }
}
```

Can we do something similar with it? Actually, yes. As an example, consider the case in which we want to provide two different implementations of the same function: the former throws exceptions in case of errors;

The choice trick is instead widely used and combines different aspects of the language. It may seem complicated initially, but it's really simple and allows us to solve a common problem with a very compact code.

Illustration by pikisuperstar - www.freepik.com

Umm, does it? We still have to define a lot of classes to detect properties (note that `has_f` serves only the purpose of probing a type for the member function `f`, but we want to also detect `g` and `h` in our example). Moreover, now we have to put everything in the body of the same function; that can be confusing and isn't desired in all cases.
How can we simplify this using one of the operators above?

First, let's introduce the `choice` class:

```
template<std::size_t N>
struct choice: choice<N-1> {};

template<>
struct choice<0> {};
```

The class is defined in such a way that `choice<N>` inherits from `choice<N-1>` and so on until `choice<0>`. It means that we can use `choice<N>` as an argument to a function that requires `choice<M>` as long as `M < N`.

We can now rewrite the first group of functions in a smarter way by means of this tool and using `decltype` as shown in the previous sections to probe (but not to evaluate!) our types and their compile-time properties:

```
template<typename T>
auto invoke(choice<3>)
-> decltype(std::declval<T>().h(), void())
{ /* ... */ }

template<typename T>
auto invoke(choice<2>)
-> decltype(std::declval<T>().g(), void())
{ /* ... */ }

template<typename T>
auto invoke(choice<1>)
-> decltype(std::declval<T>().f(), void())
{ /* ... */ }

template<typename T>
void invoke(choice<0>)
{ /* ... */ }

template<typename T>
void invoke() {
  invoke<T>(choice<100>{});
}
```

How does it work? Because of the rules of the language, the first function that matches the given arguments is as follows:

```
template<typename T>
auto invoke(choice<3>)
-> decltype(std::declval<T>().h(), void())
{ /* ... */ }
```

Here we use `decltype` to probe a compile-time property for the type `T`. Probe, not evaluate. Therefore, we have no side effects here.
SFINAE does the rest for us. In case the type `T` has a member `h`, we enter the first function. Otherwise, we receive a soft error, but the compiler continues to probe the other functions to turn it into a hard error and return to us. This isn't even an option actually because of our fallback that will accept everything that doesn't match one of the previous cases:

```
template<typename T>
void invoke(choice<0>)
{ /* ... */ }
```

Another important thing that perhaps doesn't immediately catch our attention is that there is no need to resort to the detection idiom to test our types, which relieves us from having to write a lot of code. Finally, we have as many functions as there are rules, something that is definitely easy to maintain and to reason on.

CONCLUSION

We have seen how the C++ language offers a few very interesting operators. Some aren't very useful when you want to do SFINAE; others can be used within certain limits, but one in particular seems to be good enough for most of the cases: `decltype`.
The choice trick is instead widely used and combines different aspects of the language. It may seem complicated initially, but it's really simple and allows us to solve a common problem with a very compact code. You've probably already encountered it in a simpler form, where the overload is solved by a combination of int and `char`, but the way it works is exactly the same. We just walked through an extended version of it.

Obviously the uses and abuses of `decltype` aren't limited to this example, but I'll leave the rest for future articles, hoping that you enjoyed what you've read so far. ✦ hello@humanreadablemag.com

LIBRARIES

Functional Ruby with `dry-monad`s

Article by **Michael Kohl**
Main illustration by **Sergey Konotopcev**

WHILE RUBY MAY NOT BE THE COOL NEW KID ON THE BLOCK ANYMORE, THERE'S BARELY BEEN A BETTER TIME TO BE A RUBYIST. THIS IS NOT ONLY DUE TO CONSTANT LANGUAGE IMPROVEMENTS, BUT ALSO BECAUSE OF A NEW GENERATION OF GEMS THAT ARE FRAMEWORK AGNOSTIC AND ARE DESIGNED AROUND PLAIN OLD RUBY OBJECTS (PORO).

The dry-rb [1] collection of gems is a great example of this approach, and in this article, we'll explore how `dry-monads` can help with modeling complex data transformations robustly.

The M-Word: Monad Basics

In many programming circles, monads are seen as arcane constructs that are only of interest to academics. This is unfortunate since they are not all that complicated: essentially monads are just a way to perform a series of computations within a "context." To do this, a "plain" value like a string or integer is first "wrapped" by a function called `return` in FP jargon. These wrapped values are then combined with an operation called `bind`, which removes their wrapping, performs the desired operation on their underlying value, and rewraps the result. In the end, the context is removed again, often through a concept called pattern-matching [2].

`dry-monad` in Action

All of this may still sound too theoretical, so let's look at the Maybe [3] monad as a concrete example. `Maybe`'s purpose is to model a series of computations that could return `nil` at any intermediate step. Instead of mixing business logic with repeated error checks, we "wrap" our starting value in a `Maybe`, perform all our operations, and only check the result at the very end, where it will either be of the form `Some(value)` when everything went according to plan, or `None` when a `nil` was encountered.

In a web application this could, for example, be used to return the uppercase version of a user's name, or the default value "ANONYMOUS USER" if there's no currently logged-in user or the name isn't set. Let's look at this example step by step. First, we require the Maybe monad and alias the `Dry::Monads` module to `M` to save ourselves some typing. We also set up a dummy user:

```
require 'dry/monads/maybe'

M = Dry::Monads
current_user = nil
```

Our `maybe_name` function first "wraps" the user in a `Maybe` context and uses the `bind` method to apply a block to this monadic value. Inside the block, we try to access the user's name and then repeat the same process to finally call `upcase` on it:

```
def maybe_name(user)
  M.Maybe(user).bind do |u|
    M.Maybe(u.name).bind do |n|
      M.Maybe(n.upcase)
    end
  end
end

maybe_name(current_user)
#=> None
```

Note that this doesn't require any specific checks for `nil` values. If `nil` is encountered `Maybe` returns `None`, which all subsequent steps will just pass through without trying to perform any further operations on it.

To extract the actual result we have two choices: the unsafe `value!` method, which will raise an error for `None` values, or the preferred `value_or` alternative, which allows the caller to specify a sensible default value:

```
maybe_name(current_user).value!
#=> Dry::Monads::UnwrapError: value! was called on None
maybe_name(current_user).value_or("ANONYMOUS USER")
#=> "ANONYMOUS USER»
```

Now let's try this again with an actual user:

```
user = OpenStruct.new(name: "john monadoe")

maybe_name(current_user)
#=> Some("JOHN MONADOE")
maybe_name(current_user).value!
#=> "JOHN MONADOE"
maybe_name(current_user).value_or("ANONYMOUS USER")
#=> "JOHN MONADOE»
```

Success! Admittedly the `maybe_name` function is quite verbose, especially compared to Ruby's "lonely operator" (`&.`) or Rail's `try` method, which essentially achieves the same results. However, this was mostly done for demonstration purposes; generally one would use `fmap` in this case, which, unlike `bind`, works with blocks that return unwrapped values and automatically rewraps the result:

```
M.Maybe(nil).fmap(&:name).fmap(&:upcase).value_or("ANONYMOUS USER")
"ANONYMOUS USER"

M.Maybe(current_user).fmap(&:name).fmap(&:upcase).value_or("ANONYMOUS USER")
"JOHN MONADOE"
#=> Some("JOHN MONADOE»)
```

Other Useful Monads

At this point, you may still wonder if all of this effort is really worth it just to avoid a couple of `nil` checks. However, there are different "contexts" that have been modeled as monads, and everything we covered so far (`bind`, `fmap`) also applies to them.

The `Result` monad is similar to `Maybe`, but instead of `None`, it allows us to return an error object with additional information. For example, here's a `sqrt` function, which provides an exception-safe wrapper around Ruby's `Math.sqrt`:

```
require 'dry/monads/result'

def sqrt(n)
  return M.Failure("Value needs to be >= 0") if n < 0
  M.Success(Math.sqrt(n))
end

sqrt(9)
#=> Success(3.0)

sqrt(-1)
#=> Failure("Value needs to be >= 0»)
```

If the input value `n` is outside the acceptable range, we return an error message wrapped in `Failure`; otherwise, the result is wrapped in `Success`. Of course these values are composable too:

```
sqrt(9).fmap { |n| n + 1 }.value_or(0)
#=> 4.0
sqrt(-1).fmap { |n| n + 1 }.value_or(0)
#=> 0
```

The `Result` monad is used to great effect in the dry-transaction [4] gem, which provides a business workflow DSL and is also available as an extension to dry-validation [5], a library for defining schemas and their accompanying validation rules.
Another useful monad is `Try`, which can be used for wrapping code that can potentially raise exceptions:
In case the user enters `0` (or just hits enter),

```
Try { 1 / 0 }.fmap { |n| n + 1 }
Try::Error(ZeroDivisionError: divided by 0)
```

Dividing one by zero would cause a `ZeroDivisionError`, but instead an instance of `Try::Error` is returned. With valid input, everything works as expected, and we'll receive `Try::Value` instead:

```
Try { 1 / 1 }.fmap { |n| n + 1 }
#=> Try::Value(2)
Try { 1 / 1 }.bind { |n| n + 1 }
#=>2
```

The possible result of a `Try` operation can be converted to a `Result` or `Maybe` value by using `to_result` or `to_maybe`.

Do Notation

Functional languages like Haskell and Scala provide a special syntax for working with monads, called "do notation." While it's not possible to mirror this exactly in Ruby, `dry-monads` provides a reasonable alternative [6].

The following example demonstrates how a function for transferring money could use do notation to sequence steps that can fail:

```
require 'dry/monads/result'
require 'dry/monads/do/all'

def transfer_money(params)
  sender = yield fetch_user(params[:sender_id])
  receiver = yield fetch_user(params[:receiver_id])
  amount = yield verify_amount(params[:amount])
```

Functional languages like Haskell and Scala provide a special syntax for working with monads, called "do notation." While it's not possible to mirror this exactly in Ruby, dry-monads provides a reasonable alternative.

```
  receipt = yield transfer(sender, receiver, amount)

  Success([sender, receiver, receipt])
end

def fetch_user(user_id)
  # Success(user) or Failure(error)
end

def verify_amount(amount)
  # Success(amount) or Failure(error)
end

def transfer(sender, receiver, amount)
  # Success(receipt) or Failure(error)
end
```

In the above example, every step of the process returns a `Result` value and `dry-monads` do notation uses a clever trick to extract the value from a monadic object in each method we're `yield`ing to. As soon as a `Failure` is encountered, the whole process short-circuits; otherwise, the unwrapped `Success` value gets returned.

Case Equality and Pattern Matching

Another nice feature of dry-monads is that it works with Ruby's case statement:

```
case maybe_name
when Some("JOHN MONADOE") then :john
when Some("LARRY LAMBDA") then :larry
when Some(_) then :generic_user
else :anonymous_user
end
```

Additionally, `dry-monads` also provides pattern matching with the help of dry-matcher [7]. Let's say we have a function called `login`, which authenticates a user and returns either `Success(user)` or `Failure(error)`. We can then use it in our controller like this:

```
require 'dry/matcher/result_matcher'

include Dry::Matcher.for(:login, with: Dry::Matcher::ResultMatcher)
```

```
def login
  # Success(user) or Failure(error)
end

login(user) do |m|
  m.success do |user|
    # Success case, e.g. redirect to profile page
  end

  m.failure do |err|
    # Error case, e.g. setting flash to error message
  end
end
```

This turns error handling into a first-class construct since pattern matching will fail when one of the cases is missing. So if we remove the `failure` block from the above snippet, the following exception will be raised:

```
Dry::Matcher::NonExhaustiveMatchError: cases +failure+ not handled
```

Summary

Hopefully this article demystified monads a little bit and provided some ideas and insights into how the `dry-monads` gem can be used to clean up your application code by turning concepts like failure (`Result`), absence of value (`Maybe`), or computations that can fail (`Try`) into first-class constructs that follow a common pattern and can be easily composed.✦ hello@humanreadablemag.com

ARTICLE LINKS

1. https://hrm.link/dry-rb
2. https://hrm.link/pattern-matching
3. https://hrm.link/Maybe-monad
4. https://hrm.link/dry-transaction
5. https://hrm.link/dry-validation
6. https://hrm.link/reasonable-alternative
7. https://hrm.link/dry-matcher

STORY TIME

The Plight of Open Source

Article by **Omar Shehata**
Main illustration by **Leandro Lassmar**

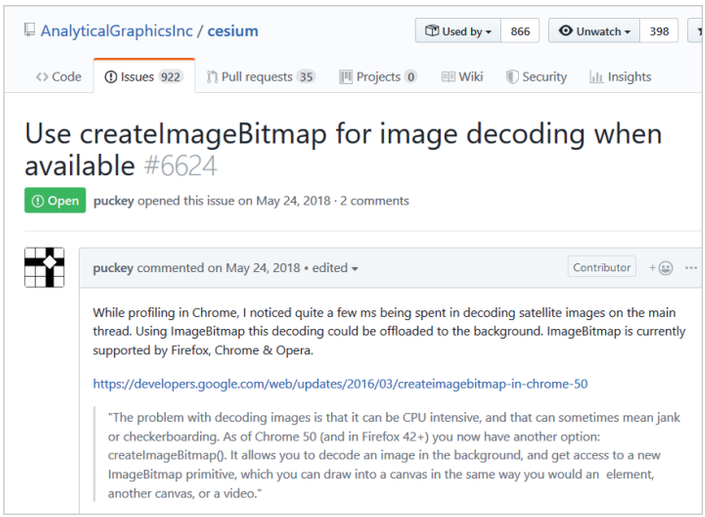
YOU'RE WORKING ON A VERY IMPORTANT PROJECT WITH A HARD DEADLINE LOOMING. THERE'S A CRITICAL BUG YOU'VE BEEN CHASING, AND YOU DISCOVER IT'S IN ONE OF THE THIRD-PARTY LIBRARIES YOU'RE USING. THANKFULLY, YOU HAD THE FORESIGHT TO RELY ON A MOSTLY OPEN-SOURCE STACK, SO EVEN THOUGH YOU'RE UNFAMILIAR WITH THE CODEBASE, IT DOESN'T TAKE TOO LONG TO TRACK THIS BUG DOWN AND PATCH IT. THE DAY IS SAVED, THE CLIENT IS HAPPY, AND YOU FEEL TRIUMPHANT!

But your story doesn't end there because you believe in giving back to open source. After all, so much of our society's digital infrastructure relies on open-source software, including your company's business. So you jump through some hoops with the legal department to sign a contributor license agreement, you spend your free time on the weekend cleaning up your patch, and you submit a pull request. The open-source maintainers thank you and say they'll try to look at it soon. You take pride in the fact that your charitable act will at the very least save someone the half a day of frustration it took you to track this down. You move on knowing you've made a valuable contribution to society. You've left the world a little bit better than how you found it.
But you forgot to test your valuable contribution in Firefox, and one of the newer maintainers discovers a bug and decides to fix it.
How hard could it possibly be?

Act I: A Deceptively Simple One-Liner

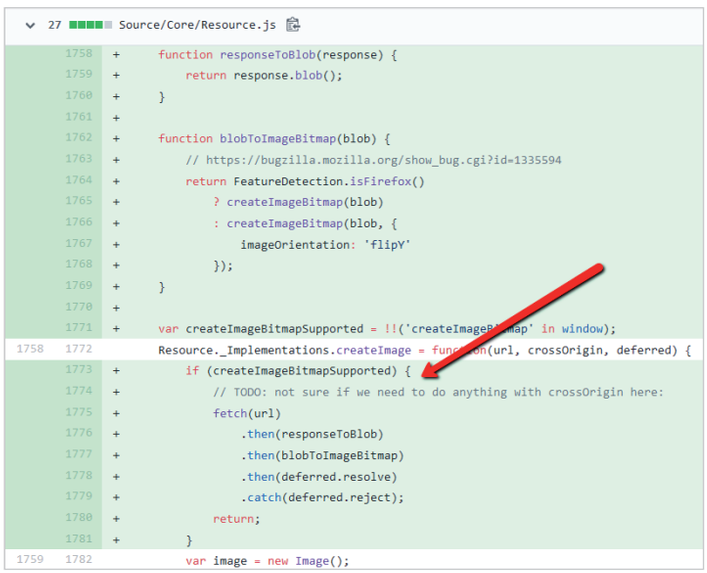
Our story begins, not with a bug, but with a benign feature request on our CesiumJS repository—to use a new browser feature to speed up image decoding.

CesiumJS is a library for visualizing 3D geospatial data on the web. Most applications that use it have some kind of satellite images as a base map, so implementing something to improve how we load these images would be great for almost all of our users. Not only did Puckey let us know about this new browser feature, he also opened a pull request implementing it two hours later!



The original feature request to speed up image decoding using the new ImageBitmapThe original feature request to speed up image decoding using the new ImageBitmap

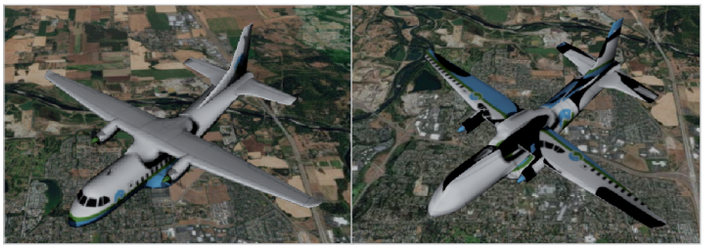
I had been part of the Cesium team for less than a year at that point, and I'd never maintained any open-source libraries, let alone one as large and as popular as CesiumJS, with over a quarter million lines of code and thousands of weekly downloads, so I was very excited to see the great promise of open source playing out: we had shared our code with the world, and the world was sharing in the burden of developing it.
The pull request was pretty straightforward; aside from some helper functions, it mostly came down to one line that said: if the browser supports ImageBitmap, use that to load the image instead of the old way, which used an HTML image element. Using ImageBitmap meant decoding the image happened on a background thread, so we'd have a smoother frame rate and potentially load scenes faster since images could now be decoded in parallel.



The original PR in its entirely. It just checked if ImageBitmap [1] was supported and used it.

It was now my job to kick this into production. I estimated it would take me one to two days to run some before and after benchmarks, write some unit tests, and make sure it worked in all browsers. Everything was going well in my initial testing. I could see there was noticeably less camera stuttering in specific situations, and the Chrome profiler showed a higher average frame rate than before!

There was just one tiny problem: all 3D models in the engine now had upside-down textures.



Airplane model in CesiumJS, before implementing ImageBitmap (left) and after (right).

Racing Bicycle model with upside down textures in CesiumJS after implementing ImageBitmap. Model by Francesco Coldesina [2].

I could tell Puckey's use case did not involve 3D models (contrary to ours, where a large part of our business is a cloud platform for hosting and tiling massive 3D models that users could then view in CesiumJS).

At this point, I still held strongly to my faith in this valuable contribution. Sure, there was one small oversight here, but it was my job as a reviewer to catch this sort of thing. Everything else was working great. Maybe Puckey just missed something obvious, like some "flip all images upside down" flag that was accidentally set.

To my delight, I did actually find exactly such a flag, right in one of the helper functions in the PR!

```
function blobToImageBitmap(blob) {
          // https://bugzilla.mozilla.org/show_bug.cgi?id=1335594
          return FeatureDetection.isFirefox()
          ? createImageBitmap(blob)
          : createImageBitmap(blob, {
          imageOrientation: 'flipY'
          });
}
```

Code snippet from the original ImageBitmap PR which would flip the image vertically when not using Firefox. MDN link for ImageBitmap [3].

It was a little peculiar that this flag was set to flip only when it doesn't detect Firefox (which just meant Chrome, since they were the only two that supported ImageBitmap). The Bugzilla page linked in the code just said that Firefox doesn't support this flip option. Given that everything was displaying correctly in Firefox, this seemed like a really easy fix. If we didn't want images to be upside down, then we shouldn't be flipping them upside down when decoding them!

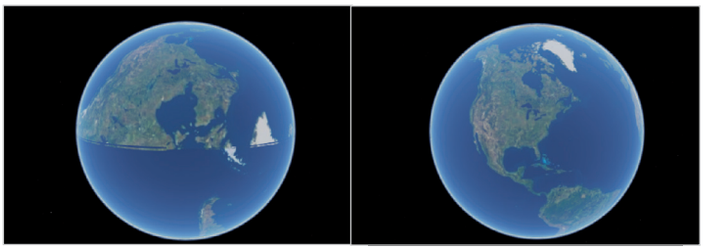
Even though it was clear that Puckey had used that flag intentionally, it seemed worth a shot. I removed it, and at first glance, everything seemed correct! This seemed too good to be true. I couldn't see what was wrong until I zoomed out a bit.

Removing the flipY option fixed the models, but something else seemed off.

I was very excited to see the great promise of open source playing out: we had shared our code with the world, and the world was sharing in the burden of developing it.

We'd have to account for the fact that some browsers might be allowing ImageBitmaps to be flipped, despite what the spec says. Flipping successfully both on decode and on texture upload would produce the wrong result again.

And then I zoomed out a little bit more.



Fixing the upside down 3D model textures caused all satellite images to be upside down (left) compared to original (right).

Now the whole world was upside down! At this point I was convinced this was a browser bug. It worked perfectly fine in Firefox after all, and ImageBitmap was a relatively newer browser API, so it wouldn't have been too surprising.

Now, in order to write a bug report for Chrome, you had to cross this barrier between the "normal developer" side and peek into "hardcore low-level engineer" land. As much as I knew that writing web apps was just a different kind of complexity than writing web browsers, and not necessarily an indication of skill or intelligence, it was still rather intimidating. To me, these engine programmers were the folks who laid down the roads I used and relied on every day. And here I was, using those very same roads to walk up to their doors and criticize their work.

Given my trepidation, I made sure my bug report was well researched. The exact problem was that when uploading the texture to the GPU, you had an option to flip it vertically, but this option seemed to be ignored when using ImageBitmap in Chrome. I learned that CesiumJS was relying on this to flip all satellite images, and nothing else (they appeared right-side up in the engine because the shader code would orient them correctly again). The MDN docs [4] stated that any source image could be flipped this way when uploading to the GPU. And since a source image could be a regular image element, or the fancy new ImageBitmap, this was, therefore, a bug!

Now just as I'm about to post my beautifully well-crafted bug report, I stumble on this comment in an older Chrome issue:
Currently the functions texImage2D and texSubImage2D does re-decoding when the TexImageSource is an ImageBitmap. The spec recently changed such that when the TexImageSource is an ImageBitmap, ***the pixel storage parameters such as flipY, premultiplyAlpha will be ignored.***

It turns out ignoring the flipY option was a feature, not a bug [5]!

My heart sank as I read these words. The spec had changed. These were forces that were far beyond my grasp. If the browser engineers were the road-builders, the authors of the web specification were like architects that ordained our fate. Ignoring the flip option was not a bug after all. The higher power in the ivory tower had changed their mind. There was no arguing with that. Their will be done.

I was starting to panic now. The only reason it worked in Firefox was because, unlike Chrome, they hadn't "fixed" this bug yet (which was three years old at this point). To fix this on our end, I would have to design a way for images that were meant to be flipped on GPU-upload to also be flipped during fetch and decode. This was complicated for two reasons:

Fetching the image was part of the resource request system, which was completely decoupled from the texture upload system. This was good design, but it meant there was no central way to implement this. I would have to find all instances in our codebase where a texture was being flipped on upload, trace back to where it was being fetched, and make sure it was getting flipped there as well.

We'd have to account for the fact that some browsers might be allowing ImageBitmaps to be flipped, despite what the spec says. Flipping successfully both on decode and on texture upload would produce the wrong result again.

So it wasn't impossible, but it was a much larger and messier change. I would no longer be reviewing Puckey's PR at that point—I'd be tossing that and starting from scratch. But I really wanted this to be a story about the community giving back to the common good, so I wanted to give it one more shot.
What if I could fix the source of the issue, which was that the satellite images in CesiumJS had a different expected orientation from everything else in the engine. After all, how hard could it be to flip an image in a shader?

I tried it out, and was quickly able to get the right orientation! But it wasn't quite the right shape...

I forgot to take into account that part of the process of drawing these flat images involved projecting them onto a 3D globe. Flipping the pixels without the corresponding projection logic meant all the projection was now reversed, and the images were very distorted.

Getting the right orientation for satellite images in the shader was easy, but I forgot about all the projection logic that would need to be updated too.

Fixing this would involve treading deep into CesiumJS code that was almost a decade old. This geographic projection was one of the very first features in the library, and it was one of the reasons users pick CesiumJS over other general 3D engines. I knew I had gone too deep when I stumbled on a 400-word essay in the code detailing the history of the quirks and clever tricks of projection in the engine. I dared not add another entry to this essay.

Two satellite image tiles and their expected projection on a 3D globe.

I retreated and implemented the "messy" solution I had come up with earlier after all. Puckey's original clear and concise PR had now mutated into this bloated, ugly implementation that threaded across several otherwise decoupled systems. You now had to make sure that any image in the engine had the correct orientation both on fetch and on GPU upload (and remember that Firefox doesn't support flip on fetch, so don't even try there).

It was confusing, but it was finally done. I spent a significant amount of time writing new unit tests and fixing hundreds of old ones (which were all relying on core library functions returning a plain image instead of an ImageBitmap).

The final PR ended up being 893 lines of code across 30 files, but it finally worked. I profiled the code one last time to write up the exact performance gain we got from this whole ordeal, only to discover that any measured gain I had seen before was now gone...

Act II: Searching for Truth

This was a deeply disturbing discovery. I suddenly had absolutely nothing to show for this massive code refactor.

The claim was that ImageBitmap allowed decoding images in a background thread, but the Chrome profiler was clearly showing that the main thread was still blocked on decoding. The original source in Puckey's issue was a Google developer article stating that ImageBitmap "allows you to decode an image in the background", so I created a minimal test to prove this to myself.

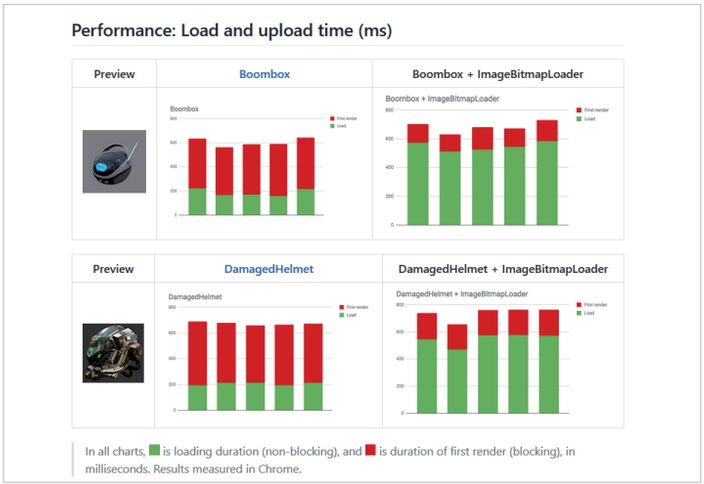
```
var image = new Image();
image.src = 'big-test-image.jpeg';

var t0 = performance.now();
createImageBitmap(image)
var t1 = performance.now();
console.log(t1 - t0);// t1 - t0 was ~500 ms
```

The createImageBitmap function returns a promise that resolves when the decoding is done. If it was really non-blocking, then the reported time between t1 and t0 should be 0. Instead, it was almost 500 ms, way more than can fit in the 16 ms frame budget needed to maintain a smooth 60 fps.

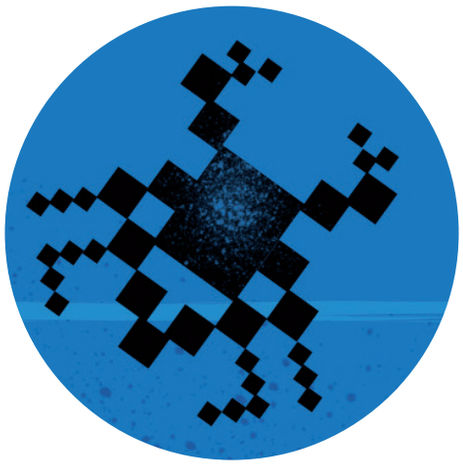
I began to doubt everything I knew now. Did I ever really see a performance improvement? My initial testing was not as thorough as it was now, partially because I had relied on and trusted the common wisdom of the internet. Everyone on the internet seemed to be convinced that it was non-blocking, and yet these five lines right in front of me proved otherwise.

My initial source of truth was the ThreeJS developers, heroes of 3D web graphics that I had long admired, who had reported success with their benchmarks after using ImageBitmap. These were the industry experts, so I took their word for it.



The ThreeJS developers had reported significantly less blocking time after using ImageBitmap in their PR. I trusted their benchmarks.

Luckily, our team lead happened to be friends with some of the people on the Chrome WebGL team and connected me to them so I could just directly ask the team how it works.



It couldn't possibly be that everyone was wrong. I began to dig deeper, beyond StackOverflow answers and blogs and into those CSS-less mailing lists. I found discussions claiming that ImageBitmap always blocked the thread it was called in, so you should always call it in a web worker. ThreeJS's loader had always used a worker thread anyway, so perhaps that was why they saw the performance boost and didn't notice that it was actually blocking (in that thread).

This seemed plausible, and was something I could work with (it would just involve another big refactor to use web workers in our image loader). But there were other rumors that ImageBitmap always decoded in the main thread, and that it was non-blocking in that it relied on "idle time" to do the decoding. This probably worked great for most web pages and single-page apps, but a real-time application like CesiumJS had no "idle time."

If anyone had the truth in this mire of guesses and speculations, it had to be the HTML specification. The spec said that an ImageBitmap is an object that can be painted to a canvas "without undue latency" with the following note:

> *The exact judgment of what is undue latency of this is left up to the implementer, but in general if making use of the bitmap requires network I/O, or even local disk I/O, then the latency is probably undue; whereas if it only requires a blocking read from a GPU or system RAM, the latency is probably acceptable.*

Note from the HTML spec [6] on ImageBitmap's decoding.

"Probably acceptable" and "probably undue" were not the words I expected from the people with the power to design the internet. It occurred to me in that moment that the spec was less like a decree and more like the grammar of a common language. We're all better off if everyone in the community adopts it, so we can all communicate better, but it had to serve the diverse and evolving needs of the community or risk becoming irrelevant. The power to make the decisions did not flow from top to bottom. The spec writers have to make decisions that they think everyone would follow (which the browser creators can disregard, as in the case of Firefox allowing ImageBitmap to be flipped when the spec says it shouldn't). The browser creators in turn are constrained by what the website authors create and expect, who are in turn constrained by what the end user does and expects.

I saw in that moment a beautiful order to our online world—where I as a web developer held as much sway in the molding of the internet as its original creators. I felt incredibly empowered, but also I still had a day job with deadlines and a manager to answer to (who was not as impressed with this epiphany as I was). And I was even more confused now about how ImageBitmap was supposed to work.

Luckily, our team lead happened to be friends with some of the people on the Chrome WebGL team and connected me to them so I could just directly ask the team how it works. They explained what I already knew: ImageBitmap is supposed to be non-blocking, yadda, yadda. I asked them to explain, then, why my code-snippet was blocking. To my surprise, they were surprised!

They recommended calling it with a blob instead of an HTMLImageElement:

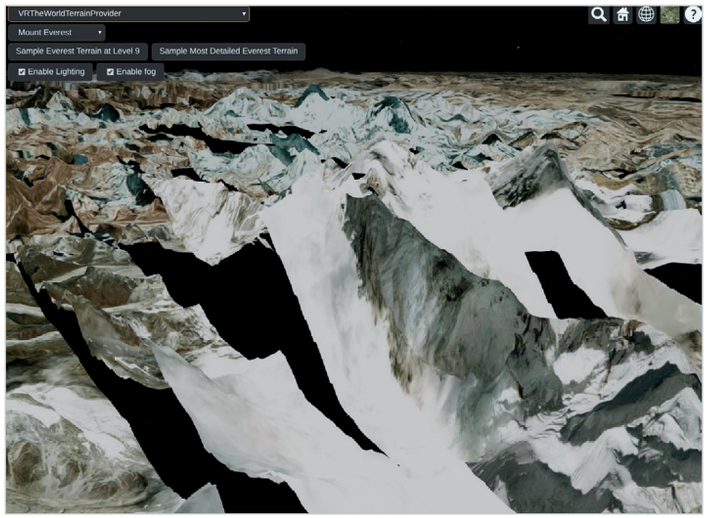
```
fetch('image.jpeg')
  .then(resp => resp.blob())
  .then(blob => {
    const t0 = performance.now();
    const ib = createImageBitmap(blob);
    const t1 = performance.now();
    console.log(t1 - t0);// t1 - t0 was now 0 ms!
    return ib;
  });
```

It turned out that since an HTMLImageElement was part of the DOM, which was not accessible in worker threads, Chrome reverted to decoding on the main thread to read it. Passing in a blob instead correctly moved the decoding off the main thread, and the measured performance improvement was back.

It was finally time to ship.

Act III: The Part Where Everything Breaks

Despite a thorough 20-day code review with much back and forth and deliberation, someone on our team ran into a pretty bad bug in our final testing on release day. One of our code examples on how to load 3D terrain had massive cracks in it:



One of our code examples in the release branch has large cracks in the ground.

Git bisect traced the issue back to my ImageBitmap code. At first glance, it looked like just an instance where I forgot to set the right orientation for this particular code path, so I went in, flipped it, and it was all good again! It was a bit strange because it wasn't the satellite images that were upside down for Mt. Everest here, it was the 3D geometry. I was sure there was a fascinating story behind how those were connected, but my biggest concern at that point was making sure I didn't hold back the release, so I pushed this quick fix. Everything seemed good and the new CesiumJS 1.56 promptly went out.

Immediately the next day we started getting bug reports from developers complaining that updating to 1.56 broke their applications. I saw instances of upside-down icons or custom post-processing effects that just stopped working. We were all very careful. How could this have happened?

I blamed myself for not further investigating that case of upside-down Mt. Everest geometry. My mistake was setting the default flipY parameter on fetching images to true. This made sense because I figured most images fetched would end up getting uploaded to the GPU for rendering. If it didn't flip by default, then any user who was fetching an image and creating a WebGL texture out of it would suddenly get upside-down textures. But because it flipped by default, it meant any use that didn't involve using it as a WebGL texture was upside down, such as if you were reading the pixels of the image to generate geometry (which is why Mt. Everest was upside down) or if you were rendering to a 2D canvas (which is why the icons and some post-processing effects were upside down).

So no matter what choice I had made, it would have broken something. To make matters worse, there were certain situations that just completely broke (apparently I had missed the note in the HTML spec [7] that declared SVGs invalid for decoding with ImageBitmap). The only robust fix was to backpedal on the whole thing, to no longer use ImageBitmap by default at all, except for situations internally in the engine that we controlled, such as loading satellite images and 3D models.

I implemented that patch and we released CesiumJS 1.56.1 one day after the 1.56 release. In the entire seven-year history of monthly CesiumJS releases, there had only ever been two other dot releases, and neither was so serious that it was a day after the main release. In the end, it took four pull requests, over a thousand lines of code across over 40 files and 81 days from the day I started looking at Puckey's initial PR to release what started out as a more or less one-line contribution.

This may have been an extreme case, but it's not that rare (for example, the logarithmic depth buffer [8] in CesiumJS got released in a very similar way; a quick contribution followed by months of work by the maintainers to get it ready). It can be very hard as an external contributor to know the full ramifications of your changes, and of course, the maintainers are responsible for any future bug it may cause. That's not to say I think it's bad to contribute to open source, but that I had severely underestimated how much work it takes from the other side before taking on this role. I also now see why contributions like writing documentation, tutorials, and just answering questions in the community are incredibly valuable.

This is also why I'm very optimistic seeing projects like the Open Collective [9] take off. Most of the time the best way to help open source is just to fund it! ✦ hello@humanreadablemag.com

ARTICLE LINKS

1. https://hrm.link/ImageBitmap
2. https://hrm.link/Coldesina
3. https://hrm.link/MDNlink
4. https://hrm.link/MDNdocs
5. https://hrm.link/flipYbug
6. https://hrm.link/*HTMLspec*
7. https://hrm.link/noteHTML
8. https://hrm.link/depthbuffer
9. https://hrm.link/OpenCollective

human readable MAGAZINE

SECURITY

Cyber Security - Encryption Key Exchanges

Article by **Phil Pearl**
Main illustration by **Sergey Konotopcev**

ENCRYPTION OF DATA, BOTH AT REST IN DATABASES AND WHILE IN TRANSIT, ENSURES BOTH SECURITY AND PRIVACY, AND WITH THAT GOES THE NEED TO PROTECT PRIVATE ENCRYPTION KEYS. THE RSA ASYMMETRIC ENCRYPTION METHOD USES A FREELY AVAILABLE PUBLIC KEY TO ENCRYPT DATA, WHICH IS THEN DECRYPTED WITH A MATHEMATICALLY LINKED PRIVATE KEY.

Hashing functions check message validity by irreversibly encrypting data of any length to a fixed length hash value. The receiver of the file or message then calculates the hash value with the agreed-to method, such as SHA-256, and compares that value to the known, public hash value. But asymmetric encryption is much slower than symmetric encryption [1], which uses the same key to encrypt and decrypt messages. And both encryption methods require secure exchange of encryption keys between the parties.
The Diffie-Hellman encryption key exchanges discussed here avoid actually sending encryption keys between parties. The keys are computed mathematically from actual values exchanged, and these values alone are insufficient for eavesdroppers to derive keys.

Human factors cause most security breaches, by users answering phishing emails or downloading malware from compromised websites or fake sites posing as the real thing. The best security is useless if you've been hacked and maybe keystroke loggers or screen scrapers have compromised your computer. But antivirus software helps to protect against this. Virtual private networks (VPNs) can also be used so that messages are encrypted and can't be traced back to specific computers. VPNs mitigate the effects of man-in-the-middle (MITM) attacks where crooks hijack data exchanges to monitor and modify content.

Bad actors pretend to be the sender to the recipient, and the recipient to the sender. But running browsers in virtual machines can limit damage in the event of a successful hack by an intruder.

The W3C released a web cryptographic API in 2017, complete with JavaScript examples. And other JavaScript security libraries include the Stanford JavaScript Cryptographic Library—SJCL—which is discussed here.

In the past, transmission security has been achieved with Secure Sockets Layer (SSL) software running on top of the TCP/IP protocol used to transport and route data over networks. Sensitive data was encrypted with the symmetric-key algorithm Data Encryption Standard (DES), but this "HTTPS" protocol using SSL over HTTP is no longer secure. SSL has been deprecated by the Internet Engineering Task Force [2] (IETF) and replaced with Transport Layer Security, TLS. By the late 1990s, DES was also no longer secure, so a competition was held by the National Institute of Standards and Technology (NIST) to find a replacement. As a result, the Rijndael symmetric-block cipher algorithm was chosen as the new Advanced Encryption Standard [3] (AES).

Protecting Data Transmission

Computer security includes using secure hardware-based logins with technologies like iris or fingerprint scanning. And multimode authentication is also used, where extra information is requested to confirm users' identities once a user name and password are successfully entered.This can include asking security questions, such as a user's mother's maiden name—but that particular piece of information is readily available and is **NOT** secure! Or a temporary PIN (personal identification number) may be sent to users via a phone call, text message, or email, and then the PIN is entered at the website.

Use of the HTTP Secure (HTTPS) protocol on the internet helps to protect data, but web pages called via HTTPS cease to be secure if they access regular HTTP pages. And operating systems and browsers must be properly configured to use the most secure transmission methods, with SSL disabled and TLS enabled.

This article focuses on encryption key exchanges rather than human factors. But don't write your password at work on a sticky note and paste it to your computer screen—you don't know who the office cleaners might be working for!

There are several requirements at the core of secure data transactions:

1. Confidentiality—Encrypt text to make it unreadable.
2. Integrity—Ensure text is tamper-proof: use a message authentication code (MAC).
3. Authentication and Non-repudiation—Content comes from trusted sources and the sender really sent the message by using a hashed MAC—HMAC.
4. Perfect Forward Secrecy—Avoid future compromise with temporary, ephemeral encryption keys.
5. Availability—Ensure timely and reliable access to client-side JavaScript code employing the SJCL library in web pages, such as a login page, may look something like the following pseudocode:

From sender - `sjcl.encrypt('secretSharedKey','plainTextToEncrypt')`
AND
To receiver - `sjcl.decrypt('secretSharedKey', 'cipherTextToDecrypt')`

Both senders and receivers must securely store the "secretShared Key",

which must not be the same as users' passwords. Zero-knowledge methods, such as Elliptical Curve Diffie-Hellman (ECDH), are used to exchange secret values, where the actual encryption key chosen from a point on an elliptical curve isn't transmitted between sender and receiver.

In the more general case, a TLS secure communications session between a sender and receiver, or client and server, might proceed as follows:

- Client opens session and checks server's identity, e.g., X.509 public key certificate.
- Server checks client's identity.
- Establish most secure mutually supported encryption suite from lists on server and client.
- Exchange asymmetrically encoded cryptographic set-up information encoded with server's public key, including set-up information for temporary, ephemeral symmetric encryption.
- Switch to symmetric encryption on completion of asymmetrically encrypted set-up information exchange, and then send the symmetrically encrypted message body.
- After message is sent, discard ephemeral keys and close session.
- Server checks integrity and decrypts message.

AES uses the Rijndael symmetric-block cipher algorithm with key lengths of 128, 192, or 256 bits, and fixed data block sizes of 128 bits for both input and output text blocks. Elliptical curve cryptography (ECC) uses points on elliptical curves to compute shorter, more secure keys than RSA,
with a 512-bit ECC key being as hard to crack as a 15,360-bit RSA key.

Diffie-Hellman Key Exchange

Let's go further and see what can offer us `typeid`. As you know, it's purpose is to return information about types, nothing less and nothing more.
Unfortunately, this operator isn't very *SFINAE-friendly* and it's not worth it to show an example, although one can perhaps build something ad hoc with it.

THE CHOICE TRICK

The Diffie-Hellman key exchange method was made significantly more secure by using elliptical curves instead of pseudo-random number generators. With Elliptical Curve Diffie-Hellman (ECDH), two parties each use different secret random numbers (private keys), x and y, and each party transmits the value of the public key (P) raised to the power of the private key, x or y. So the publicly revealed data is limited to the values of Px and Py.

The sender generates a random pre-master secret key, encrypts it with a receiver's public key, and then sends it to the receiver. The receiver uses their private key for decryption, and creates the shared master-secret.

The client-side set-up code for Diffie-Hellman follows:

```
// Most of this Javascript code was written by Tom Wu at Stanford U.

<script language="JavaScript" type="text/javascript" src="tomWu_files/jsbn0000.js">
// Above file contains Big Number math functions </script>
<script language="JavaScript" type="text/javascript" src="tomWu_files/jsbn2000.js">
// Above file contains extended Big Number math functions </script>
<script language="JavaScript" type="text/javascript" src="tomWu_files/prng4000.js">
// Above file contains Pseudo Random Number Generator </script>
<script language="JavaScript" type="text/javascript" src="tomWu_files/rng00000.js">
// Above file contains Random Number Generator.  Requires the prng file prng4000.js
</script>
<script language="JavaScript" type="text/javascript" src="tomWu_files/ec000000.js">
// Above file contains Javascript Elliptic Curve implementation </script>
<script language="JavaScript" type="text/javascript" src="tomWu_files/sec00000.js">
// Above file contains Elliprical Curves such as secp192r1 used below </script>

<script language="JavaScript">
<!--

var name;

function set_ec_params(name) {
  var c = getSECCurveByName(name);
  document.ecdhtest.q.value = c.getCurve().getQ().toString();
  document.ecdhtest.a.value = c.getCurve().getA().toBigInteger().toString();
  document.ecdhtest.b.value = c.getCurve().getB().toBigInteger().toString();
  document.ecdhtest.gx.value = c.getG().getX().toBigInteger().toString();
  document.ecdhtest.gy.value = c.getG().getY().toBigInteger().toString();
  document.ecdhtest.n.value = c.getN().toString();
  document.ecdhtest.alice_priv.value = "";
  document.ecdhtest.alice_pub_x.value = "";
  document.ecdhtest.alice_pub_y.value = "";
  document.ecdhtest.alice_key_x.value = "";
  document.ecdhtest.alice_key_y.value = "";
}

function set_secp192r1() {
  if (name == "") set_ec_params("secp192r1");
  else
 // OnLoad default curve. Code
  {
 // for 128, 160, 224, and 256
    name = "";
 // curves has been removed
    set_ec_params("secp192r1");
    rng = new SecureRandom();
    do_alice_rand();
  }
}
```

One party knows x and the value of Py and the other knows y and the value of Px, so both parties can calculate the shared key, P(xy), *as both (Py) x = P(xy)* and (Px)y = P(x*y). As long as x and y stay secret, eavesdroppers can only discover the transmitted values, Px and Py, but not the shared key.
If both x and y are ephemeral, the shared key is as well. In the next image, "Alice" is the sender and "twECDHserver.html" is the receiver's web page.

```
var rng;

function do_init(nameToUse) {
  if(document.ecdhtest.q.value.length == 0) set_secp192r1(); // set_secp192r1();
  rng = new SecureRandom();
  do_alice_rand();
}

function get_curve() {
  return new ECCurveFp(new BigInteger(document.ecdhtest.q.value),
    new BigInteger(document.ecdhtest.a.value),
    new BigInteger(document.ecdhtest.b.value));
}

function get_G(curve) {
  return new ECPointFp(curve,
    curve.fromBigInteger(new BigInteger(document.ecdhtest.gx.value)),
    curve.fromBigInteger(new BigInteger(document.ecdhtest.gy.value)));
}

function pick_rand() {
  var n = new BigInteger(document.ecdhtest.n.value);
  var n1 = n.subtract(BigInteger.ONE);
  var r = new BigInteger(n.bitLength(), rng);
  return r.mod(n1).add(BigInteger.ONE);
}

function do_alice_rand() {
  var r = pick_rand();
  document.ecdhtest.alice_priv.value = r.toString();
  document.ecdhtest.alice_pub_x.value = "";
  document.ecdhtest.alice_pub_y.value = "";
  document.ecdhtest.alice_key_x.value = "";
  document.ecdhtest.alice_key_y.value = "";
  do_alice_pub();
}

function do_alice_pub() {
  var before = new Date();
  var curve = get_curve();
  var G = get_G(curve);
  var a = new BigInteger(document.ecdhtest.alice_priv.value);
  var P = G.multiply(a);
  var after = new Date();
  alicePubX = P.getX().toBigInteger().toString();
  alicePubY = P.getY().toBigInteger().toString();
  document.ecdhtest.alice_pub_x.value = alicePubX;
  document.ecdhtest.alice_pub_y.value = alicePubY;
  document.getElementById("ecdhServer").src = "twECDHServer.html?X="+
                                              alicePubX+
                                             "&Y="+alicePubY+
                                             "&curve="+name;
}
```

Alice's values are sent to Bob, the receiver/server, in the last line of this code. The concatenated value is strengthened (stretched) by repeatedly hashing thousands of times with Password-Based Key Derivation Function 2 (PBKDF2) methods.

Since password strengthening slows calculations, a range of 500 to 2500

produces acceptable results with minimal delays. SJCL uses thousands of repeated hashes to make brute force attacks harder and the pattern used is : Hash(Salt | Hash(UserName | ":" | Password)), so the concatenated "Salt","UserName", ":", and "Password" might look something like this: "9+Va5voDXA=JohnSmith:k473#bGu!Oo6*27\Y1n0a9c~".

The colon (:) between User Name and Password makes each pair unique, and avoids hashing user name "timSmith" and password "oldman", the same as "timSmit" and "holdman". With no ":", the string for both is timSmitholdman, vs timeSmith:oldman and timSmit:holdman with a ":".

Two parties using Diffie-Hellman agree on a large prime number, "N", and a smaller number "g" that's primitive with respect to N. g being primitive to N means that integers, "k", can be found where (k = gi mod N) for all the values of i from 1 to N-1. So g is primitive to N if all of the individual values of k are also equal to all the integers from 1 to N-1.

Hash functions are one-way, so it's extremely difficult to derive the original string from the hash. To ensure validity that it hasn't been tampered with, and that it comes from a trusted source, a known and expected hash value for encrypted text is compared to a newly computed hash value.

The English language has about 100,000 words, so to brute force crack a hashed password up to 100,000 attempts might be needed. By adding a 32-bit salt to the password, the number of possible passwords are:

```
Password Possibilities           = 100,000
Salt Possibilities               = 232 ( 32-bit )
Hash calculations required = Password Possibilities * Salt Possibilities
Hash calculations required = 100,000 * 232
Hash calculations required = 429,496,729,600,000
```

Seeded/salted strings are strengthened in this way to resist attacks that use precomputed "Rainbow" tables to crack cryptographic hash functions.

With CBC (cipher block chaining) encryption, whole blocks of text of the specified length are encoded, and each encoded text block depends on the previous source blocks. If needed the last block is padded to make it the required fixed block length, and as there is no data before the first block, messages are encoded with a unique, random initialization vector (IV).

Note that compression must NOT be used. It makes text vulnerable to hacking by replacing repeating character strings with smaller tokens.

In ECDH, elliptical curve points provide the public and private encryption keys, and predefined values include N and g for an agreed-to curve.

In ECDH, elliptical curve points provide the public and private encryption keys, and predefined values include N and g for an agreed-to curve. Both parties create private and public keys, exchange public keys, and multiply their private key by the other's public key to derive the shared secret key—see the following code:

```
function get_bob_pub()
{
  var ifrm = document.getElementById("ecdhServer");
  document.ecdhtest.bob_pub_x.value = ifrm.contentWindow.bobPubX;
  document.ecdhtest.bob_pub_y.value = ifrm.contentWindow.bobPubY;
  var before = new Date();
  var curve = get_curve();
  var alicePubX = document.ecdhtest.alice_pub_x.value;
  var alicePubY = document.ecdhtest.alice_pub_y.value;
  var P = new ECPointFp(curve,
    curve.fromBigInteger(new BigInteger(document.ecdhtest.bob_pub_x.value)),
    curve.fromBigInteger(new BigInteger(document.ecdhtest.bob_pub_y.value)));
  var a = new BigInteger(document.ecdhtest.alice_priv.value);
  var S = P.multiply(a);
  var after = new Date();
  document.ecdhtest.alice_key_x.value = S.getX().toBigInteger().toString();
  document.ecdhtest.alice_key_y.value = S.getY().toBigInteger().toString();
  if (name == "") name = "secp192r1";
}
//-->
</script>
  </head>
  <body onload=«do_init();">
```

The foregoing code gets Bob's values and computes Alice's shared secret key. With compact representation, the x-coordinate of the derived point on the curve is the shared secret, and authentication data ("adata") can also be added. The encoded text block returned might look like this:

```
{
  "iv":"HV9Xro6SvCgq17ReXAsn4Q==",
  "v":1,
  "iter":1000,
  "ks":256,
  "ts":128,
  "mode":"ccm",
  "adata":"authorization data",
  "cipher":"aes",
  "salt":"ObpiMz7YLFw=",
  "ct":"p5/DJ8yRQPHOrnEUu6d4o1Q9JTKSNzrWgy8KO6MGIDy"
}
```

The initialization Vector is "iv", version number is "v", "iter" is a password-strengthening factor, key size in bits is "ks", authentication tag size in bits is "ts", "mode" is the cipher mode, "adata" is the authentication data string—can be a timestamp to limit multiple login tries. The algorithm used for encryption is "cipher", "salt" is the password Salt, and "ct" is the encrypted ciphertext that's sent between the two parties exchanging messages.

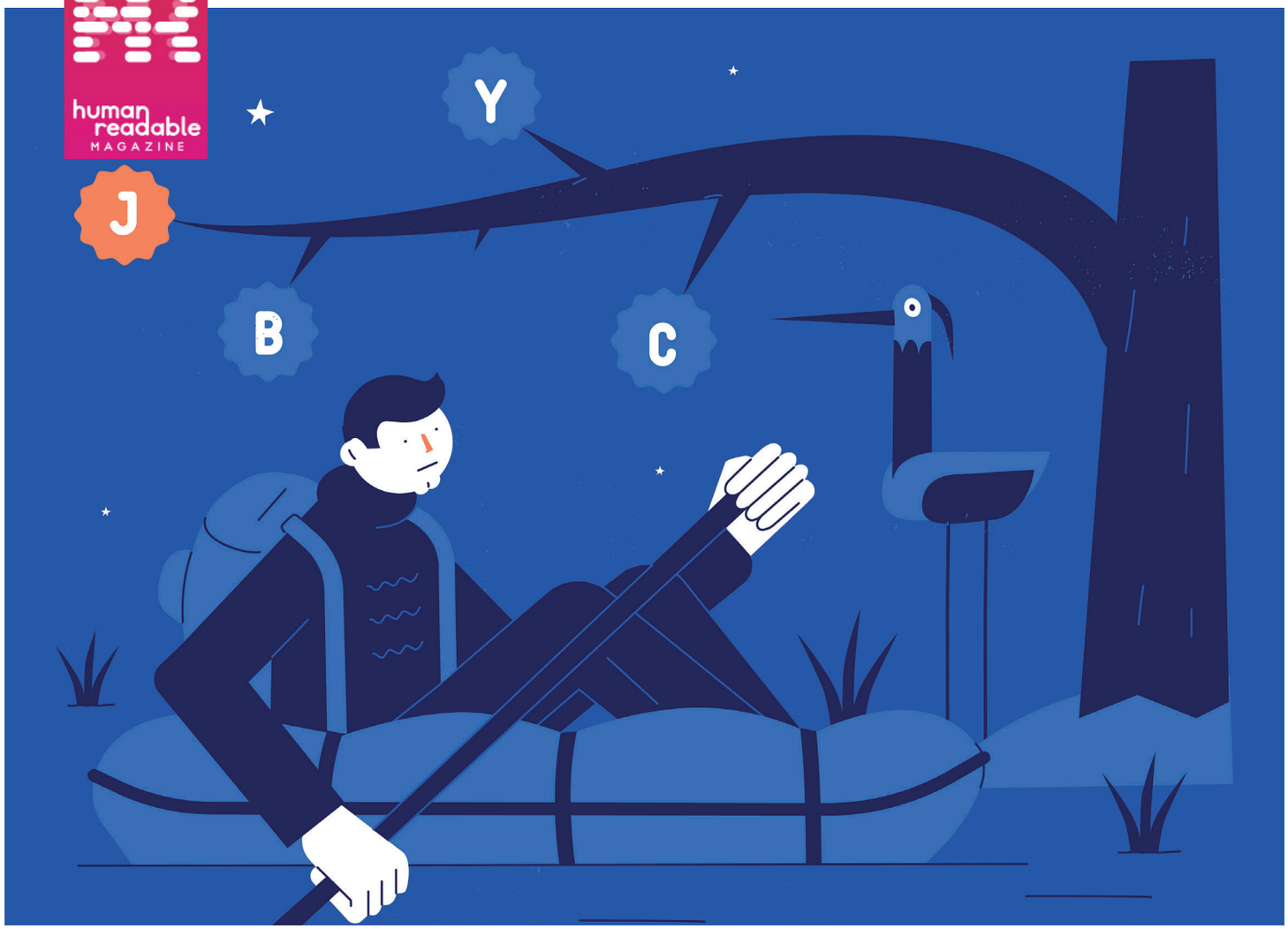
CONCLUSION

Using Elliptical Curve Diffie-Hellman (ECDH) to exchange encryption keys provides greatly improved key security. Combined with 256-bit AES encryption [4] of message content, much greater security is achieved than was possible in the past [5].
At http://web2ria.com/#95 [6], and http://bitwiseshiftleft.github.io/sjcl/demo/ [7] are the two ECDH examples (key exchange and the full encryption demo). See https://www.garykessler.net/library/crypto.html#dhmath [8] for a longer article. The example pages at Web2RIA and Github use client-side data exchanges, but both PHP and C# server-side languages have Diffie-Hellman functions, and AJAX can be used for browser-server data exchanges. ✦ hello@humanreadablemag.com

ARTICLE LINKS

1. https://hrm.link/symmetricencryption
2. https://hrm.link/IETF
3. https://hrm.link/AES
4. https://hrm.link/AES-encryption
5. https://hrm.link/greater-security
6. https://hrm.link/HumanReadableMa
7. https://hrm.link/HumanReadableMa
8. https://hrm.link/HumanReadableMa

human readable MAGAZINE



ALGORITHMS

The Wonders of the Suffix Tree through the Lens of Ukkonen's Algorithm

Article by **Eze Onukwube**
Main illustration by **Sergey Konotopcev**

THE USE OF THE RIGHT DATA STRUCTURE THAT IS ADEQUATELY SUITED FOR THE COMPLEXITY OF A PROBLEM IS ONE OF THE ATTRIBUTES THAT DISTINGUISHES EXPERIENCED DEVELOPERS FROM THE REST.

After several years of plying my trade as a software engineer, I assumed I had at least encountered most of the consequential data structures in the field. Unfortunately, important fundamental techniques do not always make it into the mainstream of computer science education.

Suffix trees, primarily used for indexing and searching strings, occupy a central position in text compression, text algorithmics, and applications in the realm of biological sequence comparison, and motif discovery. This data structure had never been on my radar until fairly recently when a job interview confronted me with its existence. Several culprits are to blame for its relative anonymity.

Introduced to the world in 1973 by Peter Weiner, suffix trees are a relatively new concept in computer science, with this newness likely fueling its obscurity. In addition, a suffix tree first needs to be built, and its construction algorithms are nontrivial. At the time of this writing, there aren't any Java or other language libraries that provide the necessary functions. This degree of difficulty in its implementation presumably limits its widespread usage. Moreover, because even the image of a suffix tree looks complicated at first sight, some people might be intimidated and discouraged from learning it. Furthermore, it is one of those few data structures developed to solve a super specific problem, which somewhat adds to the sheen of the suffix tree being in a class of its own.

However, regardless of the underlying reasons, suffix trees are perhaps the most important data structures for string processing but have flown under the radar for far too long. To the extent it can, this article hopes to remedy this problem while attempting to demystify the suffix tree's inner workings. This juggling act is a tall order, but I am hoping to validate the cliché that it is better to shoot for the moon and miss in the hope of landing, at the very least, among the stars.

The Problem, the Suffix Tree, and Ukkonen's Algorithm

A suffix tree is a compressed trie of all the suffixes of a given text as keys, with their positions within the text as values. The trie will be elaborated upon in a subsequent section, but in the meantime, in order to grasp an intuitive understanding of the suffix tree, it is better to view it from the perspective of the problem it was intended to address. The fountainhead, the patient zero, if you will, for the compelling suffix tree remedy is the substring problem. This challenge, expressed in various forms in computer science literature, boils down to finding the means to preliminary process a text in such a way that this computational string matching problem is solved in time proportional to the length of the pattern. The problem has many more applications than its face value would suggest. The most compelling is performing intensive queries on a large database, represented by T. The prototypical problem for a suffix tree can be expressed more technically as follows: Given a particular string T, we need to construct an efficient data structure S, such that S will serve as an index of T, enabling us to efficiently query text T for all the occurrences of a query pattern P. Therefore, the foremost benefit of a suffix tree is to enhance pattern matching on an indexed string. The challenge, however, is to index the text T in such a way that given a query pattern P, all the occurrences of P in T can be reported efficiently. In case you haven't already inferred it, a suffix tree is the data structure S that can meet this requirement.

Suffix trees are used for preprocessing strings, which is simply another fancy word for indexing a string in this context. The ensuing structure it generates subsequently makes it easier to search for patterns within the string. In other words, it allows us to preprocess the string, only once, in anticipation of future, unknown queries. Such future inquiries could be to determine whether P is a substring of T, how many times P appears in T, the longest repeated string in T, the longest common substring of P and T, just to mention a few of the enticing possibilities.

However, the suffix tree is not a pattern-matching algorithm. Such distinction is reserved for algorithms such as Knuth-Morris-Pratt (KMP), Boyer-Moore, and Rabin Karp, among others. As a result, this article will fixate on the construction of the suffix tree; any mention of how pattern searching can be implemented through it is regarded as an ancillary benefit.

One of the true geniuses of the suffix tree is how it allows different kinds of queries to be answered quickly, permitting its use as a particularly fast implementation of many important string operations. But the hardest part is constructing the tree. If you use the naïve algorithm, constructing a suffix tree for a string of size n takes $O(n^2)$ time, which is clearly prohibitive. However, by virtue of the fact that all the entries of its tree are suffixes of the same string, they share a lot of information that can be exploited for efficiency. Consequently, algorithms that take advantage of this commonality allow us to create the suffix tree more proficiently. For instance, by using Ukkonen's algorithm, a suffix tree can be built in linear time and space. Ukkonen's algorithm stands out because most algorithms that either construct an index or query for a pattern lack the generous running time and memory savings that it does.

To paint the Ukkonen's algorithm savings in asymptotic terms, assuming n is the length of T and m is the length of P, then the construction time and space complexity of the suffix tree are both $O(n)$, which allows the given query string (pattern) P of length m to be queried in $O(m)$ time. This is in sharp contrast to other algorithms, which may take up to $O(n+m)$ time, rendering them prohibitively inefficient to use, especially with large databases. This means that someone could determine whether the term "banana" was in the entire collection of Encyclopedia Britannica just by performing seven character comparisons!

Ukkonen's algorithm has several important attributes and innovations that enable it to deliver these linear time and space advantages. These are simple features, yet they exert a disproportionate impact on its overall performance. First, it is online, which simply means it processes the string symbols one character at a time, from left to right. This allows the index to always have the tree for the scanned part of string already ready. Imagine the algorithm is about to add a character at say, position *4* in the string. This means it already has the tree T^3 built from suffixes *0* to *3*. This capability in the tree-building phase opens the door to another remarkable feature known as the suffix link.

We will see how the suffix link operates later on, but at this point, it suffices to say that it enables the build process to shorten its migration path when required to move from one branch/node to another. This is necessary because Ukkonen's algorithm computes its suffix tree from another data structure, albeit a similar one, known as an implicit suffix tree. Implicit suffix trees do not deliver linear complexity when computed in a straightforward manner, hence the need for suffix links to speed up tree traversal. In addition, these suffix links are convenient for pattern matching after the tree has been constructed. To wring space savings from the suffix tree, instead of storing single characters in its edge labels, Ukkonen labeled them as a pair of integer pointers to the text. This edge uses *O(1)* space, even though it carries a string of arbitrary length. As a result, this edge-label compression leads to memory space reduction.

Another interesting feature of the suffix tree is the manner in which it exposes the internal structure of a string and, by so doing, provides an ease of access into it. This thread of thought is a good way to segue into unveiling the properties of the suffix tree:

```
A suffix tree of string $S$ and length $n$ is defined by the following
 characteristics:
          * Its leaf nodes will correspond exactly to its length $n$, numbered
          $1$ to $n$.
          * Apart from its root, every other internal node has at least two
          children.
          * Every edge is labeled with a non-empty substring of $S$.
          * Two edges with a common node cannot have string-labels starting
          with the same character.
          * The resultant string obtained by concatenating all the labels found
          on the path from the root to leaf *i* spells out suffix S[i..m], for
          i=1,…, s.
```

Understanding the Suffix Tree from the Perspective of a Trie

A suffix tree is built on top of a trie, so it is necessary to have some rudimentary understanding of the trie. In an overly simplified manner, the prefix and suffix distinction can serve as a mnemonic device to differentiate between a trie and a suffix tree, which is to remember that the latter deals with suffixes while the former, prefixes. Obviously, their overall differences aren't that simplistic.

A trie is simply a tree for storing (a set of) strings *S*, in which there is one node for every common prefix. The path from the root of the trie to any of its nodes corresponds to a prefix of at least one string of *S*.

In addition, the strings that are in the set *S* are stored in extra leaf nodes. A suffix tree, on the other hand, is an improvement and a more compact representation of the trie because not every trie node is explicitly represented in the tree. It corresponds to the suffixes of a given string where all nodes with one child are merged with their parent. So instead of just adding the text itself into the trie, every possible suffix of that text is added.

The trie's edge labels are in actual characters, unlike the suffix tree (Ukkonen's implementation at least), which uses numeric pointers to conserve space. Since the suffix tree removes the unnecessary branches of the trie, it consequently doesn't require as much space. As a result, it is lighter than the trie and has innovations such as suffix links that make it equally faster.

The lightness and speed advantage over the trie enables the suffix tree to be used to index DNA or optimize some large web search engines. Their differences also inform the type of use cases in which they are utilized and the questions demanded of them in queries, even where some large text is given.

Since tries store a set of strings, they can be used to search for predefined words in a text, like ensuring users do not post derogatory words by building a trie containing those forbidden, predefined words. Conversely, a suffix tree can be built to search for any words in the text. Most of us who have used the CTRL+F search feature in any text editor will appreciate this benefit function. For this hypothetical large text, once it is changed or modified in any form, you'll also be compelled to rebuild both the trie and suffix tree. Doing so for the trie is a relative trivial operation. But both the building and rebuilding of a suffix tree for a large amount of text is a complex operation.

A Suffix Tree for BANANA$

We will use BANANA$ as the text to understand how suffix trees are built. In order to avoid an implicit suffix tree and ensure suffixes end at leaves, we add $ char at the end of the string. The termination character can be any that isn't in the alphabet the string is taken from. Following are all the possible suffixes of BANANA$:

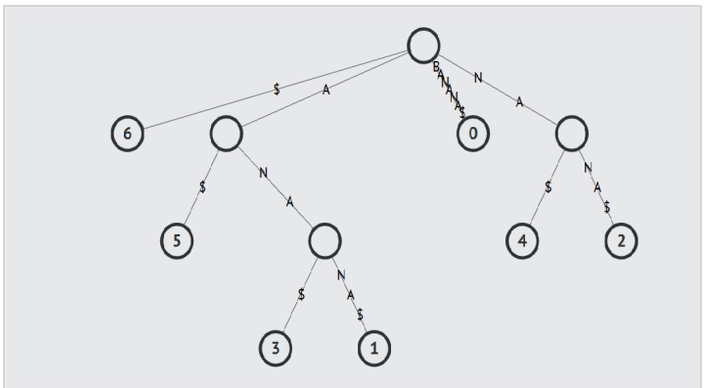


Fig. 1: Suffix Tree for BANANA$

Starting from the root node, each of the suffixes of BANANA$ can be found in the tree (BANANA$(0), ANANA$(1), NANA$(2) ...) and finishing up with a solitary $ at index 6. Because of this organization, you can search for any substring of the word by starting at the root and following matches down the tree until exhausted.

The key feature of the suffix tree is that for any leaf *i*, the concatenation of the edge-labels on the path from the root to leaf *i* will spell out the suffix of *S*, which starts at the position *i*.

Suffix Tree and Its Construction

The general approach to building a suffix tree is twofold:

1. Generate all the suffixes for that text.
2. Take all the suffixes as individual words and build a compressed trie with them.

Components of Suffix Tree

Being a tree data structure, a suffix tree is comprised of a root and internal and leaf nodes. This article only elaborates on those components and areas that have particular significance in the operations of a suffix tree.

Edges: A suffix tree of a string *S* represents all the suffixes of the string. Each edge of the suffix tree is labeled with a substring *y* of *S*, and the edge string *y* is represented by a pair *(i,j)* of positions such that the substring of *S* that begins at position *i* and ends at position *j* is identical to *y*.
Therefore, the pair *(i,j)* are essentially pointers to the text, enabling the edges to carry string labels of arbitrary length. This ensures a suffix tree can be represented with linear space since the pointers take only *O(1)* space. If the edge labels were stored as strings, space complexity would be $O(N^2)$, regardless of the number of nodes. Using two variables *(i,j)* as edge labels instead of strings takes constant space, making overall space complexity *O(1)*.

Internal Nodes: They have more than one outgoing edge and mark the parts of the tree where branching occurs. Branching only occurs whenever a repeating string is involved.

Suffix Link: In order to achieve linear time both in its construction algorithms and the many applications that utilize it, the internal nodes of suffix trees are equipped with a suffix link. The suffix link is an important piece of acceleration. Suffix links are auxiliary edges, which occur whenever two nodes share the same substring except for the first character.

Assume there is a string *S* in a tree, and its path from the root ends in a node *x*. If another string &S also present in the tree (where & is any character), and its path from the root ends in node y, then the link from x to y is known as a suffix link. Therefore, every internal node x that has a label from the root to x of more than one character must have a suffix link to exactly one other internal node.

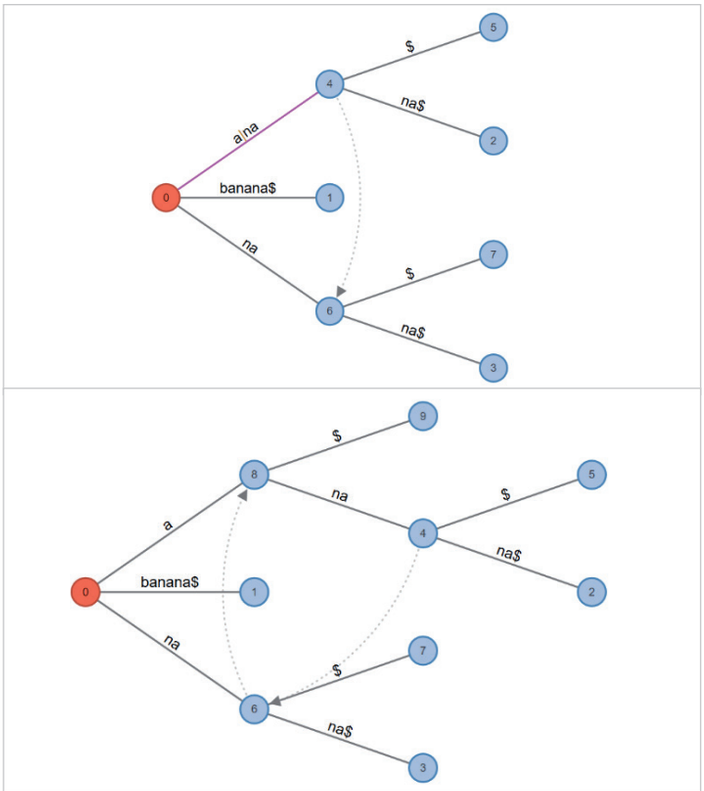


Fig. 2: Suffix Links for BANANA$

The dotted lines in Fig 2 show the suffix links from *ana* to *na*, and *na* to *a*. Suffix tree constructions based on suffix links have gained popularity because they are simple, easy to implement, can operate online in linear time, and are conveniently suited for pattern matching.
Suffix links are not part of the suffix tree definition and therefore not required. However, they assist in a surprising number of ways. For instance, after inserting the final character of a suffix S at a node x, the algorithm will need to insert the final character of suffix S-1 in *O(1)*time. To accomplish this, it uses the suffix link to jump right to *x– 1* to make the insertion. This reduces the number of branch lookup operations.

Active Point: This is the point where traversal begins in any extension. It is the location identified by a point on an edge in the tree and comprises the active node, active edge, and active length. Each extension will get the active point set correctly by the previous extension. This location is found by specifying a node, an edge from that node (by identifying the first character), and a length along that edge.

Remainder: It signals the number of suffixes that must be explicitly added. In our BANANA example, if the remainder is 3, then the last three suffixes (ANA, NA, A) must be processed.

Global End: When a node is added, it automatically becomes a leaf node. Therefore, the edge to the leaf node will have the actual end of the inserted string. As a result, the algorithm assigns a global end to the edge, and this needs incrementing each time the next character is processed, making these edges grow longer automatically.

Implementation Details

Ukkonen's algorithm constructs from a given string of text T, an implicit suffix tree Ti for each prefix S[l ..i] of S. It starts with an empty tree, then progressively adds each of the n prefixes (where length of input text is n) of T to the suffix tree.

Very High-Level Overview of Ukkonen's Algorithm

Generate I11. Keep generating Ii+1 from Ii1. At the last stage when the terminal symbol $ is added, the implicit suffix tree will automatically be converted to a true suffix tree.

High-Level Overview of Ukkonen's Algorithm (Pseudocode)

Finally, the true suffix tree for S is built from Tm by adding $.

```
Build Tree
1. For i from 1 to m – 1 do
/* begin {phase i + 1} */
     2. For j from 1 to i + 1
     /* begin {extension j} */
     3. Locate the end of the path from the root labeled S[j..i] in the current
    tree.
     If needed, extend that path by adding character S(i + l),
     therefore ensuring that string S[j..i + 1] is in the tree.
    /* end {extension j} */
/* end {phase i + 1} */
```

Phases and Extensions Decoded

Ukkonen's method utilizes an online algorithm, so all characters are processed singularly in sequence (as evidenced by the outer loop iteration), and time taken to build the suffix tree is *O(m)*.

```
1. For i from 1 to m – 1 do
```

This is the tree-building phase, where tree Ti+1 is built from tree Ti. The algorithm builds from left to right, starting with the first character: T_1 using the first character, T_2 using the second character, T_3 using the third character, ... , T_m using the mth character.
Every phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of *S[1..i+1]*.

Suffix Extension

Each phase deals in sequence with one character from the string, subsequently performing "extensions" within that "phase" to add suffixes that begin with the character.
This is helpful because of the repeating substructure of suffix trees, whereby each subtree can appear again as part of a smaller suffix.

```
2. For j from 1 to i + 1
```

In the inner loop, an attempt is made to locate the end of the path *S[j..i]* from the root. A suffix extension is performed based on the result of the previous step by adding the character *S(i+1)* to its end (if it is not there already).

So in the case of BANANA$, for phase 4 which is building on tree T_3 (suffixes: BAN, AN, NA), the extensions part of the algorithm *S[1..4]* would subsequently yield these suffixes: BANA, ANA, NA, and A because of adding S(4), which is the character A.

```
3. Locate the end of the path from the root labeled $S[j..i]$ in the current tree.
```

While suffix extensions are based on adding the next character to the suffix tree built so far, there are certain rules governing these extensions:

Rule 1

This rule can be summarized as follows: Assuming the path from the root labeled *S[j..i]* ends at a leaf edge (*S[i]* is the last character on leaf edge) then character *S[i+1]* is just added to the end of the label on that leaf edge. In extension j of phase i+1, the algorithm first finds the end of the path from the root labeled with substring *S[j..<(i+1)]*, which is depicted in line number 3 of our pseudocode. It then extends the substring by adding the character S(i+1) to its end (if it is not there already). For example, in extension 1 of phase i+1, we put string S[1..i+1] in the tree. Note that S[1..i] will already be present in the tree due to previous phase *i*, so the algorithm just needs to add S[i+1]th character in tree, assuming it isn't already there.

Rule 2

Assuming the path from the root labeled S[j..i] ends at a non-leaf edge (character(s) exist after S[i] on the path) and the next character is not s[i+1], then a new leaf edge with the label s[i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3

Assuming the path from the root labeled S[j..i] ends at a non-leaf edge (characters exist after S[i] on the path) and next character s[i+1] is already in the tree, do nothing.

Illustration of Suffix Tree Construction Using Ukkonen's Algorithm

Phase 1. This will read the first character from the string, will go through 1 extension.
Active Point=>**b**: Look from root. Add to root.
Rule 2. Extension 1 will add the suffix "b" into the tree. Creates a leaf edge **b**. Phase 1 completes here.

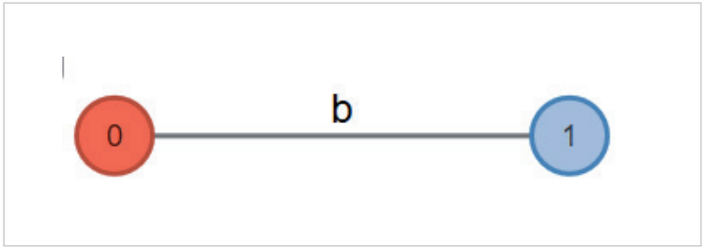


Fig. 3: Active Node: 0, Active Edge: Null, Active Length: 0, Remainder: 0

Phase 2. This will read the second character; will go through at least one and at most two extensions.
Active Point=>**a**: Look from root. Add to root.

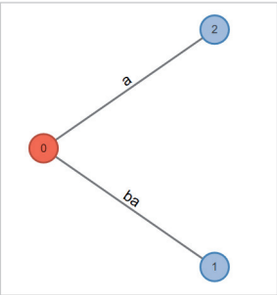


Fig. 4: Active Node: 0, Active Edge: Null, Active Length: 0, Remainder: 0

Rule 1. Extension 1 adds the suffix "ba" into the tree. Extends leaf edge from **b** to **ba**. Phase 2 extension 1 completes.
Rule 2. Extension 2 adds the suffix "a" into the tree. Creates a leaf edge a. Phase 2 extension 2 completes here.

Phase 3. This will read the third character and will go through at least one and at most three extensions.
Active Point=>**n**: Look from root. Add to root.

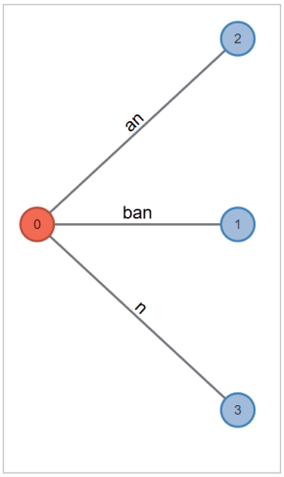


Fig. 5: Active Node: 0, Active Edge: Null, Active Length: 0, Remainder: 0

Rule 1. Extension 1 adds suffix "ban" into the tree. Extends leaf edge from **ba** to **ban**. Phase 3 extension 1 completes.
Rule 1. Extension 2 adds suffix "an" into the tree. Extends leaf edge from **a** to **an**. Phase 3 extension 2 completes.
Rule 2. Extension 3 adds suffix "n" into the tree. But there is no path from root, going out with label 'n', so create a leaf edge **n**. Phase 3 extension 3 completes here.

Phase 4. This will read the fourth character and will go through at least one and at most four extensions. Active Point=>**a**: Look from root. Find it ("ana"). Set it as the active point.

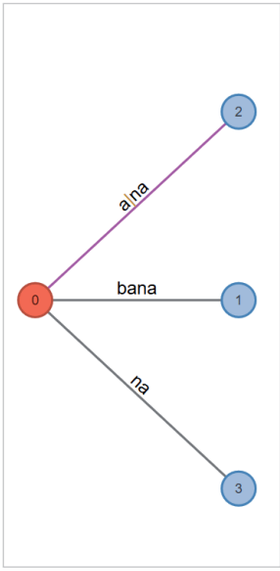


Fig. 6: Active Node: 0, Active Edge: a, Active Length: 1, Remainder: 1

Rule 1. Extension 1 adds the suffix "bana" into the tree. Extends leaf edge from **ban** to **bana**. Phase 4 extension 1 completes.
Rule 1. Extension 2 adds the suffix "ana" into the tree. Extends leaf edge from **an** to **ana**. Phase 4 extension 2 completes.
Rule 2. Extension 3 adds the suffix "na" into the tree. Extends leaf edge from **n** to **na**. Phase 4 extension 3 completes.
Rule 3. Extension 4, adds the suffix "a" into the tree, but path for label 'a' already exists in the tree. Therefore, no more work needed and phase 4 ends here.

Phase 5. This will read the fifth character and will go through at least 1 and at most 5 extensions. Active Point=>**n**: Look from previously found "a", in "_a_na". Find it. Set it as the active point.

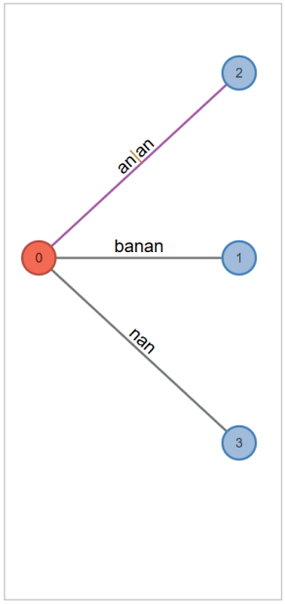


Fig. 7: Active Node: 0, Active Edge: a, Active Length: 2, Remainder: 2

Rule 1. Extension 1 adds the suffix "banan" into the tree. Extends leaf edge from **bana** to **banan**. Phase 5 extension 1 completes.
Rule 1. Extension 2 adds the suffix "anan" into the tree. Extends leaf edge from **ana** to **anan**. Phase 5 extension 2 completes.
Rule 2. Extension 3 adds the suffix "nan" into tree. Extends leaf edge from **na** to **nan**. Phase 5 extension 3 completes.
Rule 3. Extension 4, adds the suffix "an" into the tree but path for label 'an' already exists in the tree. Therefore, no more work needed and phase 5 ends here.

Phase 6. This will read the sixth character and will go through at least one and at most six extensions.
Active Point=>**a**: Look from previously found "n" in "a_n_an". Find it. Set it as the active point.

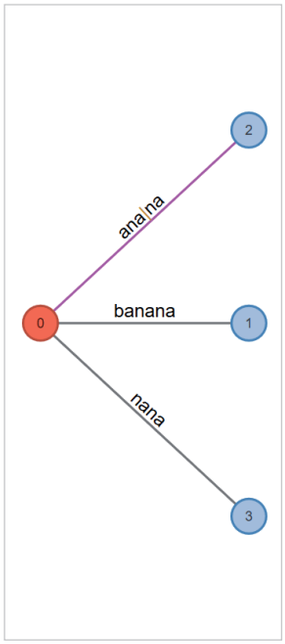


Fig. 8: Active Node: 0, Active Edge: a, Active Length: 3, Remainder: 3

Rule 1. Extension 1 adds the suffix "banana" into the tree. Extends leaf edge from **banan** to **banana**. Phase 6 extension 1 completes.
Rule 1. Extension 2 adds the suffix "anana" into the tree. Extends leaf edge from **anan** to **anana**. Phase 6 extension 2 completes.
Rule 2. Extension 3 adds the suffix "nana" into the tree. Extends leaf edge from **nan** to **nana**. Phase 6 extension 3 completes.
Rule 3. Extension 4, adds the suffix "ana" into the tree, but path for label 'ana' already exists in the tree. Therefore, no more work needed and phase 6 ends here.

Phase 7. This will read the seventh character and will go through at least one and at most seven extensions. Active Point=>**$**: Look from previously found "a" in "an_a_na". This is not found, so add an internal node and add $. Then change the active point to the same character in a smaller (suffix) edge: "a" in "n_ana".

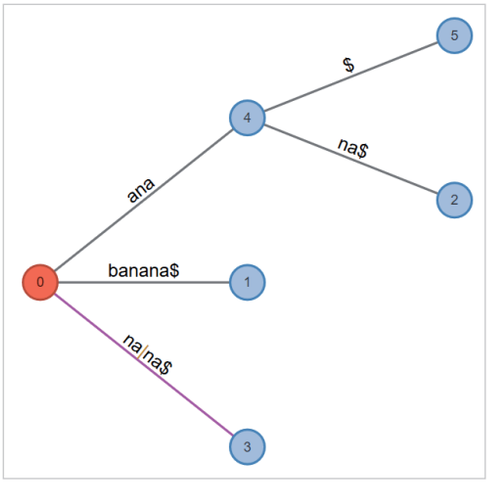


Fig. 9:
Active Node: 0,
Active Edge: n,
Active Length: 2,
Remainder: 2

Rule 1. Extension 1 adds the suffix "banana$" into the tree. Extends leaf edge from **banana** to **banana$**. Phase 7 extension 1 completes.
Rule 1. Extension 2 adds the suffix "anana$" into the tree. Extends leaf edge from **anana** to **anana$**. Phase 7 extension 2 completes.
Rule 1. Extension 3 adds the suffix "nana$" into the tree. Extends leaf edge from **nana** to **nana$**. Phase 7 extension 3 completes.
Rule 2. Extension 4 adds the suffix "ana$" into the tree, but path for label 'ana$' already exists in the tree on the edge now labeled "anana$". Therefore, the algorithm splits that edge after the matching portion "ana" by adding a new internal node which contains the unmatched portion "na$" and an edge labeled with $ as a leaf node.
This represents quite the leap from what has been previously done, so let's summarize: There's an edge from the root that starts with "ana" ending in the middle of an edge, so we perform a split adding an internal and leaf node. Active Point=>**$**: Look from previously found "a" in "n_a_na$". This is not found, so add an internal node and add the $ as leaf node. Then change the active point to same character in a smaller (suffix) edge: "a" in "_a_na$".

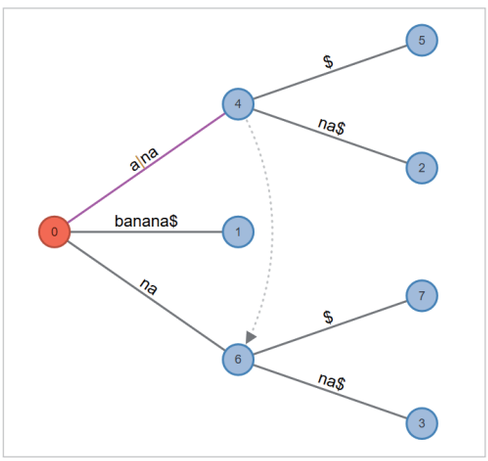


Fig. 10:
Active Node: 0,
Active Edge: a,
Active Length: 1,
Remainder: 1

Rule 2. Extension 5 adds suffix "na$" in tree but path for label 'na$' already exists in the tree on edge now labeled "nana$". Just as done previously, that edge is split after the matching portion "na" is given a new internal node which has an edge emanating from it that contains the unmatched portion "na$", along with a $ pointing to a leaf node.
Active Point=>**$**: Look from "a" in "_a_na$". This is not found, so add an internal node and add the $. Then change the active point to root.

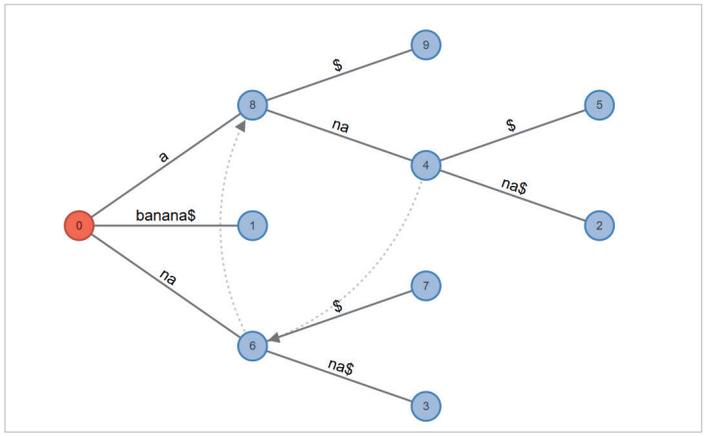


Fig. 11: Active Node: 0, Active Edge: none, Active Length: 0, Remainder: 0

Rule 2. Extension 6 adds the suffix "a$" on the path for label "ana". The edge is split after the matching portion "ana" is given a new internal node which has an edge emanating from it that contains the unmatched portion "na", along with another edge $ pointing to a leaf node.
Active Point=>**$**: Look from the root. Add to the root.

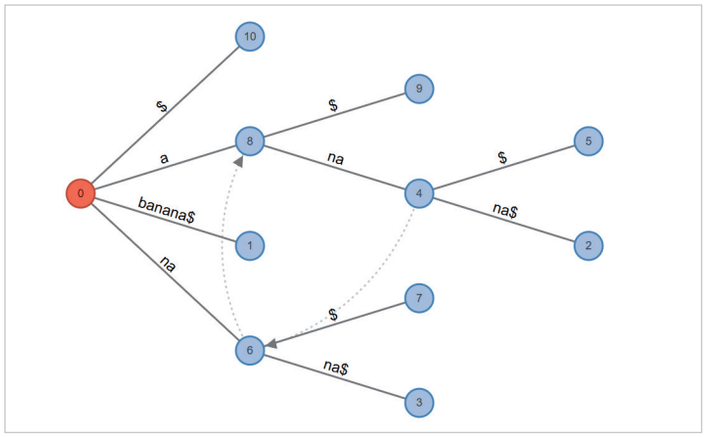


Fig. 12: Active Node: 0, Active Edge: none, Active Length: 0, Remainder: 0

Rule 2. Extension 6 adds edge labeled "$" to the root.
Our suffix tree for BANANA$ is completed.

Pattern Searching with Suffix Tree

Constructing the suffix tree is a means to an end. The purpose of the structure is to be able to make inquiries that would yield insights into a body of text (especially a large one) that would have otherwise been quite onerous to obtain.
While the scope of this article is limited to building a suffix tree, pursuing pattern searching with the structure created is not as big of a leap as one would assume. The architecture of the suffix tree provides a blueprint from which it is easy to pursue answers. For instance, the leaves below a node store the suffix indices that start with the prefix defined at that node. Because of this, you just need to count the descendant leaves after you find the node that matches your search pattern in the suffix tree. ✦ hello@humanreadablemag.com

ARTIFICIAL INTELLIGENCE

What if Algorithms Could be Fair?

Article by **Andy Kitchen, Laura Summers**
Illustration by **Leandro Lassmar**

THAT'S NOT FAIR! A SPIKE IN OUR GUT, A FLARE OF ANGER, THE WEIGHT OF RESENTMENT. FAIRNESS IS IN OUR NATURE. HUMANS ARE DEEPLY SOCIALIZED WITH A MORAL INTUITION FOR FAIRNESS; COMPUTERS ARE NOT. BUT CAN COMPUTERS BE PROGRAMMED WITH A FUNCTIONAL SUBSTITUTE FOR FAIRNESS? THIS QUESTION IS URGENT TODAY AS MORE AND MORE DECISIONS ARE MADE BY STATISTICAL ALGORITHMS.

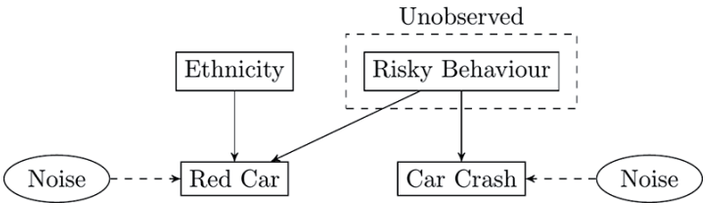
But how can we make an algorithm fair without moral intuition? Many approaches have been proposed. Unfortunately they are almost all heuristic, meaning they provide a rule of thumb that is open to interpretation and lacks a coherent underlying theory or framework. A new approach proposed recently by a group of researchers from The Alan Turing Institute called counterfactual fairness [1] provides a more principled approach to making algorithms fair and resolves many of the shortcomings of past approaches.

Counterfactual fairness poses the question, "If a protected personal attribute was hypothetically changed, does the system's decision change?" Consider the example of a woman applying to university. We could set gender as the "protected attribute," that is, an attribute that we believe is a source of unfair bias. We'd ask "If they were a man, would they have been accepted into this university?" When the answer is different in the real case and the hypothetical case, we say the decision was counterfactually unfair.

Importantly, the counterfactual fairness approach is much richer than simply hiding or ignoring the protected attribute. Consider our woman applying to university. In both the real case and the hypothetical, the candidate has the same "X-factor": intelligence, commitment, determination, and hustle. However her opportunities and circumstances are different. In the hypothetical scenario we attempt to model the effects of those differences across a lifetime, not just at the moment of the application. If our person with the same innate ability had experienced their lifetime in the switched gender, in this case as a male, would they have been accepted? This takes into account the downstream effects that would have been caused by the hypothetical change in the protected attribute.

How can you quantify the effects of a counterfactual when hypothetically changing the protected attribute? By using the relatively new field of causal modeling, pioneered by Judea Pearl and others. Causal modeling extends the classical observational tools of statistics (e.g., there is a correlation between smoking, stained teeth, and cancer) to reasoning about causes and intervention (e.g., stopping smoking will reduce cancer risk, but whitening your teeth will not). Causal modeling, or simply causality, provides the tools necessary to reason about what could cause what, design the experiments necessary to empirically test proposed causal relationships, and importantly, quantify the effect of an intervention or counterfactual scenario.

Consider the following causal diagram for an illustrative "red car" example:



Causal Diagram

In the diagram above the boxes are attributes, and the arrows indicate causality, that is, that one attribute causes another. In this example there are four attributes: "Ethnicity," "Red Car," "Risky Behavior," and "Car Crash."

We can observe "Ethnicity" and "Red Car" but cannot directly observe "Risky Behavior." "Car Crash" is our prediction target and is only available in historical training data. The protected attribute is "Ethnicity." Our goal is to predict future car accidents.

Following the arrows we can see that "Risky Behavior" is a direct cause of "Car Crash" and also of "Red Car": we believe risky drivers are more likely to buy red cars. The arrow from "Ethnicity" to "Red Car" indicates that that car color can also be caused by belonging to a red-car-preferring ethnic group. The direction of the arrows matter: if I paint someone's car red it clearly does not affect their ethnicity.

The arrows that aren't present are just as important as the arrows that are. There is no arrow from "Ethnicity" to "Risky Behavior." In this causal model ethnicity does not cause risky behavior. The "Noise" nodes indicate our awareness that these causal relationships aren't perfect. Other factors outside the model will affect the attributes, but we are choosing to treat these unspecified factors as external random variables. A full causal model will also quantify the functional relationships between nodes and specify distributions for each noise variable. Causal models explicitly represent the causal basis of bias, so our assumptions can be clearly identified and tested.

If we built an algorithm to recommend insurance premiums and used "Red Car" to naively adjust insurance premiums, the result would become biased against the ethnicity which preferred red cars. To adjust the insurance premium fairly, we need to infer as much as we can about "Risky Behavior" given observations of "Ethnicity" and "Red Car," then decide using only the a posteriori beliefs about "Risky Behavior." If "Risky Behavior" was directly observed, building a counterfactually fair model would be easy because it is not directly or indirectly caused by "Ethnicity," we could just use it directly. "If I had a different ethnicity but the same risky driving behavior, would I get the same premium?" Yes, if the decision was only made on driving behavior. The difficulty is "Risky Behavior" is not directly observable; we need to infer it from other evidence. So long as the inference of risky behavior given ethnicity and car color is done using Bayesian methods within the causal model, subsequent decisions based on that inference will be counterfactually fair because they only use information from attributes not caused by ethnicity. Furthermore ethnicity can "explain away" a red car, reducting the predicted risky behavior.

Counterfactual fairness can be understood as a highly structured and rigorous form of affirmative action. A counterfactually fair system will in most practical cases adjust outcomes to be more favorable to disadvantaged groups. However the method of calculation is rigorous and repeatable; any person or computer calculating with the same statistical relationships will arrive at the same adjustment. It is highly structured because it provides a formal framework where assumptions can be evaluated empirically, tested, and criticized.

One often proposed fairness criteria is the equal false positive rate. In the case of recidivism prediction, we may require that the number of false positives for each group be equal across protected groups. The Northpointe COMPAS system was criticized by ProPublica [2] because the proportion of black defendants incorrectly predicted to reoffend when they did not was higher than the proportion of white defendants also incorrectly predicted to reoffend. However the creators of this statistical model countered that it was fair in another way: It was "calibrated" [3]. It had equal predictive accuracy for both black and white defendants. Unfortunately these two notions of fairness cannot be achieved together, except in special situations. Counterfactual fairness can resolve incongruities like this by bringing in another critical piece of knowledge that is not available from statistics alone: the causal relationships that lead to bias. Instead of requiring ad-hoc statistical prescriptions, we can trace the complete causal structure of bias to eliminate it.

At first the attempt to mathematize something as intuitive, organic, and (dare I say it) human as fairness with a computer may seem wrong headed. But I argue this line is unreliable. There are many problems that were thought beyond the ability of a computer: chess, Go, and poker are immediate historical examples of tasks that many thought required the human spark, only to be fully mathematized via game theory, tree search, and machine learning. These games do have fixed rules, but then tasks such as speech recognition, image recognition, and natural language processing have no fixed rules, and computers have achieved significant practical performance in these domains, too. So why not a practical mathematical approach to fairness? The tools are ready to be put to the test.

Counterfactual fairness is a new approach to fairness in machine learning, statistical models, and algorithms. It draws on the new field of causality to go beyond statistical relationships and correlations and to model the root causes of differences between protected groups. A mathematized notion of fairness removes the fluff and emotion from fairness, allowing us to clearly model and compare our beliefs about causal relationships in the world, and to empirically test our claims about bias and how it affects decision-making.

Counterfactual fairness is an improvement on simple thresholds and statistical rules, giving us a rich theoretical framework to ask questions about fairness. Tools like this are the road to ensuring that algorithms—constantly making decisions about us—do so fairly. ✦

hello@humanreadablemag.com

ARTICLE LINKS

1. https://hrm.link/counterfactual-fairness
2. https://hrm.link/ProPublica-compas
3. https://hrm.link/calibrated-model

human readable magazine

for the inquisitive developer

Technical deep dives that expand and challenge
your knowledge of programming every month

COMING SOON TO

KICKSTARTER

MORNING CUP OF CODING

From the creator of Morning Cup of Coding