Why ML/OCaml are good for writing compilers

Author: Dwight VandenBerghe
Email: dwight@pentasoft.com
Date: 1998/07/28
Forums: comp.compilers

Let's use the term "ML" to mean SML or Objective Caml. I'm a devotee of Ocaml, but I have SML/NJ installed and although I prefer the distribution, tools and overall implementation of Ocaml, I'd be happy to write in SML/NJ if Ocaml wasn't around. (I can't speak for Haskell, gofer, hugs, or any of the lazy languages. I think that there are different kinds of people, some of whom like things like deferred evaluation and other who like strict call-by-value. I'm decidedly in the latter camp; I like how clean strict evaluation is, how easy it is to understand exactly what is going on, and I find that I have to give a little of that up with Haskell. But your mileage may vary.) So although when I am writing about "ML" I am writing about Ocaml, what I say also applies, I think, to SML/NJ.

So here is an unordered list of language features that seems to me to make writing compilers a pleasure rather than a horrendous chore.

1. Garbage collection. It may seem elementary to mention this, but gc is an incredible boon to programs that have lots of complex data structures with short-to-intermediate lifetimes. And compilers are, above all, programs with complex data structures. You gotta be a data structure junkie to like compilers. Well, C/C++, Pascal, and their ilk, with unrestricted pointers and malloc and their cousins, make you take out your own trash. You have to be the programmer and the janitor, at the same time. Some of us are better programmers than we are janitors. ML has, perhaps, the best gc around; it's so fast that for many real apps it's as fast as the C++ malloc/free, and maybe even a little faster in some cases. You don't have to wring your hands about using ML's gc, as you do with Java's, which is slow. It's just there, invisible, blindingly fast, and it works all the time, and your life is thus made much easier.

2. Tail recursion is optimized. Thus, once you know how to take advantage of it, you can write tree walks that don't eat up stack space, and are very fast. Now, to be fair, there are C++ compilers (like VisualC++ 5) that supposedly have some tail recursion eliminated, but you can't count on it (VC has a lot of bugs in theirs). ML is matched to recursion very well, and since many of the data structures in compilers tend to be best handled with recursive procedures, there is a good fit there.

3. The data types in ML match the compiling process. Compilers don't tend to worry about unsigned shorts vs signed chars, but use "ints" everywhere, as well as strings. Strings are used all over the place, and C/C++ are pretty terrible with strings, even in the presence of templates. There are some places in compilers where it's nice to have arithmetic on quantities a little larger than ints, so the "bignum" facility is actually useful in those cases (like when you have to fold constants or tokenize with a little more precision than the underlying numeric types, then convert back to the native form). If you don't have bignums, then you have to write your own, or resort to subterfuge. (See guru compiler writer Dave Hanson's "C Interfaces and Implementations" for an example of what I mean.)

4. The type constructors in ML are just plain wonderful for describing things like ASTs (abstract syntax trees). They implement what are sometimes called "tagged unions" - that is,

a union data type (efficient, small) that, unlike with C/C++, come along with a tag field that tells what's in the union. This is enforced: in other words, there's no way to circumvent it. ML pattern matching is designed to work with tagged unions so that the source code is incredibly readable, for functions that take a data structure as an argument. Combine this with type inference and tail recursion elimination, and you have a language that is optimized for recursive functions that take complex data structures are arguments ... sound familiar?

5. Safety. ML was conceived as a solution to the main problem faced by mathematicians using automated theorem provers: that because of the type-free, dangerous, cavalier nature of the usual language for that domain (Lisp), you could never be sure that your program was going to work. Of course, this could be said about all languages, but ML was an attempt to restrict the domain a bit, to add efficiency and safety at the same time. ML programs can't crash the system; if it compiles, it will run, and you won't get a segmentation fault. You can prove certain properties about your program, you can trust that certain kinds of errors just plain can't happen. For example, since lists are immutable, and must contain elements of a single type, you don't have to worry about screwing up a string list by putting an integer in it by accident (as you can with Scheme and Lisp). In fact, you can't put anything into it at all; you have to create a new list. This makes the underlying routines able to be much faster than lists in other languages, where you have to be concerned about double-links, destructive updating, and so on. With a blindingly fast GC system, this all works out just fine ... and you sleep better at night.

6. ML was designed for an application domain (theorem proving) that is characterized by big, hairy, recursive data structures that have complex algorithms running against them. Sound familiar?

7. Exceptions. ML implements fast and clean exception handling, which is a real joy if you've never had the pleasure of using them before. You write a table lookup by assuming that the key will be found, and then wrapping the search in a "try" block that catches the exceptional case ("not found"). So you can't ever screw up your program by forgetting to test the not-found case; if you do that, the runtime system will stop with an uncaught exception, telling just where the exception was thrown. Programs become easier to read, cleaner, more robust, when you learn to use exceptions.

8. Type inference. In a thousand lines of ML you may have to declare only two or maybe three variables, if that. It just figures out the types by how the variables are used. And it doesn't guess, like (say) Perl does. It knows for sure. One of the reasons I like Ocaml over SML is that Ocaml doesn't like operator overloading: there are different operators for float addition ("+.") and integer addition ("+"), for example. Type inference and operator overloading are uncomfortable bed partners; in my opinion, a language designer should choose between them, rather than try to serve both masters. But anything's better than Pascal or C, where the compiler just tosses out everything it knows about usage and makes you state the obvious, over and over and over again. All we can do is screw it up, and so we do, over and over.

9. Lex/yacc/burg. ML has great implementations of these standard tools, and once you know how to use them, they make a lot of jobs simpler. No, I'm not a great lex fan, nor do I prefer lalr(1) over ll(k), but I'm a pragmatist: if it's there, and it's well-implemented, I'll use it. Ocaml and SML/NJ both have really good, solid compiler tools implementations that are used by their developers. Not many languages come with better toolkits.

10. Did I mention that Ocaml is fast? I wrote a compiler for an actuarial financial modelling language in Ocaml, probably 10K lines of code, that would have been 20K or more in C++, and it compiles the largest known program in 3 seconds on my pentium 200. Most programs compile in under 1 second. I find this astounding.

11. Support. I've gotten better support from Inria (Xavier Leroy and Pierre Weis, among others) than I ever have from any other language vendor. There just aren't many problems with the language, and those few problems I've encountered have been fixed in a matter of days. Compare this to the support for, say, VC++ or Turbo Pascal.

12. Library. ML's standard library has a lot of data structure stuff in it, which helps a lot. I find it more complete and succinct and usable than the usual glommed-together mess of most other languages.

13. The module system. ML has strong, well-thought-out support for separate compilation that allows a separately compiled module to support polymorphism (that is, it can operate on arbitrary types). The visibility of the internals of the module can be exactly controlled. "Functors" can specialize a module into a specific incarnation. It's pretty cool, like C++ templates without the pain and suffering.

So it's mostly about data structures. ML is extraordinarily facile at allowing you to express complex data structures and recursive algorithms around them. The most basic data structures (lists, arrays, structs, unions, property lists, hash tables, binary trees, queues, and so on) are sitting there in the language already, well-implemented and ready to go. You start off from a place that you would have had to build up in another language.

Is ML the perfect language? Lord, no. It has many flaws, like every other language. The syntax is weird, some parts of it are hard to learn, some parts are hard to use. A simple thing like Printf can be quite strange to express. You can't use ML to advantage for many problem domains (like the one I work a lot in, embedded systems). It would be terrible to use for, say, programming an FFT inside a DSP chip. ML doesn't sit all that well with GUIs yet, as far as I've seen, and support for OOP is lacking (although I see that as a feature, not a restriction).

But all languages have some problem domains in which they shine. I think that compiler implementation is one of those areas for ML. You're writing a compiler and in middle of a function you need a 9mm crescent wrench with a box on the other end. You open up your toolkit and ... there it is, in the top drawer, bright and shiny and strong. You use it, and then a few minutes later you need a small phillips screwdriver with a clip and a magnet ... and there it is, bright and shiny and strong.

It's not that there are an extraordinary number of tools in the toolbox. (No; in fact, the toolbox is much smaller than the usual toolboxes, the ones used by your friends that contain everything but the sink.) It's that the toolbox was carefully and very thoughtfully assembled by some very bright toolsmiths, distilling their many decades of experience, and designed, as all good toolkits are, for a very specific purpose: building fast, safe and solid programs that are oriented around separate compilation of functions that primarily do recursive manipulation of very complex data structures.

Program, like, say ... compilers.

Dwight