

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2008-02/msg00994.html>

- *From:* Michael <kain0_0@xxxxxxxxxxx>
 - *Date:* Mon, 18 Feb 2008 22:20:49 +1000
-

On 2008-02-17 06:41:04 +1000, Jon Harrop <usenet@xxxxxxxxxxxxxxxx> said:

<snip>

Expressiveness lets you write code faster, that i find no fault with.
The fault i find, is allowing programmers to write poor code because they can. We should be encouraging good code.

Look at what the expressiveness did in the OCaml vs C++ examples I quoted in the last mail. The OCaml wasn't just written more quickly, it is also much easier to understand and maintain. That is exactly what I meant when I said that a language which lacks expressiveness cannot be used to write "good code" from an objective point of view: we cannot improve that C++ code because of the limitations of the language.

I did see the expressiveness difference. I'll shall be investigating those languages soon.

<snip>

To the last – That i believe was the reason for source-code. It was meant to be (but so rarely is) platform independant. From here the trade up goes from strict interpretation (maximally portable), to maximum speed (portable to exact machine copies only).

I was discussing this on the caml-list recently. As an industrialist, I don't want to ship my products in source form if I can help it because I will lose my competitive edge if someone decides to steal my work, develop it themselves and resell it.

Good point, but any well equipped (and determined) individual can reverse engineer from compiled code, look at the samba protocol. Admittedly source form would make it easier ;).

<snip>

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

I see what you mean. You would have to be careful with the generation of such expressions though. If what i understand is correct in thier limited form they could report on information that shouldn't be reported (imagine a game player looking at the opponents cash), in a more dangerous form they'd rewrite internal logic permanently.

Actually .NET already handles the encapsulation and security for you.

That's what i'm worried about. (Putting microsoft vendetta aside) Cool.

<snip>

This has advantages and disadvantages though. Lisp is unusually good at garbage collecting code. The JVM and .NET basically leave you with manual memory management in comparison. In particular, the .NET regular expression match doesn't bother deallocating the native-code for each regexp so it leaks memory. You wouldn't notice this in most applications because they only use a finite number of regexps but a long-running active website using regexp for user-defined searches will leak memory indefinitely and die. Ironically, that is exactly the kind of thing .NET was designed for...

I'd laugh but i've already had my microsoft joke for today... oh another laugh won't hurt?

<snip>

I believe the first was a Compiler/IDE feature, the way you describe it. Unless of course your meaning is that the expression tree can be accessed programmatically internal to a program in which that line was written? If so that is handy, otherwise i'm lost, why do you need to know the type of an expression you already know as a programmer?

Mostly because type errors mean that the programmer and compiler disagree about a type. That's when the programmer needs to be able to get the inferred type from the compiler.

cool.

<snip>

If all things were equal, yes. But .NET has a huge user base and a wide variety of programs than interoperate using it. That is the only reason it provides such a productivity boost in practice.

Not only that, but .NET also provides a uniform C FFI for all of its languages. So my F# code can use XNA, content in the knowledge that far more people have contributed to it, used it and tested it than the

Re: "STL from the Ground Up"

equivalent OCaml-only LablGL bindings to OpenGL under Linux.

Hmm, yes. Mind you, i'm more for cross platform technologies than for any specific vendor tech. (Yes .net doesn't have to be single vendor) Though for your particular needs i can see why it provides the boost you need.

<snip>

Right, that's why you only use this as a temporary store, e.g. for transmitting over the network, and not for persistent storage of data.

I'd hesitate at over a network too. However i see how it is usefull amongst a generation of local programs. I'd still add a note to extend the network protocol to handle a secondary martialling technique.

<snip>

I'd also like to inquire as to the compartive use of libraries?

You mean for that specific application or in general?

For the specific application. For fairness the precompiled content should be recognised too. (However those compile times look good).

<snip>

Interesting.

<snip>

You have a point there, the technique is usefull like closures. On the other hand that is a historical decision in the c type system to go for declaration based types, as opposed to implicit typing.

I think this is more a case of opinion/preference. I certainly want structures that are shared between modules to be declared. That way everyone can see what the structure is. On the other hand internal to a module structural typeing may have an application.

You've forgotten about type inference. :-)

The type may not be declared by the programmer but it is still inferred, checked and documented by the compiler.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

The real benefit is in closing the gap with dynamically typed languages, which (otherwise) make it easier to solve certain kinds of problems like interop between statically-typed domains.

I understand that there is type inference. I'm simply used to a lot of internal structures/classes being altered significantly during development. I like there to be a document/declaration/interface that can be used as explicit/implicit indicator of actions/behaviour.

<snip>

Yes it's ugly. Most ugly and/or complex code gets shuffled into libraries for a reason.

Someone still has to write it. :-)

too true.

<snip>

Tail recursion i believe is a compiler optimisation (though yes it can also be an explicit feature of a language).

I would rank it more highly than an optimization because it is essential to avoid stack overflows, i.e. it affects correctness.

Good point.

I have to agree that c++ doesn't provide current continuation or continuation passing. Though if one wanted them, an enterprising individual could emulate them. (Probably comparable to c programmers using function pointer tables to emulate objects)

Not portably AFAIK because you need to be able to replace the function call stack.

hmm, i did say emulate. Create your own "stack" implementation. Write all functions to allocate variables to this "stack". Break all functions down to those to move between continuation shifts.

The emulation really isn't that different to a task list and worker pool.

Re: "STL from the Ground Up"

I did say emulation right.

A non portable one may be more natural though ;)

. Closures.

Probably a given, you've found a hole i my knowledge.

Closures are function values that can capture variables from their environment. Closures require garbage collection to be implemented in a usable form because captured variables must be kept alive for at least the lifetime of the closure, which may outlive the scope where it is defined because functions can return closures. Indeed, that's exactly what the "d" function above did.

For a closure a GC isn't strictly necessary. Resource management like smart pointers/references/handles/copies can be used where appropriate. Admittedly this requires more ground work than a gc version.

They are also many times slower and, arguably, constitute GCs themselves. If you're using smart pointers then you're really just using garbage collection technology that is almost half a century out of date!

There are also serious problems with such techniques, like reference counting fails to collect cycles and leaks memory and even "conservative" GCs for C/C++ like Boehm do not guarantee that memory will be reclaimed and even break existing code bases.

Argueable yes, though it isn't as pervasive or random as the global kind. As for out of date, i don't believe so. Thier deterministic qualities are very usefull at times.

In simple datastructures weak pointers won't compromise intent and work to eliminate cycles. In networks, bless a particular node as a destruction parent and progress from there mark and sweep style. (Yes it is a gc, but remarkably local, deterministic and interoperable with other memory management techniques).

I personally prefer to choose the management style based upon the problem.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

Then you're competing against man-centuries of work by people who've dedicated their lives to solving the same problem. Best of luck with that, but I'll just be using the fruits of their labor. ;-)

I have no problem with using other peoples work, I just find that flogging the same horse for every problem doesn't always work best (or at all).

In practice, if you don't already have garbage collection then you limit yourself by not using things like hash consing and bloating all of your graphs into trees by copying data to avoid recombinant data structures.

Reference Counting.

That actually happened to be an example where reference counting completely fails: your graphs would never be collected. The only solution is to write a program that traverses your graph and detects unused regions (including cycles) and deallocates them. That is precisely what a GC is.

It only fails if applied naively. Graph nodes themselves are purpose made for working with graphs, and so can thier reference counters.

Also who says we must not use tailored memory management? We know beforehand often enough which trees will share nodes, we can tailor our approach to increase efficiency.

I disagree. The context I was thinking of there was my work on the Mathematica computer algebra system. The current implementation still uses reference counting and it leaks memory and dies as a consequence. They wanted to solve that and exploiting an existing GC is by far the easiest solution.

I do agree with the easiest. It's just not the only way. I like to keep an open toolbox of solutions – even if they are rarely used.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

In that case, you have absolutely no idea where cycles will appear in your data structures because it is entirely up to the user. So you must be able to account for all possibilities.

Yes

This is why parsers written in C/C++ are typically many times slower than parsers written in languages like OCaml. This is part of what OCaml was bred for, so that isn't surprising.

I agree, custom make usually beats generic make hands down.

These languages are still surprising me. Stephan Tolksdorf recently released a library for parsing in F# called FParsec that forced me to change my benchmark conclusions and I had to eat my own words as a consequence.

People are doing the most incredible things with this new breed of languages now. Even though F# is somewhat mundane as a language (being from the 35 year old ML family), its interoperability, code generation and other features make it a truly awesome tool for people like me. I can't wait to see what's next...

Neither can I.