A Monadic Semantics for Core Curry [1]

Andrew Tolmach and Sergio Antoy [2]

Dept. of Computer Science, Portland State University,
P.O.Box 751, Portland OR 97207, USA

Abstract

We give a deterministic, big-step operational semantics for the essential core of
the Curry language, including higher-order functions, call-by-need evaluation, non-
determinism, narrowing, and residuation. The semantics is structured in modular
monadic style, and is presented in the form of an executable interpreter written in
Haskell. It uses monadic formulations of state, non-determinism, and resumption-
based concurrency.

Key words: Curry, functional-logic languages, narrowing,
residuation, non-determinism, monads, resumptions.

1 Introduction

The functional logic language Curry combines lazy functional programming
with logic programming features based on both narrowing and residuation.
Describing the semantics of these features and their interaction in a common
framework is a non-trivial task, especially because functional and logic pro-
gramming have rather different semantic traditions. The "official" semantics
for Curry [4], largely based on work by Hanus [6,5], is an operational seman-
tics based on definitional trees and narrowing steps. Although fairly low-level,
this semantics says nothing about sharing behavior. Functional languages
are typically given denotational or natural ("big step") operational seman-
tics. In more recent work [1], Albert, Hanus, Huch, Oliver, and Vidal propose
a natural semantics, incorporating sharing, for the first-order functional and
narrowing aspects of Curry; however, in order to collect all solutions of non-
deterministic programs and to incorporate residuation, they fall back on a
small-step semantics. While small-step semantics have the advantage of being

```
data Exp = Var Var                  type Var = String
         | Int Int                  type Constr = String
         | Abs Var Exp              type Prim = String
         | App Exp Exp              type Pattern = (Constr,[Var])
         | Capp Constr [Exp]
         | Primapp Prim Exp Exp
         | Case Exp [(Pattern,Exp)]
         | Letrec [(Var,Exp)] Exp
```

Fig. 1. Expressions.

closer to usable abstract machines, they tend to be lower-level than big-step models, and perhaps harder to reason about.

In this paper, we describe a variant of the semantics by Albert, *et al.* that remains big-step, but delivers all solutions of non-deterministic programs and handles residuation. It is also fully higher-order. Our treatment is not especially original; rather, we have attempted to apply a variety of existing techniques to produce a simple and executable semantic interpreter. In particular, we follow Seres, Spivey, and Hoare [17] in recording multiple solutions to non-deterministic programs in a lazily constructed *forest*, which can be traversed in a variety of orders to ensure fair access to all solutions. We use *resumptions* [16, Ch. 12] to model the concurrency needed to support residuation. We organize the interpreter using *monads* in the style popularized by Wadler [19], which allows a modular presentation of non-determinism [7,3] and concurrency. We believe that the resulting semantics will be useful for understanding Curry and exploring language design alternatives. It may also prove useful as a stepping stone towards new practical implementations of the language.

Our semantics is defined as an executable interpreter, written in Haskell, for a core subset of Curry. We present the interpreter in three phases. Section 2 describes the evaluation of the functional subset of the language. Section 3 introduces non-determinism, logic variables, and narrowing. Section 4 adds residuation. Section 5 offers brief conclusions. The reader is assumed to be familiar with Haskell and with monads.

2   The Functional Language Interpreter

The abstract syntax for the functional part of our core expression language is given as a Haskell data type definition in Figure 1. The language is not explicitly typed, but we assume throughout that we are dealing with typeable expressions. Function abstractions (`Abs`) have exactly one argument; multiple-argument functions are treated as nested abstractions. Functions need not be lambda-lifted to top level. For simplicity, the only primitive type is integers (`Int`) and all primitive operators are binary; these limitations could be easily lifted. Primitive operators, which are named by strings, act only on con-

structor head-normal forms. Each constructor is named by a string c, and is assumed to have a fixed arity $ar(c) \geq 0$. Constructors can be invented freely as needed; we assume that at least the nullary constructors `True`, `False`, and `Success` are available.

Constructor applications (`Capp`) and patterns (within `Case` expressions) and primitive applications (`Primapp`) are fully applied. Unapplied or partially applied constructors and primitives in the source program must be η-expanded. `Case` expressions analyze constructed values. Patterns are "shallow;" they consist of a constructor name c and a list of $ar(c)$ variables. The patterns for a single `Case` are assumed to be mutually exclusive, but not necessarily exhaustive. More complicated pattern matching in the source language must be mapped to nested `Case` expressions. Such nested case expressions can be used to encode definitional trees [2]. Finally, `Letrec` expressions introduce sets of (potentially) mutually recursive bindings.

Curry is intended to use "lazy" or, more properly, *call-by-need* evaluation, as opposed to simple call-by-name evaluation (without sharing). Although the results of call-by-name and call-by-need cannot be distinguished for purely functional computations, the time and space behavior of the two strategies are very different. More essentially, the introduction of non-determinism (Section 3) makes the difference between strategies observable. We therefore model call-by-need evaluation explicitly using a mutable *heap* to represent shared values. (The heap is also used to represent recursion without recourse to a Y-combinator.) This approach corresponds to the behavior of most real implementations of call-by-need languages; its use in formal semantics was introduced by Launchbury [11] and elaborated by Sestoft [18], and was also adopted by Albert, *et al.* [1].

Following a very old tradition [10], we interpret expressions in the context of an *environment* that maps variables to heap locations, rather than performing substitution on expressions. The same expression (e.g., the body of an abstraction) can therefore evaluate to different values depending on the values of its free variables (e.g., the abstraction parameter). In this approach we view the program as immutable code rather than as a term in a rewriting system.

The main evaluation function is

```
eval :: Env -> Exp -> M Value
```

which evaluates an expression in the specified environment and returns the corresponding constructor head-normal form (HNF) value, embedded in a monadic computation. The monad

```
newtype M a = M (Heap -> A (a,Heap))
```

represents stateful computations on heaps. Type `A a` is another monad, representing *answers* involving type `a`.

The key type definitions, shown in Figure 2, are mutually recursive. Values correspond to HNFs of expressions. Constructed values (`VCapp`) correspond

```
type HPtr = Int
data Value =
    VCapp Constr [HPtr]
  | VInt Int
  | VAbs Env Var Exp
type Env = Map Var HPtr
data Heap = Heap {free :: [HPtr], bindings :: Map HPtr HEntry}
data HEntry =
    HValue Value
  | HThunk Env Exp
  | HBlackhole
```

Fig. 2. Key data types for evaluation.

```
data Map a b = Map [(a,b)]
mempty :: Map a b
mempty = Map []
mget :: Eq a => Map a b => a -> b
mget (Map l) k = fromJust (lookup k l)
mset :: Map a b -> (a,b) -> Map a b
mset (Map l) (k,d) = Map ((k,d):l)
```

Fig. 3. A specification for the `Map` ADT. Note that the ordering of the list guarantees that each `mset` of a given key supersedes any previous `mset` for that key. An efficient implementation would use a sorted tree or hash table (and hence would put stronger demands on the class of `a`).

to constructor applications; the components of the value are described by pointers into the heap (`HPtr`). Closures (`VAbs`) correspond to abstraction expressions, tagged with an explicit environment to resolve variable references in the abstraction body. Environments and values are only well-defined in conjunction with a particular heap that contains bindings for the heap pointers they mention.

A heap is a map `bindings` from heap pointers (`HPtr`) to heap entries (`HEntry`), together with a supply `free` of available heap pointers. Heap entries are either HNF values, unevaluated expressions (`HThunk`), or "black holes" (`HBlackhole`). We expect thunk entries to be overwritten with value entries as evaluation proceeds. Black holes are used to temporarily overwrite a heap entry while a thunk for that entry is being computed; attempting to read a black-hole value signals (one kind of) infinite recursion in the thunk definition [8,18].

Both `Env` and `Heap` rely on an underlying abstract data type `Map a b` of applicative finite maps from `a` to `b`, supporting simple get and set operations (see Figure 3). Note that `mset` returns a *new* map, rather than modifying an existing one. It is assumed that `mget` always succeeds. The map operations are lifted to the `bindings` component of heaps as `hempty`, `hget`, and `hset`.

```
newtype M a = M (Heap -> A (a,Heap))
instance Monad M where
  (M m1) >>= k = M (\h -> do (a',h') <- m1 h
                             let M m2 = k a' in m2 h')
  return x = M (\h -> return (x,h))
instance MonadPlus M where
  mzero = M (\_ -> mzero)
  (M m1) `mplus` (M m2) = M (\h -> m1 h `mplus` m2 h)
fresh :: M HPtr
fresh = M (\h -> return (hfresh h))
store :: HPtr -> HEntry -> M ()
store p e = M (\h -> return ((),hset h (p,e)))
fetch :: HPtr -> M HEntry
fetch x = M (\h -> return (hget h x,h))
run :: M a -> A (a,Heap)
run (M m) = m hempty
```

Fig. 4. The evaluation monad.

The function

```
hfresh :: Heap -> (HPtr,Heap)
```

picks and returns a fresh heap pointer. As evaluation progresses, the heap can only grow; there is no form of garbage collection. We also don't perform environment *trimming* [18], though this would be easy to add.

The evaluation function returns a monadic computation M Value, which in turn uses the answer monad A. Using monads allows us to keep the code for eval simple, while supporting increasingly sophisticated semantic domains. Our initial definition for M is given in Figure 4. Note that M is essentially just a function type used to represent computations on heaps.[3] The "current" heap is passed in as the function argument, and a (possibly updated) copy is returned as part of the function result. As usual, bind (>>=) operations represent sequencing of computations; return injects a value into the monad without changing the heap; mzero represents a failed evaluation; mplus represents alternative evaluations (which will be used in Section 3). The monad-specific operations include fresh, which returns a fresh heap location; fetch, which returns the value bound to a pointer (assumed valid) in the current heap; and store, which extends or updates the current heap with a binding. The function run executes a computation starting from the empty heap.

Type A is also a monad, representing *answers*. Note that the uses of bind, return, mzero and mplus in the bodies of the functions defined on M are actually the monad operators for A (*not* recursive calls to the M monad operators!). In this section, we equate A with the exception monad Maybe a, so that an

---

[3] We introduce the newtype and "wrapper" data constructor M only so that we can make M an instance of the Monad class, which allows us to use Haskell's "do notation" for sequences of bind operations.

5

```
instance Monad Maybe where            instance MonadPlus Maybe where
    Just x  >>= k = k x                    mzero        = Nothing
    Nothing >>= k = Nothing
    return        = Just
```

Fig. 5. The `Maybe` type as a monad.

answer is either `Just` a pair (HNF value,heap) or `Nothing`, representing "failure." Failure occurs only when a required arm is missing from a non-exhaustive `Case` expression, or when an attempt is made to fetch from a black-holed heap location. A gets instance definitions of `>>=`, `return`, and `mzero` for `Maybe` from the standard library (Figure 5). Note that (`>>=`) propagates failure.

With this machinery in place, the actual `eval` function is quite short (Figure 6). Evaluation of expressions already in HNF is trivial, except for constructor applications, for which each argument expression must be allocated as a separate thunk (since it might be shared). Evaluation of applications is also simple. Assuming that the program is well-typed, the operator expression must evaluate to an abstraction. The argument expression is allocated as a thunk and bound to the formal parameter of the abstraction; the body of the abstraction is evaluated in the resulting environment.

`Letrec` bindings just result in thunk allocations for the right-hand sides. To make the bindings properly recursive, all the thunks share the same environment, to which all the bound variables have been added.

The key memoization step required by call-by-need occurs when evaluating a `Var` expression. In a well-typed program, each variable must be in the domain of the current environment. The corresponding heap entry is fetched: if this is already in HNF, it is simply returned. If it is a thunk, it is recursively evaluated (to HNF), and the resulting value is written over the thunk before being returned.

A `Case` expression is evaluated by first recursively evaluating the expression being "cased over" to HNF. In a well-typed program, this must be a `VCapp` of the same type as the case patterns. If the `VCapp` constructor matches one of the patterns, the pattern variables are bound to the corresponding `VCapp` operands (represented by heap pointers), and the corresponding right-hand side is evaluated in the resulting environment. If no pattern matches, the evaluation fails, indicated by returning `mzero`.

To evaluate a primitive application, the arguments are evaluated to HNF in left-to-right order and the resulting values are passed to the auxiliary function `doPrimapp` (Figure 7), which defines the behavior of each primitive operator.

Finally, to interpret a whole-program expression, we can define

```
interp :: Exp -> Maybe (Value,Heap)
interp e = run (eval mempty e)
```

which executes the monadic computation produced by evaluating the program in an empty initial environment. If we are only interested in the head con-

```
eval :: Env -> Exp -> M Value
eval env (Int i) =  return (VInt i)
eval env (Abs x b) = return (VAbs env x b)
eval env (Capp c es) =
    do ps <- mapM (const fresh) es
       zipWithM_ store ps (map (HThunk env) es)
       return (VCapp c ps)
eval env (App e0 e1) =
    do VAbs env' x b <- eval env e0
       p1 <- fresh
       store p1 (HThunk env e1)
       eval (mset env' (x,p1)) b
eval env (Letrec xes e) =
    do let (xs,es) = unzip xes
       ps <- mapM (const fresh) xes
       let env' = foldl mset env (zip xs ps)
       zipWithM_ store ps (map (HThunk env') es)
       eval env' e
eval env (Var x) =
    do let p = mget env x
       h <- fetch p
       case h of
         HThunk env' e' ->
           do store p HBlackhole
              v' <- eval env' e'
              store p (HValue v')
              return v'
         HValue v -> return v
         HBlackhole -> mzero
eval env (Case e pes) =
    do VCapp c0 ps <- eval env e
       let plookup [] = mzero
           plookup (((c,xs),b):pes) | c == c0   = return (xs,b)
                                    | otherwise = plookup pes
       (xs,b) <- plookup pes
       let env' = foldl mset env (zip xs ps)
       eval env' b
eval env (Primapp p e1 e2) =
    do v1 <- eval env e1
       v2 <- eval env e2
       return (doPrimapp p v1 v2)
```

Fig. 6. Call-by-need `eval` function.

7

```
doPrimapp :: Prim -> Value -> Value -> Value
doPrimapp "eq" (VInt i1) (VInt i2) | i1 == i2  = VCapp "True" []
                                   | otherwise = VCapp "False" []
doPrimapp "add" (VInt i) (VInt j) = VInt (i+j)
doPrimapp "xor" (VCapp "True" []) (VCapp "True" []) =
                VCapp "False" []
doPrimapp "xor" (VCapp "True" []) (VCapp "False" []) =
                VCapp "True" []
...
```

Fig. 7. Auxiliary `doPrimapp` function.

structor of the result value, we can project out the `Value` component and ignore the `Heap`.

3 Non-determinism, Logic Variables, and Narrowing

We now revise and extend our interpreter to incorporate the key logic programming features of Curry. To do this, we must record multiple possible results for evaluation, by redefining the monadic type `A` of answers. (Changing `A` implicitly also changes `M`, although the code defining `M`'s functions doesn't change.) By choosing this monad appropriately, we can add non-deterministic features to our existing interpreter without making any changes to the deterministic fragment; the hidden "plumbing" of the bind operation will take care of threading the multiple alternatives. Deterministic choices are injected into the monad using `return`; non-deterministic choice will be represented by `mplus`; as before, failure of (one) non-deterministic alternative will be represented by `mzero`. The definition of `A` is addressed in Section 3.2.

3.1 Logic Variables and Narrowing

First, we show how to add logic variables and narrowing to the language and interpreter. This requires surprisingly little additional machinery, as shown in Figure 8. We add a new expression form, `Logic`, to declare scoped logic variables; the Curry expression "e `where x free`" is encoded as (`Logic "x"` e). We add a corresponding HNF `VLogic HPtr`, which is essentially a reference into the heap. `Logic` declarations are evaluated by allocating a fresh heap location p, initially set to contain `VLogic` p, binding the logic variable to this location, and executing the body in the resulting environment.

We now must consider how to handle `VLogic` values within the `eval` function. The most important change is to `Case`; if the "cased-over" expression is bound to a `VLogic` value, the evaluator performs *narrowing*. (We temporarily assume all cases are "flexible.") This is done by considering each provided case arm in turn. For a pattern with constructor c and argument parameters $x_1, \ldots, x_n$, the evaluator allocates n fresh logic variable references $p_1, \ldots, p_n$, overwrites the cased-over logic variable in the heap with

```
lift :: M a -> M a
lift (M m) = M (\h -> forestLift (m h))

data Exp = Logic Var Exp | ... as before ...
data Value = VLogic HPtr | ... as before ...

eval env (Var x) =
    lift $ ... as before ...

eval env (Logic x e) =
    do p <- allocLogic
       eval (mset env (x,p)) e

eval env (Case e pes) =
    do v <- eval env e
       case v of
         VCapp c0 ps -> ... as before ...
         VLogic p0 ->
           foldr mplus mzero (map f pes)
           where
             f ((c,xs),e') =
               do ps <- mapM (const allocLogic) xs
                  store p0 (HValue (VCapp c ps))
                  let env' = foldl mset env (zip xs ps)
                  eval env' e'

eval env (Primapp p e1 e2) =
    do v1 <- eval env e1
       v2 <- eval env e2
       checkGround v1
       checkGround v2
       return (doPrimapp p v1 v2)

allocLogic :: M HPtr
allocLogic = do p <- fresh
                store p (HValue (VLogic p))
                return p

checkGround :: Value -> M ()
checkGround (VLogic _) = mzero
checkGround _ = return ()
```

Fig. 8. Evaluating logic features.

9

(`VCapp` c `[`p_1`,...,`p_n`]`), extends the environment with bindings of x_i to p_i, and recursively evaluates the corresponding case arm in that environment. All the resulting monadic computations are combined using `mplus`; this is the only place where `mplus` is used in the interpreter, and hence the sole source of non-determinism. Note that other possible non-deterministic operators can be encoded using `Case`; e.g., McCarthy's "amb" operator [13] (usually called "or" in Curry) can be encoded as

```
amb e₁ e₂ ≡ Logic "dummy" (Case (Var "dummy")
                                 [(("Ldummy",[]), e₁),
                                  (("Rdummy",[]), e₂)])
```

Adding non-determinism makes the sharing semantics of call-by-need evaluation observable: For example, consider these two definitions:

```
coin1 ≡ Letrec [("c",amb (Int 0) (Int 1))]
                      (Primapp "add" (Var "c") (Var "c"))
coin2 ≡ Primapp "add" (amb (Int 0) (Int 1))
                              (amb (Int 0) (Int 1))
```

Here `coin1` evaluates to $\{0,2\}$, whereas `coin2` evaluates to $\{0,1,2\}$.

We also need to make another small change to the `eval` code (adding calls to `checkGround`) to guard against the appearance of a `VLogic` value in an operand position of a `Primapp`; in this case, `eval` will return `mzero`. Note that because of this possibility for failure, the left-to-right evaluation semantics of `Primapp`s can be observed. For example, the evaluation of

```
Logic "x"
  (Primapp "xor"
      (Var "x")
      (Case (Var "x") [(("True",[]), Capp "False" [])]))
```

fails, but would succeed (with `True`) if the order of arguments to `and` were reversed. This characteristic may seem rather undesirable; we consider alternatives in Section 4.

Another important component of Curry is the strict equational unification operator (`=:=`) used to specify constraints. We can almost capture the behavior of this operator using `Case`. For example, Figure 9 shows two type-specific instances of the operator. Similar instances can be generated statically for all uniform data types [14, Ch. 10.1] involving just constructors; a separate mechanism is still needed to handle primitive types and non-uniform data types. Admittedly, these encodings are not very efficient, because unifying a logic variable with a term creates a copy of the term. Moreover, applying a recursive operator such as `eqBoolList` to two logic variables always produces an infinite set of answers; this is semantically acceptable, provided that we handle non-determinism fairly (Section 3.2), but very wasteful. A more efficient approach would be to treat `=:=` as a new kind of core language expression, whose implementation could test whether a value is a logic variable and could perform direct unification of two variables. However, adding sup-

```
eqBool e₁ e₂ ≡
  Case e₁
    [(("True",[]), Case e₂ [(("True",[]), Capp "Success" [])]),
     (("False",[]), Case e₂ [(("False",[]), Capp "Success" [])])]

eqBoolList e₁ e₂ ≡
Letrec [("ebl",
          Abs "e1" (Abs "e2"
          (Case (Var "e1")
             [(("Nil",[]),
                Case (Var "e2" [(("Nil",[]),
                                  Capp "Success" [])])),
              (("Cons",["h1","t1"]),
                Case (Var "e2")
                  [(("Cons",["h2","t2"]),
                      Case (eqBool (Var "h1") (Var "h2"))
                        [(("Success",[]),
                            App (App (Var "ebl" (Var "t1"))
                              (Var "t2")))])])])))]
        (App (App (Var "ebl") e₁) e₂)
```

Fig. 9. Instances of =:= for booleans and lists of booleans. Each operator returns Success if the unification succeeds, and fails otherwise.

port for these features would noticeably complicate the interpreter (e.g., by introducing chains of pointers in the heap), so it is noteworthy that they are not essential for obtaining the desired semantics in most cases.

3.2 Monads for Non-determinism

It remains to define type A in such a way that it can record multiple non-deterministic answers. The standard choice of monad for this purpose is *lists* [19]. In this scheme, non-deterministic choice of a value is represented by a list of values; return a produces the singleton list $[a]$; mplus is concatenation (++); m >>= k applies k to each element in m and concatenates the resulting lists of elements; and mzero is the empty list []. However, if we want to actually execute our interpreter and inspect the answers, the list monad has a significant problem: its mplus operation does not model *fair* non-deterministic choice. Essentially this is because evaluating m_1 ++ m_2 forces evaluation of the full spine of m_1, so $\bot$ `mplus` $m = \bot$. If the left alternative leads to an infinite computation, the right alternative will never be evaluated at all. For example, evaluating

```
Letrec ["f",amb (Var "f") (Int 1)] (Var "f")
```

should produce the answer 1 (infinitely many times). However, if we represent answers by lists, our interpreter will compute (roughly speaking)

11

```
(eval (Var "f")) `mplus` (return (VInt 1))
= (eval (Var "f")) ++ [VInt 1]
```

If we attempt to inspect this answer, we immediately cause a recursive evaluation of `f`, which produces the same thing; we never see any part of the answer. In effect, using this monad amounts to performing depth-first search of the tree of non-deterministic choices, which is incomplete with respect to the intuitively expected semantics.

To avoid this problem, we adopt the idea of Seres, Spivey, and Hoare [17], and represent non-determinism by a lazy *forest* of trees of values (Figure 10). We set `type A a = Forest a`. As before, we represent choices as a list, with `mplus` implemented as `(++)`, but now the lists are of trees of values. To obtain the values, we can traverse the trees using any ordering we like; in particular, we can use breadth-first rather than depth-first traversal:

```
interp :: Exp -> [(Value,Heap)]
interp =  bfs (run (eval mempty e))
```

This approach relies fundamentally on the laziness of the forest structure. Non-trivial tree structures are built using the `forestLift` operator, which converts an arbitrary forest into a singleton one by making all the trees into branches of a single new tree. Applying `forestLift` to a value v before concatenating it into the forest with `mplus` will delay the point at which v is encountered in a breadth-first traversal, and hence allow the other argument of `mplus` to be explored first. For the example above, the forest answer will have the form

```
(forestLift (eval (Var "f"))) `mplus` (return (VInt 1))
= (Forest [Fork (eval (Var "f"))]) `mplus`
  (Forest [Leaf (VInt 1)])
= Forest ([Fork (eval (Var "f"))] ++ [Leaf (VInt 1)])
= Forest [Fork (eval (Var "f")), Leaf (VInt 1)]
```

Applying `bfs` to this answer will produce `VInt 1` (the head of its infinite result) before it forces the recursive evaluation of `f`.

To make use of `forestLift`, we need to add a new `lift` operator to `M`, defined in Figure 8. We have some flexibility about where to place calls to `lift` in our interpreter code. For fairness, we must make sure that there is a `lift` in every infinite cycle of computations made by the interpreter. The simplest way to guarantee this is to apply `lift` on each call to `eval`. If we do this, there is a clear parallel between the evaluation of the answer structure and the behavior of a *small-step* semantics. However, more parsimonious placement of `lift`s will also work; for example, since every cyclic computation must involve a heap reference, it suffices to `lift` only the evaluation of `Var` expressions, as shown in Figure 8.

```
newtype Forest a = Forest [Tree a]
data Tree a = Leaf a | Fork (Forest a)
instance Monad Forest where
  m >>= k = forestjoin (forestmap k m)
  return a = Forest [Leaf a]
instance MonadPlus Forest where
  mzero = Forest []
  (Forest m1) 'mplus' (Forest m2) = Forest (m1 ++ m2)


forestLift :: Forest a -> Forest a
forestLift f = Forest [Fork f]


forestjoin :: Forest (Forest a) -> Forest a
forestjoin (Forest ts) = Forest (concat (map join' ts))
  where join' :: Tree (Forest a) -> [Tree a]
        join' (Leaf (Forest ts)) = ts
        join' (Fork xff) = [Fork (forestjoin xff)]


treemap :: (a -> b) -> Tree a -> Tree b
treemap f (Leaf x) = Leaf (f x)
treemap f (Fork xf) = Fork (forestmap f xf)


forestmap :: (a -> b) -> Forest a -> Forest b
forestmap f (Forest ts) = Forest (map (treemap f) ts)


bfs :: Forest a -> [a]
bfs (Forest ts) = concat (bfs' ts)
 where bfs' :: [Tree a]  -> [[a]]
       bfs' ts = combine (map levels ts)
       levels :: Tree a -> [[a]]
       levels (Leaf x) = [[x]]
       levels (Fork (Forest xf)) = []:bfs' xf
       combine :: [[[a]]] -> [[a]]
       combine = foldr merge []
       merge :: [[a]] -> [[a]] -> [[a]]
       merge (x:xs) (y:ys) = (x ++ y):(merge xs ys)
       merge xs [] = xs
       merge [] ys = ys
```

Fig. 10. Monad of forests and useful traversal functions (adapted from [17]).

4 Residuation

In real Curry, appearance of a logic variable in a rigid position causes evaluation to *residuate*, i.e., suspend until the logic variable is instantiated (to a constructor-rooted expression). Residuation only makes sense if there is the possibility of concurrent computation—otherwise, the suspended computation can never hope to be restarted. The only plausible place to add concurrency to

```
type Flex = Bool
type Concurrent = Bool
type Prim = (String,Concurrent)
data Exp = Case Flex Exp [(Pattern,Exp)] | ... as before ...

eval env (Case flex e pes) =
    do v <- eval env e
       case v of
         VCapp c ps -> ... as before ...
         VLogic p0 | flex ->
           ... as before ...
           store p0  ...
           yield $
           ... as before ...
         VLogic _ | otherwise -> mzero

eval env (Primapp (p,concurrent) e1 e2) =
    do (v1,v2) <-
         if concurrent then
           conc (eval env e1) (eval env e2)
         else do v1 <- eval env e1
                 v2 <- eval env e2
                 return (v1,v2)
       ... as before ...

doPrimapp "and" (VCapp "Success" []) (VCapp "Success" []) =
                                      VCapp "Success" []
```

Fig. 11. Interpreter changes to support residuation.

our existing core language is for evaluation of arguments to primitives. We use an interleavings semantics for concurrency; the result is semantically simple though not practically efficient (since the number of interleavings can easily grow exponentially).

We can add residuation support to our core language by making only minor changes to our interpreter, as shown in Figure 11. We extend the core language syntax by adding a boolean flag to each primitive operator indicating whether its arguments are to be evaluated concurrently. The most obvious candidate for this evaluation mode is the parallel `and` operator normally used in Curry to connect pairs of constraints. We also add a flag on `Case` expressions to distinguish flexible and rigid cases.

The evaluator relies on significant changes to the underlying monad `M`, which is modified to describe concurrent computations using *resumptions* [16, Ch. 12], a standard method from denotational semantics. Each monadic computation is now modeled as a *stream* of partial computations. The `conc` operator takes two such streams and produces a computation that (non-deterministically) realizes all possible interleavings of the streams. The atom-

14

icity of interleaving is controlled by uses of `yield`; placing a `yield` around a computation indicates a possible interleaving point.

   With this monadic support in place, our approach to residuation is simple, and requires very few changes in the `eval` function. Attempts to perform a rigid case over an unresolved logic variable simply fail (just as in other strict contexts). However, arguments to concurrent primitives are evaluated using the `conc` operator, so that if there are any possible evaluation sequences that resolve the logic variable before it is cased over, those sequences will be tried (along with potentially many other sequences that lead to failure). To make this approach work, we must permit enough interleaving that all possible causal interactions between the two argument evaluation sequences are captured. A brute-force approach would be to `yield` before each recursive call to `eval`. However, since logic variable heap locations can only be updated by the `store` operation in the interpreter code for flexible `Case` expressions, it suffices to `yield` following that `store`.[4]

   To illustrate how interleaving works, we can define canonical code sequences for reading and writing logic variables:

```
wr x k ≡ Case True  (Var x) [(("True",[]), k)]
rd x k ≡ Case False (Var x) [(("True",[]), k)]
```

Here wr x k writes `True` into logic variable x (assumed to be not already set), and then continues by evaluating expression k. Conversely, rd x k attempts to read the contents of logic variable x (assumed to contain `True`); if this is successful, it continues by evaluating k; otherwise, it fails. Now consider the expression

```
Logic "x" (Logic "y" (Primapp ("and",True)
                     (rd "x" (wr "y" (Capp "Success" []))))
                     (wr "x" (rd "y" (Capp "Success" []))))))
```

The application of the concurrent primitive `and` causes evaluation of the two argument expressions to be interleaved. Each `wr` induces a `yield` immediately following the store into the variable, so there are three possible interleavings, shown in the three columns below. The first and third of these fail (at the point marked `*`); only the second succeeds.

execution order	left-arg	right-arg	left-arg	right-arg	left-arg	right-arg
1	rd "x" *			wr "x"		wr "x"
2	wr "y"		rd "x"			rd "y" *
3		wr "x"	wr "y"		rd "x"	
4		rd "y"		rd "y"	wr "y"	

---

[4]   It is also essential to use black-holing as described in Section 2, thus providing a (different) kind of atomicity around thunk evaluations; otherwise some interleavings might cause a thunk heap location to be updated twice with different values.

Note that it is perfectly possible to label ordinary primitives like `add` or `xor` as concurrent. This makes the result independent of argument evaluation order, thus removing one of the drawbacks we noted to the narrowing semantics of Section 3.1. In fact, it is hard to see that anything *less* than concurrency *can* achieve order-independence. In other words, making primitive applications independent of argument order seems to be no easier than adding residuation.

It remains to describe the implementation of resumptions, which is entirely within monad `M`, revised as shown in Figure 12. Each computation now returns one of two `State`s: either it is `Done`, producing a value, or it is still `Running`, carrying a new `M` computation describing the remainder of the (original) computation. In either case, the computation also returns an updated heap. Simple computations (such as primitive `fresh`, `store` and `fetch` operations) just return `Done` states. Computations returning `Running` states are generated by the `yield` operator. The `conc` operator non-deterministically tries both orders for evaluating its arguments at each stage of partial computation. As before, support for non-determinism is given by monad `A` (which does not change). A proof that this version of `M` actually obeys the monad laws is given by Papaspyrou [15].

5 Conclusion and Future Work

We have presented a simple executable semantics for the core of Curry. The full code for the three interpreter versions described here is available at

http://www.cs.pdx.edu/~apt/curry-monads.

The structure of our semantics sheds some light on how the basic components of the language interact. In particular, we can see that the addition of non-determinism, logic variables and narrowing can be accomplished just by making a suitable shift in interpretation monad. We could emphasize this modularity further by presenting the relevant monads as compositions of *monad transformers* [12]. This would extend existing work by Labra Gayo, *et al.* [9], in which transformers are used to structure an interpreter for a simple logic programming language.

A number of important Curry features are still missing from our account. We have already mentioned the need for a more efficient implementation of the `=:=` operator, using direct variable-variable unification. Curry also includes an encapsulated search facility, which reflects multiple results from a nested non-deterministic computation back into the language as a list; it is unclear how to achieve this in our framework.

While we think the semantics given in this paper is attractive in its own right, it would obviously be useful to give a formal characterization of its relationship with the various existing semantics for Curry; we have not yet attempted this. As additional future work, we plan to pursue the systematic transformation of this semantics into a small-step form suitable as the basis for realistic interpreters and compilers.

```
data State a = Done a
             | Running (M a)
newtype M a = M (Heap -> A (State a,Heap))
instance Monad M where
  (M m1) >>= k = M (\h ->
      do (s,h') <- m1 h
         case s of
           Done a' -> let M m2 = k a' in m2 h'
           Running m' -> return (Running (m' >>= k),h'))
  return x = M (\h -> return (Done x, h))
instance MonadPlus M where
  mzero = M (\h -> mzero)
  (M m1) `mplus` (M m2) = M (\h -> (m1 h `mplus` m2 h))
fresh :: M HPtr
fresh = M (\h -> let (v,h') = hfresh h in return (Done v,h'))
store, fetch  ... modified similarly ...
yield :: M a -> M a
yield m = M (\h -> return (Running m,h))
run :: M a -> A (a,Heap)
run = run' hempty
 where
   run' :: Heap -> M a -> A (a, Heap)
   run' h (M m) =
     do (s,h') <- m h
        case s of
          Done a -> return (a, h')
          Running m' -> run' h' m'
conc :: M a -> M b -> M (a,b)
conc m1 m2 = (m1 `thn` m2) `mplus` (liftM swap (m2 `thn` m1))
    where thn :: M a -> M b -> M (a,b)
          (M m1') `thn` m2 =  M (\h ->
              do (s,h') <- m1' h
                 case s of
                   Done a ->
                     return (Running (do b <- m2
                                         return (a,b)),h')
                   Running m' -> let M m'' = conc m' m2 in m'' h')
          swap :: (a,b) -> (b,a)
          swap (a,b) = (b,a)
```

Fig. 12. Monad changes to supporting residuation.

References

[1] Albert, E., M. Hanus, F. Huch, J. Oliver and G. Vidal, *Operational semantics for functional logic languages*, Electronic Notes in Theoretical Computer Science **76** (2002).

[2] Antoy, S., *Definitional trees*, in: *Proc. 3rd International Conference on Algebraic and Logic Programming (ALP'92)* (1992), pp. 143–157.

[3] Claessen, K. and P. Ljunglof, *Typed logical variables in Haskell*, Electronic Notes in Theoretical Computer Science **41** (2001).

[4] Hanus, M., editor, "Curry: An Integrated Functional Logic Language," Available at `http://www.informatik.uni-kiel.de/~mh/curry/`.

[5] Hanus, M., *A unified computation model for declarative programming*, in: *Proc. 1997 Joint Conference on Declarative Programming (APPIA-GULP-PRODE'97)*, 1997, pp. 9–24.

[6] Hanus, M., *A unified computation model for functional and logic programming*, in: *Proc. POPL'97, 24st ACM Symp. on Principles of Programming Languages*, 1997, pp. 80–93.

[7] Hinze, R., *Prological features in a functional setting — axioms and implementations*, in: M. Sato and Y. Toyama, editors, *Third Fuji International Symp. on Functional and Logic Programming (FLOPS'98)* (1998), pp. 98–122.

[8] Jones, R. E., *Tail recursion without space leaks*, Journal of Functional Programming **2** (1992), pp. 73–79.

[9] Labra Gayo, J. E., J. M. Cueva Lovelle, M. C. Luengo Díez and A. C. del Río, *Specification of logic programming languages from reusable semantic building blocks*, in: *Proc. International Workshop on Functional and (Constraint) Logic Programming*, 2001.

[10] Landin, P., *The mechanical evaluation of expressions*, Computer J. **6** (1964), pp. 308–320.

[11] Launchbury, J., *A natural semantics for lazy evaluation*, in: *Proc. POPL '93, 20th Annual ACM Symp. on Principles of Programming Languages*, 1993, pp. 144–154.

[12] Liang, S., P. Hudak and M. Jones, *Monad transformers and modular interpreters*, in: *Proc. 22nd ACM Symp. on Principles of Programming Languages*, 1995, pp. 333–343.

[13] McCarthy, J., *A basis for a mathematical theory of computations*, in: P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, North-Holland, 1963 pp. 33–70.

[14] Okasaki, C., "Purely Functional Data Structures," Cambridge University Press, 1998.

[15] Papaspyrou, N. S., *A resumption monad transformer and its applications in the semantics of concurrency*, in: *Proc. 3rd Panhellenic Logic Symposium*, 2001.

[16] Schmidt, D., "Denotational Semantics: A Methodology for Language Development," Allyn and Bacon, 1986.

[17] Seres, S., M. Spivey and T. Hoare, *Algebra of logic programming*, in: *Proc. International Conference on Logic Programming (ICLP'99)*, 1999, pp. 184–199.

[18] Sestoft, P., *Deriving a lazy abstract machine*, Journal of Functional Programming **7** (1997), pp. 231–264.

[19] Wadler, P., *The essence of functional programming*, in: *Proc. POPL'92, Nineteenth Annual ACM Symp. on Principles of Programming Languages*, 1992, pp. 1–14.