F-ing applicative functors

Andreas Rossberg
Google
rossberg@mpi-sws.org

Claudio Russo
Microsoft Research
crusso@microsoft.com

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

July 31, 2012

Essential to ML-style module systems is the concept of a *functor*, i.e. a function from modules to modules. Functors are ML's way of providing generic data structures and reusable program components. Yet, there is a bewildering schism in ML land regarding the actual semantics of functors, familiar to experts under the code words *generative* vs. *applicative*.

In Standard ML [3], each application of a given functor *'generates'* fresh copies of all the abstract types the functor defines. For example, two applications Set(Int) of a functor defining an abstract set type will produce two distinct and incompatible instances of this type.

For OCaml on the other hand, Leroy [2] suggested a semantics where every application of the same functor to the same argument (under certain syntactic restrictions) reproduces the same type. Consequently, every application Set(Int) would result in the same set type. That is, the functor is *'applicative'*, in the sense that it behaves purely functionally.

Applicative functor semantics is more flexible in handling certain modularity scenarios, such as "diamond import" dependencies, where the same functor is used along different paths, without creating type incompatibilities. But it also is far more intricate:

- Applicative semantics is unsound when module-level computations can be impure, such that type definitions depend on side effects. This is e.g. the case in the presence of modules packaged as first-class values [7, 5]. Both Moscow ML and, more recently, OCaml provide such packaged modules, and circumvent the soundness problem only by imposing severe (and rather unsatisfactory) restrictions on the unpacking construct.

- For functors that create some private, module-level state at each application, treating the functors as applicative leads to a violation of data abstraction. All existing semantics for applicative functors [2, 9, 8, 1] basically ignore this problem (though Dreyer et al. [1] at least provides a *manual* means for preventing applicative semantics through its "strong sealing" construct).

- Even for pure functors like Set, care must be taken to define the appropriate notion of module equivalence to use in comparing their arguments. For instance, applying the Set functor to a module that compares integers by $<$, and to one comparing them by $>$, should produce distinct set types. Again, none of the existing semantics ensures that in the general case.

So what's the deal with applicative functors? When do they make sense?

The key is *purity*: a functor ought only be treated as applicative if the computation performed by its body is sufficiently pure. But at the same time, we don't want (in an impure language like ML) to forbid impure functors. The only satisfactory approach for having applicative functors in the language thus is to make them *coexist* with generative ones.

Based on our F-ing modules approach [6] – which in its original incarnation [5] only considered generative functors – we present a novel design for a module system that smoothly combines generative and applicative functors. While previous designs for such a combination provided two separate variants of functor expression [4], or even two sealing operators [1], our design is both simpler and more natural: only one form of each construct exists, and the only distinction the semantics makes is solely based on purity. A functor is applicative if and only if its body is a provably pure module expression, otherwise it is generative. Likewise, sealing is *weak* (in the parlance of Dreyer et al. [1]) if and only if it seals a pure module, and *strong* otherwise. Moreover, there is a natural subtyping relation between pure and impure functor signatures. We reuse ML's *value restriction* for a simple approximation of purity of core terms.

We also revive (in refined form) the long-lost notion of *structure sharing* from Standard ML '90. Although previous work on module type systems has disparaged structure sharing as type-theoretically questionable, we observe that some variant of it is in fact *necessary* in order to provide a proper treatment of abstraction in the presence of applicative functors – i.e., a semantics where type equivalence is stable under renaming and substitution of pure and transparent module expressions. It turns out that it is straightforward to account for sharing using phantom types in the F-ing approach.

References

[1] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *POPL*, 2003.

[2] X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, 1995.

[3] R. Milner, M. Tofte, and R. Harper. *The Revised Definition of Standard ML*. MIT Press, 1996.

[4] S. Romanenko, C. Russo, and P. Sestoft. Moscow ML Version 2.0, 2000. Available at `http://www.dina.kvl.dk/~sestoft/mosml`.

[5] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. In *TLDI*, 2010.

[6] A. Rossberg, C. Russo, and D. Dreyer. F-ing modules, 2012. Submitted for publication. Available at `http://www.mpi-sws.org/~rossberg/publications.html`.

[7] C. Russo. First-Class Structures for Standard ML. *Nordic Journal of Computing*, 7(4):348—374, 2000.

[8] C. Russo. Types for Modules. *Electronic Notes in Theoretical Computer Science*, 60, 2003.

[9] Z. Shao. Transparent modules with fully syntactic signatures. In *ICFP*, 1999.