Encapsulating Non-Determinism in Functional Logic Computations*

Bernd Braßel Michael Hanus Frank Huch†

December 20, 2004

Abstract

One of the key features of the integration of functional and logic languages is the access to non-deterministic computations from the functional part of the program. In order to ensure the determinism of top-level computations in a functional logic program, which is usually a monadic sequence of I/O operations, one has to encapsulate the non-determinism (i.e., search for solutions) occurring in logic computations. However, an appropriate approach to encapsulation can be quite subtle if subexpressions are shared, as in lazy evaluation strategies. In this paper we examine the current approaches to encapsulate non-deterministic computations for the declarative multi-paradigm language Curry, show their relative advantages and the problems they induce. Furthermore, we present a new approach which combines the advantages but avoids the problems. Our proposal is based on providing a primitive I/O action for encapsulation from which various specialized search operators can be derived. In order to provide a formal foundation for this new approach to encapsulation, we define the operational semantics of this new primitive.

1 Why Encapsulate and How (Not) To

Functional logic languages are intended to integrate the best features provided in functional and logic languages (see [7] for a survey). They also form a base to improve the evaluation strategies of existing languages due to the existence of optimal evaluation strategies for functional logic languages [3]. However, there is one subtle problem when combining the worlds of functional and logic programming. Usually, the top-level of a realistic functional (logic) program is a monadic sequence of I/O operations that should be applied to the outside world (e.g., see [23]). Since the outside world cannot be copied, all non-determinism in logic computations must be encapsulated, as proposed in [11] for the declarative

*This work has been partially supported by the DFG under grants Ha 2457/1-2 and Ha 2457/5-1.

†Institut für Informatik, Christian-Albrechts-Universität zu Kiel, D-24098 Kiel, Germany. {bbr,mh,fhu}@informatik.uni-kiel.de

multi-paradigm language Curry. Modern functional logic languages are based on demand-driven evaluation strategies [3, 8] which require the sharing of common subexpressions. This can cause strange behavior if some of these shared subexpressions occur within encapsulation operators. This problem will be discussed in the following. We assume familiarity with functional logic programming in general [7], the language Curry [8, 13], and its operational semantics as specified in [1].

1.1 Problems of Combining Sharing and Encapsulation

As the connection between sharing and encapsulation is central to this article, we provide a small series of examples with increasing complexity. The function "`coin`" will play the role of the archetype of all non-determinism. It is defined as

```
coin = 0
coin = 1
```

To give a first impression of the complications of non-determinism when sharing is added, regard the following two functions:

Example 1.1 (Different Values in the Presence of Sharing)

```
withoutSharing = coin + coin
withSharing    = let x = coin in x+x
```

◇

According to the meaning of non-deterministic functions [6] or the operational semantics of Curry [1], the two functions should show different behavior. Evaluating "`withoutSharing`" will compute one of the four possible combinations of `0` and `1`, yielding `0`, `1`, `1`, or `2`. In contrast, a call to "`withSharing`" will only reduce to one of the two values `0` or `2`. The reason for this can be seen when looking at the reduction for both function calls, where non-deterministic choices will be denoted by putting | between them:

```
withoutSharing  → coin + coin → 0 + coin | 1 + coin
                → 0 + 0 | 0 + 1 | 1 + 0 | 1 + 1 → 0 | 1 | 1 | 2
```

This reduction can be seen as a reduction on terms. In contrast, a function employing sharing is a reduction on directed graphs:

```
withSharing → let x=coin in x+x →  •+•  →  •+•  |  •+•  → 0 | 2
                                    \/      \/      \/
                                   coin     0       1
```

One of the key properties of integrating functional and logic languages is to provide access to non-deterministic computations from the purely functional part of the program. To do this, a particular primitive is needed, which takes an arbitrary expression and yields all possible values of this expression in a

single data structure, e.g., a list. We will call such a function `getAllValues` and, corresponding to Example 1.1, it should show the following behavior: `getAllValues withoutSharing` should evaluate to `[0,1,1,2]` whereas the call to `getAllValues withSharing` should lead to `[0,2]`.

Such a search primitive has been proposed in [11] and is contained in the definition of Curry [13]. However, the formal definition of this search primitive in [11, 13] is based on a term rewriting semantics and does not cover the behavior when some subexpressions are shared. As we will see, there are different possibilities how to deal with sharing in the context of encapsulated search. The purpose of this paper is (1) to clarify these differences and the respective advantages and problems of the different approaches, and (2) to propose a practically useful alternative which comprises the advantages but avoids the problems.

Introducing `getAllValues`, we immediately see a problem for defining a search primitive. If a function like `getAllValues` really evaluates to a *list* of possible results, the actual sequence of this list depends on the search strategy. From a declarative point of view, both sequences `[0,1]` and `[1,0]` are legitimate results of `getAllValues coin`. From this perspective it seems mandatory that `getAllValues` should return a set rather than a list of results. Later on we will argue, however, why in our approach `getAllValues` does indeed return an ordered structure. We will continue to assume a list as the result of `getAllValues` until coming back to this point later on.

Based on the previous examples, we can now consider applications of the search operator `getAllValues` for which the expected result is less clear:

Example 1.2 (Difference Between Strong and Weak Encapsulation)

```
coinList = getAllValues coin ++ getAllValues coin
```

Judging from the discussion up to now, it is reasonable that the result of a call to `coinList` should yield the list `[0,1,0,1]`. However, what happens if we identify both calls to `coin` via sharing?

```
coinListWithSharing = let x=coin in getAllValues x ++ getAllValues x
```

Since the behavior w.r.t. sharing was left open in [11, 13], there are at least two possibilities which can be found in different implementations of Curry. The first possibility will be called *strong encapsulation* in the following. The strong encapsulation view is driven by the idea that the search operator should definitely encapsulate all non-determinism. Consequently, if the expression that is encapsulated contains a subexpression that is connected (via sharing) to some expression outside the encapsulation, this connection is cut off. Conceptually, this means that encapsulated search creates *a copy* of the expression before starting its evaluation. In this case, `coinListWithSharing` evaluates to `[0,1,0,1]`. One can argue that this is reasonable due to the correspondence to `coinList` and the property that sharing should have no effect on the computed results ("referential transparency"). Actually, the PAKCS implementation of Curry [10] does indeed feature this behavior.[1]

[1] In PAKCS `getAllValues` can be defined as `getAllValues x = findall (=:= x)`.

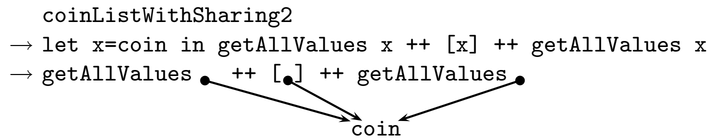
However, there is another view which we call *weak encapsulation.* This view is based on the idea that sharing should be respected even inside encapsulated search. When processing an expression shared with the outside of the encapsulation, the evaluation of this expression will *not* be encapsulated. If this expression generates non-determinism, this non-determinism also effects the computation outside which shares the expression. Thus, for `coinListWithSharing` we obtain non-deterministically one of the values `[0,0]` and `[1,1]`. Actually, the encapsulated search implemented in the Münster Curry Compiler (MCC [18]) features weak encapsulation.[2] ◇

In order to explain the complications with the combination of sharing and encapsulated search in more detail, we consider a slight modification of Example 1.2:

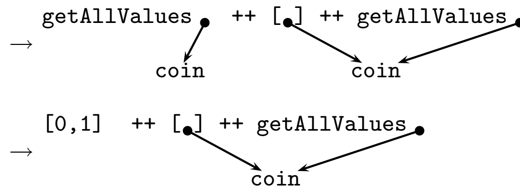
Example 1.3 (Problems of Strong Encapsulation)

```
coinListWithSharing2 = let x=coin in
                           getAllValues x ++ [x] ++ getAllValues x
```

Compared to Example 1.2, one might expect that the results of a call to `coinListWithSharing2` should yield `[0,1]++[0]++[0,1] | [0,1]++[1]++[0,1]` for strong encapsulation. However, this is not the case, as we will see by examining the reduction on the corresponding directed graphs step by step:



Since strong encapsulation duplicates subexpressions which are shared with the outside before evaluating them, the connection of `coin` inside the first `getAllValues` is cut from the other occurrences (the argument of the second `getAllValues` is still shared since its evaluation has not been initiated):



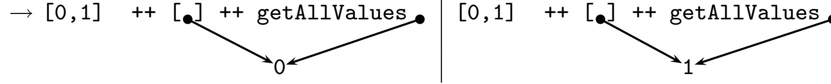
Due to the standard definition of `++` (which demands the evaluation of the left argument first), the list in the middle is evaluated in the next step. As this

[2]For reasons, which will be clarified in the following, `getAllValues` can not be defined in MCC. Instead of `getAllValues x`, we have to use the expression `findall (\y -> y =:= x)`. The analogon to `coinListWithSharing` is then:

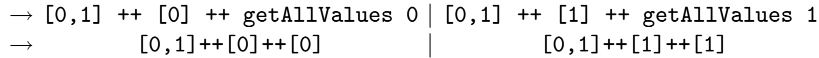
```
coinListWithSharing = let x = coin in
          findall (\y -> y =:= x) ++ findall (\y -> y =:= x)
```

which yields the values `[0,0]` and `[1,1]`.

list contains (a reference to) a call to `coin`, this step yields a non-deterministic branching:



Finally, the second call to `getAllValues` is evaluated, again cutting off the sharing connection.



It is obvious that this example is problematic from a declarative point of view. Thinking in equations, both calls to `getAllValues x` in the example should evaluate to the same result. Even worse, the result of `coinListWithSharing2` would yet be different if the expression was evaluated right-to-left rather than left-to-right. All of this shows that the strong encapsulation view does not provide a declarative functional access to non-determinism. ◇

After studying Example 1.3, it becomes clearer that there are good reasons to avoid the encapsulation of non-determinism when sharing is involved, like in the weak encapsulation view. Unfortunately, this approach is just as unsatisfying as strong encapsulation from a declarative point of view.

Example 1.4 (Problems of Weak Encapsulation)
In the weak encapsulation view, the expressions

```
findall (\y -> y =:= coin)
```

and

```
let x = coin in findall (\y -> y =:= x)
```

are not equivalent. The first yields `[0,1]`, whereas the second evaluates to `[0] | [1]`. Moreover, the encapsulation `findall (\y -> y =:= coin)` is different from `findall (=:= coin)`: again the first equals `[0,1]`, the second results in `[0] | [1]`.

Because of all this, one cannot define `getAllValues` for MCC at all. Any definition like

```
getAllValues x = findall (\y -> y =:= x)
```

results in the non-determinism being not encapsulated. ◇

1.2 Problems of Encapsulating Logical Variables

Functional logic languages allow the evaluation of function calls with logical variables as arguments. These variables will be bound non-deterministically to values corresponding to the patterns for this argument. This process is called "narrowing". Similarly to `coin`, we will use an archetype of a function inducing non-determinism by binding logical variables:

Example 1.5 (Binding Logical Variables in Curry)

```
coinBind 0 = 41
coinBind 1 = 42

detBind 0 = 43
```

A call to `coinBind` with a logical variable `x` results in the two non-deterministic alternatives `41` and `42` with `x` bound to `0` and `1`, respectively. If `coinBind` is called with a ground value, it applies normal pattern matching on this value. If the function `detBind` is called with a logical variable, it also binds this variable, in this case to the value `0`. This binding, however, does not induce non-determinism, as there is only a single pattern. ◇

Encapsulation in the presence of logical variables features some parallels to the sharing problem. Again, we can distinguish a strong encapsulation view from a weak one, and, as we will see, there is also a third view which will be called *rigid encapsulation*. The reference example for this section is

Example 1.6 (Encapsulation of Logical Variables)

```
main =  getAllValues (coinBind x)
     ++ [detBind x]
     ++ getAllValues (coinBind x)
  where x free
```

◇

1.2.1 Strong Encapsulation of Logical Variables

If we take the view of strong encapsulation, Example 1.6 should evaluate (with a left-to-right strategy) to `[41,42]++[43]++[41]`. This is because the sharing connection to the outside of `getAllValues` is cut off and the bindings to `x` are not visible on the top level, i.e., they are encapsulated. Strong encapsulation of logical variables is performed by Prolog's `findall` (see [19] for a detailed discussion). It is obvious that this behavior is just as problematic as the one discussed in Example 1.3.

Example 1.7 (Problems of Strongly Encapsulating Logical Variables)

The behavior of the following Prolog program has similarities with Example 1.3.

```
coinBind(0).
coinBind(1).

detBind(0).

t1(L) :- findall(X,coinBind(X),L),findall(X,coinBind(X),L),detBind(X).
t2(L) :- detBind(X),findall(X,coinBind(X),L),findall(X,coinBind(X),L).
t3(L) :- findall(X,coinBind(X),L),detBind(X),findall(X,coinBind(X),L).
```

The proof of the literal `t1(L)` succeeds with `L=[0,1]` and `t2(L)` succeeds with `L=[0]` whereas the proof of `t3(L)` fails because `L` is first bound to `[0,1]` and then to `[0]`, according to the left-to-right semantics of Prolog. ◇

Strong encapsulation of logical variables also interacts notably with lazy evaluation, as we will show in Section 1.4 below.

1.2.2 Weak Encapsulation of Logical Variables

What is the result of Example 1.6 in the view of weak encapsulation? In analogy to Section 1.1, we could state that `getAllValues (coinBind x)` can not encapsulate the non-determinism occurring in `(coinBind x)` because `x` is shared with the outside of the capsule. Rather, the computation splits into two non-deterministic branches. In both branches `x` is bound to a value (`0` or `1`) and, consequently, the call `(detBind x)` fails in one branch due to the unification of the instantiated variable `x` and `0`. Thus, `main` in Example 1.6 is equivalent to `[41]++[43]++[41]`.

Example 1.8 (Weak Encapsulation in Prolog) Prolog does also feature an equivalent to weak encapsulation which is called `bagof`.

```
t4(X) :- bagof(_,coinBind(X),_),bagof(_,coinBind(X),_),detBind(X).
t5(X) :- detBind(X),bagof(_,coinBind(X),_),bagof(_,coinBind(X),_).
t6(X) :- bagof(_,coinBind(X),_),detBind(X),bagof(_,coinBind(X),_).
t7(X) :- bagof(_,coinBind(X),_).
```

The proofs of all three literals `t4(X)`, `t5(X)`, and `t6(X)` succeeds with `X=0`. However, the non-determinism of `t7(X)` is not encapsulated, resulting in two solutions, `X=0` and `X=1`.[3] ◇

1.2.3 Rigid Encapsulation of Logical Variables

The reason, why Examples 1.7 and 1.8 were given in Prolog, is that none of the implementations of Curry takes one of these views of strong or weak encapsulation for logical variables. Curry implementations feature a third possibility between strong and weak encapsulation (which is also used in Oz [22]): *rigid* encapsulation.

In the view of rigid encapsulation, the whole encapsulation suspends if a logical variable declared outside the encapsulation should be bound inside the encapsulation. The suspended encapsulation can later be resumed if the variable gets bound by a concurrent computation. In this view, the evaluation of Example 1.6 would suspend. The results of rigid and weak encapsulation differ noteworthy as shown in the next example:

Example 1.9 (Rigid vs. Weak Encapsulation)

[3] It is also possible to mix the two kinds of encapsulation by attaching an existential quantifier to arguments of `bagof`. For a detailed discussion of `bagof` and the related predicate `setof` see [19]

```
main | getAllValues (coinBind x) =:= y & x=:=2 = y where x,y free
```

In rigid encapsulation, a call to `main` results in `[]` (regardless of the order in which the two concurrent constraints are evaluated), whereas weak encapsulation produces no solution. ◇

All current implementations of Curry which feature encapsulated search take a variation of the rigid encapsulation view. However, rigid encapsulation has also its problems:

Example 1.10 (Problems of Rigid Encapsulation)

```
twoBindings 43 43 = 46
twoBindings 44 43 = 46

main = twoBindings (detBind x) (head (getAllValues (detBind x)))
  where x free
```

Depending on the order in which the pattern matching for function `twoBindings` is executed, the call to `main` either suspends (in case of a right-to-left order) or evaluates to `46` (in case of a left-to-right order). Although all current implementations of Curry evaluate the given pattern left-to-right, other evaluation order would make sense for the sake of efficiency improvements [15]. ◇

From a semantical point of view a possible suspension like the one in Example 1.10 is not problematic. `main` denotes `46` under the condition that `x` is bound to `0`. The operational analogy is that the variable `x` *could* be bound by a concurrent computation, thereby waking up the suspended computation. However, a realistic program is either meant to run concurrently or it is not. If it is *not meant* to, a suspension is nothing but a run-time error. Considering Example 1.10, such a run-time error might only occur when porting the program from one implementation of Curry to another, as for instance the order of pattern matching might differ.

As we will show in Section 1.4, rigid encapsulation produces more problems when combined with laziness.

Looking at current implementations featuring rigid encapsulation, there are notable differences. In PAKCS, the computation is suspended directly when applying the search operator to an expression containing a logical variable declared outside. In MCC, the computation is only suspended when such a variable is to be bound.

Example 1.11 (A Superfluous Variable)

```
f x y = x
main = findall (\y -> y =:= f 1 x) where x free
```

The call to `main` suspends using PAKCS, whereas it reduces to `[1]` using MCC. ◇

As a less academic example one can think of a web service. In Curry such a service communicates to the outer world via ports, i.e., synchronizing by logical variables [9]. In this case, the difference between rigid encapsulation in MCC and PAKCS is that in MCC the web service may perform all sorts of start up routines before suspending on the port, whereas in PAKCS this initialization only takes place when a message comes in.

1.3 Lazy Evaluation and Search Strategies

Another notable feature of current implementations of encapsulated search is the possibility to evaluate a call to `getAllValues` lazily. The problems of eager evaluation are well known and we provide only a small example.

Example 1.12 (Eager vs. Lazy Encapsulated Search)

```
zeros = 0
zeros = zeros

main = head (getAllValues zeros)
```

Evaluating the search space eagerly, like in PAKCS, the call to `main` does not terminate. If the search space is lazily evaluated, like in MCC, `main` reduces to `0`. ◇

Of course, termination is not only influenced by lazy or eager evaluation but also by the search strategy employed. Current implementations of Curry employ depth-first search which might be influenced by the textual order of the rules. In consequence, a slight change to Example 1.12 leads to a different run-time behavior:

Example 1.13 (Depth-First Search)

```
zeros2 = zeros2
zeros2 = 0

main = head (getAllValues zeros2)
```

This program does not terminate w.r.t. eager as well as lazy evaluation of the search space provided that the rules are tried from top to bottom. ◇

However, there is the possibility to formulate the search using a primitive search operator `try` [11]. Using `try`, one can formulate search strategies different from standard depth-first search. We will discuss a similar solution in Section 2.1.

1.4 Sharing, Laziness and Logical Variables Interacting

Generally, if encapsulated search operators like `getAllValues` are evaluated lazily, then the problems of encapsulating logical variables increase. This is due to the fact that the results of a call to `getAllValues` might depend on logical variables declared outside of the capsule. We illustrate this problem by a further example.

Example 1.14 (Influencing the Search Result)

```
main i = bindHead i (getAllValues (x ?⁴ coinBind x)) where x free

bindHead i (y:ys) | y=:=i = ys
```

The interesting point of this example is that the function `bindHead` accesses the result of the search. Unifying *the first* solution with the given value `i` influences the value of *the remaining* solutions. Thus, binding the first solution to `0` leads to the remaining solutions `[41]`, and, when binding it to `1`, the remaining solutions are `[42]` (e.g., MCC shows this behavior). Moreover, a redefinition of the function `coinBind` can also lead to a difference in the number of results.

```
coinBind 0 = 41
coinBind 1 = 42
coinBind 1 = 43
```

With this `coinBind`, `main 0` reduces to `[41]` and `main 1` to `[42,43]`. ◇

On one hand, this behavior of lazy encapsulation opens the door to new concepts of search: one could inspect the first few solutions of a search and, depending on the result of the analysis, one could prune the remaining search. On the other hand, influencing a search in this way is problematic: the value of a call to `getAllValues` is not definite until it is fully evaluated. Its value depends on the whole computation before *and after* the call to `getAllValues`. It might be difficult to understand the outcome of a program if the results depends on all these factors. Therefore, it seems reasonable to provide a search primitive whose value is fully defined by the time it is called and does not depend at all on the remaining computation. Influencing the outcome of the search during its evaluation could be the purpose of specific pruning operators. However, this will not be considered in this paper.

1.5 Wish List for Future Implementations of Encapsulated Search

From the discussion above, we can now gather the desirable features of encapsulated search.

Strong Encapsulation of Sharing: One of the main reasons to provide encapsulated search is to make sure that certain parts of the program evaluation are definitely deterministic. This is especially important for I/O actions. Since weak encapsulation does not ensure encapsulation of non-determinism (see Examples 1.2 and 1.4), some variant of strong encapsulation has to be used to achieve deterministic I/O.

No Sharing of Non-Deterministic Terms with the Outside: Many of the above examples have shown that strong encapsulation can be problematic.

[4]The function `?` non-deterministically yields one of its arguments. Thus, `coin` could also be defined as `coin = 0 ? 1`.

Examining Examples 1.3, 1.6, and 1.7 we can conclude that the problem is *sharing a non-deterministic value* with the outside of the capsule. Following the old directive of language design "If you don't like it, you don't allow it" [16, p.101], future implementations of encapsulated search should omit these problems. This can be achieved by a combination of two program analyses: a non-determinism analysis like the one presented in [12, 5] and a sharing analysis like those employed for uniqueness types [4]. Some work has to be invested to clearly define "the outside":

Example 1.15 (Defining the Outside)

```
f = let x = coin in getAllValues x ++ getAllValues x
g = getAllValues (let x = coin in x ++ getAllValues x)
```

In this example the definition of `f` would not lead to problematic behavior of the encapsulated search in contrast to the definition of `g`. Consequently, a good program analysis could allow `f` but reject `g`. Note that `f` illustrates that even if we omit sharing of non-deterministic terms with the outside, there is still a difference between strong and weak encapsulation. ◇

Of course, there are various degrees of how well the program analysis approximates the information about sharing and non-determinism. For instance, uniqueness is often ensured by a monadic approach, and Curry supports the I/O monad as a construct to ensure uniqueness. Therefore, in the following we will discuss an approach to restrict encapsulated search to the I/O monad. This approach is a valid approximation of the requirement to prevent sharing of non-deterministic terms with the outside for several reasons. First, monadic I/O has to be deterministic. Therefore, no sharing of non-deterministic terms with the outside of encapsulated search can take place. Second, different calls to a search operator like `getAllValues` have to be explicitly sequentialized in the I/O monad. This will be important in one of the next points of our wish-list. Third and last, as the result of encapsulated search is a list rather than a set, its result, i.e., the order of the solutions, can depend on external factors like scheduling, memory usage etc. Because of that encapsulated search clearly bears resemblance to an I/O action.

Strong Encapsulation for Logical Variables: As discussed in Section 1.2, all implementations of Curry employ rigid encapsulation for logical variables. However, in the context of Example 1.10, we have shown that rigid encapsulation can lead to different behavior on different implementations of Curry. Although semantically correct, this unpredictable behavior is in most situations as undesirable as a run-time error. Therefore, we propose strong encapsulation like in Prolog as the best choice for encapsulating logical variables.[5]

Strong encapsulation also provides more possibilities to influence an ongoing search as detailed in Section 1.4. In that section, we also discussed increasing

[5]Note that it is still possible to formulate a suspending search by the explicit use of rigid functions.

problems when combining strong encapsulation with lazy evaluation. This problems will have to be addressed in one of the remaining points of the wishlist.

Lazy Evaluation: For a seamless combination of functional and logic programming, a lazy search primitive is desirable. Otherwise, there is no complete correspondence between the treatment of non-determinism on the top level of the interactive environment (where solutions are shown one-by-one) and the handling of non-determinism within the program. Moreover, with eager search the programmer has to think in terms of terminating reductions for the non-deterministic computations to be encapsulated.[6] In addition, lazy evaluation enables the possibility to influence an ongoing search (see Section 1.4). As an example, we will show how to implement depth-first and breadth-first search using this approach.

Explicit Sequencing: As discussed above, it is possible to influence not only the overall strategy but also to prune an ongoing search via a lazy search primitive if we bind logical variables outside the capsule. In this case the result of an encapsulated search does not only depend on the evaluation up to the start of the search but also on the remaining computation. To avoid this problem, the search primitive conceptually has to copy the binding state of the logical variables. In order to allow the programmer to define the starting point of his search, all calls to search primitives should be explicitly sequentialized. In our approach, this will be obtained by restricting the search primitive to the I/O monad.

In the next section, we propose a formal definition of a new approach to encapsulate search in (non-strict) functional logic languages. This approach features all the properties discussed in this wish list.

2 A Revised Approach to Encapsulating Search in Functional Logic Programs

In this section we first sketch our basic design of the functional access to non-deterministic computations (Section 2.1). Section 2.2 contains the formal definition of this design.

2.1 A Data Structure for Representing Search

Our approach is based on the idea that each non-deterministic computation yields a data structure representing the actual search space. The definition of this representation should be independent of the search strategy employed. The basic structure of the search space can be captured by the following algebraic data type:

[6] Note that the encapsulated search primitives of Oz [22] and Prolog's `findall` [19] are related to strict languages where sharing only occurs via logical variables.

```
data SearchTree a = Or [SearchTree a] | Val a | Fail
```

Thus, a non-deterministic computation yields either the successful computation of a completely evaluated term v (i.e., a term without defined functions) represented by `Val` v, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or` $[t_1, \ldots, t_n]$ where $t_1, \ldots, t_n$ are search trees representing the subcomputations.

Example 2.1 (Search Tree)
In order to understand our use of search trees, consider again the function "`coin`" defined as

```
coin = 0
coin = 1
```

Evaluating the expression `coin` with our search primitive will yield the search tree `Or [Val 0, Val 1]` which represents both results in one structure. Similarly, the expression `[coin,coin]` will be evaluated to

```
Or [Or [Val [0,0], Val [0,1]], Or [Val [1,0], Val [1,1]]]
```

by our search primitive. The latter represents the four possible results `[0,0]`, `[0,1]`, `[1,0]`, and `[1,1]` which can be extracted by a particular search strategy (see below). ◇

Analogously to `findall` in MCC, this structure should be provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence, it is possible to define arbitrary search strategies on the structure of search trees. For instance, variations of `getAllValues` for depth-first search and breadth-first search can be defined as follows:

```
getAllValuesD :: SearchTree a -> [a]           -- depth-first search
getAllValuesD (Val v) = [v]
getAllValuesD Fail    = []
getAllValuesD (Or ts) = concatMap getAllValuesD ts

getAllValuesB :: SearchTree a -> [a]           -- breadth-first search
getAllValuesB t = map (\(Val v) -> v) (getAllValuesB' [t])

getAllValuesB' :: [SearchTree a] -> [SearchTree a]
getAllValuesB' [] = []
getAllValuesB' (t:ts) =
    filter isVal (t:ts) ++
    getAllValuesB' (concatMap (\(Or ts) -> ts) (filter isOr (t:ts)))
```

where `isVal` and `isOr` are test predicates for the constructors `Val` and `Or`, respectively. Evaluating the search tree lazily, these functions evaluate the list of all values in a lazy manner too.

2.2 An Operational Semantics for Encapsulated Search

In this section, we present an operational semantics for a search operator, called *getSearchTree*, that handles the problems discussed in Section 1. As motivated above, the search operator evaluates expressions as in standard computations but returns the results in form of search trees. In particular, non-deterministic results should be combined into `Or`-structures. Thus, we base our semantics on the operational semantics for functional logic languages presented in [1]. In the following, we explain how to extend this operational semantics in order to generate appropriate search trees. We will explain our considerations rule by rule together with a discussion of the differences from the original rules in [1].

Instead of defining our semantics directly for Curry, we consider a core sublanguage, called FlatCurry, into which Curry programs can be translated. Local function definitions are eliminated by lambda lifting [14]. Higher-order constructs are translated to primitive functions `partcall` (partial applications) and `apply` (binary applications). Needed narrowing and residuation are implemented as case expressions, which correspond to definitional trees [2], `fcase` for flexible functions, `case` for rigid functions, and `or` for non-deterministic branching. For instance, the function `coinBind` of Example 1.5 corresponds to the FlatCurry definition

```
coinBind x = fcase x of { 0 -> 41;
                          1 -> 42 }
```

the function `coin` of Example 2.1 corresponds to the FlatCurry definition

```
coin = 0 or 1
```

Furthermore, we consider *normalized* FlatCurry programs in which functions are only applied to variables that are bound to expressions by an explicit `let` binding. These variables represent references to possibly shared expressions (see also [17]). The normalized form of an expression e is denoted by e^* (see [1] for a formal definition). For instance, the definition "`main = coinBind coin`" is normalized to

```
main = let x = coin in coinBind x
```

The basic components of the original semantics are

(a) a *heap* (a mapping from variables to expressions) to model sharing,

(b) a *control* which holds the expression currently processed, and

(c) a *stack* which provides two kinds of information: case expressions for pattern matching and variables which will be updated as soon as their corresponding expressions have been evaluated to head normal form.

For our approach we need the possibility to *encapsulate* non-deterministic computations. This is done by considering *a sequence of heaps* rather than a single heap and a *sequence of stacks* rather than a single stack. These sequences are

essentially push-down structures, the topmost is always the one currently processed. A rule which deals with the topmost heap Γ and the topmost element x of the topmost stack S has the following form:

Rulename	*Heaps*	*Control*	*Stacks*
example	$\gamma \cdot \Gamma$	e	$x \cdot S \cdot (S')$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto e]$	e	$S \cdot (S')$

Note that, for the sake of readability, the sequence of heaps grows towards the right and the stacks towards the left. "·" denotes the concatenation on sequences. As each stack in the sequence of stacks is itself modeled by a sequence, we use brackets () to separate the different stacks, whereas no such separation is necessary for the heaps.[7] For the sake of readability we write (S) instead of $\varepsilon \cdot (S)$. Finally, the notation $\Gamma[x \mapsto e]$ denotes a heap in which the variable x maps to the expression e and other mappings of x in Γ are ignored (i.e., it represents a destructive heap update). Logical variables are represented by self-references ($[x \mapsto x]$).

The heap/stack sequences grow whenever a new layer of encapsulation is needed. Thus, there may be as many layers as there are calls to *getSearchTree* in the program plus one. This additional layer is used for the top-level evaluation, in which no search (non-determinism) may be performed. This is important because the top-level is usually deterministic, featuring I/O actions to print the computed values. As discussed in Section 1, this means that any non-determinism has to be encapsulated.

As in the original approach, we assume that the evaluation starts with the designated function `main`. Thus, the initial state of the operational semantics is[8]

Heaps	*Control*	*Stacks*
$[\,]$	*main*	ε

Now we are ready to discuss the different rules applicable on the states of the operational semantics. The first two rules deal with retrieving information from the topmost heap by looking up a variable binding which is put into the control:

Rule	*Heaps*	*Control*	*Stacks*
varcons	$\gamma \cdot \Gamma[x \mapsto c(\overline{x_n})]$	x	S
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto c(\overline{x_n})]$	$c(\overline{x_n})$	S
varexp	$\gamma \cdot \Gamma[x \mapsto e]$	x	S
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto e]$	e	$x \cdot S$
where e is not constructor-rooted and $e \neq x$			

[7] Thus, the difference between $x \cdot y \cdot S$ and $x \cdot (y \cdot S)$ is that in $x \cdot (y \cdot S)$ the variable x is the only element of the topmost stack, whereas in $x \cdot y \cdot S$ both x and y are on the topmost stack.

[8] We will redefine the initial state at the end of this section to make our semantics work. However, for the moment, this definition is sufficient.

In our notation, c denotes a constructor symbol, e represents arbitrary expressions, and over-lined terms $\overline{a_n}$ represent the sequence of terms $a_1, \ldots, a_n$. If the variable to be evaluated is bound to a constructor-rooted term, rule varcons simply writes this term into the control. If it is bound to some unevaluated expression, rule varexp writes this expression into the control and pushes the variable on the stack in order to implement sharing (see rule val below). Apart from the presence of sequences of heaps and stacks, these rules are identical to the original rules of [1]. Similarly, the next three rules, val, fun and let, are identical to the original rules since they affect only the topmost heap:

Rule	*Heaps*	*Control*	*Stacks*
val	$\gamma \cdot \Gamma$	v	$x \cdot S$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto v]$	v	S
fun	γ	$f(\overline{x_n})$	S
$\rightarrowtail\!\!\!\rightarrow$	γ	$\rho(e)$	S
let	$\gamma \cdot \Gamma$	$let\ \{\overline{x_k = e_k}\}\ in\ e$	S
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[\overline{y_k \mapsto \rho(e_k)}]$	$\rho(e)$	S

where in val v is constructor-rooted or a variable with $\Gamma[v] = v$,
in fun $f(\overline{y_n}) = e$ is a program rule and $\rho = \{\overline{y_n \mapsto x_n}\}$,
and in let $\rho = \{\overline{x_k \mapsto y_k}\}$ and $\overline{y_k}$ are fresh w.r.t. all heaps

Rule val implements sharing by updating the heap with a computed value, rule fun unfolds a function call, and rule let puts bindings for fresh variables into the topmost heap.

The next rule, or, is more interesting as it strongly differs from the original:

Rule	*Heaps*	*Control*	*Stacks*
or	$\gamma \cdot \Gamma \cdot \Gamma'$	$e_1\ or\ e_2$	$S \cdot (S')$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[vs \mapsto [z_1, z_2],$ $z_{1/2} \mapsto caps(\Gamma', e_{1/2}, S)]$	$Or(vs)$	S'

where vs, z_1 and z_2 are fresh

Here and in the following $x \mapsto [z_1, \ldots, z_n]$ is a shortcut to denote that x maps to the list of variables $[z_1, \ldots, z_n]$. This is correctly expressed by the somewhat lengthy expression

$$x \mapsto l_0, \overline{l_{n-1} \mapsto z_n : l_n}, l_n \mapsto [\,] \text{ where } l_0, \ldots, l_n \text{ are fresh variables.}$$

In rule or, we lift the idea of lazy evaluation to the meta-level of search. This results in a lazy construction of the search tree, as demanded by the top-level evaluation (e.g., a search strategy like `getAllValuesD`). Whenever a non-deterministic branching is executed, we have to put the branches into a search tree headed by the constructor `Or`. Thus, the current layer of encapsulation has been evaluated to head normal form and the evaluation is finished (for now). The result $Or(vs)$ is put on the control and the current layer of encapsulation can be removed. The current context, i.e., the topmost heap, the expression on

the control, and the topmost stack, is stored for future reference in a specific structure (*caps*) in the heap of the next level of evaluation.

Whenever one of the arguments of `Or` is demanded, the context has to be restored. This is done by the rule caps.

Rule	*Heaps*	*Control*	*Stacks*
caps	γ	$caps(\Gamma, e, S)$	S'
$\rightarrowtail\!\!\!\twoheadrightarrow$	$\gamma \cdot \Gamma$	e	$S \cdot (S')$

It can be easily checked that restoring the evaluation context with caps is indeed dual to storing it with or.

The next two rules are concerned with the implementation of pattern matching. They are almost identical to the original ones presented in [1].

Rule	*Heaps*	*Control*	*Stacks*
case	γ	$(f)case\ e\ of\ \{\overline{p_k \to e_k}\}$	S
$\rightarrowtail\!\!\!\twoheadrightarrow$	γ	e	$(f)\{\overline{p_k \to e_k}\} \cdot S$
select	γ	$c(\overline{y_n})$	$(f)\{\overline{p_k \to e_k}\} \cdot S$
$\rightarrowtail\!\!\!\twoheadrightarrow$	γ	$\rho(e_i)$	S
where $p_i = c(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$			

The two variations `case` and `fcase` (short for "flexible case") correspond to the evaluation modes "rigid" and "flexible" as described in [13]. The difference will become important when discussing rule guess below.

Before discussing the remaining rules, we give an example of a simple program applying only those rules discussed so far.

Example 2.2 (Operational Semantics in Action)
Consider the following simple program:

```
coin = 0 or 1
consTree = let c = coin in
             case c of { 0 -> c;
                         1 -> c }
```

Assume that the evaluation of `consTree` is demanded during some computation. Using our operational semantics, we obtain:

	$\Gamma_1 \cdot [\,]$	`consTree`	(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{fun}}$	$\Gamma_1 \cdot [\,]$	`let c=coin in case {0->c;1->c}`	(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{let}}$	$\Gamma_1 \cdot [$`c` $\mapsto$ `coin`$]$	`case c of {0->c;1->c}`	(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{case}}$	$\Gamma_1 \cdot [$`c` $\mapsto$ `coin`$]$	`c`	`{0->c;1->c}`(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{varexp}}$	$\Gamma_1 \cdot [$`c` $\mapsto$ `coin`$]$	`coin`	`c{0->c;1->c}`(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{fun}}$	$\Gamma_1 \cdot [$`c` $\mapsto$ `coin`$]$	`0 or 1`	`c{0->c;1->c}`(S)
$\rightarrowtail\!\!\!\twoheadrightarrow_{\mathsf{or}}$	Γ'_1	`Or vs`	(S)

where $\Gamma'_1 = \Gamma_1[$`vs` $\mapsto$ `z1:vs1`, `vs1` $\mapsto$ `z2:vs2`, `vs2` $\mapsto$ `[]`,
`z1` $\mapsto$ `caps([c` $\mapsto$ `coin],0,c{0->c;1->c})`,
`z2` $\mapsto$ `caps([c` $\mapsto$ `coin],1,c{0->c;1->c})`$]$

The result is `Or` with a list argument. The two elements of this list are capsules containing the arguments of the `Or` and the heap and stack in which their evaluation should be continued.

If we later resume the evaluation of the second argument of the search tree, we get the evaluation (where Γ_2 is similar to Γ'_1 possibly containing some further updates):

	Γ_2	`z2`	S
$\rightarrowtail\!\!\!\rightarrow_{\mathsf{varexp}}$	Γ_2	`caps([c` $\mapsto$ `coin],1, c{0->c;1->c})`	`z2` S
$\rightarrowtail\!\!\!\rightarrow_{caps}$	$\Gamma_2 \cdot$ `[c` $\mapsto$ `coin]`	`1`	`c{0->c;1->c}(z2` S`)`
$\rightarrowtail\!\!\!\rightarrow_{\mathsf{val}}$	$\Gamma_2 \cdot$ `[c` $\mapsto$ `1]`	`1`	`{0->c;1->c}(z2` S`)`
$\rightarrowtail\!\!\!\rightarrow_{\mathsf{select}}$	$\Gamma_2 \cdot$ `[c` $\mapsto$ `1]`	`c`	`(z2` S`)`
$\rightarrowtail\!\!\!\rightarrow_{\mathsf{varcons}}$	$\Gamma_2 \cdot$ `[c` $\mapsto$ `1]`	`1`	`(z2` S`)`

$\diamond$

The next rule, guess, is responsible for starting those non-deterministic searches which are induced by guessing bindings for logical variables (narrowing). Consider a flexible function f. When f is called with a logical variable x as an argument, pattern matching on this variable results in a non-deterministic branching. In each branch, x is bound to a different constructor corresponding to the patterns of the `fcase`. This behavior is modeled by the rule guess which combines the rules select and or discussed above:

Rule	*Heaps*	*Control*	*Stacks*
guess	$\gamma \cdot \Gamma \cdot \Gamma'[x \mapsto x]$	x	$f\left\{\overline{c_k(\overline{x_{n_k}}) \rightarrow e_k}\right\} \cdot S \cdot (S')$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[vs \mapsto [z_1, \ldots, z_k], \overline{z_k \mapsto s_k}]$	$Or(vs)$	S'

where $vs, z_1, \ldots, z_k$ are fresh, $k > 1$, and
for all $i \in \{1, \ldots, k\} : \rho_i = \{\overline{x_{n_i} \mapsto y_{n_i}}\}, \overline{y_{n_i}}$ are fresh, and
$s_i = caps(\Gamma'[x \mapsto c_i(\overline{y_{n_i}}), \overline{y_{n_i} \mapsto y_{n_i}}], \rho_i(e_i), S)$

Note that rule guess only induces non-determinism if there is more than one pattern in the case expression. If there is only a single pattern, there is no need to perform a non-deterministic branching. Therefore, the corresponding rule guess-1 is quite simple and more similar to select than to guess:

Rule	*Heaps*	*Control*	*Stacks*
guess-1	$\gamma \cdot \Gamma[x \mapsto x]$	x	$f\{c(\overline{x_n}) \rightarrow e\} \cdot S$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto c(\overline{y_n}), \overline{y_n \mapsto y_n}]$	$\rho(e)$	S

where $\overline{y_n}$ are fresh and $\rho = \{\overline{x_n \mapsto y_n}\}$

The next three rules deal with the results of encapsulations. If the computation was successful (i.e., the remaining stack is empty), the result (a head normal form) remains on the control. The context of the current computation has to

be stored (similar to rule or) in order to access the values of the arguments correctly.

Rule	Heaps	Control	Stacks
rescons	$\gamma \cdot \Gamma \cdot \Gamma'$	$c(\overline{x_n})$	(S)
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma \left[\overline{z_n \mapsto caps(\Gamma', x_n, \varepsilon)}\right]$	$c(\overline{z_n})$	S
resvar	$\gamma \cdot \Gamma \cdot \Gamma'[x \mapsto x]$	x	(S)
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto x]$	x	S

where $\overline{z_n}$ are fresh in rescons

Failing computations are encoded by the constructor *Fail*. Obviously, the evaluation context of such a computation can be thrown away.

Rule	Heaps	Control	Stacks
fail	$\gamma \cdot \Gamma$	$c(\overline{x_n})$	$(f)\{\overline{p_k \rightarrow e_k}\} \cdot S \cdot (S')$
$\rightarrowtail\!\!\!\rightarrow$	γ	*Fail*	S'

where for all $i = 1, \ldots, k: \; p_i \neq c(\ldots)$

So far, we have extended the base semantics of [1] to compute search trees for representing non-deterministic computations. We have shown in Section 2.1 how search strategies can be defined as functions processing search trees. In order to provide this feature inside a program, the computed search trees must be made available to the programmer. As motivated in Section 1.5, a search tree should be computed by an action inside the I/O monad. Thus, we propose a predefined I/O action *getSearchTree* that lazily evaluates its argument to a search tree. Its semantics is defined by the following rule (since *getSearchTree* is an I/O action, it has a second argument, the current state of "world" that is threaded through the sequence of I/O actions, as described in [21]):

Rule	Heaps	Control	Stacks
getsearchtree	Γ	$getSearchTree(x, w)$	S
$\rightarrowtail\!\!\!\rightarrow$	$\Gamma[z \mapsto caps(\Gamma, search(x, \, Val(x))^*, \varepsilon)]$	(z, w)	S

where z is fresh

Note that *getSearchTree* does not evaluate its argument but stores it in a capsule in the heap which will be further evaluated if it is demanded by some search strategy, like `getAllValuesD`. The heap stored in this capsule is the heap before this primitive is evaluated. This corresponds to strong encapsulation of sharing since the actual encapsulated search is performed on a copy of the current argument. Furthermore, the argument is wrapped in a *search* structure (which must be normalized by means of *, see above). This is necessary in order to deliver only completely evaluated values as arguments of *Val*. Otherwise, the further evaluation of such arguments could result in non-deterministic computations. Thus, a *search* structure is evaluated by computing the normal form of the first argument and returning the second argument. This behavior is specified by the

following rules:

Rule	*Heaps*	*Control*	*Stacks*
search	γ	$search(x_1, x_2)$	S
$\rightarrowtail\!\!\!\rightarrow$	γ	x_1	$s(x_2) \cdot S$
searchcons	γ	$c(\overline{x_n})$	$s(x) \cdot S$
$\rightarrowtail\!\!\!\rightarrow$	γ	$search(x_1, search(x_2, .., search(x_n, x)..))^*$	S
searchvar	$\gamma \cdot \Gamma[x \mapsto x]$	x	$s(y) \cdot S$
$\rightarrowtail\!\!\!\rightarrow$	$\gamma \cdot \Gamma[x \mapsto z, z \mapsto z]$	y	S

Rule search initiates the evaluation of a *search* structure by putting the second argument on the stack and the first argument on the control. If the control is evaluated to a constructor-rooted term, the arguments must be evaluated as well before returning its value. This is the purpose of rule searchcons which creates *search* structures for the arguments. Similarly to rule getSearchTree, we have to ensure the invariant of our semantics that functions and constructors are only applied to variables (in order to implement sharing, as mentioned above). Thus, the nested *search* expression has to be normalized by means of the normalization function * which yields in this case:

$$
\begin{array}{l}
let\ \{\ y_n = search(x_n, x), \\
\quad\quad y_{n-1} = search(x_{n-1}, y_n), \\
\quad\quad\quad\quad \vdots \\
\quad\quad y_2 = search(x_2, y_3)\ \}\ in \\
\quad search(x_1, y_2)
\end{array}
$$

This definition fixes the evaluation order from left to right. However, this is no restriction for narrowing since we compute normal forms of expressions to be searched[9].

Rule searchvar finishes the search (as well as rule searchcons in case of 0-ary constructors) by removing the search structure if a logical variable has been computed. In order to implement strong encapsulation of logical variables (cf. Section 1.2), rule searchvar renames the computed variable.

Finally, we encapsulate all non-determinism in the top-level by wrapping the *main* I/O action that is applied to the outside world (represented by the constructor *World*()) in a search structure. Thus, the main evaluation of a program is always initiated by the following rule[10]:

Heaps	*Control*	*Stacks*
$[\,]$	$let\ m = main,$ $\quad w = World(),$ $\quad mIO = apply(m, w)\ in$ $search(mIO, mIO)$	ε

[9] In the context of residuation, the evaluation order can have an effect on the result. While one order suspends, another order can yield a result. However, this discussion is out of the scope of this paper.

[10] Note that, by applying the search primitive, the top-level computation yields the complete normal form, whereas the original semantics computed the head normal form, only.

The remaining rules of the original operational semantics [1], like the rules for primitive functions, higher-order applications (*apply*), unification (`=:=`), are not changed in our context so that we do not present them here again.

3 Examples

In the following example, we demonstrate the effect of using the search operator `getSearchTree` twice which is in some sense nested because of lazy evaluation. First, the program computes the search tree resulting from `coin`. This tree is then used in the function `getAny` to select all values in this tree non-deterministically. This non-deterministic computation is again encapsulated by `getSearchTree` and both encapsulated results are compared.

```
coin :: Int
coin = 0 or 1

(==) :: a -> a -> Bool
x == y = let a = prim_Eq(x,y),
             b = hnf(y,a)
          in hnf(x,b)

main :: IO Bool
main = let c = coin in do
          cST <- getSearchTree c
          cST' <- getSearchTree (getAny cST)
          return (cST == cST')

getAny :: SearchTree a -> a
getAny t = case t of {
             Or ts -> getAnyL ts;
             Val v -> v}

getAnyL :: [SearchTree a] -> a
getAnyL ts = case ts of {
               (x:xs) -> (getAny x) or (getAnyL xs)}
```

To compare the two results, we use the (predefined) function "`==`". It evaluates its arguments to head normal form (`hnf`) and compares them by `prim_Eq` afterwards. The function `prim_Eq` is a primitive function which, for identical top-level constructors, compares their arguments recursively by means of `(==)` and sequential conjunction (`&&`). See [1] for a more detailed description of the semantics of primitive functions.

The computation yields the value `Val False`. This (perhaps) surprising result is produced due to different searches performed for the computations of `cST` and `cST'`. A full evaluation of the search tree would yield `cST = Or [Val 0, Val 1]`. Applying the function `any` to this value and encapsulating it again by the operator `getSearchTree`, we obtain `cST' = Or [Val 0,Or [Val 1,Fail]]`. The

presented program compares the structure of the search trees exactly. However, both compute the same values and a comparison of `getAllValuesD` or `getAllValuesB` applied to `cST` and `cST'` would return `True`.

Since the execution of the program takes 266 steps, we present a simpler computation in Appendix A. Instead of comparing `cST` and `cST'`, we only compute the first solutions for `cST` and `cST'`. Despite of this restriction, the example still shows how the evaluation of one encapsulation demands the evaluation of another encapsulation.

Another example, which shows the benefit of computing the search tree lazily, is the non-deterministic computation of all numbers:

```
nat :: Int
nat n = nat (n+1) or n

nats :: IO (SearchTree Int)
nats = getSearchTree (nat 0)
```

The computation of `getSearchTree` yields the infinite search tree

```
Or [Or [Or [Or [...,Val 3], Val 2], Val 1], Val 0]
```

Since this tree is infinitely nested in the first argument of the occurring `Or`s, the application of `getAllValuesD` will not compute a solution since it does not terminate. Using `getAllValuesB` we can compute an arbitrary number of solutions by

```
nats n :: Int -> IO [Int]
nats n = nats >>= return . take n . getAllValuesB
```

4 Conclusion

We presented a revision of encapsulation of non-determinism in lazy functional logic languages. The initial discussion showed that existing approaches are inappropriate. We also showed that the discussed problems can be avoided by imposing two restrictions: (1) there should be no sharing for non-deterministic terms for any encapsulation and the outside, and (2) calls to the search primitives should be explicitly sequentialized. Both restrictions can be ensured by integrating the search primitives into the `IO` monad.

Furthermore, we have argued that a reasonable approach should strongly encapsulate sharing and logical variables and generate a representation of the encapsulated search lazily. Our solution is the lazy creation of a search tree, representing the search space of the computation via a search primitive called `getSearchTree`. Based on the data structure returned by `getSearchTree`, it is possible to define various search strategies on the level of the functional logic language.

The basic idea in the construction of the search tree is adapted from a standard operational semantics for functional logic languages with sharing and, thus, it is a consequent extension of this semantics to the level of meta-programming.

We implemented the presented semantics as an interpreter written in Haskell. Although this interpreter lacks in efficiency and garbage collection, we used it for some interesting examples confirming the potential of our approach.

The search tree presented in this paper covers only the kind of fairness one can accomplish by the operator *try* [11]. However, this is not a principal limitation of the approach. Extending search trees by

```
data SearchTree a = Or [SearchTree a] | Val a | Fail
                  | Eval (SearchTree a)
```

and yielding a value of the form `Eval...` in every `fun` rule, we would cover fair search in the stronger sense. However, the practicability of this extension can only be estimated by an implementation of the approach.

For future work, we want to prove that the presented semantics and the semantics without encapsulation [1] compute comparable results for non-deterministic computations. Furthermore, we want to show that our approach implements strong encapsulation for shared expressions and logical variables, and that the proposed properties of these encapsulation strategies hold. In a next step, we want to use the presented ideas for a new implementation of Curry. In contrast to PAKCS, which translates to Prolog, we want to use Haskell [20] as target language. This implementation should then cover all discussed aspects.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation (to appear)*, 2004.

[2] S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[4] E. Barendsen and S. Smetsers. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[5] B. Braßel. Non-determinism analysis of functional logic programs. Technical report, Christian-Albrechts-Universität zu Kiel, 2004.

[6] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[7] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[8] M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

[9] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[10] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2003.

[11] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[12] M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proc. of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 202–213. ACM Press, 2000.

[13] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[14] T. Johnsson. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer LNCS 201, 1985.

[15] P. Julián Iranzo and C. Villamizar Lamus. Analysing definitional trees: Looking for determinism. In *Proceedings of the 7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, pages 55–69. Springer LNCS 2998, 2004.

[16] P. Koopman, R. Plasmeijer, M. van Eekelen, and S. Smetsers. Functional programming in Clean. `http://www.cs.ru.nl/~clean/contents/Clean_Book/clean_book.html`, 2001.

[17] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.

[18] W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.

[19] L. Naish. All solutions predicates in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 73–77, Boston, 1985.

[20] S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.

[21] S.L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[22] C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.

[23] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

A A Complex Example

To illustrate the operational semantics, we simplify the example of Section 3 as follows (here presented as Curry code):

```
coin :: Int
coin = 0 or 1

main :: IO Int
main = do cST <- getSearchTree coin,
          cST' <- getSearchTree (selFirst cST)
          return (selFirst cST')

selFirst :: SearchTree a -> a
selFirst (Or (t:_)) = selFirst t
selFirst (Val v)    = v
```

Instead of selecting non-deterministically any of the elements of `cST`, we only select the first solution. To demonstrate how the (unevaluated) result of one encapsulation is demanded within another encapsulation, we encapsulate the result `f` again by `getSearchTree` and select the first solution of this encapsulation.

This definition of the `IO` function main is not normalized. After normalization, which requires also lambda lifting due to the presence of higher-order functions (the bind operator `(>>=)`), we get the following FlatCurry program:

```
coin :: Int
coin() = 0 or 1

main :: IO Int
main() = let c = coin(),
             cST = getSearchTree(c),
             m1 = main1()
         in cST >>= m1

main1 :: SearchTree Int -> IO Int
main1(cST) = let f = selFirst(cST),
                 cST' = getSearchTree(f),
                 m2 = main2()
             in cST' >>= m2

main2 :: SearchTree Int -> IO Int
main2(cST') = let f2 = selFirst(cST')
              in return(f2)

selFirst :: SearchTree a -> a
selFirst t = fcase t of {
               Or ts -> case ts of {
                          (t:ts) -> selFirst(t)
                        };
               Val v -> v}
```

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) act cont w = case apply(act,w) of {
                       (a,w') -> let b=apply(cont,a) in
                                     apply(b,w')
                   }

return :: a -> IO a
return x w = (x,w)
```

Since the evaluation of this program still takes 135 steps, it would be tedious to present it entirely. Hence, we restrict to sketching the important parts.

The evaluation starts in the initial configuration

	$[\,]$	`let m=main,w=World(),mIO=apply(m,w) in search(mIO,mIO)`	ε
$\rightarrowtail\!\!\!\twoheadrightarrow$	Γ_0	`search(mIO,mIO)`	ε
$\rightarrowtail\!\!\!\twoheadrightarrow$	Γ_0	`mIO`	`s(mIO)`

By means of `search`, we enforce the evaluation of `mIO` to normal form. Since this computation belongs to the top level `IO` computation, there will be no non-determinism and the result is returned as a value.

After some initializations in the heap, we reach the configuration

	Γ_1	`getSearchTree(c,w)`	S_1
$\rightarrowtail\!\!\!\twoheadrightarrow$	$\Gamma_1[\texttt{z1} \mapsto \texttt{caps}(\Gamma_1,\texttt{c},\varepsilon)]$	`(z1,w)`	S_1

The evaluation of `c` (which refers to `coin()`) is encapsulated without starting its evaluation. In the encapsulation the actual heap is stored. It will later be reused for lazily evaluating the encapsulation. The result of `getSearchTree(c,w)` is the variable referring to the encapsulation embedded in the `IO` monad.

The computation continues unfolding function `main1` with the next application of `getSearchTree`:

	Γ_2	`getSearchTree(f,w)`	S_2
$\rightarrowtail\!\!\!\twoheadrightarrow$	$\Gamma_2[\texttt{z2} \mapsto \texttt{caps}(\Gamma_2,\texttt{f},\varepsilon)]$	`(z2,w)`	S_2

Again the encapsulated computation is not evaluated yet and stored together with the actual heap. The computation continues with the evaluation of function `main2`, which immediately returns the tuple containing `f2` and the world. The tuple is stored in reference `mIO`. Since the main computation is evaluated to head normal form, the stack again has the form `s(mIO)`, which enforces the evaluation to normal form. The computation continues with the evaluation of the first tuple element `f2`, which reduces to `selFirst(z2)`. Now the two suspended evaluation

encapsulations are computed:

$$
\begin{array}{llll}
 & \Gamma_3 & \texttt{fcase z2 of \{Or ts->...;Val v->...\}} & S_3 \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3 & \texttt{z2} & \texttt{f\{...\}}S_3 \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3 & \texttt{caps(}\Gamma_2\texttt{,f,}\varepsilon\texttt{)} & S_4 = \texttt{z2}\,\texttt{f\{...\}}S_3 \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2 & \texttt{f} & (S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2 & \texttt{selFirst(z1)} & \texttt{f}(S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2 & \texttt{fcase z1 of \{Or ts->...;Val v->...\}} & \texttt{f}(S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2 & \texttt{z1} & \texttt{f}(S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2 & \texttt{caps(}\Gamma_1\texttt{,c,}\varepsilon\texttt{)} & S_5 = \texttt{z1}\,\texttt{f\{...\}f}(S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2\Gamma_1 & \texttt{c} & (S_5) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2\Gamma_1 & \texttt{coin()} & \texttt{c}(S_5) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_2\Gamma_1 & \texttt{0 or 1} & \texttt{c}(S_5) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_4 & \texttt{Or(vs)} & \texttt{z1}\,\texttt{f\{...\}f}(S_4) \\
\rightarrowtail\!\!\!\!\twoheadrightarrow & \Gamma_3\Gamma_4[\texttt{z1} \mapsto \texttt{Or(vs)}] & \texttt{Or(vs)} & \texttt{f\{...\}f}(S_4)
\end{array}
$$

$$\text{where } \Gamma_4 = \Gamma_2[\texttt{vs} \mapsto \texttt{[z1,z2]}, \texttt{z1} \mapsto \texttt{caps(}\Gamma_1\texttt{,0,c)}, \texttt{z2} \mapsto \texttt{caps(}\Gamma_1\texttt{,1,c)}]$$

The computation continues within the pattern matching in the right-hand side of `selFirst` applied to `cST'`. This enforces the evaluation of the encapsulation of the first result of coin, namely `0`. Since this computation is similar to the one shown above, we do not present further details here.

In the program, we select the first result of each encapsulated computation by means of the function `selFirst`. Since this function has type `SearchTree a -> a` (it removes the `Val` constructor), the result of the evaluation is `(0,World())`.