Purely functional non-determinism.

Based on discussions with Simon Marlowe, Simon Peyton Jones, Andy Moran, and Barry Jay.

Draft: April 13, 2000.

A great achievement of modern functional programming languages is their use in modelling other deterministic programming language features and programming paradigms. For example, it is easy to implement in a lazy language the deterministic subset of a parallel process calculus like CSP. More recently, Silvija Seres has produced an elegant implementation in Haskell of the angelic non-determinism that is implemented by back-tracking in Prolog; she includes even impure features like the cut. These models permit proof of algebraic laws and other reasoning methods that contribute to rational and reliable program design and optimisation.

Extending an idea of Hughes, I propose a project to extend these benefits to general non-deterministic programming in a purely functional setting. Three kinds of non-determinism are identified: demonic/erratic, as in CSP; angelic, as in logic programming; and probabilistic, as in simulation languages. Perhaps quantum computation is another kind. Separate monads should be defined for each kind of non-determinism, before exploring the extra complexity of their combination. These monads are simply described in functional notation, occasionally bringing in an existential quantifier. Algebraic properties can be proved from the definitions. However, the intended implementations achieve much higher efficiency by using non-functional features of a computer, e.g., random number generation. Such implementations do not have to satisfy the laws, not even beta-reduction!

The probabilistic monad lifts each type alpha to a probability distribution (alpha -> Real), and the other two kinds of non-determinism use the powerset monad, modelling each set of alpha by its characteristic function (alpha -> Bool). The lifting functors make all existing deterministic functions available in the non-deterministic domain. Demonic non-determinism is distinguished from angelic by prohibiting certain operators, e.g., intersection and the empty set. These permit the set to be efficiently implemented by choosing just a single one of its members, it matters not which. Similarly, probabilistic choice is implemented by immediate probabilistic selection of a single representative. Angelic non-determinism is implemented in the manner proposed by Seres, by storing a list of possibilities in arbitrary order. Using standard techniques, these implementation should be proved faithful to the abstract specification; then the programmer never has to think at the concrete level (except, unavoidably, when debugging).

Means are required for escaping from the non-determinism monads back to the functional world. In the case of non-determinism, this takes the form of a choice function that is defined only for unit sets. Its reliable use requires a proof of uniqueness of the result. Similarly, to escape from probability, the desired result must be obtained with probability one. Alternatively, one could escape from non-determinism to the general IO monad, or other monad that enforces only linear use of results. For this, no proof is needed, -- fortunately, because such a proof would be even more difficult.

Peyton Jones has suggested that a programmer be given the option to declare to the compiler any collection of equational laws that are postulated as valid for the *abstract* interpretation of the algebraic data types involved. Taking advantage of these, it is possible that optimisations at the higher level of abstraction can make programs more efficient than any amount of detailed optimisation at lower levels. For example, symmetry allows the angelic *x union {1}* to be optimised just to *cons 1 x*. Of course, one instance of a call may be optimised, and another instance of the same call of the same function with the same parameters may not. Thus they will deliver different concrete results. But this does not matter, because the programmer has declared a positive desire to regard the results as equivalent. If the laws are actually inconsistent, it will be possible to prove from them that everything equals everything else. Then the compiler would be justified in giving a completely arbitrary result – which can be done very fast indeed! Such algebraic collapse can be avoided by proof that the laws are satisfied by the intended abstract model.

There are two serious problems with this proposal. The first is the difficulty of high-level optimisation from laws. Even if the laws permit reduction to normal form, such forms are notoriously unoptimal. Furthermore, there is a need for at least an approximate cost model for the concrete programs produced by the optimiser at a given level. The second problem is that of providing testable assertions and diagnostic traces in a functional setting.

* John Hughes, John O'Donnell, ''Expressing and Reasoning About Non-deterministic Functional Programs,'' in Proc. of the 1989 Glasgow Workshop on Functional Prog., pp. 308-28.

* F. Warren Burton, ''Nondeterminism with Referential Transparency in Functional Programming Languages,'' in Computer Journal, 31(3), 1988, pp. 243-7.

* F. Warren Burton, ''Encapsulating non-determinacy in an abstract data type with determinate semantics,'' in Journal of Functional Programming, 1(1), 1991, pp. 3-20.

* P. L. Wadler, ''How to replace failure by a list of successes,'' in Proc. Functional Programming Languages and Computer Architecture, La Jolla, June 1995, ACM.