



Proceedings

YAPC::Europe 2012



MR PORTER NET-A-PORTER.COM THE OUTNET.COM



Duck Duck Go



O'REILLY®



Foreword	5
FatPacker: understanding and appreciating the insanity	6
Vorbild::Beitrag::POD	8
CGI.pm MUST DIE - Together we shall annihilate CGI.pm!	9
CPANTS: Kwalitative website and its tools	10
Designing the Internet of Things: Arduino and Perl	11
Dancing with WebSockets	12
Dancer 2 - Official status:	14
Practical Dancer: moving away from CGI	16
Bringing Perl to a Younger Generation	18
Asynchronous programming FTW!	22
Mojolicious	24
Continuous deployment with Perl	36
Array programming for mere mortals	37
Ontology Aware Applications	42
Distributed Code Review in Perl - State of the Practice	48
address-sanitizer - A fast memory error detector	52
Exceptional Perl 6	55
The joy of breaking stuff	59
Macros in Rakudo	66
Why statement modifiers can harm maintainability!	70
A discussion on how to organize a Perl Mongers group	72
Building C/C++ libraries with Perl	74

NOW HIRING!

World's #1 In Online Accommodation Reservations...and still growing!

Booking.com

We need: Perl Developers, MySql DBAs,
Software Developers, SysAdmins,
Web Designers, Front End Developers,
Network Engineers and more...



We use: Perl, Puppet, Apache,
MySQL, Memcache, Git, Linux, Cisco,
Juniper and more...

NOW HIRING!

Great location in the center of Amsterdam
Competitive Salary & Relocation Package
International, result driven & dynamic work environment



Interested? www.booking.com/jobs

Welcome to YAPC::Europe 2012. This is the fourteenth European Perl conference! The Frankfurt Perlmongers have great pleasure in hosting this event this year. We'd like to welcome you here in Frankfurt. The city that is now the heart of the Perl community for at least 3 days.

We have attendees from more than 40 countries all over the world, so there is a rich mix of different cultures and different people.

Such an event would not be possible without you - the attendees, speakers, and sponsors. Please take the chance to get to know lots of new people and talk to old friends. Please welcome the attendees who are new to Perl conferences and introduce them to other Perl people.

We are sure that the talks and the discussions from this Perl conference will make a major contribution to the Perl world. It will bring new ideas to the surface and existing ideas to new users. With more than 80 talk a lot of topics are covered.

In these proceedings you find the articles for some talks. We hope that the papers will help to remind what was said in the talks when you are back home.

The Frankfurt Perlmongers look forward to seeing over 80 great talks and meeting over 300 people. Enjoy the conference!

Wieland Pusch, Max Maischein, Renée Bäcker
Board of Frankfurt Perlmongers e.V.

Author: Sawyer X (xsawyerx@cpan.org)

FatPacker is yet another module that contains a high dosage of trickery which most of us probably wouldn't think of, but solves a common problem that many of us share: the need to ship a module with all of its dependencies.

This talk will cover what FatPacker is, how to use it and mainly: how it does its charm.

Bio Sawyer X

Sawyer X is a systems administrator and Perl developer, involved in various projects (most notably *Dancer*). He taught a Perl course, worked on Perl/Android, lectures often, and rambles occasionally on his blog.

Abstract

FatPacker was written by *mst* and allows us to create a packed file with all of its dependencies. We can then send that file to wherever we want, and as long as it has a Perl interpreter, it will work and will not require any prerequisite installations.

You might recognize that *cpanminus* provides a packed version under the domain <http://www.cpanmin.us/> which enables a user to run *cpanminus* straight from the command line without installing it via:

```
curl -kL cpanmin.us | perl - My::Module
```

You can also download it and use it:

```
curl -kL cpanmin.us > cpanm
perl cpanm My::Module
```

How to use

To use FatPacker from the command line, simply execute the following commands, replacing *myscript.pl* with a script file you had written:

```
fatpack trace myscript.pl
fatpack packlists-for
'cat fatpacker.trace' >packlists
fatpack tree `cat packlists`
(fatpack file; cat myscript.pl)
>myscript.packed.pl
```

How it works

Tracing

The major part of FatPacker is `App::FatPacker::Tracer`. The tracer allows to trace all the modules being used by your application. It does that by dividing to two parts:

- Setting `B::minus_c`

The tracer sets the `-c` option using `B` (the same way that `perl -c` works. Since this cannot be reversed once done, the tracer has to be run as a separate process (though FatPacker takes care of that).

This option is set in order to prevent running the script's code.

- Catching the loaded modules

Since the `-c` option is used, no code can actually run now. Or... is that so? There are still some blocks that can run. One of them is the `CHECK` block, which is pertinent to this operation, since it is used to catch the transition after compilation but before execution. This means that the `CHECK` block can run after you had already ran all the `use` statements, which are compile-time statements.

Then we can see what's in your `@INC`.

Unlike many other attempts to find used modules, FatPacker's tracer goes to the source: what `perl` actually loads. This gives it very accurate results (excluding `require` statements), and alongside the ability to add additional modules (such as those you intend to `require`), it is a powerful tool indeed.

Fetching packlists

Once we have traced all the modules being used, we use FatPacker to fetch the packlists. Those contain all the modules that were installed by the distribution that carried whatever module we've found in the trace. If we have a script that uses the wonderful `Data::Printer`, the trace will find `File/HomeDir/FreeDesktop.pm` is a module that is being used (in that form) and the packlist for it will contain all the modules that come with the

File::HomeDir distribution, which carries *File::HomeDir::FreeDesktop*.

Creating the tree

Once we have the packlist file locations saved, we can use `fatpack tree` to create a `fatlib` directory which will contain the entire *File::HomeDir* distribution.

We could also include our own stuff in a `lib` directory.

Creating the packed tree

Finally we can use `fatpack file` in order to create a single file made up of all the distributions we've collected. We then concatenate our script to the output as well and throw it all in a nice bow and there it is: a completely self-contained Perl script (sans the perl interpreter).

Caveat

FatPacker can only pack Pure-Perl modules, not XS.



EUROPE
ASIA PACIFIC
AMERICA



Wir suchen helle Köpfchen!

Die eGENTIC Systems GmbH ist der technische Dienstleister für eGENTIC, dem Weltmarktführer in der Online-Lead-Generierung. Gemeinsam sind beide Firmen seit fast zehn Jahren überaus erfolgreich am Markt aktiv und verfolgen einen dynamischen, auf weltweite Expansion angelegten Wachstumskurs. Die Tätigkeits-Schwerpunkte von eGENTIC Systems liegen in der Entwicklung von Web- und Mobile-Applikationen und der Betreuung der benötigten Server-Ressourcen.

Senior Web-Entwickler Perl / MySQL (m/w), Darmstadt

Perl ist für Sie wie eine zweite Muttersprache? Ihr Code spricht für Sie? Sie arbeiten professionell und eigenständig, denken analytisch und schätzen ein entspanntes, freundliches Arbeitsumfeld? Dann sind Sie bei uns richtig.

Das sollten Sie mitbringen

- › sehr gute Kenntnisse über aktuelle Technologien und Standards (Perl, MySQL, Javascript, HTML und XML unter Apache/Linux)
- › mehrjährige Erfahrung in der Betreuung komplexer Perl-Webapplikationen
- › sehr gute Deutsch-Kenntnisse in Wort und Schrift
- › gute Englisch-Kenntnisse in Wort und Schrift
- › ein abgeschlossenes Fachstudium oder eine ähnliche Ausbildung finden wir gut entscheidend sind für uns aber vor allem Ihr Können und Ihre praktische Erfahrung

Das sind Ihre Aufgaben

- › die Entwicklung von datenbankgestützten Web-Applikationen gemeinsam mit Ihren Teamkollegen
- › die verantwortliche Betreuung und Entwicklung der Projekte unserer Kunden
- › Konzeption, Realisierung, Integration und Pflege der Applikationen, ihrer Module und Erweiterungen
- › das Quality-Management unserer Anwendungen
- › die Kommunikation mit unseren Partnern und Kunden

Wir bieten Ihnen

- › eine bunte Mischung an Mitarbeitern und Projekten mit internationalen Partnern
- › ein überdurchschnittliches Basisgehalt und zusätzlich attraktive Bonuszahlungen
- › die Sicherheit eines etablierten Unternehmens mit festem Kundenstamm
- › ein Team, das zusammenhält, und Vorgesetzte, die Ihnen Gestaltungsspielraum geben
- › die Möglichkeit, schnell Verantwortung zu übernehmen und in kurzer Zeit aufzusteigen
- › ein faires Arbeitszeitmodell, bei dem Überstunden die Ausnahme sind und in jedem Fall ausgeglichen werden
- › neue, sehr verkehrsgünstig gelegene Agenturräume
- › individuelle Weiterbildungsmaßnahmen und bei Bedarf Englischkurse während der Arbeitszeit
- › gemeinsame Events wie Skireisen, Firmenpartys und Cocktails nach Feierabend

Sie haben Lust bekommen, uns kennenzulernen? Wir freuen uns auf Sie! Senden Sie uns Ihre Unterlagen per E-Mail an:

jobs@egentic-systems.com eGENTIC Systems GmbH · Frau Jana Keßler · Rheinstrasse 97 · 64295 Darmstadt

+49 6151-8508-0



<http://www.egentic-systems.com>

Author: Chris 'BinGOs' Williams (chris@bingosnet.co.uk)

This article is written in "dual language free style", sometimes additional subsections, sometimes just additional english paragraphs, sometimes obvious examples are not translated. Please report any language problems, you could imagine.

Bio Chris 'BinGOs'

Williams Chris is a systems administrator who is happy to deal with any operating system. He has administrated OS/2, UNIX and Windows networks in his time and mainly deals with Windows and FreeBSD systems these days.

In his spare time he finds time to be a CPAN Tester, a Perl5 porter and POEvangelist.

Abstract

Perl. Windows. The two are often seen as incompatible. But they are not.

For the past thirteen years or so I have been valiantly using Perl in a Windows environment to do useful stuff, from text mangling to Active Directory mangling.

Starting with ActivePerl from ActiveState, experiments with UWIN and Interix (now Services for Unix), dabbling with Cygwin, onto building my own Perls, taking up Strawberry Perl and ultimately back to using Cygwin for day to day work.

The Cygwin choice being more about improvements in Cygwin than limitations of 'native' Perl on Windows, things such as mintty and the new ability to use the screen utility, and being able to work in a familiar and productive command line environment, with Strawberry Perl being the choice for deployment for elsewhere in the server environment.

Active Directory manipulation using ADSI (Active Directory Service Interfaces) is one of the main uses I make of Perl, regular bulk updates to thousands of users when departmental reorganisations occur, generating management reports, to bulk importing thumbnail photo images.

Perl is the glue that mends the broken glass.

Bibliography

- Strawberry Perl: <http://strawberryperl.com>
- ActiveState: <http://www.activestate.com/perl>
- Cygwin: <http://www.cygwin.com>
- mintty: <http://code.google.com/p/mintty/>
- UWIN: UWIN Overview - <http://www2.research.att.com/~gsf/download/uwin/uwin.html>
- SFU/SUA and SUA Community: Services for UNIX: http://en.wikipedia.org/wiki/Windows_Services_for_UNIX
- SUA Community: <http://www.suacommunity.com/SUA.aspx>
- Active Directory Service Interfaces: <http://msdn.microsoft.com/en-us/library/windows/desktop/aa772170%28v=vs.85%29.aspx>

CGI.pm MUST DIE - Together we shall annihilate CGI.pm!

9

Author: Sawyer X (xsawyerx@cpan.org)

Bio Sawyer X

Sawyer X is a systems administrator and Perl developer, involved in various projects (most notably Dancer). He taught a Perl course, worked on Perl/Android, lectures often, and rambles occasionally on his blog.

The purpose of this talk is to ruthlessly and savagely attack CGI.pm in the hopes that it will make you interested at seeing better ways of accomplishing your web programming requirements.

Abstract

CGI.pm has been the de facto standard for web programming, but we're in the future age now. It's time we realize CGI.pm is no longer a benefit, but in fact keeps us away from modern ways of getting web done.



JETZT

GRATIS
VORBESTELLEN:

IHRE NEUE DOMAIN-
ENDUNG IM WEB!

In Kürze verfügbar – Web-Adressen mit neuen Endungen!

Das Web wird vielfältiger: Neben den bekannten Endungen wie .de und .com gibt es ab 2013 zahlreiche neue Adressendungen – z.B. .bayern, .shop, .news, .sport, .berlin, .koeln, .earth und viele mehr.

Jetzt die Gelegenheit nutzen und Ihre neue Adresse fürs Web vorbestellen – kostenlos und unverbindlich!

www.united-domains.de

united domains
Die besten Adressen fürs Web.

Author: Kenichi Ishigaki (ishigaki@cpan.org)

Bio Kenichi Ishigaki

Kenichi Ishigaki (aka charsbar) is a freelance programmer as well as a writer/translator.

Abstract

After the Perl QA Hackathon 2012, I have been working on the refactoring of `Module::CPANTS::ProcessCPAN`. In this session, I will explain some of the issues I have encountered, and what I have done and/or plan to do in the near future.

Background

Japanese Perl community has been enjoying the CPANTS Kwalitee Game since 2005. In the first two years, Koichi Taniguchi extracted the ranks of Japanese CPAN authors by eye, but it seemed so painful that I created `Acme::CPANAuthors::Japanese` to maintain a list of Japanese authors and took over the reporter's role at YAPC::Asia 2008. I also reported the ranks in 2009 and 2010. In 2011, I created a website for `Acme::CPANAuthors` to add something new to my annual report. Unfortunately, the official CPANTS Kwalitee site had been down for some time and I couldn't get the latest information that year. I hurriedly set up an unofficial Kwalitee site. I had to fix several issues, but it seemed to work reasonably fine with some warnings of deprecation. In 2012, I attended the Perl QA Hackathon to get some tuits to remove these warnings. I also refactored the `Acme::CPANAuthors` website before the hackathon and added similar tables and graphs as of the CPANTS Kwalitee site, hoping to improve `Module::CPANTS::Site` eventually. After the hackathon and the following OSDC.TW 2012, I updated my Kwalitee site to reflect the changes happened at the hackathon, and found something bad occurred.

Issues

One of the fundamental issues was that `Module::CPANTS::ProcessCPAN` depended on the internals of `Module::CPANTS::Analyse` so much that it easily died when the data structure was changed. It also had almost no tests, so you couldn't easily tell if a change would break things or not until you actually run the process. The processing time was another issue; it required almost three days to process the whole CPAN on the server. The way it stored analyses was also problematic; it was nice to keep the past Kwalitee records, but it was hard to merge records processed elsewhere.

Work in progress

So I started refactoring. For efficiency, I wrote a barebone script that processes the whole CPAN in parallel in the first place. With a SQLite queue and a set of workers, the processing time decreased significantly. I also tested it against the BackPAN to see if it can store the whole data reasonably, which also seemed successful so far. I wrote `Archive::Any::Plugin::Bzip2` and then `Archive::Any::Lite` during the course of this parallel testing. I also wrote something to provide only a small part of the CPAN/BackPAN, which proved to be quite useful to write a test on a distribution that spits an error while processing.

As of this writing, the refactoring is still in its early stage. My first goal is to write a series of scripts to generate enough data for the Kwalitee site, but there should be much more to be done. I hope to talk about them more in detail in the session.

Further Reading

Slides will be uploaded under <http://www.slide-share.net/charsbar/>

Author: Hakim Cassimally (osfameron, hakim@doesliverpool.com)

Bio Hakim Cassimally

Hakim is a relative newcomer to the world of electronics and making, and is finding that his university education in Medieval Italian poetry hasn't adequately prepared him for some of the challenges of the Internet of Things. His main achievement has been proposing a simplified unified 1-10 scale for electricity, thus abstracting away pointless complexities like "volts", "amps", "watts", and "ohms". He is perplexed that so far this scale hasn't taken off, but it's only a matter of time.

Hakim is co-authoring a book on the Internet of Things with Adrian McEwen. The book should be published by Wiley and Sons in late 2012 or early 2013. It has (some) Perl in it.

Abstract

An introduction to the exciting world of the Internet of Things (IoT). Connecting physical objects (bubble machines, lamps, plants, chicken-feed silos) to microcontrollers and to the internet.

We'll look at IoT in general, but also how Perl powers things like:

- Early prototypes of Bubblino
- Russell Davies's project "Ghostbox" (internet radio)
- Clockodillo (WIP) API

and others.

Bibliography

- the Book of Things: The book website contains a blog, and updated links to other relevant bibliography. <http://gutenberg.net>.
- Arduino <http://www.arduino.cc/>

Author: Damien Krotkine (dams@cpan.org)

Bio Damien Krotkine

Damien Krotkine is a senior Perl developer and team leader, vice-president of the French Perl Mongers, and rock climber. You can read some of his technical journey discoveries on his low traffic blog.

Damien is a Dancer core developer.

Abstract

Many Perl developers know vaguely what Dancer is - a simple but powerful web application framework for Perl. This talk is an attempt to explain a real-life usage of Dancer, in which unusual techniques have been used.

This use-case is interesting because it is simple enough to understand how to build a Dancer application (beyond the Hello World example), and because it demonstrates the flexibility of Dancer, by showcasing how easy it is to use specific technologies, like WebSockets or AnyEvent.

Background

In the company Damien works for, there are CPAN mirrors. They have to stay a bit behind the real CPAN for stability, but there is a tool to update modules from the real CPAN to the local mirrors. And that works fine.

Then there were the need for a web interface to trigger it, and monitor the injection of new modules (from the real CPAN), into local CPAN mirrors.

The technological challenge

Instead of using a copy-paste approach and pick up an existing web application, and bend it until it fits my needs, it was decided to analyse the needs and use adequate technology.

The problem to solve is not typical. What is required here is a web application that is a frontend to a running process (injecting a module in a CPAN mirror).

It is not like standard web applications (blog, wiki, CRUD, etc). Here there is a long running operation that shall happen only once at a time, that generates logs to be displayed, with states that need to be kept. In this regard, it's interesting to see how Dancer is versatile enough to address these situations with ease.

The rough idea

What should the web interface provide? It should provide a way to:

- input a package name,
- verify that this package exists on the CPAN, and gather some informations on it,
- check that there is a CPAN version that is newer than the one on the local mirrors
- inject the module in the mirror
- provide a way to confirm the injection, and display the process logs.

The backend

The mirrors were created using `minicpan`, and are managed with `CPAN::Mini`. Injecting a new module into these mirrors is done using `CPAN::Mini::Inject`. So injecting the module is basically done using `CPAN::Mini::Inject::inject` with the right arguments, plus some minor other things.

However, to be able to gather informations on the module, It has been decided to use the MetaCPAN API (accessed using `MetaCPAN::API`).

The frontend

The web application should display the status of the mirror: is there an injection going on, what are the logs of the last injection, etc. It should also disallow multiple injections at the same time. It should update the status and logs of the operations in real-time, without needing a refresh on the user's side.

If two clients connect to the web application, they should both see the same screen, the current status, and current injection.

To be able to do that, WebSocket technology is used, which allows bi-directional communications between the server and the client. That means a pseudo “real-time” update of the status.

An HTML will provide a basic form for inputting the module name, and some javascript will do the update of the logs.

In *Dancer*, WebSockets are easy to use thanks to `Dancer::Plugin::WebSocket`, a plugin that is built on top of `Web::Hippie`. These require the server to run on *Twiggy*, using *AnyEvent*.

Using a WebSocket, it is possible to inform the client when something changes on the server. For instance, when a running injection is finished, and it’s now possible to schedule a new one. Or when there is more log text to be displayed.

Setting up a WebSocket is very easy, so messages can be sent from the server. By default, a message content is considered as a log line, except if it has a specific format, in which case it is considered as a status change. Javascript is used to display incoming logs, and change the display of the current status. Also, the button to schedule an injection must be greyed out if an injection is already running. That’s managed using Javascript as well, reacting to the change of status communicated via the WebSocket.

In the server source code, the current status as well as the current log text are stored in simple global variables. As the web server is run under *Twiggy*, powered with *AnyEvent*, there is only one perl process. Thus the global variables are shared for all the clients, so they see the same screen, which is what is required.

The logo for Perl MAG, featuring the word "Perl" in a large, white, sans-serif font, and "MAG" in a smaller, orange, sans-serif font below it. The background is a dark blue/black rectangle with a white diagonal line.

Perl
MAG

A pink, starburst-shaped badge with a white border, containing the text "AUTHORS WANTED" in white, sans-serif font.

**AUTHORS
WANTED**

The title "The Brand New Perl Magazine" in a large, bold, sans-serif font. "The Brand New" is in orange, and "Perl Magazine" is in black. The text is set against a light blue background with a white diagonal line.

The Brand New
Perl Magazine

www.perlmag.com

Dancer 2 - Official status:

What's the official status of Dancer 2?

14

Authors: Damien (dams) Krotkine (dams@cpan.org), Sawyer X (xsawyerx@cpan.org),

Bio Damien (dams) Krotkine

Damien Krotkine is a senior Perl developer and team leader, vice-president of the French Perl Mongers, and rock climber. You can read some of his technical journey discoveries on his low traffic blog.

Damien is a Dancer core developer.

Bio Sawyer X

Sawyer X is a systems administrator and Perl developer, involved in various projects (most notably Dancer). He taught a Perl course, worked on Perl/Android, lectures often, and ambles occasionally on his blog.

Sawyer is a Dancer core developer.

Abstract

This talk will present you with the upcoming major release of Dancer: Dancer 2. It maintains the best of Dancer 1 and adds much goodness. In case you're using Dancer or considering using Dancer, and are interested in the status of the next major version, this is for you.

Dancer is awesome Dancer is awesome. Srsly!

It's fun to use, it's fun to code in. It's great to have something so Perlfully beautiful to share with others. It's a fantastic feeling to be able to showcase Perl so effectively to our friends, our coworkers and our bosses. We've had people contact us who had never used Perl before but they want to use Dancer. Indeed not only is Dancer a great tool to make our lives and work easier, but it's also a great Perl marketing tool. Dancer is how Modern Perl looks and feels like.

Dancer is also a prime example of how to create a lively, robust community. We can count our watchers on Github (500, which is quite impressive), but our fork count is even more impressive: 150 forks! This is a ration of 1 to 3 and a third. That means that (roughly) for every 16 people who watch the Dancer repo, 5 of them had forked it! The contributors count is much higher than that, since we accept contributions from tickets (GH, RT), the mailing list, personal Emails, IRC mes-

sages (in the channel and privately) and pretty much any form a user chooses to help out. We make it a point to be as accessible as possible. Some contributions were not made to the core, but to the ecosystem by writing Dancer plugins which are then available on CPAN.

Dancer has been showcased in newspaper articles, magazines, blogs (even outside the Perl blogosphere), news websites and more. Companies (such as Shutterstock, Novell, and more) are using Dancer. We've seen government websites, personal blogs, ISPs, wikis, libraries as just some of the examples of kickass projects for which people use Dancer.

By any means, we've made it!

Let's talk globals

Dancer does indeed have some set-backs, as anything does. The major issue with Dancer is that some objects are singletons, which means they are global. This led to a tricky situation (amongst others) of not being able to run two Dancer applications in a single process since they override each other in various aspects (requests, responses, serializers, hooks, etc.).

Although We could fix it locally, instead we decided on something more drastic with a much higher gain: rewriting the core, correctly. Enter **Dancer 2!**

What? What's Dancer 2?

Dancer 2 is a complete rewrite of the core (while breaking as little as possible) in order to provide a few key features:

- No globals
Requests, Responses, Serializers, Hooks, everything is a complete lexical object now. This means that you could create as many Dancer apps as you want, working on the same process.

- DSL meta layer
The DSL now has a meta layer and is translated more easily to the object layer of Dancer.

- Clean Internals
The DSL is now built on top of a clean internal layer, that can be used directly by plugins. So instead of messing with the guts of Dancer 1 because of the lack of API, plugins can use

provided features, with no fear of backward compatibility issues.

- **Clever object system**

We're using *Moo* as the object system for *Dancer 2*, which gives us major speed along with the benefit of incredible flexibility and plenty of features.

Dancer 2 also has a few keypoints which make it very interesting:

- **Open to dependencies**

Dancer 1 started with very little dependencies, trying to provide users with more control over the environment on which they are deploying, and the code they are providing. We've maintained a very strict stance on this, but since then things have changed greatly: we've noticed on one hand that *Dancer* users do not fear CPAN and enjoy adding plugins straight from it, and on the other hand we've seen tools that provide much easier installation and deployment for CPAN distributions (such as `App::cpanminus`, `carton`, `App::FatPacker`, `Pinto`, `OrePAN`, and more). So, why keep constricting the number of dependencies?

Dancer 2 is much more liberal and permissive with dependencies, and uses them whenever we find the need for them.

- **Cooperation with others**

There are several web frameworks available on CPAN. Some of them like to cooperate, such as `Catalyst` and `Web::Simple`. Working together (instead of in competition with each other) allows us to share knowledge, implementation algorithms and actual code between us. One major example is the attempt to promote a substantial feature-rich route definition between all three web frameworks. This will make it easier for users to use either one, move between them (picking whichever they want according to their given considerations) and even using them with the same code base!

Dancer 2 is already written!

Alexis Sukrieh (the founder of *Dancer*) had already written *Dancer 2*. There might be a few kinks work out, but these are just the finishing touches.

Dancer 2 development process

Dancer 2's development is organized a bit differently.

- **Time-boxed release cycle**

We're starting a time-boxed release cycle, which means that it will be released on specific intervals, much like it is done in Perl - only more frequently.

- **Policy document**

Dancer 2 carries an internal policy document which we wrote and revise. Once the document is stabilized, it will be available publicly. This document unfolds our vision for *Dancer*, making it easier for contributors and the entire *Dancer* community to understand where we're going with *dancer*, and how we intend to get there.

- **Coordinators**

The development of *Dancer 2* is being overseen by coordinators, each with their own field. This way we're able to coordinate efforts of documentation, compatibility concerns and coding (features, bugs, and whatnot).

- **Compatibility concerns**

Dancer 2 features a forward-compatibility layer in order to prevent breakage as much as possible, but at the same time make it easier for you to use *Dancer 2*'s bells and whistles.

We're also including compatibility shims to allow plugin authors to create plugins that work on *Dancer 1* and *2* at the same time. Such plugins are already out there!

We need your help

What's left for *Dancer 2* is transitioning all other *Dancer* plugins and user applications. These transitions will help us flesh out any remaining parts of *Dancer 2* that aren't covered.

We need help moving all documentation from *Dancer 1* to *2* (while applying the changes being done) and to document all the incompatibilities between *Dancer 1* and *2*, and how to move your application.

Contact us!

Author: Sawyer X (xsawyerx@cpan.org),

Instead of giving a basic short `Dancer` tutorial, we will be taking a CGI application and rewrite it in beautiful `Dancer`.

This will give you an overview of the how `Dancer` applications look like, how to write them, and how they differ from boring old ugly CGI code.

Bio Sawyer X

Sawyer X is a systems administrator and Perl developer, involved in various projects (most notably `Dancer`). He taught a Perl course, worked on Perl/Android, lectures often, and rambles occasionally on his blog.

Sawyer is a `Dancer` core developer.

Abstract

`Dancer` is a modern micro web framework that allows creating fast websites with very little effort.

Considering how comfortable, easy and appealing the new modern web frameworks that exist on Perl (`Dancer` being a major example), there is simply no need anymore for CGI (`CGI.pm`). It's arcane, problematic and hard to work with.

(for more information on that, see lightning talk `CGI.pm MUST DIE!`)

Converting a `CGI.pm` application to `Dancer` can be scary, since they work quite differently. Hopefully we'll be able to make some sense of it, and give you the proper understanding and tools on how to transform your old CGI code to fresh `Dancer` goodness.

Considerations

These are the things we have to take into account when we want to convert our application to `Dancer`.

Path resolving (pretty URLs) and parameters

The first thing you notice in `Dancer` is the route definitions. It provides the long-sought pretty URLs everyone is talking about. This is something that `CGI.pm` always lacked and in `Dancer` they are simply the way it's done.

```
# before
http://example.com/page=article&id=5
```

```
# after
http://example.com/article/5
```

You can see how clean the URL is now. The difference is that the path *implies* where you're going, instead of specifying it explicitly with the variables in the request. It's cool, though, because people don't care about the variables that you use, and seeing them in the path is just ugly.

The major benefit here is that you no longer need to parse this yourself and retain all the logic of understanding what the user wants. It's now part of the path itself, and that includes the parameters you want (or need) to accept from the user, because parameters are part of the path in `Dancer`.

```
# before
my $cgi = CGI->new;

if ( my $page = $cgi->param('page') ) {
    if ( $page eq 'article' ) {
        # user wants an article
        if ( my $id = $cgi->param('id') ) {
            # got an id
        } else {
            # problem...
        }
    } elsif ( $page eq 'subject' ) {
        # user wants a subject
        if ( my $id = $cgi->param('id') ) {
            # got an id
        } else {
            # problem...
        }
    } else {
        # invalid path,
        # return 404 or something
    }
} else {
    # no page requested,
    # is this the main page?
}

# after
get '/article/:id' => sub {
    my $id = params->{'id'};
};

get '/subject/:id' => sub {
    my $id = params->{'id'};
};
```

As you can see, using CGI.pm code, you must check for the parameters, each one separately, and prepare code for each of these cases, such as missing parameters and unknown paths.

With *Dancer* you do not need it. You simply set the paths you want, including the parameters you want. If the user reaches the correct paths, it will run the subroutines you want to run. Otherwise, it's all 404s.

You can also make your paths more flexible, by allowing missing IDs:

```
get '/article/:id?' => sub {
    if ( my $id = params->{'id'} ) {
        ...
    }
};
```

Another beautiful aspect of it is that you can set up the type checking of parameters in the route definition itself.

```
# before
if ( my $id = $cgi->param('id') ) {
    if ( $id =~ /\d+/ ) {
        ...
    } else {
        # user provided bad id
    }
}

# after
get qr{/article/(\d+)} => sub {
    my ($id) = splat;
};
```

Separated views

In CGI.pm there is often no good separation of design (the *view*) and application logic. It's quite common to find bits and pieces of HTML and CSS sitting in your code. Hell, CGI.pm even has special functions to help you write them.

In *Dancer*, your views are separated. You have a **views** directory that contains all your templates. You get a layout structure (which you can disable if you want) automatically. *Dancer* sets up the layout independently (and interoperably with its template engines) so that even if you're using a template engine that does not support such function built in (such as *Template::Tiny*), it will still work for you regardlessly. You can even have *Dancer* let the template handle the layout as well, giving you more control over the templating.

When moving your CGI.pm code, you'll need to start thinking of how to order your design and layout in template files instead of ugly hard-coded HTML. Think of the templates you need to exist, and the variables you want to send them in order for them to act in different ways.

Rendering templates from *Dancer* is very simple:

```
template index => {
    variable_name      => 'content',
    additional_variable => $more_content,
};
```

Reusability

Dancer stresses reusability, in version 2 even more so. Instead of centralizing your route decision making to a single file, you can create multiple files and add them on. Each file can contain routes of a specific purpose, and you can set a prefix for them in order to tie them together.

It's now easier than ever to set up an admin path, for example:

```
# in Admin.pm
prefix '/admin' => sub {
    get '/view'      => sub { ... },
    get '/view/id'   => sub { ... },
};
```

Plugins as refactoring

Dancer has a vibrant and warm community. It hands out rays of sunshine and rainbow cookies!

This community has provided a variety of useful plugins that cut down much of the code you need to write to accomplish certain tasks such as database handling (SQL/NoSQL/ORM), SMS sending, Email sending, LDAP, forms, RSS feeds, I18N, profiling, caching, email sending, sitemaps, and more!

By using these plugins you can dramatically remove major amounts of code and use the integrated solutions other people have already provided for you.

Summary

Moving away from CGI.pm does not take long. It cleans up your code, making it easier to handle, more extensible and flexible and attractive to play with, develop, continue work on it, and get more developers to cooperate on it.

Join the revolution, join Modern Perl!

Authors: Paul Johnson (paul@pjcj.net), Wesley Johnson (wes@wes-johnson.com)

Bio Paul Johnson

I have been using Perl for longer than I care to remember and I'm currently working on a grant to improve Devel::Cover, the Perl code coverage module.

Bio Wesley Johnson

I am half way through my 'A' level studies, which include a Computing course.

Abstract

Perl is now almost 25 years old. Some of us have been using Perl for almost all that time. Many of us have been using Perl for a sizeable fraction of that time. And the reason we have been doing so is because we believe that in those cases it is the best tool for the job, however we define 'best'.

In order for Perl to remain viable it requires a critical mass of users, and that requires involving people who are younger than the language itself.

One way that we have tried to do that is by taking part in the Google Code-In for the last few years. This is a programme aimed at introducing students between the ages of 13 and 17 to open source software.

Introduction

Programming Paradigms

Learning a programming language well enough to be proficient in using it is a commitment. It requires time and effort. Sometimes it is useful to learn a programming language because it will introduce us to new ideas and ways of programming. Every programmer should know at least one language from each of the major paradigms: imperative, object oriented, functional and logical. Each one will allow you to look at problems in a different manner and when you know them all you can choose the appropriate way to solve the problem at hand.

A programmer who hasn't been exposed to all four of the imperative, functional, objective, and logical programming styles has one or more conceptual blindspots. It's like knowing how to oil but not fry.

Tom Christiansen

It's hard to know lots of languages well, and generally we are more productive in the languages we know best and enjoy the most. For many of us, Perl is one of our favourite languages, but in order for Perl to be a viable choice for a project there are certain requirements.

Language Requirements

The language itself needs to be robust. It should have features that help the programmer and it should not be buggy. In addition, the language should have sufficient libraries available for the tasks at hand. And there should be a community around the language to provide the resources and support necessary to learn and use the language, and to ensure that the language and libraries remain in good condition.

Although many people like to argue otherwise, Perl meets all these requirements. The core is solid. It is well maintained and has relatively few serious bugs. The volunteer and grant recipients who maintain it do amazing work.

CPAN is widely touted and Perl's crowning jewel. The quality may be variable, but you are almost guaranteed to find a high quality module to assist you in your project.

And the community around Perl is second to none. Naturally, it is not perfect, but it is vibrant, and full of knowledgeable volunteers.

Perl's Viability

But Perl doesn't have quite the aura that some other comparable languages might have amongst some people. I don't want to investigate here why that might be, but it is clear that unless Perl remains viable as a language, many of us will need to start considering whether we would be better off choosing another language for our projects. And companies will stop using Perl for development which will mean fewer and less interesting Perl jobs.

Additionally, it is generally advantageous to have a number of languages competing in an area. It encourages innovation and copying which benefits all the languages and their users.

In order to keep Perl viable it requires new people to start using and eventually contributing to it. Some of the people at Google have had similar thoughts about programming and Open Source Software in general, and for many years they have been running the Google Summer of Code programme.

This is a programme for university students in which they can work over the summer in the northern hemisphere on a specific Open Source project and be paid for that effort by Google. The Perl Foundation has taken part in this programme almost every year.

Google Code-In

A few years ago Google started the Google Code-In programme (GCI). This is a similar programme to GSoC, but it is aimed at pre-university students between the ages of 13 and 17. It is held over the northern hemisphere winter and allows the students to complete a number shorter tasks of various difficulties for which they receive points. The students also receive payments for the tasks they complete, and those who receive the most points and invited to visit the Google HQ in Mountain View. The Perl Foundation has taken part in GCI each year the programme has been held.

Each organisation taking part in GCI provides a number of tasks in different areas, and categorises the tasks as easy, medium or hard. Tasks should take from a few hours to a few days to complete and students may complete as many as they would like.

Students claim a task and then get a certain amount of time to work on it. Within that time they need to complete and deliver the results of the task. Each task has one or more mentors who will determine whether the student has completed the task and will either confirm that it has been completed or provide assistance if the student needs it. The mentor can also extend the time available if they feel the student is making progress but requires more time. If the student cannot complete the task it is returned to the pool of open tasks and may be claimed by another student.

Preparation

In October 2011 my son, Wesley, tweeted "Can't wait much longer for #gci2011". Ark Keating picked up on that and replied to him, to Florian Ragwitz and to me that we should probably get ourselves organised. So we duly decided on a course of action, put up a wiki to collect suggestions for tasks and publicised it as much as we could on mailing lists, blogs, IRC and twitter.

Our schedule was quite tight. We needed to provide a credible plan to Google to gain admittance into the programme, and that involved including a certain number of tasks in each category. The categories were Code, Quality Assurance, Research, Training, Translation, Documentation and Research.

We received a wonderful response from the community - the first of many - and put together a proposal which was good enough to persuade Google to accept us into the programme. Not only did we receive numerous task suggestions, but almost all of those who made suggestions also volunteered to act as mentors for those tasks. Many were also willing to volunteer to mentor other tasks.

The Contest

On 21st November 2011, at 08:00 UTC, the contest started. Students were able to see the tasks and start claiming them. We set up an IRC channel for mentors and students with the idea that there would always be someone around to help a student who needed a little advice.

With students aged between 13 and 17 it was clear that many aspects of software development would be new to them and it soon became apparent that many would need assistance in working with git and github, or in understanding that they needed to edit a template rather than the resulting HTML. Fortunately, we were blessed with many patient mentors who were willing to spend their time assisting the students.

Mentoring

One of the GCI rules was that students could not work on more than one task at a time. That meant that as soon as a student submitted a task they were keen to get it accepted or to find out what was wrong so that they could either fix it or move on to their next task. Our mentors were, of course, all volunteers, with jobs that took priority, who might be in different time zones to the student and, perhaps, in a few cases, might even have a life.

We had a group of 50 mentors who went beyond the call of duty in trying to aid all the students in a timely fashion, but there were still occasional instances in which students had to wait a bit longer than they would have liked to get feedback.

Translations

We noticed a trend quite early on that the translation tasks were very popular. It's understandable why that is. If you are fluent in two languages the translations can be relatively easy tasks. Wesley is bilingual and was also drawn to those tasks. This caused difficulties in some organisations where students sometimes provided translations that weren't particularly good, but the organisation didn't have mentors fluent in those languages to be able to notice that. We ensured that we always had mentors fluent in the languages for which we had translation tasks,

and if we couldn't provide such mentors we didn't offer those tasks.

One of the more difficult aspects of the translation tasks was that whereas we had a pool of 50 mentors, the majority of whom were qualified to mentor on the majority of the tasks, only those fluent in both English and the language of the translation were qualified to mentor the translation tasks. In general we had two or three mentors per language, and it turned out that in many cases the translation tasks required much more mentor input than we had expected.

Results

In total we had 89 students complete at least one task. 162 tasks were completed in total out of 392 which were available.

From the Perspective of a Student

With a large interest in programming amongst young people, learning how to teach them about Perl and the community around it is essential to help to pass on Perl to the next generation.

We are already trying to do this by participating in projects such as GCI and GSoC, but what else can we do to try and encourage more people to participate in Open Source projects in general and Perl in particular?

GCI and GSoC do a very good job at encouraging students to get involved in the open source community. Using IRC is one of the greatest helps in my experience, because not only are the project mentors often online, but so are other members of the community who are frequently willing and able to help.

Tutorials

One of the things that helped me getting started with Perl was watching Gabor Szabo's videos on YouTube. They are explained very well and provide good information for people who are new to Perl.

For many young people an easy introduction is very important. When things start becoming more complex and harder to follow many lose interest and decide to do something else. It is important to make sure that any introduction is easy to follow.

Moving to Perl

Many people also learn to program in languages such as VB. That is the language that is taught at my college and despite my

dislike for it and programming in Access, there's nothing I can do about it. A great way to make Perl more relevant for younger people is to provide 'stepping stones' from more common languages such as VB.

The reason many people use Perl is because they think it is best suited for the work they want to do. If we know these reasons and are able to share them, we should do so. Teaching younger people about the advantages of Perl could encourage them to use Perl sometime in the future if not straight away.

In order to pass on our knowledge of Perl to the 'next generation', we need to teach them what makes Perl the most suitable language for our work and how to start using it. Once these things are done it should be very easy to help and encourage more people to use Perl.

Conclusion

In order for Perl to remain a viable language for new development it requires a critical mass of users. Encouraging young people to consider Perl and providing ways to allow them to do that is essential to ensure the ongoing success of the language.

GCI wasn't perfect. The programme is still young and it's evolving and improving. It required a great deal of work on our part, but I think that overall our participation in GCI this year can be counted a success. We may not have completed as many of the technical tasks as we would have liked to, but every task exposed a student to some aspect of Perl. Even the translation tasks required students to understand the subject matter and become familiar with version control at a minimum.

Our 50 mentors deserve the utmost praise for their selflessness, willingness, skill, patience and tact. These qualities in such abundance amongst so many makes it a joy and privilege to count myself a member of the Perl community. Without their efforts we would never have been able to enter the programme, let alone complete it. It's hard to over emphasise the commitment and effort put in by so many of the mentors.

I trust that our students found the programme beneficial; many of them said as much. We may find some of them contributing to Perl directly in the future. Most will likely contribute more indirectly by sharing a positive experience with Perl, by learning how to be a part of a successful project or by generally enhancing the Open Source world in general. In short, and without wishing to appear melodramatic, I believe that such an investment in the future is in some small way helping to make the world a better place.



For an open position based at our headquarters in Hamburg, we are looking for a

Software Architect

We are looking for an experienced software engineer to enhance our abilities to design and implement the core infrastructure on which our agile software development teams build new products, or add new functionality. We expect you to bring in your expertise to find innovative solutions for sometimes complex technical challenges. As a member of the architecture team, you will be responsible to ensure the alignment of new components with our overall architecture.

Summary of Key Responsibilities:

- Review and improve existing architecture/infrastructure.
- Evaluate and recommend new technology/products to improve the existing ecosystem.
- Provide architecture consulting for development projects.
- Assume responsibility for scalability/availability and performance.
- Implement robust, highly scalable, highly optimized infrastructure components.
- Maintenance and improvement of the existing code base.

Required Knowledge, Skills and Abilities:

- You have experience in web application architecture (scalability, availability, APIs).
- You have experience designing and implementing distributed systems.
- You have programmed in several languages (Perl, Ruby, Java, C, C++, JavaScript)
- You are passionate about and excel in solving challenging problems.
- You are a team player and enjoy sharing knowledge. You are fluent in English.

Plus

- Contribution in open source projects
- Experience with messaging systems
- Experience with mobile applications

What we offer:

- Working on a high availability, high performance, high traffic website with more than 12 million users
- Working on large, distributed, asynchronous architectures
- Team centric software development (pair programming)
- We use Open Source software wherever possible and we contribute back
- Central location in Hamburg, near the beautiful Alster
- Free fruits and beverages

TIP: You can read our [XING Blog](#) to gain an insight into our corporate culture, our values and vision.

XING AG, Gänsemarkt 43, 20354 Hamburg, www.xing.com



Author: Sawyer X (sawyerx@cpan.org)

Asynchronous programming, while seems tricky, once understood, is a useful and comfortable programming style that offers parallelism and separation of concerns cleanly.

We'll start with explaining callback operations, how they work and why they're super cool. From there move on to event loops and from there to actual working code.

Reflex, POE, IO::Async and AnyEvent will be showcased.

Bio Sawyer X

Sawyer X is a systems administrator and Perl developer, involved in various projects (most notably *Dancer*). He taught a Perl course, worked on Perl/Android, lectures often, and ambles occasionally on his blog.

Abstract

Asynchronous programming is a useful way of getting multiple things to run at the same time (for different values of *same time*), which is becoming more and more important.

Asynchronous programming also gives us an insight into understanding a way of separating our concerns and interests, creating applications that have hookable entries and decouple different considerations properly.

Callbacks

Callbacks are a way to indicate code that will run in certain events, but doesn't require asynchronous programming to be used and be useful. However, they are a bit hard for people to understand.

The concept is simple. Think of speaking to your parents or friends on the phone, and they need help using a website. *So, uh.. what's the URL?* and you tell them and they go to the page and then they ask *What do I click now?* and you tell them and now they ask *And what do I look for now?* and you start getting angry because they keep asking you what to do every single time. Also, you keep having to wait on the line, but that's a different kind of angry which we'll get to later.

Callbacks would be you telling your parent/friend what to do in certain cases that occur. For example, telling them to go to a certain URL and once they reach it, to click on a certain button and once they do that, to look for a certain title in the page.

This seems a bit trivial, because that's what programming really is, isn't it?

```
$mech->get($url);
$mech->click('button');
if ( $mech->content =~ /title/ ) {
    ...
}
```

So what's the difference? The difference would be to set it all up in advance.

```
$mech->get( $url, sub {
    $mech->click( 'button', sub {
        if ( $mech->content =~ /title/ ) {...}
    } )
} );
```

You might be asking yourself what's the gain of it all. The gain is multi-fold:

- Running it in a different scope

When we set up the code as a code reference, we enable it to run from inside the module we're using (in this case, `WWW::Mechanize`) instead of from our code. This means that `$mech` can make decisions and then run our code once it reached a certain decision (like a success or failure).

This means we can allow others to make the decision for us, perhaps a wiser, more correct decision.

This also means that we can provide opaque code that will get run without the runner knowing or caring what they run. This is very crucial to understand, because it allows us to write code that can be used in very flexible ways.

Sometimes this is much easier than returning information, such as when you have a lot of information to return (success, fail, reason, additional information, etc.). You can either do this using mutable global-like object attributes (the way *Mechanize* does it) or you can return a structure like an object or hash or array - or you can simply provide events that will run code for the user whenever each situation occurs and send it the additional information.

This is extremely useful when we want to make use of the next point of gain:

- Running it in the background

Imagine that conversation again with your parents/friends. Wouldn't it be great if you could tell them to just call you back when something happens instead of holding on the line with them while they slowly move their mouse and type one agonizing letter after the other, waiting for stuff to load while you feel your life being drained away? Yes, it would be great!

This is what event loops provide. They allow you to run stuff in the background and in order to make use of that, you need to have the code able to not return anything. In order to do that, we'll need to provide as many instructions upfront as we can.

Callback example

```
package MyModule;
sub something {
    my $code = shift;
    my $reply = check_stuff();
    if ( $reply eq 'good' ) {
        $code->($reply);
    }
}

package main;
something( sub {
    my $reply = shift;
    print "Good reply is $reply\n";
} );
```

Event loops

Event loops run just that: a loop. That loop keeps checking for events that need to be run, and runs them. It also provides you with a way to register new events in the loop.

When we register new events, we provide code to run. We can start a new user agent and ask to make an HTTP request, and give it a callback of code to run once it gets a reply. In that code, we can receive the headers and make decisions regarding the reply.

Examples

In the lecture you will have examples of several event loops.

That's it for now. :)


Author: John Scoles (byterock@hotmail.com)

 **Mojolicious**

intro to
Authentication and Authorization

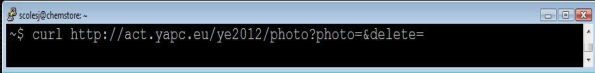
Not part of the framework

Plugins Mojolicious-Plugin-Authentication
 Mojolicious-Plugin-Authorization

 **authentication**


You are who you say you are!

Think Like a hacker!!



So many ways to spoil your day

POST/GET
AJAX
RESTful
PUT/DELETE

 **Mojolicious-Plugin-Authentication**

DB file LDAP Apache '*' Whatever

load_user

```
sub {  
  my ($app, $uid) = @_;  
  ...  
  return $user;  
}
```

validate_user

```
sub {  
  my ($app, $username, $password, $extradata) = @_;  
  ...  
  return $uid;  
}
```


mojolicious

/auth_1.pl

```

plugin 'authentication', {
    autoload_user => 1,
    validate_user => sub {
        my $self = shift;
        my $un = shift || '';
        my $pw = shift || '';
        my $extra = shift || {};
        use Authen::Simple::DBI;
        my $authDB = Authen::Simple::DBI->new(
            dsn => 'DBI:Oracle:XE',
            username => 'HR',
            password => 'HR',
            statement => 'SELECT password FROM
                brain_slug WHERE slug_id = ?');

        return $un
            if ($authDB->authenticate($un,$pw));
        return undef;
    };
};

```

mojolicious

/auth_1.pl

```

plugin 'authentication', {
    autoload_user => 1,
    load_user => sub {
        my $self = shift;
        my $uid = shift;
        return $uid;
    },
    validate_user => sub {
    ...

```

mojolicious

/auth_1.pl

authenticate

```

post '/logon' => sub {
    my $self = shift;
    if ($self->authenticate($self->param('id')
        , $self->param('pw'),
        , {brain_slug => $self->param('slug')}) {
        $self->render('home');
    }
    else {
        $self->render('index');
    }
};

```

mojolicious  /auth_2.pl

is_user_authenticated

```
get '/home' => sub {
  my $self = shift;
  $self->redirect_to('welcome')
  unless($self->is_user_authenticated());
};
```

logout

```
get '/welcome' => sub {
  my $self = shift;
  $self->logout();
  $self->render('index');
};
```

mojolicious  /auth_3.pl

current_user

```
load_user => sub {
  my $self = shift;
  my $uid = shift;
  my ($first,$last);
  given ($uid) {
    when (/fry/) {$first='Philip';$last='Fry'}
    when (/bender/) {$first='Bender';$last='Rodríguez'}
    when (/hermes/) {$first='Hermes';$last='Conrad'}
  };
  return {first=>$first,last =>$last};
},

my $user = $self->current_user();
$self->stash('first' => $user->{'first'});

<%= $first %> says:<BR>
```

mojolicious 

Configuration

```
$app->sessions->default_expiration(3600);

$self->plugin('authentication' => {load_user    => sub { ... },
  validate_user => sub { ... },
  current_user_fn => 'get_slug',
  autoload_user  => 1,
  session_key    => 'brain_slug',
});
```



```
mojolicious /auth_4.pl

Routes - Conditional

get '/home' => sub {
    my $self = shift;
    $self->redirect_to('welcome')
    unless($self->is_user_authenticated());
    ...

get '/home' => (authenticated => 1) => sub {
    ...
}
```

```
mojolicious aa\lib\AA.pm

Full App

plugin 'authentication', {
    ...
};

sub startup {
    my $self = shift;
    ...
    $self->plugin('authentication', {
        ...
    });
}
```

```
mojolicious aa\lib\AA.pm

Routes

my $r = $self->routes;
# Normal route to controller
$r->route('/welcome')->to('example#welcome');

$r->route('/login')->via('get')->to('authn#form')->name('authn_form');
$r->route('/login')->via('post')->to('authn#login');

$r->route('/home')->over(authenticated => 1)->to('example#home')->name('home');
```

mojolicious

aa\lib\AA.pm

Routes

```
my $r = $self->routes;
# Normal route to controller
$r->route('/welcome')->to('example#welcome');

$r->route('/login')->via('get')->to('authn#form')->name('authn_form');

$r->route('/login')->via('post')->to('authn#login');

$r->route('/home')->over(authenticated => 1)->to('example#home')->name('home');
```

mojolicious

aa\lib\AA\Example.pm

Controller

```
sub home {
    my $self = shift;
    my $user = $self->current_user();
    $self->stash( first => $user->{'first'});
}
```

mojolicious

aa\lib\AA\Authn.pm

New Controller

```
package AA::Authn;
use Mojo::Base 'Mojolicious::Controller';

sub form {
    my $self = shift;
}

sub login {
    my $self = shift;
    if ($self->authenticate($self->param('id')
                          , $self->param('pw'),
                          , {brain_slug => $self->param('slug')})) {

        $self->redirect_to('home');
    }
    else {
        $self->redirect_to('authn_form');
    }
}
}
```



```
mojolicious aa\lib\AA.pm

Even More Routes

$r->route('/home')->over(authenticated => 1)->to('example#home')->name('home');
$r->route('/slug')->over(authenticated => 1)->to('example#slug')->name('slug');
...
$r->route('/cat')->over(authenticated => 1)->to('example#cat')->name('cat');
...
$r->route('/dog')->over(authenticated => 1)->to('example#dog')->name('dog');
```

```
mojolicious aa\lib\AA.pm

Bridge

$r->route('/home')->over(authenticated => 1)->to('example#home')->name('home');

my $rn = $r->bridge->to('authn#check');

$rn->route('/home')->to('example#home')->name('home');
$rn->route('/slug')->to('example#slug')->name('slug');
$r->route('/logout')->to('authn#delete')->name('authn_delete');
```

```
mojolicious aa\lib\AA\Authn.pm

Controller

sub check {
    my $self = shift;
    $self->redirect_to('authn_form') and return 0
    unless($self->is_user_authenticated());
    return 1;
}

sub delete {
    my $self = shift;
    $self->logout();
    $self->redirect_to('authn_form');
}
```


mojolicious

aa\templates\lib\slug.html.ep

On a Template

```
% layout 'default';
% title 'Slug';
```

```
My Slug<BR>
```

```
Make all hosts go to the brain slug planet!
```

```
% my $user = $self->current_user();
<BR>
```

```
My Host is <%= $user->{last} %>,<%= $user->{first} %>
```

```
<BR>Click <a href="/logout" >here</a> to log out!
```

mojolicious

Authorization

What you can or cannot do!

Identify trust boundaries

Consider granularity

White-List

Black-List

Declarative

Programmatic

Role-Based Access Control (RBAC)

```

Can Play      Has      Needs
User----->  Role-----> Privilege <-----Action

```

mojolicious

Mojolicious-Plugin-Authorization

```
has_priv
```

```
sub {
    my ($app, $privilege,$extradata) = @_;
    ...
    return 1|0;
}
```

```
is_role
```

```
sub {
    my ($app, $role, $extradata) = @_;
    ...
    return 1|0;
}
```

mojolicious Mojolicious-Plugin-Authorization

privileges

```
sub {
    my ($app,$extradata) = @_;
    ...
    return $privileges[@privileges]%privileges;
}
```

role

```
sub {
    my ($app,$extradata) = @_;
    ...
    return $role[@role]%role;
}
```

mojolicious az\lib\AA\Abstract\Profile.pm

New Class

```
package Abstract::Profile;
...

my $self = shift;
my ($opt) = @_;
my $dbh = DBI->connect('dbi:Oracle:XE','hr','hr');

my $sth = $dbh->prepare('select first_name,
                             middle_name,
                             last_name,
                             role
                             from brain_slug
                             where slug_id=p_id');
...

```

mojolicious az\lib\AA\Authn.pm

```
package AA::Authn;
use Mojo::Base 'Mojolicious::Controller';
use Abstract::Profile;
our %LOGGED_IN = ();
...

if ($self->authenticate($self->param('id')
                      , $self->param('pw')
                      , {brain_slug => $self->param('slug')})) {
    my $profile = Abstract::Profile->new({id => $self->param('id')});
    $LOGGED_IN{$profile->id()} = $profile;
    $self->redirect_to('home');
}
else {
    ...
}
```



```

mojolicious az\lib\AA\Authn.pm

package AA::Authn;
use Mojo::Base 'Mojolicious::Controller';
use Authen::Users;
our %LOGGED_IN = ();
...

if ($self->authenticate($self->param('id')
                      , $self->param('pw')
                      , {brain_slug => $self->param('slug')})) {

    my $authn = Authen::Users->new({dbtype=>'SQLite', dbname=>'slug'});
    my $profile = $authn->user_info($self->param('id'));
    $LOGGED_IN{$profile->user()} = $profile;
    $self->redirect_to('home');

}
else {
...

```

```

mojolicious az\lib\AA\Authn.pm

sub check {
    my $self = shift;
    if ($self->is_user_authenticated() and
        $LOGGED_IN{$self->session('auth_data')}){
        return 1;
    }
    $self->logout();
    $self->redirect_to('authn_form');
    return 0;
}

```

```

mojolicious az\lib\AA\Authn.pm

sub delete {
    my $self = shift;

    $LOGGED_IN{$self->session('auth_data')}=undef;
    $self->logout();
    $self->redirect_to('authn_form');

}

```

```
mojolicious aa\lib\AA.pm

sub startup {
    my $self = shift;
    ...
    $self->plugin('authorization', {
        has_priv => sub {
            my $self = shift;
            my ($priv, $extradata) = @_;
            my $user = $self->current_user();
            return $user->can_i($priv);
        },
        is_role => sub {
            my $self = shift;
            my ($role, $extradata) = @_;
            my $user = $self->current_user();
            return 1
                if ($user->role() eq $role);
            return 0;
        },
        ...
    });
}
```

```
mojolicious aa\lib\AA.pm

sub startup {
    my $self = shift;
    ...
    $self->plugin('authorization', {
        ...
        user_privs => sub {
            my $self = shift;
            my ($extradata) = @_;
            my $user = $self->current_user();
            return keys(%{$user->can()});
        },
        user_role => sub {
            my $self = shift;
            my ($extradata) = @_;
            my $user = $self->current_user();
            return $user->role();
        },
        ...
    });
}
```

```
mojolicious aa\lib\AA.pm

sub startup {
    my $self = shift;
    $self->plugin('authentication', {
        autoload_user=>1,
        load_user => sub {
            my $self = shift;
            my $uid = shift;
            return $AA::Authn::LOGGED_IN{$uid}
                if ($AA::Authn::LOGGED_IN{$uid});
            return $uid;
        },
        ...
    });
}
```



```
mojolicious aa\lib\AA.pm

Conditional Routes

$rn->route('/drop')->over(has_priv =>'detatch')->to('example#drop')->name('drop');

$rn->route('/die')->over(is =>'dead')->to('example#delete')->name('die');

aa\lib\Example.pm

sub delete {
    my $self = shift;
    $self->logout();
    $self->render('example/dead');
}
```

```
mojolicious aa\lib\AA.pm

More Complex Routes

my $feed_only = $rn->route('/feed')->over(has_priv=>'feed')->to('example#feed');

foreach my $size ((qw(little some all))){
    $feed_only->route($size)->to("example#$size");
}
```

```
mojolicious aa\templates\lib\slug.html.ep

On a Template

...

My Slug<BR>

Make all hosts go to the brain slug planet!

<BR>Click <a href="/logout" >here</a> to log out!

% if ($self->has_priv('feed')) {
    Click <a href="/feed/all" >here</A> to feed on a host!
    <BR>
}%}
```


mojolicious  aa/templates/lib/slug.html.ep


```

On a Template
% layout 'default';
% title 'Slug';

% if ($self->is('king')) {
    Click <a href="/control" >here</A> to control all slugs!
    <BR>
%}


My Slug<BR>
...

```

mojolicious 

Resources

<http://mojolicio.us>
<http://mojolicio.us/perldoc/Mojolicious/Lite>
 Mojolicious::Plugin::Authentication
 Mojolicious::Plugin::Authorization
<https://github.com/byterock>

mojolicious 

John Scoles
byterock at hotmail.com

Thanks to;
 Ben van Staveren (madcat)
 Sebastian Riedel (sri)

Author: Lenz Gschwendtner (lenz@springtimesoft.com)

Continuous Deployment is a big topic if you want to push out code to production as fast as possible. The little pitfall is that it is not that straight forward if you want to use a PP approach. We at iWant-MyName came up with a pure perl tool chain all the way from your git repository via integration testing to deployment on your production servers.

Bio Lenz Gschwendtner

Lenz is a geek living at the edge of the world and loves tinkering with code. He co-founded a domain registrar called iWantMyName and is involved in several other early stage startups either directly or as a mentor. Lenz used to live in Germany and Austria but left the northern hemisphere to enjoy his live at the far end of the world.

He is currently playing with CouchDB, RabbitMQ, Redis, Mojolicious and has a keen interest in the lean startup methodology.

Abstract

A fast moving Startup should always try to automate as much as possible and maintain control in form of automated tools that report everything you need to know at any given time.

This includes testing and releasing code!

Githublicious <https://github.com/norbu09/githublicious> progressed out of the desire to have one platform written in perl that replaces Jenkins, Capistrano and a whole raft of hacked crons that all did very specific things.

Githublicious is a Mojolicious application that acts as a post commit endpoint for github and can trigger test runs and deploys. Githublicious also acts as a continuous testing platform and helps to ensure that the whole application stack is in fact working as it should.

It is based on Planet Express Ship <https://github.com/norbu09/planet-express-ship> and uses CouchDB as the data store. Planet Express Ship is a rapid prototyping platform that we developed on top of Mojolicious and CouchDB to quickly write anything from a simple tool to a fully working startup idea. Githublicious brings in Giovanni <https://github.com/norbu09/Giovanni> and TheEye <https://github.com/norbu09/TheEye>, both projects that I use for deployment and monitoring in production with a few of my projects.

Giovanni is used as a replacement for the Ruby based Capistrano and acts in much the same way. We currently deploy Catalyst, Mojolicious and Erlang based applications with Giovanni, either directly or through githublicious.

TheEye is a monitoring system that is based on Perl tests. In fact, any language is fine as long as the output is TAP. We use it with PhantomJS to test javascript heavy pages as well as with plain perl tests. We use TheEye a lot in very different situations from monitoring SSH tunnels to full walk throughs of websites where we tests every button.

In githublicious those tools help with automated deployment and continous testing.

Author: Bernd Ulmann (ulmann@vaxman.de)

Bio Bernd Ulmann

Born on July, 19th, 1970 (the same year the PDP 11 was introduced), studied mathematics and philosophy, dissertation about analog computing, professor for business computer science at the FOM in Frankfurt/Main. Spends most of his spare time collecting and restoring old computers (PDP 11, VAX and of course analog computers), building analog electronic circuits or writing programs in Perl, APL, Lang5 to solve (or create) problems that are ideally of no commercial interest but fun and illustrative to tinker with.

Abstract

A small niche in the programming language ecosystem is inhabited by so called array languages, APL being the most famous of those. *Array programming* is based on a rather unusual paradigm that allows one to solve many problems with next to no loops and much fewer conditionals than with traditional languages. Some of APL's ideas and features already made their way into Perl 6 and via some modules into Perl 5 as well. The following paper gives a short introduction to the basic ideas of array programming and describes the use of `Array::APX` -- a module offering some basic array functionality to Perl 5 users.

Basics

The main idea of array programming is to extend typical scalar operations to work transparently on n -dimensional arrays. This idea dates back to 1957 when Ken Iverson published his seminal book titled *A Programming Language*, (APL for short). What was originally intended as a new mathematical notation turned out to be quite well suited as a programming language, too. Since its inception, APL had strong proponents as well as strong opponents with (more or less good) arguments on either side.

The main reason why APL did not become a mainstream language (There are areas where APL is used quite heavily like in the financial industry.) is its arcane character set. Iverson made use of a plethora of unusual signs that were *a nightmare for [the] typist [...] and impossible to implement with punching and printing equipment currently available*. Although this has become less of a problem with current GUIs it still scares away most programmers since a typical APL program does not look like anything one most probably

has seen before -- in fact, it does not even look like a computer program. The following line is an actual APL program that generates a list of primes (More on this later but then using Perl and `Array::APX`):

$$(\sim R \in R \circ . \times R) / R \leftarrow 1 \downarrow \iota R$$

Obviously the APL notation is kind of weird. Nevertheless it turned out that the main idea of having n -dimensional data structures as the basic items to operate on is very powerful and thus many of today's languages support array programming primitives to at least some degree. Perl 6 already incorporates quite some of this functionality and for Perl 5 there are -- apart from the builtin `map` and `grep` etc. -- packages like `List::Util` that contain basic functions like `reduce` (There is also the *Perl Data Language*, *PDL* for short, <http://www.drdoobbs.com/pdl-the-perl-data-language/184410442> for example.) etc.

During the last couple of years, Mr. Thomas Kratz and the author developed an interpreter for a stack based array language, *Lang5* (Cf. <http://lang5.sf.net>), that is completely implemented in Perl. As a side effect of this, a Perl module, `Array::Deeputils`, was written that contains all of the basic array functionality of Lang5.

After experimenting and working with Lang5 for quite some time, the idea emerged to make the functionality of `Array::Deeputils` available to Perl 5 users in a more natural way which triggered the development of `Array::APX`, the *Array Programming eXtensions* for Perl which will be described in the following.

Array::APX Basics

First of all, `Array::APX` overloads some of the basic Perl operators to work on nested arrays. As an example, let us build the element-wise product of two vectors:

```
use strict;
use warnings;
use Array::APX qw(:all);

# Create two vectors [1 2 3]
# and [4 5 6]:
my $x = iota(3) + 1;
my $y = iota(3) + 3;

# Multiply both vectors and
# print the result:

print $x * $y;
```


This short program first generates two three-element vectors using the `iota` function of `Array::APX` (It is named after `ι` in APL. `iota $value` works like `(0 .. $value - 1)` and returns a blessed reference to the resulting array (The function `dress` allows one to bless any nested array into an APX object, so one could also have written `$x = dress([1, 2, 3])`; instead of `$x = iota(3) + 1`; To get rid of this `dress`, the `strip-Function` can be used.).

All unary and binary operators that are overloaded within `Array::APX` are automatically applied in an elementwise fashion to the operands of the operator. In this case the multiplication is performed on the three elements of both vectors which yields `[3 8 15]` as a result. Table 1 lists all unary and binary operators that `Array::APX` provides.

Operator	Unary	Binary	Special	Description
"			✓	Stringify
+		✓		Addition
*		✓		Multiplication
-	✓	✓		Subtraction / Negation
/		✓	✓	Division / reduce
%		✓		Modulus
**		✓		Exponentiation
&		✓		Bitwise and
		✓	✓	Bitwise or / generalized outer product
^		✓		Bitwise exclusive or
!	✓			Not
==		✓		Comparison equal
!=		✓		Comparison not equal
x			✓	Scan operation

Most of these operators are self-explanatory and will work as described on APX-data structures. Apart from the stringification the first special operator is `/` which implements either a division operation or a reduce operator depending on its context (`/` was chosen to represent the reduce operator since this symbol is used in APL for the same purpose.). Two APX-data structures `$x` and `$y` may be divided by simply writing `$x / $y`. If the left hand side of `/` is a subroutine reference and the operand on the right is an APX-structure, it will be reduced by applying the subroutine between each successive element. Thus computing `$\sum_{i=1}^{100} i$` can be done like this, yielding `5050`:

```
use strict;
use warnings;
use Array::APX qw(:all);

# Create a vector [1 .. 100]:
my $x = iota(100) + 1;

my $adder = sub {$_[0] + $_[1]};
print 'The sum of all elements is ',
      $adder / $x, "\n";
```

The next unusual operator is `|` that implements a generalized outer product if not used as a binary or like this (here a basic multiplication table will be generated) (Being a *generalized* outer product operator, `|` can work with any binary function, not just with a multiplication.):

```
use strict;
use warnings;
use Array::APX qw(:all);

my $x = iota(10) + 1;
my $m = sub {$_[0] * $_[1]};

print $x |$m| $x;
```

The resulting multiplication table looks like this (thanks to the stringification):

```
[
  [ 1  2  3  4  5  6  7  8  9 10 ]
  [ 2  4  6  8 10 12 14 16 18 20 ]
  [ 3  6  9 12 15 18 21 24 27 30 ]
  [ 4  8 12 16 20 24 28 32 36 40 ]
  [ 5 10 15 20 25 30 35 40 45 50 ]
  [ 6 12 18 24 30 36 42 48 54 60 ]
  [ 7 14 21 28 35 42 49 56 63 70 ]
  [ 8 16 24 32 40 48 56 64 72 80 ]
  [ 9 18 27 36 45 54 63 72 81 90 ]
  [10 20 30 40 50 60 70 80 90 100 ]
]
```

The last special operator to be described is `scan`, denoted by `x.` `Scan` applies a function to the elements of a vector but does not reduce the vector to a scalar. The following program example shows this mechanism. It generates a vector of the first ten squares by computing partial sums of a vector `[1 3 5 7 9 11 13 15 17 19]`, yielding 1 as the first result vector element, `1 + 3` as the second, `1 + 3 + 5` as the third and so on:

```
use strict;
use warnings;
use Array::APX qw(:all);

my $x = iota(10) * 2 + 1;
my $a = sub {$_[0] + $_[1]};

print $a x $x;
```

The result of this is the vector `[1 4 9 16 25 36 49 64 81 100]`.

Shapes

A nested array is characterized by its associated dimension vector, i.e. a vector that represents the number of elements of the array for each of its dimensions. The `rho` function is used to either determine the dimension vector of a data structure or change the shape of a data structure according to a given dimension vector.

Changing the shape (reshaping) will first flatten the structure supplied as the argument and then rearrange the elements to form the desired data structure. If this destination structure needs more elements than the original structure, its elements are reread from the beginning of the flattened list. In the extreme case, `rho` can be used to create a nested structure out of a single element vector that will be repeated over and over to fill the resulting array.

```
use strict;
use warnings;
use Array::APX qw(:all);
```

```
# Create a 3-times-3 matrix out
# of a nine-element vector:
my $vector      = iota(9) + 1;
my $dimensions = dress([3, 3]);
my $matrix      = $vector->rho($dimensions);

print $matrix;

# Create a 2-times-2 matrix and
# determine its dimension vector:
my $my_matrix    =
  dress([[1, 2], [3, 4]]);
my $my_dimension = $my_matrix->rho();

print $my_dimension;
```

This program prints

```
[
  [      1      2      3 ]
  [      4      5      6 ]
  [      7      8      9 ]
]
```

as the result of using `rho` to reshape a structure and `[2,2]` as the result of `rho` being used to determine the structure of `$my_matrix`.

If one needs a 2-times-2-times-2 matrix of 1-elements for example, this could be accomplished by `dress([1]->rho(dress([2, 2, 2])))` making use of the fact that `rho` will repeat the elements of the input data structure as often as necessary to fill the output structure.

Sometimes it is necessary to flatten a nested data structure -- although this could be accomplished using `rho` with a one element dimension vector, this can be done more easily using `collapse`. Thus `$matrix->collapse()` would return a vector containing all matrix elements.

Selecting elements etc.

`Array::APX` supplies some functions to address, select and otherwise access elements of nested structures. These are in particular the following:

`in:`

This is the set theoretic element-of operation. It tests if elements of a data structure are contained in another data structure and returns a boolean result vector. Its inverse function is

`select:`

This selects elements from a vector controlled by a boolean selection vector.

index:

This function returns an index vector denoting the position of elements of a given data structure in another structure.

subscript:

This function selects elements from a nested datastructure, controlled by an index vector. Its inverse function is

scatter:

Using `scatter` it is possible to place elements at specific positions in a multidimensional data structure.

remove:

Removes elements controlled by an index vector.

reverse:

Reverses the sequence of elements in an APX-structure.

rotate:

Generalized rotation along several axes of a multidimensional data structure.

slice:

Extracts *slices* from a data structure. The only restriction is that the slice being extracted can not be of a higher dimension than the source data structure.

transpose:

Transposes a structure along any of its axes.

Examples

Remember the arcane APL example from the very beginning? It computes a list of prime numbers up to a value stored in a variable `R`. The basic idea which is very elegant but quite resource intensive works like this:

1. Create a vector with unit stride running from 1 to `R`.
2. Remove the first vector element, so the resulting vector has the form `(2 .. R)`.
3. Create a multiplication table using two of these vectors with the multiplication as basis operation. This table has the special property that it does not contain any prime numbers since all of its entries are the product of at least two primes.

4. Use the set-theoretic `in` function to determine which elements of the vector are *not* contained in the matrix. These elements are the prime numbers we are looking for.

5. Using the inverted result of the `in` operation, select the prime elements from the vector.

Using `Array::APX` this can be implemented as follows to generate a list of prime numbers between 2 and 199:

```
use strict;
use warnings;
use Array::APX qw(:all);

my $f = sub { $_[0] * $_[1] };
# We need an outer product
my $x;

print $x->select(!($x = iota(199) +2)
->in($x |$f| $x));
```

The first two steps of the algorithm described above are accomplished with `$x=iota(199) + 2`. The resulting `$x` is then used to create the multiplication table with `$x |$f| $x`. Using `in`, a boolean selection vector is created which is inverted using `!` and used to `select` the prime number elements from `$x`.

As elegant as this method is, it is quite resource intensive since a square matrix is needed which contains a lot of products which are not necessary to generate the list of primes. Although this matrix can be shortened a bit (Restricting the matrix to run from 2 to `sqrt($x)` would not work! Why?),

it would be (a bit) faster to perform trial divisions up to `sqrt($x)`.

The following program checks if a given number `$x` is prime by

1. creating a vector of the form `[$x $x ..$x]` containing `sqrt($x)` elements and then
2. dividing this vector by the vector `[2 3... sqrt($x)]`.
3. A prime number will only be divisible by 1. Thus reducing the result vector of the `%`-operation followed by `== 0` must yield 1 for a prime number.
4. Implementing this using `Array::APX` yields a program snippet like this:

```
my $limit = int(sqrt($x));
my $adder = sub{$_[0] + $_[1]};

my $is_prime =
  $adder / (
    # Create [$x $x $x...]
  )
```

```
dress([$x])->rho(dress([$limit]))
%           # modulus
# [1 2 3 ... sqrt($x)].
(iota($limit - 1) + 2)
== 0       # Where was it divisible?
) == 1;
```

Conclusion

Using `Array::APX` it is possible to explore the ideas of array programming within the comforting environment of Perl 5. It has turned out to be of great value in education since today's students are already accustomed to languages like Perl (Early attempts using a real APL interpreter in class were not that successful since the students first had to learn the notation. Then they had to

memorize the keyboard layout and adapt to the APL environment. The typical class schedule just did not leave enough room to accomplish that.).

Bibliography

A Source Book in APL, APL PRESS, Palo Alto, 1981

Ken E. Iverson, *Formalism in Programming Languages*, in [1] [pp. 17 -- 25]

Ken E. Iverson, *Notation as a Tool of Thought* in [1][pp. 105 -- 128]



Work with Sipwise

We develop and integrate Open Source next generation communication systems for all kinds of access networks. Our main customers are small, medium and large network operators all over the world. We're always looking for highly motivated and skilled engineers to increase our team of developers and operations engineers.

Currently we're searching for

Perl and Web Developers

for our development team in Vienna / Austria. If you're interested to join a dedicated team in a professional start-up environment, check out our job offerings at

www.sipwise.com

Author: Nuno Ramos Carvalho (smash@cpan.org), Alberto Simões (ambs@cpan.org), José João Almeida (jj@di.uminho.pt)

Bio Nuno Ramos Carvalho

Nuno Ramos Carvalho is currently doing a PhD in computer science at University of Minho. He is currently also president of the Portuguese Association for Perl Programmers, and has been using and advocating Perl for almost ten years.

Bio Alberto Simões

Alberto Simões has used Perl for more than ten years. He is the maintainer of more than a dozen Perl modules and one of the five Dancer core developers. He works as a computer science teacher, and has a PhD in Natural Language Processing.

Bio José João Almeida

José João Almeida is currently an assistant professor at University of Minho, and is a lecturer in many computer science courses. He has been using and teaching Perl for more than ten years, he also develops and maintains a wide range of Perl modules and tools.

Abstract

An ontology is a formalism that can be used to represent knowledge, describing information as concepts and relations between these concepts. The adoption of these formalisms has increased in the last years, and are used in many fields like artificial intelligence and software engineering, or in the semantic web. Mainly in situations where relational databases (or more recently no-SQL solutions) may not be the most fit approach to store and manipulate data.

This article introduces how ontologies can be exploited to create rich applications using Perl. How we can store information using ontologies, retrieve and update information, infer new knowledge, and produce any arbitrary side effect. This is achieved by embedding small snippets of code written in a domain specific language inside Perl programs.

Introduction

Many flavors of programming languages and paradigms are available today. Before writing a program, one valid and pertinent question is

which programming language to choose or paradigm to use. Most of the times a general purpose programming language is adopted, like C/C++, Perl or Java. These languages are general purpose in the sense that they were not designed to solve problems in specific domains and can, theoretically, be used to perform any task.

This is a positive attribute, but what can quickly become obvious is that since these languages are so general and act a bit as a swiss army knife, they quickly become inefficient for some specific problems. Inefficient not only performance wise but also linguistic wise, i.e. a lot of complex code is required to perform a rather simple task in a specific domain. To overcome this inefficiency, Domain Specific Languages (DSL) are usually used. These languages are designed and optimized to solve particular sets of problems, meaning that writing programs to solve real world problems can be difficult or even impossible.

A good solution would combine both approaches, programs could be in their grand majority written using a general purpose programming language, taking advantage of their usual tools and libraries, and glued together small snippets of code written in a well defined set of (small) DSL to perform tasks in very specific domains. This would ensure language efficiency throughout the code. But the compiler would have to cope with different languages in the same program, which means that probably different parsers will be needed, different intermediate representations will be used, or even different compilation workflows would be required. Some well known problems emerge when several programming languages are mixed together in a single program, for example: parsers need to handle more than one language efficiently. This article advocates a simple approach for language weaving, it introduces a domain specific language to manipulate ontologies, and illustrates how the combination of this DSL with Perl allows writing elegant and easy to maintain ontology aware applications that can be used to solve real world problems.

Ontology Manipulation Language (OML) is a DSL that can be used to describe operations that manipulate information represented in an ontology. There are many different ways to define and represent an ontology, this work assumes a rather simplistic definition: a set of terms and relations between these terms.

More details about this can be found in. OML is a rule-based programming language, a program is a set of rules that are executed in order. Each rule uses as its left hand side a pattern that should be searched in the ontology, and as its right hand side, an action to be performed. Patterns describe information, terms, relations between terms, or any combination of these that can be found in the ontology. Actions describe the arbitrary operations that are to be executed when patterns are found, this includes not only operations on the ontology itself (changing its terms, relations, etc.), but also producing any arbitrary side effect.

Although many interesting programs can be written using only OML, real applications can be hard or impossible to implement. Given that this language was designed to solve problems in a very specific domain -- ontologies -- it does not provide any syntax or methods to perform many simple tasks required to solve real world problems.

This was the main motivation for weaving OML snippets in Perl programs, taking advantage of Perl's typical tools and libraries, but allowing the user to describe operations elated with ontologies in OML.

To better explain the OML language and its weaving into Perl, starts by introducing the OML syntax and illustrating some simple examples, and gives a brief overview about the OML compiler. This allows the reader to understand how the DSL works, and therefore, be able to better follow the discussion about its weaving into the Perl programming language. Examples will present some simple tools that were built based on

the ability to mix these two languages and finally, in Conclusion we will confer about OML relevance, and the advantages that arose from its weaving with a general purpose programming language.

The OML Language

This section briefly describes OML, it's syntax and how semantic actions can be defined to produce different type of results. OML is a simple language. One of the major goals during its design was to make sure that it would be easy and intuitive to learn and use, even for people without any programming skills background.

In a nutshell, OML programs are a sequence of statements which are evaluated in order. Each statement consists in a `pattern` block, everything on the left side of the *fat-arrow* operator (\rightarrow), and an `action` block, everything on the right side. Each statement always ends with a single dot (`.`), as shown here:

```
pattern → action ~ .
```

Patterns

Patterns are used to describe knowledge that exists in the ontology, they are used to represent simple collections of terms or relations between terms, or even combinations of these. If the pattern matches the ontology the corresponding action block is executed. Table 1 illustrates some patterns that illustrate what can be used.

#	Pattern	
1	<code>term(Buster)</code>	term <i>Buster</i>
2	<code>rel(ISA)</code>	relation <i>ISA</i>
3	<code>term(\$t)</code>	for all terms
4	<code>rel(\$r)</code>	for all relations
5	<code>Buster ISA cat</code>	a specific relation
6	<code>\$pet ISA cat</code>	any pet that <i>is a cat</i>
7	<code>\$pet ISA \$animal</code>	any two terms related by <i>ISA</i>
8	<code>Buster \$rel \$term</code>	for all related do <i>Buster</i>
9	<code>Buster ISA cat AND Twitty ISA bird</code>	<i>AND</i> operator
10	<code>Buster ISA cat OR Twitty ISA bird</code>	<i>OR</i> operator
11	<code>\$c ISA cat AND \$b ISA bird</code>	

The most simple pattern that can be defined is a single term or a single relation. Pattern 1 shown in Table 1 will evaluate as found if there is a term in the ontology named `Buster`. A single relation is shown in in Table 1, Pattern 2, this pattern will evaluate as found if there is at least one relation named `ISA` in the ontology. Variables can be used instead of terms, or relations names. So, Pattern 3 shown in Table 1 describes all the terms in the ontology, and Pattern 4 represents all the relations. Of course that more interesting would be to describe facts, relations between terms, a very simple example of a pattern that describes a fact is Pattern 5 in Table 1. This pattern is considered found if the term `Buster` and the term `cat` are linked by a relation named `ISA`.

Variable containers can also be used in patterns, which means that the pattern can be found more than once for a given ontology. Pattern 6 in Table 1 is one possible example. This pattern represents all the facts that relate the term `cat` with any other term by a relation named `ISA`. Another example of using variable containers is Pattern 7 in Table 1. This pattern represents all the possible combinations of facts that relate terms with the `ISA`-relation.

Patterns can be grouped together using the binary operators `AND` and `OR`, which have their traditional meaning. Patterns paired with the `AND` operator will be evaluated as found if both patterns are found, and if they are paired with the `OR` operator only one needs to be found in the ontology for the pattern be evaluated as found. Patterns 9, 10 and 11 in Table 1 illustrate this.

Actions

After being able to specify the patterns we are looking for in the ontology we need to describe the operations that are going to be executed when the pattern is actually found. Any number of operations can be executed in an action block. Operations are executed in order and can be one of the following types:

- an operation from the predefined list of operations available, this is typically used to add or change the current knowledge on the ontology, for example adding or removing terms or relations;

- or we choose to define our own operation, and write the complete code, this is typically used to produce any arbitrary side effect, updating a data base, printing, creating a PDF file or anything else.

An example, of using a predefined operation can be:

```
add(Buster ISA Mammal)
```

This adds new information to the ontology, specifically relating the term `Buster` with the term `Mammal` using the relation `ISA`. Variables found in the pattern can also be used in the action side of any statement, having their values instantiated according to the pattern found, which means we can write an action block that looks something like:

```
add($pet ISA Mammal)
```

As advertised before we can also produce any side effect, by executing any arbitrary action, for example:

```
sub {
  print "Found a term: " . $name;
}
```

The `sub` keyword has a special meaning in OML, it means that the following action block is a user defined operation and that needs to be executed as is. At the current time this block needs to be written in the programming language Perl, so this works pretty much as an anonymous function that is called later by the Perl interpreter executing the code. Remember that any side effect can be produced using this approach (anything that can be implemented in Perl at least), for example adding information to a relational database:

```
sub {
  $db->execute(
    "INSERT INTO terms (name)" .
    "VALUES ($term)"
  );
}
```

Rules

Putting everything together we can write statements (rules) that look like:

```
$city CAPITAL $country =>
  add($city ISA city)
  add($country ISA country).
```

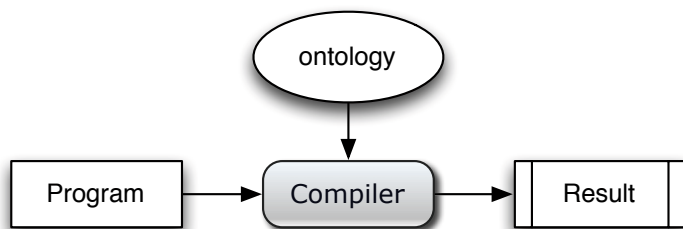
This statement says that for every two terms linked by a relation named `CAPITAL` add two new relations linking the first term with the term `city` by a re-

lation `ISA`, and the second term with the term `country` also by a `ISA` relation. Imagine a geographical ontology describing information about cities and countries, in more loosen English this statement reads: *for every city which is a capital, add a fact stating that city is a city, and country is a country.*

This is just a brief overview of what can be written in OML, a more exhaustive and complete introduction to the language can be found in my thesis.

The OML Compiler

The OML stand-alone compiler (or probably, more correctly, interpreter) takes as input a program written in OML and an ontology, and produces an arbitrary result, as illustrated in Figure 1 (left). Keep in mind that in this section we are talking about the OML specific compiler, which we feed a program completely written in OML.



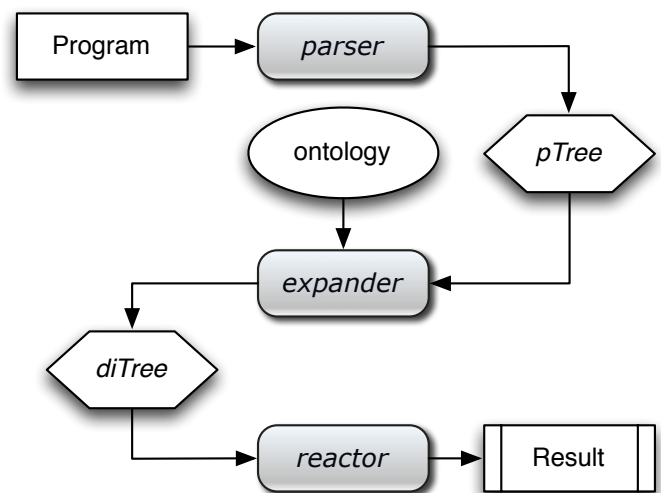
Besides the OML program itself, an ontology is also required as input. Currently there is only one backend for ontology processing available, this implies the use of ISO 2788 to represent the ontology. Of course other backends can be implemented to allow ontologies stored in other formats. The compiler interprets the rules in the program, tries to match the patterns described against the ontology, and acts accordingly with found matches and described actions producing an arbitrary result. Executing a program is divided in three main stages illustrated in Figure 1 (right):

The Parsing Stage In this stage the *parser* is responsible for analyzing the source program written in OML, and create a parsing tree (*pTree*). This tree contains the same information that is in the source program but in a more structured way.

The Expanding Stage After creating a *pTree* the control is handled to the *expander* engine, which is responsible for trying to match the patterns described in the *pTree*. The matching results, and all possible instances of the pattern in case variable containers are used, are stored in a *Domain Instantiated Tree* (*diTree*).

The Reaction Stage Finally, the *reaction* engine is responsible for actually executing the actions described in the initial program. This engine uses the *diTree* to instantiate the variables found in the action blocks for each statement, and executes the corresponding execution block for every match found.

Although this compiler can be used by itself, its main purpose is to be used by other tools, like the weaving of OML into Perl described in the next section. This is mainly because real world applications often have to deal with other tasks which are outside of the ontology domain



(updating databases, handling HTTP requests, etc.) that can be hard to implement using only OML.

The current implementation of the OML compiler (still under development) is available on CPAN (<http://search.cpan.org/dist/Biblio-Thesaurus-ModRewrite/>).

Weaving OML in Perl Programs

In this approach we consider one language to act as the hosting language, and the other as the hosted language: the DSL is the hosted language (OML in this specific case), and the general programming language is the hosting language (Perl in this specific case).

Instead of trying to handle several languages parsing and execution flows, we transform code written in the hosted language in a compilation unit that can be recognized and executed by the hosting language. And then use the compiler for this language to run the program, while keeping this transformation hidden from the programmer. Note that we do not completely rewrite the OML code in Perl, we just transform it so that is valid Perl syntax, and the tasks implemented in OML can be called by the Perl compiler.

Using this approach the application can be written in Perl, and use OML to describe the operations that deal with ontologies weaved inside the program. In practice this is done by enclosing OML code between the `OML` and `ENDOML` special tags:

```
#!/usr/bin/perl

# Perl code here
OML fname(<arguments>)

# OML code here
ENDOML
```

In this example a function called `fname` is available at runtime to be executed anywhere in the Perl program, just like any other function. A program can contain any number of OML blocks required as long each one has a different name. More examples to illustrate this are available in the next section.

Examples

The following program illustrates the weaving mechanism used for OML:

```
use
  Biblio::Thesaurus::ModRewrite::Embed;

my $term = $ARGV[0];
my $ontology =
  thesaurusLoad($ARGV[1]);

printTerms($ontology,$term);

OML printTerms(term)
  begin => sub {
    print "digraph new {"; }.
    term $r $t => sub {
      print "$term -> $t [ label " .
        "= $r ]";
    }.
    $t $r term => sub {
      print "$t -> $term [ label = " .
        ". $r ]"; }.
  end => sub { print "}; }.
ENDOML
```

This program, given an ontology and a term, creates a graph that represents all the relations for the given term, the graph is created using the GraphViz <http://www.graphviz.org/> notation (an image file can be generated from this notation). This is written in Perl, lines nine to twelve are written using OML. The special tags `OML`, and `ENDOML` delimit the code written in OML. Later in the Perl program this OML snippet is called using the OML block name: `printTerms`. Since behind the curtain this block is transformed in a Perl function, this can be called like any other functions, and arguments can be passed like shown in line six. The list of arguments that the OML block receives are defined after the block name, there is a special case of an argument called `ontology` that, in the current implementation, is required to be the first argument and should contain a reference to the ontology where this snippet is to be executed. Different ontologies can be used in different calls to the same OML block. In this specific case there is a second argument `term`, which when the OML is called is replaced by the content of the `$term` scalar when the function is called. Another subtly in this example is the patterns defined in line nine and twelve (`begin` and `end`), these are special patterns that always evaluate as found -- meaning that the corresponding action block will always be executed -- at the beginning and at the end of program execution.

The next example is a simplified version of a Perl program that is used by an online application to answer AJAX queries about points of interest. The information about the locations is stored in an ontology. Perl is handling the tasks for generating HTML headers, and the JSON format of the answer, and OML is handling the query and print of the knowledge stored in the ontology.

```
use CGI;
use JSON;
use Biblio::Thesaurus::ModRewrite::Embed;

my $filter = param('FILTER') or 'ANY';
print header,
  "{ markers: ", get_points($filter),
  " }";

OML get_points(filter)
  $point LAT $x AND $point
  LNG $y AND $point ISA $filter
  => sub {
    print to_json(
      {name=>$point, lat=>$x, lng=>$y
    });
  }.
ENDOML
```

Since the UI uses a map that can only display locations that have a latitude and longitude, the query to the ontology uses a pattern that enforces that points have a latitude $\$x$ and a longitude $\$y$. The application also allows filtering the points displayed by type (cities, countries, castles, etc.) this is why the pattern has an extra expression using the `ISA` relation. The final result is a JSON formatted list of points of interest names (including latitude and longitude), that the application then uses to populate the map displayed to the user.

Conclusion

General purpose programming languages are used to write programs that solve arbitrary problems. Since the syntax of these languages is intended to be as general as possible, the process of writing programs to deal with very specific domains can result in huge amounts of complex code, hard to maintain.

A DSL is an excellent approach for devising a language to solve problems in a very specific domain. But a problem arises: when using a DSL, most trivial tasks for a general purpose programming language can be hard to describe.

A possible solution is to write programs in general programming languages and use one or more DSL for describing tasks in specific domains. This approach produces more *linguistic-wise* efficient programs that are written in a mix of several languages.

In this particular case, a simple embedding system was designed for weaving small OML programs in Perl source code, without the need of any change to the Perl compiler itself. This was achieved by adding an extra step to the normal Perl execution workflow that takes a program written both in OML and Perl, and transforms it in a program written entirely in Perl, that can be normally executed by the Perl compiler. This extra step is completely transparent to the user, which simply writes the code and fires up the normal Perl compiler.

This approach provided an elegant and simple way of implementing applications that make use of ontologies, because OML was used to describe the tasks that have to deal with knowledge stored in the ontology. The programs described in previous sections illustrate how to perform operations on ontologies and take advantage of the synergy between the two languages.

Further work is needed mostly on the OML language than in the languages weaving. OML includes methods for most ontology manipulative tasks, but misses some features and needs speed improvements.

Regarding languages weaving, further work would be interesting to analyze how the weaving of well known DSL languages into Perl or other general programming language can be obtained. This includes further developments to the general weaving mechanism described in section "Weaving OML in Perl Programs".

Bibliography

- R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. *Generic Programming*. in Advanced Functional Programming, pages 28-115, Springer, 1999.
- M. Mernik, J. Heering, and T. Sloane. *When and how to develop domain-specific languages*. in ACM Computing Surveys, 37(4):316-344, ACM, 2005.
- T. Kosar, P. M. Lopez, P. A. Barrientos, and M. Mernik. *A preliminary study on various implementation approaches of domain-specific languages*. in Information and Software Technology, 50(5):390-405, Elsevier, April 2008.
- A. van Deursen, P. Klint, and J. Visser. *Domain-specific languages: an annotated bibliography*. in SIGPLAN Not., 35(6):26-36, June 2000.
- N. Carvalho. *OML - Ontology Manipulation Language*. MSc thesis, University of Minho, 2008.
- ISO2788 Documentation. *Guidelines for the establishment and development of monolingual thesauri*. 1986
- A. Simões and J. Almeida. *Library::* -- a toolkit for digital libraries*. in EIPub 2002 - Technologies Interactions, 2002.
- M. J. Dominus. *Higher-order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publishers, 2005

Author: Fabian Zimmermann (fabian.zimmermann@iese.fraunhofer.de)

Bio Fabian Zimmermann

Fabian Zimmermann is working at the Fraunhofer Institute for Experimental Software Engineering (IESE) as a researcher. After studying computer science at the University of Kaiserslautern, Germany, he worked as a programmer at an international company. There, he came into contact with code reviews. After joining Fraunhofer IESE in 2008, he has been working in the area of software quality assurance. He uses Perl mainly for private programming projects.

Abstract

Effective code, i.e., code that follows language-specific idioms and avoids anti-patterns, is an important goal in software development. Although the correct usage of certain idioms can be enforced by static code analysis, some issues can only be detected by humans, e.g., the use of meaningful names. Code reviews can help to improve code quality by detecting violations of these idioms as well as bugs in early phases of development. For efficient code reviews, especially in distributed development, good tool support is recommended. A suitable review tool needs to allow annotating parts of the code with comments and suggestions for improvement. This talk demonstrates how to review Perl code with an Eclipse plugin.

Facts about Software Inspections

High quality is an important issue in software development. High quality software is software that is nearly bug-free, but also easy to read and to maintain. Software inspections (The term software inspections is commonly used more generally for all kinds of software-related reviews. In this paper, it is used as a synonym for the term code review.), also known as code reviews, can help to improve code quality by detecting violations of idioms as well as bugs in early phases of development.

Invented in the 1970s by Michael Fagan, software inspections are one of the oldest software quality assurance techniques. Several studies have proven their efficiency. Especially the fact that bugs can be found before runnable code is created, is a big advantage over testing. Since inspections can find different kinds of bugs, inspections and testing should be performed together. Thus, it is a good choice to integrate both quality assurance techniques.

Inspections can also be combined with static code analysis, such as `Perl::Critics`.

Fagan suggested a very formal inspection process containing different roles and phases. These Roles are author, inspector, organizer, moderator, reader, and recorder. Since we demonstrate a lightweight version of this process, only the mandatory roles author of the code (programmer) and inspector (reviewer), who in the case of code reviews is often another programmer, are important here. We also concentrate on the reviewing phase, i.e., the phase where bugs are found.

Further information about software inspections can be found in the inspection repository (<http://inspection.iese.de>).

Goals of Software Inspections

There are several goals that can be achieved by using software inspections. Some of these goals are listed below:

Detection of Bugs

One of the most important goals of software inspections is the detection of bugs in an early phases of development. Since during detection it is often not clear if a finding is really a bug, we prefer the more neutral term issue.

Maintainable Code

Another important goal is high-quality code, i.e., code that is not only free of bugs, but also easy to read and maintain. To reach high code quality compliance with idioms and patterns should be checked.

Collecting Data

To improve the development process, data about code quality, number of bugs, bug-prone areas, etc. can be collected. Here, it is important to use this data only to improve the code and the development process, but not to blame individual programmers or reviewers for their performance.

Searching for Solutions

Reviews can also be used to find solutions for problems. The reviewers look at a piece of code and search for solutions, e.g., for debugging.

Coaching

Software inspections can also be used to coach less experienced team members. On the one hand, newbies can review the code of experienced programmers to learn what good code looks like.

On the other hand, more experienced programmers can review the newbies' code and tell them how to improve.

Creation of Common Knowledge

If every piece of code is reviewed by someone, many people gain knowledge about different parts of the code basis.

Team Building

One goal of inspections is the creation of high-quality code. Inspections can help to see this as a common goal to which every team member contributes. Thus, inspections can help to improve the team spirit.

Approval for the Author

It is very demotivating to create source code without any feedback. If the only thing that matters is code that runs somehow, programmers tend to produce bad quality and thus become disappointed. Code reviews can help to give some approval for high-quality code to the author.

Technical Solution

To perform code reviews in distributed software development, where programmers cannot easily have a meeting, good tool support is recommended. Since programmers are used to writing and reading code within their IDE, it is a good idea to let them also use this tool for reviewing.

The reviewer needs the possibility to mark suspicious parts within his or her IDE and to annotate these parts with his or her comments. The

se comments should be made available to the author and all other persons committed, i.e. the other programmers. Thus, the criticized parts of the code can be easily altered based on these comments. Additionally, the possibility exists to discuss with the reviewer and to clarify uncertainties should be provided.

Reviewing in Eclipse

Since there are several review plugins available for Eclipse (<http://www.eclipse.org/>), we use this IDE with the EPIC plugin (<http://www.epic-ide.org/>).

Although there are other Eclipse plugins, such as Jupiter (<http://code.google.com/p/jupiter-eclipse-plugin/>), we decided to use AgileReview (<http://www.agilereview.org/>) for our code reviews. The reason was that AgileReview concentrates on the basic features for reviewing, such as commenting and does not prescribe a specific reviewing process.

As an example, a browser game developed in Perl was reviewed. This project, opened in Eclipse in the Perl perspective of EPIC, is shown in Figure 2. If we as reviewers want to comment on parts of the code, we can switch to the AgileReview perspective.

In this perspective (shown in Figure 2), we can mark lines of the code and create a new comment. In this example, the reviewer is not sure, whether the value of `start_population` is correct or not. The comment (shown at 2. in Figure 2) can be assigned to a certain programmer. The corresponding lines are marked, here in yellow

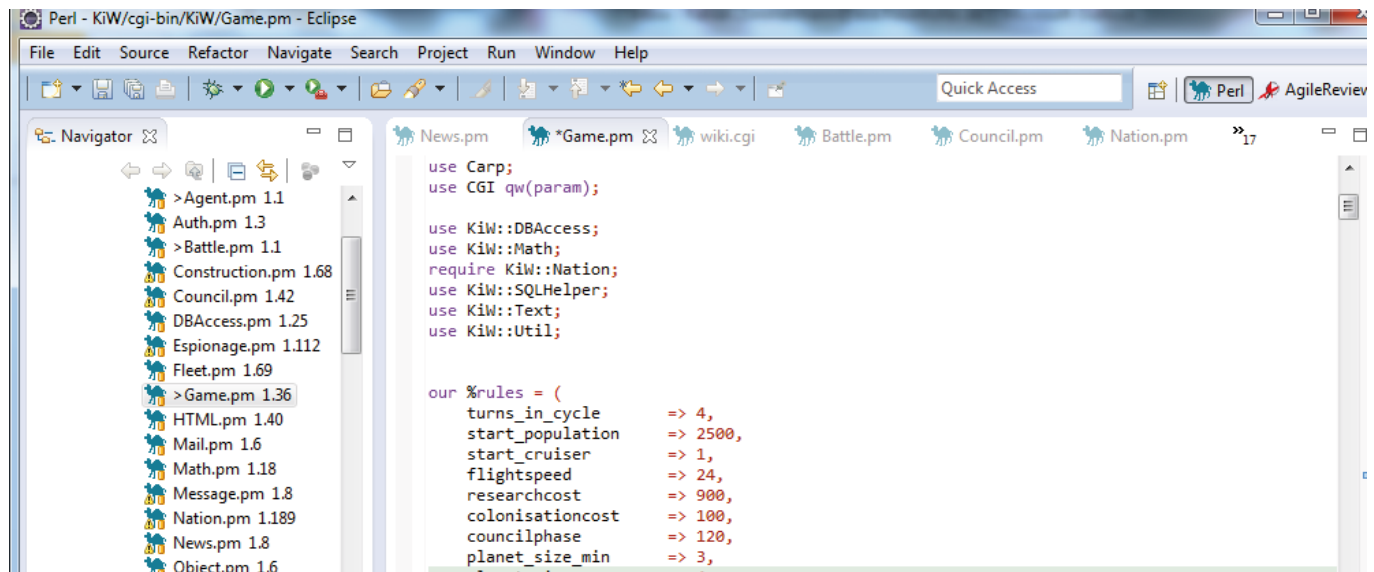


Figure 1: Project opened in Perl perspective

(see 1. in Figure 2). If more than one reviewer is active in the project, each reviewer can choose his or her own color.

Each team member can add a reply to a comment (see 3.). In this example, there is some discussion about the value. All comments from a review, containing several files, can be displayed in a separate view (see 4.). They can be filtered according to several criteria. By clicking on a comment, the corresponding lines are shown again (1.).

All required information from a review is stored in an xml-file. This file can be put into the same version control like the source code and checked out together.

Reaching the Inspection Goals

The next question is which of the inspection goals can be reached by reviewing code with AgileReview. Since AgileReview enables other programmers to review code without much

effort and within their IDE, they can detect many of the bugs. Additionally, they can indicate if idioms and common patterns are violated or code is hard to understand. Thus, they can easily coach each other in writing better code. If other programmers read code and give positive feedback, motivation increases with positive effects on the team spirit. In the end, the quality of the code will improve and the programmers will tend to write code that is easier to maintain. Even though the review data can be used to some extent, its collection and analysis is not in the focus of AgileReview.

Conclusion

Code reviews can be an important step towards improving the quality of software. With Eclipse, EPIC, and AgileReview, a tool chain can be built to easily perform code reviews in projects written in Perl.

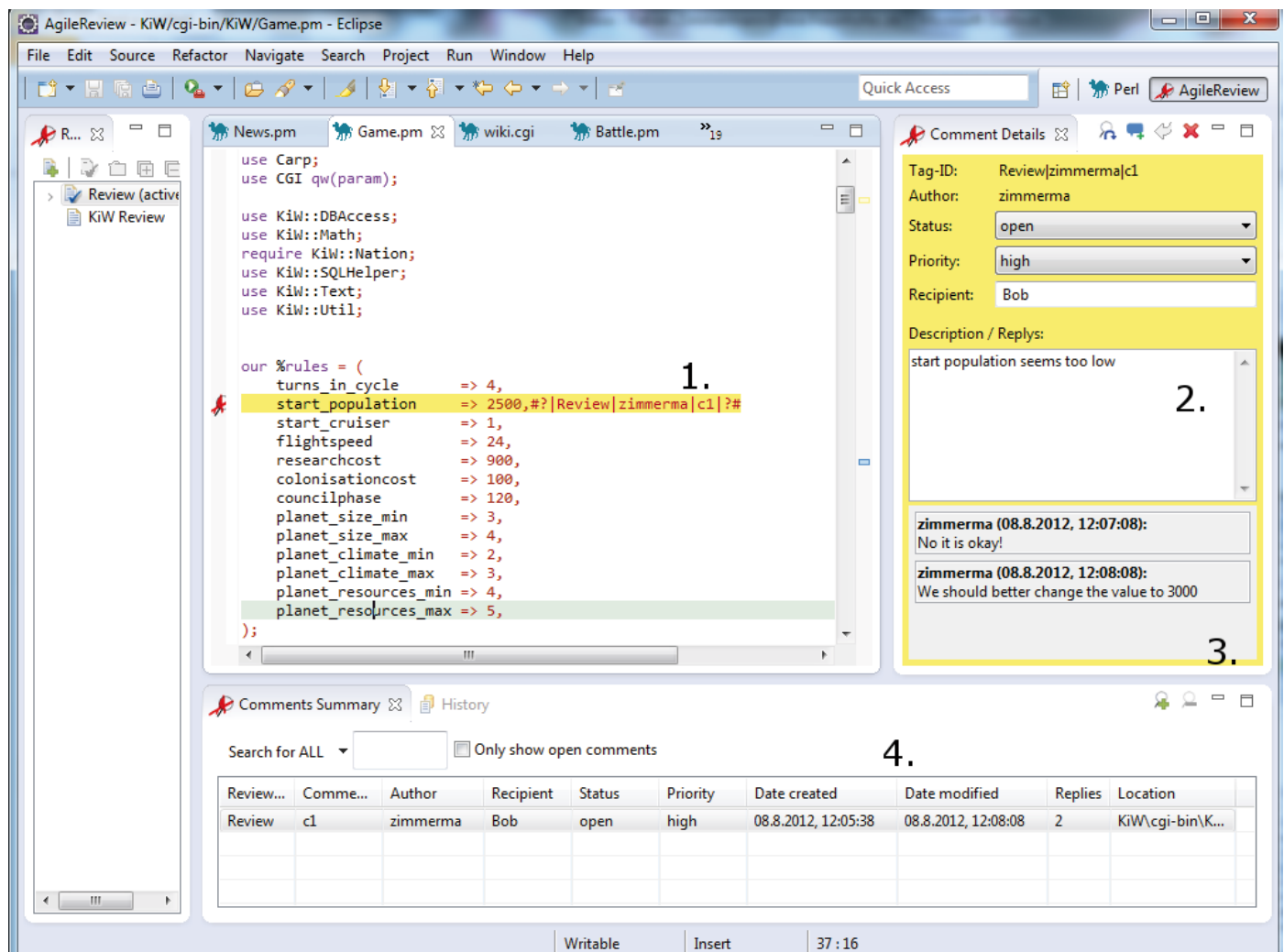


Figure 2: Project opened in AgileReview perspective

Bibliography

Michael Fagan, *Design and code inspections to reduce errors in program development*, IBM Systems Journal, vol. 15, no. 3, pp. 182-211, 1976.

Oliver Laitenberger, Marek Leszak, Dieter Stoll, and Khaled El-Emam, *Evaluating a Causal Model of Review Factors in an Industrial Setting*, 1999.

Barry Boehm and R. Victor Basili, *Software Defect Reduction Top 10 List*, IEEE Computer, vol. 34, no. 1, pp. 135-137, 2001.

Jason Cohen, *Best Kept Secrets of Peer Code Review -- Modern Approach. Practical Advice*. Smart Bear Inc., 2006.

Frank Elberzhager, Jürgen Münch, Dieter Rombach, and Bernd Freimut, *Optimizing Cost and Quality by Integrating Inspection and Test Processes*, International Conference on Software and Systems Process, pp. 3-12, 2011.

<http://search.cpan.org/~thaljef/Perl-Critic-1.118/lib/Perl/Critic.pm/>

Eclipse, <http://www.eclipse.org/>

EPIC -- Perl Editor and IDE for Eclipse, <http://www.epic-ide.org/>

AgileReview -- Code review plugin for Eclipse
<http://www.agilereview.org/>

Author: Reini Urban (rurban@cpanel.net)

Bio Reini Urban

Reini Urban maintains a lot of cygwin packages, is a parrot committer and maintains a couple of perl B modules, best known for the perl compiler suite. He works on perl core since 1995, but is also using and maintaining packages in other languages, such as Common Lisp, C, PHP, Python, Parrot and Shell script.

At cPanel he spends 95% of his time to maintain the compiler and working on improving perl5 and parrot.

Abstract

address-sanitizer (aka ASan) is a memory error detector for C/C++, superior to valgrind. It comes with clang since 3.1.

It finds:

- Use after free
- Out-of-bounds accesses to
 1. heap
 2. stack
 3. globals
- Use after return

It is very fast. The average slowdown of the instrumented program is $\sim 2\times$, it's $\sim 10\text{-}20\times$ faster than valgrind. DEBUGGING builds should just use it.

The tool works on x86 Linux and Mac, a windows port is in work.

How it works, what errors it finds, some tools.

Overview

"Memory access bugs, including buffer overflows and uses of freed heap memory, remain a serious problem for programming languages like C and C++. Many memory error detectors exist, but most of them are either slow or detect a limited set of bugs, or both. This paper presents AddressSanitizer, a new memory error detector. Our tool finds out-of-bounds accesses to heap, stack, and global objects, as well as use-after-free bugs. It employs a specialized memory allocator and code instrumentation that is simple enough to be implemented in any compiler, binary translation system, or even in hardware.

AddressSanitizer achieves efficiency without sacrificing comprehensiveness. Its average slowdown is just 73% yet it accurately detects bugs at the point of occurrence. It has found over 300 previously unknown bugs in the Chromium browser and many bugs in other software."

Typical memory bugs

- Heap OOB *out-of-bounds*
- Stack OOB *out-of-bounds* (unique to asan)
- Global OOB *out-of-bounds* (unique to asan)
- UAF *use-after-free* (aka *dangling pointer*)
- UAR *use-after-return* (unique to asan)
- UMR *uninitialized memory reads*
- Leaks *not freed pointer*

OOB:

- overflow or underflow
- read or write access

Tools

- AddressSanitizer CTI *compile-time instrumentation*
- Valgrind/Memcheck DBI *dynamic binary instrumentation*
- Dr. Memory (DBI)
- Mudflap (CTI)
- Guard Page (Library) (*Electric fence or DUMA on Linux, Page Heap on Windows, Guard Malloc in Mac*)

Comparison

see image next page

Usage

See *perldoc perlhacktips* for the Configure flags. (`-Accflags=-faddress-sanitizer ...`)

The instrumentation of pointer accesses is done during compile-time, there is no big run-time penalty. So you just prepare your asan-enabled perl, either with `-DDEBUGGING` or fully `-O3` optimized, and test your modules with it. It will not hurt.

This is different to the valgrind usecase, where you typically forget to test your module against valgrind, and suffer because it is so slow.

	AddressSanitizer	Valgrind/Memcheck	Dr. Memory	Mudflap	Guard Page
technology	CTI	DBI	DBI	CTI	Library
ARCH	x86	x86,ARM,PPC	x86	all(?)	all(?)
OS	Linux, Mac	Linux, Mac	Windows, Linux	Linux, Mac(?)	All (1)
Slowdown	2x	20x	10x	2x-40x	?
Detects:					
Heap OOB	yes	yes	yes	yes	some
Stack OOB	yes	no	no	some	no
Global OOB	yes	no	no	?	no
UAF	yes	yes	yes	yes	yes
UAR	some	no	no	no	no
UMR	no	yes	yes	?	no
Leaks	not yet	yes	yes	?	no

Beware: Never try to valgrind an asan compiled module! It will blow up.

Found perl5 errors

In Nov 2011 I found a lot of possible false positives with an early version, which was successfully used by Google.

I found this valid bug [CPAN #72700] in Scalar-List-Utils:

```
gv_init(rmcgv, lu_stash, "List::Util",
12, TRUE);
```

12 instead of 10, [CPAN #73118] in DBI and [CPAN #73111] in JSON::XS.

I wrote a better symbolizer for beautify the backtraces and adjusted my blacklist.

In March 2012 I ran the next round for the upcoming 5.16 release. I have found 4 core bugs. No invalid stack access, only globals and even one heap bug, previously undetected by valgrind.

- #111594: 0ffb95f Socket.xs heap-buffer-overflow with abstract AF_UNIX paths
- #111586: sdbm.c: fix off-by-one access to global ".dir"

- #72700: The same copy&paste List::Util BOOT bug, reading global past 2 bytes, still unfixed.
- #111610: XS::APItest::clone_with_stack heap-use-after-free on PL_curcop.

And more CPAN bugs:

- B-Generate-1.44

```
static SV *specialsv_list[6];
...
specialsv_list[6] = (SV*)pWARN_STD;
// asan warned here
```

- B-Flags-0.06

```
==19039== ERROR: AddressSanitizer \
heap-buffer-overflow \
on address 0x2b0995e2d47f \
at pc 0x2b09965e709f bp 0x7fff \
f150fcfb0 sp 0x7ffff150fcfa8
```

READ of size 1 at 0x2b0995e2d47f thread T0

The fix was:

```
- if (SvEND(RETVAL) - 1) == '\,') {
+ if (SvCUR(RETVAL) && \
    (SvEND(RETVAL) - 1) == '\,') {
```


RETVAL was an empty string in this case, and checking `SVEND - 1` for a zero size string is invalid.

After the 5.16 release in May 2012 asan was officially released with clang 3.1 and the following core errors were still present:

- heap-overflow threaded-only in `swash_init - Carp - caller - gv_stashpvn` call [perl#113060] cx corruption
- DBI use-after-free [cpan#75614]
- List::Util 1.24 [cpan#72700] (be sure to upgrade it from CPAN if you need to use 5.16.0 plain. Fixed in 1.25)
- `clone_with_stack` heap-use-after-free on `PL_curcop` [perl#111610]

The DBI error looked most serious to me, as it affected the typical use case and could be used for a security attack against databases. The author totally ignored the bugreport, marked the use-after-free pointer error of the internal database handle as unimportant. On p5p I got similar reactions, 50\% of the patches were not applied for the major release in a 6 months time-frame.

With parrot all three found bugs were fixed immediately, a new macro was introduced to mark false positives and there is only one outstanding memory bug found with asan not yet fixed. Because there is no easy testcase yet written, and it only affects rakudo.

I had to fix the DBI use-after-free bug (a refcnt error) by myself, the fix is still not present in the latest stable DBI CPAN release, and the Changelog does not even mention my bugreport and my fix and downplays the importance.

Morale

So beware when working in perl security. You will make no friends, you'll have to threaten authors with CVE's, perl maintainers are generally uncooperative and slow compared to other projects.

The official perl security list is a joke.

I got no answers to my warnings. Apparently they thought my reports are invalid. Only after others could reproduce the errors with different memory checkers and came up with different patches the holes were fixed. The typical *warnocking phenomenon* for which perl is famous.

Most replies were of the kind "I do not understand your problem. We cannot fix what we do not understand." On all other projects I found similar errors this was totally different. Projects owners are usually happy about reports of pointer errors. Nevertheless most patches are now finally accepted and the problems fixed.

In the cases I was wrong with security assumptions, I got no response back what is wrong in my logic chain. Only on IRC side-chats people came up with explanations what was wrong.

Footnotes

USENIX ATC 2012 Abstract <http://research.google.com/pubs/pub37752.html>

Adventures with clang and ASan <http://blogs.perl.org/users/rurban/2011/11/adventures-with-clang-and-asan.html>

address-sanitizer round 2 <http://blogs.perl.org/users/rurban/2012/03/address-sanitizer-round-2.html>

LLVM 3.1 with AddressSanitizer released <http://blogs.perl.org/users/rurban/2012/05/llvm-31-with-addresssanitizer-released.html>

Re: "Warnock's Dilemma"???? <http://www.nntp.perl.org/group/perl.perl6.language/2003/05/msg15407.html>

Bibliography

1. Konstantin Serebryany and Derek Brunning and Alexander Potapenko and Dmitry Vyukov. *AddressSanitizer: A Fast Address Sanity Checker*. USENIX ATC 2012. <http://research.google.com/pubs/pub37752.html>.

Author: Jonathan Worthington (jnthn@jnthn.net)

Bio Jonathan Worthington

Jonathan is originally from England, but more recently has been working his way through living in European countries that begin with an "S". After a warm winter in Spain and two lovely years spent in beautiful Slovakia, he's now reached Sweden, where he enjoys working at Edument AB as a software architect and teacher.

In the Perl world, Jonathan is best known as one of the key developers of the Rakudo Perl 6 compiler. His work has focused on the object model, type system, multiple dispatch and signatures. He's a regular speaker in the European Perl Conference and Workshop scene, and finds any invite to come and speak and enjoy a few beers with the local Perl hackers hard to resist.

When he's not hacking away on something, Jonathan loves to travel the world, go for walks, study natural languages, eat curry or just relax with friends over a pint.

Abstract

The Perl 6 exception system differs significantly from that of Perl 5. Errors generated during compilation and by the various built-ins have dedicated types. This means they can be handled in a fine-grained manner. Furthermore, details pertaining to the problem at hand can be easily accessed through methods on the exception objects. Developers of modules and applications can easily create their own exception types also.

While throwing exceptions is done with the familiar `die` function, handling them can be fairly different. Any block may handle exceptions by installing a `CATCH` phaser, and the `eval` block form is replaced with `try`.

This article provides an overview of the exception system in Perl 6, demonstrating how to work with and extend it.

What are Exceptions, anyway?

Exceptions are a mechanism for conveying that something abnormal happened during the execution of a program. In Perl the lines between compile-time and runtime are blurred, through mechanisms such as `BEGIN` and `eval`. As a result, the compiler may throw exceptions as well as the built-ins. Modules and application code

may also throw exception to convey their own notions of abnormality.

Thus an exception could result from a wide range of circumstances: trying to `eval` invalid code, trying to call a method that does not exist, trying to create a directory without sufficient permissions or trying to parse invalid JSON with a module like `JSON::Tiny`.

While we could indicate all of these situations with special return values, this ends up conflating the happy paths in our code with the sad paths, and can make it far too easy to miss that something went wrong. Exceptions make themselves hard to miss, and are communicated out of band. The language then provides mechanisms for handling exceptions.

Exploring the structure of exceptions in Perl 6 and the mechanisms the language provides for handling them is the subject of this article.

Exploring Exceptions

We enter the Rakudo REPL and decide to calculate the tangent of a number. We enter the number, then call the `tan` method on it. Sadly, we make a typo, which we are quickly informed of.

```
> 12.8.tna
Method 'tna' not found for invocant of class 'Rat'
```

The error message here contains the name of the method that was not found, the type of the object we tried to call it on, and an overall description of what happened. To us as a human, that's fairly helpful. Beneath this textual facade, however, lies an object that captures the key pieces of information and makes them available programmatically.

To explore this, we need to get hold of the exception object. At the REPL, the easiest way to do this is to put a `try` statement prefix in front of our broken method call, and then look at what is in `$!` - a special variable that contains the most recent exception in the current scope. The `perl` method gives us a representation of an object in Perl syntax, rather like `Data::Dumper` does in Perl 5.

```
> try 12.8.tna; say $!.perl
X::Method::NotFound.new(
  method => "tna",
  typename => "Rat",
  private => Bool::False
)
```

In this output, `X::Method::NotFound` is the type of the exception that was thrown. An exception type is just a class, so this tells us that there is a class called `NotFound` in the `X::Method` package. At this point, one may wonder what other exceptions related to methods can occur. Sure enough, we can take a look inside of the package's symbol table - which is really just like a hash - to find out.

```
> .say for X::Method::.keys
NotFound
InvalidQualifier
Private
```

Returning to our exception object, we can also try extracting the individual pieces of information. For example, we can get the name of the method that we failed to call:

```
> try 12.8.tna; say $!.method
tna
```

We may also be curious what this exception object inherits from; calling the meta-method `parents` tells us:

```
> try 12.8.tna; say $!.^parents
Exception()
```

By this point, you are probably starting to get the overall idea. Perl 6 has a bunch of different exception types that ultimately inherit from `Exception`, organized into packages. These objects hold information about what when wrong. On demand, they can take this information and form a human-readable message describing the problem. This is done by calling the `message` method:

```
> try 12.8.tna; say $!.message
No such method 'tna' for invocant of type 'Rat'
```

Or just by using the exception object in a string context, which delegates to `message`.

```
> try 12.8.tna; say "Oh, my: $!"
Oh, my: No such method 'tna' for
invocant of type 'Rat'
```

Handling Exceptions

We've already seen the `try` syntax. It can be used as both a statement prefix, as we have already seen, or in front of a block.

```
loop {
  try {
    my $expr = prompt "> ";
    say eval($expr);
  }
  say "FAIL: $!.message()" if $!;
}
```

Alone, `try` will capture any exception that occurs, placing it into the special variable `$!`. Sometimes, this is what you want; there may be cases where you legitimately don't care how something fails, and want to carry on regardless. Often, however, you will want to take action as a result of the failure. You could do this by checking if `$!` is set after the `try` and, if needed, examining the exception. For most cases, however, using the `CATCH` phaser is preferable.

For example, imagine we are writing a script to count effective lines of code in source files. To keep the example simple, an effective line of code is one that contains something other than a comment, whitespace or curly braces. We start off by calculating the count for a single file, which we'll hardcode the name of. Here's the code, featuring the `CATCH` phaser.

```
try {
  say lines("foo.p6".IO).grep(
    { $_ !~ /\s*[<?>|\#'|.*|'\|'|']\s*/ }
  ).elems;
  CATCH {
    when X::IO {
      note "Could not count lines in foo.p6";
    }
  }
}
```

A phaser is a mechanism used to attach code to a block that will run under specific circumstances or at a specific time. The `CATCH` phaser is used to attach code that will run if an exception is thrown in the dynamic scope of the block, and nothing else handles it first. That is, if we call a method and it does something that results in an exception and does not handle it, then our `CATCH` phaser will be run.

Inside of a `CATCH`, the exception that needs handling is placed into the topic variable, `$_`. This means that normal smart-matching can be used to indicate that we only want to handle certain types of exception. Here, we only handle IO exceptions (since all IO exceptions do the `X::IO` role). Anything that is not an IO exception will be re-thrown, and the next handler - if any - will get a chance to handle it. Just like any other place where `when` is used, it is also possible to use `default`, which will match any exception.


```
try {
  say lines("foo.p6".IO).grep(
    {$_ !~~ /\s*[<?>|'#"'.*|'\{\|\}'']\s*$/ }
  ).elems;
  CATCH {
    when X::IO {
      note "Could not count lines in foo.p6";
    }
    default {
      note "Failed to process foo.p6";
    }
  }
}
```

Note that simply having a `\verb/CATCH/` block is not sufficient. For example, if we were to have written:

```
try {
  say lines("foo.p6".IO).grep(
    {$_ !~~ /\s*[<?>|'#"'.*|'\{\|\}'']\s*$/ }
  ).elems;
  CATCH {
    note "Could not count lines in foo.p6";
  }
}
```

Then the note would be emitted, but the exception would not be considered handled, leading to it being re-thrown for the next `CATCH` block to consider. This also happens in the case that none of the `when` blocks match the exception, and there is no default.

There are a couple of other matters worth discussing with regard to `CATCH`. Firstly, it can be attached to any block, and is not coupled in any way to the `try` block. For example, it could be attached to a sub. Secondly, since it is placed inside of a block, it can see all of the lexicals in the containing block. Now we'll refactor our script to process a list of files supplied as command line arguments to the script, taking advantage of both of these features.

```
sub effective_lines($filename) {
  return lines($filename.IO).grep(
    {$_ !~~ /\s*[<?>|'#"'.*|'\{\|\}'']\s*$/ }
  ).elems;
  CATCH {
    default {
      note "Failed to count lines in $filename";
    }
    return 0;
  }
}

say [+] @*ARGS.map(&effective_lines);
```

How to die

We've now seen some ways to handle exceptions that are thrown. However, we may want to throw exceptions from our own modules or applications to signal error conditions. For example, suppose we're looking through a bunch of files that should each contain a stored procedure declarations. We write a sub that takes a file and tries to parse out the name of the stored procedure. If we do not find one, we wish to throw an exception. Here's a first attempt.

```
my rule create_sp {
  :i
  create proc[edure]
  '[dbo].'?
  [
    | '[' ~ ']' $<name>=[\w+]
    | $<name>=[\w+]
  ]
}

sub get_sp_name($sp_file) {
  my $sp = slurp($sp_file);
  if $sp ~~ /<sp=&create_sp>/ -> (: $sp)
  {
    return $sp<name>;
  }
  else {
    die "Could not find a stored
        procedure in $sp_file";
  }
}
```

The rule is used to handle the parsing and the capturing of the name. The exception throwing takes place in the sub `get_sp_name`. This should look fairly familiar to any Perl 5 programmer; we use the `die` keyword and specify a string message.

However, we may wish to go a step further and define an exception type. The emerging practice for this is to install them in the `x` package, with your module or application's package followed by the name of the exception.

```
class X::SPAnalyzer::SPNotFoundInFile
  is Exception {
    has $.file;
    method message() {
      "Could not find a stored
        procedure in $.file"
    }
  }
}
```

This can be used in place of the string message with `die`:

```
die X::SPAnalysis::SPNotFoundInFile.new(
  file => $sp_file
);
```

Alternatively, the `throw` method, inherited from `Exception`, can be called to throw them.

```
X::SPAnalysis::SPNotFoundInFile.new(
    file => $sp_file
).throw;
```

Conclusion

Perl 6 takes a fairly object-oriented approach to exceptions. However, since it is still possible to `die` with a simple string, to treat any exception object as a string and even to pattern match against it in the `when` blocks, you are not forced into the OO approach when it is overkill for the task at hand. This fits with Perl 6's general aim to let you script to your heart's content, but to give you the tools you need to refactor scripts towards more robust applications and modules as necessary.

Attaching exception handlers using phasers means that any block can have an exception handler. While it is possible to write a `CATCH` within a `try` block, it is not in any way required. In fact, this is only typically done when just a subset of the statements that

would naturally fall into an existing scope need to be protected. In these cases a bare block would suffice, but the `try` serves as a clue to the reader.

Finally, the exception system builds upon familiar concepts from elsewhere in the language. Smart-matching within the `CATCH` phasers allows the developer to draw on their understanding of `when` and `default`, while defining new exception types is simply writing classes.

Want to give it a try? All of the examples from this article were tested on the Rakudo (<http://rakudo.org/>) Perl 6 compiler, which at the time of writing has the most complete implementation of Perl 6 exceptions.

Author: Carl Mäsak (cmasak@gmail.com)

Bio Carl Mäsak

A Perl 6 programmer who also likes Perl 5 a lot. Has helped with Rakudo since 2008. Likes to report bugs for some reason. Writes a lot of Perl 6 code.

Abstract

We keep hearing so many success stories. This is a talk about things sometimes failing, and how they fail.

I am a firm believer of learning by breaking stuff. A big part of understanding how things work is related to understanding the failure modes of those same things.

This talk takes us through a handful of situations where software failed rather badly, either because of professional exploiters of failure modes (like script kiddies) of innocently curious people (like me).

Exploits

This talk takes the following thesis as a starting point, and explores it:

Every feature in a system is a potential source of exploits.

In the context of this talk, let's define "exploit" as "use outside of the intended parameters".

Those intended parameters, depending on the system, could be set by the system's originator, its user base, or just society at large. Exploits don't have to be malicious -- if I build a castle out of sugar cubes that's using sugar cubes outside of their intended parameters (sweetening stuff), but it isn't malicious.

Features

The accumulated potential for exploits grows with the number of features. If features are so bad, maybe we shouldn't add so many to our systems? The problem is that we are rather fond of features. They're the whole point of our systems. Maybe some features can be dropped. Most probably can't. Sometimes the willingness to drop a feature shifts when

the feature is considered from an exploitation point of view.

However, one thing we sometimes *can* do is think about which features can be unified. Let's think of "unification" in this case as taking code paths that belonged to individual features and reducing them into a single code path. In some sense, that allows you to retain your features but expose them as aspects of a single underlying feature. Done correctly, this can reduce the exposure to exploits. Unification can also have the advantage of making the domain model conceptually simpler, and the resulting win in manageability can lead to a net decrease in bugs.

C. A. R. Hoare has a relevant quote:

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature.

I'd like to add to this that what I've learned in recent years is that it doesn't take a lot to make a system cross the "complicated" threshold.

Bounded contexts

Everything we've said so far could make it seem like the prevalence of exploits grows roughly linearly with the amount of features. But it's much worse than that.

Every combination of features in a system is a potential source of exploits.

By sheer combinatorics, we're fucked.

This is why we talk of corner cases -- because features *interact*, whether or not we anticipate that they will.

I've always liked this tweet because it proposes such an exploit:

When I start a band, I'm gonna call it "Podcasts", just to fuck with iTunes' directory structure.

Features tend to interact because they work on the same data. Try to keep this to a minimum. In fact, whenever possible, try to put bundles of features which do not interact as far away from each other as possible. Put unrelated code paths in different *bounded contexts*, and strongly limit interaction across such contexts. In some sense, each bounded context becomes system on its own, with fewer unexpected corner cases. It's not a matter of removing all contact between different bounded contexts; rather to have such contact take place over a predefined protocol, such as inter-context message passing. This allows each context to maintain its own data without being entangled with unrelated concerns.

Positive bias

But a real source of exploits in software is *positive bias*, often referred to as *confirmation bias*: a widespread weakness in human information processing whereby we forget to eliminate hypotheses. We tend to think that finding *positive* examples of a hypothesis is enough, but in reality it's finding *negative* examples that allow us to distinguish between hypotheses and reach conclusions.

This excerpt from Chapter 8 of the fanfic "Harry Potter and the Methods of Rationality" illustrates this phenomenon better than I can.

The boy's expression grew more intense. "This is a game based on a famous experiment called the 2--4--6 task, and this is how it works. I have a *rule* - known to me, but not to you - which fits some triplets of three numbers, but not others. 2--4--6 is one example of a triplet which fits the rule. In fact... let me write down the rule, just so you know it's a fixed rule, and fold it up and give it to you. Please don't look, since I infer from earlier that you can read upside-down."

The boy said "paper" and "mechanical pencil" to his pouch, and she shut her eyes tightly while he wrote.

"There," said the boy, and he was holding a tightly folded piece of paper. "Put this in your pocket," and she did.

"Now the way this game works," said the boy, "is that you give me a triplet of three numbers, and I'll tell you 'Yes' if the three numbers are an instance of the rule, and 'No' if they're not. I am Nature, the rule is one of my laws, and you are investigating me.

You already know that 2--4--6 gets a 'Yes'. When you've performed all the further experimental tests you want - asked me as many triplets as you feel necessary - you stop and guess the rule, and then you can unfold the sheet of paper and see how you did. Do you understand the game?"

"Of course I do," said Hermione.

"Go."

"4--6--8" said Hermione.

"Yes," said the boy.

"10--12--14", said Hermione.

"Yes," said the boy.

Hermione tried to cast her mind a little further afield, since it seemed like she'd already done all the testing she needed, and yet it couldn't be that easy, could it?

"1--3--5."

"Yes."

"Minus 3, minus 1, plus 1."

"Yes."

Hermione couldn't think of anything else to do. "The rule is that the numbers have to increase by two each time."

"Now suppose I tell you," said the boy, "that this test is harder than it looks, and that only 20% of grownups get it right."

Hermione frowned. What had she missed? Then, suddenly, she thought of a test she still needed to do.

"2--5--8!" she said triumphantly.

"Yes."

"10--20--30!"

"Yes."

"The real answer is that the numbers have to go up by the same amount each time. It doesn't have to be 2."

"Very well," said the boy, "take the paper out and see how you did."

Hermione took the paper out of her pocket and unfolded it.

Three real numbers in increasing order, lowest to highest.

Hermione's jaw dropped. She had the distinct feeling of something terribly unfair having been done to her, that the boy was a dirty rotten cheating liar, but when she cast her mind back she couldn't think of any wrong responses that he'd given.

"What you've just discovered is called 'positive bias'," said the boy.

"You had a rule in your mind, and you kept on thinking of triplets that should make the rule say 'Yes'. But you didn't try to test any triplets that should make the rule say 'No'. In fact you didn't get a *single* 'No', so 'any three numbers' could have just as easily been the rule. It's sort of like how people imagine experiments that could confirm their hypotheses instead of trying to imagine experiments that could falsify them - that's not quite exactly the same mistake but it's close. You have to learn to look on the negative side of things, stare into the darkness. When this experiment is performed, only 20% of grownups get the answer right. And many of the others invent fantastically complicated hypotheses and put great confidence in their wrong answers since they've done so many experiments and everything came out like they expected."

We program in terms of scenarios and use cases that we imagine in our mind. Unless we explicitly train ourselves to think in terms of sad paths and possible vulnerabilities, we're unlikely to picture the "negative hypotheses" in our code, the things that might go wrong. The uses that might fall outside of our intended parameters.

Power features

Let's apply the danger of positive bias to so-called "power features" in programming languages. Perl is actually a good example here, because many of its power features are double-edged swords.

Somewhere inside the CPAN module `SOAP::Lite`, there is a method dispatch that (simplified) has this form:

```
$soap_object->$method_name(@parameters);
```

This makes a lot of sense, because SOAP is an RPC-like protocol in which the method name to call is passed from client to the server, and comes in as input to `\verb!SOAP::Lite!`. We don't know the method name at compile time, so it makes sense to use Perl's built-in facility to dis-

patch on the name found in a variable, here `$method_name`.

But with great power comes great exploitability. Here's a comment and a check from the `SOAP::Lite` module:

```
# check to avoid security
#   vulnerability:
# Protected->Unprotected::method(
#   @parameters)

# see for more details:
# http://www.phrack.org/phrack/58/
# p58-0x09
die "Denied access to method"
    . "($method_name)\n"
    unless $method_name =~ /\w+$/;
```

The `phrack.org` link is dead, but the Internet Archive has our back. What follows is a simplified explanation of the attack.

The first thing that the script kiddie exploits is that `$method_name` could include not just a method name but also a package, such as `HTTP::Daemon::ClientConn::send_file`. This file was never meant to be called through the SOAP dispatch mechanism, but the semantics of Perl allow it. The programmer probably never conceived of the possibility before being hit with the exploit. (Notice how even the variable name, `$method_name`, allows us to endure in our positive bias here?)

Here's a simple script which demonstrates the exploit.

```
use 5.014;
use strict;
use warnings;

package C;

sub foo {
    my ($self) = shift;
    say "foo called with arguments ",
        join ", ", map { qq["$$_"] } @_;
}

package main;

my $method_name = shift @ARGV
    or die "Usage: $0 <method name>
           <method argument>*";
my @arguments = @ARGV;

C->$method_name(@arguments);
```

Running this script looks like this:

```
$ perl explain_exploit foo Hello world!
foo called with arguments "Hello", "world!"
```

But now let's call the same script with this first parameter:

```
$ perl explain_exploit HTTP::Daemon::
ClientConn::send_file ...
```

And let's further assume that among the dependencies of `SOAP::Lite`, there is code that looks like this:

```
package HTTP::Daemon::ClientConn;

sub send_file {
    my($self, $file) = @_;
    if (!ref($file)) {
        open(F, $file) || return undef;
        # ...
    }
}
```

The second thing that the script kiddie exploits is that the above code contains an "unprotected open", the two-argument form where `$file` is allowed to contain not just the file name, but also the *mode* for the file to be opened in. Common modes are reading (<), writing (>), and appending (>>), but there's also *pipng* (!), which executes a system command. This can be very useful.

One has to be careful, though, not to allow arbitrary user input to populate the parameter `$file`. (Notice, by the way, how the naming of the variable `$file` *also* assists our positive bias in thinking everything is OK?) The ability of the second argument of an `open` call to dictate the mode is the reason two-argument `open` is strongly discouraged nowadays.

The following should send chills down your spine:

```
$ perl explain_exploit HTTP::Daemon::
ClientConn::send_file `|/bin/ps`
PID  TTY      TIME CMD
25500 pts/16   00:00:00 bash
28836 pts/16   00:00:00 perl
28837 pts/16   00:00:00 ps
```

The whole exploit is two injection attacks stacked on top of each other. The two exploitable features aren't even in the same *module*, and not even written by the same author. Perl's indirect method dispatch doesn't make it easy to separate contexts.

It's interesting to note how subverting a CPAN module this way is very close to being art. I don't condone computer intrusion in any way,

but I do admire the thinking that went into this exploit. We should all think more like this; our software would be better for it.

Script kiddies and bug reports

It is said that every good bug report should contain these three things:

- Steps to reproduce
- What you observed
- What you expected

These rules become *self-evident* when viewed from the perspective of a fourteen-year-old script kiddie:

- Steps to reproduce: *pics or it didn't happen*
- What you observed: *I 'sploited it...*
- What you expected: *...and they didn't even see it coming*

Rakudobugs

An ancient Greek myth tells about king Midas, who was granted the ability to transmute things he touched turn into gold. Although initially very excited about his powers, he quickly found the downsides of the gift, as he couldn't eat anything. Also, he inadvertently turned his daughter into a gold statue. There's probably some lesson in there -- - maybe that of riches being less important in life than some other things, like food and love.

In 2008, I discovered that I have something of the Midas touch when it comes to Rakudo bugs. Actually, it seems to be me and software, but for some reason the effect is very strong with Rakudo. I did the opposite journey king Midas did; no-one wants to have their software break all the time, but I came to accept it and consider it an asset of sorts. Better for these bugs to hit me, I reasoned, than future users of Perl 6.

So I set forth and submitted rakudobugs to our RT instance as I found them. I quickly got the epithet "bug wrangler", and learned to streamline the bug submitting process as much as possible. Somehow my brain considers submitting a rakudobug as "zero work".

As of August 2012, I've submitted 1356 tickets in four years;

slightly less than one per day. There's a total 2874 tickets in the perl6 queue, so about 47% of them were submitted by me.

I submit a fair amount of bugs for others, but the majority of the bugs I submit are things I discover by actually using Perl 6. Surprisingly many of these are found as part of refactoring programs. After a refactor, there's an implicit expectation that things will work the same. If they don't, that's a bug (or a thinko on the part of the programmer).

Then again, sometimes all it takes is trying a feature in a new way. In a sense, I hope to be the jungle guide who blasts a path through Perl 6 use cases with a machete, classifying and containing interesting bugs as they attack me.

A very rewarding kind of bugs are "Null PMC access" errors (generated by the Parrot VM, very much like a null reference exception on other VMs) and segfaults. Both of these are *by definition* use outside of the parameters of the language implementation, which should never leak VM errors to the user. As such, these bugs count as "exploits" as we have defined it.

Make no mistake: the people working on Rakudo Perl are really good developers. Incompetence is not the cause of these bugs; complexity is. If you want an example of a software design "so complicated that there are no obvious deficiencies", Perl 6 is it.

In fact, sometimes I've fantasized about constructing a huge multiplication table of all the features in Perl 6, and then just go through it cell by cell trying every pair of features to see if that digs up new bugs. Though perhaps a simple script suggesting random combinations of features would be more apt.

Some rakudobug case studies

Let me show how *bug golfing* happens. The following instance is still fresh enough in my mind that I can give an account of my thought process.

A user, nebuchadnezzar, showed up on #perl6 and reported that the following example from the Perl 6 book didn't work:

```
class Rock      { }
class Paper     { }
class Scissors  { }

multi wins(Scissors $, Paper $) { +1 }
multi wins(Paper $, Rock $) { +1 }
multi wins(Rock $, Scissors $) { +1 }
multi wins(::T $, T $) { 0 }
multi wins($, $) { -1 }
```

```
sub play ($a, $b) {
    given wins($a, $b) {
        when +1 { say "Player One wins" }
        when 0 { say "Draw" }
        when -1 { say "Player two wins" }
    }
}

play(Rock, Rock);
# output: Player two wins

given wins(Rock, Rock) {
    when +1 {say "Player One wins"}
    when 0 {say "Draw"}
    when -1 { say "Player two wins"}
} # output: Draw
```

His running hypothesis was that "given" in the subroutine does not seem to behave the same way as outside. But the program is far too large for us to say anything sensible about it. So, we golf. (Watch as we get less and less code the more we zero in on the bug. This is very typical of this kind of exploration.)

By the way, a **type capture**, like `::T` above captures the type in the variable `T`, and allows you to do type matching on it later. So the signature of `wins(::T $, T $)` is to be read as "accepts two parameters with identical type".

The first variant I come up with is this:

```
class R {}
multi w(::T, T) { 0 }
multi w($, $) { -1 }
sub p($a, $b) { w $a, $b }
say p(R, R);
say w(R, R);

OUTPUT: -1\n0\n
```

Note, there is no `given` construct. So we can put that hypothesis aside. My new hypothesis is instead that it's the `p` subroutine that does it somehow. So I try with a "pointy block" instead of a subroutine.

```
class R {}
multi w(::T, T) { 0 }
multi w($, $) { -1 }
(-> $a, $b { say w $a, $b }) (R, R);
say w R, R;

OUTPUT: -1\n0\n
```

So it wasn't the subroutines. My guess now is that it's parameter binding.

```
class R {}
multi w(::T, T) { 0 }
multi w($, $) { -1 }
my ($a, $b) = R, R;
say w $a, $b;

OUTPUT: -1\n
```


So it isn't parameter binding either. It's variables. Or rather, containers.

A **container** is the thing that allows you to assign new values to a variable, array element, or other similar mutable thing.

These two runs seem to corroborate the container hypothesis:

```
multi w(::T, T) { 0 }
multi w($, $) { -1 }
say w([1, 1]);
```

OUTPUT: -1\n

```
multi w(::T, T) { 0 }
multi w($, $) { -1 }
say w(1, 1)
```

OUTPUT: -1\n

And then, finally, the run that exposes the bug:

```
sub w(::T, T) { 0 }
say w([1, 1])
```

```
OUTPUT: Nominal type check failed for
parameter
      ``; expected Scalar but got Int
instead
      in sub w
```

So the whole bug boils down to type captures and containers not playing well together.

The next one I had already golfed a fair bit when I presented it to the channel:

```
use Test;
class A {}
(-> &c, $m { A.new(); CATCH
{ default { ok 1, $m } } }) (A, "")
```

OUTPUT: (signal SEGV)

The last line is a pointy block which we instantly invoke. An instance of `A` gets created and immediately invoked, an operation it does not support, thus generating an exception. In the catch clause, we call `ok`, provided through the `Test` module. There's nothing weird going on here; so why does it segfault?

`moritz` manages to remove the dependency on `Test`:

```
class A {}
(-> &c, $m { A.new(); CATCH { default
{ say $m } } }) (Mu.new, '')
```

```
OUTPUT: Null PMC access in find_
method('gist')
```

A Null PMC access is slightly less dramatic than a segfault, but we're still chasing the same bug here.

At this point, I've convinced myself that the `A.new()();` statement actually runs. This next run disproves that hypothesis:

```
use Test;
class A {}
(-> &c, $m { CATCH { default { ok 1, $m }
} }) (A, "")
```

OUTPUT: (signal SEGV)

Which is our first really big clue: the `CATCH` block triggers *even without any other statements in the pointy block*. Which means that the `CATCH` block catches something that is not *in* the pointy block as such.

Could it be that `CATCH` blocks (wrongly) trap binding errors? `moritz` tests:

```
sub f(&x) { CATCH { default {
say "OH NOES" } } }; f Mu.new
```

OUTPUT: OH NOES\n

Yup. And that's the bug. When we tried to print `$m`, it had a Null PMC in it because *it had not been initialized by the binder*, which gave up on the first parameter:

```
(-> &c, $m { CATCH { default { say $m } } }
) (Mu.new, '')
```

```
OUTPUT: Null PMC access in find_
method('gist')
```

Clearly `CATCH` in a block shouldn't catch binder-generated exceptions, and that was the bug here.

I always liked this next one:

```
class B;

method foo() {
  use A; # A.pm just defined a grammar
}
```

```
OUTPUT: You can not add a Method to a
module; use a class, role or grammar
```

This one happened in the interaction between the parser keeping track of whether it is inside a module, a class, or something else, and inclusion of new compilation units. The solution this time was simply to make the parser do a bit of extra book-keeping when seeing a new compilation unit.

The next one presupposes the knowledge of **named parameters** (which bind to named arguments and are identified by their name rather than their position) and **anonymous parameters** (which have just a sigil).

What happens if we have an anonymous named parameter?

```
sub foo(:$) {}
say &foo.signature.perl
```

```
OUTPUT: :(Any $?)\n
```

That shows it as an anonymous *positional* parameter, which is wrong. No-one ever considered the possibility of an anonymous named parameter up until the point when this rakudobug was submitted.

We end this exposition with the infamous snowman-comet bug:

```
say "abc" ~~ m <unicode
    snowman>.(.)<unicode comet>
```

```
OUTPUT: abc\n
```

Now, regexes may be delimited with matching opener and closer characters. A Unicode snowman and comet are *not* matching opener and closer characters. (Though I admit it would be quite cute if they were.)

This only worked for regexes. Other quoting constructs were not susceptible to this. The bug did eventually get fixed, but not so much by finding the cause of it, as by building a next-generation regex engine.

Conclusion

Software is hard. When you can, avoid creating so many features. They will be used against you.

When you can, isolate features from each other so that they can't interact. Identify subsystems of features that can be thus isolated.

As a programmer, be wary of positive bias and the way it hides exploits from you when you code.

Due to positive bias, power features are sources of exploits. Script kiddies are inventive; they will *chain* exploits in order to take control of your environment.

Perl 5's indirect method call is a useful power feature. The data passed to it *must* be validated if it's user input.

Perl 5's two-argument `open` is a power feature, but it's unsafe. It has been obsoleted by three-argument `open`. Do not use two-argument `open`. Do not load modules that use it.

The number of corner cases grows with the square of the number of features. Ask yourself where your threshold of keeping track of such combinations lies.

Experience shows that these corner cases are not just a theoretical concern; they show up all the time.

Bibliography

"Sculpture of Bangor Castle (Town Hall) made of sugar cubes, North Down Museum, Castle Park, Bangor", by Lancastrian.

<http://www.flickr.com/photos/lan-cashire/7493606694/>.

<http://twitter.com/tpope/status/11815295584833536>

http://lesswrong.com/lw/iw/positive_bias_look_into_the_dark

http://en.wikipedia.org/wiki/Confirmation_bias

<http://hpmor.com>

<http://web.archive.org/web/20060212053435/http://www.phrack.org/phrack/58/p58-0x09>

<http://www.mythweb.com/Encyc/entries/midas.html>

<http://rt.perl.org/rt3/Ticket/Display.html?id=114394>

<http://rt.perl.org/rt3/Ticket/Display.html?id=114134>

<http://rt.perl.org/rt3/Ticket/Display.html?id=73886>

<http://rt.perl.org/rt3/Ticket/Display.html?id=69492>

<http://rt.perl.org/rt3/Ticket/Display>.

Author: Carl Mäsak (cmasak@gmail.com)

Bio Carl Mäsak

A Perl 6 programmer who also likes Perl 5 a lot. Has helped with Rakudo since 2008. Is currently working on a grant implementing macros in Rakudo.

Abstract

It's 2012, and Perl 6 is finally reaching a certain level of maturity. It's also getting macros.

Macros are a sort of code templates. They help you fold boilerplate into your programs for greater maintainability. Lisp has them, so they must be cool.

I'm currently in the middle of implementing macros in Rakudo, one of the leading Perl 6 implementations.

This talk takes you through the basics of macros, some of the subtleties of implementing them, and how having them makes Perl 6 less like a Swiss army knife and more like a Swiss army.

What are macros?

Macros are code templates. Just like HTML templates allow you to specify a mostly-constant HTML document with some interesting values inserted here and there, a macro allows you to specify a mostly-constant chunk of code with some interesting parametric chunks of code here and there.

Perl 6 allows you to dynamically prepare the template that gets inserted into the mainline code. That's right, the macro *runs* right in the middle of compile-time, much like a `BEGIN` block. This creates some interesting challenges having to do with crossing from compiling the program to running it, and back.

Much of the unique power of macros stems from straddling this boundary, essentially allowing you to program your program.

Before we dive into macros, let's make sure we understand lexical scoping and ordinary routines fully.

Lexical scoping

Lexical scoping is the idea that a variable declaration is *scoped* to the block it appears in:

```
{
  # $var not defined here
  my $var;
  # $var defined here
}
# $var not defined here
```

What's especially powerful about this is that the scope information is available to the parser. We don't have to wait until runtime to get scoping errors. (This is what we mean by *lexical* scoping, as opposed to dynamic scoping.)

Routines are shaped funny

We all know the "shape" of arrays and hashes: arrays hold sequences and let us access them with an integer to get stuff out. Hashes hold mappings and let us access them with a string to get stuff out.

What is the shape of a routine?

Well, routines remember *computations* and let us access them with a bunch of parameters. Essentially, they are shaped like little programs. They can contain anything, including access to variables and its own variable declarations.

A *closure* is a function value with at least one variable defined outside of itself. Like so:

```
sub outer {
  my $x = 42;
  sub inner {
    say $x; # 42
  }
  inner();
}
```

Together, those outside variables make up the *environment* of the closure. I do this a lot in my Perl 6 programming, because the environment of inner subs contains the parameters of outer subs, and so the inner subs need much fewer parameters.

It's so straightforward it doesn't even feel strange. But it actually is kinda strange and wonderful. It gets more obviously strange and wonderful when allow the closure to escape the scope of its environment.

Note that an *anonymous function* is not the same as a closure. An anonymous function is just a function literal that lacks a name. (Why are we so obsessed with functions having names?)

We don't go around calling integer literals "anonymous integers".) The confusion arises because we often use anonymous functions for their capacity to generate closures, like here:

```
sub counter-creator(Int $start) {
    my $counter = $start;
    return sub { $counter++ };
}
```

(The `sub` is included to make this more readable to Perl 5 people. It's not really necessary in Perl 6.)

We've now returned the closure out of its environment. The environment lives on because it is referenced by the closure.

```
my $c1 = counter-creator(5);
say $c1(); # 5
say $c1(); # 6
say $c1(); # 7
```

And we can prove to ourselves that each invocation to the outer function yields a unique closure with its separate environment:

```
my $c2 = counter-creator(42);
say $c2(); # 42
say $c1(); # 8
say $c2(); # 43
```

Because a closure behave like this, a variable that's part of the closure's environment, like `$counter`, will have to be allocated on the heap rather than on the stack. Put differently, the presence of closures in a programming language necessitate garbage collection. See also the funarg problem.

Note that the `$counter` variable is completely encapsulated inside the outer function. We can provide piecemeal access to it in exactly the same way as we can with objects. Closures and objects are equal in power. Which carries us into the next section.

A koan

Because closures and objects are equal in power, they can be defined in terms of one another, like so:

- An object can be made out of a closure. Data hiding comes from declaring va-

riables in the closure's environment. Behavior comes from calling the closure. We can emulate method dispatch by passing the method name as a first parameter.

- A closure is a kind of function object with its environment stored as data, and one method: `apply`.

This duality has been immortalized in a koan by Anton van Straaten:

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened.

For the purposes of this text, a closure is a callable thing with internal state, just like an object. An object's private environment is its class, and a closure's private environment is the totality of the variables defined outside of itself.

ASTs are closures

AST objects are closures. They are not *like* closures, they *are* closures. They are a representation of executable code (potentially) using variables declared outside of themselves.

If our actions are limited to the following, we can work with ASTs while preserving their environments:

- Extracting a sub-AST out of an AST.
- Inserting an AST into another.
- Inserting synthetic AST nodes into an existing AST.

If we manage to talk about an AST without

an environment (by creating one from scratch, for example), we could make that AST do things and participate in code, as long as we don't refer to any outside variables.

The five stages of macro

Let's look at the lifetime of an ordinary subroutine through compilation and running. For simplicity, let's assume it's called exactly once.

- α : The subroutine is parsed.
- β : The subroutine call is parsed. (This may happen before the subroutine is parsed, actually, because subs can be post-declared. Nevermind.)
- χ : The subroutine call is run.
- δ : The subroutine runs.

Macros are more entwined in the process of parsing than that, and so for macros we can identify five stages:

- a: The macro and the `quasi` are parsed.
- b: The macro call is parsed. Immediately as the macro call has been parsed, we invoke the macro.
- c: The macro runs. As part of this, the `quasi` gets incarnated and now has no holes anymore. An AST is returned from the macro.
- d: Back in parse mode, this AST is inserted into the call site in lieu of the macro call.
- e: At some point in the distant future, when compiling is over, the inserted macro code is run.

The steps **b** and **d** correspond to the relatively uninteresting step β . The runtime step **c**, corresponding to step χ , is sandwiched between the parse-time steps **b** and **d**. In short, subroutines have a clear separation of parse-time and runtime. Macros deliberately mix runtime with parse-time.

Hygienic macros

According to Wikipedia "Hygienic macros are macros whose expansion is guaran-

teed not to cause the accidental capture of identifiers." That is, we don't want variables (or other names) to collide between the macro and the rest of the program. If we make it so that they don't, we've successfully made the macro hygienic.

Hygiene is a big deal for all languages with macros in them, since the lack of hygiene can cause weird behaviors due to unintentional collisions. We'll get back to various techniques used to achieve hygiene.

How closures cause hygiene

Let's reach for an example to illustrate how hygiene just falls out naturally when we treat ASTs as closures.

```
macro foo($ast) {
  my $value = "in macro";
  quasi {
    say $value;
    {{{$ast}}};
  }
}

my $value = "in mainline";
foo say $value;
```

Keeping in mind that ASTs retain a "link" to their point of origin, we step through the stages of a macro:

- a:
 - The macro and the `quasi` are parsed.
 - `$value` in the `quasi` block is recognized to refer to the declared variable in the `macro` block.
- b:
 - The macro call is parsed. Immediately as the macro call has been parsed, we invoke the macro.
 - *An AST is created out of `say $value` as a natural effect of parsing. This AST is rooted in the mainline, so the `$value` variable refers to the one in the mainline.*
- c:
 - The macro runs. As part of this, the `quasi` gets incarnated and now has no holes anymore.
 - *`$value` gets inserted, but retains its link to the mainline.*
 - An AST is returned from the macro.
 - *This AST retains its link to the `macro` block.*
- d:
 - Back in parse mode, this AST is inserted into the call site in lieu of the macro call.
 - *Because of how it was constructed, the AST as a whole links to the `macro` block, but a part of it links to the mainline.*

e:

- At some point in the distant future, when compiling is over, the inserted macro code is run.
- *And voila, it prints in macro and then in main-line.*

This is perhaps the smallest example that shows how things stay out of the way of each other. Each step is simple and fully general. It works out similarly even for more composed cases, when the going gets really tough.

Other approaches to hygiene

Wikipedia lists five ways to achieve hygiene in macros:

Obfuscation. Using strange names that won't collide with anything else.

Temporary symbol creation. Also known as "gensym".

Read-time uninterned symbol. Essentially giving symbols inside of a macro their own namespace.

Packages. Keeping the macro's symbols in a separate package.

Hygienic transformation. The macro processor does gensym for you.

None of these ways rely on ASTs-as-closures. And yet that seems to be all that's required to solve the problem of macro hygiene.

When I have this working -- and, after thinking about this for over half a year, I don't see any reason it shouldn't -- I'm going to edit the Wikipedia page to include **Closures** as a sixth option.

Conclusion

Lexical scoping and all of its consequences may be the best idea in computer science, ever. Closures are a natural consequence of taking both lexical scoping and first-class function values seriously.

Function values are shaped not just according to the computation they perform, but also according to the environment they perform it in. This may sound like a weakness, but it's actually a great strength. We can use it to achieve encapsulation and data hiding, just like with objects.

The work on macros in Rakudo is coming along fine. I feel I have gained a deeper appreciation of lexical scoping and closures because of it. And there's more to come.

Bibliography

http://en.wikipedia.org/wiki/Funarg_problem.

<http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg03277.html>

https://en.wikipedia.org/wiki/Hygienic_macro

Author: H.Merijn Brand (h.m.brand@xs4all.nl), (Tux) <http://tux.nl>

Bio H.Merijn Brand

Merijn, in the perl community better known under his alias Tux, is using mainly open source utilities and C to exchange data between sources, porting open source to commercial OS's and support the Open Source community as widely as possible

Treasurer of the perl5 Configure landscape and author and maintainer of several widely used modules on CPAN.

Work currently forces Merijn to write java too.

Abstract

Programming style is and has always been and will always be a point of discussion.

In this Lightning-Talk I will try to show where statement modifiers will cause havoc when trying to understand code someone else has written.

This talk will specifically address a "problem" that only occurs in perl, as other languages like java and C do not have statement modifiers. When working in a development environment where multiple languages are used, *portable* style issues start to be very important.

Availability

The talk will be available after the presentation next to previous talks from Merijn on his site: <http://tux.nl/perl.html>.

All talks on that site use navigat on that is best supported in either Opera or Firefox with Space-Next plugin.



Now you can hack on DuckDuckGo

DuckDuckHack

Create instant answer plugins for DuckDuckGo

duckduckhack.com

Author: Salve J. Nilsen (sjn@cpan.org)

Do you have a Perl Mongers group? Are you one of the clueful people who have figured out the doubleplus-goodness of being part of a strong technical community? Or do you just want to become one of those people? Then keep reading! Maybe you'll pick up some ideas on how to make even cooler stuff happen! :)

Bio Salve J. Nilsen

One of the original Oslo.pm people who seem to end up organizing stuff more often than hacking. Salve is the fellow behind the first Perl QA hackathon, two Nordic Perl Workshops in Oslo, another hackathon for the EPO and Perl 6 people, one more Perl 6 hackathon, and the 2012 „Moving to Moose“ hackathon. Was leader of Oslo.pm during most of its first nine years, and is currently looking for a good employer. :)

Abstract

The Perl community is all volunteer-driven. There's no big company behind it all and no boss that makes sure everything is being taken care of. There is *no one* to pick up this gauntlet but ourselves. *We* are the ones who have to make things happen. If you're not afraid to roll up your sleeves, your community will always be grateful to you if you involve yourself in your local Perl Mongers group. To make your efforts more rewarding, Salve has made a list of suggestions for things to do, based on his experiences from Oslo.pm during the past 10 years.

Some things that didn't work

Oslo.pm was founded a few weeks after YAPC::EU in Munich in 2002. During the first few meetings we've had to figure out what to do. Here are some of the things that we tried and that didn't work.

Expecting anyone to show up all by themselves: Spending time on promoting is *critical*. Even if you don't like doing it, learn to do it anyway. It'll pay off immensely.

Forcing anyone to do something: Volunteerism and force really don't go together. No one is the boss and anyone who helps does it because they're nice or because they get something out of it themselves.

Getting people excited about something they don't care about: To avoid this, find out what people care about! Listen to the guys. Talk with them about the things they care about.

Take notes, and then use them to adapt your plan. :)

Keeping things simple and low-key: Simple is boring. Do something fun now and then, and accept that there's some work involved.

Accepting help from people who tend to say „no“: Getting a „no“ for an answer is depressing, especially from people who start out by saying yes. You need to be able to trust your cohorts, so avoid the „yes-but-no“ and other negative people.

Thinking „someone else will take care of it“: Sometimes this works out, but it's always a gamble. Being certain is *much* better, but that probably means *you* have to do something about it.

There's more! But these failures are for the pub. Find a Perl Mongers organizer, buy them something tasty to drink, and ask them! :)

Some things that did work

Do something that is a little ambitious: Ambition *is* good. For example, invite a community instead of one person. Do something no one else has done before. Buy a round of beer on everyone. Convince a Perl community superstar to visit. Invite people from other communities to a cool Perl event. Find out what other technical meetup groups have done, and try to do it better! :)

Try to spend attention on having a good time: Always be nice to people, and always spend some attention on the new guys. Say hi, give your „loudest“ wave and a nice smile, and ask them if they want to sit down and join the party. Even classic introverts can learn how to be nice to strangers, and who knows - maybe they have something interesting to share? :)

Try to involve your local Perl shops: All of them! The moment you find someone working at a Perl shop, ask if your Perl Mongers group can have a meeting at their place next time. Have a presentation there, and then go out and have a beer (or something) with the other employees there - and now and then you might find someone who wants to come back.

Build relationships with the bosses in different Perl shops: Technical bosses are awesome resources for a Perl Mongers group! You can help them by supplying a good technical forum, they can help with location, and quite likely some neat perks like pizza at meetings or even sponsorship for your more ambitious events. If you can't do it yourself, have someone else take care of it! Let them know that you care about your local technical community.

Organize courses: If you have a good relationship with different companies in your area, ask them what kind of courses they need, and try to help them with this. This might require some work (which you probably aren't afraid of since you're reading this), but in return you get a better community and - if you play it well - maybe even some funds in the .pm group coffers. Having some funds to play with can make a *huge* impact on how ambitious you can get.

Organize a hackathon: Hackathons are very easy to organize, especially if you pick a topic people are excited about and you have good relationships with your local Perl shops. Ask your fellow Perl Mongers about what they care about, and pick something a couple of you are really excited about. Then DO IT!

When it comes to value for time invested, hackathons are *very* good. If you want some ideas on how to do it, check out the author's blog post about *what we can learn when we accidentally a hackathon*: <http://code.foo.no/2012/05/29/a-perl-6-patterned-hackathon>, or search for „organizing a Perl hackathon“ with your favourite search engine.

Organize a workshop: Perl workshops can be any size, from low-key Saturdays with one track of speakers and a social event in the evening, to almost-YAPC sized multi-track conferences. Find someone that want to help you organize the event, and then scale it accordingly. Need pointers on how to do it? Get in touch with some of the people that have done it already! We're a friendly bunch in the Perl community, so just go ahead and ask. Maybe start by dropping by the #yapc channel on <irc://irc.perl.org> and say what you're planning?

Organize a trip: Being part of an Open Source community doesn't have to be all sitting in front of a computer at the office! Why not do it somewhere nice? There are many nice places to visit, and there's nothing wrong in enjoying the world you're trying to improve anyway. :)

If you can't be the first guy, then be the second one: Or rather, if you can't be the *n*'th guy, be the *n+1*'th. Helping someone else can be both much easier and much more rewarding than being „the one in charge“. Personally, I'd recommend anyone to *be the 5th guy*. The 5th guy is part of a bigger group, so there's plenty of support and still plenty of things to do. But if you can't be the 5th guy, *be the 4th guy instead*. Or the 3rd guy if there's no 3rd already.

Why? Because making cool stuff happen is *much* easier and more fun with others. It's all about motivation, and by being the 5th guy, you're showing the others that you care and that you appreciate what they do enough to spend your own time on it.

Do something cool, then tell about it: And finally, when you've done something cool, *tell everyone about it*. Seeing other people do something awesome is actually *nice*! :)

Links

Oslo Perl Mongers

- Main website: <http://oslo.pm/>
- Facebook: <https://facebook.com/groups/oslo.pm/>
- A retrospective: <http://code.foo.no/2012/01/23/an-oslo-pm-retrospective>

Salve J. Nilsen

- Blog: <http://code.foo.no/>
- Twitter: <http://twitter.com/sjoshuan>
- Facebook: <http://facebook.com/sjoshuan>

Other writings

- On the importance of supporting Perl events: <http://act.yapc.eu/mtmh2012/news/878>
- It's the secondary goals that make a Perl event memorable: <http://act.yapc.eu/mtmh2012/news/864>
- A Perl 6-patterned hackathon: <http://code.foo.no/2012/05/29/a-perl-6-patterned-hackathon>
- I'm on a (Perl) boat!: <http://code.foo.no/2011/05/11/im-on-a-perl-boat>

Author: Alberto Simões (ambs@cpan.org)

Biography: Alberto Simões

I am just another Perl hacker. Programming Perl or more than a dozen years, I use it mainly for web development and natural language processing.

My academic formation is in computer science, with a MSc and a PhD in natural language processing using Perl, of course.

My main activity is teaching at university level. Unfortunately I am not lucky to teach Perl. This last year I have taught Bison and Flex in a Languages Processing course, and used a pseudo-language for a course in Artificial Intelligence for Games.

I do research in natural language processing, with emphasis in the Portuguese language. This explains the amount of modules in the `Lingua::` and `Lingua::PT::` name-spaces. The need for speed makes me rely on C libraries which in turn causes my need for building C and C++ libraries using Perl.

Finally, in the Perl Community, I am member of the YAPC Europe Foundation board, the chair of The Perl Foundation Grants Committee, treasurer of the Portuguese Perl Programmers Association and a sporadic Dancer hacker.

Abstract

We all know that the popular *AutoTools* is evil. *AutoConf* is messy, *AutoMake* a confusion, and the lack of an automated way to build this kind of projects on most Linux distributions, Mac OS X and Windows makes these applications portability poor.

If together with all this, your non-Perl code (let's talk mostly of C and C++) will be only used from within your own Perl module, it is a headache and a shame to make available these libraries in tarballs that somehow are not easy to install.

After years of user complains, I decided to bundle some libraries inside my Perl modules. I am sorry for anyone who wants to use the library from another language (I do not think there is such a user yet), but bundling the code in a Perl module that can be automatically installed by any `CPAN` tool is just great. And it is even greater if they work mostly out of the box on the three major operating systems (for Windows I need Strawberry Perl).

In this article I explain how I bundle some modules that include C or C++ libraries, like `Lingua::Identify::CLD`, `Lingua::NATools`, `Lingua::Jspell` or `Text::BibTeX`.

Introduction and Motivation

Before starting, I would like to make a statement: this article is my point of view on how things are in the open source world, and how I choose to solve some of those issues. My solution is not unique, and not the best, surely.

Most C and C++ libraries that are available in open source projects are shipped in tarballs with a *configure* script and a *makefile*. That is an acceptable approach, but it is not the best. Other approaches are available, like *cmake*, but it didn't have much impact (at least, yet).

Most developers keep using the well known *AutoTools*, not because of their quality or even flexibility, but because everybody uses them, and the alternatives are not widespread. Also, there are lots of macros already available, and most *autoconf* scripts are just a bunch of copy and paste blocks from another projects, with minor changes, where maintainers just hope to work, and pray everyday

What are these tools? They are mostly written in Perl (yes, that is true) and they use `M4` macros, a language nobody on earth understands and likes to use. These Perl scripts parse a definition of a configuration file, and generate a shell script, that will try to detect how to build your application. It will detect a C compiler, a bunch of libraries, the size of your integers, where the required libraries are, and a lot of more useful and not so useful variables.

So, for a user to build one of these libraries, he needs to have a Perl interpreter, *autoconf*, *automake*, *libtool*, *m4*, *make*, and all the software needed to really build the library, typically a C compiler (gcc, probably) and all the libraries the software depends on.

This tool chain is too big to be easy to maintain, and to have users to install on non-Linux systems. Even Mac OS X that is supposed to be an Unix system has problems with these tools.

I have been bitten by *AutoTools* a lot of times, having to give support to users to install a library I maintain, just because they want to use a Perl module that uses that library.

I got enough from it, and decided to go all Perl. If *AutoTools* are written in Perl, users need Perl. Then, if I go for Perl as a requirement, I am not requesting that much. Then, I will probably need some modules that are not in the core. But those should be easy to install using any CPAN tool. With this in mind I decided to invest some time, and I got a tool chain of Perl modules to compile any C and C++ code that might be bundled with any of my Perl modules.

In the next section I will describe each of the modules I use in my tool chain for building my modules that include C or C++ code: `Lingua::Identify::CLD`, `Lingua::NATools`, `Lingua::Jspell`, `Text::BibTeX` or `Lingua::FreeLing3`. Then, I will describe briefly how some of these modules' build systems is working (no, I will not detail the code, you can see it on CPAN). At the end I will draw some conclusions.

Do not expect a step by step tutorial of how to write your module build system. Each module is different and has its own specifics. Also, some code blocks might have been simplified.

Modules Tool Chain

In this section I describe the modules used, and for what they are used. I will not enter in detail on how to use them, or how to glue them with each other.

Module::Build

Although the most used module in CPAN, unfortunately `ExtUtils::MakeMaker` has a big limitation: the rules are expressed in a *makefile*, where actions are shell commands. This makes it harder to detect what commands to run, and run them accordingly.

Regarding this, `Module::Build` has a big advantage. As the rules are expressed as Perl functions there is a lot more versatility. It is easy to write a module to subclass `Module::Build` and rewrite some of the methods according with our needs.

ExtUtils::CBuilder

We are talking about building C software, which means we need a C compiler. For that, Perl bundles in its core modules the `ExtUtils::CBuilder` module, that is able to detect, using the `Config` module, which C compiler to use and with which flags. It also has information on how to link a library and how to link an executable. It even guesses some C++ flags quite well.

Its main drawback is that it is prepared to build Perl libraries: what I mean with Perl libraries is, the libraries that are built with XS code and are loaded with `DynaLoader`. These libraries have some differences from the standard libraries, at least on some operating systems. For instance, in Mac OS X, the Perl libraries are known as **bundles** and the standard C libraries are known as **dyld** libraries (standard dynamic libraries). They have different functionality (not that I am aware of the real differences, my knowledge in this aspect is just superficial).

ExtUtils::ParseXS

Also in the Perl core modules is `ExtUtils::ParseXS`. It parses XS files and generates the corresponding C (or C++) code. If you use `ExtUtils::MakeMaker` or `Module::Build` built-in mechanism for building C extensions you are using this module by default. As I am building my extensions manually I need to call it myself, to create the C/C++ glue file.

ExtUtils::Mkbootstrap

Just like the previous module, `ExtUtils::Mkbootstrap` is also a Perl core module, and used by the usual build tools to compile XS code. It is used to create a file used by `DynaLoader` to load dynamically your extension. Again, as I am building everything by hand, it is part of my tool chain.

ExtUtils::PkgConfig / PkgConfig

At the moment I am using `ExtUtils::PkgConfig` to detect some libraries that bundle a `.pc` file. This mechanism was used originally with Gnome (if I recall correctly), and now is common on most libraries. To include a pkg-config file is easy, and can make the life of other programmers easier. So, why not.

The problem with `ExtUtils::PkgConfig` is that it uses the `pkg-config` binary file. This extra dependency is a problem. Some time ago it was discussed that it should exist a pure Perl implementation of this module. `PkgConfig` seems to be that module, but I am not using it yet. But I will probably migrate my build scripts to use `PkgConfig` in the future.

Config::AutoConf

With `Config::AutoConf` I start presenting two modules written by me (and with a lot of patches and contributions) to help me with the C and C++ libraries build process.

`Config::AutoConf` is a module to mimic some of the behavior you can get with `autoconf`. It has some methods to help detect if a library is available, if it can be linked with, if some header files are available, if some binary is available, etc.

To complete some of these tasks `Config::AutoConf` uses `ExtUtils::CBuilder` to build some simple C programs and detect if they build and run properly.

ExtUtils::LibBuilder

Finally, `ExtUtils::LibBuilder` is probably the most relevant module to build standalone system libraries, but also, the module that needs more work. It is a hairy module that uses a set of heuristics to, based on the information from `Config` and using `ExtUtils::CBuilder`, detect how to build a standalone system library. For that, it uses some magic, looks to the system type, tries to build a bunch of files, and if it succeeds, it returns the required flags for building the library.

This module is known to work in all Linux variants, Mac OS X and Windows with Strawberry Perl. I also think it should work with Cygwin, but I didn't check it myself. I would love to have it work with other compilers, like the Microsoft ones, but I do not have them for test, neither the time or interest to do it myself.

My Build Modules

As I stated previously, I build my modules sub-classing `Module::Build`, and rewriting rules to compile the C code, build libraries, etc. This section describes briefly the structure of these modules implementation.

Lingua::Jspell

To start with, let us look into `Lingua::Jspell`. This module is a morphological analyzer, whose code, written in C, is derived from the well known `ispell` (therefore the name: `i++ = j`). It is mainly used for the Portuguese language, but it is language independent (well, at least for western European languages) accordingly with the dictionary used.

Regarding the technical details, `Lingua::Jspell` is composed by C code that is linked into a standard C library (`libjspell`), some C code that should be linked against

`libjspell`, and a pure Perl module (the interface at the moment is performed using a bidirectional pipe, but in the future I plan to interface using XS).

With this description you can argue that the library should be shipped in an independent tarball. You are correct. But we end up noticing that nobody was using the C library by itself, and the work involved in maintaining *AutoTools* scripts was too much.

From the build chain described above, this module uses `Config::AutoConf`, `ExtUtils::CBuilder`, `ExtUtils::LibBuilder` and, of course, `Module::Build`.

The `Build.PL` script's algorithm is composed by:

1)

The script starts by using `C<Config::AutoConf>` to detect the `C<ncurses>` library. In fact, I look for the header file,

```
Config::AutoConf
->check_header('ncurses.h');
```

and then, for the `tgoto` function, for checking link capability:

```
Config::AutoConf->check_lib(
    'ncurses', 'tgoto',
)
```

2)

Follows a big hack to find out where to install the C standard library. I do not want to install it in the usual place where Perl places the XS libraries, or the system will have trouble finding it, for instance, for the standalone binaries. For Unix systems I get the path where Perl would install binaries (usually `/usr/bin/` or `/usr/local/bin/`) and I replace the `bin` portion with `lib64` or `lib`, and check if they exist. I use the first one available. If none is available, I create the folder and cross my fingers.

For Windows I used to install in the `C:\Windows` path, but with Windows version 7 that folder is write protected. The solution (not the best, I know) was to split the `PATH` environment variable and try to write a dummy file in each folder. The first one that allows me that operation is the place where I will place the `dll` file.

3)

All this information is stored both as `Module::Build` configure data (that will create a module named `Lingua::Jspell::ConfigData`) and in the builder notes. I also add build elements, so the `Module::Build` knows where to place the built files.

Example of a builder note being stored:

```
$builder->notes('libdir' => $libdir);
```

Saving in the configure data:

```
$builder->config_data(
    'libdir' => $libdir,
);
```

And defining build elements:

```
$builder->add_build_element('usrlib');
$builder->install_path(
    'usrlib' => $libdir
);
```

Truthfully, not just the library folder is computed here, but also (for Unix systems) the `pkg-config` path (where the `.pc` file should be placed). But I will skip these details.

4)

Finally, generate the build scripts, invoking the `Module::Build` method:

```
$builder->create_build_script;
```

Regarding the `Module::Build` subclass, a lot of more work is needed. `Module::Build` subclasses redefine build rules, where each rule is named `ACTION_actionname`. For instance, `ACTION_code` is called when you run `Build`, after running `Build.PL` for configuring the module.

Usually I subclass this action with a method that just calls my build methods. Also, I prepare a `ExtUtils::LibBuilder` instance that will be used to build the library and compile the source code. This could be done every time I need it, but this way the initial tests performed by `ExtUtils::LibBuilder` to configure the build system are executed only once.

```
sub ACTION_code {
    my $self = shift;

    # create some folders I need to use
```

```
    for my $path (catdir("blib", "pcfile"),
                    catdir("blib", "incdir"),
                    catdir("blib", "bindoc"),
                    catdir("blib", "script"),
                    catdir("blib", "bin")) {
        mkpath $path unless -d $path;
    }
}
```

```
# create the LibBuilder object
# and save it
my $libbuilder =
    ExtUtils::LibBuilder->new;
$self->notes(
    libbuilder => $libbuilder
);

# dispatch every needed action
$self->dispatch("create_manpages");
$self->dispatch("create_yacc");
$self->dispatch("create_objects");
$self->dispatch("create_library");
$self->dispatch("create_binaries");

# and now, call superclass.
$self->SUPER::ACTION_code;
}
```

These are the methods invoked, and how they behave:

create_manpages

As the name says, this method creates manpages from some `pod` files I have to document the C binaries that will be built. This is done searching for all `pod` files in a specific directory, and running `pod2man`.

At the moment I am running `pod2man` binary with exactly this name. This might be a problem for some installations that have a version concatenated in the binary name, or cases in which the binary is not available in the default binary path.

```
sub ACTION_create_manpages {
    my $self = shift;

    # get a list of pod files
    my $pods = $self->rscan_dir("src",
        qr/\.(pod$)/);

    # get our module version
    my $version = $self->notes('version');

    # for each pod file
    for my $pod (@$pods) {
```

```

# compute the man page name
# (and its path)
my $man = $pod;
$man =~ s!\.pod!.1!;
$man =~
s!src!catdir(„bllib“,„bindoc“)!e;

# skip if the file is
# up to date
next if $self->up_to_date(
    $pod, $man
);

# now, run directly the
# pod2man command
`pod2man --section=1
  --center="Lingua::Jspell"
  --release="Lingua-Jspell-$version"
  $pod $man`;
}
}

```

Note that here I place the manpages in the `bllib\bindoc` folder. Citing the documentation, *Documentation for the stuff in script and bin. Usually generated from the POD in those files. Under Unix, these are manual pages belonging to the ,man1' category.*

create_yacc

In this method I compute the C file from the yacc source file. The method is quite straightforward.

```

sub ACTION_create_yacc {
    my $self = shift;

    my $ytabc = catfile(
        'src', 'y.tab.c'
    );
    my $parsey = catfile(
        'src', 'parse.y'
    );

    return if $self->up_to_date(
        $parsey, $ytabc
    );

    my $yacc = Config::AutoConf
        ->check_prog(„yacc“,„bison“);
    if ($yacc) {
        `$yacc -o $ytabc $parsey`;
    }
}

```

Although the generated file is shipped in the module tarball, if `yacc` or `bison` are available, I recompute it. This makes it easy

for me to make changes and use the same `make-file` to build the module.

create_objects

I build the library in two steps. First I compute the object files in this method. To create the object files I do not need to use the `LibBuilder` module, therefore I access the `cbuilder` field in the `Builder` object. Then, search for all the C files, set the compile flags to use `ncurses` accordingly with the detection performed in `Build.PL`, and build each file, if needed.

```

sub ACTION_create_objects {
    my $self = shift;

    my $cbuilder = $self->cbuilder;
    my $c_files =
        $self->rscan_dir('src', qr/\.c$/);
    my $extra_compiler_flags =
        "-g ". $self->notes('ccurses');

    for my $file (@$c_files) {
        my $object = $file;
        $object =~ s/\.c/.o/;
        next if
            $self->up_to_date($file, $object);
        $cbuilder->compile(
            object_file => $object,
            source       => $file,
            include_dirs => [„src“],
            extra_compiler_flags =>
                $extra_compiler_flags
        );
    }
}

```

create_library

The final step to create the library is just to link the object files that were built in the last step in a standard dynamic library (`.so`, `.dyld` or `.dll` accordingly with the operating system). This is one of the places where I need to use the `LibBuilder` object.

The process is quite simple. Start by obtaining the `LibBuilder` object, detect which extension the current operating system uses, and define the object files that are needed for the library. I could search for all the files with `.o` extension, but there are some files that I do not want to include in the library. Therefore, I decided to just list them all.

Then, define the library name and the place where it will be placed, define the linker flags, and run the `link` method in the `LibBuilder` object.

```

sub ACTION_create_library {
    my $self = shift;

    # get details on the builder and
    # lib extension
    my $libbuilder =
        $self->notes('libbuilder');
    my $LIBEXT = $libbuilder->{libext};

    # define what files will be
    # linked together
    my @files = qw!correct defmt
        dump gclass good hash
        jjflags jslib jspell lookup
        makedent sc-corr y.tab!;

    my @objects = map {
        catfile("src", "$_.o")
    } @files;

    # define where the resulting library
    # will be placed
    my $libpath = $self->notes('libdir');
    $libpath = catfile(
        $libpath, "libjspell$LIBEXT"
    );
    my $libfile = catfile(
        "src", "libjspell$LIBEXT"
    );

    # define the linker flags
    my $extralinkerflags =
        $self->notes('lcurses').
        $self->notes('ccurses');
    $extralinkerflags.=
        " -install_name $libpath"
        if $^O =~ /darwin/;

    # link if the library is not
    # up to date
    if (!$self->up_to_date(
        \@objects, $libfile)
    ) {
        $libbuilder->link(
            module_name => 'libjspell',
            extra_linker_flags =>
                $extralinkerflags,
            objects => \@objects,
            lib_file => $libfile,
        );
    }

    # create a folder where to place
    # the library
    my $libdir = catdir(
        $self->blib, 'usrlib'
    );
    mkpath( $libdir, 0, 0777 )
        unless -d $libdir;

```

```

# copy if needed
$self->copy_if_modified(
    from => $libfile,
    to_dir => $libdir,
    flatten => 1 );
}

```

This code could be cleaned a little bit, but probably in a later release.

create_binaries

This method is very similar to the `create_objects` but, instead of creating object files from source files, it will compile binary files from object files. Again, this method will use the `LibBuilder` object for this task.

```

sub ACTION_create_binaries {
    my $self = shift;

    # get details on the builder
    # and binary extension
    my $libbuilder =
        $self->notes('libbuilder');
    my $EXEEXT = $libbuilder->{exeext};

    # define flags
    my $extralinkerflags =
        $self->notes('lcurses').
        $self->notes('ccurses');

    # define the binary that will
    # be created
    my $exe_file = catfile(
        "src", "jspell$EXEEXT");

    # what is the needed object file
    my $object = catfile(
        "src", "jmain.o");

    # if needed, link the executable
    if (!$self->up_to_date(
        $object, $exe_file)
    ) {
        $libbuilder->link_executable(
            exe_file => $exe_file,
            objects => [ $object ],
            extra_linker_flags =>
                "-Lsrc -ljspell $extralinkerflags");
    }

    # and if needed, copy the file
    $self->copy_if_modified(
        from => $exe_file,
        to_dir => "blib/bin",
        flatten => 1 );
}

```


As it can be seen, this division in small tasks makes it easy to follow. And they all have a similar base (very similar with *Makefile* rules): find the source files, apply a command (or more than one) to each of them, and obtain a set of target files.

The usual next step in the build process is to run `Build test`. This invokes the `ACTION_test` method. Usually I would not need to subclass this method, but as my tests need the binary to work, I need it to find the proper library at run time. More important, I need it to find the library it just linked, and not another version that may be hanging in the file system. For that, I just tweak the library search path, taking care to do it correctly accordingly with the operating system in which we are running.

```
sub ACTION_test {
    my $self = shift;

    if ($^O =~ /mswin32/i) {
        $ENV{PATH} = catdir(
            $self->blib, "usrlib").
            ";$ENV{PATH}";
    }
    elsif ($^O =~ /darwin/i) {
        $ENV{DYLD_LIBRARY_PATH} =
            catdir($self->blib, "usrlib");
    }
    elsif ($^O =~ /(?:linux|bsd|sun|solaris|dragonfly|hpux|irix)/i) {
        $ENV{LD_LIBRARY_PATH} =
            catdir($self->blib, "usrlib");
    }
    elsif ($^O =~ /aix/i) {
        my $oldlibpath =
            $ENV{LIBPATH} ||
            '/lib:/usr/lib';
        $ENV{LIBPATH} = catdir(
            $self->blib, "usrlib").
            ":$oldlibpath";
    }

    $self->SUPER::ACTION_test
}
```

Finally, if everything worked correctly, the next usual command would be `Build install`. Or, probably `Build fakeinstall` if you want to test how things would be installed before really installing them. In my case, both `ACTION_install` and `ACTION_fakeinstall` start by calling a custom action named `pre_install`.

The `pre_install` action does some paths cleanups, and copies some files that don't

need to be built to the proper place in the `blib` staging folder. I will not share here the code, as it doesn't have much to talk about, and I prefer not to waste space with it. The more interesting portion might be a call to `fix_shebang_line` and `make_executable` to some scripts that I edit manually, and therefore `Module::Build` doesn't place in the correct place. I also check if the path where the library will be installed is a standard one, and if not, warn the user to add the path to `ldconfig`, or things will not work properly.

The install action has just one extra step, that is running `ldconfig` if it is available, the operating system is Linux and user as root. In fact, I probably should look to the `uid`. In a next release.

Lingua::Identify::CLD

This module is a Perl interface to the Google's Chromium Compact Language Detector (CLD) library. CLD is not available as a tarball at the moment I am writing this. It is available in a Google code repository, only. After talking with its maintainer, I decided to bundle the entire library code in my module.

There aren't many differences from the previous module. The main differences are the use of C++ code, and the fact that in this case I have XS code, and therefore, I need to use `ExtUtils::ParseXS` and `ExtUtils::Mkbootstrap`.

To compile the C++ code, and link the library, the only differences are adding some libraries like `libstdc++`, and set the C++ option to true when invoking the `compile` or `link` methods:

```
$builder->compile(
    object_file => $object,
    source       => $file,
    include_dirs => ["cld-src"],
    extra_compiler_flags =>
        $extra_compiler_flags,
    'C++' => 1);
```

and

```
$libbuilder->link(
    module_name => 'libcld',
    extra_linker_flags =>
        $extralinkerflags,
    objects => $o_files,
    lib_file => $libfile,
    'C++' => 1);
```

Other than that, I need to take care of building the XS portion. For inspiration I gave a look to other modules that do this by hand,

like the case of `Glib`. This will be, probably, the longest method in this article.

The code is commented, but the main steps are:

1)

Define the `xs` source file, the C file that will be generated, and the object file that will be created.

2)

Process the `XS` file with `ExtUtils::ParseXS`, saying I am processing a C++ file.

3)

Given the C++ source file, create the object file from it. For this I use the standard `CBuilder` builder object.

4)

The next step is the creation of a bootstrap file. It is created by `ExtUtils::Mkbootstrap` module. I am not aware of the details of this file, nor why it's not always created. I confess I just copied this bunch of code from another module, and it seems to work. Open source is great.

5)

The next step is building the library that will be loaded by `DynaLoader`. First I define the linker flags, and then use the standard `CBuilder` builder object to create the library. This time I am not using `LibBuilder` because I am not building a standalone one, but one that will be used only by the Perl module.

```
sub ACTION_compile_xscode {
    my $self = shift;
    my $cbuilder = $self->cbuilder;

    my $archdir = catdir(
        $self->blib,
        qw`arch auto Lingua
        Identify CLD`);
    mkpath( $archdir, 0, 0777 )
        unless -d $archdir;

    # set file names
    my $cfile = catfile(„CLD.cc“);
    my $xsfile = catfile(„CLD.xs“);
    my $ofile = catfile(„CLD.o“);
```

```
# create CLD.cc from CLD.xs
if (!$self->up_to_date(
    $xsfile, $cfile)) {
    ExtUtils::ParseXS::process_file(
        filename => $xsfile,
        `C++`    => 1,
        prototypes => 0,
        output    => $cfile);
}

# create CLD.o from CLD.cc
my $extra_compiler_flags =
    $self->notes(„CFLAGS“);
if (!$self->up_to_date(
    $cfile, $ofile)) {
    $cbuilder->compile(
        source      => $cfile,
        include_dirs => [
            catdir(„cld-src“)
        ],
        `C++`      => 1,
        extra_compiler_flags =>
            $extra_compiler_flags,
        object_file => $ofile);
}

# Create .bs bootstrap file,
# needed by Dynaloader.
my $bs_file =
    catfile( $archdir, „CLD.bs“ );
if ( !$self->up_to_date(
    $ofile, $bs_file ) ) {
    ExtUtils::Mkbootstrap::Mkbootstrap(
        $bs_file);
    if ( !-f $bs_file ) {
        # Create file in case
        # Mkbootstrap didn't do
        # anything.
        open( my $fh, „>“, $bs_file )
            or confess
            „Can't open $bs_file: $!“;
    }
    utime( (time) x 2, $bs_file );
    # touch
}

# set linker flags
my $extra_linker_flags =
    „-Lcld-src -lcld -lstdc+“;
$extra_linker_flags .=
    „ -lgcc_s“ if $^O eq „netbsd“;
my $objects = [ $ofile ];
my $lib_file = catfile(
    $archdir,
    „CLD.$Config{dlexth}“ );

if ( !$self->up_to_date(
    [ @$objects ], $lib_file ) ) {
    my $btparselibdir =
        $self->install_path(„usr/lib“);
```

```

$cbuilder->link(
    module_name =>
        'Lingua::Identify::CLD',
    extra_linker_flags =>
        $extra_linker_flags,
    objects        => $objects,
    lib_file       => $lib_file,
);
}
}

```

Of course I could split these steps in different methods, dispatched from here. That can be done in a future release, who knows.

Text::BibTeX

`Text::BibTeX` is a module to parse BibTeX files. It uses a C library, named `btparse` for that task. This library was available (and an old version still is) as a separated tarball, but because of the same reasons already discussed, I bundled the C code in the Perl module.

It uses the same idea of the previous modules, with no big differences.

Lingua::FreeLing3

This module is an API to a C++ library, named `FreeLing`. It is used to perform natural language processing tasks, like parsing, dependency parsing, name entity recognition, etc.

In this case I do not bundle any C code from the library, only a `XS` file generated from SWIG. Some of the code described above was used in this module as well.

The main reason not to use the `Module::Build` capabilities to build `XS` code directly is that I needed to detect `FreeLing` by myself, and detect which libraries it needs to be linked against. Therefore, I decided to take the build system in my hands.

Lingua::NATools

Finally, `Lingua::NATools` is a toolkit for processing and aligning parallel corpora. It has been used by a lot of researchers to extract probabilistic translation dictionaries. At first, I bundled the Perl module inside the `AutoTools` tarball. It was a mess, and few users were able to install and compile it properly.

With the expertise I gained with the other modules (mainly `Lingua::Jspell` and `Text::BiBTeX`) I decided to do the other way around, and included the C part in a Perl module.

The main difference from this tool to the others is the high amount of C and Perl dependencies (the method to detect is the same as above, for `Lingua::Jspell`), and the fact that the tests include some binaries for themselves (that are not installed, just compiled and executed during the testing stage).

For that, the `ACTION_test` method starts by dispatching to a `create_test_binaries` action, very similar to the `create_binaries` action, that builds the binaries.

Final Conclusions and Remarks

All this process is very tedious to maintain, but once I get it working, I do not need to change it much. I am not completely happy with the code quality or maintainability. Also, not happy with `ExtUtils::LibBuilder` implementation. But for now, this seems to work on my major target architectures.

When I have time, and a lower stack of to-do items, I might patch `Module::Build`, or create some `Module::Build::Library`, that automates some of these tasks, as they seem easy to automate. The result of not having that module yet, is that some of the modules I have described have better build implementations than others.

Finally, I would be happy with suggestions, patches, and comments. I think this kind of tool chain is important to the Perl community, and can make the difference for some specific projects.



cPanel®



