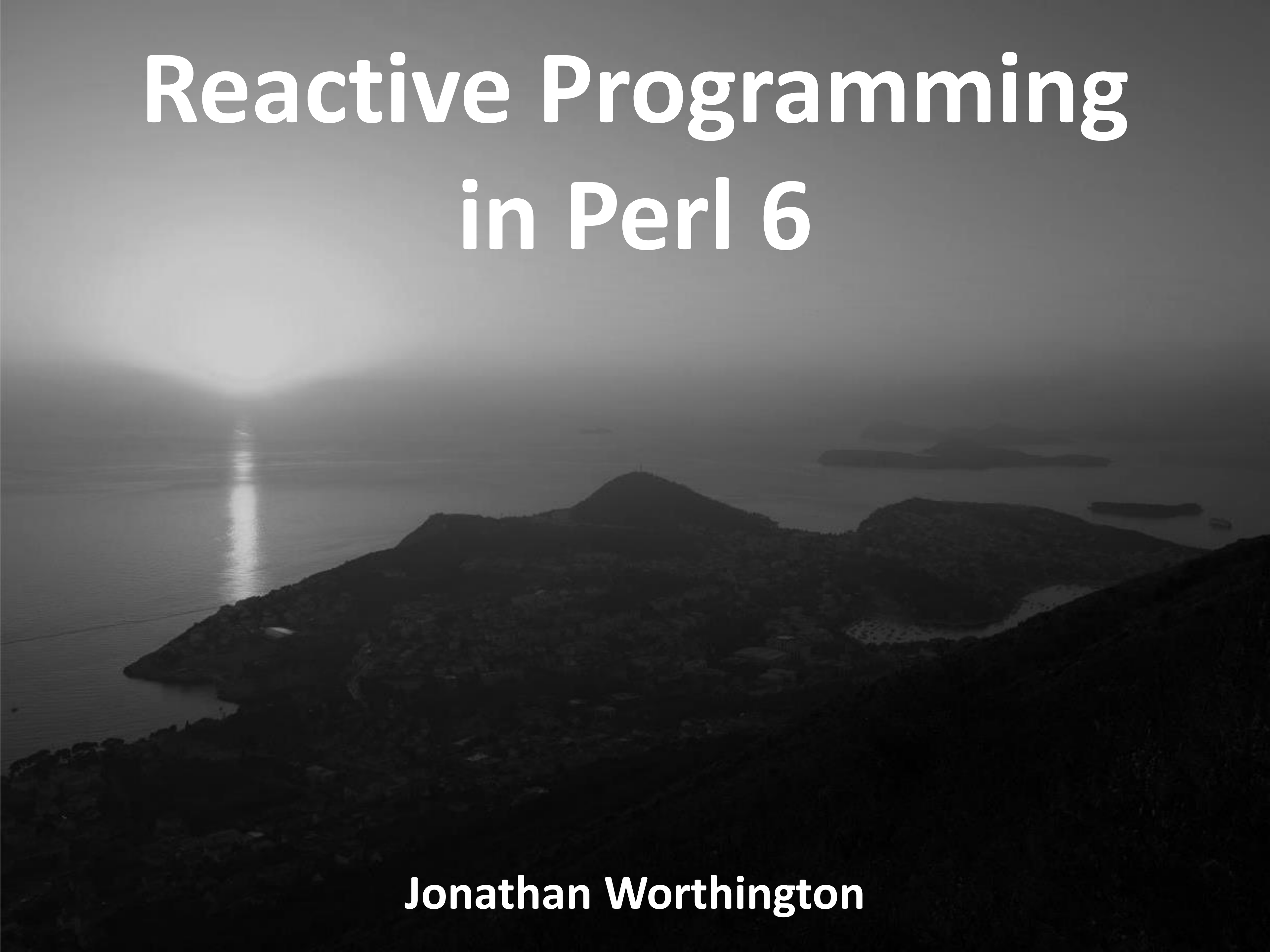


Reactive Programming in Perl 6

Jonathan Worthington

Asynchronous data

It's all around us, in all kinds of systems:

Events in GUI applications

Web requests / responses

File change notifications

Ticks of a timer

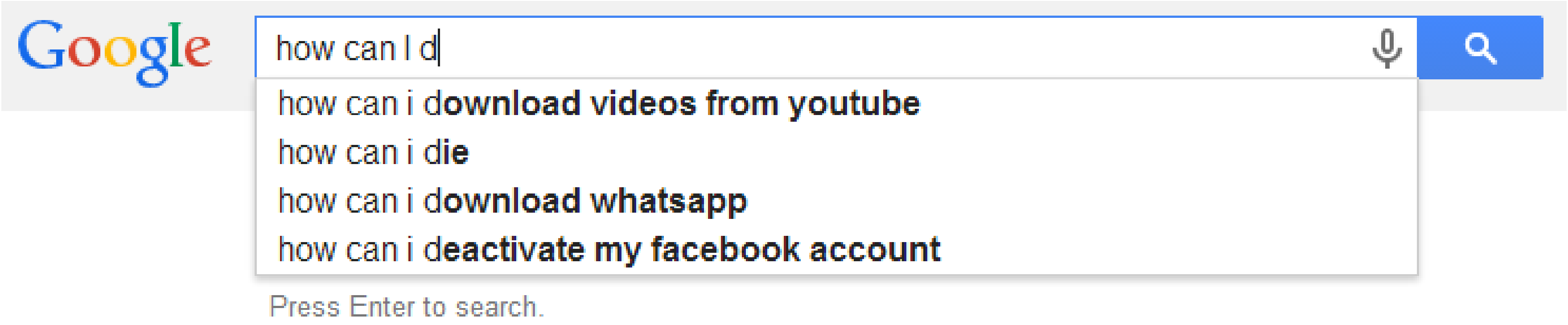
UNIX signals

Characteristics of asynchronous data

You don't get to choose when the data arrives

Multiple sources of asynchronous data can produce data in whatever order they please

Responses may arrive out of order with respect to requests (consider auto-complete)



how can I d

how can i download videos from youtube

how can i die

how can i download whatsapp

how can i deactivate my facebook account

Press Enter to search.

Is this about threads?

Doesn't have to be

But in reality, sometimes *will* be

Users like responsive GUI applications
➔do long computations on a thread
➔result arrives asynchronously

Many web applications in the world are multi-threaded (consider .Net or Java, where multiple request-processing threads are active)

So, in summary…

AAARRRRRRRRGGGGHHHHHHH!!!!

An aside: lazy processing

Before we look at asynchrony, let's consider synchrony a bit. We know that normal file I/O is blocking and synchronous. However, we can work through the data a little at a time:

```
while (!eof($fh)) {
    my $line = <$fh>;
    next if $line =~ /^\#/;
    # …
}
```

This is an example of the iterator pattern - moving through a list of things one at a time

Working with lists of things

We don't just have to use the typical imperative programming constructs to deal with lists

We can factor out the flow control, with things like `map`, `grep`, `sort` - and many more in Perl 6!

```
my @members   = get_gold_members();
my @entrants = @members.grep(*.points > 10000);
my @winners   = @entrants.pick(10);
my @names     = @winners.map(*.name);
```

So how does this relate to the iterator pattern?

Perl 6 lists are lazy!

In Perl 6, normal lists can be processed a bit at a time. In fact, they can even be infinitely long!

Here we make an infinite list of Fibonacci numbers, grep out the even ones, and show 10:

```
my @fibs      := 1, 1, * + * ... Inf;
my @even_fibs := @fibs.grep(* % 2 == 0);
say @even_fibs[^10];
```

Normal assignment is mostly eager - to avoid giving nasty surprises! So we use binding here.

But who gives a #@%&?

When the hell does the everyday programmer need the Fibonacci numbers?!

Back to files!

Lines in a file are just a lazy list. So in Perl 6 you can just write a `for` loop over the lines in a file:

```
my $fh = open('omg-loads-of-data.txt');
for $fh.lines -> $line {
    …
}
```

And, of course, use `grep`:

```
for $fh.lines.grep({ !/^ '#'/ }) -> $line {
    …
}
```

Factoring out flow control

What makes things like `map` and `grep` powerful is they enable us to factor out flow control

Things like `uniq` and `squish` go a step further, and factor out state:

```
my @all_results  = @bing_top_10, @google_top_10;
my @uniq_results = @all_results.uniq(:as(*.url));
```

Inside of here is a hash and a whole bunch of stateful operations on it - that we can forget
➔ work at a higher abstraction level

What if we could do this for asynchrony?

If such a way exists, then we can…

…factor out the complexity and recurring problems of asynchronous programming

If such a way exists, then we can…

…factor out the complexity and recurring problems of asynchronous programming

…be able to compose different sources of asynchronous data in a sane way

If such a way exists, then we can…

…factor out the complexity and recurring problems of asynchronous programming

…be able to compose different sources of asynchronous data in a sane way

…make kicking work off to another thread, and updating a UI with the results, not hurt

If such a way exists, then we can…

…factor out the complexity and recurring problems of asynchronous programming

…be able to compose different sources of asynchronous data in a sane way

…make kicking work off to another thread, and updating a UI with the results, not hurt

…end suffering, bring world peace, make cats and dogs love each other, and other crap

Enter category theory

I USED TO BE A CATEGORY THEORIST...
...BUT THEN I TOOK AN ARROW TO THE KNEE
memegenerator.net

Iterators and observers are duals

Iterators

=

Give me a value

Give me a value

…

Observers

=

OMG a value! Do something!

OMG a value! Do Something!

…

What does it all mean?

If we can define something on iterables, then we can also sanely define it on observables

That is, we can define the familiar operations on synchronous data on asynchronous data too

Into them we can factor not only flow control and state, but also thread-safely, synchronization, running things on the right thread, timing issues - many of the things that make this all so hard!

Supplies

In Perl 6, we call the thing that can throw asynchronous data at you a `Supply`

For example, `Interval` makes a Supply that can throw ascending numbers at you per time unit:

```
my $ticker = Supply.interval(1);
```

This is an on-demand supply; we must tap it (providing an action) to start getting the ticks:

```
$ticker.tap({ say "Started $_ secs ago"; });
```

map the future!

Thanks to duality, we can implement things like map and grep on supplies!

These produce a new supply that, when tapped, will in turn tap its source, transform each value thrown at it, and throw it onwards:

```
my $ticker   = Supply.interval(1);
my $ticktock = $ticker.map({
    $_ % 2 ?? 'tock' !! 'tick'
});
$ticktock.tap(&say);
```

Supplies and concurrency

Supplies only introduce concurrency if needed. For example, the following is single-threaded:

```
my $beer = Supply.new;
$beer.tap({ say "I'll drink a $_" });
$beer.more('Chimay');
$beer.more('Duvel');
```

By contrast, our interval example scheduled its callbacks on the thread pool. If we do not keep the main thread alive (e.g. by sleeping for a while), then our program would exit right away.

Let's build something real!

I love Git. Once I hand my work to it, I know that it won't be lost. But what about before I commit?

Enter inter-commit! It will make backups of files each time I save them, keeping an index of them.

When I commit, it throws the backups away automatically (because Git has the files now)

Let's see how we can implement it with Perl 6's asynchronous programming support

IO notifications

Modern operating systems can provide notifications upon changes to files

These occur asynchronously, and are thus exposed in Perl 6 as a supply:

```
my $commits = IO::Notification.watch_path(
    '.git/logs/HEAD');
$commits.tap({
    say 'OMG a commit!';
});
```

Clearing the backups on commit

We're going to keep the backups in a directory .inter-commit. We can thus do the on-commit cleanup of that directory with:

```
my $commits = IO::Notification.watch_path(
    '.git/logs/HEAD');
$commits.tap({
    for dir('.inter-commit') {
        unlink($_);
    }
});
```

Now, let's turn to the backups…

Detecting file changes

Watching a directory produces notifications of changes to files in that directory:

```
my $changes = IO::Notification.watch_path('.');
$changes.tap(&say};
```

This works, but oddly we find ourselves getting duplicate notifications on some platforms:

```
Change.new(path => "awesome.p6", event =>
FileChangeEvent::FileChanged)
Change.new(path => "awesome.p6", event =>
FileChangeEvent::FileChanged)
```

De-duplication

So, how do we de-duplicate them?

A user won't change and save a file more than once per second - but they may save multiple files at once. So, we use `uniq` to filter out duplicates by path, but make the filter entries expire after a second has elapsed:

```
my $all    = IO::Notification.watch_path('.');
my $dedupe = $all.uniq(:as(*.path), :expires(1));
$dedupe.tap(&say);
```

Making the backups

We want to make sure we don't trigger a copy on changes to the backup directory itself. Other than that, the rest is not too hard:

```
IO::Notification.watch_path($dir)\
    .uniq(:as(*.path), :expires(1))\
    .map(*.path)\
    .grep(* ne '.inter-commit')\
    .tap(-> $backup {
        ++state $change_id;
        spurt '.inter-commit/index', :append,
            "$change_id $backup\n";
        copy $backup, ".inter-commit/$change_id";
    });
```

A slight problem: race conditions

If the user saves a few files together, we may get the notifications being processed concurrently by the various threads in the thread pool

We're vulnerable to races on the change ID state variable as well as appending to the file:

```
…   .tap(-> $backup {
        ++state $change_id;
        spurt '.inter-commit/index', :append,
            "$change_id $backup\n";
        copy $backup, ".inter-commit/$change_id";
    });
```

Making the backups

The trick is to use `act` instead of `tap`. This promises that the block will never be executed concurrently (act = actor semantics ☺)

```
IO::Notification.watch_path($dir)\
    .uniq(:as(*.path), :expires(1))\
    .map(*.path)\
    .grep(* ne '.inter-commit')\
    .act(-> $backup {
        ++state $change_id;
        spurt '.inter-commit/index', :append,
            "$change_id $backup\n";
        copy $backup, ".inter-commit/$change_id";
    });
```

Putting the pieces together

We'll put the two watchers into private methods, drop them in a class and create a supply that can serve as a log of things that happen:

```
class InterCommitWatcher {
    has $.log;

    submethod BUILD(:$base) {
        $!log = Supply.new;
        self!watch_HEAD();
        self!watch_dir($base);
    }
    …
}
```

The entry point

Write a MAIN sub so "inter-commit watch" starts watching, and shows log entries

```
multi sub MAIN('watch') {
    unless '.git/HEAD'.IO.e {
        note "Use inter-commit in a Git repo";
        exit(1);
    }

    mkdir '.inter-commit';
    my $icw = InterCommitWatcher.new(base => '.');
    $icw.log.tap(&say);
    sleep;
}
```

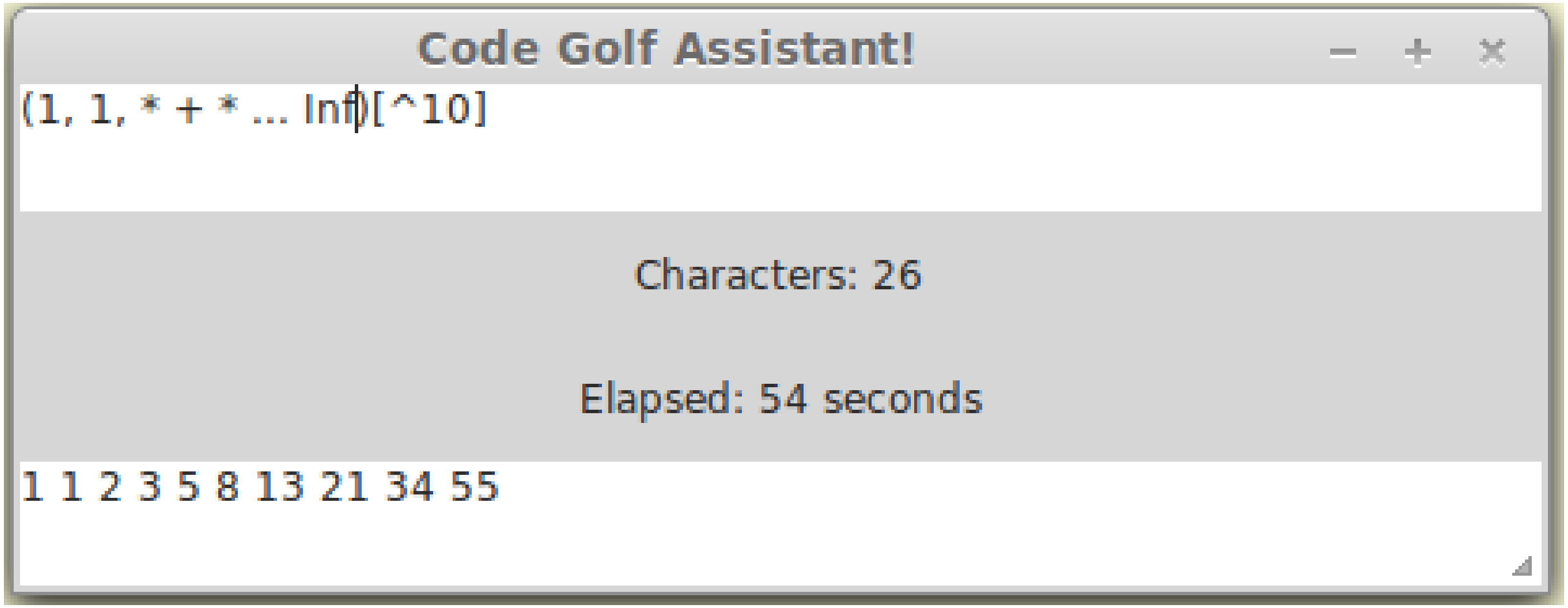
Composing multiple asynchronous things

Supplies and the methods available on them were certainly helpful here - but we were only dealing with a single source of asynchronous things

For our second example, we'll see how we can effectively juggle 3 different sources of asynchronous data, namely:

UI events
Timers
Background computation on a thread

Code golf assistant



```
(1, 1, * + * ... Inf)[^10]
```

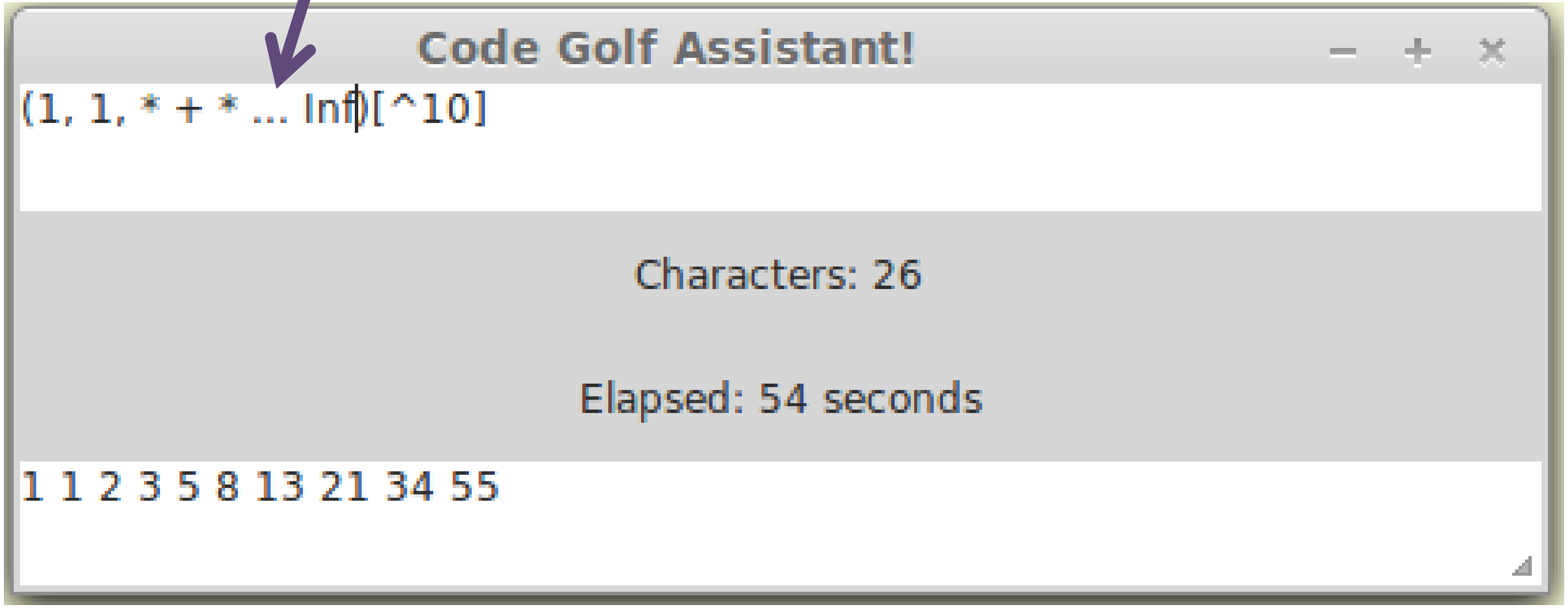
Characters: 26

Elapsed: 54 seconds

```
1 1 2 3 5 8 13 21 34 55
```

Code golf assistant

Type code here



```
(1, 1, * + * ... Inf)[^10]
```

Characters: 26

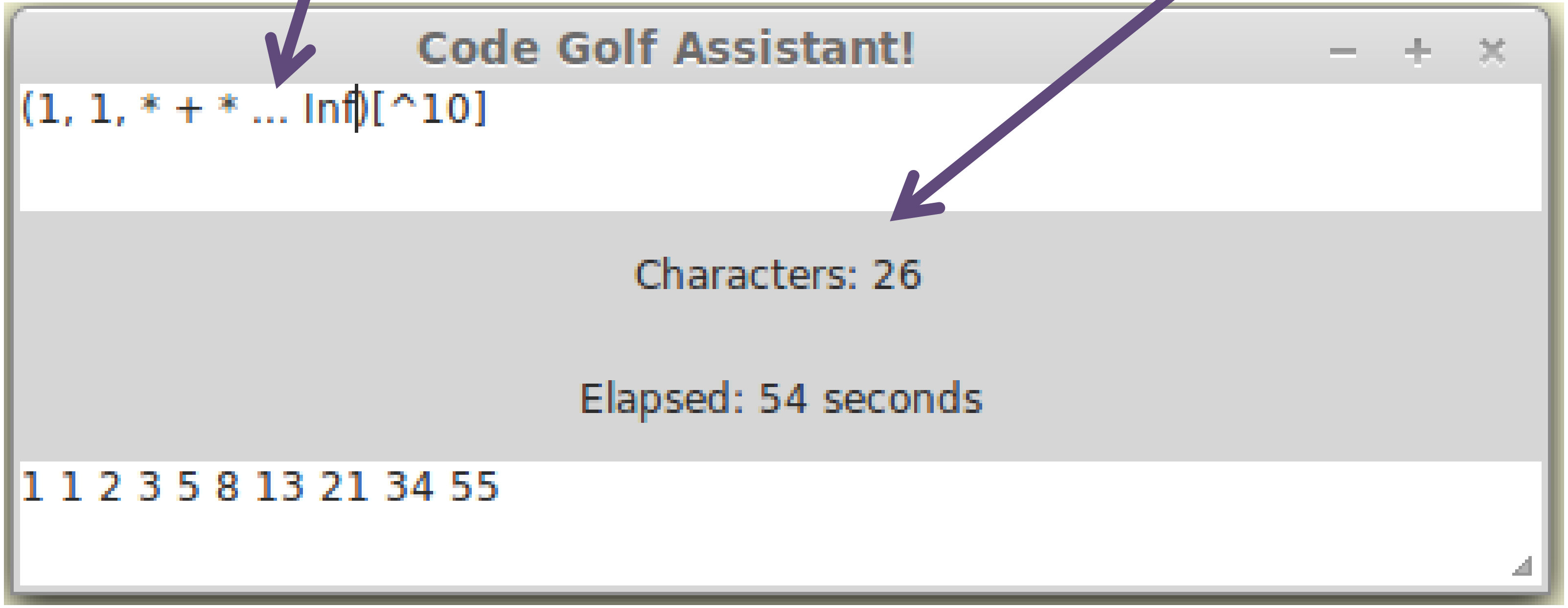
Elapsed: 54 seconds

```
1 1 2 3 5 8 13 21 34 55
```

Code golf assistant

Type code here

Char count updates automatically



```
(1, 1, * + * ... Inf)[^10]
```

Characters: 26

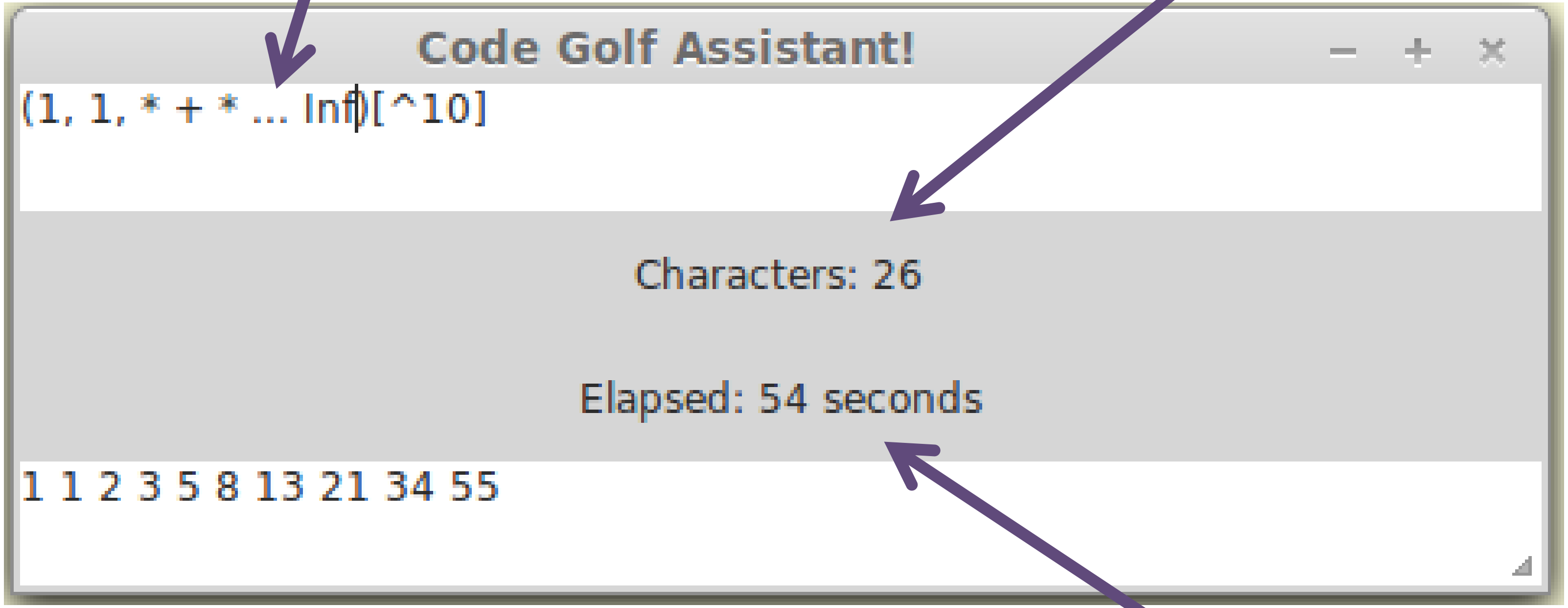
Elapsed: 54 seconds

```
1 1 2 3 5 8 13 21 34 55
```

Code golf assistant

Type code here

Char count updates automatically



```
(1, 1, * + * ... Inf)[^10]
```

Characters: 26

Elapsed: 54 seconds

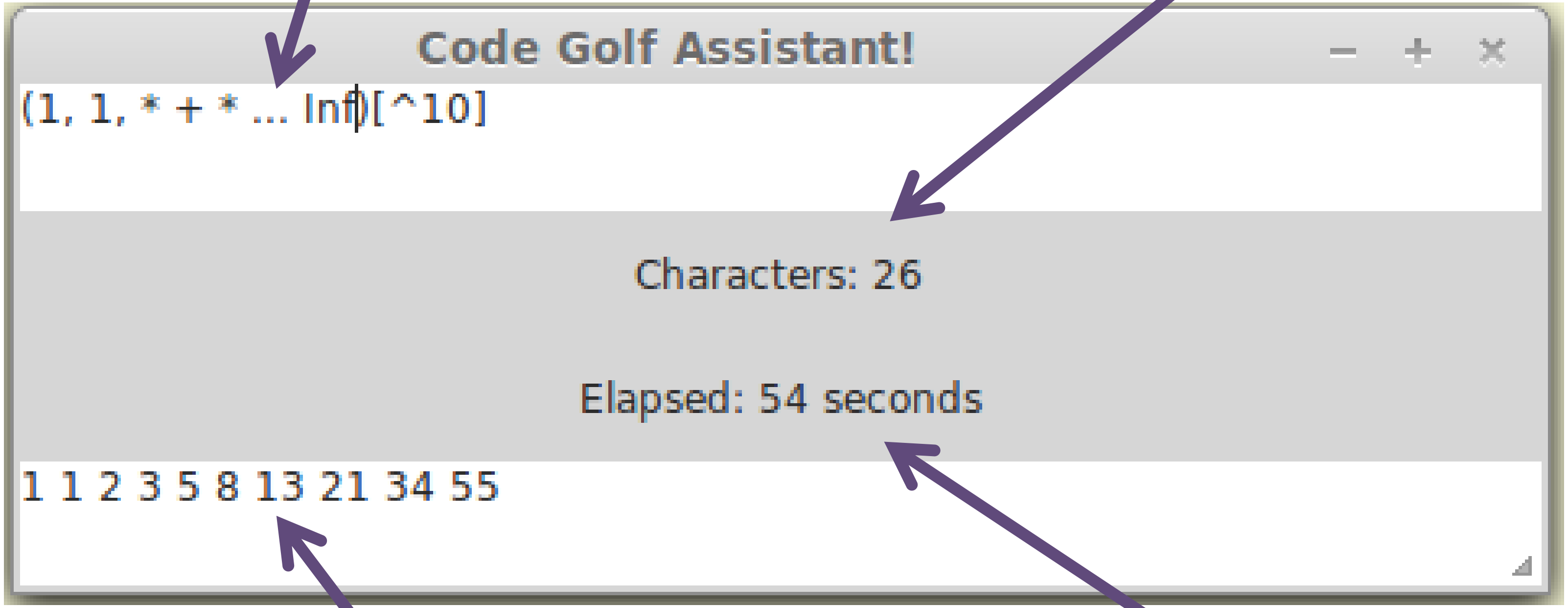
```
1 1 2 3 5 8 13 21 34 55
```

Show how much time I've wasted

Code golf assistant

Type code here

Char count updates automatically



```
(1, 1, * + * ... Inf)[^10]
```

Characters: 26

Elapsed: 54 seconds

```
1 1 2 3 5 8 13 21 34 55
```

Run code in background and show result

Show how much time I've wasted

I had a slight problem...

Nobody wrote a GTK binding for Perl 6 yet

I had a slight problem…

Nobody wrote a GTK binding for Perl 6 yet

So I wrote GTK::Simple on the train here, to enable me to write the golf assistant ☺

Setting up the UI

Add various controls, and keep them in variables so we'll be able to refer to them later

```
my $app = GTK::Simple::App.new(
    title => 'Code Golf Assistant!');

$app.set_content(GTK::Simple::VBox.new(
    my $source  = GTK::Simple::TextView.new(),
    my $chars   = GTK::Simple::Label.new(
        text => 'Characters: 0'),
    my $elapsed = GTK::Simple::Label.new(),
    my $results = GTK::Simple::TextView.new(),
));
```

Events are exposed as supplies

UI events are exposed as live supplies (since the events happen whether they are tapped or not)

This means many things can tap a given event

Here's how we update the character count label whenever the code in the source textbox changes:

```
$source.changed.tap({
    $chars.text =
        "Characters: $source.text.chars()";
});
```

The ticking seconds

Here, we need to be a little careful. It may at first be tempting to just write:

```
Supply.interval(1).tap(-> $secs {
    $elapsed.text = "Elapsed: $secs seconds";
});
```

However, this will probably end very badly

Timers fire in the thread pool, as we saw earlier - but you should only update a user interface from the main thread of the application!

Schedulers

Schedulers are at the heart of Perl 6 concurrency

Schedulers are relatively simple from the outside: you give them work to do, and they make it happen (for example, `ThreadPoolScheduler` schedules work onto a pool of threads)

The GTK::Simple module includes a scheduler, `GTK::Simple::Scheduler`, that accepts work, hooks into the GTK event loop, and runs the work on the main, user-interface, thread

Safely updating the UI

The `schedule_on` method takes a scheduler, and makes sure the next step of the asynchronous data pipeline executes using it

This means we can ensure that the UI updates are done safely on the main thread

```
Supply.interval(1).schedule_on(
    GTK::Simple::Scheduler
).tap(-> $secs {
    $elapsed.text = "Elapsed: $secs seconds";
});
```

Running the code

Let's start with the simplest thing that could possibly work, and then deal with its issues

```
$source.changed.tap({
    $results.text = (try EVAL .text) // $!.message
});
```

This sucks in two key ways:

It evaluates the code on every single keystroke

and

Evaluates it on the UI thread, freezing up the UI

Waiting for a stable value

Rather than running the code on every single keystroke, it makes more sense to do it when the user stops typing for a bit

The unchanged method waits for the source it taps to have no new data for a certain time period, and then propagates the latest value - which here maps to the user stopping typing

```
$source.changed.unchanged(1).tap({
    $results.text = (try EVAL .text) // $!.message
});
```

Evaluating the code on another thread

Fraught with danger!

Evaluating the code on another thread

Fraught with danger!

Of course, we need to update the UI on the main thread - but we already know how to do that

Trickier is dealing with this situation:

Start to evaluate a thing that takes a while
Then evaluate something that runs quickly
Show the result of that latest thing
Then the old, slow thing is done and overwrites it

The `start` method

The start method schedules a block of code to run on the thread pool scheduler

```
$source.changed.unchanged(1).start({
    (try EVAL .text) // $!.message
})
```

It then immediately pushes a supply to whatever taps it. This means we are now dealing with a supply of supplies - the inner ones representing the evaluation of each piece of code!

The `migrate` method

A supply of supplies is an asynchronous stream of asynchronous streams. The `migrate` method always taps the latest available stream, and ignores results from earlier ones - ensuring we will never overwrite a new result with an old one!

```
$source.changed.unchanged(1).start({
    (try EVAL .text) // $!.message
}).migrate().schedule_on(
    GTK::Simple::Scheduler
).tap(
    { $results.text = $_ }
);
```

Entering the runloop

With everything set up, all that remains is to enter the GTK runloop:

```
$app.run();
```

And with that, we've implemented all of the features for the code golf assistant - in 29 lines!

In those 29 lines we've handled UI events, worked with time, used multiple threads, handled race conditions, and provided a responsive UX!

Aside (if time): inside GTK::Simple

I didn't write a single line of C code; everything is done with the Perl 6 NativeCall module

Here's a simple example:

```
use NativeCall;

my class GtkWidget is repr('CPointer') { };

sub gtk_widget_show(GtkWidget $widget)
    is native('libgtk-3.so.0')
    {*}
```

Aside (if time): C callbacks to supplies

```
has $!changed_supply;
method changed() {
    $!changed_supply //= do {
        my $s = Supply.new;
        g_signal_connect_wd(
            $!gtk_widget, "changed",
            -> $, $ {
                $s.more(self);
                CATCH { default { note $_; } }
            },
            OpaquePointer, 0);
        $s
    }
}
```

Wrapping up...

Asynchronous things aren't uncommon, and have been getting increasingly important

Dealing with them is traditionally complicated, because the mechanisms used compose badly

Reactive programming enables a lot of the difficult things to be factored out, and also enables easy composition

"Make the easy asynchronous things easy"

Thank you!

Questions?

You can find the code samples from the talk at
github.com/jnthn/perl6-reactive-samples

If you want to contact me…
Email: jnthn@jnthn.net
Twitter: @jnthnwrthngtn