IO in a Lazy Functional Logic Language

Matthew Mirman

August 22, 2011

Suppose we have a lazy, functional logical language, with subtyping, parameterized types, and either rank 1 bounded quantification, or type classes. I present a type level static solution to ensuring that logical nondeterminism does not enter the IO monad, and cause unwanted effects. In the case of a nominal type system with type classes, I also present a method for ensuring the unifiability of the free variables.

The language has three logical functions: free x in e, findall x in e, and $e_1 =:= e_2$. free x in e states that x is a free logical variable in e. findall x in e searches for all x that satisfy e, and makes x a free logical variable. $e_1 =:= e_2$, unifies e_1 with e_2 and nondeterministically branches on free variables. In order to unify, the variables must be unifiable, and constructed from products and sums only.

We assign rough types to these functions as follows:

$free :: (Unifiable\ a) => (a \to b) \to b$,

$findall :: (Unifiable\ a) => (a \to Success) \to [a]$ and

$(=:=) :: Success \to Success \to Success$.

In a pure lazy functional setting, such as haskell, IO has traditionally been accomplished with the IO monad. In order to not deviate from this pattern, it is necessary to ensure that computations still make sense from within the IO monad, and the scope of the free variables does not make it to effects, unnecessarily. Ensuring that the nondeterminism of free does not make it into the IO monad statically can be made into a type level problem by the following system.

Suppose we have two parameterized types $Many\ a$ and $Single\ a$, with the following subtyping rules:

$(S - ReturnQuantity)\frac{\Gamma \vdash a <: b}{\Gamma \vdash Single\ a <: Many\ b}$

First, a typeclass solution: assume $Single$ and $Many$ are instances of the built in of the $ReturnQuantity$ typeclass. We then construct a new application function

$\$\$ :: ReturnQuantity\ m => (a \to m\ b) \to\ (m\ a \to m\ b)$

The main of the program will have type $Single\ (IO\ ())$, and $return :: a -> Single\ (m\ a)$

Then the two determinism altering logical functions will have the following types:

$free :: (Unifiable\ a) => (a \to Many\ b) \to Many\ b$

$findall :: (Unifiable\ a) => (a \to Many\ Success) \to Single\ [a]$

This way, any function which is deterministic can be used anywhere a nondeterministic function can be used, but not visa versa.

To make this feature not cumbersome to the user, we walk the abstract syntax tree, and every time there is a function application

$e_1\ e_2$, we transform it to $e_1\ \$\$e_2$. This can be done with the following rule:

$\phi(e_1\ e_2) = \phi(e_1)\$\$\phi(e_2)$.

$\phi(\lambda x.e) = \phi(\lambda x.e)$.

An advantage of this transformation is the proof of decidability of type inference is trivial, as the proof that the syntactic transformation preserves the intended semantics is trivial, and the decidability of type inference after the transformation is known.

with obvious extensions to objects and switchables.

The associated solution with rank 1 bounded quantification would be to assign the following type:

$\$\$ :: \forall m <: Many => (a \to m\ b) \to\ (m\ a \to m\ b)$