Strange Loop Notes

Release

Nathan Yergler

September 15, 2013

CONTENTS

Strange Loop is a conference held in St Louis, Missouri. Topics include emerging languages, concurrent and distributed systems, new database technologies, front-end web, and mobile apps.

These are Nathan Yergler's notes from Strange Loop 2012 and the Emerging Languages Camp. You can find the ReStructured Text source for these notes in the git repository.

CHAPTER

ONE

EMERGING LANGUAGES CAMP

Date 2012-09-23

Location Marriott Union Station, St Louis, Missouri

Wanted to provide a venue for talking about new languages that wasn't for people who had completed a PhD or were working on one.

1.1 Symbiotic Languages: Transpiling into JavaScript

Authors Jeremy Ashkenas

Time 9:30 am - 10:00 am

Session https://thestrangeloop.com/sessions/symbiotic-languages-transpiling-into-javascript

"I like to think of this as the place where the back wood medicine folk of programming languages get to come out of the woodwork and talk about what they're doing."

Only a few people here since the first emerging languages camp, which was in a cramped room in the first floor of OSCON. What made it great was discussion, people asking questions. So do that.

Here today to talk about Symbiotic Language: languages that "compile" to the source code of an existing language. These are interesting because writing VMs is really, really hard, and we've developed some strong VMs that might be 15-20 years old, but they're still useful. HotSpot/Java, V8/Dart. Using an existing VM means you get to pick some of the properties of the host language you think are useful, and cherry pick those, and leave behind the ones you don't want.

There's a spectrum of options between compiling and *trans*piling. Transpiling means just translating some language to the source (or occassionaly the byte code) of the target, host language. Retains a lot of the semantics of the host language, since you're working primarily with keywords and tokens. Once you start to change the semantics (i.e., by implementing "special functions" in your language that the new language calls into to implement). CoffeeScript tries not to add these "features", because they want to stay close to the JavaScript semantics.

Charlie Nutter, who works on JRuby, has explored this: he implemented 99.5% of the Ruby semantics on Java. He's also tried the other approach: implementing Java semantics with Ruby code.

CoffeeScript maintains a wiki of different languages that can be compiled into JavaScript. It's a long list!

Jeremy's experience is primarily around CoffeeScript: "It's *Just* JavaScript". JavaScript has proven remarkably versatile and robust for something designed in about ten days. CoffeeScript transpiles to that, and has come a long way since it was first developed during the first Emerging Langauges Camp [shows graph of StackOverflow vs GitHub from RedMonk; CoffeeScript has grown to something like #11 in two years].

So the choice is what you want to preserve vs deviate. One thing they can't do is negative array indices: you'd need to inject some function into the transpiled source code to support it, since you don't have nearly enough information at compile time.

One thing they *can* do is "everything is an expression". Lots of existing Javascript is expressions, but there are also control flow objects that aren't in plain JavaScript. CoffeeScript enables this by function wrapping: shows examples of using a complex if, a try/catch, and a loop as an expression.

The other semantic change CoffeeScript makes is the addition of classes, which CoffeeScript transpiles into the appropriate prototype declaration. You can also do interesting things like executable class bodies, which allows you to do interesting things like change the body of a class based on some value. [shows example of a Pirate class that speaks in English if century > 1700, otherwise Spanish]

The politics of programming languages are such that even if you build something interesting, there are significant barriers to adoption: "I have to use the JVM", "It doesn't work on the web", etc. By transpiling you give people a shim to start playing with your new language. CoffeeScript is a really interesting case study of this: it's come very far in two years without corporate support or backers.

It used to be that you learned a new language to program on a new platform. With symbiotic languages, we're building the same sort of systems we were already building, but hopefully doing so in a more expressive, clean, powerful way. This means our code has two audiences: other programmers, and other programmers in the target language [I think I got this right] – and by extension, the target platform compiler.

});

1.2 Bandicoot: Code Reuse for the Relational Model

Authors Ostap Cherkashin

Time 10:00 am - 10:30 am

Session https://thestrangeloop.com/sessions/bandicoot-code-reuse-for-the-relational-model

Link http://bandilab.org

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/CherkashinChrobak-Bandicoot.pdf?raw=true

Ostap has been working on Bandicoot for about four years, mostly part time, but recently their time has been increasing. Bandicoot is open source, and hosted on github. Bandicoot began because Ostap was writing code in many languages including Java and SQL. He was fascinated because when he wrote something in SQL, you could write something once and it would work with no records, one record, thousands, or millions of records. And there's been lots of research on relational systems: fast joins, etc. At the same time there's been a lot of research on concurrency and control. But we're using a 40 year old language (SQL) with poor re-use.

Bandicoot is a set based programming system that aims to improve the interface to relational data. Bandicoot is a new language and new runtime for doing this.

Introduces a test case he's going to use: two CSV files, one of books, one of discounts. Example: want to apply discount and find all books with a price greater than 100.0$, and then list all the genres.

Bandicoot runs over HTTP: you write a program, and the runtime exposes the functions over HTTP.

Variables define sets of data, and committed data persists across runs of Bandicoot. Bandicoot understands CSV, so you can post data to a function to store data.

Bandicoot supports eight relational algebra operators: four unary, four binary. The `select` operator allows you to apply a boolean filter to some set of data. `join` is a binary operator that takes two sets and joins them into a single set based on some relationship. The `project` operator selects distinct subsets from a set.

By allowing you to define "functions" that apply operators to sets, Bandicoot provides a way to re-use code and composite functions. Everything is a relation is Bandicoot, so you can easily pass things around for composition.

Working On:

Attribute Sets allow you to define your types (schemas) in terms of composition, as well.

Modules for grouping functionality together.

http://github.com/bandilab

http://mingle.io (try it online)

1.3 Elm: Making the Web Functional

Authors Evan Czaplicki

Time 10:30 am - 11:00 am

Session https://thestrangeloop.com/sessions/elm-making-the-web-functional

Link http://elm-lang.org/

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/Czaplicki-ElmMakingTheWebFunctional.pdf?raw=true

The focus is making the web functional, but the real question is "why elm?". It's inspired by the Kubler-Ross model: Acceptance, Denial, Anger, Bargaining, Depression.

Wants to:

- make GUI programming more

pleasant: reduce the time/headache from idea to reality, make people ask "How was it not this way before?"

- Also wants to make programming

more accessible: no installation required, interactive compiler online. Quick visual feedback. Examples! Easy path from novice to Expert.

These goals are accomplished through:

- functional GUIs: enforces safe

programming practice, plays nice with concurrency, beauty/elegance.

- Accessibility: target the web, be open source, great resources/examples

"But aren't GUIs imperative?" is the objection. Perhaps, but there's a lot to learn from functional programming, and the fact that GUIs have been imperative is an artifact of poor tools. Elms is an effort to get the tools there.

A GUI is made up of computations, graphics, and reactions. The question is how to do each of these in a functional way. We know how functional computation works: it's pretty well understood. Graphics have historically been very imperative, but we've been moving to higher level abstractions, from pixels in a matrix to triangles, to OpenGL, etc. The idea of more abstraction is to reduce the number of steps between "I want a pentagon" and "I have a pentagon".

Elm works with `Elements`: rectangular "things" we put on the screen. Some basic functions that return Elements include plainText, Image, fittedImage. Elm also supports Markdown for text formatting.

Elm attempts to make things that are conceptually simple simple to program. So things like alignment ("put this in the middle") or flow ("put these one after another") are simple to express (unlike, say, HTML).

[shows demonstrations, including Collage, which lets you composite things easily.]

So graphics can be done in a high-level, compositional, functional manner. But how do you handle reactions? If a value is immutable (in a functional language), how do you deal with user input? Elm introduces the idea of time-varying values: a stream of input like the mouse position. By introducing this idea, you also get some signaling/auto-update: things that depend on the mouse position will automatically update when it does. Elm is functional, so you can change the way things react/interact without changing the way the graphics are drawn.

Elm can be used for writing games: imperative game programming is pretty flexible (pixel flipping, etc) – too flexible in the opinion of the author. Elm requires you to use good structure. [Demonstrates Pong using Elm] Every Elm game must have three parts: a model, the state, and the view. You might think of this as the functional equivalent of MVC.

1.4 Plan: A New Dialect of Lisp

Authors David Kendal

Time 11:00 am - 11:30 am

Session https://thestrangeloop.com/sessions/plan-a-new-dialect-of-lisp

"A look at the future direction of programming languages."

Starts by talking about "every programming language ever". We usually think about languages in terms of "System" (fast, low-level, static, compiled, "fast") vs "Scripting" (high-level, dynamic, interpreted "slow"). But in the last few years it's become more common to write "real" apps in "scripting" languages like Ruby, etc. Maybe there's a third side: Embedded Languages. Mid-level, static/dynamic, "latched fence" models, and often quite fast. Plan is an attempt at writing a fast, mid-level Lisp.

So how do languages become successful? A lot of people think that success is correlated to having a large number of libraries. But there are a small number of built-in libraries that are actually really important for composing larger systems: HTTP, JSON, etc in modern systems. And a successful language will have a way to distribute modules (i.e., CPAN).

Traditionally Lisp didn't support a lot of polymorphism for types, and you still see this in Scheme. Plan addresses this by doing pattern matching for types, and tagging objects with type metadata. Plan would like to provide a comfortable path for people working today in Python and Ruby into Lisp-i-ness.

Some familiar syntax can be transparently converted to Lisp syntax:

```
a[b]

  => (get-prop a b)

(set a[b] c)

  => (set (get-prop a b) c)
```

1.5 Clever, Classless, and Free?

Authors

- Håkan Råberg

Time 11:30 am - 12:00 pm

Session https://thestrangeloop.com/sessions/clever-classless-and-free

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/Raberg-CleverClasslessAndFree.pdf?raw=true

Invited to speak, and decided he wanted to talk about some experience over the last few years. He comes from a consulting background, and for a while had to live in the Java world. If not for Clojure, he'd probably still be in the Java world. As such, he was a pragmatist. Extreme programming background: TDD yourself to freedom, testing will save you. Reached the conclusion that there are some mistakes in this approach, and decided to try to move from the pragmatist side of things to the "other" end. So he's taking two years off to explore this.

1.5.1 Enumerable.Java

(Clever)

Took a break in 2010 from corporate job, traveling around Asia, attempting to program while he traveled. Found himself in Kuala Lumpur, near the end of his year of travel, and now it's time to start programming :). Decided he wanted to solve the problem of lambdas in Java. And he did this by working on porting the `enumerable` module from Ruby to Java.

> map(xs, lambda(x, x.toUpperCase()));

But x and lambda are static things that don't really do anything: `toUpperCase` is the part that you actually care about.

[Leftover Lambda, Bill Hoyt]

At runtime you create a new class, move the byte codes over to the new class, and replace the local references to the new class:

```
map(xs, new Fn1<String, String>()
{
    public String call(String x) {
        return x.toUpperCase();
    }
 });
```

This is basically "lambdas for Java 5": load time or AOT compilation. Implemented via ASM Bytecode Macro, and triggered by annotated static methods. But no Java syntax expansion. And using JRuby, he was able to make it RubySpec 1.87 compliant.

(enumerable.org)

1.5.2 shen.clj

(Classless)

Mark Tarver's Shen: A portable version of Qi II, consisting of a kernel lisp (klambda) of 46 primitives. Mark provided two ports: GNU CLISP and Steel Bank Common Lisp. There have been multiple ports: Javascript, Clojure (2), Haskell, JVM. shenlanguage.org

1.5.3 Now

(Free)

As software increasingly structures the contemporary world, curiously, it also withdraws, and becomes harder and harder for us to focus on as it is embedded, hidden, off-shored or merely forgotten about.

> – David M. Berry

1.6 The Reemergence of Datalog

Authors Michael Fogus

Time 12:40 pm - 1:20 pm

Session https://thestrangeloop.com/sessions/the-reemergence-of-datalog

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/Fogus-Datalog.pdf?raw=true

Also known as "Return of the Living Datalog". Michael likes turtles.

A common way we look at data is as a "rectangle" (table). Rectangulation falls down a bit when describing relationships, sparse data, multi-valued, and leads to Place-Oriented Programming (PLOP).

Dealing with data in Java often involves lots of code that obscures what the data actually is: you write a lot of code to accomplish what you want, and mistake the menu for the meal. So Java programmers try to make the data more familiar/comfortable, by mapping it into what they understand: classes. "ORMG!" This reinforces a false dychotomy between code and data.

So how do we unify code and data? Unification. With unification you "punch holes" in your data and provide places for variables. Then you can try to fit data in, like a key in a lock. Unification diverges from pattern matching because it can leave variables there: you don't have to match everything, or the unification can introduce new variables.

Unification closes the gap between data and code, but doesn't quite get us to the point where you can *use* the data in what we'd normally consider a program. Prolog is one of the ways we try to close that gap completely. Prolog programs assert facts, and use rules to reason: X spawned Y, and if A spawned B, B is a descendant of A. This is great: our data can now sort of be treated as code. But it has a few problems: clause-order dependence, non-termination, and imperative infection.

Datalog is a query language (not general purpose, not Turing complete), very explicit with its bindings, and relatively simple. Datalog began its life in 1977, and work was done until 1995 when it was declared "not relevant". In 2002, though, it began to gain use as a way to describe security and topologies.

Datalog is a logic programming language with recursive queries and recursive joins. Datalog works off a simplified Entity Attribute Value (EAV) model. This EAV model means Datalog is suitable for querying sparse datasets.

Datomic is an implementation of Datalog which removes the need for a database, and introduces the notion of "time travel". Instead of always specifying a database, Datomic allows you to pass in a set of raw data. This makes it useful to test your queries. Keeping track of time in a relational database can be tricky. Datomic allows you to specify a fourth field in the tuple as a time field (actually transaction). This allows Datomic to perform total ordering of transactions, and allows you to bound queries by time.

Daedalus is also a Datalog implementation with a notice of time, although it's different from Datomic. Because it's designed to support distributed processing, Daedulus is based on a tick model (with some accommodation for unreliable network connections).

A third implementation, Cascalog, is written in Clojure, and provides map/reduce processing. This also means you have order independence.

Bacwn is another Datalog (also Clojure based?) which provides negation. Negation utilizes a NOT predicate, which lets you take the same query and return the logical "inverse". [Shows example using MST3K characters, querying first for all characters on the SoL, then those not on the SoL.]

So what about query time? Query plans provide one way to do things, but we don't get any guarantee that that's what the engine will do. You can hint your query, but that's a black art. Prolog requires you order for termination, but Datalog requires that you order for speed (many naive Datalog queries will run slowly). Pluggable optimizers may be the way forward: plug in an optimizer than knows your own data without impacting other Datalog optimization techniques.

1.7 Roy

Authors Brian McKenna

Time 1:20 pm - 2:00 pm

Session https://thestrangeloop.com/sessions/roy

Link http://roy.brianmckenna.org/

Roy is an altJS language (http://altjs.org). While he was compiling the list of languages, he played with lots of them, and none of them satisfied his

Writing correct JavaScript is hard, partially because everything is mutable. So Roy is CoffeeScript meets OCaml meets Haskell.

console.log "Hello World"

console.log ("Hello World");

Roy is statically typed:

```
console.log ("40" + 2)
```

Won't work in Roy – incompatible types.

```
let f x : Number = x
```

Defines a function f which takes a Number `x` and returns a Number (inferred in this case).

Roy preserves comments in your source.

You can also do "static duck typing": extensible records. You can specify which properties an object needs to have, then anything with those properties can be used.

This lets Roy provide pattern matching.

Roy provides a module system. When compiling to a browser module, those are assigned to the global scope. But you can also compile to CommmonJS, which will use the require system, as well as an async module def.

1.7.1 Future of Roy

- Functional Lenses
- Deferred type errors

1.8 Julia: A Fast Dynamic Language for Technical Computing

Authors Stefan Karpinski

Time 2:00 pm - 2:40 pm

Session https://thestrangeloop.com/sessions/julia-a-fast-dynamic-language-for-technical-computing

Link http://julialang.org/

Julia is a fast, dynamic language for technical computing. "Technical Computing" is sort of a made up term, but it includes languages like Matlab, Maple, Mathematica, R, SciPy, etc. There are at least forty technical computing languages. Julia had three feature goals: dynamic language, sophisticated parametric type system, and multiple dispatch. Existing languages have some of these, but the combination is somewhat unique in Julia.

Julia can be Matlab-like: code is defined in functions, [shows example]. But you can also write lower-level, non-vectorized code [shows example of their quicksort micro-benchmark]. Julia supports distributed computation, and macros for constructs _like_ distributing computation.

Shows the difference between how Python and C store arrays. C requires that arrays have a single type, so you can store (say) the list of floating point things in contiguous memory. Python lets you do anything in the list, so it stores a list of pointers to the elements. Julia wanted to be able to store the information contiguously in memory, and still have flexibility. This means you can't change a type once it's declared: you can't make it bigger, add fields, etc. [Did I mishear that this means there's no/limited sub-classing?]

Discussion of how Julia handles multiple dispatch. Uses explicit promotion instead of overloading.

[Live coding demo of hypothetical Modular Int type.]

Julia performs very well compared to Python, Matlab, Octave, R. Julia runs on LLVM, so fast to start with, and they're working on performance.

Julia performs no static type checking.

1.9 Rust

Authors David Herman

Time 2:40 pm - 3:20 pm

Session https://thestrangeloop.com/sessions/rust

Link http://www.rust-lang.org/

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/elc/Herman-Rust.pdf?raw=true

A Haiku to describe Rust:

a systems language

pursuing the trifecta

safe, concurrent, fast

Mozilla Research is interested in the platform side of the web: Boot To Gecko, for example. And at a lower level, thinking about how they'd design things for the future. Looking at a set of tabs in a browser, you think about them in terms of security and sandboxing, but Mozilla Research thinks about them in term of threading, performance, etc: like the browser is an operating system that provides a bunch of stuff for "apps".

So why Rust? Mozilla's codebase is *huge*, and lots of C++, which is not designed for security, etc. Started working on Rust in 2009, self hosting on LLVM in 2011. Rust is the love child of C++, Erlang, and Caml, and Haskell.

1.9.1 Fast

Sometimes abstractions remove duplicated code, but introduce performance issues. Rust strives to provide zero cost abstractions. For example, lambdas are aggressively inlined, and you can annotate code to push the compiler towards inlining. Rust also provides C++ like structured data: direct access instead of indirection. [Shows example of a `Point(x,y)` struct. You *can* deal with the indirection if you needed. Rust is currently calling these "borrowed pointers" to reinforce the idea that they're on the stack, and not necessarily long lived. Rust also provides syntax for allocating on the heap, when needed. These can be passed as borrowed pointers, if needed.

Four ways to use a structure:

- Directly

- Borrowed Pointer (*)
- Heap Pointer (&)
- Unique (Owned) Pointers (~)

You can "move" ownership of pointers; doing so effectively eliminates the local variable so you no longer have access.

1.9.2 Concurrent

Actor-like language, so you allocate tasks. When you create a stack, you have a small stack allocated, which will dynamically grow. Task creation is pretty cheap. Tasks can not have pointers to data in other tasks, so they can be garbage collected independently. [Sounds like they're roughly the same as processes.] Tasks can allocate unique pointers, which are created on the shared heap. When a Task needs to communicate with another Task, it simply `moves` the pointer to the other Task. This pointer pass is very cheap. And because it's a unique pointer, you're guaranteed that no other Task points to it.

1.9.3 Safe

Rust has generics (parametric polymorphism). "Classes" are flat structs, and you can attach methods after the fact.

You can declare `traits` which provide a way to do Type Classes [feels like interfaces to me].

ARCs ("Automic Reference Counting") structs require that the data within it must be deeply immutable. These can be freely shared between Tasks.

Even though Rust is designed for safety, there's no such thing as a completely safe language: every language has a way to do something "unsafe". Rust has that, too, but they're branded with a huge hazmat sign, and if you touch them, your code is branded unsafe.

http://smallcultfollowing.com/babysteps

http://pcwalton.github.com/

1.10 Grace: an open source educational OO language

Authors James Noble

Time 3:50 pm - 4:30 pm

Session https://thestrangeloop.com/sessions/grace-an-open-source-educational-oo-language

Link http://gracelang.org/

Slides https://github.com/strangeloop/strangeloop2012/blob/bab046557f9b9f8c90c8152baa09a21fedd0405d/slides/elc/Homer-GraceAnOpenSourceEducationalOOLanguage.pdf?raw=true

Grace is an open source educational OO language.

The languages currently used to teach programming are adequate, but they're tailored for industrial sized problems, with features to match. These aren't always what's needed for teaching.

Grace is targeted at CS1 or CS2 students, with lots of flexibility for how the instructor wants to teach things. This means optional types to support different teaching orders. In Grace, simple programs should be simple. They should have an understandable semantic model, and it should be a general purpose language. [This sounds a lot like my experience teaching: minimal magic is important when you're teaching people how to program.] Grace programs are not supposed to have "incantations" [OK, it's exactly what Vern & I concluded.] Incantations, like "public static void"

in your first Java program aren't meaningless, but they don't mean anything to the beginning programmer. This leads to things like students marking *every* method they write `public static`, which is *not* what they want.

Grace is not exciting like other languages discussed today. It's taking old ideas and trying to combine them for a particular purpose.

Grace distinguishes mutable and immutable bindings through the use of different keywords (`var`, `def`, respectively). And you can teach either functions or objects first, depending on the pedagogy.

Grace has optional typing, and when used, types come after the name. Things that are "more important" come closer to the beginning of the line.

Grace supports "method requests": everything (operators, calls, print, control structures, everything) comes down to a method request. This undergirds the consistent semantic model: no exceptions to explain to beginning programmers. This also implies that there are blocks and lambdas, although you don't have to expose students to them. Finally, because they're method requests, you can add your own control structures for students (i.e., a while block with an explicit invariant).

```
method while(c: Block) do(a: Block) {
  c.apply.ifTrue {
    ...
  }
}
```

Grace provides data hiding support for objects using annotations:

```
def pt = object {
  var x := 2
  var y is readable := 3
  var z is public, readable, writable := 4
}
```

Trying to access x will give no such methods; y will give requested confidential method. And you can also add your own annotations [consistent semantic model].

Grace is designed to prevent null pointer exceptions, to allow students to focus on the basics of writing programs.

Grace is still under development: nothing is set in stone, but some things are "stone adjacent".

1.11 Elixir: Modern Programming for the Erlang VM

Authors Jose Valim

Time 4:30 pm - 5:10 pm

Session https://thestrangeloop.com/sessions/elixir-modern-programming-for-the-erlang-vm

Link http://elixir-lang.org/

Why Elixir? First, the Erlang virtual machine is great, so Elixir attempts to expose the great parts of that VM in a different way, while addressing some of the shortcomings of the host language. The Erlang VM was built for concurrency and for hot deployment of updates. Second, multi-core is here to stay. Erlang was built for concurrency, and a lot of the features that make it great at that also make it great at supporting multi-core hardware.

The goals of Elixir are:

- Productivity

  Elixir attempts to increase productivity by eliminating boilerplate code. Everything in Elixir is an expression, which makes the model more flexible. Elixir also supports macros. The combination of the two features means

that domain specific languages (DSLs) are easy to develop in Elixir. [Shows example of a test case DSL.] Macros also support pattern matching.

- Extensibility

  Elixir's goal of extensibility is a direct critique of Erlang. This is accomplished through the use of Protocols.

- Compatibility

  Being compatible with existing Erlang tooling is an explicit goal of Elixir. There is no conversion cost for calling Erlang from Elixir and vice-versa.

  Elixir works out of the box with existing code like OTP.

1.12 Visi: Cultured and Distributed

Authors David Pollak (@dpp)

Time 5:10 pm - 5:50 pm

Session https://thestrangeloop.com/sessions/visi-cultured-distributed

Link http://visi.pro/

In a lot of fields there's a tension between compatibility and innovation: how much do you do that's comfortable and familiar, and how much do you try to push the bounds of what can be done.

[Demonstration of Mesa.]

[Demonstration of Visi.]

Visi delineates computing modes: sinks (output), sources (input), and references. Things that don't have side effects can be freely moved around. The delineation means there's a clear place to check if there have been changes, and check for serialization safety.

Visi allows you to write a Markdown document with prose and model interleaved. Just write the document as Markdown and put your Visi code in fences.

[This sounds a lot like Python docstrings, and the way Zope was using them during the 3.X days.]

Visi has full type inference, so it won't display prompts for the values until it can infer what the types should be.

CHAPTER

TWO

MONDAY

2.1 In Memory Databases

Authors Michael Stonebraker

Time 9:00 am - 9:50 am

Session https://thestrangeloop.com/sessions/in-memory-databases-the-future-is-now

Link http://voltdb.com/

VoltDB: Not Your Father's Transaction Processing

"It's a pleasure to be here, because I can look out and there's no one in the audience in a suit."

The buzzword du jour in research is "big data". The standard marketing around big data is that I have too much, it's coming too fast, or from too many places. He's focusing on the aspect of "too fast", and high velocity ingest.

Transaction processing 30 years ago was highly intermediated. 1000 transactions per second was considered a stretch goal (HPTS 1985). In this traditional world, ACID was the gold standard, and the workload was a mix of updates and queries. This was the bread and butter of Ingres and Oracle.

In the last 25 years, the intermediaries have been removed, which has led to a cooresponding increase in transaction volume. Transaction processing now encompasses much more than just data processing; it includes things like multiplayer games, social networking, ads, etc. Retaining someone's state in a multiplayer game is a huge transaction processing problem. Ad placement isn't just a TP problem, it's a real-time TP problem.

In addition to disintermediation, the rise of sensor tagging (marathons, taxis, etc) is also adding to the volume of transactions.

High velocity ingest (really anything upstream from Hadoop) adds to the volume, as well.

But the workload still looks about the same: a mix of queries and updates, ACID required, but at two orders of magnitude the velocity and volume.

Reality Check:

TP databases grow in size at the rate transactions increase: everything you see at Amazon before you click "Buy" is non TP. For *most* people, 1TB is a really big transaction processing database. You can buy a terabyte of memory for about $50k (say, 65Gb x 16). Moore's Law has eclipsed TP databases, so it's possible to keep your database in main memory.

Instrumenting the Shore DBMS prototype to understand performance, only about 4% of work is actually useful work: buffer pools, locking, latching, and recovery take the rest of the time.

To go faster than traditional systems, you need to focus on overhead and get rid of *all* major sources of overhead. If you focus on better B-trees, this only impacts about 4% of the path length. So don't bother focusing on the actual transaction. The real gains *must* come from focusing on overhead.

So why give up on SQL to use a NoSQL system? Thirty years ago there was a debate between people advocating SQL and people advocating writing operations directly. SQL won because it could *compile* down to those same operations. Betting against the compiler isn't very smart, and high level languages (like SQL) provide better code, independence, etc. More subtly, stored procedures are good: they let you move the code to the data instead of the other way around, which is faster.

If you actually need consistency, using a system that doesn't provide ACID means you need to write it yourself. "That is a fate worse than death." And if you don't need ACID today, can you guarantee you won't need it tomorrow? You need ACID if any part of your problem consists of saying "Do both A and B, or neither." Examples: funds transfers, integrity constraints, or multi-record state.

"Eventual consistency" means "creates garbage": non-commutative updates, integrity constraints. "What happens if someone buys the last inventory item and the primary fails before it replicates? When that system comes back up, you could end up with -1 in inventory, which is usually an illegal state." Eventual consistency only works when you can apply updates in any order and wind up with the same result.

NoSQL is fine for non-transactional systems with single record updates. It is not appropriate for TP.

So how do you build a SQL-based, ACID-compliant system that performs better than the legacy systems? You need a system that scales to large clusters (one node solutions are no longer interesting), that has automatic sharding, and that focuses on OLTP. By focusing on OLTP problems you have, you can specialize and gain more speed. A specialized hammer will be more effective than a generic system attempting to be both a hammer and a screwdriver.

Storing data in main memory is great, and most main memory databases can spill cold data to disk without significant overhead.

Eliminating the Write Ahead log ("Mohan kool-aid"): modern TP requires high availability, which implies replication and fail over/fail back. So there's no need to recover from the Aries-style write-ahead log. Yabut: what if the power goes out? The WAL is the "slow" option, so you can do periodic check-pointing, with a command log. That log might only use a stored procedure identifier + parameters, with group commit, meaning you log less than a traditional WAL. Recovery time is worse, but total cluster failures are rare [hopefully].

If you eliminate multithreading, you can go even faster, because there's no shared data structures to latch access on. For multicore processors, your system can divide memory into n buckets, one per core, and pretend you're on separate CPUs.

Finally, eliminate row level locking.

VoltDB:

- Subset of SQL (getting larger)
- 70X faster than legacy DBMS on TPC-C
- 5-7X faster than Cassandra using VoltDB K-V layer
- Scales to 384 cores (biggest iron they could find)

Beware of vendors who:

- Use Multi-threaded
- Implements WAL
- Uses ODBC/JDBC for high volume

2.2 Monad examples for normal people

Authors Dustin Getz

Time 10:00 am - 10:50 am

Session https://thestrangeloop.com/sessions/monad-examples-for-normal-people-in-python-and-clojure

Link

Large codebases are copmlex. Technologies like Spring, EJB, AOP, etc all had a common goal: make the code look more like the requirements. If you write composable functions, your boss could write the code: it reads like the real requirements. But in real life, you have to live with NullPointerExceptions. You can write a bind or a pipe higher order function that wraps some of this complexity.

The big picture goal is to write code that looks like the business logic. The difference between an API and a DSL is how well thought out and flexible it is.

Monads are a design pattern for composing fucntions that have incompatible types, but which are logically composable.

[This talk moved very quickly and presupposed quite a bit of knowledge. I was unable to keep up with notes, but reading the Wikipedia page on Monads (http://en.wikipedia.org/wiki/Monad_(functional_programming)) was useful.]

2.3 Functional Design Patterns

Authors Stuart Sierra

Time 11:00 am - 11:50 am

Session https://thestrangeloop.com/sessions/functional-design-patterns

Slides https://github.com/strangeloop/strangeloop2012/raw/master/slides/Sierra-FunctionalDesignPatterns.html

Had the idea for the talk about six months ago at Clojure West. When he went to write it, he realized that "design pattern" is sort of a loaded term. People associate the term with the "Gang of Four", which "sounds like an ominous cult, or worse, a Senate committee." GoF is the product of its time, and described a lot of great starting points, which people took and corupted. Some people say that design patterns are an anti-pattern: that if your language needs them, your language has a problem.

In 1996, Norvig gave a talk where he talked about how most of the patterns in GoF are invisible or grossly simplified in dynamic languages. But then goes on to talk about how at one point a sub-routine call was also considered a "design pattern".

Going to focus on patterns that show up slightly differently in functional languages. And he's not going to talk about Monads, even though some of the patterns he'll describe are monadic.

Monads are useful for writing programs, but he doesn't find them very useful for *explaining* them.

Architectural Patterns: describing an entire system

Design Patterns: describing a specific task/operation

Idioms: low-level patterns specific to a programming language

2.3.1 State Patterns

State/Event

- Derive state from previous state + input
- Need to recover past states (and perhaps inputs)
- Need to visualize intermediate states

This is the pattern of a single function that takes a state and an event. Modeling your state this way is powerful – it allows you to do things like take the starting point and all inputs and reduce to the end state. Makes it a great pattern for testing systems. Allows you to make assertions about the state of the system over time. You also have a lot of flexibility about how you store the state: at one end of the spectrum, you only store inputs/events, not state, since it's derived. One of the downsides is that every input/event in your system has to be a data structure. That's sort of the point, but it can add complexity.

Consequences

- An input to the system can cause multiple events/side effects
- Generated events can trigger state change

One function takes state + input, and returns a sequence of events.

Another applies that sequence of events to state using reduce.

You need to decide if you're going to allow recursive consequences.

A problem with this is that you can't just compose the consequences to get to the current state.

2.3.2 Data Building Patterns

Accumulator Pattern

- Large collection of inputs – maybe larger than memory
- Small or scalar result

One of the essences of functional programming: lazy sequences – map, mapcat, filter, etc – and `reduce`.

This has a built-in assumption of ordered, linear processing, that you're going to deal with things one at a time.

Reduce/Combine

- Input is tree-like
- Divide and conquer approach
- Associative combination of intermediate results (a + b) + c = a + (b + c)

Utilizes a reducer and a combiner function. The combiner provides a way to "roll up" one level to a level "up". Doesn't assume linear processing (hence the associative requirement). In some simple cases (addition, for example), the reducer and combiner may be the same function.

Recursive Expansion

- Build a result from primitives
- Abstractions are built in layers
- Recurse until there's no more work
- Examples: macro expansion, Datomic transaction fns

A function takes an expander and some input, and calls expander with input (and after the first call, the result of the previous call), until the return value equals the input value.

2.3.3 Flow Control Patterns

Pipeline Pattern

- Some process with many discreet steps
- one execution path – no branching
- Each step has a similar "shape" – a map or record in Clojure

Because each step needs to take and return the same "shape" of data, the code can wind up being a little longer. But the result is very clear: you can easily see the steps that are being taken. And because you have to work with the same shape of data, the resulting pipeline is composable into other, larger pipelines

Wrapper Pattern

- Similar to the Pipeline
- Possible branch at each step

Instead of composing a list of functions (steps), you use higher order functions that could do something before or after an individual step.

Because each step can do things before and after, it can become difficult to reason about where something is happening.

Token Pattern

- An Operation may not have an identity
- But you may need to cancel it

So you wrap the operation with something that returns a "token" – something that can cease the operation and get you back to your original state. The scheduled thread pool in Java works this way.

Observer Pattern

- Register an observer with a stateful function

The observer could take the old and new state, along with either the delta, the triggering event, or the container.

Strategy Pattern

- Many processes with similar structures
- Extension points for future variations
- This is a GoF pattern which starts to disappear in Clojure

Clojure protocols are an implementation of this. Another way to do this is by passing around a map of the functions. This *feels* functional, but it has some performance overhead: every invocation requires a map lookup.

2.4 A Type Driven Approach to Functional Design

Authors Michael Feathers

Time 1:00 pm - 1:20 pm

Session https://thestrangeloop.com/sessions/a-type-driven-approach-to-functional-design

Been thinking about how we design functional programs. It's gained ascendancy in the past 5-10 years, but you don't hear people talk about how you *design* those programs. This is in contrast to ascendancy of OOP, when everyone had an opinion they were happy to share.

Had this thought that the Haskell type signature was useful for describing how to assemble programs. Wrote some Ruby code, and even in Ruby he was adding comments to describe expectations that *looked* a lot like Haskell type signatures. For example:

```
map :: (a -> b) -> [a] -> [b]
```

Describes the function `map` that takes a function as a parameter that takes a and reduces to b; it can also take a list of "a" and return a list of "b". There's no clear demarcation between input and return, the return value just happens to be the last thing returned.

```
region 7 9 "expersexchange"

region :: Int -> Int -> String -> String
```

So `region` is a function that takes two indices and a string, and returns another string.

Thinking about a line break algorithm in these terms. Describing the steps in terms of types helped him understand the "shape" of the data.

2.5 A Whole New World

Authors Gary Bernhardt

Time 12:20 pm - 12:50 pm

Session https://thestrangeloop.com/sessions/a-whole-new-world

Link

An Editor

Layers Overlaying orthogonal information Diff layer Crash layer

[I stopped taking notes. This was awesome, even if Gary is a liar ;)]

2.6 The High Order Rubyist

Authors Robert Pitts

Time 1:30 pm - 1:50 pm

Session https://thestrangeloop.com/sessions/the-higher-order-rubyist

Functional programming in Ruby

So why bother? Ruby lets you write succinct code and easily create DSLs. Unfortunately as you write larger systems, maintenance becomes an issue. His theory is that by approaching problems more directly and declaratively will help with this issue.

Ruby includes some batteries that get you started on this approach. The Enumerable module includes common higher order functions. It *also* includes some helpful destructing syntax.

Ruby also has some first-ish class functions that look a lot like lambdas: blocks, procs, lambdas, [something else].

Ruby 1.9 added `Proc.curry` to support currying and partial application.

Continuations previously had a bad reputation in Ruby – poor performance and memory usage. `callcc` improves that performance. Note that continuations may be removed from a future version of Ruby, still under discussion

Tail call optimizations are available in Ruby 1.9. Not enabled by default, so you can either recompile or pass Ruby configuration.

In addition to these built-ins, there are other attempts to add functional properties to Ruby.

- Hamster

  Similar semantics to core Ruby, except for some cases to address [im]mutability.

- Stunted

  Allows you to define bags of functions in a module, and then mix those into other classes. Nice for composability.

- Ruby#Facets

  Well tested and been around a while.

- Celluloid

2.7 Building an Impenetrable ZooKeeper

Authors Katheleen Ting

Time 2:00 pm - 2:50 pm

Session https://thestrangeloop.com/sessions/building-an-impenetrable-zookeeper

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/Ting-BuildingAnImpenetrableZooKeeper.pdf?raw=true

ZooKeeper is the unsung hero, and a lot of time people don't know that it's there until it's down. Because ZooKeeper is so important, it's important to make it durable.

ZooKeeper is fairly stable, so more often the things that bring ZK down are misconfigurations, not bugs.

ZooKeeper is a coordinator for distributed applications. It is designed to remove the need for custom coordination code/solutions. ZK is used by HBase, HDFS, Solr, Kafka, etc.

Misconfigurations are any diagnostic ticket that require a ZK/config file change. These comprise 44% of tickets at Cloudera [eep!]. Typically ZK is straight forward to set up and operate, and issues tend to be client rather than ZK issues.

A 3 ZK Ensemble consists of three ZK machines: one leader, two followers. All three store a copy of the same data. This full replication ensures durability.

Leader is elected at startup, changes are coordinated through the leader, and clients talk to followers. Changes are accepted when a majority of ZKs agree.

Common Misconfigurations:

1. Too Many Connections
   - ZK has a limited number of connections; defaults to 60 [per IP?]
   - HBase clients have leaked connections in the past, so they have to be closed manually
2. Connection Closes Prematurely

   Need to increase wait time for recovery. [Didn't understand this completely.]
3. Pig Hangs Connecting to HBase
   - Caused by Pig not knowing the location of the ZK quorum.
   - This can be resolved with Pig10
4. Client Session Time Out
   - ZK defaults session timeout to 40s, while HBase needs a 180s timeout for garbage collection.
   - This will cause them to agree on the shorter session timeout
   - HBase will begin to timeout under IO load, because it needs more time
   - This may also be caused by co-locating ZK with something IO intensive like a DataNode or RegionServer
   - ZK has relatively low IO requirements, but durability requires that changes fsync before reporting as accepted.
5. Clients Lose Connections
   - The ZK transaction log is optimized for mechanical spindles and sequential IO
   - SSD provides little benefit to ZK, and suffers from latency spikes
6. Unable to Load Database: Unable to Load Quorum Server
   - If there is disk corruption, ZK will refuse to load
   - If you have two other running ZK instances, you can safely wipe the database and it will replicate from the other two when it comes back up
7. Unable to Load Database - Unreasonable Length Exception
   - ZK allows a client to set data larger than the server can read from disk
   - ZK includes the metadata when calculating the size
   - Increase the max size to work around
   - This is a bug in ZK which will be fixed in a future release
8. Failure to Follow Leader
   - If your ZK nodes comes up and is not the leader and can not contact the leader, it will not be able to restart

Because ZK operates by majority, recommend having an odd number of servers in an ensemble: if you have 2 servers in an ensemble, and one goes down, you're down (1 is not a majority of 2). Recommend:

- 1 if you only want coordination
- 3 if you want reliability for production environments
- 5 if you want to be able to take one down for maintenance

But more isn't always better: more servers means you need to wait for more votes in elections. You can use Observers to provide more followers that do not participate in elections.

You can verify the configuration using zk-smoketest.

Best Practices

- Separate spindles for dataDir and dataLogDir – improves latency and avoids competition
- Allocate 3 or 5 servers
- Run zkCleanup.sh via cron

2.8 Principles of Reliable Systems

Authors Garrett Smith

Time 3:30 pm - 4:20 pm

Session https://thestrangeloop.com/sessions/lessons-from-erlang-principles-of-reliable-systems

Reliability

This is unsexy like the 1980s Automotive Quality Wars. The lesson from the quality wars is that things that break suck. But things that keep going are awesome. Awesome like the Terminator who keeps going after getting shot in the face with a shotgun. People will spend money for quality, and they will develop loyalties to things that don't break. They will avoid things that break (or that are perceived to break).

Reliability (quality) is central to the Erlang community, in large part due to the [mythic] number of 9 9's of uptime put forward by Joe Armstrong. Erlang was a commercially motivated language, not academically motivated, so quality and reliability had an associated cost for the creators. Erlang came out of PLEX, another language Ericsson designed. PLEX was a real time, very parallel language, but it was very low level, and therefore very expensive to use. The idea was to find or develop a new language that had an OS independent VM and had great support for parallelism and concurrency. Erlang was the product of this, and because of its motivations it prizes pragmatism over purity.

Principles of Reliability

- Isolation
- Fault detection and Recovery
- Separation of concerns
- Black box design
- State management
- Avoid complexity

2.8.1 Isolation

The Erlang VM is designed to operate processes. You can kill individual Erlang processes without impacting other processes on the VM. We see isolation all around us in the physical world, sort of by definition. So we need to think about how to apply the same ideas to software and think about how to keep things isolation from one another.

2.8.2 Fault Detection and Recovery

In order to recover from a failure, you need to be able to detect the failure first. This isn't as easy as it sounds, especially at the thread level. When thinking about how to detect failure, you want to detect this as quickly as possible to "fail fast". Once the failure has been detected, you need a strategy to recovery. Erlang addresses this by "turning it off and on again" – restarting the piece of code that registered the failure.

2.8.3 Separation of Concerns

Separation of Concerns is the principle of focusing on one thing, and doing it well. [Cohesion, etc.] By keeping code focused, it's easier to reason about it, test it, and limit the scope for a change. This also means that if something fails, the scope of failure is limited.

2.8.4 Black Box Design

Black box design is an approach to designing things where you treat your components as an appliance. The appliances in your home may be quite complex (washer, microwave, etc), but the interface it presents is limited by design. Thinking about code as an appliance means you try to make it easy to set up (just plug it in?), push the start button, provide minimal controls, and reboot or replace to fix.

Erlang is effectively an "operating system" for your code. So you write "systems", and individual "programs" within that service handle some specific concern.

2.8.5 State Management

Erlang doesn't "hate" state, but it doesn't like it very much. Messing with data is costly, and as soon as state enters your application you have to deal with additional complexity during recovery, failover, repair, and synchronization. All of these are hard to get right. If you can avoid state, you should, either by avoiding it completely, or ensuring it's someone else's problem.

2.8.6 Avoid Complexity

Reducing complexity means there's fewer edges to test. Things like dependencies, hierarchies, resource sharing, and fear all are indicators of complexity. Something simple is something reliable. And if something isn't completely obvious, spend some time making sure someone else could understand it.

2.8.7 How to Do This

OS Process Isolation

- No shared memory
- Communicate via message passing
- Process termination ("fault") detection
- Techniques: IO "servers", 0MQ, TCP/HTTP

Actors

- Processes have overhead, and at some scales aren't feasible
- Actors provide semantically isolated memory
- Inter-thread communication via message passing (queue inserts)

Fail Fast

- Avoid defensive practices – let things fail
- Let exceptions propagate and log them
- Use assertions – and leave them in!
- Exiting the process isn't a bad idea if you're running under a supervisor
- runit, launchd provide process monitoring/supervision [speaker recommends runit]

Think Small

- Narrow the scope as much as possible
- Aim for functional-style programming – average functions are four lines long
- Think about a Micro SOA
- Avoid building for "the future"

2.9 Engineering Elegance: The Secrets of Square's Stack

Authors Bob Lee

Time 4:30 pm - 5:20 pm

Session https://thestrangeloop.com/sessions/engineering-elegance-the-secrets-of-squares-stack

@crazybob

As CTO of Square, solely focused on technology, not managing people.

2.9.1 Persistent Queues

Card processing has two phases:

1. Authorization Sets aside the funds for the merchant.
2. Capture "Commit"

This actually works pretty well for mobile: you enter the amount and the authorization occurs. After the signature screen, you can capture (commit). Since the mobile network is inherently faulty, capture occurs in the "background" – capture is queued, and it will eventually be handled.

This gives the feeling of responsiveness, but this is obviously critical code, so what happens if it fails? Their answer is *persistent queues*. SQLite was an open, but felt like a mismatch for building a queue (b-tree vs fifo, etc).

Needed atomicity and durability. Began by investigating what you get from the filesystem. Renaming is atomic, fsync is durable (unless your hardware lies to you), but there were questions about whether or not segment/block writes are atomic. After investigating, it appears that you can rely on them being atomic.

There are a few traditional strategies for implementing this sort of application:

- write / fsync / rename / fsync (note you need both fsyncs to avoid operation re-ordering where you'd wind up with an empty file)
- rollback log
- journal

Solving a specific use case (FIFO queues), so built and open sourced Tape (http://github.com/square/tape/) to accomplish this.

2.9.2 Server Stack

Original stack written using Ruby, but ran into scalability problems. Ruby requires one process per request, and isn't great about sharing memory. This limits you to roughly 20-25 request workers per machine, which isn't great for a high volume transaction processing.

Tried using JRuby to take advantage of better concurrency support, but many of the underlying libraries weren't built for concurrency. All the core payment processing code now uses Java.

The port to Java began about 1.5 years ago. Bob spent the previous five years at Google, who has their own proprietary JVM stack. In addition, Bob had been on Android for 3 years, so he was out of touch with the latest and greatest. What he found:

- One Repository for all the Java code
  - Don't version internal dependencies, everything builds against master
  - This means that if you want to change something low level, you also have to fix everything that uses it
- One JAR for deploying the application
  - Hot deployment of WAR files often has problems with memory leaks
  - Instead of an application server, they use a single JAR that has a `main` method that runs an embedded server.
  - Typical way to do this is unpacking and repacking into a single JAR.
  - OneJAR (available on SourceForge) actually allows you to do nested JARs. This speeds up the build and delays when you run into the 64000 file limit per JAR
  - This does mean you can't do classpath scanning (but you probably shouldn't be anyway).
  - You can also do self-executing JARs
- Java Stack
  - Jetty
  - JAX-RS (Jersey) – higher level abstraction over Servlets
  - JPA (Hibernate)
  - Guice
  - *Dropwizard* – they don't use it, but probably going to migrate there.

2.9.3 Rethinking Publish / Subscribe

Lots of messages to pass around; i.e., payment processing needs to tell a capture system, a settlement system, risk systems, etc about things that happen. The standard approach would be to use a messaging server for that, with reliability implemented with additional messaging servers. The producer and consumer are probably already HA systems, so this adds another cluster to deploy and maintain.

Instead of messaging, they're using a message feed based system.

1. Client asks for all records
2. Server responds with the records and *current version*

3. Later, Client asks for the delta since the last version it saw
4. Server responds with delta

Requires:

- Immutable sequence of events
- Total ordering
- Centralized server

Benefits:

- Stateless
- Fewer moving parts
- Bootstrapping for free

Further ideas:

- Partitioning feeds
- Caching and replicating feeds
- PubSubHubBub

2.9.4 Dependency Injection

Guice is a dependency injection container developed at Google in 2006.

Guice uses a DSL written in Java, which means that for a tool to understand your Guice configuration, it needs to execute your module.

If they could go back and do it again, maybe code generation would be a better way to go than a DSL. Java has an annotation processing API (JSR-XXX), which Bob served on the expert group for, which hasn't gained much adoption.

Guice also has provider methods, which if they had existed in Guice 1, might have obviated the need for the binder API.

Developed Dagger (http://github.com/square/dagger), which applies the learnings from Guice for better dependency injection. Use of code generation means that many errors are compiler errors instead of run time errors. Resulted in increased startup speed for their Android applications.

Engineering blog: http://corner.squareup.com

CHAPTER

THREE

TUESDAY

3.1 Computing Like the Brain

Authors Jeff Hawkins

Time 9:00 am - 9:50 am

Session https://thestrangeloop.com/sessions/computing-like-the-brain

Inspired by an article by Crick who wrote about the brain, saying that we're missing a broad framework to interpret neuroscience data; it was a data rich but theory poor field. Difficult getting a gig doing neuroscience full-time, which is why he wound up doing Palm and then Handspring.

Gave himself two tasks:

1. Discover operating principles of the neocortex
2. Build systems based on these principles

Start with anatomy and physiology, which are constraints on how the theoretical principles could work, then you develop some principles, and model them in software. Eventually that software gets written to silicon. There's tons of papers published on A&P of brain that are unassimilated by theory.

The neocortext is a predictive modeling system. It's responsible for generating and processing our senses. And senses are not single sense" they're arrays of senses: your retina is an array of a million sensors, streaming data in at an incredible rate. The brain is born with incredible capacity, but no knowledge. So the brain has to build a model of the world. When you see something – like someone speaking on a stage – your brain is invoking the model and making predictions about what will happen next, and using those to detect anomalies and deltas. And finally it generates actions, like speech. The brain is not a computing system, it's a memory system.

Top three principles of the neocortex:

The neocortext is a hierarchy: sensory information bubbles up the hierarchy, and then signals are pushed back down. And it's interesting because it appears that everything in the neocortex works the same: a single algorithm for sight, hearing, touch.

The primary memory in the neocortext is sequence memory. When you speak, you're playing back things you've learned in time sequence. And when you hear something you're processing a time sequence of inputs. Even vision works this way.

The brain uses sparse distributed representations. At any given time only a few cells are used.

Computers typically use dense representations: a few bits, using all combinations of 1s and 0s. The individual bits don't really mean anything – the representation is given meaning by the programmer. Sparse distributed representations (SDRs) have thousands of bits at minimum, with few 1's, mostly 0's. Roughly 2% are active at any time, but each bit has semantic meaning. That meaning is *learned*, not assigned. When you want to represent something in the brain, the brain picks the top best matches of "bits" for that information.

SDR has some interesting properties: you can compare two SDRs, and if they have shared 1 bits, they have semantic similarity. Because they are sparse structures, this is unlikely to happen by chance. You also don't need to store *all* the bits (since they're mostly 0), you can store the indices to the positive bits. You can also sub-sample – it's mathematically demonstrate that it's OK to only store the top 10 bits. Even if you have a false positive, it's unlikely to happen, and if it does, it's going to be semantically similar (so not really a false positive). Finally, if you take the union of a set of SDRs, you can compare any new SDR's positive bits to the set union's positive bits and accurately establish membership. Intelligent machines will be built on SDR.

Sequence memory has properties that act as "coincidence detectors". If the same stimuli arrive at the same time, they have a large impact on the cell body. If they arrive one after another, they do not. The cell can "or" them together to determine when a coincidence occurs.

Cells become active from input from the world, and then form connections to a sub-sample of previously active cells. That allows it to predict its own future activity. Multiple predictions can occur at once. Sequences of predictions are established using "layers" of cells – with 40 active columns and 10 cells per column, you get 10^40 ways to represent the same input in different contexts. This allows the brain to understand the difference between "two", "too", and "to".

To build an online learning system, you have to train on every new input. If a pattern does not repeat, forget it. If it repeats you reinforce it. For many years we thought of learning as the strengthening of synapses. That happens, but we know today that synapses *grow* (in a matter of seconds), so it's more useful to think of synapses as forming and unforming.

These models are being applied to predictive analysis. Today we take in data and store it in databases, and then build models and visualizations. The challenges are data preparation (velocity is too fast), model obsolescence, and the lack of people who can do the work. The future of this is taking data streams and feeding them to online models which lead to actions. The requirements of that application are automated model creation, continuous learning, [something else].

Grok is Numenta's engine for acting on data streams. The product feeds data streams through encoders to generate SDRs, feeds them to sequence memory, predict anomalies, and generate actions.

Users create the data stream, and define the problem – what to predict, how often, and how far in advance.

[Shows examples of Grok applications]

Predictions aren't either right or wrong – there's subtlety, and even missed predictions (or things that happened that you didn't predict) can help train the system.

Future of Machine Intelligence

More theory that needs to be developed — sensory/motor integration, attention, more hierarchy research. (People used to think there was a motor "part" of your brain, but we now know that every part of the brain has motor output.)

Today we're building these models in AWS, but you can imagine in the future you could build distributed hierarchies using distributed sensors or networks, much like the brain is hierarchical.

Currently need to do lots of tricks to make this fast in software, but talking to hardware companies about how they might make it faster, cheaper, and lower powered. Interesting implications for memory, but interconnects are a little challenging – chips aren't good at lots of connections like these have (but sub-sampling and sparsity might help).

The applications today are around prediction/anomaly detection. The class applications that people think of are speech/language/vision, but not sure that's very interesting. The interesting thing to him is building systems that can work faster than the brain.

3.2 Behind the Mirror

Authors Chris Granger

Time 10:00 am - 10:50 am

Session https://thestrangeloop.com/sessions/behind-the-mirror-the-birth-of-light-table

Link

In 1974 people at the MIT AI Lab were writing code using TECO – the Text Editor and COrrector (actually "Tape" Editor, because that's the medium they were using). What made TECO interesting was that it wasn't an editor like we think about it – it was a *language* for text manipulation. In the original paper for TECO they coined an interested acronym – YAFIYGI: "You ask for it, you get it."

One of the people at MIT thought like we might today – that this was cumbersome. He visited the Stanford AI lab, and saw they had a different way of editing text: WYSIWYG. Returning to MIT, this individual – Stallman – begin writing macros on top of TECO to create a more functional WYSIWYG editor. That became Emacs, and led to a huge increase in usability.

Thirty five years later, he was hired as the program manager for Visual Studio, eventually owning C# and VB. He was asked to think about the future of the IDE, but the underlying question – how do people use it now – didn't have a satisfactory answer. He found no one had done an end to end analysis of Visual Studio. They'd studied individual new features, but not the product as a whole. Granger did a usability study of Visual Studio. One interesting thing he found is that the people who swear they don't touch the mouse actually *did* use it. They used it when they were reading the code, though, not necessarily while writing it.

Granger was expecting to find that Visual Studio was too complicated, that it's too "noisy" (distracting). There was some evidence that this was true, but no vocalized it: no one mentioned out loud that their attention was divided or diverted. His conclusion is that they didn't vocalize it because they were too busy trying to do something else: trying to keep the state of the program in their head. The primary things they used were the editor, the explorer, and the debugger. This felt really similar to the way things worked forty years before.

Granger set out to try and re-imagine the way tools work. But the first work wasn't Light Table. He started learning Clojure, a lisp that runs on the JVM. Learning/using Clojure taught him that the important thing wasn't keeping the entire program in their head, it was abstracting away enough of the program to keep one part in their head. "Great programmers are able to create and traverse abstractions." Programmers deal with abstraction – there are frameworks for everything, and we as programmers create and consume abstraction. Programmers learn about abstractions by "poking" at them.

Wrote a prototype of Light Table in six days while he was on vacation, without a network connection. People responded well, and asked to put it on Kickstarter [really?!], and Granger expected it to fail. Instead it raised over $300,000. His conclusion: people agree we're in the dark, and we're disconnected from the systems we're building.

[Demonstrates using Light Table to build a tool for showing git status and modeling abstractions from a game he's been writing with his brother.]

3.3 Apache Cassandra Anti-Patterns

Authors Matthew Dennis

Time 11:00 am - 11:20 am

Session https://thestrangeloop.com/sessions/apache-cassandra-anti-patterns

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/Dennis-ApacheCassandraAntiPatterns.pdf?raw=true

Don't run C* on a SAN. Cassandra was designed for commodity hardware, so it didn't really plan for SAN/high performance hardware. It's not only unnecessary, it actually performs worse on SANs than it does on commodity hardware. C* uses (un)coordinated IO, so each node assumes it has local disk and attempts to maximize the bandwidth it uses. If you try to use a SAN, you wind up hammering your SAN.

Cassandra uses a commit log used for recovery; putting it on the same volume as the data directory causes problems because they have conflicting IO patterns. Commit logs are 100% sequential appends, but the data directory is usually

random reads. The commit log is very sensitive to other processes moving the disk head. This problem only shows up under load, so it's sometimes difficult to find when testing.

Oversize JVM heaps are an issue – 4-8G is good, 10-12 is fine ("correct" or "not bad"), 16GB is the max. Greater than 16GB is a problem, as is setting the JVM heap to the same size as the RAM on the box. This is due to increasing "GC suckage".

Scheduled repairs should be run with "-pr". This prevents it from communicating work to other nodes, therefore reducing the work load from duplicated work.

C* requires a lot of file handles, so the common default of 1024 is absolutely not sufficient. This *does not* show up in testing, even when testing with large datasets. It shows up with load spikes, and fails in unpredictable ways. 32K-128K is common.

Putting a load balancer in front of C* is completely unnecessary and only adds another point of failure. The clients will usually balance between the available nodes on their own without this.

Sometimes people try to restrict clients to a single node. This actually takes work, and causes problems. Don't do it.

Having an unbalanced ring used to be the number one problem encountered. An unbalanced ring leads to hotspots on the node with a larger range. OPSC automates the resolution of this with two clicks, even across multiple data centers. Related to this, always specify your `initial_token` instead of letting C* pick for you. The initial token specifies where in the range of 0 to 2^127 the node sits.

The Row Cache is a Row Cache, not a Query Cache, Slice Cache, or any kind of Cache. Asking for *less* than the entire row requires deserialization of the cached row to pick out the pieces you want, working against the Row Cache. If you ask for the entire Row, it will use the cached version that's stored outside the JVM (which means you need to take it into account when sizing memory for the machine). If you turn on the Row Cache, ask for the entire row. Related, large (2GB) rows are still a problem for the cache.

If you think you need the Byte Ordering Partitioner (formerly Order Preserving Partitioner), you probably don't. [He didn't say why, just that it's a big problem.]

Batches are set in a single message and must fit in memory on both the client and server. This makes unbounded batches a real problem. The best batch size is an empirical exercise based on your specific load, hardware, data, etc.

Rotational disks require seek time – and Cassandra was designed for them. 5ms is a fast seek time, but remember that that's hard overhead for your queries. SSDs solve this, but remember that this is overhead on top of the software when you use rotational disks. Note that you can totally run C* on consumer SSDs, doesn't need "Enterprise" SSD.

C* usually deals with Big Data, so a 32 bit JVM usually doesn't work.

If you're running C* on AWS, EBS Volumes are problematic. They have nice features, but they're unpredictable. A better approach is striping ephemeral drives and spinning up new nodes when one fails. It's not clear whether provisioned IOPS EBS are a good fit.

At this point your should be running a Sun^WOracle JVM – r22 or later. Some people are successfully using OpenJDK, but it hasn't been well tested.

Super Columns have 10-15% overhead for both reads and writes: the entire super column needs to be held in memory. Most C* devs dislike them.

DataStax offers a free version of their Ops Center – no excuse not to use it.

http://slideshare.net/mattdennis

3.4 Programming by Voice: Becoming a Computer Whisperer

Authors Tavis Rudd

Time 11:30 am - 11:50 am

Session https://thestrangeloop.com/sessions/programming-by-voice-becoming-a-computer-whisperer

Link

Wants to demonstrate how he programs using voice. He used dictation a few years ago to escape RSI, but it's now a serious technology you can use even if you don't use RSI.

[Demonstrates writing Clojure – in Emacs – by speaking into a microphone.]

[Demonstrates writing Emacs Lisp by speaking.]

3.5 Eventually Consistent Data Structures

Authors Sean Cribbs

Time 1:00 pm - 1:50 pm

Session https://thestrangeloop.com/sessions/eventually-consistent-data-structures

Link

Works one Riak, an eventually consistent data store (which some people may call a database). Voldemort and Cassandra are also eventually consistent. Riak is not ACID compliant, as we heard yesterday.

We have lots of duels/duals in CS – OOP v Functional, etc. The duel/dual of safety vs liveness was defined my Lamport in 1977 in "Proving the Correctness of Multiprocess Programs". Safety means "nothing bad happens" (partial correctness), where liveness means "something good eventually happens" (termination). Forcing or encouraging one property will reduce the other. Peter Bailis talked about this in his blog post, "Safety and liveness: Eventual consistency is not safe". It's not safe *by itself*.

With Eventual Consistency, you have multiple independent actors who are replicating data amongst themselves with loose coordination (for both reading and writing). They also have convergence – moving towards a single shared state. If you don't have convergence, you don't necessarily have *inconsistency*, but you definitely don't have eventual consistency. Unlike ACID systems, EC systems do not have total ordering of events. [This is a problem for some people.]

So what do you do about consistency when there's no clear winner? Throw one out? Keep both? Cassandra throws one out, Riak and Voldemort raises conflicts ("siblings" in Riak). So what do you do in this state? Semantic Resolution – using domain specific business rules to resolve – is the most obvious approach, but in practice it can be really hard.

"Ad hoc approaches have proven brittle and error prone."

3.5.1 Conflict Free Replicated Data Types

Instead of opaque data types/blobs in your data store you have useful abstractions. And because we're in a replicated environment, you have multiple independent copies. They're conflict free because they resolve automatically toward a single value. Described in the paper "Logic and Lattices for Distributed Programming". These structures are rooted in the theory of monotonic logic.

Bounded Join Semi-Lattices

```
<S, f, t>
```

S is a set – possibly unbounded – of all possible values. t is a member of S [less than all other values?]. And f is a function describing the least-upper bound (join/merge) on S. This provides a partial ordering for the values of S.

[Slides show *lset* and *lmax* lattices]

Lattices give us determinism in how we merge our conflicts – there is only one way to merge.

Another paper, "A comprehensive study of Convergent and Commutative Replicated Data Types", also provides some excellent information.

Two flavors of CRDTs:

- Convergent

  The data you're transmitting is the *state*; weak messaging requirements

- Commutative

  The data you're transmitting describe operations. This requires reliable broadcast, and causal ordering is sufficient.

Registers

A place to put yourself.

Concurrent updates to this type do not commute, so who wins? The two strategies are the basic strategies used by Cassandra/Riak. Last Write Wins (LWW-Register) used by Cassandra, Multi-Valued (MV-Register) used by Riak.

Counters

Replicated integers with two operations: increment and decrement. An operation based counter does not depend on delivery order (since addition is commutative).

G-Counter is a Grow Only Counter, with a minimum value of 0. You keep track of how each member of the cluster has counted.

PN-Coutners are similar, but you can go positive or negative. Again, you keep track of state for each member, and use a function to derive the actual value and resolve conflicts.

Sets

G-Set describes a set that can only be added to.

2P-Set (two phase set) describes a set where once something is removed from the set, it can not be re-added. Two G-Sets composed into a single type. One set describes the additions, the other the removals. The "value" is the difference of the two sets.

U-Set – every value has a tag that indicates uniqueness

OR-Set (Observed Remove set)

Graphs

"Unfortunately they're really complicated and error prone." :)

Working with Graphs in a distributed environment you can run into problems when two simultaneous additions create a cycle, potentially violating a global invariant.

3.5.2 Use Cases

- Social graph – OR Set
- Web page visits – G Counter

- Shopping Cart – Modified OR Set
- "Like" button – U-Set (handles lots of concurrent writes)

3.5.3 Challenges

CRDTs are often inefficient, presenting a challenge for garbage collection (which may require synchronization).

It's also not clear who's responsible for the synchronization. Some client libraries implement this – mochi/statebox (Erlang), reiddraper/knockbox (Clojure), etc – but the clients aren't participating in replication, so there's some possible inefficiencies and additional garbage created.

Riak will be implementing support for these on the server side.

3.6 Expressing Abstraction - Abstracting Expression

Authors Ola Bini

Time 3:30 pm - 4:20 pm

Session https://thestrangeloop.com/sessions/expressing-abstraction-abstracting-expression

Slides https://github.com/strangeloop/strangeloop2012/blob/master/slides/sessions/Bini-ExpressingAbstractionAbstractingExpression.pdf?raw=true

Part of the group of people who took JRuby from a "toy" to "real application" level. Since then he's done things from writing a YAML parser to regex engines to re-implementing OpenSSL on Java ("that was sort of complicated"). Since then been thinking about programming languages.

When he started on JRuby he was working with Java during the day, with a background in Lisp, and wanted something different. After JRuby he began working on AIoki (sp?), a language experiment designed to explore expressiveness.

Three questions come to mind when thinking about expressiveness:

1. Why are new languages still being created?
2. Is it worth choosing languages strategically?
3. Does language actually matter?

Expressiveness is defined as effectively conveying thought or feeling. Focusing on efficacy is a good place to start when evaluating expressiveness. An expressive language is a language that makes it easy to put my thoughts down into code without a lot of steps in between.

An alternate definition is "a language construct is expressive if it enables you to write an API that can't be written without the construct." where "write" implies "use", and where there's some large restructuring needed if the construct doesn't exist.

But beware the Turing Tar Pit: where everything is possible but nothing is easy [shows quote re: including buggy subset of Lisp].

A lot of thinking on languages cites Paul Graham's Blub Paradox, which states you have a scale with languages placed on it from least to most powerful. If a programmer is using a language Blub roughly in the middle of the scale, she can't accurately evaluate the expressiveness and power of a language higher up on the scale. She doesn't have the context or knowledge to do so.

Aspects of Expressiveness

(these are really dimensions – scales)

Regularity, readability, learnability (not sure if that's actually interesting for the question of expressiveness, and maybe it's a derivative of regularity and readability).

But the core is Essence vs. Ceremony. Everything I have to say not related to my problem is Ceremony, and it's in my way.

Precision vs. Conciseness: if you only want to say the things you need to say, you also need to be OK with leaving out some parts that influence other parts of the program (precision).

For his experiments he chose expressiveness over performance, and unsurprisingly, the language ran quite slowly. And he thinks he made a mistake: performance is a part of expressiveness. If your language is concise and lets you write the essence of something, but runs too slowly to be of practical use, it's of limited value.

The theoretical side of expressiveness: "More expressive means that the translation of a program with occurrences of one of the constructs C to the smaller language require a global reorganization of the entire program."

Some people say that if you have "patterns" in your language, then your language is deficient in some aspect of expressiveness. That's in contrast to the current thinking in the Java community, which states that patterns are to be used to enhance understanding.

3.6.1 Practical Expressiveness

Abstraction is slightly more well defined in programming languages, and in most cases abstractions add to the expressive power of a programming language. There are several types of abstractions we use day to day. Objects are one of the most common. And abstracting classes of objects (esp in prototype based languages) is pretty common, too (the joke about every Scheme programmer writing their own Class system). Macros are an abstraction over the structure of code.

One thing you don't see a lot of is abstracting the relationships between things. There are some examples – Actors in Erlang, dataflow variables in Mozart [?], and Java FX – but it's the exception rather than the rule. Spreadsheets are actually an example of this – cells provide an abstraction over the relationships between values.

If your language doesn't have a Macro facility, then all of the abstractions that add expressiveness by hiding ceremony aren't available to you.

But not all macros are created equally. C-style macros are pretty limited, and are little more than text replacement. Lisp macros, "AST Macros", aren't actually AST macros. They operate on an S expression, which is an abstraction of the AST. C++'s template system is a Turing Complete template/macro system [yikes!].

Static typing actually is a way of expressiveness in a language, but it's double-edged.

Generics – and Type Classes in Haskell and Scala – are a powerful feature of a language that are an abstraction in and of themselves, but they also enable additional abstractions. This makes them pretty interesting to study.

Abstractions in general are leaky. You see this clearly in object-relational mappers. There are two classes of ORMs: those that try to completely hide the fact that you're operating against SQL, and those that are closer to the metal. An example of the latter is ActiveRecord in Rails. [My instinct is that Django's is like this, too.] The difference between these two approaches is that in systems like Hibernate (an example of the former), you *know* how to solve a problem using SQL, but you can't get low enough to fix it.

Spolsky's Law: "All non-trivial abstractions, to some degree, are leaky."

So Spolsky is probably right, but *why* are they leaky? Abstractions are relative to what you're trying to do: they have context. They're not absolutes. You can imagine different libraries approaching an abstraction differently, depending on how they expect to be used. Abstractions hide things, but only in one direction. You can think of the leakiness as coming from the sides, issues that are orthogonal to the one the abstraction was created against.

3.6.2 Linguistics

Simile is sort of a type class, it's a way to add a new meaning of something, or add abstractions.

Redundancy is something we see in natural language that we don't see in programming languages. If you count how many times "I have a dream" appears in that speech, it's a lot! And we do it with purpose in linguistic language, unlike in programming languages where they usually wind up being ceremony (see pre-Java 7 declaration/instantiation of parameterized types).

You also have a lot of different ways of saying the same thing in natural languages. Sometimes that's true in programming languages, sometimes it's not. Ruby and Perl let you say certain things in many ways, while Python tends towards one "right" way. In natural language you use different ways to say the same thing to provide additional context, or expressiveness.

In linguistics we talk about syntax, semantics, and pragmatics. Syntax is pretty understandable, and we know that semantics is how identifiers relate to one another. Linguistic pragmatics is less well known, and is how the context of something contributes to meaning, how the context influences our choice of how to say something.

At the end of the day, natural and programming languages are about communication. [We write for the next engineer.] We need to communicate with team members as well as the computers that run our code. We communicate indirectly to people paged in the middle of the night due to a bug [:)].

One of the ways we can change the way we communicate is through syntax. Syntax is actually more important for communicating than it is for computers: you'll find an entire PLT community that says syntax doesn't matter. Just as there's syntactic sugar, there's syntactic salt (that which makes your code look bad), and syntactic sacharrrine (which feels like overkill – too much sugar).

3.6.3 Design Principles

- One paradigm
- Minimal core/concepts
- Simplicity
- First Class functions
- Flexibility
- Skinnable type system

So how far away is the truly expressive language? It's not clear, and it's not clear that expressiveness is *always* better. Maybe it's already here, just not evenly distributed. Expressiveness and abstractions are relative, both to the people using it and the subjects they're being applied to. So maybe what you want is a meta-expressive language. This is one of the reasons DSLs have become so popular.

3.7 Taking Off the Blindfold

Authors Bret Victor

Time 4:30 pm - 5:20 pm

Session https://thestrangeloop.com/sessions/taking-off-the-blindfold

Slides http://worrydream.com/LearnableProgramming/

Visible Programming: Designing a programming environment around how human beings do human being things

Talk about programming environments: the software we use to create other software.

The purpose of a programming environment is not to increase productivity (implies “type faster”), but to see and understand what the program is doing.

Five principles for the design of a programming environment

1. Read the vocabulary – what the program says
2. Follow the flow – what happens when
3. See the state – what’s going on during execution
4. Create by reacting – sculptors start with a lump of clay and incrementally turn it into an element. The sculptor does not do a single pass and produce an elephant.
5. Create by abstracting – we have a tower of abstractions, and we start at the bottom with a simple abstraction, and the move up. The environment should provide the tools to do this.

3.7.1 Read the Vocabulary

If you look at a bit of javascript code, there’s a lot of questions to answer before you can work on the code. What do the functions do? What do these arguments mean? What’s the range and units? Right now the answer is usually “look it up” or “look at the manual”. What if you could hover over the parameters and see what they mean.

Make the meaning transparent, and explain in context.

[This talk was very visual, difficult to describe in notes.]