

Re: "STL from the Ground Up"

Source: <http://coding.derkeiler.com/Archive/General/comp.programming/2008-02/msg00871.html>

- *From:* Jon Harrop <usenet@xxxxxxxxxxxxxxxx>
 - *Date:* Sat, 16 Feb 2008 20:41:04 +0000
-

Michael wrote:

On 2008-02-16 11:28:26 +1000, Jon Harrop <usenet@xxxxxxxxxxxxxxxx> said:

Michael wrote:

On 2008-02-15 13:38:54 +1000, Jon Harrop <usenet@xxxxxxxxxxxxxxxx> said:

For one, yes. There are many other benefits as well like safety, a high-level intermediate language than can interoperate with many other languages, run-time compilation and so on.

Yes, it provides safety in the face of incorrect programming and subtle flaws. In somecases this is usefull. However it is usually more agreeable to have well written code in the first place.

If your language lacks expressive features then you cannot write code well in it. I give some examples of this below. Maybe you can tweak my C++ but it will never rival the OCaml/F# code that it more powerful in every case.

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

Re: "STL from the Ground Up"

because of the limitations of the language.

It is perfectly acceptable for
java, C#, and your lovely languages to be directly compiled
into
processor native instructions.

How could you provide run-time code generation, reflection, platform
independence?

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.

Runtime code generation can be effected by embedding a
compiler/interpreter.

You also need dynamic linking (and unlinking!) to be able to call your
run-time generated code from the main application.

To support reflection you make a compiler embed type and functional
information in the compiled program. If you want it complete for
libraries too, you'll need direct support for libraries, or allow
programmatic hooks into the reflection system.

You are really describing how reflection and code generation are
implemented. I agree, of course, but C++ still doesn't provide any of that.
If you look at something like Lisp, for example, its bundled capabilities
are awesome by comparison.

Run time compilation? Is a feature of an IDE – XCode can
be set to
perform run time behind the scenes compilation of c, c++,
objective-c
and java. (At last reference many months ago, the feature
isn't likely

Re: "STL from the Ground Up"

to have been dropped).

That is not what I was referring to. I mean the ability to translate expressions into native code and evaluate them on-the-fly all within your own program. Nothing to do with the IDE.

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 their 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.

. Pattern matching.

Regex, yes? Usually seen as a library in anything below a scripting language.

I was referring to pattern matching over algebraic data types but regular expressions are another good example: they run much faster if they're run-time compiled to native code.

Hmm, i find that they'd be faster if they were already compiled. Though in the context of the previous i believe you mean from an interpreted byte code (platform independent) which i'd have to give you.

Exactly, yes. Languages without run-time code generation (like OCaml) tend to match regular expressions by compiling to a bytecode that is interpreted. Languages with run-time code generation (like F# and Lisp) can generate and build a custom native-code program specifically tailored to each and every regular expression used in the program.

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...

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

The compiler infers the types of all subexpressions for you and makes them available to you via graphical throwback in the GUI. The type inferred by the F# compiler is:

```
val f : 'a * ('b * 'c) -> ('a * 'b) * 'c
```

...

This is the main reason why OCaml and F# typically require 1/4 as much code to do the same thing. Note that this is not dynamic typing and, in particular, there is absolutely no run-time performance cost in F#.

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.

. Interoperability.

I can't think of anything more common than c. Most OS's are written in it, and export to it.

Provided you're only talking to the OS that's fine but few applications are written in C. F# code can interoperate with C, C++, C#, VB and Excel with a high-level interface that is both type safe and garbage collected.

I see what you mean, but that is the point isn't it? The code interoperates at the lowest denominator (your correct in that c doesn't have to be the denominator). What i believe your referring to is the range of decent application bindings and interfaces. That is roughly the same as my using an objective binding to opengl, or praising boost for providing code that seamlessly works with the full features of c++.

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 equivalent OCaml-only LablGL bindings to OpenGL under Linux.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

. Run-time compilation.

IDE. Look at XCode, i'm sure there are others.

Nothing to do with the IDE. Look at MetaOCaml, EVAL in Lisp, LLVM or the MSIL code emitter in .NET for examples of run-time compilation.

Definetly usefull.

There are some interesting differences. I spent a while playing with MetaOCaml, which provides a completely high-level and type-safe form of metaprogramming. However, I found that I only wanted to use metaprogramming when performance was critical and the safety of MetaOCaml often gets in the way and cripples the performance of the generated code. So I would actually rather target something like MSIL directly. It isn't even that hard: you can write an optimizing native code compiler in F# using .NET's MSIL emitter in a few hundred lines of code.

. Marshalling.

This sounds like RPC and related. See Pattern Matching.

These languages use uniform internal representations of data structures. This allows these data structures to be marshalled to file, disk or over a network (also called pickling and serialization).

I see that this is usefull too. But what happens (and it will) when the internal representations change in the future? (re Java) A more durable solution would keep the language seperate from data representation.

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.

. Compile times.

I would rather a slightly longer compile vs run time.

Our compile times dropped from 24hrs with g++ to only 30secs with ocamlpt. As I said, our run times became 5x faster as well.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

Granted quite a difference. I believe you are referring a c++/g++ to Ocaml/ocamlpt?

Yes. Non-trivial use of templates completely cripples g++.

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

You mean for that specific application or in general?

This is why OCaml programs that make heavy use of complicated pattern matches can be orders of magnitude faster than C++ programs. C++ compilers have no hope of performing such optimizations because the information is not available to them. In particular, the C++ language does not even support closed sum types.

Hmm, i believe you are talking technique here and not language.

Yes. This technique is provided by many languages though, including SML, OCaml, F# and Haskell.

To this
a compiler can flatten, nest, break down, discard, genericise, inline, outline as much as it wants of a program – providing the end result is a program with the expected behaviour. Take a look a sql (and related database query languages).

No, the C++ language doesn't convey the necessary information to the compiler so the compiler's optimizer does not know the desired semantics in enough detail and the rearrangements it can perform are very limited as a consequence.

Even if you believe in the mythical "smart enough compiler", C++ is still a lot slower today.

On the otherhand looking at it in terms of language, i can only see the compiler being more effective due to it's native understanding of the description. Imagine a completely new feature Y, implement it in terms of a language as a description (not as a part of the language) and i wouldn't doubt that ocaml would be in a similar boat to c++.

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

You can retrofit new features onto OCaml using its macro system (camlp4). It is a bit of a botch job and you end up making lots of incompatible miniforks of the language if you're not careful but people have done some quite interesting things with it. In particular, there are several macros that extend OCaml's built-in pattern matching with extra constructs. When you're doing that, OCaml should still keep a lead over C++ because C++ doesn't have pattern matching.

. Parallelism.

Yes, similar argument as with concurrency. It is best to let the compiler decide, or the system decide.

The compiler's hands are tied unless the language exposes room for parallelism.

Not really. Compilers can take advantage of guarantees in the language, eg encapsulation, dependancies, partial ordering.

Even if you take the extreme case of a purely functional programming language that completely forbids mutation so everything is trivially parallelizable, I still don't believe that.

My reason is simple: forking a computation to make it run in parallel is very costly. You either pay for a process fork (~1cs), a thread spawn (~1ms) or a mutex (~1us). Even if you amortize over a static per-CPU threadpool for the immutable case, there is still a major slowdown from aggressive allocation (if your allocator is concurrent then that is ~5x slower) and you've got the overhead of a function call.

If the compiler just blindly parallelizes everything possible then it will inject those performance hits all over your code, making everything run a lot slower. So what you've gained from more CPUs you've lost from inefficiency.

I just can't see how a compiler can possibly make this work and I've tried several systems that claimed to do this and none of them worked.

Admittedly unless you take pains to write the algorithms to be friendly to such manipulation, your likely to end up with difficulties on naive compilers. The easiest way to provide a compiler with such information is to build it into the language. However that still doesn't address the core issue, that of the compiler understanding the program (not just the language it is written in).

Re: "STL from the Ground Up"

SGI told me that with regard to their automatically-parallelizing compiler that we used years ago. That was the demo. In practice, I couldn't even get it to parallelize the most trivial of examples let alone any real code. The total performance improvement it gave us was zero.

So I don't buy the idea that compilers can do a good job of automatically parallelizing legacy code. I believe that languages can make concurrency transparent for the compiler, as Erlang does most successfully, but that is very different. This is why I don't believe a lot of what the Haskell community have been saying about parallelism recently. They still haven't produced a single working example.

. Structural typing.

...

The nearest C++ equivalent is not only hideous but it can't even handle common subexpressions because it lacks a garbage collector to determine when an expression becomes unreferenced:

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.

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.

```
const Expr operator+(const Expr e1, const Expr e2)
{ return Expr(new Plus(e1.e->clone(), e2.e->clone())); }
const Expr operator*(const Expr e1, const Expr e2)
{ return Expr(new Times(e1.e->clone(), e2.e->clone())); }
```

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

Re: "STL from the Ground Up"

Someone still has to write it. :-)

What till you require newer features in ocaml that have yet to be introduced/standardised.

Then I migrate to a newer generation of languages, which is exactly what I'm doing now with F#.

. Recursion.

Most languages support recursion, C and C++ do. Are you instead referring to thier optimisation?

Tail recursion, call with current continuation and continuation passing style. C++ provides none of these.

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.

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.

. 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

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"

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.

Libraries can't work around all missing language features. Say you use flex and bison to write a parser given a BNF grammar. You might want to save memory by hash consing subexpressions (automatically sharing all identical subexpressions) but that turns your abstract syntax tree into a graph. To deallocate it you must now start doing graph traversals, which means you're literally writing a garbage collection.

Yes i agree. Libraries aren't silver bullets. Mind you, how memory is managed depends upon what you need. GCs aren't bad, they just aren't good. They are wonderful when the only resource to be cleaned is memory. To often though deletion requires side effects to occur, messages, i/o port closeing, alteration of memory in a live object. And not as often but still, we'd like the effects to occur soon and not never.

Yes. Don't rely upon the GC to collect resources other than memory. That can extend beyond malloc though. I used OCaml's GC to collect memory-based resources from the GPU and it worked beautifully. Not recommended, perhaps, but it can work extremely well.

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

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. ;-)

Re: "STL from the Ground Up"

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.

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.

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.

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...

--

Dr Jon D Harrop, Flying Frog Consultancy Ltd.
<http://www.ffconsultancy.com/products/?u>

Re: "STL from the Ground Up"

Re: "STL from the Ground Up"