Ben Lippmeier
Australian National University

Type Inference and Optimisation for an Impure World.

In the land of functional programming a bitter war rages between the ideals of purity and the temptation of side effects. On one hand, the banishment of effects from a pure language makes programs easier to reason about, permits safe lazy evaluation and accommodates a host of clever compiler optimisations. On the other hand, computational effects are required to support destructive update and for interaction with the outside world. At some point, all practical, general purpose languages must interact with the outside world. This interaction is a computational effect, and effectful expressions have an implicit ordering which must be respected. To change the order of conflicting effects would be to change the meaning of the program.

There are two traditional methods for keeping effects in order. The simplest approach - that of allowing arbitrary, untracked effects at any point in the program - drastically reduces the applicability of compiler optimisations such as let-floating and the full laziness transform. Both of these optimisations involve the re-ordering of function applications and cannot be performed when there is any chance re-ordering their associated effects. The other approach - that of encapsulating all side effects in some form of state monad - tends to fracture a language into two separate sub-languages with reduced compositionality. One of the results of this fracturing can be seen in Haskell with the proliferation of combinators with both 'pure' and monadic versions, ie map vs mapM, filter vs filterM, foldl vs foldM. Both versions perform the same conceptual operation, yet each is distinct and incompatible.

One solution for this problem, and the solution to be outlined in this talk, is to allow arbitrary effects in the program and then perform a type based effect analysis at compile time. The result of this effect analysis is used to annotate the intermediate language with the lub of the effects that can be generated by each function application. This information is used to guide the optimisations so that the order of conflicting effects can be preserved.

We have a working, prototype compiler which accepts a Haskell-like language and uses C as its target. We use the second order lambda calculus as an intermediate language and include region, effect and closure information along with the types. Effect information is used to guide our optimisations, region information is used to mask effects which are local to a given function, and closure information is used to track the sharing of regions between functions. The language allows arbitrary data structures to be updated without requiring the use of the Ref type as in ML. We support programmer-introduced lazy evaluation, and make use of our type system to ensure that only expressions without visible effects may be suspended. We also build on Leroy's work on using closure typing to eliminate the unsoundness introduced into a Hindley-Milner style type system by polymorphic update.