Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 246 (2009) 39–53
www.elsevier.com/locate/entcs

A Technique to Build Debugging Tools for Lazy Functional Logic Languages

Bernd Braßel[1,2]

Institute of Computer Science, University of Kiel
Olshausenstr. 40, 24098 Kiel, Germany

Abstract

This paper is based on a recently developed technique to build debugging tools for lazy functional programming languages. With this technique it is possible to replay the execution of a lazy program with a strict semantics by recording information of unevaluated expressions. The recorded information is called an *oracle* and is very compact. Oracles contain the number of strict steps between discarding unevaluated expressions. The technique has already been successfully employed to construct a debugger for lazy functional languages.
This paper extends the technique to include also lazy functional *logic* languages. A debugging tool built with the technique can be downloaded at `www-ps.informatik.uni-kiel.de/~bbr`.

Keywords: Lazy functional languages, logic languages, debugging tools

1 Introduction

It has often been remarked that the advanced features of modern functional (logic) languages pose an obstacle when trying to find errors, cf. for instance [14,17]. Therefore in recent years, sophisticated techniques have been developed to support the programmer with useful tools to find bugs in his programs. The most influential technique is often called "Algorithmic Debugging" (and sometimes "Declarative Debugging") and was originally developed in the context of logic programming [16]. It has also been adopted to functional programming, e.g. in [15], and also to functional logic programming, see [9] for the most recent work. It has, however, also been argued that declarative debugging is not always the tool of choice, and that other tools provide complementary views, cf. [10]. Consequently, the most flexible tool for functional languages, HAT [18], provides several views among which the user can switch arbitrarily. The drawback of such flexibility is paid with a severe

[1] This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.
[2] Email: `bbr@informatik.uni-kiel.de`

doi:10.1016/j.entcs.2009.07.014

overhead in the usage of resources. Every HAT session records megabytes of data about the execution of a given program, often in the hundreds. Much of the work invested into HAT was concerned with making this huge amount of data manageable with acceptable response times. The resulting system was highly optimized for the case of functional programming and is, in consequence, not easy to port to broader settings like functional logic languages. An according attempt developed in [6,5,3] did not lead to the implementation of tools with satisfying performance.

Instead we have developed a different technique in [4] which enabled us to built a fast and stable debugging system within a short time. This technique has first been developed to build a debugger for lazy functional languages [8]. This work describes the extension to the broader setting of functional logic languages.

Before presenting the extension in Section 2, we will first describe the basic idea in Section 1.1. Section 3 contains the description of a debugging tool for the functional logic language Curry based on the technique. Section 4 concludes.

1.1 Leftmost Innermost Evaluation with Oracle

A simple observation was made during the development of [3]: all of the more sophisticated approaches to support debugging in lazy programming languages try to present information about the program's execution *as if* the semantics was eager. In other words, the job of the debugging tools was to wind back the aspects of a complex evaluation strategy and map it to a simple one. Now the reasoning was like this: If this mapping of lazy to eager evaluation is the core of successful debugging techniques, all of these approaches could be derived from mapping a given lazy derivation to an eager one. The information for such a mapping, however, is smaller by magnitudes than that needed by tools like HAT or by that developed in [6]. It can be compressed to counting left-most innermost steps between "discarding steps", i.e., such steps which discard expressions not needed for the whole evaluation. For example, evaluating the expression `(head (tail (from 0)))` in the context of the following program

```
from :: Int -> [Int]
from n = n : from (n+1)

head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

```
head (tail (from 0)) => head (tail (0:from (0+1)))
                     => head (from (0+1))
                     => head (0+1:from ((0+1)+1) => 0+1 => 1
```

can be described as: "Do three steps innermost then discard the next two left-most innermost expressions and do two more eager steps." In short the information can be comprised to the list of eager steps `[3,0,2]`. The first number is decreased and

a leading zero means a discard step. The example derivation can then be mapped to the eager evaluation:

```
                          [3,0,2]
head (tail (from 0)) =>[2,0,2] head (tail (0:from (0+1)))
                     =>[1,0,2] head (tail (0:from 1))
                     =>[0,0,2] head (tail (0:1:from (1+1)))
                     =>[0,2]   head (tail (0:1:from _))
                     =>[2]     head (tail (0:1:_))
                     =>[1]     head (1:_)
                     =>[0]     1
```

In [4] we have formalized a technique to automatically record and replay such step information. Apart from showing the soundness of the approach we were able to prove interesting properties about the magnitude of resources needed to compute the oracle information. In [8] we have then proposed a tool for debugging lazy functional programs with the oracle approach. This paper is concerned with the extension of the approach to functional logic languages.

2 The Extension to Functional Logic Programming

The main topic of this paper is how to transfer the oracle technique described above to the more general setting of functional logic programming. We can identify three main topics of the extension:

(i) operations defined by non-deterministic branching

(ii) free variables in conjunction with narrowing

(iii) free variables in conjunction with unification

From these three topics we will discuss only the two first ones, unification is left for future work.

Before we discuss the details of our solution we first give two well known examples for the two topics. First we will introduce the non-deterministic operation `(?)` on base of which all following non-deterministic operations will be defined. `(?)` takes two arguments and non-deterministically returns one of them.

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

Using `(?)`, we can define an operation which inserts a given argument anywhere into a given list.

```
insert :: a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) = x:y:ys ? y:insert x ys
```

Note that `(?)` has a very low precedence (the lowest possible in Curry actually). Based on `insert` there is an expressive way to define permutations and permutation

sort, cf. [12].

```
permute :: [a] -> [a]
permute []     = []
permute (x:xs) = insert x (permute xs)

permSort :: [Int] -> [Int]
permSort l | sorted permutation = permutation
  where permutation = permute l

sorted :: [Int] -> Bool
sorted []  = True
sorted [x] = True
sorted (x:y:ys) | x<=y = sorted (y:ys)
```

The above declaration of `permSort` indeed defines a sorting operation, e.g., the call `(permSort [2,1,3])` evaluates to `[1,2,3]` and nothing else.

As an example for the use of free variables together with narrowing, we will use the standard definition of a function on lists `head` and a Boolean function `guard`. Both functions are only partially defined such that there is a "narrowing" effect when applying them.

```
guard :: Bool -> a -> a
guard True x = x
```

The running example for narrowing in the following will be `(let x free in guard (head x) 1)` which evaluates to `1` with `x` bound to `(True:y)` for a fresh variable `y`.

After introducing the two running examples we now turn to examine two general concepts in functional logic programming, namely *generators* and computations on *search trees*. Applying these concepts approaches will then lead to the extension of the oracle technique.

2.1 Generators

There are two recent developments in the theory of functional logic languages that simplify the task to extend the oracle technique to this setting considerably. One is the observation that free variables coincide with so called "generator functions" as discovered independently in [2] and [11]. The second is that deterministic and non-deterministic aspects of a functional logic program can be neatly divided into a deterministic evaluation on one hand and a projection of the result according to a set of choices on the other hand. This second idea was described in detail in [7]. Before applying both ideas to the case at hand we illustrate them with regard to the running examples.

A generator for a given type is a function that non-deterministically evaluates to all possible values of that type. For example the generator for Boolean values is straightforward:

```
genBool :: Bool
```

```
genBool = True ? False
```

With regard to types with an infinite set of values we have to be more specific about the structure of a generator. Each alternative on the right-hand side should introduce exactly one of the types' constructors. The constructors' arguments can then be again calls to other generator functions of an appropriate type. According to [2, Definition 3] and in compliance with [11, Definition 3.3] the generator for lists of Boolean values is:

```
genBoolList :: [Bool]
genBoolList = [] ? genBool : genBoolList
```

The resulting function has several interesting properties. Apart from the fact that eventually each possible value is generated (compare to [2, Lemma 1] and [11, Lemma 3.4]), the so defined generator is *productive* in the sense that each alternative produces a head normal form of arbitrary depth in a finite number of steps while introducing only a *finite* number of non-deterministic branches.

In order to be more true to Curry's type system, which allows polymorphic functions and constructors, we change the above definitions slightly. It is easy to see that the claims from [2,11] still hold.

```
type Generator a :: () -> a
genBool :: Generator Bool
genBool _ = True ? False
genList :: Generator a -> Generator [a]
genList gen _ = [] ? gen () : genList gen ()
```

The extension with the artificial argument `()` is necessary because of call-time choice. Without it, `genList` would produce lists that either contain only `True` or only `False` but not, e.g., `[True,False]`.

The connection between generators and free variables can be illustrated with the narrowing example. Evaluating the expression replacing the variable with a generator, i.e., `(guard (head (genList genBool ())) 1)` also yields `1` with the expression being evaluated only as far as needed, that is to the expression `(True:genList genBool ())`. Not only with regards to semantics there is a close correspondence between generators and free variables, e.g., the result is `1`. But also operationally the correspondence is tight as could be shown in [7]. Wherever we have a free variable in the substitution part of a narrowing derivation, we find a non-evaluated generator in the expression for the derivation with generators.

2.2 Search Trees

The second insight important for the presented work was introduced in [7]. There the operational machinery of functional logic programming is separated into a deterministic part computing on so called search trees and a logic part projecting values out of that tree. The general idea is that all types τ are extended by two new constructors `Fail :: τ` which presents a failure and `Or :: Ref -> τ -> τ -> τ` which represents a non-deterministic branching. The reference of type `Ref` is used

by the projection to identify identical choices with respect to call-time choice as explained below. The new `Or` constructors are all introduced by the `(?)` operation, which now – in contrast to the first version above – looks like this:

```
(?) :: a -> a -> a
x ? y = Or r x y where r fresh
```

The generation of fresh references utilized for `(?)` is the only non-deterministic feature needed in order to define the complete operational behavior of functional logic languages as detailed in [7]. In the following we assume that references in `Or` constructors are simple integers although any type with a well-defined identity would suffice. In order to gain the full expressiveness of functional logic programming each pattern matching in the program is extended by a case for the `Or` constructor. For example, the declaration of `insert` is completed by the following rules:

```
insert x (Or r y z) = Or r (insert x y) (insert x z)
insert _ Fail = Fail
```

The definition of `permute` and the other operations introduced above are completed likewise. The only definition for which completion is a bit more complicated is `sorted`, as described below. To get an idea how the completed operations behave consider the following evaluation of `permute [3,1,2]`:

```
permute [3,1,2] => insert 3 (permute [1,2])
                => insert 3 (insert 1 (insert 2 []))
                => insert 3 (insert 1 [2])
                => insert 3 (Or 1 [1,2] [2,1])
{-new rule!-}   => Or 1 (insert 3 [1,2]) (insert 3 [2,1])
                => Or 1 (Or 2 [3,1,2] (1:insert 3 [2]))
                        (Or 3 [3,2,1] (2:insert 3 [1]))
                => Or 1 (Or 2 [3,1,2] (1:Or 4 [3,2] [2,3]))
                        (Or 3 [3,2,1] (2:Or 5 [3,1] [1,3]))
```

To project the search tree to a value we need a set of choices. Such a set defines for which reference which alternative to take. For example the one choice could be `(1,1)` representing that for reference `1` we take the first alternative whereas `(2,2)` would represent the choice to take the second alternative for reference `2`. Projecting the resulting tree of the above example by the set of choices `[(1,1),(2,2),(4,2)]` we would obtain the list `[1,2,3]`. A search now boils down to the systematic construction of sets of choices. In order to illustrate this last point we also complete the definition of operation `sorted` defined above. As `sorted` matches on more than one constructor we have to introduce an auxiliary function. In general, in the resulting program each function matches at most one constructor. This makes it possible to continue the computation in the arguments of an `Or` node. The completed declaration of `sorted` is accordingly:

```
sorted :: [Int] -> Bool
sorted []  = True
sorted (x:xs) = sorted2 x xs
```

```
sorted (Or r x y) = Or r (sorted x) (sorted y)
sorted Fail = Fail

sorted2 x [] = True
sorted2 x (y:ys) | x<=y = sorted (y:ys)
sorted2 x (Or r y z) = Or r (sorted2 x y) (sorted2 x z)
sorted2 _ Fail = Fail
```

The guard "`| x<=y = sorted (y:ys)`" can be considered as syntactic sugar for `(guard (x<=y) (sorted (y:ys)))` giving us the opportunity to illustrate the last part of declaration completion, which is that constructors missing in the matching of the original definition will be mapped to `Fail`:

```
guard :: Bool -> a -> a
guard True       e = e
guard False      _ = Fail
guard (Or r x y) e = Or r (guard x e) (guard y e)
guard Fail       _ = Fail
```

With this it is easy to verify that `sorted (permute [3,1,2])` evaluates to:

```
sorted (permute [3,1,2]) = Or 1 (Or 2 Fail (Or 4 Fail True))
                                (Or 3 Fail (Or 5 Fail Fail))
```

The importance of the references in the `Or` constructors can now be seen when considering the evaluation of calls to `permSort`. Considering its definition the operation `permSort` inserts its argument in those places in the tree where there is a `True`. Therefore a total evaluation of the expression `(permSort [3,1,2])` is:

```
permSort [3,1,2] =
  Or 1 (Or 2 Fail (Or 4 Fail (Or 1 (Or 2 [3,1,2] (1:Or 4 [3,2]
                                                         [2,3]))
                                   (Or 3 [3,2,1] (2:Or 5 [3,1]
                                                         [1,3])))))
       (Or 3 Fail (Or 5 Fail Fail))
```

The important point is that for any projection from that tree to a value this result is equal to

```
permSort [3,1,2] = Or 1 (Or 2 Fail (Or 4 Fail [1,2,3])) Fail
```

The reason is that any path to the sub tree corresponding to `(permute [3,1,2])` leads over the choices `[(1,1),(2,2),(4,2)]`. Therefore these choices are also applied for this sub tree leading to the one possibility `[1,2,3]` only. In addition any choice including `(1,2)` can only yield a failure.

Just as a remark, irrelevant alternatives like those in the sub tree for `(permute [3,1,2])` are often discarded before they are evaluated at all because of laziness. In those cases where they are evaluated they were needed by a former part of the evaluation as in the above example. Nevertheless, it is a good idea to cut away such irrelevant alternatives as soon as possible to free the memory and prevent un-

necessary lookup of choices. However, possible optimization techniques are beyond the scope of this paper and we concentrate on the main advantage of the described technique for building debugging tools. This advantage is that because the main computation is now clearly separated in a deterministic derivation and a projection, the oracle technique developed to replay functional programs can be extended for functional logic programs in a straightforward way.

2.3 A Functional Logic Oracle

There are two main ideas we can directly use to extend the oracle technique to functional logic languages:

- non-deterministic choices can be treated like constructors
- free variables can be replaced by non-deterministic operations

These two ideas can be put together to translate functional logic derivations to strict derivations with oracle. For example, consider the lazy evaluation of the expression `head (insert 3 [1,2])`:

```
head (insert 3 [1,2]) => head (Or 1 [3,1,2] (1:insert 3 [2]))
                      => Or 1 (head [3,1,2]) (head (1:insert 3 [2]))
                      => Or 1 3 (head (1:insert 3 [2]))
                      => Or 1 3 1
```

In order to describe the innermost derivation with oracle, all we need to do is *describe the derivation as if it was a purely functional*. This means, we state that we do the first leftmost innermost step because the corresponding redex `(insert 3 [1,2])` was unfolded and in the result which is `(Or 1 [3,1,2] (1:insert 3 [2]))` the next leftmost innermost redex which is `(insert 3 [2])` is not evaluated and after that all remaining redexes are unfolded. The resulting oracle is therefore `[1,3]` and we can replay the derivation as:

```
                          [1,3]
head (insert 3 [1,2]) =>[0,3] head (Or 1 [3,1,2] (1:insert 3 [2]))
                      =>[3]   head (Or 1 [3,1,2] (1:_))
                      =>[2]   Or 1 (head [3,1,2]) (head (1:_))
                      =>[1]   Or 1 3 (head (1:_))
                      =>[0]   Or 1 3 1
```

As detailed in [7] the search tree approach is well suited to implement search strategies as traversals on the tree structure. This is also compatible with the presented approach. If, for instance the programmer would have been interested in a first solution only with, e.g., a depth first strategy, the above expression would have been evaluated to `(Or 1 3 (head (1:insert 3 [2])))` only and the corresponding oracle would have been `[1,2,0]`.

But not only can we describe the evaluation of non-deterministic operations with the same oracle but also the narrowing of free variables. This can be done by describing how the corresponding generator is evaluated. For example the narrowing

derivation

```
guard (head x) 1 ={x/(a:z)} guard a 1
                 ={a/True}  1
```

can be described by the oracle `[2,7,0]`. The first three decisions describe the binding of the variable as the evaluation of the corresponding generator:

```
    [2,7,0] genList genBool ()
=>[1,7,0] Or 1 [] (genBool () : genList genBool ())
=>[0,7,0] Or 1 [] (Or 2 True False : genList genBool ())
=>[7,0]   Or 1 [] (Or 2 True False : _)
```

The next three steps describe the application of `head` to the generated result:

```
    [7,0] head (Or 1 [] (Or 2 T F:_))
=>[6,0] Or 1 (head []) (head (Or 2 T F:_))
=>[5,0] Or 1 Fail (head (Or 2 T F:_))
=>[4,0] Or 1 Fail (Or 2 T F)
```

And finally `[4,0]` describes the application of `guard` assuming that the programmer was searching for the first solution only, as above.

```
    [4,0] guard (Or 1 Fail (Or 2 T F)) 1
=>[3,0] Or 1 (guard Fail 1) (guard (Or 2 T F) 1)
=>[2,0] Or 1 Fail (guard (Or 2 T F) 1)
=>[1,0] Or 1 Fail (Or 2 (guard T 1) (guard F 1))
=>[0,0] Or 1 Fail (Or 2 1 (guard F 1))
=>[0]   Or 1 Fail (Or 2 1 _)
```

The main result of the consideration is that in order to extend the oracle approach to functional logic programming, the definition of an oracle does not need any change. The main requirement is that the events are recorded in compliance with the operational semantics described in [7].

3 The debugging Tool

We have implemented the ideas introduced in the last section into a debugging tool for the language Curry. The tool is an extension of the one presented in [8] for the functional subset of Curry. In this section we will describe the basic ideas of how to present Curry derivations to the user.

3.1 Representing Non-Determinism

The derivations with `Or` constructors and their references as well as the more technical details of the completed reductions are not suited for the programmer who is looking for a bug in his Curry program. Therefore several conventions help to get a more simple view on derivations.

- unfolding of generator functions is never seen (trusted functions)

- the references of `Or` constructors are hidden; a value like `(Or 2 True False)`, for instance is represented as `(True ? False)`
- irrelevant parts of search trees (recall the `permSort` example from above) are always omitted
- when an `Or` node has only a single valid alternative; this alternative is shown rather than any failures
- calls to auxiliary functions like `sorted2` in the above example are omitted

Accordingly, the following output is generated by a step by step examination for `permSort [2,1]` in our debugging tool:[3]

```
permSort [2,1]
 permute [2,1]
   permute [1]
     permute [] => []
     insert 1 [] => [1]
   permute [1] => [1]
   insert 2 [1]
     insert 2 [] => [2]
   insert 2 [1] => [2,1] ? [1,2]
 permute [2,1] => [2,1] ? [1,2]
 sorted ([2,1] ? [1,2])
   2 < 1 => False
   1 < 2 => True
   sorted [2]
   sorted [2] => True
 sorted ([2,1] ? [1,2]) => True
permSort [2,1] => [1,2]
```

As described in [8], the tool also features a declarative debugging mode. In this mode the user can state correctness or faultiness of sub derivations to isolate erroneous rules.

3.2 Bubbling

In order to omit the representation of references in `Or` constructors without loosing semantically important information, we have adopted *bubbling* as first presented in [1]. Bubbling is related to the approach presented in Section 2.2 in the sense that non-deterministic branching is treated (almost) like a constructor. The `Or` constructors in Section 2.2 are "lifted up" one step at a time by the rules added for completion like `head (Or r x y) = Or r (head x) (head y)`. In bubbling, in contrast, when a `(?)` is at a needed position it is moved up in the term structure until a "proper dominator" has been copied, i.e., a symbol which is above all references to that `(?)`. The exact definition of bubbling [1, Definition 5] is rather

[3] For this presentation, we have deleted some redundant lines and added the spacing.

technical but the idea is quite intuitive and we will use the style of [13]. Consider the following example:

```
let l=insert 1 [2] in (head l,l) =>
let l=[1,2] ? [2,1] in (head l,l)
```

In the approach described in Section 2.2 the end result of this derivation would be `(1 ?1 2,[1,2] ?1 [2,1])`. The references (here denoted as subscript to `(?)`) are then needed to reconstruct the fact that both parts of the tuple share the same choice, i.e., that `(1,[2,1])` is not a valid projection of the result. In bubbling, in contrast, the next step is to copy the whole `let` expression. (If there was an outer context of this expression, that context would not be copied.) In the notions of [1], the tuple constructor is the dominator.

```
let l=[1,2] ? [2,1] in (head l,l) =>
let l=[1,2] in (head l,l) ? let l=[2,1] in (head l,l) =>
(1,[1,2]) ? (2,[2,1])
```

The advantage of this technique is that `?` is never duplicated and, thus, no references are needed. This is why we use the technique to omit the references when presenting values. In our tool, the derivation is presented as

```
main
 insert 1 [2]
   insert 1 [] => [1]
 insert 1 [2] => [1,2] ? [2,1]
head ([1,2] ? [2,1]) => 1 ? 2
main => (1,[1,2]) ? (2,[2,1])
```

As you can see, `head` is applied to the non-deterministic argument `([1,2] ? [2,1])` but the presentation at the end is the result of a bubbling procedure in the pretty printer.

3.3 Representation of Free Variables

As discussed in Section 2.3, the oracle approach maps free variables to the evaluation of the corresponding generator. As a consequence, in the strict evaluation all bindings of a given variable are already computed before any operation is applied to that variable. (Compare to the evaluation of `(guard (head x) 1 where x free)` above.) So far, the debugging tool building on this technique can therefore only access all of the bindings for that variable which will occur in the whole derivation. This can be unexpected for the user and the aim of this section is to explain how a special representation for free variables can be supported. To achieve this, we have to make some adjustments to the search tree mechanism as described in 2.2. The first change is that we need to be able to distinguish between `Or` nodes originating from generator functions and those stemming from a call to `(?)`. To keep most of the described mechanism as similar as possible, we introduce this distinction to the type `OrRef`:

```
data OrRef = Generator Int | NoGenerator Int
```

Accordingly, we change the definitions of `(?)` and the generator operations as follows:

```
(?) :: a -> a -> a
x ? y = Or (NoGenerator r) x y where r fresh
genBool :: Generator Bool
genBool _ = Or (Generator r) True False where r fresh
```

Each narrowing step has to change the `Or` reference such that the corresponding result is not treated as a free variable anymore. For this we use the auxiliary function `narrow`:

```
narrow :: OrRef -> OrRef
narrow (Generator x)   = NoGenerator x
narrow (NoGenerator x) = NoGenerator x
```

Function `narrow` is called in those rules which were added to each function to treat the `Or` case. For example, function `head` is now completed with the following rule (instead of the one used in Section 2.2):

```
head (Or r x y) = Or (narrow r) (head x) (head y)
```

With these changes we can now give a special treatment to free variables and can show the user the derivation of `tuple x where x free)` where `tuple x = (x,not x)` as follows:.

```
main
tuple A
  not A => True ? False
tuple A => (False,True) ? (True,False)
main => (False,True) ? (True,False)
```

3.4 Unification

One of the main open problems with generator functions is how to reclaim the power of the unification operator `(=:=)`. This operator introduces a new quality to functional logic programming. Where *narrowing* only binds free variables to non-variable constructor terms, `(=:=)` can also bind free variables to other variables. In this section we can only sketch the main ideas to solve this problem for the oracle approach.

In order to include unification in the presented technique, we need a further extension of the information contained in `Or` references. In addition to the possibility to tell generator branches from ordinary ones introduced in Section 3.3, we need information about the `Or` references of the children of a generator. As a simple example, to establish the equality between to generated boolean lists

```
genList genBool ()=:=genList genBool () =>
Or (Generator 1) [] (genBool ():genList genBool ()) =:=
```

```
Or (Generator 2) [] (genBool ():genList genBool ())
```

we need not only to establish a connection between the references `1` and `2` but also between the references of the respective arguments of the `(:)`. The according change to the declaration of `OrRef` is:

```
data OrRef = Generator Int [Int]
           | Narrowed  Int [Int]
           | NoGenerator Int
```

Each generator has to include the references for its direct children in the `Or` reference of the parent. The dummy parameter now becomes a proper argument.

```
type Generator a = Int -> a
genBool i = Or (Generator i []) True False
genList gen i = Or (Generator i [j,k]) [] (gen j:genList genBool k)
  where j,k fresh
```

A free variable, e.g. `x::[Bool]`, is now introduced by `genList genBool r where r fresh`.

The next step would be to design a type of constraints. For unification we need only a single kind of constraints but, in principle, other kinds of constraints can be treated in the same way.

```
data Constraint = Eq OrRef OrRef | ...
```

We need an additional constructor `Constraint :: Constraint -> a -> a`. Constraints have to be lifted just like `Or` constructors or `Fail`, e.g.:

```
head (Constraint c x) = Constraint c (head x)
```

Finally, the projection from search trees to values described in Section 2.2 has to be extended to a constraint solver. Due to lack of space we cannot describe the implementation of such a solver and leave this part for future work.

The described extension is implemented in the debugging tool such that the evaluation of `(x=:=True:y &> (x,y) where x,y free)` can be shown as

```
main
A =:= (True : B) => Success
Success &> (True : B,B) => (True : B,B)
main => (True : B,B)
```

4 Related Work and Conclusion

We have presented a recently developed technique to record compact data about programs written in a lazy functional language. We have shown how this technique can be extended to include the advanced features of lazy functional *logic* languages, especially with respect to narrowing and non-deterministic operations. The extension to unification has only been sketched and a more thorough treatment is part of future work.

With respect to related work, the presented approach is complementary, as also described in Section 1. There are many debugging tools for lazy functional languages, cf. [10] for a survey but also for functional logic languages, cf. [9] for the most recent paper. The presented approach is about a technique to record less data about programs and how this technique can be employed to create efficient debugging tools. With the ideas developed in [3] we think that this technique can be applied to integrate all of the related approaches. The transfer of the framework described in [3] and its application to build different debugging views is therefore the next step of the development. The debugging tool built on the presented technique can be downloaded at `www-ps.informatik.uni-kiel.de/~bbr` and includes apart from a step/skip mode the possibility of declarative debugging and virtual I/O as described in [8].

References

[1] S. Antoy, D.W. Brown, and S.-H. Chiang. On the correctness of bubbling. In *Proc. RTA'06*. To appear in Springer LNCS, 2006.

[2] S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

[3] B. Braßel. A framework for interpreting traces of functional logic computations. *Electronic Notes in Theoretical Computer Science*, 177:91–106, 2007.

[4] B. Braßel, S. Fischer, M. Hanus, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, pages 265 – 276, 2007.

[5] B. Braßel, S. Fischer, and F. Huch. A program transformation for tracing functional logic computations. In *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 141–157. Technical Report CS-2006-5, Università ca' Foscari di Venezia, 2006.

[6] B. Braßel, M. Hanus, F. Huch, and G. Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'04)*, pages 179–190. ACM Press, 2004.

[7] Bernd Braßel and Frank Huch. On a tighter integration of functional and logic programming. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.

[8] Bernd Braßel and Holger Siegel. Debugging Lazy Functional Programs by Asking the Oracle. In Olaf Chitil, editor, *Proc. Implementation of Functional Languages (IFL 2007)*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[9] Rafael Caballero, Mario Rodríguez-Artalejo, and Rafael del Vado Vírseda. Declarative Diagnosis of Missing Answers in Constraint Functional-Logic Programming. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *FLOPS*, volume 4989 of *LNCS*, pages 305–321. Springer, 2008.

[10] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, pages 176–193. Springer LNCS 2011, 2001.

[11] Javier de Dios Castro and Francisco J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electron. Notes Theor. Comput. Sci.*, 188:3–19, 2007.

[12] 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.

[13] F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM Press, 2007.

[14] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[15] H. Nilsson and J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering*, 4(2):121–150, 1997.

[16] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[17] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

[18] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.