Call by Effect

Kevin Ellis

September 25, 2009
Reconcile side effects with lazy evaluation

- Concise: No explicit managing of state or evaluation order
- Natural: Code should “just work”
- Call by Effect (attempts) these two requirements
Adding two numbers from a file

- Haskell:
  \[ \text{liftM2 (+) readi readi} \]

- Clean:
  \[
  \text{let (x,f2) = readi f1} \\
  \text{(y,f3) = readi f2} \\
  \text{in (x + y,f3)}
  \]

- ML:
  \[ \text{readi () + readi ()} \]

- A language using Call by Effect would express this identically to the ML code.
Mapping across a stream

- A lazy language:
  ```
  map f [] = []; map f (x:xs) = f x : map f xs
  ```

- A strict language:
  ```
  map f [] = []; map f (x:xs) =
      delay (f x) : delay (map f xs)
  ```

- A language using Call by Effect would express this identically to the lazy code.
Effect Types

- Tells us if a term could have a side effect or not
- Each term has a type and an effect
- map: $\forall \alpha, \beta, \sigma. (\alpha \xrightarrow{\sigma} \beta) \rightarrow [\alpha] \sigma \rightarrow [\beta]$
Overview

- If a term has a side effect: evaluate it
- If a term doesn’t have a side effect: substitute it unevaluated
- Call by Value + Call by Name + Effect Types = Call by Effect
Semantics

\[ t_1 \mid q \rightarrow t'_1 \mid q' \quad (E-\text{App}) \]

\[ t_1 t_2 \mid q \rightarrow t'_1 t_2 \mid q' \]

\[ \emptyset \vdash t_2 : \tau, \emptyset \]

\[ (\lambda x. t_1) t_2 \mid q \rightarrow [x \mapsto t_2] t_1, q \quad (E-\beta_N) \]

\[ \emptyset \vdash t_2 : \tau, \{\text{Unit}\} \]

\[ t_2 \mid q \rightarrow^* v_2 \mid q' \]

\[ (\lambda x. t_1) t_2 \mid q \rightarrow (\lambda x. t_1) v_2 \mid q' \\ (E-\beta_V) \]
Properties

- Soundness: Preservation and Progress. (Thm. 1,2)
- There exists terms which have a normal form under call by name but diverge under call by effect. (Thm. 3)
- Let $t$ be a well-typed term in the lambda calculus. Then $t$ has the same normal form under call by name and call by effect. (Not in paper)
- There exists terms which have a normal form under call by effect but diverge under call by value. (Thm. 4)
- Let $t | q \rightarrow^* v | q'$ under call by value. Then there exists $v'$ such that $t | q \rightarrow^* v' | q'$ under call by effect. (Thm. 5)
Monomorphic

- Implementation is easy: translate operational semantics into code
- Problem: Runtime typing of terms!
- Not a problem in a monomorphic language: can statically find all types
- Perform a type-directed translation into a calculus with explicitly annotated applications
Translation

- Original:
  add 1 (print 2)
- Translation:
  strict (lazy add 1) (lazy print 2)
- Every application has been statically resolved.
Let-Polymorphism

- Polyinstantiation upon generalized effect variables in type scheme
- Type-directed translation upon monomorphic translation
- Can determine from just the definition of a term all monomorphic instantiations
- No polymorphic recursion
Polyinstantiation

```
let poly = \ x y -> x (y 0)
in poly (const 5) print
```

polyinstantiates to:

```
let poly' = \ x y -> strict x (y 0)
    poly'' = \ x y -> lazy x (y 0)
in poly' (const 5) print
```
Mapping over a tree

Tree a ::= Leaf a | Branch (Tree a) (Tree a)

let tmap f (Leaf a) = Leaf (f a)
    tmap f (Branch l r) = Branch (tmap f l) (tmap f r)
in tmap print (tmap (+1) tree)
take (readInteger stdin) (map (**2) [1..])

where map would be automatically replaced with a call to the monomorphically map for mapping pure functions across lists.
Differences

- Scheme: lazy evaluation is easy in a strict language
- Haskell: Side effects look like an imperative language
- Clean: State objects are used in a single-threaded manner
- Call by Effect: Lazy and strict evaluation mashed together
Disadvantages

- No clean separation of pure from stateful code
- No explicit programmer control over sequencing of effects (can be added)
- No separation of an effect and an action that performs that effect: `mapM_ action VS map action`
- No polymorphic recursion, etc (yet)
Advantages

- Automates the process of deciding when a term should be evaluated
- Can lead to more concise code
- Can eliminate the need for monadic and non-monadic versions of higher-order functions
Summary and contributions

- Created a new evaluation strategy
- Implemented it
- Compared it to monads and uniqueness types
- Demonstrated it’s usefulness in certain cases
Future Work

- Expand implemention
- No more polyinstantiation
- Find an elegant way of restricting side effects
In the paper...

- Code examples, including interactions with the interpreter
- Extended discussion
- Proofs