JFP **18** (4): 553–564, 2008.
doi:10.1017/S095679680800676X First published online 24 April 2008 Printed in the United Kingdom

Caml trading – experiences with functional programming on Wall Street

YARON MINSKY and STEPHEN WEEKS

Jane Street Capital, New York Plaza, New York, NY 10004
(*e-mail:* {yminsky,sweeks}@janestcapital.com)

Abstract

Jane Street Capital is a successful proprietary trading company that uses OCaml as its primary development language. We have over twenty OCaml programmers and hundreds of thousands of lines of OCaml code. We use OCaml for a wide range of tasks: critical trading systems, quantitative research, systems software, and system administration. We value OCaml because it allows us to rapidly produce readable, correct, efficient code to solve complex problems, and to change that code quickly to adapt to a changing world. We believe that using OCaml gives us a significant advantage over competitors that use languages like VB, Perl, C++, C#, or Java. It also makes finding and hiring high-quality software developers *easier* than with mainstream languages. We have invested deeply in OCaml and intend to use OCaml and grow our team of functional programmers for the foreseeable future.

1 Introduction

The world of financial trading is dominated by a small number of mainstream programming languages: Perl and VBA (the latter often within Excel) for quick-and-dirty applications; C and C++ for performance-critical software; and Java and C# for everything else. There are some subfields, notably derivatives modeling, where functional programming languages have gained a foothold (LexiFi, 2007; CSFB, 2007). But overall, functional programming has little presence on Wall Street.

This report will discuss the experiences of Jane Street Capital, a financial firm that has gone counter to the standard development practices of Wall Street by adopting OCaml (Leroy *et al.*, 2007) as its primary development platform. Jane Street is a proprietary trading company, which is to say that the company's business is to use its own capital base to trade profitably in the financial markets. We have no customers and we do not actively solicit outside investors. Our expertise lies largely in market-making and arbitrage in equities and equity derivatives. Jane Street was founded in 2000 with three employees. Since then, we have grown into a global organization, with over 130 employees and offices in New York, Chicago, Tokyo and London.

Trading is an intensely technological business, and getting more so by the day. Jane Street puts a large amount of effort into developing custom systems for a variety of needs: management of historical data; quantitative research; live monitoring of positions and risk; trading systems; order management and transmission systems;

and so on. At present, the development of software to satisfy all of these needs is being done in OCaml. Below, we will discuss the reasoning and history behind the decision to switch to OCaml, as well as some of the problems that we have found with OCaml along the way.

2 Why OCaml?

As Jane Street has grown, its technology has grown and changed with it. Initially, Jane Street's technology was largely based on Excel and VBA. These were excellent tools for quickly building the systems we needed, but it was clear from the start that this was not a viable long-term strategy. The systems at the time were insufficiently modular, leading to a great deal of cut-and-paste code; they were too slow, despite heroic optimiziations; and they were too difficult to modify with confidence in the correctness of the results.

Correctness and safety are obviously essential in a trading context. Automated systems that trade rapidly magnify the cost of mistakes. From our perspective, the most prized ability of any trading system is the ability of that system *not* to trade. But there are other important considerations outside correctness. Reliability is essential, since downtime at the wrong time can mean missed opportunities. Performance matters as well; there is a lot of data to be processed and there is substantial financial gain to be had by responding quickly to that data. In addition, a language must help to manage the inherent complexity that comes from a sophisticated trading operation that spans a wide variety of financial markets, data sources, and trading strategies. Finally, a language must allow code to be developed rapidly in response to changing market conditions and emerging opportunities.

The firm's first contact with OCaml came in 2002. That year, Yaron Minsky started working at Jane street, and having some experience with OCaml as a graduate student at Cornell started using OCaml for quantitative research. Over time the research group grew and, along with it, the use of OCaml at the firm. After some experiments with using C# for systems software, the firm decided in 2005 to switch to using OCaml as the primary development language.

Jane Street's move to OCaml has been highly successful. Since 2005, we have rewritten large swaths of our internal infrastructure, fielding systems more sophisticated than previously possible, and far more efficient. Moreover, we have been able to sustain a rate of change to our core trading systems that is an order of magnitude higher than would have been conceivable previously. All of this has happened while raising the standards of safety and correctness to which we hold our systems. The net result has been a significant increase in the current, and we believe future, profitability of our core business. In the following, we will discuss some of the properties of OCaml that we believe helped us succeed to the degree that we did.

2.1 Readability

One of the easiest ways that a trading company can put itself out of business is through faulty software. We believe that code review is an essential ingredient in

building reliable software. For this reason, Jane Street has long insisted on complete and careful code reviews for our core trading systems. In particular, a number of the partners of the firm personally review almost every line of code that goes into these systems. Perhaps surprisingly, some of the partners who do the code review have minimal training in programming and computer science. For one of them, VB was the first programming language he used in any serious way, and OCaml was the second.

Given the importance placed on code review, the ease with which code in a given language can be reviewed is critical. We have found that OCaml makes it possible to write code that is considerably easier to read, review, and think about than code written in more mainstream languages, due to a number of factors:

Terseness Other things being equal, expressing program logic more concisely makes that logic easier to read. There are obvious limits to this; readability is not generally improved by reducing all function names to single characters. But OCaml allows for a pleasantly terse coding style while giving sufficient context.

One part of being terse is avoiding duplicated code. We try very hard to avoid saying the same thing over and over in our code, not just because it makes the code longer, but because it has been our experience that it is hard for human beings to read boilerplate as carefully as it deserves. There is a tendency to gloss over the repeats, which often leads to missed bugs. Moreover, as the code evolves, it is difficult to ensure that when one instance is updated, its siblings are updated as well. Higher order functions and functors are powerful tools for factoring out common logic.

Immutability OCaml is not a pure language, but the default in OCaml is for immutability. Imperative code is an important part of the language, and it is a feature that we use quite a bit. But it is much easier to think about a codebase where mutability is the exception rather than the rule.

Pattern Matching A great deal of what happens in many programs is case analysis. One of the best features of ML and similar languages is pattern-matching, which provides two things: a convenient syntax for data-directed case analysis, and a proof guaranteed by the compiler that the case analysis is exhaustive (our coding practices eschew wildcard matches). This is useful both when writing the case-analysis code in the first place, and also in alerting the programmer as to when a given case analysis needs to be updated due to the addition of some new cases.

Labeled Arguments If a function takes multiple arguments of the same type, the arguments can easily be switched at a call. In most situations, we label arguments of the same type to prevent such errors. We also find labels useful in other situations, e.g., to name a function supplied as an argument to a higher order function, in order to give the reader a clue as to what the function does.

Polymorphic Variants If a function returns an ad-hoc sum type, we use a polymorphic variant (Garrigue, 1998) and include the description of the type directly in the specification of the function. For example, we would write

```
val f : unit -> [ `A | `B ]
```

rather than

```
type result = A | B
val f : unit -> result
```

This makes it easier for a reader (they do not have to connect to the definition of `result` with its use) as well as for someone coding against the library, since they do not have to qualify the constructors in order to match against the result of `f ()`.

Because exceptions are not tracked by the type system, we avoid them as much as possible. Rather than have a function that returns a value or raises an exception, we would have a function that returns a sum of polymorphic variants. For example, we would prefer

```
val int_of_string : string -> [ `Ok of int | `Invalid ]
```

rather than

```
val int_of_string : string -> int
```

This style makes it clear to the reader that something bad could happen and makes the coder handle the problem. And polymorphic variants make it possible to do without any type declarations, module qualifiers, or other syntactic baggage. One could imagine going futher and using monadic syntax like Haskell's do notation (Peyton Jones, 2002) to simplify client code, but we have not found this necessary. We find a mix of polymorphic variants and the rare use exceptions to give a nice mix of usability, readability, and error detection.

Types The main point of code review is for the reader to put together an informal proof that the code they are reading does the right thing. Constructing such a proof is of course difficult, and we try to write our code to pack as much of the proof into the type system as possible.

There are a number of different features of the type system that are useful here. Algebraic data types are a powerful way of encoding basic invariants about types. One of our programming maxims is "make illegal states unrepresentable", by which we mean that if a given collection of values constitutes an error, then it is better to arrange for that collection of values to be impossible to represent within the constraints of the type system. This is of course not always achievable, but algebraic datatypes (in particular variants) make it possible in many important cases.

Data-hiding using signatures can be used to encode tighter invariants than are possible with algebraic data types alone. The importance of abstraction in ML and Haskell is well understood, but OCaml has a nice extra feature which is the ability to declare types as *private* (Blanqui *et al.*, n.d.) in the signature. Just like an abstract type, a private type cannot be constructed except using functions provided in the module where the type was defined. Unlike private types, however, the values can still be accessed and read directly, and in particular can be used in pattern-matches, which is very convenient.

Another useful trick is phantom types (Fluet & Pucella, 2006). Phantom types can be used to do things like implement capability-style access control for data

structures, or keep track of what kind of validation or sanity checking has been done to a given piece of data. All of this can be done in a way that the compiler proves ahead of time that proper book-keeping is done, thus preventing run-time errors.

Modularity Our initial systems based on Excel and VBA involved an enormous amount of cut-and-pasted code, both within a system, and between different variants of the same system. When changes needed to be made, they needed to be done in one place and then manually copied to others. This is obviously a difficult and error-prone process, and something that could be improved considerably by moving to almost any language with decent support for modularity. But our experience has been that the standard functional programming mechanisms (closures, parametric polymorphism, modules, and functors) for providing modularity in OCaml are significantly better suited to the task than the standard object-oriented mechanisms (objects, classes, and inheritance).

When we first tried switching over from VB to C#, one of the most disturbing features of the language for the partners who read the code was inheritance. They found it difficult to figure out which implementation of a given method was being invoked from a given call point, and therefore, difficult to reason about the code. It is worth mentioning that OCaml actually *does* support inheritance as part of its object system. That said, objects are an obscure part of the language, and inheritance even more so. At Jane Street, we almost never use objects and never use inheritance. We use standard functional programming techniques and code reviewers find that style more comprehensible. In particular, they can reason by following static properties of the code (module boundaries and functor applications) rather than dynamic properties (what class an object is).

2.2 Performance

Trading requires a lot of computational resources, both for batch-oriented research applications and for production systems that must respond quickly to the markets. An automated trading system may receive tens of thousands of updates per second and should react as quickly as possible to data. A few milliseconds in response time can make a substantial difference in profitability. We also store many tens of gigabytes of market data per day, and the amount is growing exponentially. Batch research applications must perform complex computations on hundreds of gigabytes of data in a reasonable amount of time.

There are a lot of good things to say about the performance of the OCaml compiler and runtime. The code generation is very good, despite there not being much optimization (although the cross-module inlining is important). OCaml's compiler uses a straightforward approach to data representation and code optimization. This makes the compiler simpler, which contributes to our confidence in the correctness of generated code. It also leads to an important aspect of OCaml's performance – predictability. A simple compiler and optimizer makes it easier to understand the time performance and space usage of a program. It is also easier to reason about performance tradeoffs when deciding between different ways of writing a piece of

code, or when changing code to improve the performance of a bottleneck. With OCaml it is possible to look at a piece of code and understand roughly how much space it is going to use and how fast it is going to run. This is particularly important in building systems that react to real-time data, where responsiveness and scalability really matter.

OCaml's allocator and garbage collector are extraordinarily fast. One might think that automatic memory management would make low latency applications such as ours impossible, but in fact we have had good success. OCaml uses a generational collector (Lieberman & Hewitt, 1983) and collections of the nursery take tens or at most hundreds of microseconds. For the old generation, much of the collection is done incrementally and by controlling the slices of incremental collection one can avoid latency hiccups. The only non-incremental part, compaction, is extremely fast and can be done very rarely or not at all in some applications. Finally, some applications are run such that they are usually not fully loaded. These applications can *eagerly* garbage collect when they are not busy, essentially making collection free and having little or no impact on latency.

OCaml provides a high quality foreign function interface (chapter 18 of (Leroy *et al*., 2007)) which allows for very efficient bindings to C and C++ libraries. We need to handle such libraries to deal with market data provided by vendors. The need to write our own bindings to external libraries is not just an obligation, it is also in a surprising way an advantage. There are many libraries that have only inefficient bindings to C# and Java, which means that the only way to use them efficiently is to write in C or C++. By writing our own bindings directly to the C or C++ libraries, we are generally able to get much better performance than can be found in other managed languages.

2.3 Macros

OCaml, like any language, has its limitations (which we will discuss in more detail later on). One way of mitigating the limitations of a language, as the Lisp community has long known, is to modify the language at a syntactic level. OCaml has an excellent tool for making such modifications called `camlp4` (de Rauglaudre, 2003). `camlp4` is a macro system that understands the OCaml AST and can be used to add new syntax to the system or change the meaning of existing syntax. It has also been used to design new domain-specific languages that translate down into OCaml code and can then be compiled using OCaml.

Probably the best thing that we have done with the macro system to date is our addition of a set of macros for converting OCaml values back and forth to s-expressions. If you write the following declaration while using our s-expression macros:

```
module M = struct
 type dir = Buy | Sell with sexp

 type order =
    { sym : string;
```

```
      price : float;
      qty : int;
      dir : dir; }
 with sexp
end
```

you will end up with a module with the following signature:

```
module type M = sig
 type dir = Buy | Sell
 type order =
    { sym : string;
      price : float;
      qty : int;
      dir : dir; }

 val sexp_of_dir : dir -> Sexp.t
 val dir_of_sexp : Sexp.t -> dir
 val sexp_of_order : order -> Sexp.t
 val order_of_sexp : Sexp.t -> order
end
```

The s-expression conversion functions were written by the macros, which are triggered by the `with sexp` at the end of the type declaration. It is worth noting that a compile-time error would be triggered if the `with sexp` declaration were also not appended to the `dir` type, since the functions for converting `order`s to and from s-expressions refer to the corresponding functions for the `dir` type.

The s-expression conversion functions allow you to do simple conversions back and forth between OCaml values and s-expressions, as shown below.

```
# sexp_of_order { sym: "IBM";
                  price = 38.59;
                  qty = 1200;
                  dir = Buy };;
- : Sexp.t = ((sym IBM) (price 38.59) (qty 1200) (dir Buy))

# order_of_sexp
  (Sexp.of_string "((sym IBM) (price 38.59) (qty 1200) (dir Buy))");;
- : order = { sym: "IBM"; price = 38.59; qty = 1200;
              dir = Buy; }
```

This fills an important gap in OCaml, which is the lack of generic printers and safe ways of marshaling and unmarshaling data (OCaml does have a marshaling facility, but it can lead to a segfault if a programmer guesses wrong as to the type of some marshaled-in value). Macros can serve this and many other roles, making it possible to extend the language without digging into the guts of the implementation.

For those who are interested, our s-expression library has been released under an open-source license, and is available from our website (JaneOCaml, 2007). We

also plan to release a library for fast and safe marshaling and unmarshaling using a binary protocol. There has also been some recent progress in adding generic functions to OCaml using macros (Yallop, 2007).

3 OCaml pitfalls

For all of OCaml's virtues, it is hardly a perfect tool. The language lacks features in some places (and provides too many in others), the libraries are inconsistent and incomplete, the compiler's simple optimizer misses a lot, the language and runtime do not support true parallelism, and the language and build environment are not ideal for programming in the large. Many of these deficiencies are further hindered by the "cathedral" development model adopted by INRIA, which slows the pace of improvement.

3.1 Generic operations

One of the biggest annoyances of working in OCaml is the lack of generic printers, i.e., a simple general purpose mechanism for printing human-readable representations of OCaml values. Generic human-readable printers are really just one class of generic operations, of which there are others such as generic comparators and generic binary serialization algorithms. One way of writing such generic algorithms in OCaml is through use of the macro system, as we have done with the s-expression library. That is an adequate solution, but OCaml would be a better platform if more generic operations were available by default.

3.2 Objects

In our opinion, having an object system in OCaml (Rémy & Vouillon, 1998) is a mistake. The presence of objects in OCaml is perhaps best thought of as an attractive nuisance. Things that other languages do with objects are better achieved in ML using features like parametric polymorphism, union types and functors. Unfortunately, programmers coming in from other languages where objects are the norm tend to use OCaml's objects as a matter, of course, to their detriment. In the hundreds of thousands of lines of OCaml at Jane Street, there are only a handful of uses of objects and most of those could be eliminated without difficulty.

3.3 Optimization

OCaml's code generator is good, but the lack of optimization does have a cost. At Jane Street, the lack of optimization often causes us to make a tradeoff between readability and performance. We are aware that closures, functors, type abstraction, and simple data representations have costs, and we keep those costs in mind when programming.

All OCaml programmers that care about performance learn to write in a style that pleases the compiler, rather than using more readable or more clearly correct code. To get better performance they may duplicate code, expose type information, manually

pack data structures, or avoid the use of higher-order functions, polymorphic types, or functors. In short, programmers may sometimes avoid the same features that make OCaml such a pleasant language to program with. It is possible to address this problem by using more aggressive optimization techniques (e.g., whole-program optimization as is used in the MLton Standard ML compiler (MLton, 2007)). Unfortunately, it does not seem likely that such optimization will be available any time soon for OCaml.

3.4 Parallelism

OCaml does not have a concurrent garbage collector, and as a result, OCaml does not support truly parallel threads. Threads in OCaml are useful for a variety of purposes: overlaying computation and I/O; interfacing with foreign libraries that require their own threads; writing responsive applications in the presence of long-running computations; and so on. But threads cannot be used to take advantage of physical parallelism. This is becoming an increasingly serious limitation with the proliferation of multi-core machines.

It is not obvious what the right solution is. The OCaml team has made it clear that they are not interested in the complexity and performance compromises involved in building a truly concurrent garbage collector. And there are good arguments to be made that the right way of handling physical concurrency is to use multiple communicating processes. The main problem here is the lack of convenient abstractions for building such applications in OCaml. The key question is whether good enough abstractions can be provided by libraries or whether language extensions are needed.

3.5 Programming in the large

One of the best features a language can provide is a large ecosystem of libraries and components that make it possible to build applications quickly and easily. Languages such as Perl and Java have done an excellent job of cultivating such ecosystems and one of the keys to doing so successfully is providing good language and tool support for programming in the large. Things that can help include facilities for managing namespaces (modules are *not* enough), tools for tracking dependencies and handling upgrades between packages, systems for searching for and fetching packages, and so on.

The most notable work in this direction in the OCaml world is *findlib* (Stolpmann, 2003), an excellent package system that makes it easy to use invoke installed OCaml packages; and *GODI* (Stolpmann, 2007), a system for managing, downloading, upgrading, and rebuilding OCaml packages. GODI is still rough around the edges, with an idiosyncratic user interface and sometimes temperamental behavior. One thing that would greatly help GODI or a system like it to take off would be if it was included in the standard distribution.

3.6 The cathedral

The team that developed the OCaml compiler has historically adopted a cathedral-style development model for the compiler and the core libraries. Contributions are

for the most part only accepted from members of the team, which largely consists of researchers at INRIA. While the OCaml developers do an outstanding job, it is hard not to think that the system could be improved by leveraging the talents of the larger OCaml community.

One example of how OCaml could be improved by a more open development process is the standard library. The current standard library is implemented well and provides reasonable coverage, but it is missing a lot of useful functionality and has a number of well-known pitfalls (perhaps the most commented upon is the fact that a number of the functions in the list module are not tail-recursive).

As a result, many people have implemented their own extensions to the standard library. There have even been projects, like ExtLib (Cannasse *et al.*, n.d.), that have tried to gain acceptance as "standard" extensions to the standard library. It is, however, hard to get the community to coalesce around a single such library without getting it absorbed into the standard distribution.

But the standard library is just one place where a more open development process could improve things. There is a lot of energy and talent swirling around the OCaml community and it would be great if a way could be found to tap into that for improving the language.

4 Personnel

Personnel is one area in which OCaml has been an unmitigated success for us. Most importantly, using OCaml helps us find, hire, and retain great programmers. One of the things we noticed very quickly when we started hiring people to program in OCaml was that the average quality of applicants we saw was much higher than what we saw when trying to hire, say, Java programmers. It's not that there are not really talented Java programmers out there, there are. It is just that for us, finding them was much harder. The density of bright people in the OCaml community is impressive and it shows up in hiring, when reading the OCaml mailing list, and when reading the software written by people in the community. That pool of talent is probably the single best thing about OCaml from our point of view.

Once we find great OCaml programmers, we have a good chance at hiring them, because the fact that we use OCaml is seen as an indication that Jane Street is an interesting place to work. Furthermore, OCaml contributes to a vibrant intellectual atmosphere that programmers appreciate and in which they are very productive. So once we hire great OCaml programmers, they do not want to leave.

Another perhaps surprising result is that we have had success in having other programmers (and even traders) in the firm with no previous experience with functional programming learn OCaml and become productive. OCaml has also been useful as a common language for communication of mathematical, algorithmic, and trading ideas among people from many different backgrounds.

5 Conclusion

OCaml has been tremendously successful at Jane Street, and is a major part of Jane Street's continuing success. All of Jane Street's businesses rely on software written

in OCaml and there are a number of businesses in which we would no longer be competitive without that software. OCaml has been directly responsible for an increase in the current and future profitability of our business. It has allowed us to develop code to very high standards of safety and correctness while rapidly adapting to changing markets. There are strategies we now engage in that are too complex to have even been contemplated previously.

A number of factors contributed to Jane Street's successful adoption of OCaml. First, OCaml is a particularly good and fit for trading, a business that imposes stringent requirements for correctness and offers significant benefits for agility and performance. Our early successes in using OCaml for research made it easier to see that OCaml could succeed as the firm's primary language. Jane Street's small size and the the deep involvment of top people in technology made it possible for the critical decisions to be made. And the need for complex in-house specialized software meant that that there were fewer benefits of developing it as a mainstream language.

Our experiences using OCaml in a commercial environment have strengthened our belief in the value that powerful programming languages can allow to an organization nimble enough to take advantage of them. OCaml succeeds on many fronts at once: it is a beautiful and expressive language; it has an excellent implementation; and its community is second to none.

But we believe that OCaml can become better yet. We hope that our example will encourage the growth of the language by showing students and universities that statically typed functional languages are practical, real-world tools, and that there really are jobs where students can make use of such languages. Hopefully, this will encourage the teaching, study, and development of functional languages.

OCaml is part of the family of functional languages that are seeing increased commercial use. Other functional languages such as Erlang and Haskell have many of the same advantages as OCaml when compared to mainstream languages, and would be plausible choices for proprietary trading company. It is difficult to predict how things would have gone differently had we chosen a different functional language. What we can say with confidence is that we are very happy with our choice of OCaml and would make the same choice again given what we now know.

References

Blanqui, F., Hardin, T., & Weis, P. On the implementation of construction functions for non-free concrete data types. In the *16th European Symposium on Programming, ESOP 2007*., Braga, Portugal, pp. 95–109.

Cannasse, N., Hurt, B., Yoriyuki, Y., & Hellsten, J. *OCaml ExtLib – Extended Standard Library for Objective Caml*. http://code.google.com/p/ocaml-extlib/, Dec 2007.

CSFB. (2007) Credit Suisse First Boston http://www.csfb.com/. Accessed December 2007.

de Rauglaudre, D. (2003) *Camlp4 tutorial*. http://caml.inria.fr/pub/docs/tutorial-camlp4/index.html. Accessed December 2007.

Fluet, M. & Pucella, R. (2006) Phantom types and subtyping. *J. Funct. Program.* **16**(6), 751–791.

Garrigue, J. (1998) Programming with polymorphic variants. In *1998 ACM SIGPLAN Workshop on ML*, Baltimore, Maryland.

JaneOCaml. (2007) http://www.janestcapital.com/ocaml/index.html.

Leroy, X., Doligez, D., Garrigue, J., Rmy, D., & Vouillon, J. 2007 (May) *The Objective Caml system, documentation and user's manual – release 3.10*. INRIA.

LexiFi. (2007) http://www.lexifi.com/. Accessed December 2007.

Lieberman, H., & Hewitt, C. (1983) A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* **26**(6), 419–429.

MLton. (2007) http://mlton.org/. Accessed December 2007.

Peyton J., Simon. (2002) *Haskell 98 language and libraries*. In *The Revised Report*. http://haskell.org/onlinereport/.

Rémy, D., & Vouillon, J. (1998) Objective ML: An effective object-oriented extension to ML. *Theory prac. object syst*. **4**(1), 27–50.

Stolpmann, G. (2003). *The findlib User's Guide*. http://www.ocaml-programming.de/packages/documentation/findlib/guide-html/. Accessed December 2007.

Stolpmann, G. (2007) *GODI, The source code Objective Caml distribution*. http://www.ocaml-programming.de/godi. Accessed December 2007.

Yallop, J. (2007). Practical generic programming in OCaml. In *2007 ACM SIGPLAN Workshop on ML*. Freiburg, Germany. http://code.google.com/p/deriving/.