Run-Time Debugging

for Functional Logic Languages

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Parisa Haj Sadeghi

Kiel

October 2009

1. Gutachter: *Prof. Dr. Michael Hanus*
2. Gutachter: *Priv.-Doz. Dr. Frank Huch*

Datum der mündlichen Prüfung: 18 Juni 2009

to my family

Acknowledgment

It was a pleasure for me to work with all the wonderful people in our team here in Christian Albrechts University of Kiel.

First of all, I would like to thank Prof. Dr. Michael Hanus for giving me the opportunity to become a member of his team. He gave me the chance to attend several interesting conferences. I learned a lot while conducting my research and am convinced that this knowledge will be my great assistance in the future.

This thesis would be incomplete without my special thanks to Priv.-Doz. Dr. Frank Huch for being a great advisor to me. Dr. Huch spent a lot of time in helping me and replied to all my different questions regarding functional logic languages. His ideas and support had a major influence on this thesis.

Finally, I wish to thank the members of my family for their unwavering support during the past five years. Their love and faith that have enabled me to pursue my dream even when the path appeared difficult.

Abstract

This thesis describes the design, implementation and use of a run-time debugging tool for understanding the lazy semantics and locating failures in the functional logic language Curry. We provide a means for programmers to step in the evaluation order of program expressions at a source code level. Every expression evaluated is detected by a program coverage in a layout of the source code. Its run-time value can be represented to the user. The user can stop the execution of a program whenever he or she chooses to do so. A means to backward stepping is also provided. For large programs, we record only partial computations that are generated by evaluating selected expressions from the user.

To achieve these means, we suggest and use some annotations in programs. Representation of intermediate steps of evaluations in a single-step mode is also provided by a distributed programming technique. Stepping in the real order of lazy evaluations could be helpful in searching for failures in simple programs and to beginners in understanding the behavior of functions in functional logic languages.

Contents

Chapter 1

Introduction

> "The reality of the other person lies not in what he reveals to you, but in what he cannot reveal to you. Therefore, if you would understand him, listen not to what he says but rather to what he does not say."
>
> G. Khalil Gibran

Whenever we conclude a study on the subject of software engineering, the recommendations are abstracted in words as *the best software quality in practice* [37, 58]. To develop software, programmers should test programs to assure the accuracy of code. It may not always be possible to identify a program failure by only looking at the source code. Because most software is complex, it may be necessary to observe execution steps of a program to see what is happening at run time. Stepping into the code helps the programmer to understand a relationship between expressions and can lead her/him to locate the position of failures - called bugs - that should be removed from programs.

It takes programmers too much time to find, locate and remove program failures. Tools are needed that support the efforts of programmers to test, trace and debug code [45, 56]. Software testing involves assessing the reliability of software, as opposed to debugging systems. Testing the quality of software not only reveals how usable the software product is, but also provides a review of software programming methods to improve the solutions that are written by the programmers.

Tracers/debuggers are useful tools for illuminating the dynamic nature of programs. They are magnifying glasses that help a programmer to understand a program, as well as to find, isolate and remove program failures. The tools that are implemented as debuggers

usually do not remove the bugs, although they are often called debuggers.

There are several formal approaches to the testing and debugging of programs that are written imperatively or declaratively. The aim of this thesis is to implement a tester and a tracer for programs written in `Curry` [25], a declarative multi-paradigm language.

One of the advantages of declarative languages like Curry is that programs may be written without specifying the evaluation order of expressions. However, because of the difficult-to-predict evaluation order, this advantage makes it difficult to locate an unexpected failure in a program. It is desirable to have tools that support programmers to show executed parts of programs and also follow the evaluation order of expressions. This thesis offers ideas for the testing and tracing of functional logic programs. The ideas suggested are realized in the implemented tool iCODE (*Interactive Curry Observation DEbugger*). iCODE supports programmers with three useful sub-tools:

- COOiSY (*Curry Object Observation Interactive SYstem*)
- CUTTER (*CUrry TesTER*)
- C-Ace (*Curry trACEr*)

COOiSY is an improvement version of COOSy [10] to observe the values of functions and data structures during an execution. CUTTER is a tool to observe the executed parts of programs for different test cases generated by programmers. C-Ace is a tracer developed to follow the evaluation order of program expressions. iCODE is written in Curry using meta-programming, socket-programming and GUI-programming libraries [24].

1.1 *Contributions of the Thesis*

This thesis makes the following contributions:

- There are a variety of methods available for testing functional or functional logic programs [15, 39, 31, 19]. Most of them [15, 19] generate different test cases automatically. Unfortunately, the automatically generated test cases sometimes do not execute parts of programs that may contain bugs. It is important to know which parts of a program have not been executed. They may then be tested by other test cases.

  Representation of executed parts of programs based on the desired expressions is one of the approaches of program slicing [72] (Section 3.5.2). With respect to program slicing, we suggest a program testing method for different test cases generated by the

programmer. This method is implemented by the tool CUTTER. By manipulating a program, we can highlight executed parts of this code in a tree representation of the program or in the original source code. Showing highlighted executed expressions helps the programmer to generate other test cases for running non-executed parts of a program that have not yet been highlighted and which may be executable. We also represent the number of function calls that help the programmer to focus on the quantity of function definitions and also to locate non-terminating computations that may exist in a program.

- The task for most implemented tracers for non-strict languages is first to record internal details of a computation and then to review this information in order to locate failures. It is clear that, for a large computation, intermediate information contains a great number of details. Users need too much time to review these details. With some tools [5], it is possible to put check-points in programs. With the use of check-points, the tool represents the internal details of computations only for interval between the check-points. Some other tools [20, 10] allow the programmer to annotate desired expressions from a program. For these tools, the information recorded is related only to annotated expressions and not to the whole program. Representation of information to the user can be similar to a textual visualization [20, 10], graphical visualization [53], computation tree [5], evaluation dependency tree [47, 71] or other different representation.

  This thesis offers a new idea for reviewing the internal details of a computation. To avoid storing large computations in a file, we use a distributed programming technique [23]. That means, we represent the information of computations during (not after) an execution. The different modules of our project are clients and servers, which communicate with each other by sending the internal details of computations as messages. The messages provide the information about expressions evaluated. They are represented to the user as textual visualization during execution. Representing values as a textual visualization in iCODE is an extension of Gill's idea [20] that is also implemented in COOSy [10].

- In COOSy, the evaluation of functions or data structures can be observed by annotating program expressions. It is performed manually by the user. Unfortunately, manipulating programs requires the kind of effort that discourages the users from using this kind of tool. On the base of Gill's idea [20], each annotation in Haskell [35, 33] has a type class restriction on the expression being observed. There are also instances for all the Haskell base types, such as `Int`, `Bool`, etc. In contrast

to Haskell, Curry does not provide type classes. Therefore, for every observed expression a special observer should be added to programs that are related to the type of values observed. This could be another problem for programmers, especially beginners. Our solution for these problems is to generate all of these details automatically. The user should be required to only select the expressions desired for observation. Before executing a program, all source code modifications are performed automatically following selection of expressions by the user.

- Sometimes users would like to be able to move forwards and backwards to explore the internal details of evaluations in a single-step mode. By navigating evaluated expressions like [71] manually, the tool may lose orientation. However, the tool can follow an evaluation when the user answers a sequence of questions. If the question is not answered truthfully, the tool provides inaccurate information (Section 3.5.1). The idea of this thesis is to represent the internal details of evaluations in an automatic method of navigation. The user also will have the ability to stop the execution of a program at any desired point.

  The step-by-step provision of representations in iCODE is an interesting approach that helps beginners to understand the lazy execution semantics of Curry. It can also be helpful in locating bugs in simple programs.

1.2 *Structure of the Thesis*

Chapter 2: The work reported in this thesis concerns an approach to testing and tracing the execution of programs written in the functional logic language Curry. This chapter provides an overview of this language.

Chapter 3: In this chapter, we review some testing, tracing and debugging methods.

Chapter 4: One of the research contributions of the thesis is related to the observation system COOSy. Applications of COOSy are offered in this chapter. In the observation system, run-time values of evaluated expressions can be represented by inserting appropriate observers in programs. We also discuss in this chapter how programs should be manipulated for this purpose.

Chapter 5: The algorithms of COOSy and iCODE are discussed in this chapter.

Chapter 6: The operational semantics of the observe applications is discussed in this chapter.

Chapter 7: One of the ideas in our work is representing highlighted dynamic slices in a tree representation of programs or in the original source code. This method is discussed

in this chapter.

Chapter 8: The most important thing that a well-implemented tool can do to help programmers track down problems is to provide easy-to-understand information. Programmers think in expressions of their source code. She/He has a detailed map of the program in her/his mind. Displaying information about the executions in a lay-out of source-lines can be an interesting representation. It is an idea proposed in this thesis. We have realized our idea in the implemented tool iCODE. The architecture of iCODE is described in this chapter. It concentrates on two different architectures. One is provided for representation of information from evaluations in a single-step mode. The other is provided for display of the most recent information from executions without representing the internal details.

Chapter 9: In this chapter we briefly compare other existing debuggers to iCODE.

Chapter 2

Functional Logic Languages

"Remember the flight, the bird is mortal."

Forough Farrokhzad

In this chapter, we first compare the concept of imperative and functional programming languages. Then, we introduce the main features of functional languages, like Haskell. Finally, we discuss Curry and the features that Curry added to Haskell to develop a functional logic language.

2.1 *Imperative in contrast to Functional Programming*

The style of pure functional programming is based on the method in which all computations are defined as applications of functions to arguments. Unlike imperative languages, there are no side effects when expressions are evaluated.

To illustrate this difference, we consider an example. In this example, we compute the sum of three numbers. In most imperative languages, such as `C` [63, 52], this example can be implemented by using three variables: a counter to count up to 3, an array in which to store the three numbers and an accumulator to accumulate the intermediate results of computations. We write this example in `C`:

```
int main() {
  int a[3]    = {11,21,37};
  int sum     = 0;
  int counter = 0;
```

```
  while (counter < 3) {
    sum     = sum + a[counter];
    counter = counter + 1;
  }
  return (sum);
}
```

In this program, the expression `a[counter]` is defined as an array with `counter` elements. The expression `while (<condition>) {<expressions>}` repeatedly computes the `expressions` until the `condition` is no longer true. `sum` is an accumulator to store intermediate computations and `counter` to count the number of computations. All variables that are used in the program need to be instantiated dependent on the type of these variables. In our example, we instantiate the array `a[3]` with three elements `{11,21,37}` and `sum` and `counter` with zero.

The while-expression repeatedly adds an element of the array to the accumulator and triggers the counter until the three elements are added and the counter reaches 3. By representing the intermediate results in the following table, the computation should become clear.

sum	counter
0	0
0+11 = 11	0+1 = 1
11+21 = 32	1+1 = 2
32+37 = 69	2+1 = 3

As we see, the values are stored and change during program execution. The program contains a sequence of assignments. It is constructed with imperative instructions that specify how the computation proceeds. Computer languages that use this kind of programming method are called `imperative languages`.

To compare the C program above with a functional programming method, let us rewrite the program in Haskell language [35, 33]. To produce lists in Haskell programs, brackets are used: `[]` as an empty list and `[a,b,c]` as a list with three elements - `a`, `b` and `c`. The symbol ":" is also used to produce a list. The list `(x:xs)` indicates that the first element of this list is named `x` and that the list of all remaining elements is named as `xs`. `xs` can be an empty or non-empty list. Consider the following Haskell program:

```
main       = sum [11,21,37]
sum []     = 0
sum (x:xs) = x + sum xs
```

The symbol "=" is used for function definitions. The left-hand side of the symbol "=" represents the name of a function and the arguments to which this function applies. The right-hand side of the symbol "=" is an expression that causes a program to compute the result of the function. In our example, the left-hand side of the function `sum` shows that this function can be applied to two different lists. In the first equation, it can be applied to an empty list. In the second equation, it can be applied to a non-empty list. The right-hand side of the first equation shows that an application of `sum` to an empty list offers zero as a result. The second application has another definition. An application of `sum` to a non-empty list offers an addition between the first element of the list and the result of an application of `sum` to the remaining list elements.

Note that the program is implemented by applying functions to arguments. This is the basic method of computations in functional languages. In other words, in contrast to imperative programs that are a sequence of assignments, functional programs are sets of function definitions that are built on function applications. Let us review the intermediate applications of our functional program in the following table.

apply	*result*
main	sum [11,21,37]
sum	11 + sum [21,37]
sum	11+(21+sum [37])
sum	11+(21+(37+sum []))
sum	11+(21+(37+0))
+	11+(21+37)
+	11+58
+	69

Compare the two tables. In imperative programs, we must store values. The stored values change during the execution of assignments. In functional programs, values are passed only as arguments from one function to the other.

A nice feature of Haskell as a functional language is that every function has a type that specifies the nature of its arguments and results. The type of functions is inferred from function definitions during the compilation of a program. It detects many type errors before executing a program. Type errors are one of the problems inherent in languages that check the type of variables during the execution of a program and not during compile time. For example, if a type error is contained in a branch-statement, it is possible that it will not be detected. Another example is type conversion.

Let us consider the following program that has been written in C:

```
int main (int x) {
   char y;
   y = x;
   y = y+1;
   return y;
}
```

In the above example, integer `x` is assigned to character `y` (y=x). If a value of `x` is large during execution, it is possible that this value will be lost.

Now consider the following program, which uses an unset variable:

```
int main (int x) {
   int y;
   if (x) {
     y = x;
   }
   return (y);
}
```

Note that the variable `y` receives a value of `x` within the if-expression. In the case of false, `y` is returned with an unknown value. However, in the case of true, the value of `x` is assigned to `y`. That means that the error will not be detected in the case true.

2.2 *Functional Core*

Functional programming languages have several nice features [32]. The structure of this kind of program is like the methods that we use in mathematics. The semantics of these languages is defined in terms of its foundation in lambda calculus [54, 40]. Further, functional languages offer an interesting polymorphic type system, higher-order functions, I/O monads and a pattern matching method for equational definitions of functions. Lazy semantics add another nice property to functional programs: function parameters are evaluated only on need.

In this section we briefly discuss the features of functional languages like Haskell [35, 33].

2.2.1 *Mathematics Style*

In mathematics, every function application is defined by the name of the function and its arguments, which are enclosed in parentheses. For example, the expression `2x+f(x,y)`

means that we have a multiplication of `2` and `x`. The result of the multiplication must be added to the result of the application between the function `f` and its arguments `x` and `y`. Functional programs use this style to define function applications in programs. In Haskell or Curry, the application above can be easily defined by using spaces between arguments and `*` to denote multiplications:

```
2*x+f x y
```

Now it is easy to define a function definition like `g(x,y)=2x+f(x,y)` as an equation in a functional program. We must only separate the arguments with spaces:

```
g x y=2*x+f x y
```

Furthermore, in functional programs like Haskell, a function application has a higher priority than other operations. For example, the expression `f x y+x` means `(f x y)+x`.

Note that each variable in the left-hand side of an equation occurs only once (left-linear). For example, the equation of `f x x=2*x` is unacceptable, since the variable `x` appears twice in the left-hand side of the equation.

2.2.2 *Pattern Matching*

The left-hand side of function definitions can contain *patterns*. They are arguments related to the function definition. For example, in the following function definition, `0` and `1` are patterns:

```
f 0 = 10
f 1 = 20
```

Note that the definition of the function `f` is represented in two equations. Having different rules (equations) for a function requires a strategy to compare function patterns with actual parameters. Comparing patterns and parameters is called pattern matching. Pattern matching is applied to patterns in the order in which they are defined in function equations. In our example, `0`, and then `1`, is matched. The sequence of this matching should become clear when we represent all functions by case expressions:

```
f x = case x of
        0 -> 10
        1 -> 20
```

The case mechanism of Haskell allows pattern matching to be used in the right-hand side of a function definition, in our example by matching `x` against `0` and `1` to choose between

two possible results. In the example above, a function application of `f` to an argument (`x`) chooses the first result `10`, if the given argument is `0`. If the second pattern (`1`) is matched, the second result (`20`) is chosen.

2.2.3 *Lambda Expressions*

Functions can be constructed using lambda expressions. Every lambda expression needs patterns to define arguments and a body that defines how the function works:

```
\pattern -> body
```

Unlike mathematics, which needs a function name for every function definition, functions have no names in lambda expressions. Lambda expressions are defined with the symbol "\" in Curry. The entire structure of lambda expressions can be used like function names in function applications. For example, the lambda expression `\x -> x+1` can be defined as an increment function. Applying this function to an argument (`y`) is represented as:

```
(\x -> x+1) y
```

2.2.4 *Function Types*

All expressions have types, specially functions. A function is a mapping from arguments to results. Arguments and results have also types. If the type of arguments is `TypeArgs` and the type of results is `TypeRes`, the mapping is represented by `TypeArgs -> TypeRes`. Imagine that this type is specified for a function (e.g., `fun`). Now the function `fun` has a type that is defined as follows:

```
fun :: TypeArgs -> TypeRes
```

A type can be related to a collection of values. For example, if the function `fun` takes no arguments but returns an integer as a result, the type of this function can be:

```
fun :: Int
```

As another example, imagine that the function `fun` takes an integer as an argument and returns Boolean values as the result. For this definition, we need a map between the type of arguments and results:

```
fun :: Int -> Bool
```

Haskell and Curry are strongly typed. That means that every value has a type that can be inferred from programs at time of compilation. Inferring types is done by a process

called *type inference*. For example, if the application `fun expr` exists in a program so that `fun` is a function of type `T1 -> T2` and `expr` is an expression of type `T1`, the application `fun expr` is of type `T2`.

2.2.5 *Curried Function Definitions*

Functions that take their arguments one at a time are called curried. For example, the function `add` that is defined as `add x y = x+y` can be translated to:

```
add = \x -> (\y -> x+y)
```

Note that the function `add` takes an argument and returns a function. This function takes the second argument and returns the result `x+y`. In other words, the function `add` is applied to its two arguments (e.g., of type `Int`) as follows:

```
add :: Int -> (Int -> Int)
```

The above definition is also used for functions with more than two arguments. For instance, if we define the function `add3` with three arguments:

```
add3 x y z = x + y + z,
```

the type is:

```
add3 :: Int -> (Int -> (Int -> Int)).
```

To avoid using parenthesis in programs written in Haskell or Curry, the type constructor (`->`) is defined as right-associative, meaning that the type of `add3` can be written without parenthesis as follows:

```
add3 :: Int -> Int -> Int -> Int
```

Note that function applications are left associative. For example, an application of `add3` to three arguments like (`add3 1 2 3`) means:

```
(((add3 1) 2) 3)
```

This definition brings up another feature of functional languages called *partial application*. It means that a curried function can be applied to arguments while the number of arguments is less than that defined in the function type. Let us consider the increment function as a simple example:

```
increment :: Int -> Int
increment x = x + 1
```

The function `increment` needs one argument and the operator `+` needs two arguments. By means of partial applications, we can define `increment` as follows:

```
increment = (+) 1
```

As we see, there are no arguments for the function `increment` and the operator `(+)` has only one argument (`1`). Note that the type of `increment` is un-changed:

```
increment :: Int -> Int
```

It means that every function call of `increment` needs one argument.

2.2.6 *Data Types*

As we have discussed, Haskell and Curry are strongly typed. A completely new type can also be defined using the *data* construct in these languages. For example, we can specify that the type `Color` has different values like `Blue`, `Red`, and `Green`:

```
data Color = Blue
           | Red
           | Green
```

The symbol `|` means "or" and `Blue`, `Red` and `Green`, values of the type `Color` are called *constructors*. The name of each constructor begins with an upper-case letter. Note that we are not allowed to use the same constructor name for different data types.

Constructors can also have arguments. As an example, let us define a data type for binary trees. Every tree can have leaves and nodes. Nodes construct other sub-trees that have the type of the tree's root-node:

```
data Tree = Leaf Int
          | Node Tree Int Tree
```

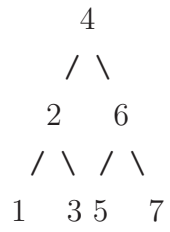
The data type is named `Tree`. It has two constructors - `Leaf` and `Node`. The constructor `Leaf` has one argument of type `Int`. The constructor `Node` is defined by three arguments that construct two sub-trees of the node and a value of type `Int`. The type of sub-trees is equal to the type of the parent node (`Tree`).

User-defined data types can be passed as arguments to functions. To illustrate this possibility, we define a function to compute the depth of a binary tree:

```
depth :: Tree -> Int
depth (Leaf x) = 1
```

```
depth (Node leftSubTree x rightSubTree) = 1 + max (depth leftSubTree)
                                                  (depth rightSubTree)
max :: Int -> Int -> Int
max x y = if x>y then x else y
```

Suppose that we wish to apply the function `depth` to a binary tree with the following values:



This tree can be represented by the data type `Tree`:

```
binaryTree :: Tree
binaryTree = Node (Node (Leaf 1) 2 (Leaf 3))
                  4
                  (Node (Leaf 5) 6 (Leaf 7))
```

Now we should only write an application: `depth binaryTree`. The result is 3.

2.2.7 *Parametric Polymorphism*

In functional languages there are several predefined types like `Int`, `Char`, `Bool` and etc. These languages also feature polymorphic types which can be used for different types of an argument. One example is the identity function `id`:

```
id :: a -> a
id x = x
```

The type `a` shows that the function `id` can take every type. Polymorphic types are represented by *type variables*. A type variable must begin with a lower-case letter.

A list is one of the important data structures that has polymorphically typed elements. A list is a set of elements of the same type. If the elements of a list are of type `T`, the type of the list is represented as `[T]`. Lists can be empty (`[]`) or have elements that are separated by commas (e.g., `[1,2,3]`). A list can also be represented by the constructor `(:)`. This constructor needs the first element of the list and the remainder list. For example, the list `[1,2,3]` can also be represented as: `(:) 1 [2,3]`.

We use this definition to define a data type for lists that have polymorphic type elements:

```
data [a] = [] | (:) a [a]
```

Let us write a function to apply to this defined data type. For example, we will define the function `length`:

```
length :: [a] -> Int
length [] = 0
length (a:aList) = 1 + length aList
```

This function computes the length of a given list that contains elements of any type. For example, it can compute the length of a list with integers as elements, such as `length [1,2,3,4]`. This application offers 4 as the result. The function `length` can also be applied to a list of strings, like `length ["Hello","World","!"]` that returns 3 as the result and so on. It means that the function returns an integer independently of the type of its list arguments.

2.2.8 *Higher-Order Functions*

Functions can be passed as parameters to other functions or returned as results of function applications. The best example is the function composition. Using lambda expressions, we can define the meaning of the function composition as follows:

```
f .  g = \x -> f (g x)
```

With the help of mathematics style of functional programs, we can define a function composition by the infix operator ".". The lambda calculus shows that the infix operator "." takes the argument `x` and the two functions `f` and `g` and returns a composition between the result of the two functions on the argument `x`.

For another example, look at the following program:

```
main xs = map increment xs
```

By means of higher-order functions, we have used the function `increment` as an argument for the function `map`. The type of `map` is defined as follows:

```
map :: (a->b) -> [a] -> [b]
```

`map` takes a function (e.g., `f`) of type `a -> b` and a list (e.g., `l`) of type `[a]` as arguments and returns a list (e.g., `l'`) of type `[b]` where the function `f` is applied to every element of the list `l` to generate the elements of `l'`. This function in our example applies the function `increment` to every element of the list `xs`. For example, the application `main [1,2,3]` offers `[2,3,4]` as the result.

2.2.9 *Lazy Semantics*

Lazy semantics in functional languages provides two features:

- *Non-Strictness*
- *Sharing reductions*

Non-Strictness: Most programming languages are strict. This means that, in a function call, all arguments of the function are evaluated before the function itself is called. In the strict semantics, if an evaluation of an argument fails, the function call also fails and we have a run-time failure. In the non-strict semantics, only arguments are evaluated that are needed to compute the right-hand side of a function definition (*lazy strategy*). For example, in the function definition (`f x = 1`), no argument given as `x` will ever be evaluated, because it is not needed to compute the right-hand side of this function.

As an example, consider the following recursive function definition:

```
nonTermFun :: Int
nonTermFun = 1 + nonTermFun
```

Evaluating `nonTermFun` produces larger and larger expressions and does not terminate:

```
   nonTermFun                  {applying nonTermFun}
 = 1 + nonTermFun              {applying nonTermFun}
 = 1 + (1 + nonTermFun)        {applying nonTermFun}
 = 1 + (1 + (1 + nonTermFun))  {applying nonTermFun}
 = ...
```

Now consider the expression `fst(0,nonTermFun)` where `fst` selects the first component of a pair:

```
fst(x,y) = x
```

There are different evaluation strategies to compute an expression. One common strategy is called *innermost* evaluation. In this strategy, all arguments of a function are always fully evaluated before the function itself is applied. In other words, arguments are passed by values (*call-by-value* evaluation). If we use this strategy to compute the expression `fst(0,nonTermFun)`, it results in non-termination:

```
   fst(0,nonTermFun)           {applying nonTermFun}
 = fst(0,1 + nonTermFun)       {applying nonTermFun}
 = fst(0,1 + (1 + nonTermFun)) {applying nonTermFun}
 = ...
```

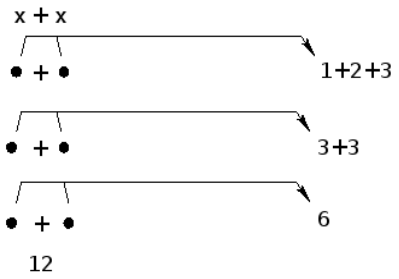


Figure 2.1: Sharing the reductions

Another common evaluation strategy is called *outermost* evaluation. In this strategy, functions are to be applied before their arguments are evaluated. Consequently, we say that arguments are passed by names (*call-by-name* evaluation). In contrast to innermost evaluation, the computation of the above example in an outermost manner results in termination in only one step:

```
   fst(0,nonTermFun)  {applying fst}
 = 0
```

Note that in the above evaluation strategy, only the argument is evaluated that is needed to compute an application of the function.

Sharing reductions: Program expressions can be computed more than once in the right-hand sides of function equations. The lazy semantics provides a means to share the results of evaluations. This means that (1) each expression is evaluated at most once, (2) a copy of the result is kept and (3) other equal expressions have indicators that point to it.

Consider the following simple function:

```
f = let x = 1+2+3 in x+x
```

In this definition, the expression `x=1+2+3` is evaluated once when computation of the `x` on the left in the expression `x+x` begins. In other words, when the computation of the next `x` is started, the evaluation of `x=1+2+3` is not needed, because it has already been evaluated and the result can be shared by pointers to the result. The sequence of evaluations is presented in Figure 2.1. In the first step of this figure, the operator `+` applies. In this step, we keep a single copy of the argument `x` and create two pointers to it. In the second step, the evaluation of `1+2` is completed and both pointers share the result. At the end, the value of both `x` is `6`.

The use of non-strictness in conjunction with sharing is called *lazy evaluation*. Languages that use this strategy are called *lazy programming languages*.

2.2.10 *Input/Output Monads*

Many programs require functions that can behave interactively during program execution, such as functions that read input from the keyboard or write output to the screen. The functional language Haskell provides such an ability without compromising the basic semantics of pure functional languages. Generating interactive functions in Haskell is based on a style of programming named *imperative functional programming* [34, 41]. Imperative functional programming has its theoretical concepts in the monadic style of representing side effects. Monads [70] are bridges between practical programming and functional specification methods.

Interactive functions in Haskell that support the monadic style take the state of the world as an argument and return a world as a result. The type of interactive functions could be defined by `IO` (short for input/output):

```
type IO = World -> World
```

where `World` denotes the type of all states of the outside world.

Every computation of type `IO a` results in side effects behind what it represents. The visible effect is accessed by returning a value of the type `a`, if no failure occurs. For example, a function can read a character from the keyboard. Results of such functions are of type `IO Char`. In general, the type `IO a` can be considered as:

```
type IO a = World -> (a,World)
```

Note that, if an interactive function generates no result, its type is represented by `IO ()`.

To return a result to the world, the following function can be used:

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

The function `return` only returns values of `x` without performing any modification on the world. Interactive functions that perform different applications without modifying the world can be defined as a sequence of applications. This sequence is represented by the operator `>>`:

```
(>>) :: IO a -> IO b -> IO b
```

Function applications that are defined as modifying the world are also performed in a sequence. In other words, the modified world from the first application is returned to the second as the current state of the world for the second application. This sequence applies to all applications in interactive functions. Such a sequence is conceptually defined by the operator (`>>=`):

```
(>>=) :: IO a -> (a -> IO b) -> IO b
f >>= g = \ world -> case (f world) of
                       (x,modified_world) -> g x modified_world
```

In this definition, the function `f` is first applied to the current state of the world and returns the modified world that is the argument given for the second function `g`. This sequence is an application area for monads and is a concept of sequential languages (imperative languages).

Interactive functions can also be constructed by the `do` notation. For example, the following program takes a string from the world and returns the length of the string:

```
computeLength :: IO Int
computeLength = do str <- getLine
                   return (length str)
```

The type of this function is defined as `IO Int`, because the function returns an integer as the result (the length of the string).

2.3 *Functional Logic Programming and Curry*

Functional logic languages (also named *declarative languages*) bridge functional (Section 2.2) and logic languages [67] to provide a combination of the advantages of these two languages. Some important functional logic languages are `Curry` [25], `Oz` [68], `Toy` [13] and `Mercury` [64]. The implementations described in this work is based on Curry. The design of the multi-paradigm language [26] Curry was started in 1997 [27] in order to introduce two features [11, 6] of logic languages to Haskell:

- Non-deterministic computations
- logical variables

2.3.1 *Non-Determinism*

In functional logic languages, functions can behave non-deterministically [11, 6]. That means that a function can offer different values for a given set of arguments. Functions that behave non-deterministically have overlapping [4] in their equations or contain logical variables in their definitions.

As an example, we can consider different teachers in a school. Some teachers teach only one subject and others teach more than one. In most programming languages, we

need a data structure in which to store the information related to teachers. However, with the help of the non-deterministic semantics of Curry, we can write the following equations for the function `teach`:

```
teach :: Name -> Subject
teach Pitt   = Mathematics
teach Pitt   = ComputerScience
teach Potter = Physics
teach Potter = Mathematics
teach Kidman = Sport

data Name    = Pitt | Potter | Kidman
data Subject = Mathematics | Physics | ComputerScience | Sport
```

Every function application of `teach Pitt` offers two results - Mathematics and ComputerScience.

2.3.2 *Logical Variables*

In functional programs, variables that are used in the right-hand side of an equation constitute the parameters that belong to the left-hand side of this equation. For example, the following equation is acceptable (the operator `==` returns true when the right and left expressions of the operator are equal):

```
f x xs = (x:xs) == [1,2,3]
```

but the following equation:

```
f x xs = (x:ys) == [1,2,3]
```

is wrong, because the variable `ys` does not appear in the left-hand side of the equation and will not bind to a value during program execution. The functional logic language Curry provides the means to use such variables. These kinds of variables are named logical variables. Since values of logical variables are unknown, these kinds of variables are also named *uninstantiated variables*.

Logical variables must be declared `free` in local definitions of programs and can only occur in the right-hand side of equations. Note that Curry is strongly typed. It means that a variable occurs in an expression represents all possible values with respect to the type of this expression.

As an example, consider the following simple program. In this example, the variable `ys` is defined logically:

```
f x xs = (x:ys) =:= [1,2,3]
    where ys free
```

The operator"`=:=`" is called *constrained equality*. There are two differences between the constrained equality "`=:=`" and the Boolean equality "`==`". The operator "`==`" returns true if both left and right expressions of the operator have equal values. Otherwise, it returns false as a result. It means that the result of such an equality is of the `Boolean` type. The operator "`=:=`" has another type - `Success`. Look at the constrained equality in our example. If there is a list (ys) that can be appended to a value of `x` and generate the list `[1,2,3]`, the operation of "`=:=`" terminates successfully. Otherwise, the equation fails. For example, a function call like `f 1 [4]` in our example represents success as the result, but the function call `f 2 [4]` fails.

Another difference between the operator "`==`" and "`=:=`" is that the operator "`=:=`" narrows (*flexible* operator), whereas the operator "`==`" residuates (*rigid* operator):

- *Narrowing*: Narrowing [8] is an evaluation strategy to instantiate logical variables. If one expression `expr` contains the uninstantiated variable `x`, narrowing with a non-deterministic manner instantiates `x` so that the evaluation of `expr` may continue. Instantiating a variable means replacing the value that is guessed by narrowing. For example, we consider the following program:

  ```
  add 0 x = x
  add 1 x = 1+x
  ```

  The expression `let var free in add var 1 =:= 1` can be narrowed by instantiating `var` to `0` and rewriting `add 0 1` to `1` (by *term rewriting* [38]). Note that any other digit is not an acceptable guess, because only the first defined rule in program returns `1` when the second argument is `1`. In other words, the addition of no other digit with `1` yields `1` as a result. The evaluation of the expression `let var free in add var 1 =:= 1` terminates successfully where `var` is bound to `0`.

- *Residuation*: If the expression `expr` contains the variable `x`, residuation will suspend the evaluation of `expr` and transfer control to other parts of the program until `x` is instantiated to evaluate `expr`. Now consider the `add` function definition. This time, we evaluate the expression `let var free in (add var 1 =:= 1 & var=:=3-3)`. This expression is evaluated by residuation where the instantiation of the variable `var` needs an evaluation of `3-3`. In other words, an instantiation of `var` is possible by evaluating other parts of the program related to this variable. After instantiating

this variable, the expression `add 0 1` is matched with the first rule defined in the program and is reduced to the right-hand side of this rule. Now the program returns success as a result.

To illustrate how narrowing binds a logical variable by the run-time searching method [29], consider again the `add` example. The function `add` takes `0` or `1` as the first argument. The second argument is arbitrary. To understand the semantics of logical variables, we apply the function `add` to a logical variable `var` and an argument `3`:

```
let var free in add var 3
```

The evaluation of this application is successful when the logical variable `var` is bound to `1` or `0`. At first, we consider the binding of `1`:

`add var 3` $\leadsto_{\{\texttt{var/1}\}}$ `(1+3)` $\leadsto$ `4`

The first binding that is represented as {`var/1`} was successful. It caused an evaluation of the second equation of `add`. Now the search is continued until all possible bindings are successful. The second value that `var` can bind to is `0`. It causes an evaluation of the first equation of `add`:

`add var 3` $\leadsto_{\{\texttt{var/0}\}}$ `3`

There are no more values that can bind to `var`. Note that there are two different results for one function application.

There are different narrowing strategies to improve the method of guessing logical variables. One of them is called *needed narrowing* [3]. It corresponds to the lazy evaluation in Haskell, with differences. One difference is that unification [55] is used instead of pattern matching for parameter passing. This means that, if an argument of a function contains a logical variable, it is bound to some terms as tightly as possible.

As an example, we consider the following program containing a declaration of natural numbers using the data type `Nat` and two function definitions. The first function `add` is defined for addition between natural numbers and the second one `leq` is considered to be a "less than or equal" function (the underscore indicates an unnamed variable):

```
data Nat = O
         | S Nat

add O     y = y
add (S x) y = S (add x y)

leq O     _     = True
```

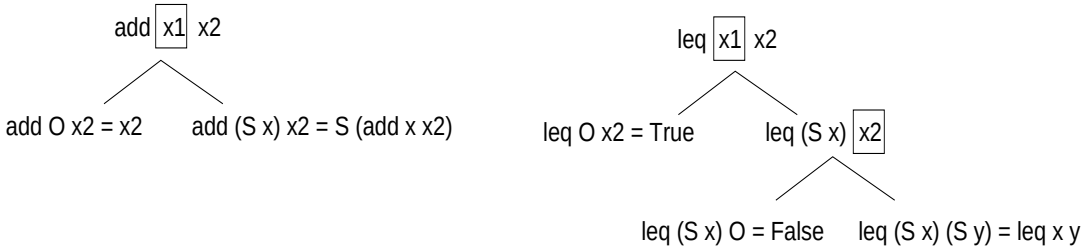


Figure 2.2: Definitional trees of `add` and `leq`

```
leq (S _)  O    = False
leq (S x) (S y) = leq x y
```

Now we evaluate the expression `let var1,var2 free in leq var1 (add var2 O)`. Look at the program. The first argument of `leq` in all its rules is a constructor-rooted term. By an application of `leq var1 (add var2 O)`, needed narrowing instantiates `var1` to `O` or `S z` where `z` is a fresh variable. If `var1` is bound to `O`, only the first rule of `leq` is used but in another case, the second or third rule of `leq` is applicable. A selection between these two rules is possible when the second argument of `leq` is instantiated, because this argument is a constructor-rooted term in the second and third rules of `leq`. To evaluate the second argument `add var2 O`, `var2` should be instantiated to `O` or `S w` where `w` is a fresh variable. As we see, the patterns defined in the left-hand sides are important for applications of needed narrowing. The order of variable bindings can be represented as a tree named *definitional tree* [28]. The definitional trees of our example for the functions `add` and `leq` are presented in Figure 2.2.

2.4 *Meta-Programming*

Meta-programs or code-generating programs are written to generate programs or program parts. Meta-programming is used in different ways in different languages. Compiler tools that generate programs as outputs are examples of meta-programming.

Curry supports the meta-programming technique to transform Curry programs in another structure. There are two methods for this transformation that are implemented in FlatCurry and AbstractCurry libraries [17]. Each defines data types for transformations.

To transform programs into AbstractCurry, the following function is defined:

```
readCurry :: ModuleName -> IO CurryProg
type ModuleName = String
```

`readCurry` takes the name of a Curry program and returns a program that is constructed in the data type `CurryProg`. All detail of a Curry program are represented in `CurryProg` by different data type constructions [17]:

```
CurryProg :: ModuleName -> [ImportedModule] -> [TypeDeclaration]
          -> [FunctionDeclaration] -> [OperatorDeclaration]
          -> CurryProg
```

As an example, suppose that we have written the following simple increment function in the file `example.curry`:

```
increment :: Int -> Int
increment x = x + 1
```

An application of `readCurry` to the file `example` returns the following data term:

```
CurryProg "example"
          ["Prelude"]
          []
          [CFunc ("example","increment") 1 Public
                 (CFuncType (CTCons ("Prelude","Int") [])
                            (CTCons ("Prelude","Int") []))
                 (CRules CFlex
                         [CRule [CPVar (0,"x")]
                                [(CSymbol ("Prelude","success"),
                                  CApply
                                    (CApply (CSymbol ("Prelude","+"))
                                            (CVar (0,"x")))
                                    (CLit (CIntc 1)))]
                                []])]
          []
```

It shows that the name of the program is `example`, the only imported module to this program is `Prelude` and there is no type declaration in the program:

```
CurryProg "example" ["Prelude"] [] ...
```

The program has only one function definition named as `increment` of type `Int -> Int`:

```
CFunc ("example","increment") 1 Public
      (CFuncType (CTCons ("Prelude","Int") [])
                 (CTCons ("Prelude","Int") [])) ...
```

The rule defined of this function has one variable in its left-hand side:

```
CRules CFlex [CRule [CPVar (0,"x")] ...
```

The right-hand side of this rule is an application of `x+1`:

```
CApply (CApply (CSymbol ("Prelude","+")) (CVar (0,"x")))
       (CLit (CIntc 1))
```

The generated data term is useful for manipulating Curry programs. We have used the AbstractCurry in our implemented tool iCODE.

To transform programs into FlatCurry, the following function is defined:

```
readFlatCurry  :: ModuleName -> IO FlatProg
```

In comparison to `CurryProg`, the data type `FlatProg` contains less information about the left-hand and right-hand sides of function definitions:

```
FlatProg :: ModuleName -> [ImportedModules] -> [TypeDeclaration]
     -> [FuncionDeclaration] -> [OperatorDeclaration] -> FlatProg
```

Let us compare the output of `readFlatCurry` to the previously generated data term by `readCurry`. Applying `readFlatCurry` to the file `example` generates the following data term for `increment`:

```
FlatProg "example"
     ["Prelude"]
     []
     [Func ("example","increment") 1 Public
           (FuncType (TCons ("Prelude","Int") [])
                     (TCons ("Prelude","Int") []))
           (Rule [1]
                 (Comb FuncCall ("Prelude","+")
                       [Var 1,Lit (Intc 1)]))] []
```

The information generated can be used to derive a flat representation [28] of a Curry program. In such a representation, patterns are translated to case expressions. This means that functions are defined by a single rule (the syntax of flat programs is discussed further in Chapter 6). Note that all predefined operators are also transformed to function applications. For instance, the operator `+` in the above flat form is transformed to a function call:

```
Comb FuncCall ("Prelude","+") [Var 1,Lit (Intc 1)]
```

In the AbstractCurry representation of our example, it is represented as a Curry application:

```
CApply (CApply (CSymbol ("Prelude","+")) (CVar (0,"x")))
       (CLit (CIntc 1))
```

FlatCurry can be useful for more back-end oriented program manipulation.

2.5 *Socket Connections*

Aspects of computer architecture, such as networking and distribution [69], are reflected in the architecture of stored data in computer systems [60]:

- *Centralized Systems*: All components of a personal computer (PC) are only accessible by one user at time. These computers are called centralized systems. Data that is located in a centralized system can be run on a single computer system. The other computer systems have no access to this data. The relationship between different components of a personal computer as a centralized system is presented in Figure 2.3.

- *Networking*: This architecture allows some tasks to be executed on a server system and some tasks on client systems (see Figure 2.4).

- *Distribution*: Data can be distributed across parts or sites in an organization. This architecture allows distributed data to be located where they are generated or most needed, but accessible from other parts or sites. Keeping multiple copies of data in different sites allows large organizations to have access to data even if one site is defected.

Since personal computers and networks have become faster and cheaper, using distribution and client/server systems have become common. A system that is considered to be a server should satisfy requests generated by client systems. The requests could be to create, update or read data. A general structure of a client/server system is presented in Figure 2.4.

Connection between a server and clients can be established by sockets. A socket generates a two-way communication link between two programs running on the network. Every socket generated by a server is bound to a specific port number that is a shared resource. To make a connection, a server listens to its sockets. Clients know the name

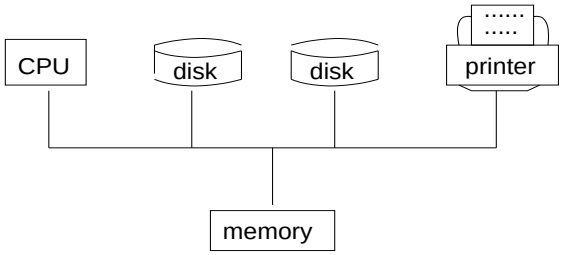


Figure 2.3: A centralized computer system

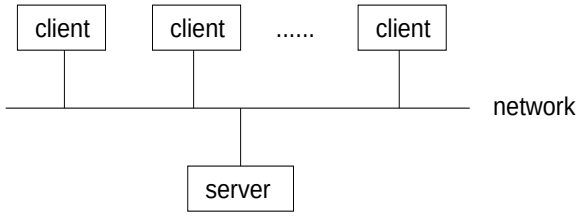


Figure 2.4: The relationship between a server and clients

of the machine or computer on which the server is running and the port number on which the server is listening. Each client needs to introduce itself to the server. For this purpose, clients use local ports (see Figure 2.5). Port numbers uniquely identify clients and servers and enable connections only between components that share these port numbers. Connections requested by clients can be accepted by servers. After an acceptance, the server knows the local port of the client. Listening to the next requests or messages from the same client can be done easily by the generated client-socket (see Figure 2.6).

Modules of a project written in Curry can be visualized as clients and servers that communicate with each other by socket connections. Listening to requests in Curry is easily defined by the following definition:

```
listenOnFresh :: IO (PortNumber,Socket)
type PortNumber = Int
```

By this function, a server creates a socket that is bound to a free port number. Returning this port number to a client allows the client to send a message by a connection to the generated socket:

```
connectToSocket :: String -> Int -> IO Handle
```

The first argument is the name of a machine or computer that is used for the server and the second argument is a port number. The abstract data type `Handle` is used like

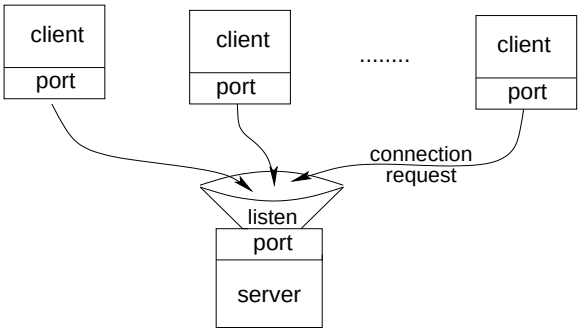


Figure 2.5: The requests for a connection from the clients

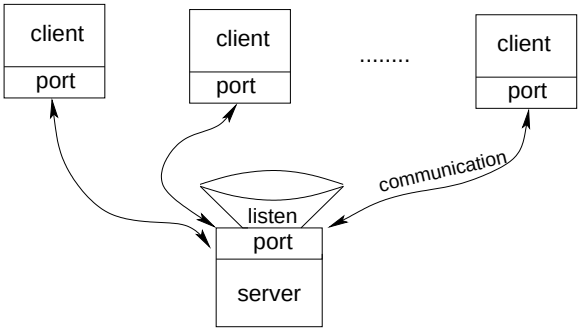


Figure 2.6: Communication through the generated sockets

memory for transporting requests and answers during communication.

To generate two-way communication, each request from a client should be accepted by a server:

```
socketAccept :: Socket -> IO (String,Handle)
```

Every acceptance from a server is returned as a string to identify a client and a handle to transport messages. The communication is continued by applying the two functions, `connectToSocket` and `socketAccept`.

We have used this technique to implement a run-time tracer for Curry. Clients and servers are different modules that communicate with each other by sending messages. The messages are information about evaluated expressions. Because sending a message can be delayed by suspending a socket-accept from a server, we have provided a single step representation for the user.

2.6 *Graphical User Interfaces and Tcl/Tk*

Tcl (Tool Command Language) is a string-based command language for controlling and extending applications. Tk (Tool Kit), a set of windowing commands, is one of the useful extensions to Tcl. Tk extends the main facilities of Tcl for building graphical user interfaces. Tcl and Tk together [51, 36] provide a programming system for developing and using *Graphical User Interface* (GUI) applications.

Every programming language should support programmers in the implementation of GUIs. The functional logic language Curry provides an easy interface implementation method by the programming system Tcl/Tk [24]. Many interesting GUI applications can be written using the functional semantics of Curry. The string based commands are hidden from the user and can be run by function definitions from the user.

An application generated by GUI library [24] has a wish (windowing shell). A wish is an empty window on a monitor and can read commands from standard input. Every wish can contain widgets. Widgets are parts of a wish with special appearances and behaviors. A wish can also be supposed as the main widget of an application.

Executing a widget is easy with the following function definition in the GUI library of Curry:

```
runGUI  :: String -> Widget -> IO ()
```

The first argument of this function defines a title of the main window. The second argument is a set of widgets of type `Widget`. By importing the GUI library to a Curry program and calling `runGUI`, we can generate a GUI. As an example, we consider two widgets `Label` and `Button`:

```
Label :: [ConfItem] -> Widget
Button :: (GUIPort -> IO()) -> [ConfItem] -> Widget
```

Widgets in Curry can be implemented with different configuration items (e.g., Background, Height, Width, Text) of type `ConfItem`:

```
data ConfItem = Background String
              | Height Int
              | Width Int
              | Text String
              | ...
```

The first argument of `Button` is a function that takes a command to change the state of widgets. For instance, we can use the two widgets `Label` and `Button` to generate a simple GUI (see Figure 2.7):

Figure 2.7: A simple GUI in Curry

```
main = runGUI "Hello World!"
              (Col[] [Label [Text "press the button"],
                      Button exitGUI [Text "Hello!"]])
```

Whenever the button is pressed, the command related to `exitGUI` is activated and the wish will disappear. Each wish also needs a geometry manager to manage the position of widgets. A vertical layout of widgets is defined by `Col` and a horizontal layout by `Row`.

We have used the GUI library of Curry in order to have different representation methods of observations in the implemented tool iCODE.

Chapter 3

Assuring Correctness of Programs

"What does a fish know about the water in which it swims all its life?"

Albert Einstein

In this chapter, we review the meaning of the useful testers, tracers and debuggers. They are provided for the purpose of developing software. Every part of a large project can be written by a member of a team. Understanding the code that is written by other team members and also assuring code accuracy are the aim of the tools mentioned.

3.1 *Bugs*

Programs are written to generate expected results. However, programs sometimes offer no results or unexpected results. This indicates that the code has *defects*. In other words, **code contains bugs**.

Defects are parts of code that have been written by programmers. Most parts of code are so related to each other that an execution of one part is reflected in other parts. A defect in one part causes related parts to behave incorrectly, although they contain no defects. We can say that **there is a bug in this state** of execution.

The wrong behavior of a buggy state causes a *failure*, which is an observable *error* in a program. It indicates that **there is a bug in the behavior of a program**.

Testers specify whether the behavior of programs is correct or incorrect. However, testers do not locate the true position of defects. In order to detect defects in code and fix them, we need debuggers. Stepping in evaluated parts of programs in a forward or

backward manner is a task for tracers.

3.2 *Testing*

When programming begins, the complexity of code is zero. As programming progresses, the complexity of various modules rises. Programmers must be sure that all parts of a code work properly. A programmer compiles programs after each manipulation of source code. The compilation permits the programmer to do more than get executable programs. Programmers test the compilability of code to find, for example, syntax errors or typing mistakes. Unfortunately, some errors are not detected by compilers. Programmers need other tools to assure the correctness of programs.

Before searching for failures, programmers should test programs to determine whether or not programs have failures. Most failures are usually found by testing programs. Testing is a process of executing programs with the intention of producing failures. However, we need a tester, because:

- we should control whether the results generated by a program are the same as the expected results.
- we should know whether a program generates a failure.
- we should verify whether fixing a failure (e.g., by a debugger) has been successful.

A program can be tested while it is (or is not) running by two methods. They are called:

- *Dynamic testing*: In this model, the tester requires units under test to be executed. Dynamic testers can be defined in two groups, white box and black box [44].
- *Static testing*: In this model [18, 9], the tester analyzes the software without running it. During a static analysis, the code is examined and information is accumulated. Such tools can determine the actual nesting of branch statements, the size of subprograms or functions, the unused code or variables and other interesting information for programmers.

  Static tester tools can be more useful for languages that have a type checking possibility at run time, particularly when type failures have occurred in the code (Section 2.1). This kind of failure in languages that controls types of expressions in the compile time can be found by compilers.

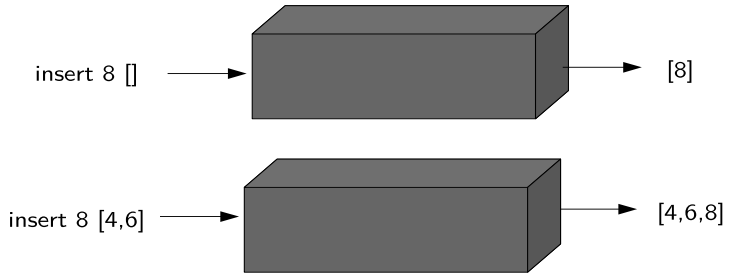


Figure 3.1: Black Box Testing

Let us consider the following simple program written in Curry. The program inserts an element into a sorted given list:

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
   | x>=y      = y:insert x ys
   | otherwise = x:ys
```

It is assumed that the second rule of `insert` has a defect. It causes a wrong answer when the inserted element is smaller than the first element of the given list.

To assure the accuracy of this program, we should test it. For this purpose, we run the program and test it by a testing tool that acts dynamically on the program. Running the program needs test cases. In black box testing, we can compare generated and expected results of given test cases. For example, generating the two test cases `insert 8 []` and `insert 8 [4,6]` offers `[8]` and `[4,6,8]` as the results (see Figure 3.1). What we had expected and the results are the same. It seems that there is no defect in the program. Let us test the program by using a white box tool and compare the behavior of these two testing tools. With a white box tester, we have access to the code under test. Figure 3.2 represents a white box tester for our two generated test cases.

The results of both test cases are what we had expected. However, the white box gives more information about executed parts of the program (the executed parts are highlighted in green). Note in both test cases that the last line of the program is not highlighted. In other words, it has not been executed. To assure the correctness of this part, we generate another test case: `insert 1 [4,6]`. The result of the running system is not correct while the last line is highlighted (see Figure 3.3). This means that all parts of the program are executable, but there is a bug in the program that causes the incorrect result. To locate this bug, we need a tracer/debugger. We will discuss these tools in the next section.

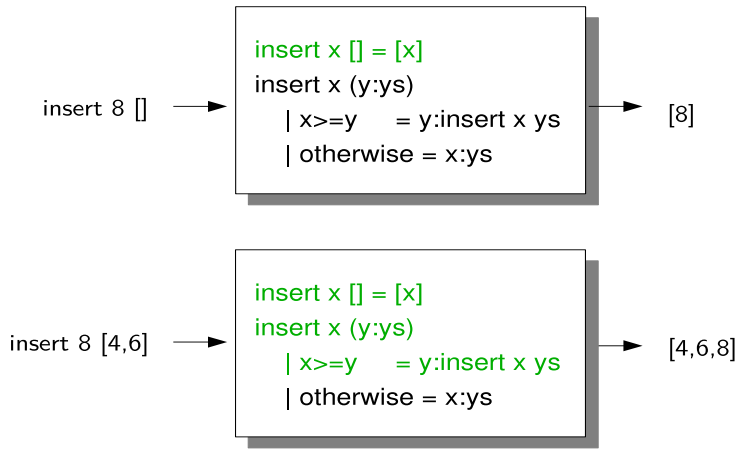


Figure 3.2: A white box testing of two generated test cases: `insert 8 []`, `insert 8 [4,6]`

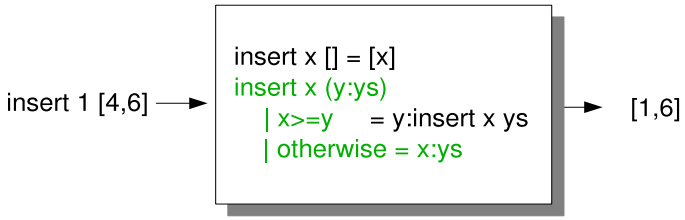


Figure 3.3: A white box testing of the test case: `insert 1 [4,6]`

Every implemented tester needs a test manager. With a test manager it becomes possible to define test cases and group them into test suites that can be run with one single user interaction. When a program execution has been completed, the execution results need to be checked for accuracy. This can be done manually or automatically:

- *Manual checks*: With this model, a programmer can compare the results of programs to her/his expectations. It is also possible for the tester to check the results of executions with the programmer's expected results. With this kind of testing tool [31], a programmer stores her/his expected outputs, together with test cases, and lets the tester check the equality.

- *Automatic checks*: With this model, the tool generates different test cases automatically [15, 39, 19] and compares the results of executions to predicates, which are the expected properties written by programmers.

One advantage of automatic testing is a reduction in the time necessary, which is a disadvantage of manual checking. Unfortunately, a programmer has no control on tests with an automatic generation of test cases in black-box testing tools. It can happen that some

parts of code are not executed. These parts could contain defects that are not detected by a black-box testing tool. For example, imagine that, for the last insertion sort program, the two test cases (`insert 8 []` and `insert 8 [4,6]`) are generated automatically and in a black-box representation. By controlling only the result of the program execution, we will have no idea that our program has a bug, because we had no control of the generated test cases and no access to the units under test.

Let us discuss this problem by an example. The following simple program generates a non-terminating computation when one of the given list elements is not greater than zero:

```
inc :: Int -> Int
inc x = if x>0 then x+1 else inc x

main :: [Int] -> [Int]
main l = map inc l
```

The program is written to increment all elements of a given list that are positive. We would like to test the program by a black-box automatic testing tool. Now assume that the tool automatically generates 100 different lists and applies the function `main` to all of these lists. In other words, 100 tests are generated automatically. The tool also offers correct results for all of the 100 test cases, because there is no element smaller than zero in the generated lists. Should we trust such tools? Now assume that we test the program by a white-box manually testing tool. After writing a test case, the tool represents the executed parts of the program. It makes sense that new test cases should be written to test non-executed parts of the program.

Another problem that could arise in an automatic testing tool is that the use of expected properties may be incomplete and contain errors. As a simple example, we test the following function, which reverses a given list:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

This function can satisfy the following properties:

```
reverse [x]          = [x]
reverse (xs++ys)     = reverse ys ++ reverse xs
reverse (reverse xs) = xs
```

To test the program by a tester that generates test cases automatically (e.g., QuickCheck [15]), we need to add some of these properties to the program. For example, we write the second property into the program (the name of a property function begins with `prop_`):

```
prop_RevApp xs ys = reverse (xs++ys) == reverse ys ++ reverse xs
```

To test this property, the tool needs to know the type of the arguments. Thus, we must specify a type for arguments on which the property is tested. For example, we test the property for a list of integers:

```
prop_RevApp :: [Int] -> [Int] -> Bool
```

Now the program with the inserted property can be reloaded and tested. The tool takes the property and applies it to a number of automatically generated arguments. If results for generated test cases are correct, the tool reports that the tests were successful.

Properties are parts of code that are written by programmers. Every code implemented by programmers can contain defects. For example, if we define a wrong property like the following:

```
prop_RevApp xs ys = reverse (xs++ys) == reverse xs ++ reverse xs
```

the tool will report that the property was not successful for some test cases. Now the question involves the location of the defect. Is it in the original code or in the defined property?

In this thesis, we represent the tester `CUTTER`. It is a white-box tester and uses the program slicing method (Section 3.5.2) to highlight executed parts of programs. Test cases generated by programmers can be grouped and stored for rerunning the tester (e.g., after manipulating source code). All other manipulations needed are performed automatically by the tool.

3.3 *Debugging*

Programmers need debuggers [73] to frequently test each program state to locate defects. Values of evaluated expressions in each state must be represented by an understandable method. If debuggers do not report understandable information, programmers are misled and may spend hours in tracking down the wrong bug.

Briefly, we can say that debuggers should support programmers by the following facilities:

- The act of debugging programs must not change the behavior of programs.

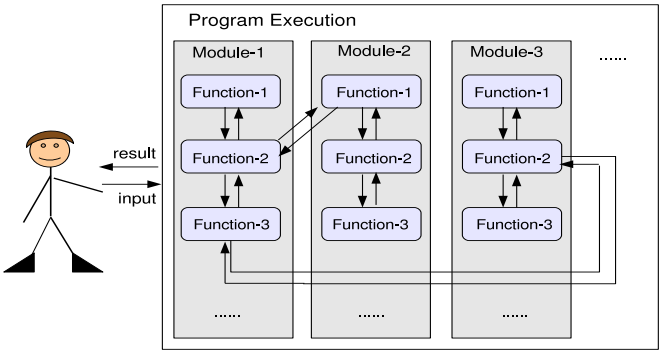


Figure 3.4: The relationship between functions

- Debuggers must not mislead programmers. In other words, programmers should always be able to trust debuggers.
- Intermediate information from evaluations should be represented so that the trail of evaluations is recognized and the user will know which expression was evaluated and when.
- The best representation might be a display of the information in a layout of source code.

Every programmer makes mistakes. Writing programs may be easier than finding mistakes in code. Programmers spend too much time in writing complex programs, but would like to have a tool to use to find the reason for failures.

In functional programs, there are a variety of functions that are related to each other by the exchange of values or variables. A programmer is stated in the outer layer of a program execution (see Figure 3.4). Every buggy result from a function sends out the rest of the program execution in the direction of the wrong generation. It may happen that only one function in a large project is defined incorrectly. The result of this function causes related functions to generate the faulty results.

Debuggers should help users to follow related functions. They should also represent the activity of each executed function definition to the user. Representing this information can be provided during or after a program execution. Conventional debuggers provide this information after a program execution. Debuggers that represent this information during a program execution are called run-time debuggers (see Figure 3.5). This kind of debugger needs a controller for source-lines executions. For this reason, most run-time debuggers modify code so that every state of code can be checked at run time. They keeps track of program's expressions and immediately complain if the programmer misuse an expression. Run-time debuggers can provide useful information about:

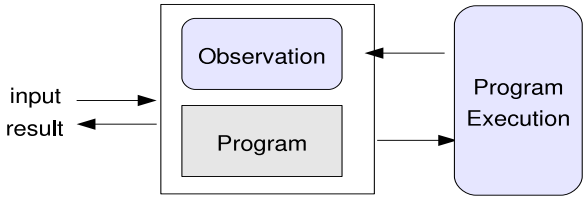


Figure 3.5: Run-time Debugger

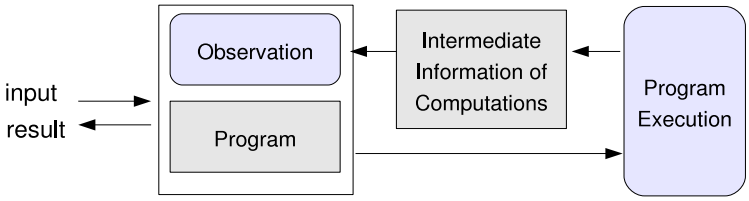


Figure 3.6: Debugger

- non-terminating computations
- run-time failures
- points of interactions between a program execution and standard output

Many debuggers represent intermediate information of computations after terminating the program execution. During execution, intermediate information is recorded as a relationship between function computations (Figure 3.6). It is clear that it is expensive to record all computation steps of a large program:

- a program execution will slow-down
- it causes a shortage of memory

To avoid the problems mentioned above, different methods are suggested. Some of these methods are:

- recording special states of executions (e.g., the state of some desired expressions or only branches).
- recording information of specific parts of programs (e.g., by check-points).

In this thesis, in order to avoid the problems mentioned above, we suggest a distributed programming method. In this method, the information of executions is not recorded in a file. Instead, it is sent as messages from different modules in a run-time debugging system.

Further, the first solution above is also used in our implemented tool. The information of computations does not belong to all program expressions. It is information about special states of executions that are affected by executing selected user expressions.

The goal in using a debugger is to detect all failures in programs. But how do we know, how much of our code has been controlled? Some debuggers support a *program-coverage* tool which is used to analyze the amount of the source code that has actually been checked. We have also provided this ability in our implemented debugging system.

3.4 *Tracing*

Programs are expressions that are written one after another by a programmer. Particularly, declarative programs are expressions that are written without consideration to the order in which they will be executed. Failures arising during program execution forces programmers to think about the order of evaluations. Having tools that support programmers following the evaluation order of expressions is desirable. Sometimes, there is also a need to review represented information to understand the relationship between program expressions. Tracers are provided for this purpose. A tracer acts like a cassette recorder (see Figure 3.7):

- *Play*: runs a program and represents side effects and intermediate results of evaluated expressions.
- *Pause*: stops a program execution whenever desired and lets it resume running when needed.
- *Stop*: ends a program execution.
- *Reward*: tracks back on executed parts from a point at which the program execution was stopped.
- *Forward*: follows a program execution from a point at which a reward stepping was stopped.

The only difference between real cassette recorders and tracers is the act of the reward button. Execution systems and tracers are two separate tools. Tracers cannot reward a real execution process. For this purpose, tracers store information about execution paths and provide users with the means to backtrack on this recorded information. We have also provided such a means in every viewing tool of our implemented observation system.

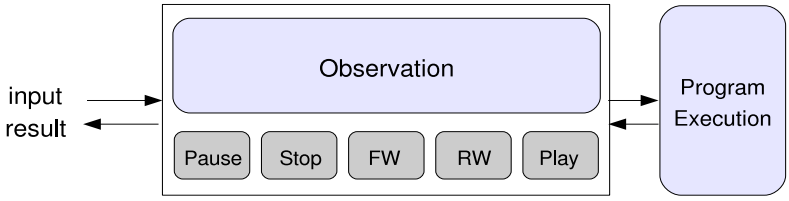


Figure 3.7: Tracer

3.5 *Some Debugging/Tracing Methods*

A user has successful use of a debugger/tracer if she/he at least knows that a bug exists and has an idea where it may be contained (e.g., by a white-box tester). However, representing all information of computations to the user requires a great deal of time. Therefore, it might be useful if the user had an ability to select parts of programs or computations to test or trace. There are a variety of methods to record a dependency of intermediate information of computations from parts or whole of programs and to step in this information. An overview of some existing debugging/tracing methods could be helpful to determine their advantages or disadvantages.

3.5.1 *Declarative/Algorithmic Debugging*

Algorithmic debuggers work by means of the declarative semantics of programs. These debuggers generate information of computations as a *computation tree* [59]. Nodes in computation trees are defined by equations representing function applications. Children of nodes are applications that are generated from the related function definition. Algorithmic debugging can produce a sequence of questions to navigate in computation trees. Each question asks the user whether an equation is correct or not. A bug could be located in a function definition of a node when all children of an incorrect equation are correct. Unfortunately, such navigations that are directed manually and by answers from the user are not trustworthy, because, if the user answers incorrectly, the navigation could be sent in a wrong direction. On the other hand, a part of the program would not be examined, although it contains a defect. Further, answering a long series of questions during debugging a large program could be boring for programmers.

Some methods are suggested [50, 61, 48] to reduce the number of questions from the user. In [61], the algorithmic debugging is combined with the program slicing method [72, 49, 62]. In [48] another idea is presented. The idea is to generate an evaluation dependency tree during a program execution and browsing of this tree by the user. The user can begin debugging from a node of interest.

There is also another technique for optimization of declarative debuggers [46, 47]: It does not require all parts of a program to be traced. For large computations, we can trace only a part of a program by demand.

Let us illustrate the process of algorithmic debugging by the following simple example:

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
   | x<=y      = x:ys
   | otherwise = y:insert x ys

sort :: [Int] -> [Int]
sort []     = []
sort (x:xs) = insert x (sort xs)

main :: [Int]
main = sort [2,1,3]
```

The program is written to sort a list of integers in based on `insertion sort`. We assume that the function `insert` has a defect. When the element inserted in the list is smaller than the head of the list, the function offers `x:ys` instead of `x:y:ys`. The program does not work properly. Calling the function `main` returns `[1,2]`, instead of `[1,2,3]`. We would like to locate the position of the bug by an algorithmic debugger.

The algorithmic debugger generates a computation tree while executing the program. The computation tree in Figure 3.8 represents the relationship between function applications when the function `main` is called.

Generating the computation tree is based on the derivation system of the program rules. This derivation in our example is represented as follows:

```
main =>  sort [2,1,3]
     => insert 2  (sort [1,3])
     => insert 2 (insert 1  (sort [3]))
     => insert 2 (insert 1 (insert 3  (sort [])))
     => insert 2 (insert 1  (insert 3 []))
     => insert 2  (insert 1 [3])
     =>  insert 2 [1]
     => [1,2]
```

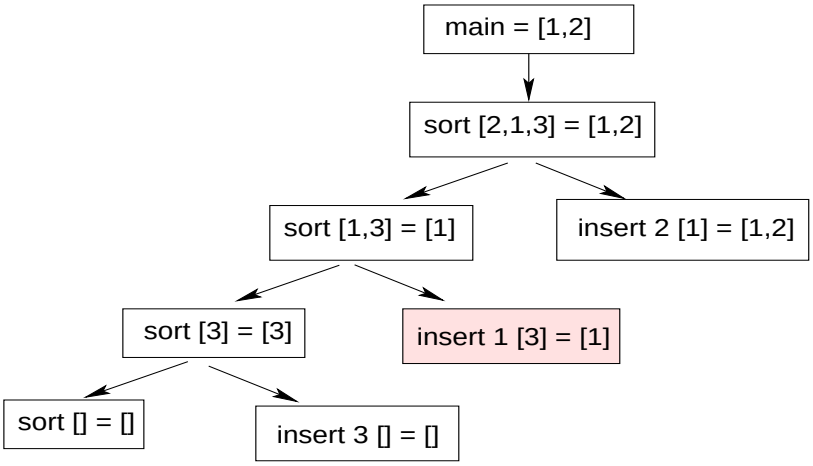


Figure 3.8: Computation Tree

After recording intermediate information as a computation tree, the navigation can be started at the root of this tree. A debugging tool that is implemented by the algorithmic debugging method first asks us whether the final result from the execution process is correct. We answer negatively:

```
main = [1,2]? no
```

Now, the children of this node must be examined:

```
sort [2,1,3] = [1,2]? no
```

The bug should be located either in the left sub-tree of this node or in the right sub-tree. The debugger asks the next question and we answer it:

```
sort [1,3] = [1]? no
```

The navigation is continued:

```
sort [3] = [3]? yes
```

The equation is correct. At this moment, a navigation of the sub-trees of this node is not required. The tool should know whether the right sub-tree of the last faulty node is correct. It is not:

```
insert 1 [3] = [1]? no
```

This node has no children. The bug is isolated in the highlighted node of the tree (see Figure 3.8).

3.5.2 *Program Slicing*

Program slicing [72, 49, 62] is another method to record a dependency between computations. It is a technique for simplifying programs by focusing on some selected aspects of semantics. The process of slicing deletes parts of a program that have no effect on the desired parts that were selected by the programmer. Deleting these parts and representing a small part of a program helps the user to focus on a small environment to better locate bugs. Represented parts of a source code is named as a *static slice*. Note that executing a program by giving special values can represent another slice to the programmer. Such a slice is called a *dynamic slice*. Static slices are usually larger, but sufficient for every possible execution of the program. Dynamic slices are smaller, but suffice for a single input or value.

By the use of debugging tools that are implemented with the program slicing method, the user can identify an arbitrary expression at some points of a program to define a *slicing criterion*. This is a subset of the program being debugged. After determining this subset, the tool extracts a fragment of this program that contains values related to the expression at that points. The extracted fragment is the program slice mentioned above.

Each program slice is either a *backward slice* or a *forward slice*. A backward slice contains parts of a program that can have effects on the slicing criterion, whereas a forward slice contains parts that are affected by the slicing criterion.

For example, if we select the expression `main` from the last insertion sort example, the program slice that is executable will belong to the last line of the program (`main = sort [2,1,3]`), because the expression `main` is appeared in the right-hand side of no rules. In other words, executing `main` affects other parts of the program, but executing another function (e.g., `sort`) has no effect on `main`.

Let us discuss the program slicing further by means of an example. In the following program, we have defined three simple functions written in Curry:

```
f :: [Int] -> [Int] -> ([Int],[Int])
f xs ys = let ls  = restOfList xs
              ls' = headOfList ys in
          (ls,ls')
restOfList :: [Int] -> [Int]
restOfList []     = []
restOfList (x:xs) = xs
headOfList :: [Int] -> [Int]
headOfList []     = []
headOfList (x:xs) = [x]
```

The function `f` takes two lists and offers a pair of lists. The functions `restOfList` and `headOfList` are applied to two given lists to generate pairs. The components of the pairs are the rest and first elements of the given lists.

To extract a slice, we select the variable `xs` in the function `f`. The program slice related to this criterion should be represented and other parts should be deleted. For this purpose, we represent the program slice darker than the deleted parts as follows:

```
f :: [Int] -> [Int] -> ([Int],[Int])
f xs ys = let ls  = restOfList xs
              ls' = headOfList ys in
          (ls,ls')

restOfList :: [Int] -> [Int]
restOfList []     = []
restOfList (x:xs) = xs

headOfList :: [Int] -> [Int]
headOfList []     = []
headOfList (x:xs) = [x]
```

The variable `xs` is an argument of `restOfList` that generates `ls` as a result. This means that the program slice can be represented without the unused parts of the program. The static slice for the selected criterion is represented as follows:

```
f :: [Int] -> [Int]
f xs = let ls  = restOfList xs in
         ls

restOfList :: [Int] -> [Int]
restOfList []     = []
restOfList (x:xs) = xs
```

To represent a dynamic slice, we need a value for execution of the program. Assume that the variable `xs` in the function `f` is bound to an empty list (`xs=[]`). Now the slice related to the criterion is not the same as the last static slice, because the second rule of the function `restOfList` has not been executed:

```
f :: [Int] -> [Int] -> ([Int],[Int])
f xs ys = let ls  = restOfList xs
```

```
            ls' = headOfList ys in
          (ls,ls')

restOfList :: [Int] -> [Int]
restOfList []     = []
restOfList (x:xs) = xs

headOfList :: [Int] -> [Int]
headOfList []     = []
headOfList (x:xs) = [x]
```

and the simplified program in the dynamic slice is smaller:

```
f :: [Int] -> [Int]
f xs = let ls  = restOfList xs in ls

restOfList :: [Int] -> [Int]
restOfList []     = []
```

In this thesis we use the dynamic slicing method to represent executed parts of programs by highlighting executed expressions for each criterion selected by the user.

3.5.3 *Redex Trail*

Redex trail [65] is a tracing method that provides an ability for backward step on the recorded information of computations. A backward stepping results from a failure of an initial expression that locates a bug. The redex trail system generates a graph of computations. Nodes of this graph are related to values or function applications. Each node has a link that is directed to its parent. Parents are also function applications. Because sub-expressions of a function application could be used by other nodes, the graph's function applications may have links to shared sub-expressions.

Let us consider a simple example to see how a redex trail is generated. To keep the result small, we select the following program that returns the sum of all elements of a given list:

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

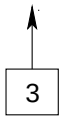


Figure 3.9: The first generated node of redex trail

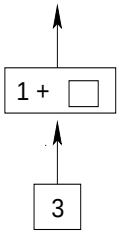


Figure 3.10: The first and second generated nodes of redex trail

The application of `sum` to the list `[1,2]` returns `3` as the result. The redex trail system starts with the result and generates a graph. The first step of this generation is presented in Figure 3.9. The result is generated from `x+sum xs` that requires a recursive call to `sum` (see Figure 3.10). The result of the recursive call is obtained in the empty box when the graph reaches the right-hand side of the first rule. Figure 3.11 represents the complete graph generated for our example.

Debugging a large program causes a large redex trail to be generated. Showing all details of computations is not always necessary. Further, it is also possible to select some parts of interest to examine [66].

Of course, the user is not expected to see and understand a complete graph. Implemented tools [71] that use redex trail need viewers to represent such information in an understandable manner to the user. For example, the trail of the last example by calling `main` is represented in Figure 3.12. The redex-trail system shows the program output (or a possibly error message) and enables exploring the program computation backwards. In our example, the result of `sum [1,2]` is `3`. The parent of this result is `1+2`. The subexpression `2` in this redex is constructed by computing `2+0` where `0` is a result of `sum[]`. Meaning that in the backward presentation of computations, the first generated trail is `sum []`. The parent of this expression is `sum [2]` and so on (see Figure 3.12).

3.5.4 *Observing Intermediate Data Structures*

Sometimes, programmers would like to inspect the results of special data structures or function calls in the right-hand sides of function rules. To find out what is actually

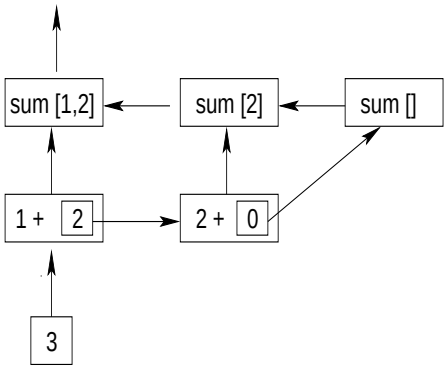


Figure 3.11: The generated redex trail

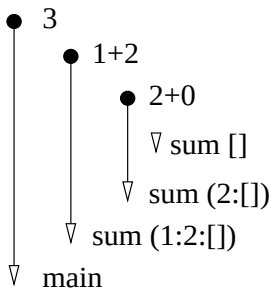


Figure 3.12: The trail of `sum [1,2]`

happening in a concrete failing run, we should observe what is going on and judge whether the values generated are faulty or not. For this purpose, we should consider different principles as follows:

- whatever we observe should be an effect of the original run and program results. Otherwise, it would be difficult to find out a reason for the program failure.
- a program execution is a large sequence of huge program states. An observation is effective if we know which parts of a program state should be observed.
- a program expression might have different values at different moments of program execution. To have an efficient observation, we should know at which moments of execution the expression should be observed.

One of the simplest way to observe results of program expressions is using *print/trace* statements. Unfortunately, inserting only a print/trace statement into a program is not sufficient. Failures may be located in an execution path of a program expression. It is necessary to insert another print statement and re-execute the program. If the programmer is an expert and has experience, it might be easier to find positions where print

statements should be inserted. The programmer can point her/his finger to an expression and indicate that this expression should be examined. However, if the programmer has never experienced this kind of failure, she/he has also no experience in isolating a failure. It is clear that finding an area that contains a failure would be difficult. Furthermore:

- print/trace statements do not help us in understanding the code.
- we can not observe the behavior of function calls in programs using print/trace statements. It is only possible to observe the results generated of functions.
- inserting even more than one print statement into the program does not help us to observe the relationship between executed expressions or function arguments.

The above features are provided in observation debugging systems [20, 10, 57].

The observation system [20] defines an identity function (`observe`) that could be inserted in each program expression. Results of this identity function show how an expression is evaluated. In comparison to the print statement that writes generated information to the standard output, `observe` records information of computations in a file. Recorded information belongs to expressions annotated by observer functions and not to the whole program expressions. In other words, if a partial application of an expression is annotated, only the partial evaluation is recorded in the file.

For example, we assume that the function `head` is applied to the list `[1,2,3]` in a program:

```
main = head [1,2,3]
```

Writing a print statement in the program only offers the result of `head`. However, we would like to know how `head` is applied to the list and how it is computed. Annotating `head` by an `observe` function returns the following information:

```
\(1:_) -> 1
```

This shows how the function `head` returns the first element of the given list without executing the remaining elements of the list that are represented by an underscore.

Before representing this information, it is necessary to follow the steps of derivation and record the information in a file. The derivation is started by executing the function `head`:

```
<start>
```

The function `head` has a left-hand side and a right-hand side that should be executed:

```
\<LHS> -> <RHS>
```

Executing the left-hand side of the function results in a list:

```
\(<headOfList>:<tailOfList>) -> <RHS>
```

In the next step of the derivation, `<headOfList>` is executed and evaluated to `1`:

```
\(1:<tailOfList>) -> <RHS>
```

Now the result is obtained and represented as the right-hand side:

```
\(1:<tailOfList>) -> 1
```

Note that `<tailOfList>` is not executed, because of the lazy semantics of functional languages.

The observation method represented in [20] is also extended in [53] and [10] for functional logic languages. In this thesis, we use the observation method for improving COOSy [10] to acquire an interactive observation tool called COOiSY [57]. The technique above, in combination with the program slicing (Section 3.5.2), helped us to implement the runtime tracer C-Ace for Curry programs. Each step of the evaluations can be represented to the user in a single step mode (Section 8.1.2). The single step representation, besides highlighting (Section 7.2) dynamic slices (Section 3.5.2), is useful to understanding of the lazy semantics of the functional logic language Curry.

3.6 *Existing Debuggers/Tracers for Curry*

The multi-paradigm language Curry combines all aspects of functional languages with logical languages. A debugger/tracer for this language should support both deterministic and non-deterministic features of programs. Different debuggers/tracers are implemented for this language. We briefly describe them:

- *CIDER*: This tracer [30] is a graphical environment with a visualization of the expression evaluation tracing method. It is implemented by using meta-programming and GUI programming libraries of Curry. CIDER contains a program editor, tools to analyze function definitions in programs, a tool to draw dependency graphs and a graphical debugger. CIDER shows information of the computations for every selected function from the user. The information is represented after analyzing selected functions as a textual result or as a graph dependency. The tool displays a trace of computations as a sequence of narrowing steps. Unfortunately, such a

sequence makes it difficult for the user to trace non-deterministic computations. CIDER shows non-deterministic steps as a backtracking on represented steps. This means that, if one sequence gives no solutions, another sequence or path should be backtracked. An evaluation fails when no path leads to a result. Following steps in a large computation could be lead to the loss of orientations. This is another problem that encountered in using this tool.

- *Münster*: This debugger [43] is implemented with the declarative debugging method (Section 3.5.1). While executing a program, the tool records information about computations in an execution tree. After navigating the tree, one answers a series of questions. Münster asks the user about the correctness of computations. The user should answer yes or no. Answering these questions orients the navigation on the tree until the failure is found. Unfortunately, if the user answers incorrectly, the tool will lose orientation and represent incorrect information about the failure.

- *COOSy*: This debugger [10] is an extension of [20] with supporting features of functional logic languages that are non-deterministic computations and logical variables. It is implemented by the observation debugging method. The tool shows the results of every data structure or function annotated by the user. COOSy does not represent the order of evaluations and cannot support the ability to backtrack on the evaluations.

  Because one of the research contributions of this thesis is related to the observation system in COOSy, we discuss it further in Chapter 4.

- *TeaBag*: Like CIDER, this debugger [5] uses the expression evaluation tracing method. TeaBag represents computation information in two parts. One part is a tree structure of computations. Another part is a linear sequence of steps for one path through a tree structure. Each path from a parent to a child shows a sequence of deterministic steps. TeaBag lets the user set breakpoints on functions. This feature is useful for large computations. By setting breakpoints, only the steps of executed expressions between breakpoints will be represented to the user.

Chapter 4

The Observation System

"The secrets eternal neither you know nor I,
And answers to the riddle neither you know nor I,
Behind the veil there is much talk about us,
When the veil falls, neither you remain nor I."

Omar Khayyam

Curry is a declarative language that computes program expressions by the lazy evaluation method. In this method, the order of evaluations is not the same as with the order of program expressions. However, programmers do not think about the evaluation of every program expression during run time. They write programs for the expected outputs. For example, consider the following program. The function `main` is a combination of functions: `reverse`, `map`, `increment` and `filter`.

```
increment :: Int -> Int
increment x = x + 1

reverse :: [Int] -> [Int]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

main :: [Int] -> [Int]
main xs = let ys = filter (<5) xs
              zs = map increment ys in
          reverse zs
```

After applying the function `main` to the list `[1,8,3]` in our example, the first expression

that would be computed is `reverse zs`. To compute this expression, Curry needs the value of `zs` and to evaluate `zs` the expression `map increment ys` must be executed. The result of this expression is found when the value of `filter` is offered as `ys`. In other words, the evaluation is started with the expression `reverse zs` but the first computed expression is `filter (<5) xs`:

```
main [1,8,3]
        => reverse zs
        => reverse (map increment ys)
        => reverse (map increment (filter (<5) [1,8,3]))
        => reverse (map increment [1,3])
        => reverse [2,4]
        => [4,2]
```

During the writing of the program, we do not consider the evaluation order of expressions. This means that, if the program offers an unexpected output, which function or expression has not been defined correctly will not be evident: `filter`, `map`, `increment`, `reverse` or a (sub-)expression.

The reason for observation systems like [20, 10, 57] is to represent values of functions and data structures with respect to lazy evaluation. Values are represented to the user by manipulating programs before executing code.

4.1 *Applications*

By using the observation system COOSy [10] in different examples in this section, we discuss how the behavior of program expressions can be inspected.

Values of program expressions can be observed by adding the function `observe` to the corresponding program expression, while the module `Observe` is imported into the program:

```
observe :: Observer a -> String -> a -> a
```

The `observe` function does not change the natural behavior of programs. It is an identity function on the third argument. The second argument of the function `observe` is a label to distinguish different observations. It is printed only as a side effect. The function `observe` in [20] needs a type class restriction on the expression being observed. Every type class defines a constructor that can be applied to a type variable. The type class can also contain a function that can be applied to all instances of the class that are defined

by the programmer. Curry, in contrast to Haskell [35, 33], does not provide type classes [22, 16]. Therefore, to use the function `observe` in Curry programs, we need to add special observers as the first argument. Observers are related to the type of the observed values. For each predefined base type τ (e.g., Int, Float, Char), the observer is defined as $o\tau$. For example for an expression of type `Int` the observer `oInt` should be used and for `[Int]` the observer `oList oInt`. Let us discuss the behavior of observers in different examples.

Example 1 (*Expressions*): Consider the last program discussed. At first, we observe the value of `map increment ys` in the right-hand side of the function `main`. The type of this expression is `[Int]` and the related observer for this type is `oList oInt`:

```
main xs = let ys  = filter (<5) xs
              zs = observe (oList oInt)
                           "result"
                           (map increment ys) in
          reverse zs
```

The manipulated program should be loaded and executed. We apply the function `main` to the list [1,8,3]. The result of the observation system for the label `result` is as follows:

```
result
------
[2,4]
```

Example 2 (*Function Applications*): It is also interesting to observe the behavior of defined functions of a program. Observers for functions can be constructed by means of the right associative operator:

```
(~>) :: Observer a -> Observer b -> Observer (a -> b)
```

In the example above, we can also use a functional observer for the function `map`. The function `map` is a predefined function in the Curry libraries with the following type definition:

```
map  :: (a -> b) -> [a] -> [b]
```

It applies a function of type `a -> b` to all elements of a list of type `[a]`. To observe `map`, we need an observer for this type of function. Since the program is defined as applying to integers, the type of the function `map` could be specialized to:

```
map  :: (Int -> Int) -> [Int] -> [Int]
```

Now, we can define an observer and add the function `observe` to the program:

```
main xs =
    let ys  = filter (<5) xs
        zs = observe ((oInt ~> oInt) ~> oList oInt ~> oList oInt)
                     "map-result" map increment ys in
      reverse zs
```

This time, calling the application `main [1,8,3]` represents the result of `map` as following:

```
map-result
----------
{\{\3 -> 4
  ,\1 -> 2
  } [1,3] -> [2,4]}
```

Function definitions are represented as `{\arguments ->result}`. The information represented for the label `map-result` indicates that the function `map` offers `[2,4]` as the result and has two arguments: `{\3 -> 4 , \1 -> 2}` and `[1,3]`. The first argument is the result of the function `increment` that is called twice for two different list elements 1 and 3. The second argument `[1,3]` is the result of the function `filter`. The elements of this list show that the integer 8 has been filtered from the input list.

Example 3 (*Polymorphism*): The functions `map` and `filter` are two predefined functions in Curry. They have polymorphic types so that these functions can be applied to different lists with different type definitions.

Annotating expressions by observers has a benefit. It is possible to use different observers for the same type. This allows selective masking of substructures in large observed data structures (e.g., by the predefined observer `oOpaque` that represents every data structure by the symbol `#`). To show this opportunity in our example, we change the observer of `map` to a polymorphic type:

```
main xs =
    let ys = filter (<5) xs
        zs = observe ((oOpaque ~> oOpaque) ~> oList oOpaque
                                           ~> oList oOpaque)
                     "map-result" map increment ys in
      reverse zs
```

After re-executing the annotated program, the result of the function `observe` from the new manipulated program without considering the type of the list elements is:

```
map-result
----------
{\{\# -> #
  ,\# -> #
  } [#,#] -> [#,#]}
```

Example 4 (*Lazy Behavior*): We can also annotate expressions to understand the lazy behavior of a program. For this purpose, we apply the function `head` to the result of the above example:

```
firstElem :: [Int] -> Int
firstElem xs = head (main xs)
```

In applying the function `firstElem` to a given list (e.g., [1,8,3]), only the first element of the list is computed and the values of the remaining list elements are not evaluated because of the lazy evaluation strategy (Section 2.2.9). To see this behavior of the function `head`, we add an observer to this function:

```
firstElem xs = observe (oList oInt ~> oInt)
                       "head-result"
                       head (main xs)
```

The observation system represents:

```
head-result
-----------
{\(4:_) -> 4}
```

Note that the first element of the list is evaluated to a value and the remaining elements of the list are not computed. The un-evaluated expression is represented by an underscore sign.

Example 5 (*Non-Determinism*): Curry is a functional logic language. In Curry programs, functions can be defined non-deterministically [6]. This means that functions can return different results for given sets of arguments. In other words, a function can be defined with different rules so that rules with different right-hand sides have equal left-hand sides (Section 2.3.1).

As an example, let us consider the following program that performs a non-deterministic computation:

```
add :: Int -> Int -> Int
add x y = x + y
main :: Int
main = add (0?1) 1
```

The expression `(0?1)` yields 0 or 1. This means that we could also define the function `main` by two different rules:

```
main = add 0 1
main = add 1 1
```

To understand the behavior of non-deterministic computations, we apply an observer to the function `add` in the right-hand side of `main`:

```
main = observe (oInt ~> oInt ~> oInt) "add-result" add (0?1) 1
```

Calling the function `main`, we obtain:

```
add-result
----------
{\0 1 -> 1}
{\1 1 -> 2}
```

The function is called twice and offers two results, 1 and 2. Because of the non-deterministic behavior of the program, some parts of the first computation are shared by the second one. These parts are highlighted by a lighter color that helps to understand the non-deterministic computations. Note that if we change the order of arguments for the function `add` as follows:

```
main = add 1 (0?1)
```

we obtain:

```
add-result
----------
{\1 0 -> 1}
{\1 1 -> 2}
```

The first argument is shared for the second computation of `add`.

Example 6 (*Logical Variables*): Curry also supports uninstantiated variables in programs. These variables or logical variables can also be observed. To see this, consider the following program:

```
add :: Int -> Int -> Int
add 0 1 = 1
add 1 1 = 2
main :: Int
main = add x 1
   where x free
```

In this example, the variable `x` is defined logically (Section 2.3.2). We would like to observe the behavior of `x` when the function `main` is called:

```
main :: Int
main = add (observe oInt "x-result" x) 1
   where x free
```

By executing the annotated program we obtain:

```
x-result
--------
(?/0)
(?/1)
```

After computing all possible solutions, Curry binds the free variable `x` to the two values, `0` and `1`. The logical variable is represented by a question mark and the values that the variable is bound to. The highlighted question mark shows that the free variable `x` is bound non-deterministically.

Example 7 (*Faulty Program*): Observing functions and expressions helps programmers not only to understand the behavior of programs, but also to locate bugs in programs. Let us consider a faulty program to find the position of the failure. The following program computes the largest element of a given list:

```
applyMax :: [Int] -> Int
applyMax xs = foldl max 0 xs
max :: Int -> Int -> Int
```

```
max x y
  | x > y = x
  | x < y = y
```

After applying the function `applyMax` to the list [2,3,1,3], we expect to have 3 as the largest element. Instead, the program gives "No more solutions". There is a bug in the program. In order to locate this bug and show the behavior of the function `max`, we add an observer in the right-hand side of `applyMax`:

```
applyMax xs = foldl (observe (oInt ~> oInt ~> oInt)
                             "max-result"
                             max) 0 xs
```

The manipulated program should be loaded and executed. Applying the function `applyMax` to the list [2,3,1,3] activates the observer. The observation system represents the following information:

```
max-result
----------
{\3 3 -> !
,\3 1 -> 3
,\2 3 -> 3
,\0 2 -> 2
}
```

The function `max` is called 4 times. The result of this function in the fourth call is represented by the exclamation mark. It indicates that the evaluation has started, but that the computation has not been completed. This happens when the two arguments of `max` are equal `{\3 3 -> !}`. We examine the program again. There is no rule definition for equal values. The bug is located.

In all of these examples, we have seen how, by adding observe functions to program expressions, the observation system represents the behavior of expressions to the user.

Unfortunately, adding or removing observer functions to or from programs, and also defining appropriate observers, are efforts that discourage programmers from using observation debuggers. We need a tool to generate all observers automatically and add annotations automatically to program expressions. This ability is provided in the implemented tool iCODE. The user should only select arbitrary expressions that are represented in the tool's browser.

4.2 *Generating Observers*

The purpose of debuggers is to locate bugs in programs. Bugs can be generated because of wrongly defined functions or buggy expressions in programs. We would like to inspect the behavior of functions and data structures with respect to the observation systems. As we have described in the last section, for every observed expression in Curry programs, we need an observer that is related to the type of expression. This means that, for an automatic generation of observers, we should infer the type of expressions from a program and also generate special observers for observed expressions.

In this section, we show how we can generate observers for different kinds of program expressions.

4.2.1 *Observing Functions*

The most important feature of a convenient observation tool is the observation of top-level functions. In Curry, these functions can be defined by one or more rules and may behave non-deterministically. The reason to observe such a function should be that every call to this function is observed. The easiest way to realize this behavior is to add a wrapper function that adds the observation to the original function.

By the help of the meta-programming library (see Section 2.4), we infer the type of functions. This requires only a converter to convert each base type τ to the observer $o\tau$. Consider the following simple program that computes the reverse of a given list:

```
reverse :: [Int] -> [Int]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

In this example, an observation of all calls to `reverse` can be obtained as follows:

```
reverse = observe (oList oInt ~> oList oInt)
                  "reverse"
                  wrapReverse
   where
         wrapReverse []     = []
         wrapReverse (x:xs) = reverse xs ++ [x]
```

Note that we reuse the original function name for the wrapper function. By leaving the recursive calls in the right-hand side of the original function definition unchanged, we guarantee that the programmer observes each application of `reverse`. The only problem

might be shadowing between the name of the wrapper function and other local functions that could exist on the right-hand side. In order to avoid this conflict, we define the wrapper function by a let expression:

```
reverse = observe (oList oInt ~> oList oInt)
                  "reverse"
                  (let wrapReverse []     = []
                       wrapReverse (x:xs) = reverse xs ++ [x] in
                    wrapReverse)
```

To observe the behavior of the annotated function (`reverse`), let us apply this function to the given list `[1,2,3]`. After executing the program, the result of the function `observe` is represented as follows:

```
reverse
-------
{\[1,2,3] -> [3,2,1]}
{\[2,3] -> [3,2]}
{\[3] -> [3]}
{\[] -> []}
```

`reverse` is called four times recursively. The wrapper function gives us the possibility of showing all function calls on the given list. This technique can also be applied to selected local functions to be observed.

The wrapper function is generated automatically by iCODE. The tool also automatically generates labels to observer functions that help the programmer to later identify the observations. Top-level functions are simply labeled with their names. Local functions are labeled with a dot-separated list of function names leading to a description of their observed functions.

4.2.2 *Observing Data Types*

Observers are generated by types of expressions or functions. In Curry programs, data types (Section 2.2.6) are defined by the user and functions can be applied to expressions of these types. Further, observers are needed to observe values of the related expression type. COOSy [10] provides useful abstractions for this task and iCODE provides an automatic derivation of these observers. This derivation cannot only be used for defined data types in the program, but also for data types that are imported from other modules. We outline the idea by an example. Consider the following data type for natural numbers:

```
data Nat = O | S Nat
```

It defines two constructors:

```
O :: Nat
S :: Nat -> Nat
```

For the data type `Nat` we need the observer `oNat` (we discuss the implementation of observers further in Chapter 5). COOSy provides generic observers `o0`,`o1`,`o2`,... for constructors of arity 0,1,2,... by which the observer `oNat` can be defined as follows:

```
oNat :: Observer Nat
oNat O     = o0 "O" O
oNat (S x) = o1 oNat "S" S x
```

Now let us consider a program that has expressions of type `Nat`:

```
coin :: Nat
coin = O
coin = S O

plus :: Nat -> Nat -> Nat
plus O x     = x
plus (S x) y = S (plus x y)

main = plus coin O
```

`coin` is defined non-deterministically and has two values, zero and one. `plus` takes two arguments of type `Nat` and perform an addition of these two arguments.

To observe the function `plus`, we need the observer `oNat` that is generated automatically by iCODE. As we have discussed in Section 4.2.1, to observe the top-level function `plus` we should generate a wrapper function. The observer of the function `plus` is defined as follows:

```
plus = observe (oNat ~> oNat ~> oNat)
               "plus"
               (let wrapPlus O x     = x
                    wrapPlus (S x) y = S (plus x y) in
                 wrapPlus)
```

Calling the function `main` activates the observation system to represent the behavior of `plus` as a textual visualization:

```
plus
----
{\ O O -> O}
{\ (S O) O -> S O}
{\ O O -> O}
```

The function `coin` has two different values. The first value is matched to the pattern in the first rule of `plus`. It causes the first function call of `plus`. Note that when the value of `coin` is `(S O)`, the second rule of `plus` is executed. In the right-hand side of this rule, `plus` is called recursively. This causes the third call of `plus`.

For type constructors, an observer needs observers for the arguments as well, like `oList`. The construction should be evident in the following example:

```
data Tree a b = Branch a b [Tree a b] [Tree a b]


oTree :: Observer x1 -> Observer x2 -> Observer (Tree x1 x2)
oTree oa ob (Branch x1 x2 x3 x4) =
    o4 oa ob (oList (oTree oa ob)) (oList (oTree oa ob)) "Branch"
       Branch x1 x2 x3 x4
```

In this way, generic observers for all data structures defined in the program are generated and added automatically to the program. These observers are used for observations of functions and expressions of values of this type. This method is applied to the imported data types. Thus, for all imported modules, the data type observers can be generated and automatically imported to the program by the tool.

However, polymorphism brings up a problem. How can polymorphic functions be observed? The function can be used in different type instantiations. Hence, the only observer we can assign to its polymorphic arguments is `oOpaque` (see Example 3 of Section 4.1).

4.2.3 *Observing Expressions*

Sometimes it is not sufficient to observe functions defined in a program. Observations of sub-expressions in the right-hand sides of rules may become necessary to find a bug. For this purpose, we provide a tree representation (Section 7.1) of the whole program, in which the user can select arbitrary (sub-)expressions of the program to be observed. Corresponding calls of `observe` are automatically added to expressions, while the type of each selected expression is inferred and a corresponding observer is generated.

For this purpose, we use the meta-programming features of Curry (Section 2.4). In the abstracted tree of a Curry program, which is generated by the meta-programming library,

the type of function definitions are provided at time of compilation and by the process of type inference (see Section 2.2.4). If we convert each selected expression to a function, the type of this expression can become accessible. For this purpose, we substitute auxiliary functions for selected expressions. These functions are defined locally and contain the whole selected expressions in the right-hand sides.

To understand this idea let us consider the function `reverse`:

```
reverse :: [Int] -> [Int]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

To sketch the type of the expression `xs` in the right-hand side of the second rule, we insert the auxiliary function `auxExp` for the selected expression into the program:

```
reverse (x:xs) = reverse (let auxExp () = xs in xs) ++ [x]
```

In Curry programs all locally defined functions must have at least one argument (in another case they are considered to be local patterns). For this purpose, we have written an empty tuple as an argument for the auxiliary function `auxExp`. Now, by using the meta-programming library, we infer the type of the auxiliary function:

```
auxExp :: () -> [Int]
```

The result of the auxiliary function is the selected expression of type `[Int]`. After recognizing the type of the expression by the above technique, we can generate the appropriate observer as `oList oInt`. Now, the function `observe` is added to the program. iCODE generates all of this information automatically. It also generates the related label with the corresponding function name that the expression belongs to and a string-representation of the selected expression, where they are separated by a dot:

```
reverse (x:xs) = reverse (observe (oList oInt) "reverse.xs" xs) ++ [x]
```

This technique is also used for function calls in the right-hand side of rules. As another example, we sketch the type of `reverse` that is applied to `xs` in the right-hand side of the second rule. At first we need to substitute an auxiliary function for the selected expression:

```
reverse (x:xs) = (let auxExp () = reverse in reverse) xs ++ [x]
```

and infer the type of `auxExp`:

```
auxExp :: () -> ([Int] -> [Int])
```

Now the observer of the selected expression can be defined easily:

```
oList oInt ~> oList oInt
```

4.2.4 *Observing Imported Modules*

iCODE supports the addition of observations to different modules of a project. When the user selects functions or expressions of a module to be observed, a new module is generated that contains the observer calls. We do not manipulate the original modules but use a new module for observations, because the observation session may be necessary again, when we try to debug the same project.

For example, assume that we have already generated an observer for the function `reverse` from `module1`. The annotated program is recorded in the new module `module1_observe`. Now, in a large project we need to import the function `reverse` to another module called `module2`. To observe the behavior of functions that apply `reverse` in their right-hand sides, we should import the observer of `reverse` from `module1_observe`. This control is done automatically by the tool. In other words, since observer calls in (even indirectly) imported modules must also be executed, the tool can check for each imported module whether or not an observer version is available and uses this for execution.

Chapter 5

Implementing Debuggers by Observations

"I have learned that everyone wants to live at the top of the mountain, without knowing that real happiness is in how it is scaled."

G. Garcia Marquez

A part of the work in this thesis has been extracted from the observation method in COOSy [10]. Constructing representations of observations in COOSy is defined in a bottom-up manner that is of no use for a single-step mode representation. For this purpose, we construct information of evaluations in a tree structure. To extract this information and represent it to the user, we use a top-down algorithm. Using this algorithm the constructed tree is translated to a textual format and represented in a single-step mode to the user.

In this chapter, we first explain the expected events that are generated by observer functions in COOSy. Then, we discuss how we change COOSy's algorithm in order to represent a trace of computations in a single-step mode.

5.1 *Algorithm of COOSy*

To observe values of evaluated expressions, we add annotations to programs. In this approach, each annotation is implemented as the identity function `observe` (Chapter 4). Whenever an annotated expression is executed, its related observer function is also executed. The functional logic language Curry evaluates program expressions by lazy eval-

uation. This means that an annotated expression will be evaluated only if its value is needed for computations. During a program execution, an observer function generates information about an executed annotated expression. COOSy records this information in a trace file. These details are different events, each of which shows either that the computation has started or has been completed. In other words, there are two kinds of events to distinguish evaluated expressions from failed or non-terminated computations:

- *Demand event* shows that a value is needed for a computation.
- *Value event* shows that the computation of a value (to head normal form) has succeeded.

The events in COOSy are defined in a data type as follows:

```
data Event = Value  Arity String Index ParentID
           | Demand ArgNr Index  ParentID

type Arity    = Int
type ArgNr    = Int
type Index    = Int
type ParentID = Int
```

The indices are defined to distinguish the relationship between events. Every event needs an index representing its own identifier (`Index`) and another index representing its parent (`ParentID`). For instance, a demand event that requests a value event is defined as the parent of the related value. `Arity` shows the number of arguments that are related to a value. With respect to the lazy evaluation strategy, a demand event is required for every evaluated argument. `ArgNr` of demand events shows this dependency. Every value event contains a string that shows what should be printed out to the user.

Let us briefly discuss the algorithm of COOSy with respect to the events defined above:

1. Whenever a value of an annotated expression is needed, a computation of this expression makes progress. Beginning this computation generates a demand event. For this demand, `ArgNr` and `ParentID` are initialized by zero and `(-1)`. These numbers show that the demand generated belongs to no value arguments and is only the start of an evaluation. However, calling the function `observe` activates the observing function `observer`. Both functions are independent of the observed type (note that for every type `Type`, an observer is defined as `oType` (Chapter 4)):

   ```
   observe oType label exp = observer oType label exp 0 (-1)
   ```

2. By calling `recordEvent`, the function `observer` requests an index from a global state (to have distinct indices for different events, the indices are taken from a global state) and writes a demand event to the trace file:

```
observer oType label exp argNr parentID = unsafePerformIO $
   do eventID <- recordEvent label (Demand argNr) parentID
      return (oType exp label eventID)


recordEvent label event parentID =
   do eventID <- getNewID
      writeToTraceFile label (event eventID parentID)
      return eventID
```

   The generation of indices is easily performed by the function `getNewID`:

```
getNewID = do
   maybeStr <- getAssoc stateName
   maybe (setAssoc stateName "1" >>
          return 0)
         (\ str -> let Just (n,"") = readInt str in
                   setAssoc stateName (show (n+1)) >>
                   return n)
         maybeStr
 where stateName = "coosyState"
```

   The function `setAssoc` defines a global association between two strings. This global state can be defined only if both arguments of the function `setAssoc` can be evaluated to ground terms. Therefore, by using the function `getAssoc`, we obtain the value associated to a string that is defined as the constant `coosyState`.

3. The function oType related to the annotated expression is executed. This function in the code above is responsible for recording a value in the trace file.

   As an example of a function oType, let us implement observing the type `[t]`. It is supposed that every list element has the type `t`. To observe every element of a list, we define the observer `ot` in our example. A data type for lists can be represented as follows:

```
data List t = []
            | (:) t (List t)
```

`List` has two constructors, `[]` and `(:)`, with arities `0` and `2`. To implement `oList`, we begin with the first constructor:

```
oList ot [] label parentID =
  unsafePerformIO $
      do recordEvent label (Value 0 "[]") parentID
         return []
```

The next constructor `"(:)"` takes two arguments. The first argument has the type `t` and the second argument is a list of type `[t]`. For both arguments, we need to call the function `observer` again:

```
oList ot (x:xs) label parentID =
  unsafePerformIO $
      do eventID <- recordEvent label (Value 2 "(:)") parentID
         return ((:) (observer ot label x 1 eventID)
                     (observer (oList ot) label xs 2 eventID))
```

Depending on the arity of constructors, COOSy provides different oTypes for the predefined types in Curry libraries. This technique can also be used to define oTypes for user-defined data types.

To see generated events, let us consider the following simple program:

```
main = head [1,2,3]
```

We would like to observe the list `[1,2,3]`. For this purpose, this list must be annotated by the function `observe`:

```
main = head (observe (oList oInt) "result" [1,2,3])
```

Assume that the first index requested from the global state is zero. Now, the first recorded event that shows the beginning of the computation is: `Demand 0 0 (-1)`. The evaluation begins and the evaluated value is a list constructor `(:)` with arity 2. For the evaluated list, a value event is generated: `Value 2 "(:)" 1 0`. Note that the demand event with index 0 is defined as the parent of this value. Because of the lazy evaluation strategy, only the first element of the list is computed for the function definition of `head`. This means that the next generated event is a demand that is related to the first argument of the value event: `Demand 1 2 1`. The computation of the first list element causes a value event (`Value 0 "1" 3 2`) to be recorded in the trace file. No other values are needed to execute the program. The computation is terminated. In this step, all recorded events are:

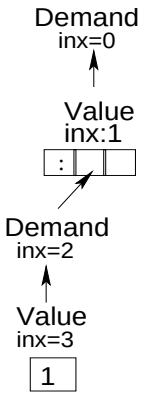


Figure 5.1: The relationship between events by observing [1,2,3]

```
[(Demand 0 0 (-1)),
 (Value  2 "(:)" 1 0),
 (Demand 1 2 1),
 (Value  0 "1" 3 2)]
```

For a better understanding of the relationship between recorded events, we display a tree representation of the events in Figure 5.1. Note that the second argument of the value event (index 1) is not evaluated. An unevaluated argument in COOSy is represented by an underscore. This trace is represented to the user as: `(1:_)`

In the observation system, the user cannot only observe data structures, but also functions. For this purpose, the data type `Event` is extended:

```
data Event = Value ...
           | Demand ...
           | Fun Index ParentID
```

`Index` is an identifier for the function event and `ParentID` points to the parent of this function.

For an example, we annotate the function `head` in our previously discussed example:

```
main = (observe (oFun (oList oInt) oInt) "head" head) [1,2,3]
```

Now, in the third step of COOSy's algorithm, the observer `oFun` generates a function event, instead of a value event:

```
oFun (oList oInt) oInt head "head" 0 [1,2,3] =
   (unsafePerformIO $
```

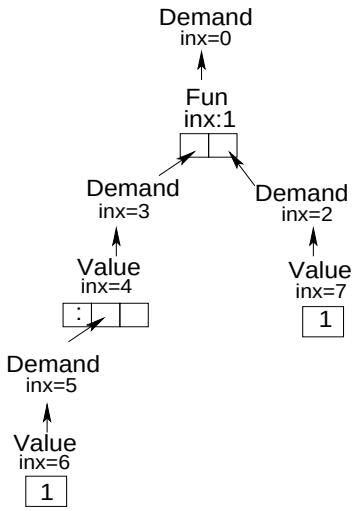


Figure 5.2: The relationship between events by observing `head`

```
do eventID <- recordEvent "head" Fun 0
   return (\ x -> (observer oInt
                            "head"
                            (head (observer (oList oInt)
                                            "head"
                                            x
                                            1 eventID))
                            2
                            eventID))) [1,2,3]
```

This means that, after executing the program and recording the first event `(Demand 0 0 -1)` in the trace file, a function event (`Fun`) that has index 1, as a child of the first event is needed: `Fun 1 0`

Function applications are defined by two parameters representing an argument and the result of the corresponding function. In COOSy, functions are represented in curried form (see Section 2.2.5). That is, the result of a function with arity two is again a function. Furthermore, we need two event applications for the two parameters of our example:

```
Demand 2 2 1
Demand 1 3 1
```

To understand the relationship between events of the trace file, we represent the entire trace as a tree (see Figure 5.2). From this tree, a string is constructed that is represented to the user as: `{\(1:_) -> 1}`.

Curry also supports non-deterministic computations. Hence, a single demand event can correspond to more than one value event. To determine which value belongs to which branch of the computation, the data type `Event` is extended with the new argument `LogPar` [10]:

```
data Event = Value  Arity String   Index    ParentID LogPar
           | Demand ArgNr Index    ParentID LogPar
           | Fun    Index ParentID LogPar
type LogPar = Int
```

A logical parent (`LogPar`) is an index for an event that just occurred in the same branch of the computation. For instance, look at Figure 5.2. After recording the value event with index 6, the event with index 7 is recorded. This means that the event with index 6 is the logical parent of the event with index 7.

Let us discuss the logical parents further by means of the following example:

```
add :: Int -> Int -> Int
add x y = x + y

num :: Int
num = 0
num = 1

main :: Int
main = add 1 num
```

In this program, the function `num` is defined non-deterministically. Calling `main` returns two results, 1 and 2. We would like to observe the function `add` in the right-hand side of `main`:

```
main = observe (oFun oInt (oFun oInt oInt)) "add" add 1 num
```

Executing this annotated program records corresponding events in a trace file. These events are represented as a tree in Figure 5.3. Consider this figure. The first argument of `add` is simply evaluated to `1` (in the value event with index 6). The right sub-tree of `add` is a function (index 3) again. This function takes the second argument of `add` and returns the result. Because of the non-deterministic behavior of `num`, we have two different values for this argument (indices 8 and 10). The results are also represented in two different

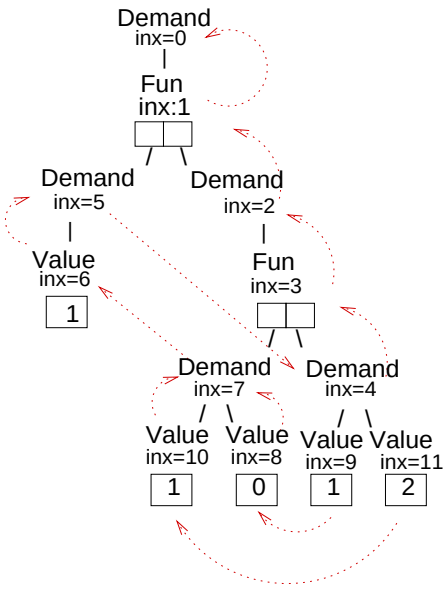


Figure 5.3: The trace of a non-deterministic computation

nodes of the tree (indices 9 and 11). To recognize which argument belongs to which result, we need to know the order of the events generation in the branches. This is provided by the logical parents (dotted arrows in Figure 5.3). If we separate the two generated traces, parts of the tree that correspond to a generated result appear (see Figure 5.4). This is easily done by following the logical parents.

Computing with logical variables is another aspect of Curry that should be observed in the observation system. When a value is observed, it should be controlled, whether or not it is a logical variable. This means that the observer must test an annotated expression before proceeding. To distinguish this kind of variables, the data type `Event` is extended:

```
data Event = Value ...
           | Demand ...
           | Fun ...
           | LogVar Index ParentID LogPar
```

For every recorded logical variable, the index of the related event, the index of the parent and also a logical parent are recorded in the trace file. What a logical variable can bind to, is either guessed by the search strategy or computed directly by code. To test whether a logical variable is already bound, the function `isVar` [17] is used:

```
isVar :: a -> Bool
```

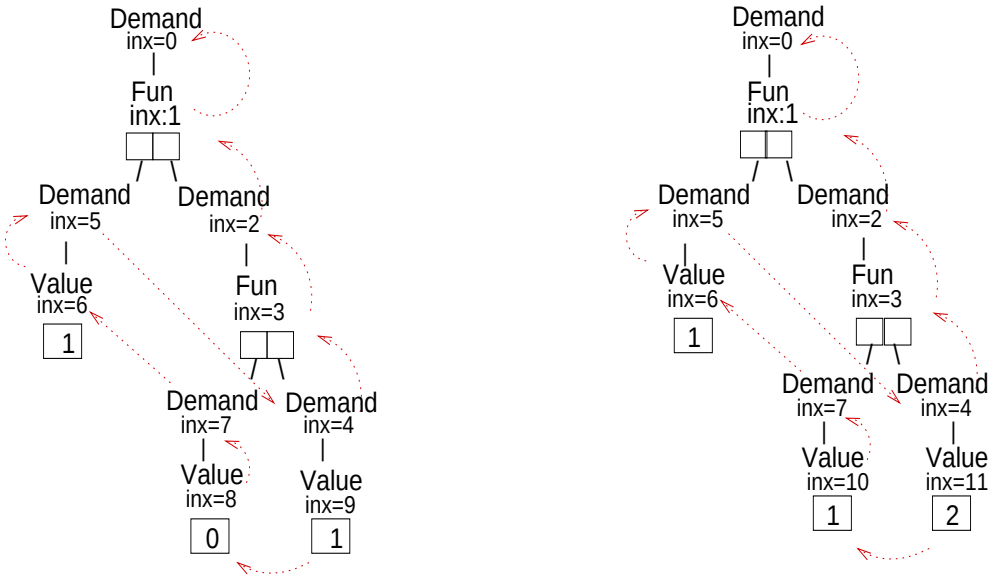


Figure 5.4: Separating the trace of evaluations

After testing the variable by `isVar`, the function `ensureNotFree` is used:

```
ensureNotFree :: a -> a
```

This function evaluates an argument to head-normal form and returns a result to continue the computations. If the result of this function is an unbound variable, the computation is suspended until the variable is bound.

Inserting logical parents to events and extending the data type `Event` for logical variables needs an update for the first and second steps of COOSy's algorithm. In the first step, the function `observer` needs a new argument (`logParents`) to provide information about logical parents:

```
observe oType label exp = observer oType label exp 0 (-1) logParents
   where logParents free
```

In the second step, calling `recordEvent` not only requests a new index for the actual event, but also a logical parent for this event:

```
recordEvent label event parentID logPars = do
    eventID <- getNewID
    let (logPar,logVar) = getLogPar parentID logPars
    ...
    writeToTraceFile label (event eventID parentID logPar)
    return (eventID,newLogVar)
```

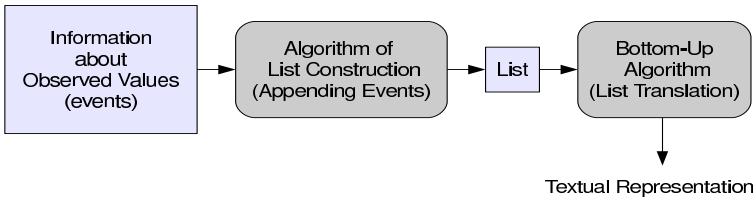


Figure 5.5: The process of the events transformation in COOSy

```
  where newLogVar free

getLogPar parentID logPars
    | isVar logPars = (parentID,logPars)
    | otherwise     = getLogPar (head logPars) (tail logPars)
```

Note that, whenever a new event is written into the trace file, a list (`logPars`) of indices (`eventID`s) terminated by a logical variable is bound to the corresponding `eventID`. The list is used because the order of evaluation is unknown. Before writing an event to the trace file, the logical variable of the list `logPars` is found and bound to the new `eventID` followed by a new logical variable [10].

5.2 *Expected Events in iCODE*

In COOSy, information about observed values is recorded as a list of events in a trace file. To present the list of recorded events to the user, this list is constructed as a tree of events (see Section 5.1) in a bottom-up manner (see Figure 5.5). For example, to represent the recorded events in Figure 5.1, COOSy's algorithm starts with the index 3 (value event) and follows the parents related, first index 2 then 1 and 0.

This algorithm is of no use for a continuous update of observed data terms, since the whole observed data structure must be re-constructed at each step. As a solution, we record the information about observed values in a tree structure and use a top-down algorithm that allows extensions in all possible leaf positions of the presented data structure (see Figure 5.6). For instance, during the computation, non-evaluated arguments (represented by underscores) may flip into values, but values within a data structure will no longer change. However, we must consider the non-determinism within Curry by which values may later be related to different non-deterministic computations.

To avoid storing huge amounts of events in a trace file, we define a global state to record events. The Global library [17] of PAKCS helps for this purpose. In this library, an entity `g` with an initial value `v` of type `t` and a storage mechanism `s` is described by:

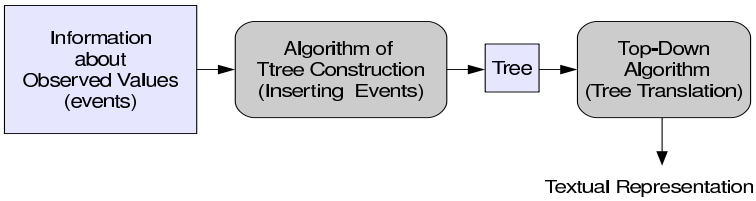


Figure 5.6: The process of the events transformation in iCODE

```
g :: Global t
g = global v s
```

Using this library we implement a top-down algorithm. For this purpose, we store the events in the following data structure:

```
TreesTable :: Global [([Index],EvalTree)]
TreesTable = global [] Temporary
data EvalTree = Open SubTreeNr
              | Value  Arity String Index [EvalTree]
              | Demand ArgNr        Index [EvalTree]
              | Fun                 Index [EvalTree]
              | LogVar              Index [EvalTree]

type Index     = Int
type ArgNr     = Int
type Arity     = Int
type SubTreeNr = Int
```

`TreesTable` is a kind of global state that is accessible and modifiable within the whole program. The list of indices `[Index]` in this data structure represent all nodes occurring in the corresponding evaluation tree (`EvalTree`) in the order in which they were added. This is necessary since the evaluation order is not statically fixed. However, the index list is extended in its head position whenever the evaluation tree is extended (in an `Open` leaf). The `Open` node is used when the evaluation of an argument is not started. Every constructed EvalTree is represented to the user by a pretty printing method and using a top-down algorithm (see Figure 5.6) , so that an open node is represented by an underscore (_), a demand by an exclamation mark (`!`), a function by a mapping from arguments to results (\args ->results) and a logical variable by a question mark (`?/v`). Values are represented by strings that are their related constructors.

Let us consider the following program while the list `[1,2,3]` is annotated to be observed:

```
main = head (observe (oList oInt) "result" [1,2,3])
```

When executing the program, the first generated event is a demand event `(Demand 0 0 (-1) (-1))` that is defined as the root of the tree and has one open sub-tree. The index of the event is also recorded in the head position of the index list:

```
[([0] ,
  Demand 0 0 [(Open 1)])]
```

The generated evaluation tree is translated to a string and represented to the user as `!`. The exclamation mark shows that an evaluation has begun, but that the computation has not been completed.

The second generated event is a value `(Value 2 "(:)" 1 0 0)`. Because the parent of this event has the index 0, the evaluation tree can be extended in its open sub-tree and the index of the value event is recorded in the index list:

```
[([1,0] ,
  Demand 0 0 [(Value 2 "(:)" 1 [Open 1 , Open 2])])]
```

The arity of the recently generated event is 2. This means that two open sub-trees are needed for this event. The open sub-trees represent unevaluated parts of the observed value and are displayed to the user by underscores: `(_:_)`. The first open sub-tree (`Open 1`) is considered to be the head of the list and the second one (`Open 2`) to be the rest list.

The third generated event is a demand event `(Demand 1 2 1 1)`. The argument number of this event (`1`) shows that it belongs to the first open sub-tree of its parent:

```
[([2,1,0] ,
  Demand 0 0 [(Value 2 "(:)" 1 [Demand 1 2 [(Open 1)],
                                Open 2])])]
```

The tree in this step indicates that the evaluation of the first list element has begun - `(!:_)`.

The next generated event is a value event (`Value 0 "1" 3 2 2`). The parent of this event has index 2. This means that the open sub-tree of the parent can be extended. The index of the recently generated event is also inserted into the index list to record the order of evaluations:

```
[([3,2,1,0] ,
  Demand 0 0 [(Value 2 "(:)" 1 [Demand 1 2 [(Value 0 "1" 3 [])],
                                Open 2])])]
```

The computation has been completed. Note that the second open sub-tree of the value event with index 1 is not extended, because the execution of `head` needs only the value of the first list element. The generated tree of this step is represented to the user as `(1:_)`.

Evaluation trees are extended until an execution has been completed or failed. Note that, within a non-deterministic computation, parts of a value are used in more than one computation. In this case, a logical parent indicates which part of an evaluation tree is shared within the other related non-deterministic computations. We only consider the sub-tree consisting of nodes from the index list to the logical parent of the new event. This sub-tree with the corresponding indices is copied as an additional evaluation tree to the global `TreesTable`. To present shared parts of a computation to the user, we extend the data type `EvalTree` as follows:

```
data EvalTree = Open ...
              | Value ...
              | Demand ...
              | Fun ...
              | LogVar ...
              | Share EvalTree
```

The constructor `Share` has an argument. This argument is considered to be the new parts of the evaluation tree in a non-deterministic computation.

For example, we again consider the following simple program from the last section that performs a non-deterministic computation:

```
add :: Int -> Int -> Int
add x y = x + y

num :: Int
num = 0
num = 1

main :: Int
main = add 1 num
```

In Section 5.1 we have observed the function `add` in the right-hand side of `main` by COOSy. Now we observe the same expression by iCODE.

The function `main` offers two results: 1 and 2. For the first result after annotating the function `add` to be observed, ten events are generated. The first event is a `Demand` that is stored in `EvalTree` with index `0`:

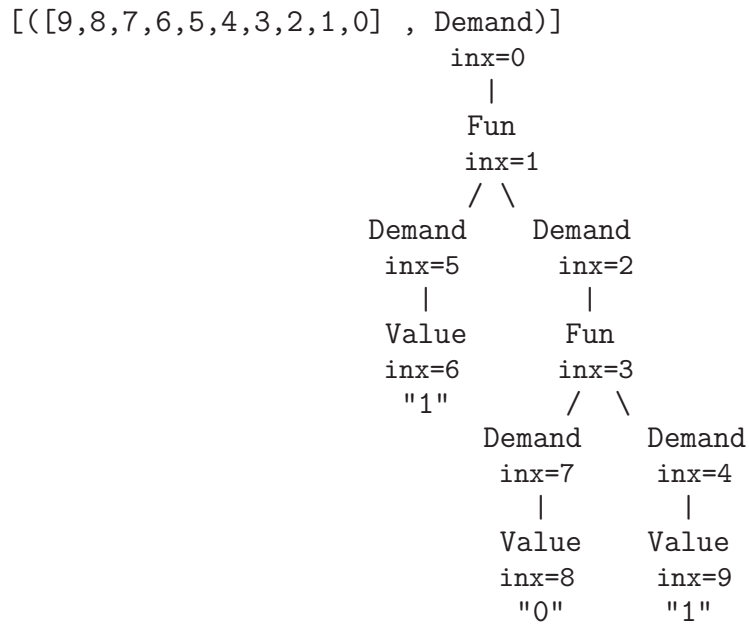
```
[([0],
  Demand 0 0 [(Open 1)])]
```

The second generated event is a fun event with logical parent `0` that should be substituted in the open sub-tree of its parent:

```
[([1,0],
  Demand 0 0 [Fun 1 [(Open 1),
                     (Open 2)]])]
```

This fun event is stored as a node with two sub-trees presenting the argument and the result of the corresponding function.

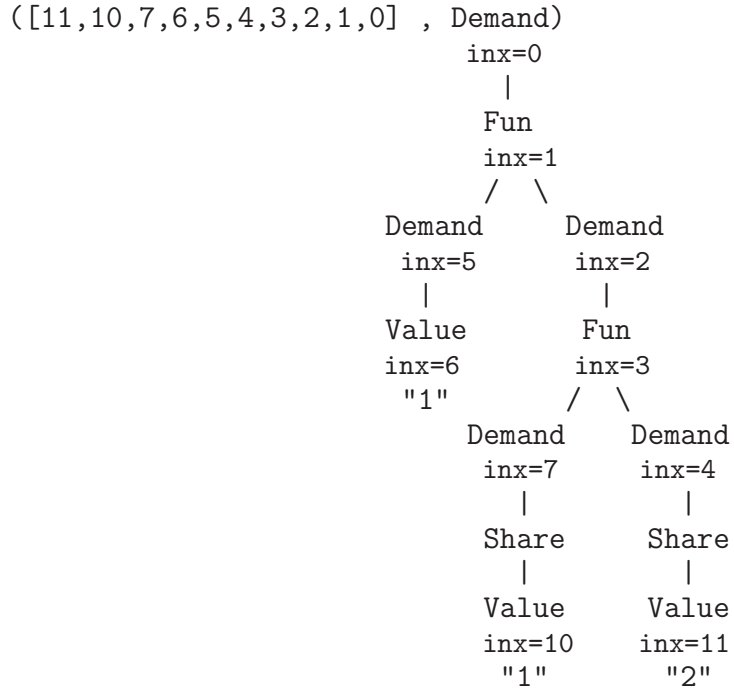
After adding the remaining eight events to the evaluation tree, we obtain



which is presented to the user as `{\1 0 -> 1}`.

The next generated event is a value event with logical parent `7`. This index does not occur in the head position of the index list. Hence, we detect a non-deterministic computation. The observed value of this computation shares the nodes with indices 0 to 7 with the first tree. Therefore, we copy this part where `Share` shows which parts of the tree are shared. The copied parts are extended with the new event 10. This means that the same function is called with another argument (`1`) in this non-deterministic branch.

After adding the last event we obtain



which is presented to the user as `{\1 1 -> 2}`. The highlighted parts show the shared parts of the computation. This top-down construction method helps us to provide fast pretty printing for each intermediate step of observations in the implemented tool iCODE.

Instead of recording events in a trace file, we need an algorithm in order to store the events in an evaluation tree. For this purpose, we change the function `recordEvent` from the second step of COOSy's algorithm. This function is simply changed by replacing a new function `sendToShow` instead of `writeToTraceFile`:

```
recordEvent label event parentID logPars = do
    eventID <- getNewID
    let (logPar,logVar) = getLogPar parentID logPars
    ...
    sendToShow (EventMessage label (event eventID parentID logPars))
    return (eventID,newLogVar)
  where newLogVar free
```

iCODE is implemented by a distributed architecture that will be discussed in Chapter 8. In this section we restrict our discussion to the global state of evaluation trees. We discuss `EventMessage` in the related chapter. Hence, an execution of the function `sendToShow` causes the generation of evaluation trees by the following steps:

1. Reading the initialized global state `TreesTable`.

2. Inserting the actual event into the related evaluation tree.

3. Translating the updated tree (from step 2) to a string for representation to the user.

4. Replacing the new updated tree (from step 2), instead of the last existed tree in the global state.

As long as a new event is generated, the above steps are repeated. If an execution of a program is terminated or failed, this algorithm is stopped and the latest constructed tree is represented to the user. These steps belong to the algorithm of the tree construction presented in Figure 5.6 and performed by executing the following code:

```
...
tree    <- readGlobal TreesTable
newTree <- insertToTree event tree
showEvent (translateToStr newTree) label
writeGlobal TreesTable newTree
...
```

The function `showEvent` presents the translated tree in an appropriate viewer that belongs to the `label` of the annotated expression.

To insert a generated event to the related evaluation tree, we should look up the index of this event in the indices lists of `TreesTable`. It brings up four situations:

1. If the logical parent of the newly generated event is (-1), the event is the root of a new evaluation tree that is inserted into `TreesTable`.

2. If the logical parent of the event appears in the head position of an index list, then:

   (a) the related evaluation tree is extended in an open leaf corresponding to the parent of the event.

   (b) the index list is extended in its head position by inserting the index of the event.

3. If the logical parent is not equal to the first element of the index list, but exists in the list, then:

   (a) a copy of the index list is provided such that the indices from the head position to the logical parent are removed.

   (b) a copy of the related evaluation tree is provided such that all events for which there is no index in the list (from step a) are removed from the tree.

(c) the nodes of the tree (from step b) which belong to the shared parts are extended with the event `Share`.

(d) the index of the recently generated event is added to the head position of the copied part (from step a) of the index list.

(e) the event is inserted into the copied part of the tree (from step c).

4. If the logical parent does not exist in an index list, then the algorithm is repeated for the other trees existing in `TreesTable`.

These steps are performed whenever the function `insertToTree` is called:

```
insertToTree event t =
  let logPar  = takeLogicalParent event
      eventID = takeEventID event in
   case event of
     COOSY.Demand argNr ownId _ _ ->
      if logPar == -1
        then ([eventID],ICODE.Demand argNr ownId [Open 1]):t
        else insertNextEvent event t
     _ -> insertNextEvent event t

insertNextEvent event (((p:ps),events):ts)
   | logPar == p =
       let newEvents=insertEvent event events in
         (eventID:p:ps,newEvents):ts
   | elem logPar ps =
       let psCopy     = dropWhile (logPar/=) ps
           eventsCopy = filterEvents psCopy events
           newEvents  = insertEvent event eventsCopy in
        (eventID:psCopy,newEvents): ((p:ps),events):ts
   | otherwise = ((p:ps),events):insertNextEvent event ts
  where logPar  = takeLogicalParent event
        eventID = takeEventID event

filterEvents [] tree        = tree
filterEvents (n:nodes) tree = head (filterEvents' (n:nodes) tree)

filterEvents' _ (Open n) = [Share (Open n)]
```

```
filterEvents' nodes (ICODE.Value arity str ownId subTrees)
  | elem ownId nodes = [ICODE.Value arity str ownId
                                    (concatMap (filterEvents' nodes)
                                               subTrees)]
  | otherwise = [Share (Open 1)]
filterEvents' nodes (ICODE.Demand argNr ownId subtrees)
  | elem ownId nodes = [ICODE.Demand argNr ownId
                                     (concatMap (filterEvents' nodes)
                                                subtrees)]
  | otherwise = [Share (Open argNr)]

filterEvents' nodes (ICODE.Fun ownId subtrees)
  | elem ownId nodes = [ICODE.Fun ownId
                                  (concatMap (filterEvents'' nodes)
                                             subtrees)]
  | otherwise = [Share (Open 1)]
filterEvents' nodes (ICODE.LogVar ownId subtrees)
  | elem ownId nodes = [ICODE.LogVar ownId
                                     (concatMap (filterEvents' nodes)
                                                subtrees)]
  | otherwise = [Share (Open 1)]
filterEvents' nodes (Share tree) = [filterEvents nodes tree]

filterEvents'' nodes (t,t') =
     [(head (filterEvents' nodes t),head (filterEvents' nodes t'))]
```

The function `insertEvent` finds the parent related to the event in the tree (`eventsCopy`) and substitutes this event for the corresponding open sub-tree of the parent. After performing each insertion, the newly constructed tree can be represented to the user.

By storing the indices of generated events in a list, we can also perform backward and forward stepping through the observations represented to the user. This is performed by filtering `TreesTable` with respect to a subset of considered indices.

To remove evaluation trees from `TreesTable`, we have defined a clear-button in each trace window. Furthermore, when the observed program is restarted, `TreesTable` is cleared automatically.

Chapter 6

Operational Semantics

"Be brave. Take risks.
Nothing can substitute experience."

Paulo Coehlo

In this chapter, we prove that annotating program expressions by observer functions does not change the behavior of programs. Furthermore, we formalize the annotations of observations and events. For this purpose, we define an operational semantics that is an extension of the semantics represented in [1, 2].

6.1 *Functional Logic Languages*

Launchbury [42] models an operational semantics that represents the semantics behind sharing for the lazy evaluation strategy. The new features (non-determinism and logical variables) of functional logic languages need an extension of the operational semantics represented in [42]. This extension is defined in [1, 2, 12], so that, in order to provide a simple operational description, programs are translated to a `flat` form [28]. The flat representation of functional logic programs is useful to provide an explicit representation of the pattern matching strategy using case expressions. The syntax of flat programs is presented in Figure 6.1.

Every program P contains a sequence of function definitions D. The left-hand side of a function definition contains different variable arguments x_i, $i \in \{1, \ldots, n\}$ and the right-hand side is an expression e. Expressions are composed by variables, data constructors, function calls, case expressions, disjunctions and let bindings. We define the set of flat programs as $Prog$ and the set of expressions as Exp, meaning that $P \in Prog$ and $e \in Exp$.

P	::=	$D_1 \ldots D_m$	(program)
D	::=	$f(x_1, \ldots, x_n) = e$	(function definition)
e	::=	x	(variable)
	|	$C(e_1, \ldots, e_n)$	(constructor call)
	|	$f(e_1, \ldots, e_n)$	(function call)
	|	$case\ e\ of\ \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\}$	(rigid case)
	|	$fcase\ e\ of\ \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\}$	(flexible case)
	|	$e_1\ or\ e_2$	(disjunction)
	|	$let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e$	(let binding)
	|	$prim\_g\ (e_1, e_2)$	(primitive functions)
	|	$hnf(x_1, x_2)$	(head normal form evaluator)
p	::=	$C(x_1, \ldots, x_n)$	(pattern)

Figure 6.1: Flat program

There are primitive functions that are applied to their arguments only if these arguments are evaluated to constructor terms. For instance, $a + b$ can be reduced to a value if the variables a and b are already evaluated to values. Note that the variables can also be uninstantiated in a program. Since parameters of a function call are not evaluated in the lazy evaluation strategy but transfered to the body of this function, primitive functions in Curry are implemented as function definitions. In our semantics, primitive functions named as g, which are applied to expressions e_1 and e_2, are represented by the prefix "$prim\_$":

$prim\_g(e_1, e_2)$

However, variable arguments of calls to these primitive functions need to be instantiated before applying functions to the variables. To evaluate arbitrary arguments to head normal form, primitive functions are implemented using the function hnf. As in [2], we consider hnf applications $hnf(e_1, e_2)$ that first evaluate e_1 to head normal form before they return e_2 as a result. For instance, the Curry expression $a + b$ that is translated to a call of a primitive function in flat programs is considered as:

$a + b = hnf(a, hnf(b, prim\_+(a, b)))$

Using hnf functions, the arguments a and b are evaluated to head normal forms before the primitive function $prim\_+$ is applied to the values of these arguments.

By means of non-deterministic computations in functional logic languages like Curry, expressions can be evaluated to different results. To simplify such expressions, conjunctions (or-expressions) are used. For instance, the function *select* non-deterministically

returns one of its arguments:

$select\ x\ y\ =\ x$

$select\ x\ y\ =\ y$

With respect to the syntax of flat programs, function definitions must be represented as $f(x_1, \ldots, x_n) = e$. This means that, instead of a currified notation, a Curry application like *select* e_1 e_2 is annotated with brackets in the flat program:

$select(e_1, e_2)$

This representation is restricted to first-order functions. The representation of higher-order functions needs another transformation that is discussed in Section 6.2.3.

Since *select* acts non-deterministically on its arguments, the two rules of the function *select* are combined by the operator *or* in only one rule:

$select(x, y) = x\ or\ y$

The syntax of the flat representation contains two different case expressions:

$(f)case\ e\ of\ \{p_1 \rightarrow e_1; \ldots; p_k \rightarrow e_k\}$

where e and e_i are expressions, p_i is a pattern and $1 \leq i \leq k$.

The difference between *case* and $fcase$ appears when the argument e evaluates (at run time) to a logical variable. In this case, *case* suspends the evaluation of the variable but $fcase$ non-deterministically binds this variable to a pattern (p_i) to continue the evaluation (see Section 2.3).

As an example, let us consider the function *and* with the following definition in a Curry program:

$and\ False\ \_\ =\ False$

$and\ True\ x\ =\ x$

Note that *and* returns $False$ if the first argument is $False$ independently of the second argument. Only if the first argument is $True$, the result depends on the second argument. We convert the above function definition into a flat form using case expressions:

$$and\ (x, y) = fcase\ x\ of\ \{False \rightarrow False\ ;\ True \rightarrow y\}$$

Another feature of functional logic languages beside non-determinism is the use of logical variables. Logical variables are those variables in a rule that do not appear on the left-hand side (see Section 2.3.2). Logical variables are declared to be free in local definitions. For instance, if the variable x is a free variable in the Curry expression e, we can write: $let\ x\ free\ in\ e$. [7] pointed out that the use of extra variables causes no problem if they are renamed whenever a rule is applied. With respect to this definition, [2] suggests

a circular binding definition for logical variables in flat programs ($let\ x = x\ \ in\ e$) to mean that x is not instantiated.

In summary, we transform programs into a flat form such that:

- functions are represented by only one rule with an explicit representation of the pattern matching by nested case expressions.
- non-deterministic definitions are represented by or-expressions.
- logical variables are introduced by circular let bindings on these variables [7].

The semantics represented in [2] describes the operational semantics of functional logic languages in two stages:

- *Natural semantics (big step)* that defines the intended results by relating expressions to values.
- *Implementation-oriented semantics (small step)* that reasons about operational aspects of programs.

To represent the operational semantics of our work, we use the big-step semantics that is discussed in the next section.

6.1.1 *Natural Semantics*

As in [2], to represent a natural semantics (called big step), a normalization process similar to [42] is first applied. In this process, all arguments of functions, constructors and case patterns are transformed into variables. The transformation process of an expression e is represented by e^*. It is introduced as follows, while φ denotes either a constructor or a function symbol. A function symbol can also be a primitive or a hnf function. $\overline{o_n}$ denotes a sequences of objects $(o_1, \ldots, o_n)$:

$$
\begin{array}{rcl}
x^* & = & x \\
\varphi(\overline{x_n})^* & = & \varphi(\overline{x_n}) \\
\varphi(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_n)^* & = & let\ x_i = {e_i}^*\ in \\
 & & \quad \varphi(x_1, \ldots, x_{i-1}, x_i, e_{i+1}, \ldots, e_n)^* \\
 & & \text{where } e_i \text{ is not a variable and } x_i \text{ is fresh} \\
(let\ \{\overline{x_k = e_k}\}\ in\ e)^* & = & let\ \{\overline{x_k = e_k^*}\}\ in\ e^* \\
(e_1\ or\ e_2)^* & = & e_1^*\ or\ e_2^* \\
((f)case\ e\ of\ \{\overline{p_k \rightarrow e_k}\})^* & = & (f)case\ e^*\ of\ \{\overline{p_k \rightarrow e_k^*}\}
\end{array}
$$

In this process, variables do not need to be transformed, but expressions (e) have to be normalized (e^*). The normalization process introduces a new let binding for each non-variable argument. Note that this can be modified in order to generate one single let expression with bindings for all non-variable arguments of a function or constructor call. For example, a function application of f to the arguments 1 and 2 (i.e., $f(1,2)$) in a normalized flat program is represented as:

$$\begin{array}{l} let\ x = 1, \\ \quad y = 2\ \ in \\ f(x,y) \end{array}$$

where x and y are fresh variables.

By applying the normalization process, a program and all its expressions are transformed and are ready to be interpreted by the natural semantics. The derivation rules are represented in Table 6.1, where:

$$\begin{array}{rcl} P & \in & Prog \\ \Gamma, \Delta, \Theta & \in & Heap = Var \mapsto Exp \\ x, y & \in & Var \\ e & \in & Exp \\ v & \in & Value ::= x \mid C(\overline{e_n}) \end{array}$$

A value in this definition is a constructor-rooted term (i.e., a term whose outermost function symbol is a constructor symbol) or a logical variable. A heap is a mapping from variables to expressions to model sharing. We use the following notations for heaps:

- $[]$ denotes an empty heap.
- $\Gamma[x]$ denotes the value associated to the variable x in the heap Γ.
- $\Gamma[x \mapsto e]$ denotes the heap Γ' obtained by updating the heap Γ where x is bound to e. Formally:

$$\Gamma'[y] = \begin{cases} e & if\ x = y \\ \Gamma[x] & otherwise \end{cases}$$

- $\Gamma[x] = x$ means that x is a logical variable.
- $[x_1 \mapsto e_1, x_2 \mapsto e_2, ..., x_n \mapsto e_n]$ denotes a heap that maps variable x_i to expression e_i where $\forall i \neq j$, $x_i \neq x_j$ and $i \in \{1, ..., n\}$, $j \in \{1, ..., n\}$.

In the derivation rules represented in Table 6.1, the judgment $(\Gamma : e \ \Downarrow \ \Delta : v)$ is interpreted such that the expression e in the context of the heap Γ evaluates to the value v

together with the (possible modified) heap Δ. For instance, a proof of a judgment (called *derivation tree*) in a rule such as follows:

$$\frac{\Gamma : a \;\Downarrow\; \Delta : b \qquad \Delta : c \;\Downarrow\; \Theta : d}{\Gamma : e \;\Downarrow\; \Theta : v}$$

means that the expression e is evaluated to the value v, if a is evaluated to b (i.e., $\Gamma : a \;\Downarrow\; \Delta : b$) and c is evaluated to d (i.e., $\Delta : c \;\Downarrow\; \Theta : d$), where a and c are expressions and b and d are values. The derivation tree can also be represented as follows:

$$\begin{array}{l} \Gamma : e \\ \left[\begin{array}{l} \left.\begin{array}{l} \Gamma : a \\ \vdots \\ \Delta : b \end{array}\right\} \textit{the first subproof} \\ \left.\begin{array}{l} \Delta : c \\ \vdots \\ \Theta : d \end{array}\right\} \textit{the second subproof} \end{array}\right. \\ \Theta : v \end{array}$$

The proof attempt of a judgment may also fail. This occurs in two cases:

- Derivation is not terminated. This corresponds to an infinite loop (e.g., because of an infinite recursive call to a function).
- There is no rule which can be applied for derivation of a subproof (e.g., because of a pattern matching failure).

Let us briefly explain the derivation rules of Table 6.1:

VarCons: Evaluating a variable yields a constructor-rooted term, if there is a binding from the variable to this term in the heap.

VarExp: This rule evaluates a variable (x) in the context of a heap (Γ) where the heap contains a binding for the variable ($x \mapsto e$). The derivation tree of this rule shows that the variable x can be evaluated to the value v only if the expression e is evaluated to this value. If expression e cannot be evaluated to a value, the derivation fails. Note that, in contrast to [42], *VarExp* does not remove the binding of the variable from the heap, because generating fresh variable names by let bindings will be easier.

Table 6.1: Natural Semantics for Functional Logic Programs

$VarCons$	$\Gamma\,[x \mapsto t] : x \Downarrow \Gamma\,[x \mapsto t] : t$
$VarExp$	$\dfrac{\Gamma\,[x \mapsto e] : e \;\Downarrow\; \Delta : v}{\Gamma\,[x \mapsto e] : x \;\Downarrow\; \Delta\,[x \mapsto v] : v}$
Val	$\Gamma : v \;\Downarrow\; \Gamma : v$
Fun	$\dfrac{\Gamma : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : f(\overline{x_n}) \;\Downarrow\; \Delta : v}$
Let	$\dfrac{\Gamma\left[\overline{y_k \mapsto \rho(e_k)}\right] : \rho(e) \;\Downarrow\; \Delta : v}{\Gamma : let\{\overline{x_n = e_k}\}\ in\ e \;\Downarrow\; \Delta : v}$
Or	$\dfrac{\Gamma : e_i \;\Downarrow\; \Delta : v}{\Gamma : e_1\ or\ e_2 \;\Downarrow\; \Delta : v}$
$Case$	$\dfrac{\Gamma : e \;\Downarrow\; \Delta : C(\overline{y_n}) \qquad \Delta : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : (f)case\ e\ of\ \{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$
$FCase$	$\dfrac{\Gamma : e \;\Downarrow\; \Delta : x \quad \Delta\,[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \;\Downarrow\; \Theta : v}{\Gamma : fcase\ e\ of\{\overline{p_k \to e_k}\} \;\Downarrow\; \Theta : v}$
$PrimFun$	$\Gamma : prim\_ \oplus (x_1, x_2) \;\Downarrow\; \Gamma : \Gamma[x_1] \oplus \Gamma[x_2]$
HNF	$\dfrac{\Gamma : x_1 \;\Downarrow\; \Delta : v_1 \qquad \Delta : x_2 \;\Downarrow\; \Theta : v_2}{\Gamma : hnf(x_1, x_2) \;\Downarrow\; \Theta : v_2}$

where in

$VarCons$:	t is constructor-rooted
$VarExp$:	e is not constructor-rooted and $e \neq x$
Val:	v is constructor-rooted or a variable with $\Gamma\,[v] = v$
Fun:	$f(\overline{y_n}) = e \in P$ and $\rho = \{\overline{y_n \mapsto x_n}\}$
Let:	$\rho = \{\overline{x_n \mapsto y_n}\}$ and $\overline{y_n}$ are fresh variables
Or:	$i \in \{1, 2\}$
$Case$:	$p_i = C(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$
$FCase$:	$p_i = C(\overline{x_n})$, $\rho = \{\overline{x_n \mapsto y_n}\}$ and $\overline{y_n}$ are fresh variables
$PrimFun$:	$\oplus$ is a primitive function

Val: If a value is evaluated, the value is returned and the heap is kept without any modification.

Fun: By this rule, function calls are evaluated to values where parameters of corresponding function rules are substituted. Note that the program P is considered to be a global parameter of the calculus.

Let: All bindings of a let expression are added to the heap and proceed with the evaluation of the main argument (e).

Or: The completely new rule that is added to Launchbury's model [42] is the rule that is related to non-deterministic computations. This rule shows that a derivation can proceed by evaluating the expression e_1 or by evaluating e_2.

$Case$: The case rule is strict in its main expression (e). It only succeeds if the constructor returned by the evaluation of e occurs in the pattern list of the case expression. In this case, all arguments of the constructor replaces the variables to continue the derivation process.

$FCase$: If the main argument (e) of the case expression is evaluated to a logical variable (x), this rule is used. $FCase$ non-deterministically binds this variable to the patterns ($x \mapsto \rho(p_i)$) and evaluates the corresponding branch. Pattern variables are also renamed ($\overline{x_n \mapsto y_n}$). Finally, the heap is updated with the renamed logical variables of patterns.

$PrimFun$: Primitive functions are applied to their arguments only if these arguments are already evaluated and bound to values in the heap ($\Gamma[x_1]$ and $\Gamma[x_2]$). It is supposed that there are no variable chains in the heaps. Since logical variables are not always instantiated to constructor-rooted terms but can also be bound to other logical variables, chains of bindings might occur in heaps. In $PrimFun$, $x1$ and $x2$ are considered to be bound to constructor-rooted terms.

HNF: This rule is used when a hnf function is being evaluated. The rule first evaluates the argument x_1 to head normal form (v_1) before it returns v_2 as a result. Note that in this semantics, a logical variable is also a head normal form ($Value ::= x|C(\overline{e_n})$).

Now, we will examine the derivation rules by means of a simple program written in Curry:

$f\ x\ =\ x$

$main\ =\ f\ 1$

In order to represent the operational semantics of this program, we first transform it:

$f(x)\ =\ x$

$main \ = \ let\ y = 1 \ \ in \ \ f(y)$

Now, we need an empty heap and an expression to evaluate:

$[] : main$

The expression $main$ can be evaluated using $VarExp$:

$[] : let\ y = 1\ in\ f(y)$

To evaluate the let expression, Let is needed:

$[y \mapsto 1] : f(y)$

The derivation of the function application $f(y)$ can be performed by Fun:

$[y \mapsto 1] : y$

Now $VarCons$ evaluates the variable y:

$[y \mapsto 1] : 1$

In this step, the execution of $f(y)$ is completed by evaluating to 1. Then, the derivation trees of Let and $VarExp$ are also completed. The following is how the derivation tree of this program is represented:

$$
\left[\begin{array}{l}
[] : main \\
\left[\begin{array}{l}
[] : let\ y = 1\ in\ \ f(y) \\
\left[\begin{array}{l}
[y \mapsto 1] : f(y) \\
\left[\begin{array}{l}
[y \mapsto 1] : y \\
[y \mapsto 1] : 1
\end{array}\right] \\
[y \mapsto 1] : 1
\end{array}\right] \\
[y \mapsto 1] : 1
\end{array}\right] \\
[y \mapsto 1] : 1
\end{array}\right]
$$

6.2 *A Semantics for Observations*

Our operational semantics, named *observation semantics*, is supposed to illustrate what happens when an annotated expression in the debugging system is executed. As we have discussed in Section 4.2, showing intermediate information of evaluations (run-time values) is possible for annotated program expressions by observer functions. In this section, we perform two steps to represent the observation semantics of a normalized program (see Figure 6.2):

- *Labeling normalized programs*: This gives labels to arbitrary expressions and function definitions selected by the user.

- *Extracting events*: This generates a tree of events in a top-down manner.

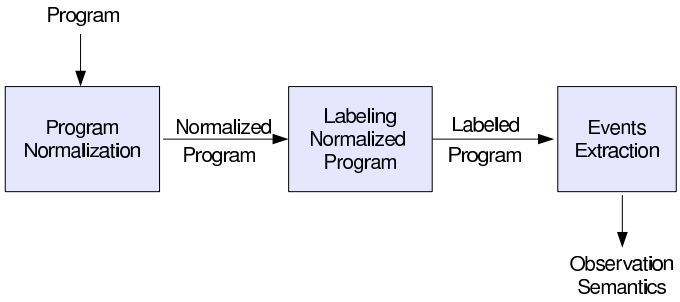


Figure 6.2: The process of observation semantics

6.2.1 *Labeling Normalized Programs*

Annotating expressions by observer functions must not change the behavior of programs. This process involves three steps:

1. Programs are transformed into flat form [28] where all arguments of functions and constructors are variables (the transformation process is discussed in Section 6.1.1).

2. As discussed in Chapter 4, arbitrary expressions or function definitions can be selected for annotation by observer functions. Adding observer functions to programs is performed automatically by the tool. In this stage of the labeling process, we only give the label "O" to every expression e^o that is selected by the user ($O(e^o)$). This means that, if values of the labeled expression are needed for a computation, they will be observed. Function calls $f(\overline{x_n})$ can also be observed. In the observation semantics, an observed function is defined as $f^o(\overline{x_n})$. Note that, if a function is selected to be observed, we replace $f^o(\overline{x_n})$ instead of $f(\overline{x_n})$ in the program.

   In this semantics, we define the set of labeled expressions as Exp^o, meaning that $O(e^o), f^o(\overline{x_n}) \in Exp^o$ (see Figure 6.3).

3. As discussed in Chapter 5, executing annotated expressions by observer functions causes the generation of events and constructions of evaluation trees. In the debugging system, we use indices to provide a relationship between events. In the observation semantics, instead of using indices, we bind labeled expressions ($O(e^o)$) to fresh variables ($let\ y = O(e^o)\ in\ y$ where y is fresh) and use these variables to define a relationship between events. Note that, for selected function calls, the name of functions are used.

The structure of labeled programs is presented in Figure 6.3. The syntax in this figure indicates that a labeled program (P^o) can contain labeled function definitions with

labeled expressions ($f^o(\overline{x_n}) = O(e^o)$) as well as non-labeled ones ($f^o(\overline{x_n}) = e^o$). A labeled program can also contain function definitions that are not selected in order to be observed ($f(\overline{x_n}) = e^o$). In this approach, labeled programs are defined in the set $Prog^o$, meaning that $P^o \in Prog^o$.

The structure of expressions (e^o) shows that a programmer can observe the behavior of a function on its arguments ($f^o(\overline{x_n})$) or the result of a function call that is a value $O(f(\overline{x_n}))$.

Every sub-expression of or-, case-, fcase- and let expressions can also be observed by a programmer. In other words, besides observing the whole expressions, programmers can select sub-expressions for observation.

We have extended the syntax of expressions with observers. Whenever the computation of a labeled expression ($O(e^o)$) or a labeled function ($f^o(\overline{x_n})$) is needed, the evaluation is started and an observer is applied to the selected expression or function. An application of an observer to an expression e^o is represented as $Obs(e^o)$ and to a function $f(\overline{x_n})$ as $obs\_f(\overline{x_n})$. Note that such a conversion is not performed in the observation system. It is only used to formalize the derivation rules for the generation of events.

After the labeling process, the program is ready for execution in the observation system to generate events. Before we discuss how events are extracted from a program, let us look at some simple examples. Suppose that we would like to observe every application of a function (f) with the following definition in a Curry program:

$f\ x\ y\ =\ x + y$

For this purpose, the related function definition must be selected to be observed. The debugging system annotates this function definition by an observer function (see Section 4.2) in the program. In the observation semantics, we only give the label "^{o}" to the name of the function definition. This means that the above program must be transformed as follows:

$f^o(x, y)\ =\ +(x, y)$

$+(x, y)\ =\ let\ a = prim\_+(x, y),\ b = hnf(y, a)\ \ in\ \ hnf(x, b)$

Note that the operator + is considered to be a predefined function using the hnf function (i.e., $x + y = hnf(x, hnf(y, prim\_+(x, y)))$) before the transformation process is performed.

Now, assume that we would like to observe the arguments and result of only one application of this function in a special part of a Curry program, such as:

$main\ =\ f\ 1\ 2$

For this purpose, we select the function call f. This selected function call is labeled and

P^o	::=	$\overline{D^o_m}$	(labeled program)
D^o	::=	$f^o(\overline{x_n}) = e^o$	(labeled function definition)
	|	$f(\overline{x_n}) = e^o$	(function definition)
e^o	::=	$O(e^o)$	(labeled expression)
	|	x	(variable)
	|	$C(\overline{x_n})$	(constructor call)
	|	$f(\overline{x_n})$	(function call)
	|	$f^o(\overline{x_n})$	(labeled function call)
	|	$case\ e^o\ of\ \{\overline{p_n \rightarrow e^o_n}\}$	(rigid case)
	|	$fcase\ e^o\ of\ \{\overline{p_n \rightarrow e^o_n}\}$	(flexible case)
	|	$e^o_1\ or\ e^o_2$	(disjunction)
	|	$let\ \overline{x_n = e^o_n}\ in\ e^o$	(let binding)
	|	$prim\_g(x_1, x_2)$	(primitive functions)
	|	$hnf(x_1, x_2)$	(head normal form evaluator)
	|	$Obs(e^o)$	(expression observer)
	|	$obs\_f(\overline{x_n})$	(function observer)
p	::=	$C(\overline{x_n})$	

P^o	$\in$	$Prog^o$	
e^o, e^o_i	$\in$	Exp^o	
x, x_i	$\in$	Var	$where\ i \in \{1, \ldots, n\}$

Figure 6.3: Labeled flat program

the program is transformed as:

$main\ =\ let\ a = 1,\ b = 2\ \ in\ \ f^o(a, b)$

We can also observe only the result of the Curry application $f\ \ 1\ \ 2$ as a selected expression:

$main = let\ a = 1,\ b = 2,\ c = O(f(a, b))\ \ in\ \ c$

Note that, with respect to the third step of the labeling process, the selected expression $O(f(a, b))$ is represented by a binding to the fresh variable c.

6.2.2 *Extracting Events*

We assume that a program has already been transformed and can be used for representing the dynamic semantics of our work. Executing labeled expressions or functions causes events to be generated (see Section 5.2). Events are pieces of information about evaluation

steps of labeled expressions. In order to collect events, we use a list of tree structures named *evaluation trees*:

$T,\ T',\ T'' \ \in \ EvalTree^*$

Evaluation trees are constructed by different events:

$$\begin{array}{rcl} EvalTree & ::= & Demand(ref, [EvalTree]) \\ & | & Value(ref, String, [EvalTree]) \\ & | & Fun(ref, [EvalTree]) \\ & | & LogVar(ref, [EvalTree]) \\ & | & Open(ref) \end{array}$$

The references (ref) are unique identifiers to distinguish the events (e.g., in the implementation, we use integers as references and in our observation semantics, we use variables). They help us to define a relationship between different nodes (events) of the tree (as defined in the third step of the labeling process). The string in the value event is the name of the constructor of a value that is represented to the user as a textual visualization. The open node is always ready to receive a new event to extend the tree. When an evaluation fails, the evaluation trees contain intermediate information about evaluations until a failure appears.

Let us discuss the operational semantics of our work with respect to the defined events. To represent the operational semantics of observations, we need four components:

- a *labeled program* P^o ($\in Prog^o$) that contains labeled functions and expressions.
- a list of *evaluation trees* ($\in EvalTree^*$) to collect generated events.
- a *heap* ($\in Heap$) to illustrate the lazy semantics and sharing that can also contain bindings for labeled expressions.
- a labeled *expression* or a non-labeled one ($\in Exp^o$) that is being evaluated.

A labeled program represented as P^o is considered as a global parameter in our semantics. A heap is a binding from variables to expressions. Expressions can be labeled or unlabeled. A value in our semantics is a constructor-rooted or a logical variable:

$$\begin{array}{rcl} P^o & \in & Prog^o \\ \Gamma, \Delta, \Theta & \in & Heap = Var \mapsto Exp^o \\ e^o & \in & Exp^o \\ x, y & \in & Var \\ v & \in & Value ::= x \mid C(\overline{x_n}) \end{array}$$

In this semantics, an empty list of evaluation trees is represented by []. To insert an event into the related evaluation tree, we need the reference of this event. A reference shows to which open sub-tree a generated event belongs. A replacement of an open node by an event (*event*) with the reference ref in the list of evaluation trees *trees* is represented by $orepl_{event}(trees)$:

$$
\begin{aligned}
&orepl_{event}([]) && = [] \\
&orepl_{event}(Open(ref) : ts) && = \\
&\quad if\ getRef(event) = ref\ then\ (event : ts)\ else\ (Open(ref) : orepl_{event}(ts)) \\
&orepl_{event}(Demand(ref, tn) : ts) && = (Demand(ref, orepl_{event}(tn)) : orepl_{event}(ts)) \\
&orepl_{event}(Value(ref, str, tn) : ts) && = (Value(ref, str, orepl_{event}(tn)) : orepl_{event}(ts)) \\
&orepl_{event}(Fun(ref, tn) : ts) && = (Fun(ref, orepl_{event}(tn)) : orepl_{event}(ts)) \\
&orepl_{event}(LogVar(ref, tn) : ts) && = (LogVar(ref, orepl_{event}(tn)) : orepl_{event}(ts))
\end{aligned}
$$

$$
\begin{aligned}
getRef(Demand(ref, ts)) &= ref \\
getRef(Value(ref, str, ts)) &= ref \\
getRef(Fun(ref, ts)) &= ref \\
getRef(LogVar(ref, ts)) &= ref \\
getRef(Open(ref)) &= ref
\end{aligned}
$$

There are two different cases to insert a new generated event into the list of evaluation trees. If there is no open sub-tree that is related to the reference of the generated event, the latter is considered to be the root of a new evaluation tree. In this case, the new tree must be added to the list of evaluation trees. However, if the evaluation shares parts of other computations then, there is an event with an equal reference as the reference of the event generated. In this case, the event generated replaces the sub-tree with equal reference. This new generated tree is also inserted to the list of evaluation trees:

$$
\begin{aligned}
&rrepl_{event}([]) && = [] \\
&rrepl_{event}(Open(ref') : ts) && = \\
&\quad if\ ref = ref'\ then\ (event : ts)\ else\ (Open(ref') : rrepl_{event}(ts)) \\
&rrepl_{event}(Demand(ref', tn) : ts) && = \\
&\quad if\ ref = ref'\ then\ (event : ts) \\
&\qquad\qquad else\ (Demand(ref', rrepl_{event}(tn)) : rrepl_{event}(ts)) \\
&rrepl_{event}(Value(ref', str, tn) : ts) = \\
&\quad if\ ref = ref'\ then\ (event : ts) \\
&\qquad\qquad else\ (Value(ref', str, rrepl_{event}(tn)) : rrepl_{event}(ts))
\end{aligned}
$$

$$
\begin{aligned}
&rrepl_{event}(Fun(ref', tn) : ts) \quad = \\
&\quad if\ ref = ref'\ then\ (event : ts) \\
&\qquad\qquad else\ (Fun(ref', rrepl_{event}(tn)) : rrepl_{event}(ts)) \\
&rrepl_{event}(LogVar(ref', tn) : ts) \quad = \\
&\quad if\ ref = ref'\ then\ (event : ts) \\
&\qquad\qquad else\ (LogVar(ref', rrepl_{event}(tn)) : rrepl_{event}(ts))
\end{aligned}
$$

where in all rules $ref = getRef(event)$

This transformation of trees is abstracted in our derivation rules, as follows:

$$
\begin{aligned}
subst_{event}trees \ = \ & if\ \ oexists_{ref}(trees) \\
& \quad then\ \ orepl_{event}(trees) \\
& \quad else\ \ if\ \ rexists_{ref}(trees) \\
& \qquad then\ \ (rrepl_{event}(trees) : trees) \\
& \qquad else\ \ (event : trees)
\end{aligned}
$$

where $ref = getRef(event)$

$$
\begin{aligned}
&rexists_{ref}([]) \quad = False \\
&rexists_{ref}(Open(ref') : ts) \quad = \\
&\quad if\ \ ref = ref'\ \ then\ \ True \\
&\qquad\qquad else\ \ rexists_{ref}(ts) \\
&rexists_{ref}(Demand(ref', tn) : ts) \quad = \\
&\quad if\ \ ref = ref'\ \ then\ \ True \\
&\qquad\qquad else\ \ or(rexists_{ref}(tn), rexists_{ref}(ts)) \\
&rexists_{ref}(Value(ref', str, tn) : ts) \ = \\
&\quad if\ \ ref = ref'\ \ then\ \ True \\
&\qquad\qquad else\ \ or(rexists_{ref}(tn), rexists_{ref}(ts)) \\
&rexists_{ref}(Fun(ref', tn) : ts) \quad = \\
&\quad if\ \ ref = ref'\ \ then\ \ True \\
&\qquad\qquad else\ \ or(rexists_{ref}(tn), rexists_{ref}(ts)) \\
&rexists_{ref}(LogVar(ref', tn) : ts) \quad = \\
&\quad if\ \ ref = ref'\ \ then\ \ True \\
&\qquad\qquad else\ \ or(rexists_{ref}(tn), rexists_{ref}(ts))
\end{aligned}
$$

$$
\begin{aligned}
&oexists_{ref}([]) &&= False\\
&oexists_{ref}(Open(ref') : ts) &&=\\
&\quad if\ \ ref = ref'\ \ then\ \ True\ \ else\ \ False\\
&oexists_{ref}(Demand(ref', tn) : ts) &&=\\
&\quad or(oexists_{ref}(tn), oexists_{ref}(ts))\\
&oexists_{ref}(Value(ref', str, tn) : ts) &&=\\
&\quad or(oexists_{ref}(tn), oexists_{ref}(ts))\\
&oexists_{ref}(Fun(ref', tn) : ts) &&=\\
&\quad or(oexists_{ref}(tn), oexists_{ref}(ts))\\
&oexists_{ref}(LogVar(ref', tn) : ts) &&=\\
&\quad or(oexists_{ref}(tn), oexists_{ref}(ts))
\end{aligned}
$$

Note that no application of *oexists* and *rexists* is used in the implemented debugging system. To implement the debugger, we used indices instead of references. In our implementation, the first generated event, by the application of an observer function, has a special index (-1). This index shows that its related event must be the root of a new evaluation tree (see Chapter 5). In the operational semantics, we use *oexists* and *rexists* to control whether the generated event is the first generated event of an observer function or has a parent.

Labeled expressions are arbitrary expressions selected by the user. Thus, for every rule defined in Table 6.1, we need two different rules that specify whether or not an expression is selected. If an expression or a function is not selected to be observed (non-labeled expressions or functions), the definition of the rule is like the rules of Table 6.1. In this case, the evaluation trees are updated when a labeled sub-expression is evaluated. In the observation semantics, a proof of a judgment in a rule such as $T, \Gamma : e^o \Downarrow^o T', \Omega : v$ means that the expression e^o is evaluated to the value v where the list of evaluation trees T is updated to T'. For instance, if only one sub-expression of a case expression is selected:

$case\ O(e^o)\ of\ \{\overline{p_k \rightarrow e_k^o}\}$

then the rule $Case$ from Table 6.1 is represented as:

$$\frac{T, \Gamma : O(e^o) \Downarrow^o T', \Delta : C(\overline{y_n}) \qquad T', \Delta : \rho(e_i^o) \Downarrow^o T'', \Theta : v}{T, \Gamma : case\ O(e^o)\ of\ \{\overline{p_k \rightarrow e_k^o}\} \Downarrow^o T'', \Theta : v}$$

where $p_i = C(\overline{x_n})$ and $\rho = \{\overline{x_n \mapsto y_n}\}$. The same definition holds for other possible labeled sub-expressions of case, let and or expressions (see Table 6.2).

We also need extra rules to generate demand events. They are used when labeled (sub-)expressions or functions are evaluated. As we have discussed in Chapter 5, begin-

Table 6.2: Derivation Rules to Observe Sub-Expressions

$OOr: \quad \dfrac{T,\Gamma : e_i^o \quad \Downarrow^o \quad T',\Delta : v}{T,\Gamma : e_1^o \; or \; e_2^o \quad \Downarrow^o \quad T',\Delta : v}$ $where \;\; i \in \{1,2\}$
$OLet: \quad \dfrac{T,\Gamma[\overline{y_n \mapsto \rho(e_k^o)}] : \rho(e^o) \quad \Downarrow^o \quad T',\Delta : v}{T,\Gamma : let\{\overline{x_n = e_k^o}\} \; in \; e^o \quad \Downarrow^o \quad T',\Delta : v}$ $where \;\; \rho = \{\overline{x_n \mapsto y_n}\} \;\; and \;\; \overline{y_n} \; are \; fresh$
$OCase: \quad \dfrac{T,\Gamma : e^o \quad \Downarrow^o \quad T',\Delta : C(\overline{y_n}) \qquad T',\Delta : \rho(e_i^o) \quad \Downarrow^o \quad T'',\Theta : v}{T,\Gamma : (f) case \; e^o \; of \; \{\overline{p_k \rightarrow e_k^o}\} \quad \Downarrow^o \quad T'',\Theta : v}$ $where \;\; p_i = C(\overline{x_n}) \;\; and \;\; \rho = \{\overline{x_n \mapsto y_n}\}$

ning observations first generates demand events and then activates observers to generate events that are related to the type of annotated expressions. In the following, we first discuss demand generators (Table 6.3) and then we briefly explain the derivation rules of the observation semantics for expressions or functions that are selected for observation (Table 6.4).

Look at Figure 6.3. The syntax of e^o represents different observations: (1) Observation of selected expressions ($O(e^o)$), (2) Observation of selected function calls ($f^o(\overline{x_n})$) and (3) Observation of selected function definitions ($f^o(\overline{x_n}) = e^o$). These three observations need three different rules to generate demands (see Table 6.3):

$OFunDem$: This rule is used when a labeled function call is evaluated. In this case, the rule generates a demand event that is inserted into the list of evaluation trees. The demand event contains an open sub-tree that is extended by a possible continuation of the evaluations. $OFunDem$ also applies an observer to the function f by converting the name of the function to obs_f. This helps us to formalize the derivation rule of function calls to observe evaluated arguments and results of the function.

$ODefDem$: This rule is used when a function call is evaluated and there is a labeled function definition in the program. The rule generates a demand event and applies an observer to the function. With this method, we guarantee that every application of this

Table 6.3: Demand Generators

$$OFunDem: \quad \frac{subst_{Demand(f,[Open(f)])}T, \Gamma : obs\_f(\overline{x_n}) \Downarrow^o T', \Delta : v}{T, \Gamma : f^o(\overline{x_n}) \Downarrow^o T', \Delta : v} \quad where \;\; f(\overline{y_n}) = e^o \in P^o$$

$$ODefDem: \quad \frac{subst_{Demand(f,[Open(f)])}T, \Gamma : obs\_f(\overline{x_n}) \Downarrow^o T', \Delta : v}{T, \Gamma : f(\overline{x_n}) \Downarrow^o T', \Delta : v} \quad where \;\; f^o(\overline{y_n}) = e^o \in P^o$$

$$OVarDem: \quad \frac{subst_{Demand(x,[Open(x)])}T, \Gamma[x \mapsto Obs(e^o)] : x \Downarrow^o T', \Delta : v}{T, \Gamma[x \mapsto O(e^o)] : x \Downarrow^o T', \Delta : v}$$

function is observed although only the function definition is selected by the user.

Note that, if a function call and also its related function definition are selected in order to be observed, the generation of events will be performed once and represented twice to the user.

$OVarDem$: With respect to the third step of the labeling process (Section 6.2.1), all labeled expressions represented as $O(e^o)$ are bound to fresh variables by let bindings. $OVarDem$ is used when such variables (x) are evaluated. The rule generates a demand event and updates the heap by applying the related observer to the expression. In our semantics, this application is defined by labeling expressions with $Obs : Obs(e^o)$. Using expression observers, we formalize the derivation rules for the generation of events.

Now we assume that the execution of a labeled expression has begun and a demand event has already been generated. In the following, we explain how the next events are generated (see Table 6.4) during evaluating the expression:

$OVarCons$: This rule is used in order to evaluate a variable (x) that has already begun to be observed and is bound to an observed constructor term ($x \mapsto Obs(C(\overline{x_n}))$) in the heap. The rule updates the heap by a new binding of the variable to the term ($x \mapsto C(\overline{x_n})$)

Table 6.4: Derivation Rules for Observe Annotations

OVarCons :

$$T, \Gamma[\overline{y_n \mapsto e_n^o}, x \mapsto Obs(C(\overline{y_n}))] : x \;\Downarrow^o\; subst_{event} T, \Gamma[\overline{y_n \mapsto O(e_n^o)}, x \mapsto C(\overline{y_n})] : C(\overline{y_n})$$

$$where \;\; event := Value(x, "C", [\overline{Open(y_n)}])$$

OVarExp :

$$\frac{T, \Gamma[x \mapsto e^o] : e^o \;\Downarrow^o\; T', \Delta[\overline{y_n \mapsto e_n^o}] : C(\overline{y_n})}{T, \Gamma[x \mapsto Obs(e^o)] : x \;\Downarrow^o\; subst_{Value(x,"C",[\overline{Open(y_n)}])} T', \Delta[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto O(e_n^o)}] : C(\overline{y_n})}$$

$$\frac{T, \Gamma[x \mapsto e^o] : e^o \;\Downarrow^o\; T', \Delta[y \mapsto y] : y}{T, \Gamma[x \mapsto Obs(e^o)] : x \;\Downarrow^o\; subst_{LogVar(x,[Open(y)])} T', \Delta[x \mapsto y, y \mapsto O(y)] : y}$$

OVal :

$$T, \Gamma[x \mapsto Obs(x)] : x \;\Downarrow^o\; subst_{LogVar(x,[Open(x)])} T, \Gamma[x \mapsto O(x)] : x$$

$$T, \Gamma : C(\overline{x_n}) \;\Downarrow^o\; T, \Gamma : C(\overline{x_n})$$

OFunCall :

$$\frac{subst_{Fun(f,[\overline{Open(arg)}, Open(result)])} T, \Gamma : observeArgs \;\Downarrow^o\; T', \Delta : v}{T, \Gamma : obs\_f(\overline{x_n}) \;\Downarrow^o\; T', \Delta : v}$$

$$where \;\; \overline{arg_n} \; and \; result \; are \; fresh \; variables$$

$$observeArgs = let \; \overline{arg_n = O(x_n)}, result = O(\rho(e^o)) \; in \; result$$
$$\rho = \{\overline{y_n \mapsto arg_n}\}$$
$$f(\overline{y_n}) = e^o \in P^o$$

OFCase

$$\frac{T, \Gamma : e^o \;\Downarrow^o\; T', \Delta[x \mapsto O(x)] : x \quad subst_{event} T', \Delta[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i^o) \;\Downarrow^o\; T'', \Theta : v}{T, \Gamma : fcase \; e^o \; of \{\overline{p_k \to e_k^o}\} \;\Downarrow^o\; T'', \Theta : v}$$

$$where \;\; p_i = C(\overline{x_n})$$
$$\rho = \{\overline{x_n \mapsto y_n}\}$$
$$event := (Value(x, "C", [\overline{Open(y_n)}]))$$
$$\overline{y_n} \; are \; fresh$$

and replaces the related open node of the evaluation tree by a value event. This value event has a list of open sub-trees that belong to the constructor arguments. Constructor arguments are labeled ($\overline{y_n \mapsto O(e_n^o)}$) and will be observed in a possible evaluation of them.

OVarExp: It is used to observe a variable (x) that has already begun to be observed and is bound to an observed expression ($x \mapsto Obs(e^o)$) in the heap. The variable (x) is evaluated to a value only if the expression e^o is evaluated to this value. The returned value by evaluating e^o is a constructor-rooted term or a logical variable. As a result, there are two different rules for observations:

- *Constructor*: If e^o is evaluated to a constructor term $C(\overline{y_n})$, then the binding $x \mapsto C(\overline{y_n})$ replaces the original binding in the heap. This replacement is made because of the semantics of sharing. This means that another (possible) execution of x can share the result recorded in the heap. In this step, the proof is completed and the observed variable is evaluated to a constructor term that updates its evaluation tree with a value event and a list of open sub-trees for constructor arguments. Note that the variable arguments of the constructor are labeled ($\overline{y_n \mapsto O(e_n^o)}$). By this method, we guarantee that evaluated variable arguments are observed.

- *Logical variable*: If the expression e^o is evaluated to a logical variable (y), the evaluation tree is updated by a $LogVar$ event with an open sub-tree. The open sub-tree belongs to the logical variable by which a possible binding will be observed. This means that the original binding of the logical variable ($y \mapsto y$) is replaced by a labeled logical variable ($y \mapsto O(y)$).

OVal: This rule is used, if the observation of a logical variable (x) has already begun and there is a binding for this variable in the heap. *OVal* simply replaces this logical variable on itself and updates the tree by a LogVar event. The heap is also updated by labeling the logical variable ($x \mapsto O(x)$). By this method, we guarantee that every binding of this variable is observed.

Note that constructor-rooted terms do not appear as a Var in the heap ($Heap = Var \mapsto Exp^o$). For a derivation of these terms, we use the related rule defined in Table 6.1. The evaluation trees are only transfered from one step to another one.

OFunCall: Every time an observed function call ($obs\_f(\overline{x_n})$) is evaluated, this rule is used. The rule inserts a Fun event into the list of evaluation trees. The open sub-trees of the Fun event belong to the arguments and the result of the related function. These sub-trees can be extended whenever the arguments or results are evaluated. The evaluated function is also reduced to a let expression. The let expression binds the fresh variables

$\overline{arg_n}$ and *result* to the labeled arguments and result. By this method, we guarantee that evaluated arguments are observed by the user.

OFCase: If the main argument (e^o) of a case expression is evaluated to a labeled logical variable, this rule is used. In comparison to *FCase*, we need a value event to represent the related value (p_i) of the observed logical variable to the user.

OOr, *OLet* and *OCase*: If or-, let- or case expressions are selected in order to be observed, $OVarExp$ is used. We can also select sub-expressions of such expressions in order to be observed. In this case, events generated are depend on the labeled sub-expressions (see Table 6.2).

The definitions and rules should become clear in the following simple program that is written in Curry:

$f\ x\ =\ x$

$main\ =\ f\ 1$

In the Curry program above, we would like to observe the result of the application $f\ 1$ in the right-hand side of *main*. For this purpose, (1) the program must be normalized, (2) the selected expression must be labeled and (3) the labeled expression must be bound to a fresh variable:

$f(x)\ =\ x$

$main\ =\ let\ y = 1,$

$\qquad\qquad z = O(f(y))\ \ in$

$\qquad z$

To follow the evaluation order, we need an empty list to represent the evaluation trees, an empty heap and an expression to execute:

$[], [] : main$

We briefly explain each step of the evaluation that is represented in Figure 6.4[1].

Execution of the expression *main* requires $VarExp$. This rule reduces *main* to its right-hand side:

$[], [] : let\ y = 1,\ z = O(f(y))\ \ in\ \ z$

Now a let expression is being evaluated. In this step, *Let* is used:

$[], [y \mapsto 1, z \mapsto O(f(y))] : z$

Note that the variable z is bound to a labeled expression ($z \mapsto O(f(y))$) in the heap. In this step, $OVarDem$ generates a demand event. Because this demand event is the only event generated, it is considered to be the root of an evaluation tree ($subst_{event}[] = [event]$).

[1]For illustration purposes in all examples of this section, we represent a demand node by D, a value by V, a function by F and an open node by P

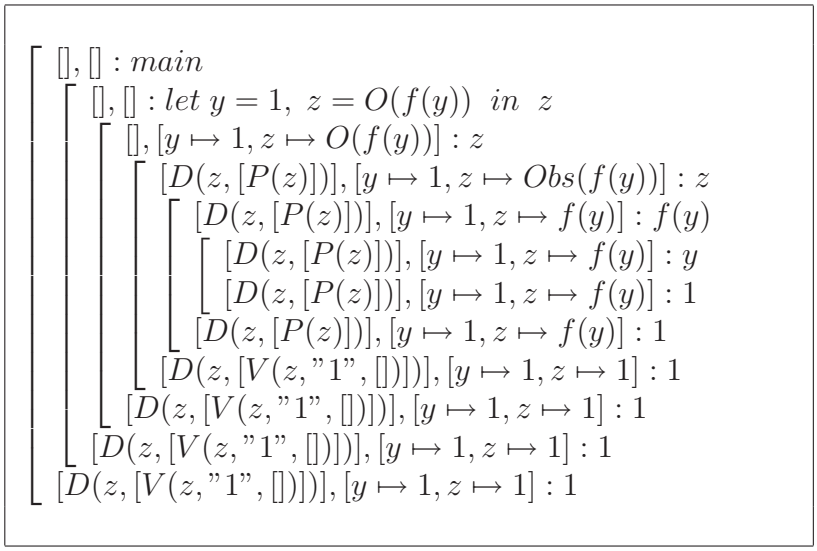


Figure 6.4: Derivation tree of the application $f(1)$

The generated demand has an open sub-tree with the reference z in which a possible continuation of evaluations of z is substituted. The rule also applies an observer to the expression and updates the heap:

$[D(z, [P(z)])], [y \mapsto 1, z \mapsto Obs(f(y))] : z$

Now $OVarExp$ is used:

$[D(z, [P(z)])], [y \mapsto 1, z \mapsto f(y)] : f(y)$

The derivation by this rule can be completed when the expression $f(y)$ is evaluated (using Fun):

$[D(z, [P(z)])], [y \mapsto 1, z \mapsto f(y)] : y$

A variable is being evaluated. This variable is bound to a constructor-rooted term in the heap ($y \mapsto 1$). This requires a derivation by $VarCons$:

$[D(z, [P(z)])], [y \mapsto 1, z \mapsto f(y)] : 1$

In this step, $VarCons$ is completed which causes Fun to be completed:

$[D(z, [P(z)])], [y \mapsto 1, z \mapsto f(y)] : 1$

and after that $OVarExp$:

$[D(z, [V(z, "1", [])])], [y \mapsto 1, z \mapsto 1] : 1$

Look at the evaluation tree. When the evaluation of z has been completed, the open sub-tree of the related reference z is replaced by a value event. The generated value (1) has no arguments (arity=0). Therefore, we have inserted an empty list as the sub-tree of the generated event. The heap is also updated by binding z to 1.

Finally, Let, and after that $VarExp$, are completed, both by:

$[D(z, [V(z, "1", [])])], [y \mapsto 1, z \mapsto 1] : 1$

The generated evaluation tree shows that only the result of the selected expression is observed. Let us compare the derivation tree of the observation semantics to the natural semantics of functional logic languages represented in Section 6.1.1. The difference appears when the evaluation of labeled expressions begins. The observation system requires subproofs to generate events (by the rules $OVarDem$ and $OVarExp$ in the above example), whereas the natural semantics does not. Furthermore, in the observation semantics, a heap contains extra bindings for the results of labeled expressions ($z \mapsto 1$ in the above example).

Let us consider another simple program written in Curry:

$f\ x\ =\ x$

$main\ =\ f\ 1\ +\ f\ 2$

In this example, the identity function f is applied twice in the expression ($f\ 1 + f\ 2$). We would like to observe the function definition of f. As we have discussed in Section 4.2, top-level function definitions can be annotated by observer functions, so that every application of these functions is observed.

To represent the observation semantics, the program must be transformed and the function definition must be labeled as follows:

$$
\begin{array}{lcl}
f^o(x) & = & x \\
main & = & let\ a = f(c), \\
 & & \quad b = f(d), \\
 & & \quad c = 1, \\
 & & \quad d = 2\ \ in \\
 & & +(a,b) \\
+(a,b) & = & let\ y = prim\_+(a,b), \\
 & & \quad x = hnf(b,y)\ \ in \\
 & & hnf(a,x)
\end{array}
$$

To start the evaluation, we need the observed program P^o that contains the transformed function definitions. We also need an empty list to represent evaluation trees, an empty heap and an expression for being evaluated:

$[], [] : main$

In the next step, $VarExp$ is used and $main$ is evaluated to a let expression:

$[], [] : let\ a = f(c), b = f(d), c = 1, d = 2\ \ in\ \ +(a,b)$

The evaluation is continued using the derivation rules:

$$
\begin{array}{l}
[],[] : main \quad -- \text{ using VarExp} \\
\quad [],[] : let\ a = f(c), b = f(d), c = 1, d = 2\ in\ +(a,b) \quad -- \text{ using Let} \\
\quad\quad [],[M] : +(a,b) \quad -- \text{ using Fun} \\
\quad\quad\quad [],[M] : let\ y = prim\_+(a,b), x = hnf(b,y)\ in\ hnf(a,x) \quad -- \text{ using Let} \\
\quad\quad\quad\quad [],[M, y \mapsto prim\_+(a,b), x \mapsto hnf(b,y)] : hnf(a,x) \quad -- \text{ using HNF} \\
\quad\quad\quad\quad\quad \left.\vdots\right\} \text{ the derivation of “}hnf(a,x)\text{”} \\
\quad\quad\quad\quad\quad X \\
\quad\quad\quad X \\
\quad\quad X \\
\quad X \\
X
\end{array}
$$

where $M := a \mapsto f(c), b \mapsto f(d), c \mapsto 1, d \mapsto 2$

The evaluation of *main* can be reduced to X, when the evaluation of $hnf(a,x)$ yields X. To evaluate the hnf function (by rule HNF), a and x must first be evaluated. The derivation of a is presented in Figure 6.5.

The variable x reduces to an hnf function (see Figure 6.6). The evaluation method of the first argument (b) of this hnf is the same as the evaluation of a. The second argument y needs a computation of the primitive function $prim\_+$.

After evaluating a and x, the application $hnf(a,x)$ is evaluated to 3:

$$
\begin{array}{l}
[],[S] : hnf(a,x) \quad -- \text{ using HNF} \\
\quad [],[S] : a \\
\quad \left.\vdots\right\} \text{ the derivation of “}a\text{”} \\
\quad [H],[N, arg \mapsto 1, res \mapsto 1] : 1 \\
\quad [H],[N, arg \mapsto 1, res \mapsto 1] : x \\
\quad \left.\vdots\right\} \text{ the derivation of “}x\text{”} \\
\quad [H,H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3 \\
[H,H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3
\end{array}
$$

Now the result of the entire evaluation can be obtained:

Figure 6.5: The derivation tree of a

$[H],[N, arg \mapsto 1, res \mapsto 1] : x$ — using VarExp
- $[H],[N, arg \mapsto 1, res \mapsto 1] : hnf(b, y)$ — using HNF
  - $[H],[N, arg \mapsto 1, res \mapsto 1] : b$ — using VarExp
    - ⋮ *the derivation of "b"*
    - $[H, H'],[K, arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 2$
  - $[H, H'],[K, arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : y$
    - $[H, H'],[K, arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : prim\_ + (a, b)$
    - $[H, H'],[K, arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$
  - $[H, H'],[K', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$
- $[H, H'],[K', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$

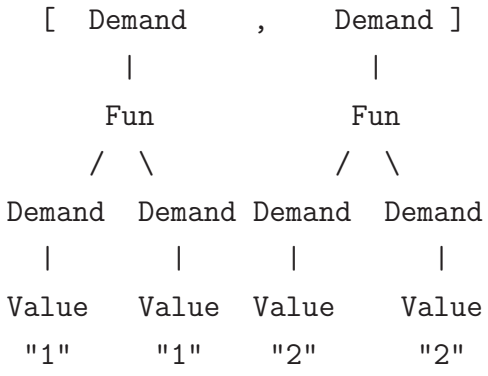
$[H, H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$

where
$H := D(f,[F(f,[D(arg,[V(arg,"1",[])]), D(res,[V(res,"1",[])])])])$
$H' := D(f,[F(f,[D(arg2,[V(arg2,"2",[])]), D(res2,[V(res2,"2",[])])])])$
$K := a \mapsto 1, \underline{b \mapsto 2}, c \mapsto 1, d \mapsto 2, y \mapsto prim\_ + (a, b), x \mapsto hnf(b, y)$
$K' := a \mapsto 1, b \mapsto 2, c \mapsto 1, d \mapsto 2, \underline{y \mapsto 3}, x \mapsto hnf(b, y)$
$K'' := a \mapsto 1, b \mapsto 2, c \mapsto 1, d \mapsto 2, y \mapsto 3, \underline{x \mapsto 3}$

Figure 6.6: The derivation tree of x

$[],[] : main$ — using VarExp
- $[],[] : let\ a = f(c), b = f(d), c = 1, d = 2\ \ in\ \ +(a, b)$ — using Let
  - $[],[M] : +(a, b)$ — using Fun
    - $[],[M] : let\ y = prim\_ + (a, b), x = hnf(b, y)\ in\ hnf(a, x)$ — using Let
      - $[],[S] : hnf(a, x)$
      - ⋮ *the derivation of "hnf(a, x)"*
      - $[H, H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$
    - $[H, H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$
  - $[H, H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$
- $[H, H'],[K'', arg \mapsto 1, res \mapsto 1, arg2 \mapsto 2, res2 \mapsto 2] : 3$

The result is 3. The recorded evaluation trees show that the function f is applied twice and with two different inputs and outputs (1,1) and (2,2):



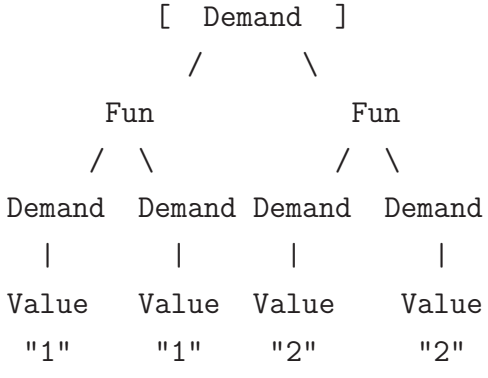
We have given a label only to the function definition of f, but all applications of this function are observed. The evaluation trees are represented to the user as: `{\1 -> 1}, {\2 -> 2}`.

Using higher-order functions in Curry, we can define a local pattern (g) that reduces to the function f of the latest example:

$f\ x\ =\ x$

$main\ =\ let\ g = f\ in\ g\ 1 + g\ 2$

Note that, if we select the pattern g in order to be observed, both generated trees are considered to be the sub-trees of the first generated demand event:



However, both applications of the function f are observed: `{\1 -> 1; \2 -> 2}`. We discuss the observation of higher-order functions in Section 6.2.3.

Observe annotations do not change the behavior of the lazy semantics in programs. To illustrate this claim, we consider another example by means of the sharing:

$main\ =\ let\ x = 1 + 2\ \ in\ \ x * x$

In this Curry program, we would like to observe the expression (`1+2`). We know that, because of the sharing strategy, the expression `1+2` is evaluated only once, but shared by both variables (x) in $x * x$. We use the derivation rules and show this definition.

The program must first be transformed and the expression x must be labeled:

$$
\begin{aligned}
main \;=\; & let\ a = 1, \\
& \quad b = 2, \\
& \quad c = O(x), \\
& \quad x = +(a, b)\ \ in \\
& *(c, c) \\
+(a, b) \;=\; & let\ w = prim\_+(a, b), \\
& \quad u = hnf(b, w)\ \ in\ \ hnf(a, u) \\
(a, b) \;=\; & let\ z = prim\_(a, b), \\
& \quad y = hnf(b, z)\ \ in\ \ hnf(a, y)
\end{aligned}
$$

All steps of the entire evaluation are presented in Figure 6.7. In the subproof1 of this figure, execution of the variable c is started and is evaluated to 3 ($c \mapsto 3$) at the end of the subproof. In this step, the open sub-tree of the demand event of observed element c is replaced by a value event.

In the subproof2, the second c of the expression $(c * c)$ is executed. However, because the heap contains a binding for c, the value to which c is bound is only replaced. The list of evaluation trees contains only one tree that belongs to the evaluation of the first occurred c.

Curry supports logical variables. To observe such variables, we consider an example written in Curry:

$$main \;=\; let\ x\ free\ \ in\ \ f\ x\ x$$

$$f\ True\ True \;=\; True$$

We would like to observe the logical variable x. To represent the observation semantics of this program, we need to perform the labeling process:

$$main \;=\; let\ x = O(x)\ in\ f(x, x)$$

$$f(x, y) \;=\; fcase\ x\ of\ \{True \rightarrow fcase\ y\ of\ \{True \rightarrow True\}\}$$

Now, we execute the program by calling $main$:

$[], [] : main$
$[], [] : let\ a = 1, b = 2, c = O(x), x = +(a, b)\ in\ *(c, c)$
$[], [a \mapsto 1, b \mapsto 2, x \mapsto +(a, b), c \mapsto O(x)] : *(c, c)$
$[], [a \mapsto 1, b \mapsto 2, x \mapsto +(a, b), c \mapsto O(x)] : let\ z = prim\_*(c, c), y = hnf(c, z)\ in\ hnf(c, y)$
$[], [B, c \mapsto O(x)] : hnf(c, y)$
$[], [B, c \mapsto O(x)] : c$
$[D(c, [P(c)])], [B, c \mapsto Obs(x)] : c$
$[D(c, [P(c)])], [B, c \mapsto x] : x$
$[D(c, [P(c)])], [B, c \mapsto x] : +(a, b)$
$[D(c, [P(c)])], [B, c \mapsto x] : let\ w = prim\_+(a, b),$
$u = hnf(b, w)\ in\ hnf(a, u)$
$[D(c, [P(c)])], [A, c \mapsto x] : hnf(a, u)$
$[D(c, [P(c)])], [A, c \mapsto x] : a$
$[D(c, [P(c)])], [A, c \mapsto x] : 1$
$[D(c, [P(c)])], [A, c \mapsto x] : u$
$[D(c, [P(c)])], [A, c \mapsto x] : hnf(b, w)$
$[D(c, [P(c)])], [A, c \mapsto x] : b$
$[D(c, [P(c)])], [A, c \mapsto x] : 2$
$[D(c, [P(c)])], [A, c \mapsto x] : w$
$[D(c, [P(c)])], [A, c \mapsto x] : prim\_+(a, b)$
$[D(c, [P(c)])], [A, c \mapsto x] : 3$
$[D(c, [P(c)])], [A1, c \mapsto x] : 3$
$[D(c, [P(c)])], [A1, c \mapsto x] : 3$
$[D(c, [P(c)])], [A2, c \mapsto x] : 3$
$[D(c, [P(c)])], [A2, c \mapsto x] : 3$
$[D(c, [P(c)])], [A2, c \mapsto x] : 3$
$[D(c, [P(c)])], [A2, c \mapsto x] : 3$
$[D(c, [P(c)])], [A3, c \mapsto x] : 3$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : 3$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : 3$ } $subproof1$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : y$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : hnf(c, z)$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : c$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : 3$ } $subproof2$
$[D(c, [V(c, 3", [])])], [A3, c \mapsto 3] : z$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : prim\_*(c, c)$
$[D(c, [V(c, "3", [])])], [A3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [A4, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [A4, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, \underline{y \mapsto 9}, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, y \mapsto 9, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, y \mapsto 9, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, y \mapsto 9, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, y \mapsto 9, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$
$[D(c, [V(c, "3", [])])], [a \mapsto 1, b \mapsto 2, x \mapsto 3, z \mapsto 9, y \mapsto 9, u \mapsto 3, w \mapsto 3, c \mapsto 3] : 9$

$where:$

$B := a \mapsto 1, b \mapsto 2, x \mapsto +(a, b), z \mapsto prim\_*(c, c), y \mapsto hnf(c, z)$
$A := B, \underline{u \mapsto hnf(b, w)}, \underline{w \mapsto prim\_+(a, b)}$
$A1 := B, u \mapsto hnf(b, w), \underline{w \mapsto 3}$
$A2 := B, \underline{u \mapsto 3}, w \mapsto 3$
$A3 := a \mapsto 1, b \mapsto 2, \underline{x \mapsto 3}, z \mapsto prim\_*(c, c), y \mapsto hnf(c, z), u \mapsto 3, w \mapsto 3$
$A4 := a \mapsto 1, b \mapsto 2, x \mapsto 3, \underline{z \mapsto 9}, y \mapsto hnf(c, z), u \mapsto 3, w \mapsto 3$

Figure 6.7: Representing sharing by observations

$$
\begin{array}{l}
[],[]:main \quad -- \text{ using VarExp} \\
\quad [],[]:let\ x = O(x)\ in\ f(x,x) \quad -- \text{ using Let} \\
\quad\quad [],[x \mapsto O(x)]:f(x,x) \quad -- \text{ using Fun} \\
\quad\quad\quad [],[x \mapsto O(x)]:fcase\ x\ of\{True \to fcase\ x\ of\{True \to True\}\} \quad -- \text{ using OFCase} \\
\quad\quad\quad\quad [],[x \mapsto O(x)]:x \quad -- \text{ using OVarDem} \\
\quad\quad\quad\quad\quad [D(x,[P(x)])],[x \mapsto Obs(x)]:x \quad -- \text{ using OVal} \\
\quad\quad\quad\quad\quad [D(x,[LogVar(x,[P(x)])])],[x \mapsto O(x)]:x \\
\quad\quad\quad\quad [D(x,[LogVar(x,[P(x)])])],[x \mapsto O(x)]:x \\
\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:fcase\ x\ of\ \{True \to True\} \\
\quad\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:x \\
\quad\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
\quad [D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True \\
[D(x,[LogVar(x,[V(x,"True",[])])])],[x \mapsto True]:True
\end{array}
$$

The rule $OVal$ updates the evaluation tree by a LogVar event. Using $OFCase$, the first occurred x is bound to $True$ and the open node of the LogVar event is replaced by a value event. The value of the evaluated variable is represented to the user as $(?/True)$. In this step, the heap contains a new binding for x. The logical variable x is used twice in the program. The second existed x of the program is also bound. However, because the heap contains a binding $(x \mapsto True)$ for x, the value to which x is bound is only replaced.

In the examples discussed, we have seen that each evaluation step of observed expressions is recorded in a list of trees. If a failure occurs during a program execution, recorded information helps the programmer to recognize whether or not the failure is located in the observed expression. Let us consider a faulty program that has been written in Curry:

$$f\ =\ h\ []$$

$$h\ (x:xs)\ =\ x$$

The function h in this program is considered to be similar to the function `head` in Curry. The program has no results, if the function h is applied to an empty list. In other words, calling the function f returns `No more solutions` in PAKCS [17]. We would like to select the function definition of h and observe the behavior of each possible application of this

function in the program. For this purpose, we need a labeled program in flat form:

$f \ = \ let\ a = []\ \ in\ \ h(a)$

$h^o(xs) \ = \ case\ xs\ of\ \ \{(y : ys) \rightarrow y\}$

We start the evaluation, by executing f when P^o contains the transformed function definitions:

$[], [] : f$

The evaluation will be followed using derivation rules. The related derivation tree is presented in Figure 6.8. Look at this derivation tree. The first subproof of the case rule is terminated by binding the main argument (arg) of the case expression to an empty list. In the second subproof, the empty list must be matched with the patterns in the case expression. In our example, there is only one pattern in the case expression that is not matched with the empty list.

There is a failure. The PAKCS system yields `No more solutions` and our observation system represents the information recorded in the evaluation tree

```
      Demand
        |
       Fun
       / \
Demand   Demand
  |
Value
"[]"
```

to the user in a textual form (`{\[] -> !}`). It shows that the observed function h takes an empty list as input and begins to evaluate the right-hand side, but yields no result (!).

6.2.3 *Observation in Curried Form*

We have represented an observation semantics for our implemented observation system. This semantics is an extension of [2]. In order to provide a simple operational description, [2] assumes that programs are translated into a flat form.

In our observation system, a function is evaluated in curried form. This means that the result of a function can be a function again. For example, a function application like (f 1 2) for the function definition $f\ x\ y = x + y$ is translated to ((f 1) 2). However, the first-order definition of flat programs does not allow a representation of such a higher-order functions. For higher-order functions, we need extra transformations in programs.

$[], [] : f$ $\quad -- \text{ using VarExp}$

$[], [] : let\ a = []\ in\ h(a)$ $\quad -- \text{ using Let}$

$[], [a \mapsto []] : h(a)$ $\quad -- \text{ using ODefDem}$

$[D(h, [P(h)])], [a \mapsto []] : obs\_h(a)$ $\quad -- \text{ using OFunCall}$

$[D(h, [F(h, [P(arg), P(res)])])],$

$[a \mapsto []] : let\ arg = O(a), res = O(case\ arg\ of\ L)\ in\ res$

$[D(h, [F(h, [P(arg), P(res)])])],$

$[a \mapsto [], arg \mapsto O(a), res \mapsto O(case\ arg\ of\ L)] : res$

$[D(h, [F(h, [P(arg), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto O(a), res \mapsto Obs(case\ arg\ of\ L)] : res$

$[D(h, [F(h, [P(arg), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto O(a), res \mapsto (case\ arg\ of\ L)] : case\ arg\ of\ L$

$[D(h, [F(h, [P(arg), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto O(a), res \mapsto (case\ arg\ of\ L)] : arg$

$[D(h, [F(h, [D(arg, [P(arg)]), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto Obs(a), res \mapsto (case\ arg\ of\ L)] : arg$

$[D(h, [F(h, [D(arg, [P(arg)]), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto a, res \mapsto (case\ arg\ of\ L)] : a$

$[D(h, [F(h, [D(arg, [P(arg)]), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto a, res \mapsto (case\ arg\ of\ L)] : []$

$[D(h, [F(h, [D(arg, [V(arg, "[]", [])]), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto [], res \mapsto (case\ arg\ of\ L)] : []$

$[D(h, [F(h, [D(arg, [V(arg, "[]", [])]), D(res, [P(res)])])])],$

$[a \mapsto [], arg \mapsto [], res \mapsto (case\ arg\ of\ L)] : []$

the second subproof of caserule fails

where $L := \{(y : ys) \to y\}$

Figure 6.8: Observation of a faulty program

Table 6.5: Derivation Rules for Partial Applications

PrimApply :

$$\frac{\Gamma[z \mapsto prim\_apply(x,y)] : x \;\Downarrow\; \Delta : \varphi(\overline{x_k})}{\Gamma[z \mapsto prim\_apply(x,y)] : z \;\Downarrow\; \Delta[z \mapsto \varphi(\overline{x_k},y)] : \varphi(\overline{x_k},y)}$$

where $\varphi(\overline{y_n}) = e \in P$ with $k < n-1$

$$\frac{\Gamma[z \mapsto prim\_apply(x,y)] : x \;\Downarrow\; \Delta : \varphi(\overline{x_k}) \qquad \Delta[z \mapsto \varphi(\overline{x_k},y)] : z \;\Downarrow\; \Theta : v}{\Gamma[z \mapsto prim\_apply(x,y)] : z \;\Downarrow\; \Theta[z \mapsto v] : v}$$

where $\varphi(\overline{y_n}) = e \in P$ with $k = n-1$

Apply :
$$\frac{\Gamma : let\ a = prim\_apply(x,y)\ in\ hnf(x,a) \;\Downarrow\; \Delta : v}{\Gamma : apply(x,y) \;\Downarrow\; \Delta : v}$$

PartVal : $\Gamma : \varphi(\overline{x_k}) \;\Downarrow\; \Gamma : \varphi(\overline{x_k})$

where $\varphi(\overline{y_n}) = e \in P$ with $k < n$

PartVarCons : $\Gamma[x \mapsto \varphi(\overline{x_k})] : x \;\Downarrow\; \Gamma[x \mapsto \varphi(\overline{x_k})] : \varphi(\overline{x_k})$

where $\varphi(\overline{y_n}) = e \in P$ with $k < n$

To avoid these transformations, all function applications are represented by a first-order semantics, as discussed in our observation semantics.

For representing higher-order functions, [2] suggests a translation of functions into applications of a distinguished function *apply*. This function can be defined by a set of first-order rules. For instance, the function application $((f\ 1)\ 2)$ can be translated to:

$let\ x = 1,\ y = 2,\ u = f,\ z = apply(u,x)\ \ in\ \ apply(z,y)$

The function *apply* is defined using the primitive function *prim_apply*:

$apply(x,y) = hnf(x, prim\_apply(x,y))$

In contrast to the definition of primitive functions in the structure of the flat programs (e.g., $x+y = hnf(x, hnf(y, prim\_+(a,b)))$), the second argument of *apply* does not need to be evaluated to head normal form. However, it is assumed that the first argument is in head normal form. Note that partial applications are considered to be a constructor-rooted term. This means that functions with missing arguments (e.g., $(f\ 1)$ in the example

above) are considered as constructor-rooted terms. Using this definition, we extend the set of values as follows:

$v \in Value ::= x \mid C(\overline{x_m}) \mid \varphi(\overline{x_k})$
where $\varphi(\overline{y_n}) = e \in P$ and $k < n$

The derivation rules of the primitive function *prim_apply* and the function *apply* are presented in Table 6.5, where φ is a constructor symbol or a function ($\varphi(\overline{y_n}) = e \in P$). The condition $k < n$ shows that the application $\varphi(\overline{x_k})$ can also be a partial application. Note that the syntax of expressions in flat programs (see Figure 6.1) is also extended:

$$\begin{array}{rcl} e & ::= & x \\ & \mid & \ldots \\ & \mid & prim\_apply(e_1, e_2) \\ & \mid & apply(e_1, e_2) \end{array}$$

We briefly explain the rules defined in Table 6.5:

PrimApply: If the variable evaluated (z) is bound to a *prim_apply* function in the heap and the first argument (x) of *prim_apply* is evaluated to a partial application ($\varphi(\overline{x_k})$) with $k < n - 1$, the first rule defined is used. This rule extends the partial application by the second argument of *prim_apply* as $\varphi(\overline{x_k}, y)$ and returns it as a result. However, if the first argument of *prim_apply* is evaluated to a partial application ($\varphi(\overline{x_k})$) with $k = n - 1$ then the extension of the partial application generates a function call which can be evaluated using $VarExp$ as defined in Table 6.1. In this case, the second rule of *PrimApply* is used.

Apply: If an *apply* function is being evaluated, first *apply* is reduced to its right-hand side then the right-hand side is evaluated. The result of this evaluation (v) can be a constructor-rooted term, a partial application ($\varphi(\overline{x_k})$) which is considered to be a constructor-rooted term, or a variable ($v \in Value$).

PartVal: If a partial application is being evaluated, it is returned as a value.

PartVarCons: If a variable evaluated (x) is bound to a partial application in the heap, this application is returned as a result.

To represent the observation semantics of a partial application, we extend the labeling process, as discussed in Section 6.2.1:

1. The program is normalized.

2. Function applications are transformed to *apply* functions.

3. Selected expressions (e) to be observed are labeled ($O(e)$).

4. The labeled expressions are bound to fresh variables ($let\ x = O(e)\ in\ x$).

For example, to observe an application of the function *add* to its first argument (*add* 1), in the following Curry program:

$$\begin{array}{l} add\ x\ y\ =\ x + y \\ main\ =\ let\ inc = add\ 1\ \ in \\ \qquad\qquad (inc\ 1 + inc\ 2) \end{array}$$

we transform the program as follows:

$$\begin{array}{l} add(x, y)\ =\ +(x, y) \\ main\ =\ let\ x = 1, \\ \qquad\qquad y = 2, \\ \qquad\qquad z = add, \\ \qquad\qquad inc = O(apply(z, x)), \\ \qquad\qquad a = apply(inc, x), \\ \qquad\qquad b = apply(inc, y)\ \ in\ \ +(a, b) \\ +(x, y)\ =\ let\ a = prim\_+(x, y), b = hnf(y, a)\ \ in\ \ hnf(x, b) \\ apply(x, y)\ =\ let\ a = prim\_apply(x, y)\ \ in\ \ hnf(x, a) \end{array}$$

Using this transformation, the derivation of the function application (*add* 1) can be represented to the user.

Note that, whenever the evaluation of a labeled expression is started, an observer is applied to it (see Section 6.2.1). Expression observers are represented by the label "*Obs*". In this section, labeled partial applications by observers are considered to be values:

$$\begin{array}{l} v \in Value^o ::= x \mid C(\overline{x_m}) \mid \varphi(\overline{y_k}) \mid Obs(\varphi(\overline{y_k})) \\ \qquad \text{where } \varphi(\overline{z_n}) = e^o \in P^o \text{ and } k < n \end{array}$$

Let us briefly discuss the observation rules for partial applications that are presented in Table 6.6 and 6.7 (for this purpose, we have also updated the function *subst*):

OApply: If the variable evaluated is bound to a labeled *apply* function, *OApply* is used. This rule generates a demand event and reduces *apply* to its right-hand side where *prim_apply* is labeled by an expression observer ($Obs(prim\_apply(x, y))$). This transformation is useful to formalize the generation of events by the other derivation rules. The result of evaluations can be an observed partial application ($Obs(\varphi(\overline{x_k}))$, a constructor-rooted term or a variable ($v \in Value^o$).

OPrimApply: If the observation of an *apply* function has already begun, its related *prim_apply* is labeled by an expression observer; in this case, *OPrimApply* is used. There

Table 6.6: Observation Rules for Partial Applications

OApply :

$$\frac{T1, \Gamma[z \mapsto apply(x,y)] : let\ a = Obs(prim\_apply(x,y))\ \ in\ \ hnf(x,a)\ \Downarrow^o\ T', \Delta : v}{T, \Gamma[z \mapsto O(apply(x,y))] : z\ \Downarrow^o\ T', \Delta[z \mapsto v] : v}$$

where $T1 := subst_{Demand(a,[Open(a)])}T$

OPrimApply :

$$\frac{T, \Gamma[z \mapsto prim\_apply(x,y)] : x\ \Downarrow^o\ T', \Delta : \varphi(\overline{x_k})}{T, \Gamma[z \mapsto Obs(prim\_apply(x,y))] : z\ \ \Downarrow^o\ \ T', \Theta[z \mapsto Obs(\varphi(\overline{x_k}, y))] : Obs(\varphi(\overline{x_k}, y))}$$

where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n-1$

$$\frac{T, \Gamma[z \mapsto prim\_apply(x,y)] : x\ \Downarrow^o\ T', \Delta : \varphi(\overline{x_k})\quad T', \Delta[z \mapsto Obs(\varphi(\overline{x_k}, y))] : z\ \Downarrow^o\ T'', \Theta : v}{T, \Gamma[z \mapsto Obs(prim\_apply(x,y))] : z\ \ \Downarrow^o\ \ T'', \Theta[z \mapsto v] : v}$$

where $\varphi(\overline{y_n}) = e \in P^o$ and $k = n-1$

OApplyResult :

$$\frac{T, \Gamma[z \mapsto apply(x,y)] : let\ a = prim\_apply(x,y)\ in\ hnf(x,a)\ \Downarrow^o\ T', \Delta : v}{T, \Gamma[z \mapsto apply(x,y)] : z\ \Downarrow^o\ T', \Delta[z \mapsto v] : v}$$

OPrimApplyResult :

$$\frac{T, \Gamma[z \mapsto prim\_apply(x,y)] : x\ \Downarrow^o\ T', \Delta : Obs(\varphi(\overline{x_k}))\quad T1, \Delta[z \mapsto E] : L\ \Downarrow^o\ T'', \Theta : V}{T, \Gamma[z \mapsto prim\_apply(x,y)] : z\ \ \Downarrow^o\ \ T'', \Theta[z \mapsto V] : V}$$

where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n-1$

$L := let\ arg = O(y)\ \ in\ \ z$

$V := Obs(\varphi(\overline{x_k}, arg))$

$E := O(\varphi(\overline{x_k}, arg))$

$T1 := subst_{Fun(\varphi,[Open(arg),Open(z)])}T'$

$$\frac{T, \Gamma[z \mapsto prim\_apply(x,y)] : x\ \Downarrow^o\ T', \Delta : Obs(\varphi(\overline{x_k}))\quad T1, \Delta[z \mapsto E] : L\ \Downarrow^o\ T'', \Theta : v}{T, \Gamma[z \mapsto prim\_apply(x,y)] : z\ \ \Downarrow^o\ \ T'', \Theta[z \mapsto v] : v}$$

where $\varphi(\overline{y_n}) = e \in P^o$ and $k = n-1$

$L := let\ arg = O(y)\ \ in\ \ z$

$E := O(\varphi(\overline{x_k}, arg))$

$T1 := subst_{Fun(\varphi,[Open(arg),Open(z)])}T'$

Table 6.7: The remainder Observation Rules for Partial Applications

$OPartVarDem$: $\dfrac{subst_{Demand(x,[Open(\varphi)])}T, \Gamma[x \mapsto Obs(\varphi(\overline{x_k}))] : x \ \Downarrow^o \ T', \Delta : Obs(\varphi(\overline{x_k}))}{T, \Gamma[x \mapsto O(\varphi(\overline{x_k}))] : x \ \Downarrow^o \ T', \Delta[x \mapsto Obs(\varphi(\overline{x_k}))] : Obs(\varphi(\overline{x_k}))}$ where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n$ (in the case of $k = n$, $OVarDem$ is used)
$OPartVarCons$: $T, \Gamma[x \mapsto Obs(\varphi(\overline{x_k}))] : x \ \Downarrow^o \ T, \Gamma[x \mapsto Obs(\varphi(\overline{x_k}))] : Obs(\varphi(\overline{x_k}))$ where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n$ (in the case of $k = n$, $OVarExp$ is used)
$OPartVal$: $T, \Gamma : Obs(\varphi(\overline{x_k})) \ \Downarrow^o \ T, \Gamma : Obs(\varphi(\overline{x_k}))$ where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n$

are two rules defined for $OPrimApply$. If the variable evaluated (z) is bound to an observed *prim_apply* in the heap and the first argument (x) of *prim_apply* is evaluated to a partial application ($\varphi(\overline{x_k})$) with $k < n - 1$, the first defined rule is used. This rule extends the partial application by the second argument of *prim_apply* as $Obs(\varphi(\overline{x_k}, y))$ and returns it as a result. However, if the first argument of *prim_apply* is evaluated to a partial application ($\varphi(\overline{x_k})$) with $k = n - 1$, then the extension of the partial application generates a function call which can be evaluated to a value (using $OVarExp$ as defined in Table 6.4). In this case, the second rule of $OPrimApply$ is used.

OApplyResult: If the variable evaluated (z) is bound to an *apply* function in the heap, this rule is used. The evaluation of an *apply* function yields an observed partial application, a constructor-rooted term or a variable ($v \in Value^o$).

OPrimApplyResult: It is used if the variable evaluated is bound to a *prim_apply* function. *OPrimApplyResult* first evaluates the first argument of *prim_apply* to an observed partial application ($Obs(\varphi(\overline{x_n}))$). Then, this partial application is extended by the second argument of *prim_apply*. Note that the second argument of *prim_apply* and also the extended partial application are labeled ($O(\varphi(\overline{x_k}, arg))$). By this method, we observe the result of the function application which can be a function again. There are two cases. If the evaluation of the first argument yields an application with missing arguments

($k < n-1$), the first rule defined is used. This rule returns the observed extended partial application as a result. In other case ($k = n-1$), the second rule evaluates the extended application to a value (using $OVarDem$ as defined in Table 6.3). Both rules generate a Fun event with two open sub-trees for the argument and result of the partial application.

$OPartVarDem$: If the variable evaluated (x) is bound to a labeled partial application in the heap ($x \mapsto O(\varphi(\overline{x_k}))$), this rule is used. The rule generates a demand event and applies an observer to the application. This observed application ($Obs(\varphi(\overline{x_k}))$) is returned as a result. If the labeled application ($O(\varphi(\overline{x_k}))$) in the heap is a function call with $k = n$ then, the rule $OVarDem$ from Table 6.3 is used.

$OPartVarCons$: If the evaluation of a labeled partial application has already begun, this rule is used. In this case, an expression observer is applied to the application in the heap ($x \mapsto Obs(\varphi(\overline{x_k}))$). $OPartVarCons$ returns this observed application as a result without any modification on the heap. Note that if the application $\varphi(\overline{x_k})$ is a function call with $k = n$, then the rule $OVarExp$ from Table 6.4 is used.

$OPartVal$: An observed partial application that is being evaluated, is returned unchanged.

As an example, we observe the partial application *add* in the following Curry program:

$$
\begin{aligned}
&add\ x\ y\ =\ x + y\\
&main = let\ \ inc = add\ 1\ \ in\\
&\qquad\qquad (inc\ 1 + inc\ 2)
\end{aligned}
$$

For this purpose, the program must first be transformed:

$$
\begin{aligned}
&add(x, y)\ =\ +(x, y)\\
&main\ =\ let\ x = 1,\\
&\qquad\qquad y = 2,\\
&\qquad\qquad z = O(add),\\
&\qquad\qquad inc = apply(z, x),\\
&\qquad\qquad a = apply(inc, x),\\
&\qquad\qquad b = apply(inc, y)\ \ in\ \ +(a, b)\\
&+(x, y)\ =\ let\ a = prim\_+(x, y), b = hnf(y, a)\ \ in\ \ hnf(x, b)\\
&apply(x, y)\ =\ let\ a = prim\_apply(x, y)\ \ in\ \ hnf(x, a)
\end{aligned}
$$

To present the operational semantics, we need two empty lists. The first list for the evaluation trees and the second one for the heap. We need also an expression to execute:

$[], [] : main$

To evaluate this expression, $VarExp$ as defined in Table 6.1 is used:

$$
\left[\begin{array}{l}
[],[]:main \quad -- \text{ using VarExp} \\
\left[\begin{array}{l}
[],[]:let\ A\ in\ +(a,b) \quad -- \text{ using Let} \\
\left[\begin{array}{l}
[],[A1]:+(a,b) \quad -- \text{ using Fun} \\
\left[\begin{array}{l}
[],[A1]:let\ c=prim\_+(a,b), d=hnf(b,c)\ in\ hnf(a,d) \quad -- \text{ using Let} \\
\left[\begin{array}{l}
[],[A2]:hnf(a,d) \quad -- \text{ using HNF} \\
\left.\begin{array}{l}\vdots\end{array}\right\} the\ derivation\ of\ \text{“}hnf(a,d)\text{”} \\
X
\end{array}\right. \\
X
\end{array}\right. \\
X
\end{array}\right. \\
X
\end{array}\right. \\
X
\end{array}\right.
$$

where: $A := x = 1, y = 2, z = O(add), inc = apply(z, x), a = apply(inc, x), b = apply(inc, y)$

$A1 := x \mapsto 1, y \mapsto 2, z \mapsto O(add), inc \mapsto apply(z, x), a \mapsto apply(inc, x), b \mapsto apply(inc, y)$

$A2 := A1, \underline{c \mapsto prim\_+(a, b)}, \underline{d \mapsto hnf(b, c)}$

$main$ is evaluated to X if $hnf(a, d)$ is evaluated to this result. The first argument of hnf is bound to an $apply$ function in the heap ($a \mapsto apply(inc, x)$) which needs a derivation by $OApplyResult$. The derivation tree that is related to a is presented in Figure 6.9. For illustration purposes, the heaps are abstracted as follows:

$A3 := x \mapsto 1, y \mapsto 2, \underline{z \mapsto Obs(add)}, inc \mapsto apply(z, x), a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), k \mapsto prim\_apply(inc, x), t \mapsto prim\_apply(z, x)$

$A4 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto apply(z, x), a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), k \mapsto prim\_apply(inc, x), \underline{t \mapsto O(add(arg))},$

$A5 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto apply(z, x), a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), k \mapsto prim\_apply(inc, x), \underline{t \mapsto Obs(add(arg))}, arg \mapsto O(x)$

$A6 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), \underline{inc \mapsto Obs(add(arg))}, a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), k \mapsto prim\_apply(inc, x), t \mapsto Obs(add(arg)), arg \mapsto O(x)$

$A7 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), \underline{k \mapsto O(add(arg, arg2)}, t \mapsto Obs(add(arg)), arg \mapsto O(x)$

$A8 := A7, \underline{arg2 \mapsto O(x)}$

$A9 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), a \mapsto apply(inc, x), b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), \underline{k \mapsto 2}, t \mapsto Obs(add(arg)), \underline{arg \mapsto 1}, \underline{arg2 \mapsto 1}$

$A10 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), \underline{a \mapsto 2}, b \mapsto apply(inc, y),$
$c \mapsto prim\_+(a, b), d \mapsto hnf(b, c), k \mapsto 2, t \mapsto Obs(add(arg)), arg \mapsto 1, arg2 \mapsto 1$

$TREE := D(z, [F(add, [D(arg, [V(arg, \text{"1"}, [])]),$
$D(t, [F(add, [V(arg2, \text{"1"}, []), D(k, [V(k, \text{"2"}, [])])])])])$

When the evaluation of a is complete, the second argument of $hnf(a, d)$ is evaluated. d is bound to a hnf function in the heap. The first argument of this function is b that

Figure 6.9: The derivation tree of "a"

must first be evaluated:

$[TREE], [A10] : b$ $\quad --$ using OApplyResult

- $[TREE], [A10] : let\ u = prim\_apply(inc, y)\ in\ hnf(inc, u)$ $\quad --$ using Let
  - $[TREE], [A10, \underline{u \mapsto prim\_apply(inc, y)}] : hnf(inc, u)$ $\quad --$ using HNF
    - $[TREE], [A10, u \mapsto prim\_apply(inc, y)] : inc \quad --$ using OPartVarCons
    - $[TREE], [A10, u \mapsto prim\_apply(inc, y)] : Obs(add(arg))$
    - $[TREE], [A10, u \mapsto prim\_apply(inc, y)] : u \quad --$ using OPrimApplyResult
      - $[TREE], [A10, u \mapsto prim\_apply(inc, y)] : inc \quad --$ using OPartVarCons
      - $[TREE], [A10, u \mapsto prim\_apply(inc, y)] : Obs(add(arg))$
      - $[TREE2], [A10, \underline{u \mapsto O(add(arg, arg3))}] : let\ arg3 = O(y)\ in\ u$
        - $[TREE3], [A10, u \mapsto O(add(arg, arg3)), \underline{arg3 \mapsto O(y)}] : u$
          - $[TREE3], [A10, \underline{u \mapsto Obs(add(arg, arg3))}, arg3 \mapsto O(y)] : u$
          - $\vdots$ *the derivation of "u" using OVarExp*
          - $[TREE4], [A10, \underline{u \mapsto 3}, \underline{arg3 \mapsto 2}] : 3$
        - $[TREE4], [A10, u \mapsto 3, arg3 \mapsto 2] : 3$
      - $[TREE4], [A10, u \mapsto 3, arg3 \mapsto 2] : 3$
    - $[TREE4], [A10, u \mapsto 3, arg3 \mapsto 2] : 3$
  - $[TREE4], [A10, u \mapsto 3, arg3 \mapsto 2] : 3$
- $[TREE4], [A11] : 3$

where:

$$A11 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), a \mapsto 2, c \mapsto prim\_+(a, b),$$
$$d \mapsto hnf(b, c), arg3 \mapsto 2, k \mapsto 2, t \mapsto Obs(add(arg)), arg \mapsto 1, arg2 \mapsto 1, \underline{b \mapsto 3}, u \mapsto 3$$

$$TREE2 := D(z, [F(add, [D(arg, [V(arg, "1", [])]),$$
$$D(t, [F(add, [D(arg2, [V(arg2, "1", [])]),$$
$$D(k, [V(k, "2", [])])]),$$
$$\underline{F(add, [P(arg3), P(u)])})])])$$

$$TREE3 := D(z, [F(add, [D(arg, [V(arg, "1", [])]),$$
$$D(t, [F(add, [D(arg2, [V(arg2, "1", [])]),$$
$$D(k, [V(k, "2", [])])]),$$
$$F(add, [P(arg3), \underline{D(u, P(u))}])])])$$

$$TREE4 := D(z, [F(add, [D(arg, [V(arg, "1", [])]),$$
$$D(t, [F(add, [D(arg2, [V(arg2, "1", [])]),$$
$$D(k, [V(k, "2", [])])]),$$
$$F(add, [\underline{D(arg3, [V(arg3, "2", [])])},$$
$$\underline{D(u, [V(u, "3", [])])}])])])$$

Look at the derivation tree. When $OPrimApplyResult$ evaluates the variable u, the first argument (inc) of its related $prim\_apply$ function shares the result ($Obs(add(arg))$)

and also the evaluation tree generated by this argument.

Now the second argument of $hnf(b, c)$ must be evaluated. It is bound to a primitive function ($prim\_+$) in the heap and needs a derivation by $PrimFun$:

$$\left[\begin{array}{l} [TREE4], [A11] : c \\ \quad \left[\begin{array}{l} [TREE4], [A11] : prim\_+(a, b) \\ [TREE4], [A11] : 5 \end{array}\right. \\ [TREE4], [..., c \mapsto 5, ...] : 5 \end{array}\right.$$

The derivation tree is complete (see Figure 6.10). The result of the evaluation of *main* is 5. The entire evaluation tree that is recorded in the first list generated by the derivation tree is represented to the user as a textual visualization:

```
{ \1 1 -> 2 ;
  \1 2 -> 3 }
```

Existing only one bracket means that parts of the function application of *add* are shared. The application (*add* 1) is shared for the both function calls ((*add* 1) 1) and ((*add* 1) 2).

6.2.4 *Correctness*

In this section, we prove the correctness of our observation semantics, where we denote:

T and T' are lists of evaluation trees ($T, T' \in EvalTree^*$) and

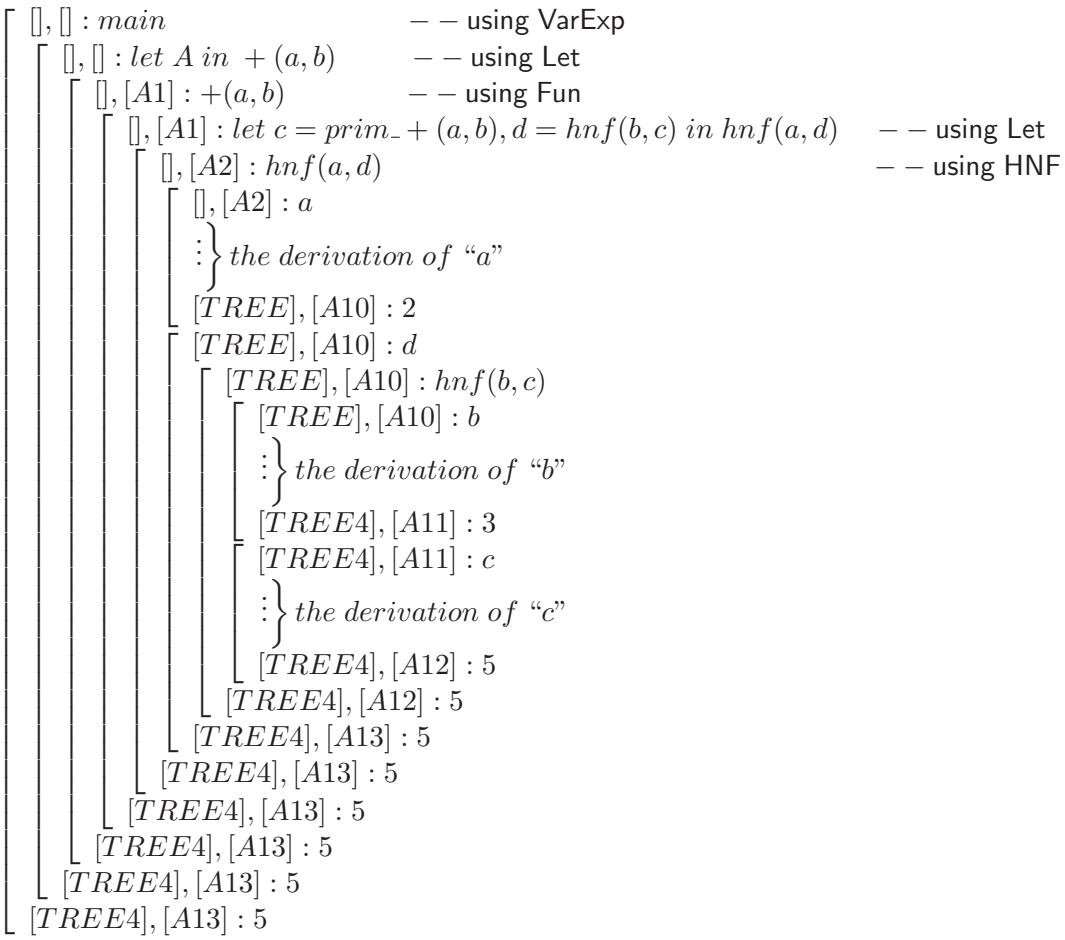
Ω, Γ, Δ and Θ heaps ($\Omega, \Gamma, \Delta, \Theta \in Heap$).

Definition 1:

To state the correctness theorem, we denote by $\natural(\Theta)$ a removal of all observation labels from a heap Θ:

$$\natural[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] \quad = \quad [x_1 \mapsto \partial(e_1), \ldots, x_n \mapsto \partial(e_n)]$$

In this definition, the function ∂ removes all observation labels from a program expression:



where :

$$A12 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), a \mapsto 2, \underline{c \mapsto 5}, d \mapsto hnf(b, c),$$
$$arg3 \mapsto 2, k \mapsto 2, t \mapsto Obs(add(arg)), arg \mapsto 1, arg2 \mapsto 1, b \mapsto 3, u \mapsto 3$$
$$A13 := x \mapsto 1, y \mapsto 2, z \mapsto Obs(add), inc \mapsto Obs(add(arg)), a \mapsto 2, c \mapsto 5,$$
$$\underline{d \mapsto 5}, arg3 \mapsto 2, k \mapsto 2, t \mapsto Obs(add(arg)), arg \mapsto 1, arg2 \mapsto 1,$$
$$b \mapsto 3, u \mapsto 3$$

Figure 6.10: The derivation tree of "*add*"

$$
\begin{array}{rcl}
\partial(f(\overline{x_n}) = e) & = & f(\overline{x_n}) = \partial(e) \\
\partial(O(e)) & = & \partial(e) \\
\partial(Obs(e)) & = & \partial(e) \\
\partial(x) & = & x \\
\partial(C(\overline{x_n})) & = & C(\overline{x_n}) \\
\partial(f(\overline{x_n})) & = & f(\overline{x_n}) \\
\partial(f^o(\overline{x_n})) & = & f(\overline{x_n}) \\
\partial(obs\_f(\overline{x_n})) & = & f(\overline{x_n}) \\
\partial(case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}) & = & case\ \partial(e)\ of\ \{\overline{p_n \rightarrow \partial(e_n)}\} \\
\partial(fcase\ e\ of\ \{\overline{p_n \rightarrow e_n}\}) & = & fcase\ \partial(e)\ of\ \{\overline{p_n \rightarrow \partial(e_n)}\} \\
\partial(e_1\ or\ e_2) & = & \partial(e_1)\ or\ \partial(e_2) \\
\partial(let\ \overline{x_n = e_n}\ in\ e) & = & let\ \overline{x_n = \partial(e_n)}\ in\ \partial(e) \\
\partial(apply(x, y)) & = & apply(x, y)
\end{array}
$$

We extend this definition also for a program (P):

$$
\begin{array}{rcl}
\partial(P) & = & \partial(\overline{D_n}) \\
\partial(\overline{D_n}) & = & \partial(\overline{D_1}), \partial(\overline{D_2}), ..., \partial(\overline{D_n}) \\
\partial(D) & = & \partial(f(\overline{x_n}) = e)
\end{array}
$$

Definition 2:

To prove the correctness of the rules defined for partial applications, we denote by $\neg(v)$ a removal of all labels from a value v:

$$
\begin{array}{rcll}
\neg(Obs(v)) & = & v & \\
\neg(x) & = & x & \\
\neg(C(\overline{x_n})) & = & C(\overline{x_n}) & \\
\neg(\varphi(\overline{x_k})) & = & \varphi(\overline{x_k}) & \text{where } k < n \text{ and } \varphi(\overline{y_n}) = e \in P^o
\end{array}
$$

Definition 3:

To prove the correctness, we denote by $\sqsubset$ a relation between two heaps (e.g., $\Gamma \sqsubset \Omega$).

$$
\begin{array}{rcll}
[] & \sqsubset & \Omega & (\text{if } \Omega \neq []) \\
[x_1 \mapsto e_1, ..., x_n \mapsto e_n] & = & [y_1 \mapsto e'_1, ..., y_m \mapsto e'_m] & (\text{if } \forall i, \exists j \text{ such that } x_i = y_j \text{ and } e_i = e'_j \\
 & & & \text{and if } \forall j, \exists i \text{ such that } x_i = y_j \text{ and } e_i = e'_j \\
 & & & \text{where } i \in \{1..n\}, j \in \{1..m\} \text{ and } n = m) \\
[x_1 \mapsto e_1, ..., x_n \mapsto e_n] & \sqsubset & [y_1 \mapsto e'_1, ..., y_m \mapsto e'_m] & (\text{if } \forall i, \exists j \text{ such that } x_i = y_j \text{ and } e_i = e'_j \\
 & & & \text{where } i \in \{1..n\}, j \in \{1..m\} \text{ and } n < m) \\
\Gamma & \sqsubseteq & \Omega & (\text{if } \Gamma{=}\Omega \text{ or } \Gamma \sqsubset \Omega)
\end{array}
$$

Theorem:

Let P *be a labeled program* ($P \in Prog^o$),
$\partial(P)$ *a program with no labeled expressions* ($\partial(P) \in Prog$),
e *a labeled expression* ($e \in Exp^o$),
$\partial(e)$ *an expression with no observations* ($\partial(e) \in Exp$),
v *a labeled value* ($v \in Value^o$) *and*
$\neg(v)$ *a value with no observations* ($\neg(v) \in Value$).

$\forall\Gamma, \forall\Omega, \forall\Delta, \forall\Theta$ *with* $\natural(\Gamma) \sqsubseteq \natural(\Omega)$ *holds*
if $\natural(\Gamma) : \partial(e) \Downarrow \natural(\Delta) : \neg(v)$
then $T, \Omega : e \Downarrow^o T', \Theta : v$ with $\natural(\Delta) \sqsubseteq \natural(\Theta)$
and if $T, \Omega : e \Downarrow^o T', \Theta : v$
then $\natural(\Gamma) : \partial(e) \Downarrow \natural(\Delta) : \neg(v)$ with $\natural(\Delta) \sqsubseteq \natural(\Theta)$

Proof: The proof is by induction on the structure of derivation trees.

- *Base Case*: Derivation trees (with or without observations) have only one node.
- *Inductive Steps*: Derivation trees (with or without observations) have more than one nodes.

Base Cases: We distinguish different base cases, depending on the form of the expression evaluated where $\forall\Gamma, \forall\Omega$ holds $\natural(\Gamma) \sqsubseteq \natural(\Omega)$.

- **(Constructor)**
  The rule Val yields:

$$\natural(\Gamma)[\overline{x_n \mapsto \partial(e_n)}] : C(\overline{x_n}) \;\Downarrow\; \natural(\Gamma)[\overline{x_n \mapsto \partial(e_n)}] : C(\overline{x_n})$$

  $OVal$ is used in the observation semantics:

$$T, \Omega[\overline{x_n \mapsto e_n}] : C(\overline{x_n}) \;\Downarrow^o\; T, \Omega[\overline{x_n \mapsto e_n}] : C(\overline{x_n})$$

  The result of evaluations in both rules are equal ($C(\overline{x_n})$).

  Removing labels from the heaps yields:

  $\natural(\Gamma) \cup [\overline{x_n \mapsto \partial(e_n)}] \sqsubseteq \natural(\Omega) \cup \natural[\overline{x_n \mapsto e_n}]$

$$= \natural(\Omega) \cup [\overline{x_n \mapsto \partial(e_n)}]$$

$\natural(\Gamma) \cup [\overline{x_n \mapsto \partial(e_n)}] \sqsubset \natural(\Omega) \cup [\overline{x_n \mapsto \partial(e_n)}] \quad (if\ \natural(\Gamma) \sqsubset \natural(\Omega))$

$\natural(\Gamma) \cup [\overline{x_n \mapsto \partial(e_n)}] = \natural(\Omega) \cup [\overline{x_n \mapsto \partial(e_n)}] \quad (if\ \natural(\Gamma) = \natural(\Omega))$

Therefore, $OVal$ holds for constructors.

- **(Variable bounded to constructor)**
  The rule $VarCons$ represents the following derivation tree:

  $$\natural(\Gamma)[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] : x \ \Downarrow\ \natural(\Gamma)[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] : C(\overline{y_n})$$

  $OVarCons$ yields:

  $$T, \Omega[x \mapsto Obs(C(\overline{y_n})), \overline{y_n \mapsto e_n}] : x \ \Downarrow^o\ T', \Omega[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto O(e_n)}] : C(\overline{y_n})$$

  The result of evaluations in both rules are equal ($C(\overline{y_n})$).

  Removing labels from heaps yields:

  $$\begin{aligned}\natural(\Gamma) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] &\sqsubseteq \natural(\Omega) \cup \natural[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto O(e_n)}] \\ &= \natural(\Omega) \cup [x \mapsto \partial(C(\overline{y_n})), \overline{y_n \mapsto \partial(O(e_n))}] \\ &= \natural(\Omega) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}]\end{aligned}$$

  $\natural(\Gamma) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] \sqsubset \natural(\Omega) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}]\ (if\ \natural(\Gamma) \sqsubset \natural(\Omega))$

  $\natural(\Gamma) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] = \natural(\Omega) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}]\ (if\ \natural(\Gamma) = \natural(\Omega))$

  Therefore, $OVarCons$ holds for constructors.

- **(Variable bounded to itself)**
  The rule Val yields:

  $$\natural(\Gamma)[y \mapsto y] : y \ \Downarrow\ \natural(\Gamma)[y \mapsto y] : y$$

  $OVal$ represents the following derivation tree:

  $$T, \Omega[y \mapsto Obs(y)] : y \ \Downarrow^o\ T', \Omega[y \mapsto O(y)] : y$$

  Both rules return the variable y unchanged.

  Removing labels from the heaps yields:

  $\natural(\Gamma) \cup [y \mapsto y] \sqsubseteq \natural(\Omega) \cup \natural[y \mapsto O(y)]$

$$= \natural(\Omega) \cup [y \mapsto \partial(O(y))]$$
$$= \natural(\Omega) \cup [y \mapsto \partial(y)]$$
$$= \natural(\Omega) \cup [y \mapsto y]$$

$\natural(\Gamma) \cup [y \mapsto y] \sqsubset \natural(\Omega) \cup [y \mapsto y] \qquad (if\ \natural(\Gamma) \sqsubset \natural(\Omega))$

$\natural(\Gamma) \cup [y \mapsto y] = \natural(\Omega) \cup [y \mapsto y] \qquad (if\ \natural(\Gamma) = \natural(\Omega))$

Therefore, $OVal$ holds for logical variables.

- **(Partial application)**
  The rule $PartVal$ offers:

  $\natural(\Gamma) : \varphi(\overline{x_k}) \ \Downarrow\ \natural(\Gamma) : \varphi(\overline{x_k}) \qquad$ where $\varphi(\overline{y_n}) = \partial(e) \in P$ and $k < n$

  $OPartVal$ is used in the observation semantics:

  $T, \Omega : Obs(\varphi(\overline{x_k})) \ \Downarrow^o\ T, \Omega : Obs(\varphi(\overline{x_k})) \qquad$ where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n$

  The equality of results can be proved by removing labels:

  $\varphi(\overline{x_k}) = \neg(Obs(\varphi(\overline{x_k})))$
  $\qquad = \varphi(\overline{x_k})$

  Therefore $OPartVal$ holds for partial applications.

- **(Variable bounded to partial application)**
  The rule $PartVarCons$ yields:

  $$\natural(\Gamma)[x \mapsto \varphi(\overline{x_k})] : x \ \Downarrow\ \natural(\Gamma)[x \mapsto \varphi(\overline{x_k})] : \varphi(\overline{x_k})$$

  where $\varphi(\overline{y_n}) = \partial(e) \in P$ and $k < n$

  $OPartVarCons$ is used in the observation semantics:

  $$T, \Omega[x \mapsto Obs(\varphi(\overline{x_k}))] : x \ \Downarrow\ T, \Omega[x \mapsto Obs(\varphi(\overline{x_k}))] : Obs(\varphi(\overline{x_k}))$$

  where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n$

  The equality of results can be proved by removing labels:

  $\varphi(\overline{x_k}) = \neg(Obs(\varphi(\overline{x_k})))$
  $\qquad = \varphi(\overline{x_k})$

  Removing labels, from the heaps of the roots, yields:

$$\begin{aligned}\natural(\Gamma) \cup [x \mapsto \varphi(\overline{x_k})] &\sqsubseteq \natural(\Omega) \cup \natural[x \mapsto Obs(\varphi(\overline{x_k}))] \\ &= \natural(\Omega) \cup [x \mapsto \partial(Obs(\varphi(\overline{x_k})))] \\ &= \natural(\Omega) \cup [x \mapsto \partial(\varphi(\overline{x_k}))] \\ &= \natural(\Omega) \cup [x \mapsto \varphi(\overline{x_k})]\end{aligned}$$

$$\natural(\Gamma) \cup [x \mapsto \varphi(\overline{x_k})] \sqsubset \natural(\Omega) \cup [x \mapsto \varphi(\overline{x_k})] \qquad (if\ \natural(\Gamma) \sqsubset \natural(\Omega))$$

$$\natural(\Gamma) \cup [x \mapsto \varphi(\overline{x_k})] = \natural(\Omega) \cup [x \mapsto \varphi(\overline{x_k})] \qquad (if\ \natural(\Gamma) = \natural(\Omega))$$

Therefore $OPartVarCons$ holds.

Inductive Steps: We distinguish different cases, depending on the form of the expression evaluated where $\forall\Gamma, \forall\Omega$ holds $\natural(\Gamma) \sqsubseteq \natural(\Omega)$.

- **(Variable bounded to expression)**
  The evaluation of an expression can yield two different values:
  - *Constructor*: The derivation tree of $VarExp$ is represented as follows:

$$\dfrac{\dfrac{subproofs1}{\natural(\Gamma)[x \mapsto \partial(e)] : \partial(e) \ \Downarrow\ \natural(\Delta)[x \mapsto \partial(e), \overline{y_n \mapsto \partial(e_n)}] : C(\overline{y_n})}}{\natural(\Gamma)[x \mapsto \partial(e)] : x \ \Downarrow\ \natural(\Delta)[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] : C(\overline{y_n})}$$

    In observation semantics, the rule $OVarExp$ is used:

$$\dfrac{\dfrac{subproofs2}{T, \Omega[x \mapsto e] : e \ \Downarrow^o\ T', \Theta[x \mapsto e, \overline{y_n \mapsto e_n}] : C(\overline{y_n})}}{T, \Omega[x \mapsto Obs(e)] : x \ \Downarrow^o\ T'', \Theta[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto O(e_n)}] : C(\overline{y_n})}$$

    We assume that our theorem holds for subproofs 1 and 2. Now by induction over the structure of the derivation tree, we know that our theorem also holds for the sub-trees

    $\natural(\Gamma)[x \mapsto \partial(e)] : \partial(e) \ \Downarrow\ \natural(\Delta)[x \mapsto \partial(e), \overline{y_n \mapsto \partial(e_n)}] : C(\overline{y_n})$

    and

    $T, \Omega[x \mapsto e] : e \ \Downarrow^o\ T', \Theta[x \mapsto e, \overline{y_n \mapsto e_n}] : C(\overline{y_n})$.

    Both rules return the constructor $C(\overline{y_n})$ as a result. Removing labels from the heaps of the roots yields:

    $\natural(\Delta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] \sqsubseteq \natural(\Theta) \cup \natural[x \mapsto C(\overline{y_n}), \overline{y_n \mapsto O(e_n)}]$

$$= \natural(\Theta) \cup [x \mapsto \partial(C(\overline{y_n})), \overline{y_n \mapsto \partial(O(e_n))}]$$
$$= \natural(\Theta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}]$$

$$\natural(\Delta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] \sqsubset \natural(\Theta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] \quad (if\ \natural(\Gamma) \sqsubset \natural(\Omega))$$
$$\natural(\Delta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] = \natural(\Theta) \cup [x \mapsto C(\overline{y_n}), \overline{y_n \mapsto \partial(e_n)}] \quad (if\ \natural(\Gamma) = \natural(\Omega))$$

Therefore, $OVarExp$ holds for expressions.

– *Logical variable*: The rule $VarExp$ offers:

$$\frac{\dfrac{subproofs1}{\natural(\Gamma)[x \mapsto \partial(e)] : \partial(e) \ \Downarrow\ \natural(\Delta)[x \mapsto \partial(e), y \mapsto y] : y}}{\natural(\Gamma)[x \mapsto \partial(e)] : x \ \Downarrow\ \natural(\Delta)[x \mapsto y, y \mapsto y] : y}$$

The rule $OVarExp$ represents the following derivation tree:

$$\frac{\dfrac{subproofs2}{T, \Omega[x \mapsto e] : e \ \Downarrow^o\ T', \Theta[x \mapsto e, y \mapsto y] : y}}{T, \Omega[x \mapsto Obs(e)] : x \ \Downarrow^o\ T'', \Theta[x \mapsto y, y \mapsto O(y)] : y}$$

We assume that our theorem holds for subproofs 1 and 2. Now by induction over the structure of the derivation tree, we know that our theorem also holds for the sub-trees

$\natural(\Gamma)[x \mapsto \partial(e)] : \partial(e) \ \Downarrow\ \natural(\Delta)[x \mapsto \partial(e), y \mapsto y] : y$

and

$T, \Omega[x \mapsto e] : e \ \Downarrow^o\ T', \Theta[x \mapsto e, y \mapsto y] : y.$

Both rules return the logical variable y as a result. Removing labels, from the heaps of the roots, yields:

$$\natural(\Delta) \cup [x \mapsto y, y \mapsto y] \sqsubseteq \natural(\Theta) \cup \natural[x \mapsto y, y \mapsto O(y)]$$
$$= \natural(\Theta) \cup [x \mapsto \partial(y), y \mapsto \partial(O(y))]$$
$$= \natural(\Theta) \cup [x \mapsto y, y \mapsto \partial(y)]$$
$$= \natural(\Theta) \cup [x \mapsto y, y \mapsto y]$$

$$\natural(\Delta) \cup [x \mapsto y, y \mapsto y] \sqsubset \natural(\Theta) \cup [x \mapsto y, y \mapsto y] \quad (if\ \natural(\Delta) \sqsubset \natural(\Theta))$$
$$\natural(\Delta) \cup [x \mapsto y, y \mapsto y] = \natural(\Theta) \cup [x \mapsto y, y \mapsto y] \quad (if\ \natural(\Delta) = \natural(\Theta))$$

Therefore, $OVarExp$ holds for logical variables.

- **(Function call)**
  In the rule Fun, each application of a function is reduced to its right-hand side where its arguments are substituted:

$$\frac{subproofs1}{\frac{\natural(\Gamma)[\overline{x_n \mapsto \partial(e_n)}] : \rho(\partial(e)) \;\Downarrow\; \natural(\Delta)[\overline{x_m \mapsto \partial(v_m)}] : \neg(v)}{\natural(\Gamma)[\overline{x_n \mapsto \partial(e_n)}] : f(\overline{x_n}) \;\Downarrow\; \natural(\Delta)[\overline{x_m \mapsto \partial(v_m)}] : \neg(v)}}$$

  where $m \leq n$, $\rho = \{\overline{y_n \mapsto x_n}\}$ and $f(\overline{y_n}) = \partial(e) \in P$

  Evaluating observed functions by $OFunCall$ offers:

$$\frac{subproofs2}{\frac{\frac{\frac{\frac{T'', \Omega[\overline{x_n \mapsto e_n}, \overline{arg_n \mapsto O(x_n)}, res \mapsto \rho'(e)] : \rho'(e) \;\Downarrow^o\; T''', \Theta[V] : v}{T'', \Omega[\overline{x_n \mapsto e_n}, \overline{arg_n \mapsto O(x_n)}, res \mapsto Obs(\rho'(e))] : res \;\Downarrow^o\; T'''', \Theta[V] : v}}{T', \Omega[\overline{x_n \mapsto e_n}, \overline{arg_n \mapsto O(x_n)}, res \mapsto O(\rho'(e))] : res \;\Downarrow^o\; T''''', \Theta[V] : v}}{T', \Omega[\overline{x_n \mapsto e_n}] : let\ \overline{arg_n = O(x_n)}, res = O(\rho'(e))\ in\ res \;\Downarrow^o\; T''''', \Theta[V] : v}}{T, \Omega[\overline{x_n \mapsto e_n}] : obs\_f(\overline{x_n}) \;\Downarrow^o\; T''''', \Theta[V] : v}$$

  where $\overline{arg_n}$ and res are fresh,

  $f(\overline{y_n}) = e \in P^o$,

  $\rho' = \{\overline{y_n \mapsto arg_n}\}$,

  $V := (\overline{x_m \mapsto v_m}, \overline{arg_m \mapsto v_m}, res \mapsto v)$ and

  $m \leq n$.

  The equality of results in sub-trees

  $\natural(\Gamma)[\overline{x_n \mapsto \partial(e_n)}] : \rho(\partial(e)) \;\Downarrow\; \natural(\Delta)[\overline{x_m \mapsto \partial(v_m)}] : \neg(v)$

  and

$$T'', \Omega[\overline{x_n \mapsto e_n}, \overline{arg_n \mapsto O(x_n)}, res \mapsto \rho'(e)] : \rho'(e) \Downarrow^o T''', \Theta[\overline{x_m \mapsto v_m}, \overline{arg_m \mapsto v_m}, res \mapsto v] : v$$

  can be proved as in correctness of $OVarExp$. Therefore, $OFunCall$ holds for function calls.

  The reason to label a function call is to observe evaluated arguments and the results of this function. Observing arguments in the observation semantics is performed by labeling arguments. This inserts extra bindings into the heap. Removing labels from the heaps yields:

  $\natural(\Delta) \cup [\overline{x_m \mapsto \partial(v_m)}] \sqsubseteq \natural(\Theta) \cup \natural[\overline{x_m \mapsto v_m}, \overline{arg_m \mapsto v_m}, res \mapsto v]$

$$= \natural(\Theta) \cup [\overline{x_m \mapsto \partial(v_m)}, \overline{arg_m \mapsto \partial(v_m)}, res \mapsto \partial(v)]$$

$$\natural(\Delta) \cup [\overline{x_m \mapsto \partial(v_m)}] \sqsubset \natural(\Theta) \cup [\overline{x_m \mapsto \partial(v_m)}, \overline{arg_m \mapsto \partial(v_m)}, res \mapsto \partial(v)] \quad (if\ \natural(\Delta) \sqsubseteq \natural(\Theta))$$

- **(let expression)**
  The rule *Let* offers the following derivation tree:

$$\frac{\dfrac{subproofs1}{\natural(\Gamma)[\overline{y_k \mapsto \rho(\partial(e_k))}] : \rho(\partial(e)) \ \Downarrow\ \natural(\Delta) : \neg(v)}}{\natural(\Gamma) : let\ \{\overline{x_n = \partial(e_k)}\}\ in\ \partial(e) \ \Downarrow\ \natural(\Delta) : \neg(v)}$$

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

  The rule *OLet* is used in the observation semantics:

$$\frac{\dfrac{subproofs2}{T, \Omega[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \ \Downarrow^o\ T', \Theta : v}}{T, \Omega : let\ \{\overline{x_n = e_k}\}\ in\ e \ \Downarrow^o\ T', \Theta : v}$$

  The equality of results can be proved as in correctness of $OVarExp$.

- **(or-expression)**
  The rule *Or* offers:

$$\frac{\dfrac{subproofs1}{\natural(\Gamma) : \partial(e_i) \ \Downarrow\ \natural(\Delta) : \neg(v)}}{\natural(\Gamma) : \partial(e_1)\ or\ \partial(e_2) \ \Downarrow\ \natural(\Delta) : \neg(v)}$$

  where $i \in \{1, 2\}$

  The rule *OOr* is used in the observation semantics:

$$\frac{\dfrac{subproofs2}{T, \Omega : e_i \ \Downarrow^o\ T', \Theta : v}}{T, \Omega : e_1\ or\ e_2 \ \Downarrow^o\ T', \Theta : v}$$

  The equality of results in the subproofs 1 and 2 can be proved as in correctness of $OVarExp$.

- **(fcase-expressions)**
  The rule *FCase* offers:

$$\frac{\dfrac{subproof1}{\natural(\Gamma) : \partial(e) \ \Downarrow\ \natural(\Delta)[x \mapsto x] : x} \quad \dfrac{subproof2}{\natural(\Delta)[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(\partial(e_i)) \ \Downarrow\ \natural(\Theta) : \neg(v)}}{\natural(\Gamma) : fcase\ \partial(e)\ of\ \{\overline{p_k \mapsto \partial(e_k)}\} \ \Downarrow\ \natural(\Theta) : \neg(v)}$$

where $\rho = \{\overline{x_n \mapsto y_n}\}$ and $p_i = C(\overline{x_n})$

The rule $OFCase$ offers:

$$\frac{\dfrac{subproof3}{T, \Omega : e \ \Downarrow^o \ T', \Psi[x \mapsto O(x)] : x} \quad \dfrac{subproof4}{T'', \Psi[x \mapsto \rho(p_i), \overline{y_n \mapsto y_n}] : \rho(e_i) \ \Downarrow^o \ T''', \Upsilon : v}}{T, \Omega : fcase \ e \ of \ \{\overline{p_k \mapsto e_k}\} \ \Downarrow^o \ T''', \Upsilon : v}$$

The equality of results in the subproofs 1 and 3 and also in the subproofs 2 and 4 can be proved as in correctness of $OVarExp$, meaning that $OFCase$ holds for fcase expressions.

- **(prim_apply)**
  The first rule of $PrimApply$ offers the following derivation tree:

$$\frac{\dfrac{subproofs1}{\natural(\Gamma)[z \mapsto prim\_apply(x,y)] : x \quad \Downarrow \quad \natural(\Delta) : \varphi(\overline{x_k})}}{\natural(\Gamma)[z \mapsto prim\_apply(x,y)] : z \quad \Downarrow \quad \natural(\Delta)[z \mapsto \varphi(\overline{x_k}, y)] : \varphi(\overline{x_k}, y)}$$

  where $\varphi(\overline{y_n}) = \partial(e) \in P$ with $k < n - 1$

  The first rule of $OPrimApply$ has the following derivation tree:

$$\frac{\dfrac{subproofs2}{T, \Omega[z \mapsto prim\_apply(x,y)] : x \ \Downarrow^o \ T', \Theta : \varphi(\overline{x_k})}}{T, \Omega[z \mapsto Obs(prim\_apply(x,y))] : z \quad \Downarrow^o \quad T', \Theta[z \mapsto Obs(\varphi(\overline{x_k}, y))] : Obs(\varphi(\overline{x_k}, y))}$$

  where $\varphi(\overline{y_n}) = e \in P^o$ and $k < n - 1$

  The results of the sub-trees

  $$\natural(\Gamma)[z \mapsto prim\_apply(x,y)] : x \quad \Downarrow \quad \natural(\Delta) : \varphi(\overline{x_k})$$

  and

  $$T, \Omega[z \mapsto prim\_apply(x,y)] : x \ \Downarrow^o \ T', \Theta : \varphi(\overline{x_k})$$

  are equal ($\varphi(\overline{x_k})$). The equality of the extended partial application, in the root of the derivation trees, can be proved using Definition 2:

  $$\varphi(\overline{x_k}, y) = \neg(Obs(\varphi(\overline{x_k}, y)))$$

  $$\varphi(\overline{x_k}, y) = \varphi(\overline{x_k}, y)$$

The second rule of $PrimApply$ offers

$$\frac{\dfrac{subproof1}{\natural(\Gamma)[z \mapsto M] : x \;\Downarrow\; \natural(\Delta) : \varphi(\overline{x_k})} \quad \dfrac{subproof2}{\natural(\Delta)[z \mapsto \varphi(\overline{x_k}, y)] : z \;\Downarrow\; \natural(\Theta) : \neg(v)}}{\natural(\Gamma)[z \mapsto M] : z \;\Downarrow\; \natural(\Theta)[z \mapsto \neg(v)] : \neg(v)}$$

where $M := prim\_apply(x, y)$

$\varphi(\overline{y_n}) = \partial(e) \in P$ with $k = n - 1$

The second rule of $OPrimApply$ has the following derivation tree:

$$\frac{\dfrac{subproof3}{T, \Omega[z \mapsto M] : x \;\Downarrow^o\; T', \Psi : \varphi(\overline{x_k})} \quad \dfrac{subproof4}{T', \Psi[z \mapsto Obs(\varphi(\overline{x_k}, y))] : z \;\Downarrow^o\; T'', \Upsilon : v}}{T, \Omega[z \mapsto Obs(prim\_apply(x, y))] : z \;\Downarrow^o\; T'', \Upsilon[z \mapsto v] : v}$$

where $M := prim\_apply(x, y)$

$\varphi(\overline{y_n}) = e \in P^o$ and $k = n - 1$

The subproofs 1 and 3 have equal results. The equality of the results in the subproofs 2 and 4 can be proved as in correctness of $OVarExp$. As a result, $OPrimApply$ holds for partial applications.

- **(apply)**
  The rule *Apply* offers the following derivation tree:

$$\frac{\dfrac{subproof1}{\natural(\Gamma) : let\ a = prim\_apply(x, y)\ in\ hnf(x, a) \;\Downarrow\; \natural(\Delta) : \neg(v)}}{\natural(\Gamma) : apply(x, y) \;\Downarrow\; \natural(\Delta) : \neg(v)}$$

  $OApply$ has the following derivation tree:

$$\frac{\dfrac{subproof2}{T', \Omega[z \mapsto apply(x, y)] : let\ a = Obs(prim\_apply(x, y))\ in\ hnf(x, a) \;\Downarrow^o\; T'', \Theta : v}}{T, \Omega[z \mapsto O(apply(x, y))] : z \;\Downarrow^o\; T'', \Theta[z \mapsto v] : v}$$

  The equality of results in the sub-trees

  $\natural(\Gamma) : let\ a = prim\_apply(x, y)\ in\ hnf(x, a) \;\Downarrow\; \natural(\Delta) : \neg(v)$

  and

  $T', \Omega[z \mapsto apply(x, y)] : let\ a = Obs(prim\_apply(x, y))\ in\ hnf(x, a) \Downarrow^o T'', \Theta : v$

  can be proved as in correctness of $OLet$, meaning that $OApply$ holds for partial applications.

Chapter 7

Program Coverage

"Even after all this time,
The sun never says to the earth, 'You owe Me.'
Look, What happens With a love like that,
It lights the whole sky."

Hafez

Testers specify whether the behavior of programs are acceptable. Tests can be performed automatically or by programmers. In both cases, it is possible that given inputs to programs will never generate unexpected outputs, although programs experience failures. Furthermore, programmers prefer to follow the order of a program execution in some cases, instead of observing the functions to see the program behavior during an execution. Knowing which parts of a program have been executed is useful for programmers, because this restricts the possible locations of a bug to executed parts.

As discussed in Section 3.5.2, we can extract parts of a program that are affected by executing a special expression using the program slicing (dynamic-forward slicing). We use this definition and allow the programmer to select arbitrary expressions. During the execution of a program, expressions are bound to values and reflect the execution of other parts of the program. For example, calling the function `reverse` on an empty list causes the execution of the first rule of this function. We would like to represent this information as follows:

```
reverse [] = []
reverse (x:xs) = (x:xs)
```

However, applying `reverse` to a non-empty list executes both rules of the function `reverse`. The idea is to highlight both rules that are executed:

```
reverse [] = []
reverse (x:xs) = (x:xs)
```

We suggest highlighting executed expressions by two methods:

- Highlighting of function rules in source code.
- Highlighting of (sub-)expressions in a tree representation of programs.

7.1 *Tree Representation of Programs*

To analyze a program, so that a programmer has access to all expressions of this program, we suggest a tree-representation method of the source code. For this purpose, we use the meta-programming library of Curry and represent all defined functions and data structures in a tree. By means of the Curry GUI library, we also provide convenient access to this tree in the browser of the tool.

Programmers can define different functions and data types in programs. In this thesis, we would like to represent a method to locate bugs in Curry programs. Because bugs can exist in the right-hand side of functions, the user defined functions are represented in the tree. This means that the first level of the tree belongs to global functions (in the representation below, the sign "+" means that the tree in this node is extendable and "-" means that the sub-tree for this node is already extended):

```
- CurryProgram
    + Function1
    + Function2
      ...
```

Function definitions have rules and right-hand sides of rules have expressions that should be represented to the user.

```
- CurryProgram
    - Function1
          - Rule1
               + RightHandSide
          + Rule2
            ...
    + Function2
      ...
```

To better understand this definition, let us consider the following simple program that offers the reverse representation of a given list:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

The function `reverse` is defined by two rules that are represented in the tree as `reverse(1)` and `reverse(2)` to the user:

```
- CurryProgram
    + reverse(1)
    + reverse(2)
```

All expressions within each of these rules must be represented in the tree. The only expression in the right-hand side of the first rule is the empty list. In other words, only the expression `[]` is represented to the user:

```
- CurryProgram
    - reverse(1)
        - []
    + reverse(2)
```

The second rule with the right-hand side `reverse xs ++ [x]` contains three function calls: (++), reverse and (:). The right-hand side of each function takes two expressions as parameters. In addition, each function call and all partial applications are represented:

```
- CurryProgram
    - reverse(1)
        - []
    - reverse(2)
        - reverse xs ++ [x]
            + (++) (reverse xs)
            + [x]
```

The definition above is also used for all sub-expressions that extend the tree with more sub-trees:

```
- CurryProgram
    - reverse(1)
        - []
    - reverse(2)
```

```
- reverse xs ++ [x]
    - (++) (reverse xs)
        (++)
      - reverse xs
          reverse
          xs
    - [x]
       - (x:)
           (:)
           x
         []
```

7.2 *Highlighting*

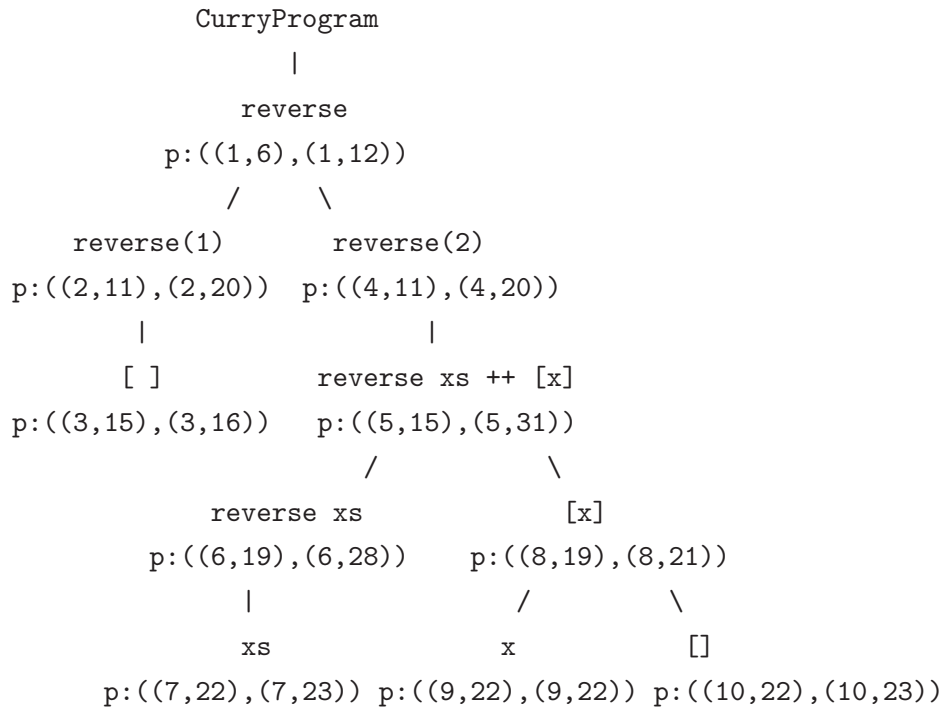
To highlight program expressions, we annotate expressions with the identity function `markLine`:

```
markLine :: ModuleName -> Position -> a -> a

type ModuleName = String
type Position   = (RC,RC)
type RC         = (Row,Col)
type Row        = Int
type Col        = Int
```

This function is applied to the position (second argument) of the actual expression (third argument) in a source code or in a tree representation of the program. The first argument in this definition is the name of the module that is defined as the root of the tree. `Positions` are recognized by the position of the first and last character of expressions (in a source code or in a tree representation of the program). To find the position of expressions we use the abstract syntax tree of programs that is generated by the Curry meta-programming library. Expressions of every sub-tree are indexed from top to bottom and in each tree level from left to right, while the representation of partial applications are ignored (partial applications are represented only in the browser of the tool (see Section 8.1.1)). As we have seen in the last section, each expression of the tree is represented by separate lines. As a result, the column of positions in the tree is considered to be zero.

Let us show positions of expressions in the tree representation of `reverse` without considering partial applications. The right-hand side of rules and rule sub-expressions have positions that should be recognized:



Expressions can be annotated with the identity function `markLine`. Each call to annotated expressions causes expressions to be marked by a Tcl/Tk command [51, 36] that is implemented in the tool's GUI library of Curry.

For example, if the expressions `reverse xs` and `x` from the second rule are selected to be highlighted, marker functions are added to the program as follows:

```
reverse (x:xs) = markLine "CurryProg" ((6,19),(6,28)) (reverse xs) ++
                 [markLine "CurryProg" ((9,22),(9,22)) x]
```

After execution of the annotated program (i.e., by `reverse [1,2]`), the highlighted tree is represented as follows:

```
- CurryProgram
   - reverse
        - reverse(1)
            - []
        - reverse(2)
            - reverse xs ++ [x]
                - reverse xs
                    - xs
```

```
- [x]
  - x
  - []
```

It is also possible to see executed parts of a program in original source code. This is useful when the programmer wishes only to know which function rules are executable. Note that, in this case, positions of rules are dependent on the positions of first characters of these rules in original source code. For example, the position of the first rule of the reverse function in the following program is (2,9):

```
main = reverse [1..10]
  where reverse []     = []
        reverse (x:xs) = [x] ++ reverse xs
```

Highlighted expressions show which parts of programs are executed. Non-executed parts of programs belong to dead-code that can be removed from source code, or are executable and should be tested by other applications. It is also possible that, because of the lazy behavior of programs, expressions are not executed. Our debugging system provides such a feature that can be useful for testing small separate functions of a program. It helps the programmer to concentrate on a small environment of the program to debug.

7.3 *Highlighting Semantics*

With respect to the accuracy of the observation semantics (Section 6.2), we extend it for our implemented highlighter. For this purpose, we have two stages:

- *Labeling normalized programs*: This gives labels to arbitrary units selected by the programmer (a unit is a function with all its right-hand side expressions).
- *Collecting slices*: This generates a tree of slices for recognition of executed expressions of selected units.

7.3.1 *Labeling Normalized Programs*

Labeling programs require two steps:

1. Programs are transformed into flat form, whereas all arguments of functions and constructors are variables (see Section 6.1.1).

P^m	::=	$\overline{D_k^m}$	(labeled program)
D^m	::=	$f^m\ (\overline{x_n}) = e^m$	(labeled unit)
	|	$f(\overline{x_n}) = e$	(function definition)
e^m	::=	$M(x)$	(labeled variable)
	|	$M(C(\overline{x_n}))$	(labeled constructor call)
	|	$M(f(\overline{x_n}))$	(labeled function call)
	|	$M(case\ e^m\ of\ \{\overline{p_n \rightarrow e_n^m}\})$	(labeled rigid case)
	|	$M(fcase\ e^m\ of\ \{\overline{p_n \rightarrow e_n^m}\})$	(labeled flexible case)
	|	$M(e_1^m\ or\ e_2^m)$	(labeled disjunction)
	|	$M(let\ \overline{x_n = e_n^m}\ in\ e^m)$	(labeled let binding)
e	::=	x	(variable)
	|	$C(\overline{x_n})$	(constructor call)
	|	$f(\overline{x_n})$	(function call)
	|	$case\ e\ of\ \{\overline{p_n \rightarrow e_n}\}$	(rigid case)
	|	$fcase\ e\ of\ \{\overline{p_n \rightarrow e_n}\}$	(flexible case)
	|	$e_1\ or\ e_2$	(disjunction)
	|	$let\ \overline{x_n = e_n}\ in\ e$	(let binding)
	|	$prim\_g\ (e_1, \ldots, e_n)$	(primitive functions)
	|	$hnf(x_1, x_2)$	(head normal form evaluator)
p	::=	$C(\overline{x_n})$	

Figure 7.1: Marked flat program

2. As discussed in Section 7.2, it is possible to highlight expressions by annotating expressions with marker functions. Adding marker functions to program expressions is performed automatically by the debugging system. In the labeling process, we give only the label "M" to all expressions of units. These Units are selected functions with all its right-hand side expressions. The units of un-selected functions are not marked. Labeling expressions involves highlighting these expressions only if their evaluations are needed. Every labeled function is also represented by $f^m(\overline{x_n})$. Note that the labeled function $f^m(\overline{x_n})$ replaces the original one $f(\overline{x_n})$ in the program.

The syntax of labeled programs is presented in Figure 7.1. It indicates that a labeled program (P^m) contains labeled and un-labeled function definitions. If a unit that is related to a function definition is selected for highlighting, the function name and all of its right-hand side expressions are labeled ($f^m(\overline{x_n}) = e^m$). We define the set of labeled programs as $Prog^m$ and the set of expressions as Exp^m, meaning that $P^m \in Prog^m$ and $e, e^m \in Exp^m$.

As an example, let us select the unit related to the function f in the following Curry program:

$f\ []\ =\ []$

$f\ (x : xs)\ =\ xs$

After transforming the program and labeling the unit (in a tree representation mode), we obtain:

$$f^m(xs) = M(case\ M(xs)\ of \\ \{[] \rightarrow M([]); \\ (y : ys) \rightarrow M(ys)\})$$

Note that, by activating the highlighter in original source code, the debugging system adds only one marker function to the right-hand side of each function rule ($e^m := M(e)$). In this case, only the outermost expressions are observed:

$$f^m = M(case\ xs\ of \\ \{[] \rightarrow []; \\ (y : ys) \rightarrow ys\})$$

7.3.2 *Collecting Slices*

At this stage, it is assumed that a program has already been transformed and can be used to represent the dynamic semantics of the highlighter. Whenever labeled expressions are executed, their related slices are recorded in a tree structure (named *slices tree*):

$S, S' \in SlicesTree$

Information recorded in slices trees is extracted and represented to the user by highlighting slices in a viewer. Slices in this semantics are represented as:

$$\begin{array}{rcl} SlicesTree & := & Slice(position, counter, [SlicesTree]) \\ & | & Empty \end{array}$$

position is the position of a labeled executed expression. *counter* is a number that shows how many times the related expression is executed.

Let us discuss the operational semantics of the highlighter with regard to the defined slices trees. In order to represent the semantics of the highlighter, we need four components:

- a *labeled program* P^m ($\in Prog^m$) that contains labeled units.
- a *slices tree* ($\in SlicesTree$) to collect generated slices.
- a *heap* ($\in Heap$) to illustrate the lazy semantics and sharing, which can also contain bindings for labeled expressions.

- a marked or an unmarked *expression* ($\in Exp^m$) that is being evaluated.

The derivation rules of the highlighting semantics are represented in Table 7.1. A heap is a binding from variables to expressions. Expressions are labeled or unlabeled. A value is a constructor-rooted term or a logical variable:

$$\begin{array}{lcl} P^m & \in & Prog^m \\ \Gamma, \Delta & \in & Heap = Var \mapsto Exp^m \\ e, e^m & \in & Exp^m \\ v & \in & Value ::= x \mid C(\overline{x_n}) \end{array}$$

In the highlighting semantics, the position of an executed expression e is considered to be e^p. An empty tree is represented by $\{\}$. An updated or new inserted slice with the position e^p in the slices tree *tree* is represented as $ins_{e^p}(tree)$. The function *ins* inserts a new extracted slice into the slices tree while the order of positions is sorted:

$$\begin{array}{l} ins_{e^p}\ (\{\}) = Slice(e^p, 1, [Empty, Empty]) \\ ins_{e^p}\ (Slice(e_1^p, m, [t_1, t_2])) = \\ \quad if\ \ e^p < e_1^p\ \ then \\ \quad\quad if\ \ t_2 = Empty\ \ and\ \ t_1 = Slice(e_2^p, k, ts) \\ \quad\quad\quad then \\ \quad\quad\quad\quad Slice(e_2^p, k, [Slice(e^p, 1, [Empty, Empty]), \\ \quad\quad\quad\quad\quad\quad Slice(e_1^p, m, [Empty, t_2])]) \\ \quad\quad\quad else \\ \quad\quad\quad\quad Slice(e_1^p, 1, [ins_{e^p}\ (t_1), t_2]) \\ \quad if\ \ e^p > e_1^p\ \ then \\ \quad\quad if\ \ t_1 = Empty\ \ and\ \ t_2 = Slice(e_2^p, k, ts) \\ \quad\quad\quad then \\ \quad\quad\quad\quad Slice(e_2^p, k, [Slice(e_1^p, m, [t_1, Empty]), \\ \quad\quad\quad\quad\quad\quad Slice(e^p, 1, [Empty, Empty])]) \\ \quad\quad\quad else \\ \quad\quad\quad\quad Slice(e_1^p, 1, [t_1, ins_{e^p}(t_2)]) \\ \quad if\ e^p = e_1^p\ \ then \\ \quad\quad\quad\quad Slice(e_1^p, m + 1, [t_1, t_2]) \end{array}$$

If an expression is executed for the first time, the counter is 1 and, if an expression is already executed and has a slice in the tree, the counter of the existing slice is increased.

Let us briefly explain the derivation rules defined in Table 7.1 where:

Table 7.1: Derivation Rules for Highlighter

$MVarCons:$ $S, \Gamma[\overline{y_n \mapsto e_n}, x \mapsto M(C(\overline{y_n}))] : x \quad \Downarrow \quad S1, \Gamma[\overline{y_n \mapsto mark(e_n)}, x \mapsto C(\overline{y_n})] : C(\overline{y_n})$ $where \quad \text{S1} := ins_{(C(\overline{y_n}))^p} S$
$MVarExp:$ $\quad \dfrac{ins_{e^p} S, \Gamma[x \mapsto e] : e \quad \Downarrow \quad S', \Delta : v}{S, \Gamma[x \mapsto M(e)] : x \quad \Downarrow \quad S', \Delta[x \mapsto v] : v}$
$MVal:$ $S, \Gamma[x \mapsto M(x)] : x \quad \Downarrow \quad ins_{x^p} S, \Gamma[x \mapsto x] : x$ $S, \Gamma : C(\overline{x_n}) \quad \Downarrow \quad S, \Gamma : C(\overline{x_n})$
$MFunCall:$ $\dfrac{ins_{(f(\overline{x_n}))^p} S, \Gamma[\overline{x_n \mapsto mark(e_n)}] : \rho(e) \quad \Downarrow \quad S', \Delta : v}{S, \Gamma[\overline{x_n \mapsto e_n}] : M(f(\overline{x_n})) \quad \Downarrow \quad S', \Delta : v}$ $where \quad f(\overline{y_n}) = e \in P^m \; and \; \rho = \{\overline{y_n \mapsto x_n}\}$ $\dfrac{ins_{(f(\overline{x_n}))^p} S, \Gamma[\overline{x_n \mapsto mark(e_n)}] : \rho(e^m) \quad \Downarrow \quad S', \Delta : v}{S, \Gamma[\overline{x_n \mapsto e_n}] : M(f(\overline{x_n})) \quad \Downarrow \quad S', \Delta : v}$ $where \quad f^m(\overline{y_n}) = e^m \in P^m \; and \; \rho = \{\overline{y_n \mapsto x_n}\}$
$MFunDef:$ $\quad \dfrac{ins_{f^p} S, \Gamma[\overline{x_n \mapsto e_n}] : \rho(e^m) \quad \Downarrow \quad S', \Delta : v}{S, \Gamma[\overline{x_n \mapsto e_n}] : f(\overline{x_n}) \quad \Downarrow \quad S', \Delta : v}$ $where \quad f^m(\overline{y_n}) = e^m \; \in \; P^m \; and \; \rho = \{\overline{y_n \mapsto x_n}\}$

$$
\begin{array}{rcl}
mark(x) & = & M(x) \\
mark(C(\overline{x_n})) & = & M(C(\overline{x_n})) \\
mark(f(\overline{x_n})) & = & M(f(\overline{x_n})) \\
mark(case\ e\ of\ \{\overline{p_n \to e_n}\}) & = & M(case\ mark(e)\ of\ \{\overline{p_n \to mark(e_n)}\}) \\
mark(fcase\ e\ of\ \{\overline{p_n \to e_n}\}) & = & M(fcase\ mark(e)\ of\ \{\overline{p_n \to mark(e_n)}\}) \\
mark(e_1\ or\ e_2) & = & M(mark(e_1)\ or\ mark(e_2)) \\
mark(let\ \overline{x_n = e_n}\ in\ e) & = & M(let\ \overline{x_n = mark(e_n)}\ in\ mark(e))
\end{array}
$$

$MVarCons$: In order to evaluate a variable (x) that is bound to a labeled constructor-rooted term ($x \mapsto M(C(\overline{y_n}))$), we reduce the variable to this term and update the slices tree S for the position of this constructor. Note that all variable arguments of the constructor are labeled ($\overline{y_n \mapsto mark(e_n)}$). These variables are highlighted for possible execution.

$MVarExp$: This rule is used to highlight a variable that is evaluated to a constructor-rooted term or a logical variable. The variable x can be evaluated to a value (v) if there is a binding for this variable to a labeled expression ($x \mapsto M(e)$). The rule evaluates this expression (e) to a value (v) and returns it as a result for x, while the slices tree is updated.

$MVal$: This rule is used when a logical variable is labeled. The rule returns the variable and updates the slices tree. Note that, if a constructor-rooted term is being evaluated, the slices tree is returned without any modification.

$MFunCall$: This rule is used, if a labeled function application ($M(f(\overline{x_n}))$) is being evaluated. $MFunCall$ updates the slices tree and gives labels to all arguments of the function ($\overline{x_n \mapsto mark(e_n)}$). By this method, we guarantee that evaluated arguments are highlighted.

$MFunDef$: If there is a labeled function definition ($f^m(\overline{y_n}) = e^m$) in the program and an application of this function is evaluated ($f(\overline{x_n})$), this rule is used. The rule updates the slices tree for the position of the function definition and evaluates the function application to a value (by the rule Fun as defined in Table 6.1).

The definitions and Rules should become evident by the following example written in Curry:

$main\ =\ head\ [1,2]$

$head\ (x : xs)\ =\ x$

The function *head* in this definition returns the first element of a given list as the result. Suppose that we select the unit that is related to the function *main* to see the executed parts of this unit. To represent the highlighting semantics, we must transform the program as follows:

$$
\begin{array}{l}
\lceil\ \{\}, [] : main \quad -- \text{ using MFunDef} \\
\quad \lceil\ \{Sl(main^p, 1, [])\}, [] : let\ A\ in\ M(head(l)) \quad -- \text{ using Let} \\
\quad\quad \lceil\ \{Sl(main^p, 1, [])\}, [B] : M(head(l)) \quad -- \text{ using MFunCall} \\
\quad\quad\quad \lceil\ \{Sl(main^p, 1, [Sl((head(l))^p, 1, [])])\}, \\
\quad\quad\quad\quad [C] : case\ l\ of\ \{(y : ys) \rightarrow y\} \quad -- \text{ using Case} \\
\quad\quad\quad\quad \lceil\ \{Sl(main^p, 1, [Sl((head\ (l))^p, 1, [])])\}, \\
\quad\quad\quad\quad\quad [C] : l \quad -- \text{ using MVarCons} \\
\quad\quad\quad\quad\ \{Sl((head\ (l))^p, 1, [Sl(main^p, 1, []), Sl(((:)(x, r))^p, 1, [])])\}, \\
\quad\quad\quad\quad\quad [D] : (:)(x, r) \\
\quad\quad\quad\quad \lceil\ \{Sl((head\ (l))^p, 1, [Sl(main^p, 1, []), Sl(((:)(x, r))^p, 1, [])])\}, \\
\quad\quad\quad\quad\quad [D] : x \quad -- \text{ using MVarCons} \\
\quad\quad\quad\quad\ \{Sl((head\ (l))^p, 1, [Sl(main^p, 1, []), \\
\quad\quad\quad\quad\quad\quad Sl(((:)(x, r))^p, 1, [Sl(1^p, 1, [])])])\}, [E] : 1 \\
\quad\quad\quad \lfloor\ \{Sl((head\ (l))^p, 1, [Sl(main^p, 1, []), \\
\quad\quad\quad\quad\quad Sl(((:)(x, r))^p, 1, [Sl(1^p, 1, [])])])\}, [E] : 1 \\
\quad\quad \lfloor\ \{Sl((head\ (l))^p, 1, [Sl(main^p, 1, []), \\
\quad\quad\quad\quad Sl((:)(x, r))^p, 1, [Sl(1^p, 1, [])])])\}, [E] : 1 \\
\quad \lfloor\ \{Sl((head(l))^p, 1, [Sl(main^p, 1, []), \\
\quad\quad\quad Sl(((:)(x, r))^p, 1, [Sl(1^p, 1, [])])])\}, [E] : 1 \\
\lfloor\ \{Sl((head(l))^p, 1, [Sl(main^p, 1, []), \\
\quad\quad Sl(((:)(x, r))^p, 1, [Sl(1^p, 1, [])])])\}, [E] : 1
\end{array}
$$

where

$A := x = 1, y = 2, z = 3, r = (:)(y, z), l = (:)(x, r)$

$B := x \mapsto 1, y \mapsto 2, z \mapsto 3, r \mapsto (:)(y, z), l \mapsto (:)(x, r)$

$C := x \mapsto 1, y \mapsto 2, z \mapsto 3, r \mapsto (:)(y, z), \underline{l \mapsto M((:)(x, r))}$

$D := \underline{x \mapsto M(1)}, y \mapsto 2, z \mapsto 3, \underline{r \mapsto M((:)(y, z))}, \underline{l \mapsto (:)(x, r)}$

$E := \underline{x \mapsto 1}, y \mapsto 2, z \mapsto 3, r \mapsto \underline{M((:)(y, z))}, l \mapsto \underline{(:)(x, r)}$

Figure 7.2: Highlighting semantics of the function *head*

$$
\begin{aligned}
main^m = let\ x &= 1,\\
y &= 2,\\
z &= [],\\
r &= (:)(y, z),\\
l &= (:)(x, r)\ \ in\\
&M(head(l))
\end{aligned}
$$

$$head\ (xs) = case\ xs\ of\ \{\ (y : ys) \rightarrow y\ \}$$

Note that the function name ($main$) and also the right-hand side of this function ($head(l)$) are labeled. To start the evaluation, we need the labeled program P^m as a global parameter, an empty slices tree ($\{\}$), an empty list for the heap and an expression to evaluate:

$$\{\}, [] : main$$

Now we can follow the derivation by the defined rules of Table 7.1. For illustration purposes, we represent every slice *Slice* as *Sl*. The derivation tree is presented in Figure 7.2. Consider the slices tree in this figure. There are no slices for the remaining elements (r) of the list $[1, 2]$. This means that only the first element of this list has been executed. The debugging system represents this information by highlighting related positions in a viewer:

```
- main
   - head [1,2]
      - [1,2]
         1
         - [2]
            2
            []
```

The highlighting semantics indicates that the evaluation order of expressions is the same with operational semantics of [2]. We only generate a slice by starting an evaluation for each labeled expression.

7.4 *Implementing Highlighters*

In functional programs, functions can be executed recursively so that parts of a program can be executed more than once. Knowing the number of function calls can be useful in locating a position of non-terminating computations. In order to provide this information, besides highlighting expressions, we record the position and counter of executed expressions in a global state that we define as follows:

```
SlicesTree :: Global [(ModuleName,ExecSlices)]
SlicesTree = global [] Temporary
data ExecSlices = Slice Position Counter [ExecSlices]
                | Empty
type ModuleName = String
type Position   = (RC,RC)
type RC         = (Row,Col)
type Row        = Int
type Col        = Int
type Counter    = Int
```

`SlicesTree` is a kind of global state for storage of information about executed expressions (`Slice`). We record slices in a sorted tree (`ExecSlices`) by sorting positions. To represent the number of function calls, we also record a counter for each position. Counters are increased when the position of the next executed expression already exists in `ExecSlices`. In other words, each position has no more than one `Slice` in `ExecSlices`.

Let us consider the reverse program as an example and apply the function `reverse` to a list with 10 elements: `reverse [1..10]`. The first executed expression in the program is the expression `reverse`. This expression has the position ((1,6),(1,12)) in the tree representation. This means that the first slice is recorded as `Slice ((1,6),(1,12)) 1`. The next three executed expressions are `(reverse xs) ++ [x]`, `reverse xs` and `xs` that are inserted as slices to the tree:

```
                  Slice ((5,15),(5,31)) 1
                        /      \
  Slice ((1,6),(1,12)) 1        Slice ((6,19),(6,28)) 1
                                      \
                                      Slice ((7,22),(7,23)) 1
```

When the execution has been completed, counters of re-executed slices have been increased and we obtain all recorded slices as follows:

```
                     Slice ((5,15),(5,31)) 10
                          /        \
       Slice ((1,11),(1,17)) 11   Slice ((7,22),(7,23)) 10
              \                     /                \
Slice ((3,15),(3,16)) 1  Slice ((6,19),(6,28)) 10  Slice ((9,22),(9,22)) 10
                                                   /   \
                         Slice ((8,19),(8,21)) 10  Slice ((10,22),(10,23)) 10
```

The recorded slices represent the program executed expressions. They are represented to the user by highlighting the related positions in a tree representation of the program. The number of function calls is also represented to the user. In our example, the position of the expression `reverse` is ((1,6),(1,12)) and is called 11 times during the execution (Slice ((1,6),(1,12)) 11), ten times for the ten elements of the given list and once for the empty list (`(1:(2:(3...(10:[])))))`).

To highlight the slices collected, we use the following configuration item (see Section 2.6) defined in the GUI library of our implemented debugger:

```
data ConfItem = ... | TextMark [Position] Color [Position] Color
type Position = (RC,RC)
type RC       = (Row,Col)
type Row      = Int
type Col      = Int
type Color    = String
```

For example, by converting the configuration of a viewer to

```
TextMark [((10,22),(10,23))] "Red"
         [((1,6),(1,12))..((9,22),(9,22))] "Green"
```

we give a red color to the position of the currently executed expression and a green color to the other executed expressions.

Note that representation of executed rules in the original source code is implemented in the same manner.

Chapter 8

A Tool for Observing Program Behavior

> "An idea that is developed and put into action is more important than an idea that exists only as an idea."
>
> Buddha

Writing programs in lazy functional languages allows programmers to ignore thinking about the order of evaluations. In other words, programmers need only to concentrate on the logic of program's structure and not on detailed concepts of the lazy evaluation strategy. However, this advantage makes it difficult to locate bugs when programs yield unexpected outputs. Catching unexpected outputs in programs is one of the most critical requirements of a debugger.

Other bugs could be detected during run time, such as division by zero or non-terminating computations because of recursive calls to functions. It could also happen because of pattern-matching failures that interrupt the execution of programs and offers `"No more solutions"` as a result.

Sometimes, after run-time failures occur, locating bugs may be too difficult. Having a tester detect which parts of a program are executed and which expression is the last executed expression could be helpful. By detecting executed parts, programmers have a view of a surrounding area that might contain failures. In order to understand a relationship between executed expressions, it is desirable to also have tracer support to follow the evaluation order of expressions.

In this chapter, we discuss how the architecture of the tracer and tester in the implemented tool iCODE is provided. iCODE is a small portable tool for Curry programs. All

parts of the tool are written in Curry. In iCODE, the user has different views of her/his program behavior. The tool not only shows the behavior of each program expression in separate viewers, but also represents the evaluation order of right-hand side expressions in the original source code or in a tree representation of programs. By representing executed parts of source code and also the number of function calls, the tool is useful for finding a surrounding area that may contain program failures.

8.1 *Implementation and Design*

In this section, we briefly discuss how different parts of iCODE are implemented and designed.

8.1.1 *Browser*

We provide users with the means to select and also de-select arbitrary expressions from a program in a browser (see Figure 8.1). Programs are analyzed and all of their expressions are represented in the browser. The representation of expressions is considered in a tree format (the tree representation of programs is discussed in Section 7.1).

In the tree representation, program expressions are considered nodes. Nodes in trees have paths. Paths lead us to find positions of expressions in a program to which we can add observer functions.

For instance, we consider the reverse example to find the path of the expression `xs` in the right-hand side of the second rule:

```
reverse :: [Int] -> [Int]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

The function `reverse` has three functions in the right-hand side of its second rule: (++), reverse and (:). The outermost function is `(++)`. It takes two expressions `(++) (reverse xs)` and `[x]` as parameters. In the tree representation discussed in Section 7.1, we have seen that these parameters are represented as two sub-trees and that our selected expression is located in the left (first) sub-tree. Sub-trees are implemented as the data type `Node`:

```
data Node a = GlobalFunction a
            | Apply (a,Branch)
            | Var (a,Branch)
type Branch = Int
```

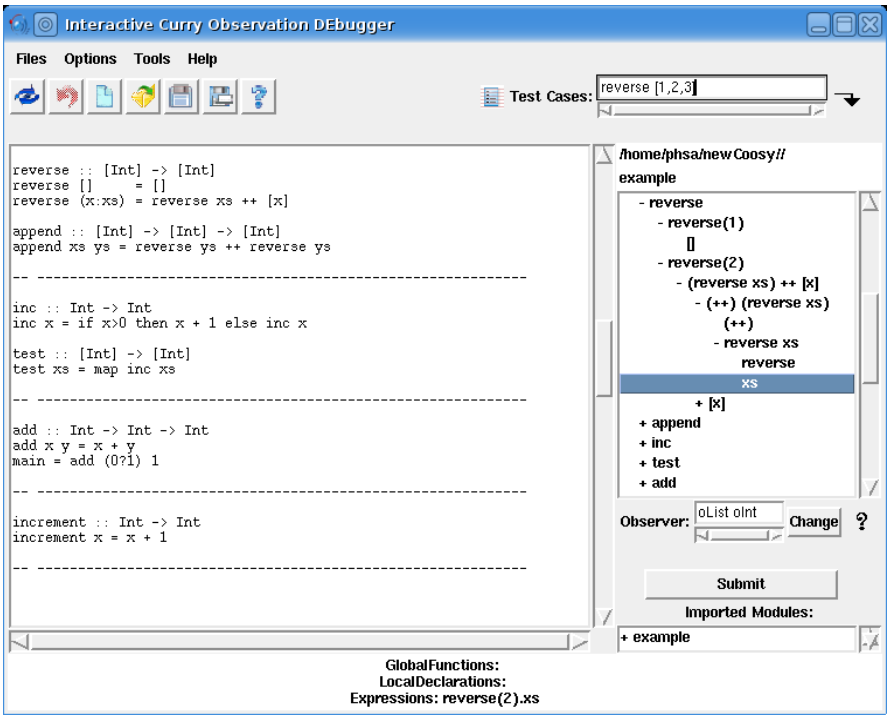


Figure 8.1: The browser

`Branches` in this data type are defined as indices for sub-trees which are labeled from left to right for children at every tree level. `Apply` is defined as an application and the constructor `Var` is considered for variables. Lists of nodes offer paths for program expressions. The path of our selected expression `xs` is represented as follows:

```
[GlobalFunction "reverse(2)",
 Apply ("reverse xs ++ [x]", 1),
 Apply ("(++) (reverse xs)", 1),
 Apply ("reverse xs", 2),
 Var   ("xs", 2)]
```

The list above shows that the selected expression belongs to the second rule of the function `reverse`:

```
GlobalFunction "reverse(2)"
```

and to the first sub-tree of this function that is an application:

```
Apply ("reverse xs ++ [x]", 1).
```

The first sub-tree of this application is:

```
Apply ("(++) (reverse xs)", 1)
```

and in its second sub-tree we have another application:

```
Apply ("reverse xs", 2).
```

The second sub-tree of this node contains a variable that is our selected expression:

```
Var ("xs", 2).
```

On the other hand, in the background of the tree represented to the user, we have a list of program nodes previously mentioned:

```
- CurryProgram
    - GlobalFunction "reverse(1)"
        - Var ("[]",1)
    - GlobalFunction "reverse(2)"
        - Apply ("reverse xs ++ [x]", 1)
            + Apply ("(++) (reverse xs)", 1)
            + Apply ("[x]",2)
```

In the Curry meta-programming library (see Section 2.4), there is a possibility to represent programs as abstract syntax trees with all function definitions and user-defined data types. As we have already described, we first find a path for every selected expression. Then, it is only necessary to search these paths within the abstracted tree of the program to add observer functions. When selected expressions are found, observer functions are added to the abstracted tree. This results in a manipulated source code. The manipulation of programs is performed automatically by iCODE.

8.1.2 *Single Stepping*

As we have seen in Chapter 5, each time a computation of an annotated expression makes progress in the observation system of COOSy, information about observed values are recorded in a trace file. To avoid storing huge amounts of redundant data in trace files, we use client/server architecture. Clients and servers are different modules that communicate with each other by using TCP communication. The aim of using the distributed architecture is to represent intermediate steps of evaluations with the possibility of forward and backward stepping. This architecture also provides another advantage. All views are implemented by independent system processes that may even be distributed on multiple computers to avoid a slow-down or shortage of memory of observed expressions.

To generate the TCP connection between modules, we need `Sockets` (see Figure 8.2). The defined events from Chapter 5 are messages that are sent through the sockets from the observe applications to viewers. By sending the first event from the observation application, communications are started. For this purpose, a `Handle` (see Section 2.5) can be used. We implement a handle as a global state to transport requests and answers during communications between clients and servers:

```
icodeHandle :: Global [Handle]
icodeHandle = global [] Temporary

sendToShow msg = do
   h  <- readGlobal icodeHandle
   h' <- sendMessage msg h
   writeGlobal icodeHandle h'
```

To determine if a socket connection already exists, we only check to see if the `Handle` list is empty or not:

```
sendMessage msg handleTable =
   ...
   let strMsg = show msg
   case handleTable of
      [] -> do h <- connectToSocket "localhost" portNr
               hPutStrLn h strMsg
               ...
               hGetLine
               return (h:handleTable)
      _  -> do let h = head handleTable
               hPutStrLn h strMsg
               ...
               hGetLine
               return (handleTable)
```

The function `hPutStrLn` puts a message (of type String) into the handle to be sent to a viewer. Sending events can also be done in a single-step mode. In this mode the programmer can watch how the evaluation proceeds so that the representation of information in viewers can be delayed until the user presses a forward button, which is provided in the stepping window for this purpose.

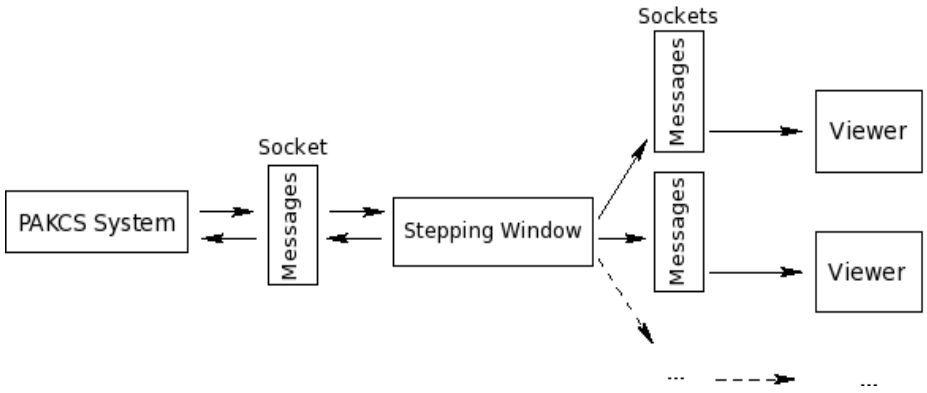


Figure 8.2: Socket connection

By pressing the forward button, an acknowledge message is sent from the stepping window to the observe application (PAKCS system) to continue the computation. The function `hGetLine` in the above code is responsible for this purpose. The next message is sent only when `hGetLine` receives an acknowledge message from the stepping window. The stepping window in this process is a server for the observe application. Communications between clients and servers can be continued only when the server accepts received messages from clients (see Section 2.5).

Messages that are sent from observe applications are represented in appropriate viewers as textual visualization. Each message contains an event and a label of the observation it belongs to. The structure of messages is defined in the following data type:

```
data MSG = EventMessage Label ICODE.Event
type Label = String
```

After every acceptance from the server (stepping window), the message is sent to the appropriate viewer that is recognized by its related label. In Chapter 4, we have seen that, for each observed expression, a label is needed to match observed values with annotated expressions. Viewers are distinguished by labels. The user may conveniently arrange these viewers on the screen and even close observations which are not interesting any more.

The distributed architecture can also be used for program coverage, as described in Chapter 7. For this purpose, we extend the defined data type `MSG`:

```
data MSG = EventMessage ...
         | LineNrMessage Label Position
type Label = String
type Position = (Int,Int)
```

`LineNrMessage` contains labels and positions of annotated expressions in programs. By sending positions from observe applications, appropriate line numbers of viewers are highlighted.

The kind of representations in viewers is distinguished by the kind of messages. Therefore, we need a message-handler in the stepping window:

```
msgHandler (EventMessage label event) handle =
    ...
    sendToTraceViewer label event
    ...
    hPutStrLn handle "send the next message"
msgHandler (LineNrMessage label position) handle =
    ...
    sendToHighlightViewer label position
    ...
    hPutStrLn handle "send the next message"
```

The function `hPutStrLn` is executed only when the representation of the next executed expression is requested from a user. Note that communications between the stepping window and viewers are also implemented by socket connections (see Figure 8.2).

The whole architecture above is activated by the user after selecting arbitrary expressions and pressing a submit button in the tool's browser (Figure 8.3). Separating

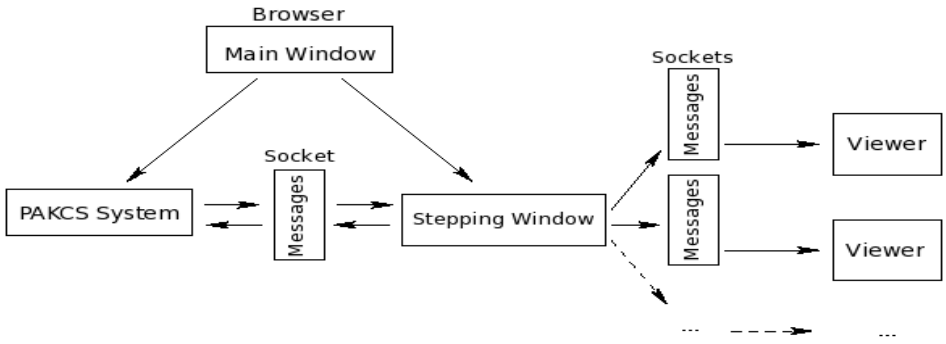


Figure 8.3: A relationship between the browser and the debugging system

the browser from the client/server mechanism has a benefit: It avoids aborting the main window when the user defined program has failures or non-terminating computations. In other words, the main window is always ready to receive any request from the user.

8.1.3 *Controlling Executions*

Locating the position of bugs in programs is a task of debuggers. Testers should specify whether or not the behavior of programs is acceptable. Tests can be performed automatically or by programmers. Sometimes, it is possible that given inputs to programs will never generate unexpected outputs, although programs do have failures. In order to detect executed and non-executed parts of programs, we use the highlighting technique that is discussed in Chapter 7. This means that program expressions should be annotated by marker functions. Every call to annotated expressions records the position of these expressions in a tree structure (see Section 7.2). Highlighting related positions in a source code is implemented with two different architectures in two different sub-tools of iCODE:

- *Run-time slicer*: When using this sub-tool, every call to annotated expressions activates the client/server architecture of the tool (Section 8.1.2) to send messages from the marker application (PAKCS system) to appropriate viewers. Whenever a value of an annotated expression is needed, the expression is executed and a message is sent to the server (viewer). The viewer highlights the position of this expression in a tree representation of the program or within the original source code using the GUI library.

  As we have discussed in Section 8.1.2, representing information in the distributed architecture of our tool can be done in a single step mode. In this mode, the programmer controls execution paths to see the behavior of programs.

- *Slicer*: In this sub-tool, we do not use the client/server architecture. It only records positions of annotated executed expressions in an intermediate file. The tool highlights all recorded positions in a viewer by pressing a forward button from the user.

8.1.4 *Arranging Test Cases*

Having only true results from the run-time system does not mean that the program has had no failure. The program should be tested by writing different test cases. Writing test cases by programmers can be almost 30 percent of the testing task. There are different implemented testers, which generate test cases automatically [15]. Unfortunately, they sometimes produce too many test cases without detecting parts of programs that do not work properly.

One of the ideas in this thesis is to represent parts of a program that are executable. This representation, besides showing the result of the program execution, helps the programmer to decide what to generate as other test cases to test non-executed parts of the

program. With our implemented tool, the programmer can also write more than one test case and run all at the same time. For this purpose, we record every written test case in the following data structure:

```
data TestCase a = Test String a
```

For each recorded test case, there is also a string which is printed as a side effect. It is only a hit for the programmer knowing which test case is executed. To run a sequence of test cases, we add a new function to the program. It generates all tests without recompiling the program. For this purpose, we define the non-deterministic function `applyTestCase`, which takes test cases as parameters:

```
applyTestCase :: TestCase a -> a
```

For example, if we would like to test the reverse program by two test cases `reverse []` and `reverse [1,2,3]`, we can add the function `applyTestCase` to the program using the following definition:

```
applyTestCase (Test "TEST CASE: reverse []" (reverse [])) =
   reverse []
applyTestCase (Test "TEST CASE: reverse [1,2,3]" (reverse [1,2,3])) =
   reverse [1,2,3]
```

This function behaves as an identity function on each test case generated by the programmer. Note that non-deterministic functions offer different results. After executing a non-deterministic function and receiving a result from the run-time system of Curry, the programmer can press a button (e.g., `";"`) to demand the next possible result. With the help of this method, the tool offers results of generated test cases before a run-time error occurs.

8.1.5 *Following Evaluations*

As we have seen in Chapter 7, program coverage is a useful feature for testing separate functions of programs. Another interesting feature of program coverage is evident when run-time values of highlighted executed expressions are also represented to the user (especially in functional logic languages that compute expressions using lazy semantics). In addition, following evaluations in source code is an interesting approach to find unexpected program results.

For example, to see the behavior of the function `reverse`, we apply this function to an empty list: `reverse []`. The result of the evaluation is an empty list, while only the

first rule of `reverse` is executed. To display this behavior, we would like to highlight the executed rule in the program and represent the result of the function `reverse` as follows:

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

  result
  ------
  []
```

For this purpose, we combine the two annotating methods (see highlighting in Section 7.2 and observing values in Section 4.2) for the new function `markAndObserve` to observe and mark expressions:

```
markAndObserve :: Observer a -> ModuleName -> Position -> a
                  -> a
```

This is an identity function on the fourth argument that is a program expression. The first argument is an observer that defines the special observation behavior for the type of the observed value. The second argument is a label to relate observations to corresponding viewers. The third argument is the position of the annotated expression in the source code or tree representation of the program.

The idea is to implement a run-time tracer that helps programmers to follow the evaluation order of expressions. For this purpose, we use the distributed architecture represented in Section 8.1.2.

By each call to the function `markAndObserve` in the distributed architecture, different messages are sent from the observe applications (PAKCS system). Messages show which expressions must be highlighted and what is the value of evaluated expressions.

With respect to the data type `MSG` discussed in Section 8.1.2, each message contains: (1) a label to distinguish different viewers, (2) the position of the evaluated expression in a tree representation of the program or within the original source code and (3) an event representing the intermediate results of the evaluated expression.

8.2 *Important Sub-Tools*

With the help of the methods that are discussed in Section 8.1, we review our different implemented sub-tools in iCODE.

8.2.1 *Curry Object Observation Interactive System - COOiSY*

Sometimes programmers wish to inspect the result of special data structures or function calls in the right-hand side of function rules. For this purpose, observers are provided.

To observe run-time values of program expressions, the desired expressions must be selected in the browser of iCODE. COOiSY annotates the selected program expressions automatically. The annotations do not change the natural behavior of programs. They allow the user to observe how the evaluation proceeds.

The intermediate results are represented to the user by a textual visualization [10, 57] which could be:

1. an underscore (`_`) for non-evaluated arguments
2. an exclamation mark (`!`) for evaluated arguments
3. a value that shows the argument is computed
4. a representation for functions by `{\arguments ->result}`
5. using a lighter color for representation of non-deterministic computations.

iCODE can also represent the intermediate results of program evaluations in a single-step mode. In this mode, the programmer can show the real order of evaluations so that the representation in a viewer can be delayed until the user presses the forward button in the stepping window (see Figure 8.4). By pressing the forward button, an acknowledge message is sent from the stepping window to the PAKCS system [17] to continue the computation. The run-time values are represented in appropriate viewing tools.

In a large computation, some steps of evaluations may be of no interest for observation. For this purpose, we provide a jumping possibility on the nth step (Figure 8.4). For example, by selecting `(+10)`, the user can jump ten steps and if it is needed, the stepping method is changed again to the single-step mode by selecting `(+1)`. De-selecting the check-box causes only the end result of computations to be represented. In each intermediate step, the user can review the last observed information using the backward and forward buttons that are provided in viewing tools.

To review the textual representation of observations and show run-time values of an expression, let us consider the function `increment`:

```
increment :: Int -> Int
increment x = x + 1
```

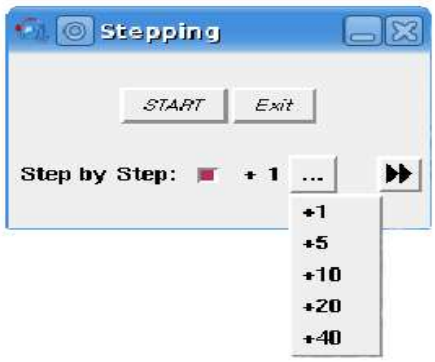


Figure 8.4: The stepping window with different actions for the forward button ((+n): jumping n steps)

We select the function `increment` to observe the behavior of this function during the execution. The evaluation order of the function `increment` is represented as follows, when it is applied to 2:

```
increment
---------
!
{ \ _ -> _ }
{ \ _ -> ! }
{ \ ! -> ! }
{ \ 2 -> ! }
{ \ 2 -> 3 }
```

By following the intermediate steps, we can observe the run-time values of the left- and right-hand side expressions of `increment`. The last step represents the result of the evaluation. The function `increment` takes 2 as an input and yields 3 as the result. The step-by-step representation of information helps the programmer to find the reason for a run-time error or an incorrect answer.

For another example, we consider the following program:

```
main = head [1,2,3]
```

Inserting only a print/trace statement into the function call `head` does not help us to know how the function `head` behaves. We would like to know how the first element of the list is computed. For this purpose, we select `head` and activate COOiSY. Figure 8.5 displays the result of `head` in a viewer of the tool. It shows that `head` returns the first

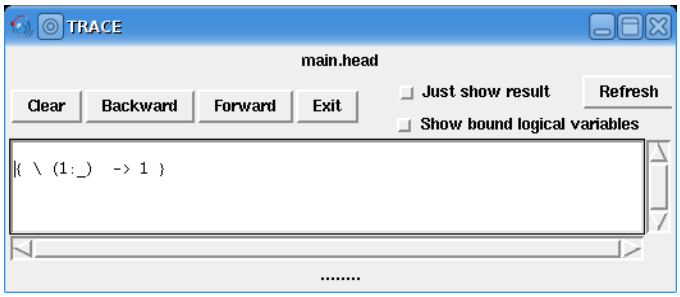


Figure 8.5: Showing the result of `head` in the viewing tool of COOiSY

element of the given list without executing the remaining elements of the list that are represented by an underscore.

By selecting more than one expression to observe, the tool opens separate viewers for the selected expressions. Separating viewers helps the user to concentrate on the special behavior of program expressions.

8.2.2 *Curry Tester - CUTTER*

One interesting feature of CUTTER represents executed parts of programs by marking executed expressions by two methods: (1) Highlighting function rules in source code, (2) Highlighting program expressions in a tree representation of the program. In both methods, the last executed expression is highlighted in red color and other executed expressions in green. Unmarked expressions belong to un-executed parts of the program. Before activating the tester, a user can select a single unit, a number of integrated units or an entire program for being tested. Units in functional programs are functions that have all of their right-hand side expressions.

For example, we apply the function `increment` in the following program:

```
test :: Int
test = increment (head [1,2,3])
```

Calling the function `test` offers `2` as a result. The computation is lazy. This means that the function `head` causes only the first element of the list to be executed. We would like to see executed parts of the program in a tree representation. The highlighted executed expressions are represented in Figure 8.6. The highlighted parts of the function `test` show that only the first element of the list has been executed because the remainder of the list is not marked. The viewer also shows that both functions are called once during the execution. Representing the number of function calls is useful to locate non-terminating computations in programs.

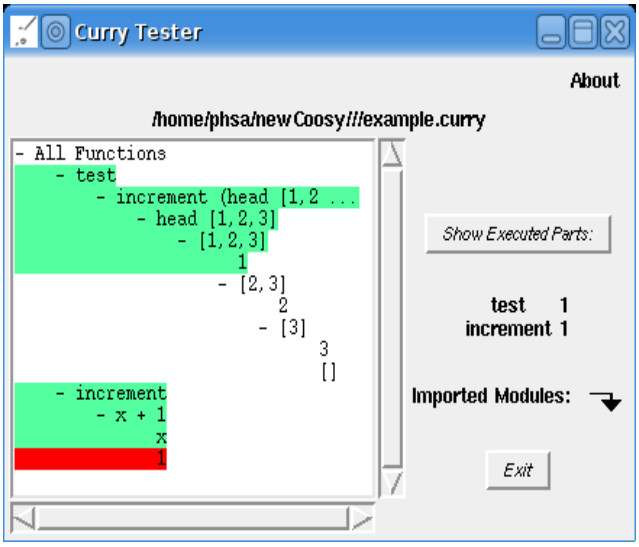


Figure 8.6: The highlighted expressions in the viewing tool of CUTTER

Let us illustrate this ability of the tester by another example. The following program computes the increment of all positive elements of a given list:

```
inc :: Int -> Int
inc x = if x>0 then x + 1 else inc x

test :: [Int] -> [Int]
test xs = map inc xs
```

The program is written with a recursive call to the function `inc` in the else-branch of this function. Every call to `inc`, when its argument is no larger than zero, results in non-terminating computations. We would like to test the program to locate the function that is executed infinitely. As the first test case, we call the function `test` with an application to the list `[1..5]` when the function `inc` is selected as a single unit to be tested (this time in a tree representation of the selected unit).

By pressing the forward button in the viewing tool, the executed expressions are highlighted and the number of function calls is displayed to the user (see Figure 8.7). The marked slices show the executed parts of the program after calling five times the function `inc` and the result `[2,3,4,5,6]` from the PAKCS system is acceptable. The viewer shows that some parts of the program are not executed (unmarked expressions). The non-executed parts belong to the else-branch of the example above. Because all elements of the given list are larger than zero (`x>0`), the then-branch is executed by each application of the function `inc` to the elements of the given list.

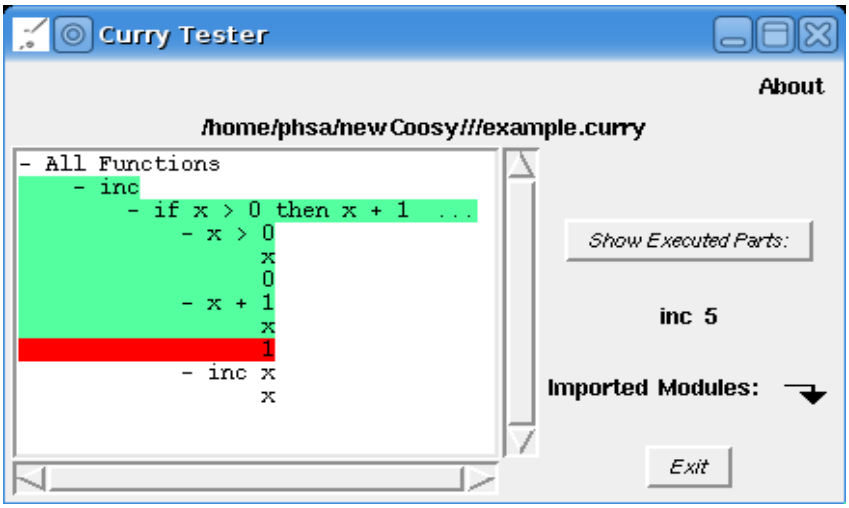


Figure 8.7: Executed parts of the function `inc` by calling `test [1..5]`

In order to test the program, we must be sure that all of its parts work properly. For this purpose, we use a test case for the non-executed parts by applying the function `test` to another list: `test [0..5]`. The first element of the given list is zero. This means that the else-branch should be executed. However, we have no result from the run-time system, although the function calls for `inc` are increased repeatedly (242 in Figure 8.8). The failure is located in the else-branch of the function `inc`. When the argument `x` is no larger than zero, the function `inc` is applied repeatedly to `x`. It brings up a non-terminating computation in the program. Figure 8.8 displays the new covered slices on

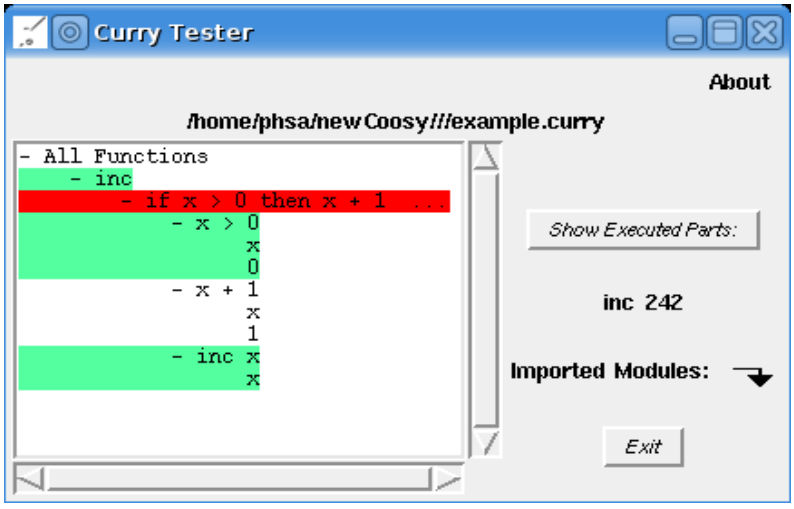


Figure 8.8: Executed parts of the function `inc` by calling `test [0..5]`

the selected function `inc`, whereas the last executed expression is highlighted in red and

the other executed expressions are highlighted in green.

We can see that having only true results after a few tests does not mean that the program has had no failure. Sometimes, a review of executed expressions helps us to concentrate on the behavior of functions in programs. In this view, which is only a text representation of programs, the user shows which expressions of the selected area are executable. Separating the color of the last executed expression from others is also useful in locating the position of a run-time error (e.g., when the PAKCS system offers only "No more solutions" as a result).

We can also write the two different test cases above at the same time and execute them by the test manager of the tool. For this purpose, we should activate the test manager from the browser (see Figure 8.9).

Figure 8.9: Test manager for arranging test cases

It is possible to write different test cases in the "New Test" place and provide a number to arrange the order of test cases. We can also delete or correct the inserted tests. Assume that we would like to apply the two test cases `test [1..5]` and `test [0..5]` from the last example in the order in which they are written. After writing the test cases, the tool adds the non-deterministic function `applyTestCase` (see Section 8.1.4) to the program:

```
applyTestCase (Test "TEST CASE: test [1..5]"
                    (test [1..5])) = test [1..5]
applyTestCase (Test "TEST CASE: test [0..5]"
                    (test [0..5])) = test [0..5]
```

Whenever the tester is activated, the function `applyTestCase` is called automatically. It generates the first test case that causes executed program expressions to be highlighted.

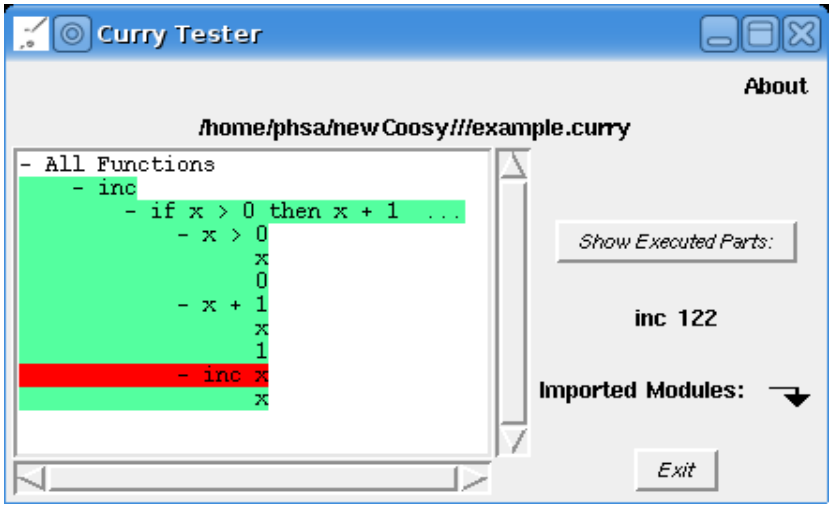


Figure 8.10: The generation of the two test cases: `test [1..5]` and `test [0..5]`

The represented information in the viewer is the same as that shown in Figure 8.7 and the PAKCS system yields [2,3,4,5,6] as the result:

```
> TEST CASE: test [1..5]
> [2,3,4,5,6]
```

By pressing a button (e.g., `";"`), we request the next result from the non-deterministic function `applyTestCase`. This time, the second test case is applied:

```
> TEST CASE: test [0..5]
>
```

The program is not terminated and the viewer shows that all program expressions have been executed (see Figure 8.10). Covering all parts of the function `inc` means that all function expressions are executable. However, the number of function calls (122 in Figure 8.10) helps us to focus on the behavior of the function `inc`. It may be caused a non-terminating computation.

8.2.3 *Curry Tracer - C-Ace*

Following evaluations within a source code is an interesting approach for finding unexpected outputs. The means to do so is provided in C-Ace.

We illustrate this ability by an example. To keep the result view small, let us take the `increment` function: `increment x = x+1`. We would like to follow the evaluation order of the function `increment` when it is selected as a single unit to be traced. To execute

an application of this function to an integer, the right-hand side of the function must be computed. This computation can be completed when the expression `x` is evaluated. This means that three expressions are evaluated: `increment`, `x+1` and `x`.

Figure 8.11 displays the last six steps of this evaluation when the program is executed by the application `increment 2` (for illustration purposes, the trace productions are represented in a sequence of viewers). The exclamation mark shows that the evaluation is started, but the computation has not yet been completed and the function application is displayed by `{\arguments ->result}`. This means that the textual visualization `{\! ->!}` in the first viewer shows that the evaluation of the left- and right-hand sides of the function `increment` has started. Because, `increment` is being evaluated in this step, it is highlighted in a darker color. In the next step, the value of `increment` is represented as `{\2 ->!}` (see viewer 2).

Now, the computation of the expression `x+1` can be continued (fifth viewer) when the expression `x` is evaluated to `2` (third viewer) and the constant `1` is also recognized (fourth viewer).

The last viewer represents the value of the function `increment` as `{\2 ->3}`. With this step, the computation is completed.

The evaluation order of expressions can be represented not only in a tree representation of the program but also in the original source code. It is assumed when the programmer wishes to follow only the evaluation order of function rules and not all right-hand side expressions. To illustrate this ability, let us consider another example:

```
reverse :: [Int] -> [Int]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

append :: [Int] -> [Int] -> [Int]
append xs ys = reverse ys ++ reverse ys
```

Suppose that we have written an incorrect variable in this program. The second call of the function `reverse` in the right-hand side of `append` is applied to `ys` instead of `xs`. Generating the test case `append [1] [2]` yields an unexpected result `[2,2]`. With the help of the tracer, we wish to follow the executed functions to find the failure. Figure 8.12 displays an intermediate step of the evaluations. The figure shows that the function `append` is highlighted in red color. This means that, in this step, `append` is being executed. `append` yields a list of numbers as the result. The evaluation of this function in Figure 8.12 shows that the first element of the list is `2` and the evaluation of the second element is started `(2:!:_)`. The computation of the second list element needs a function call on

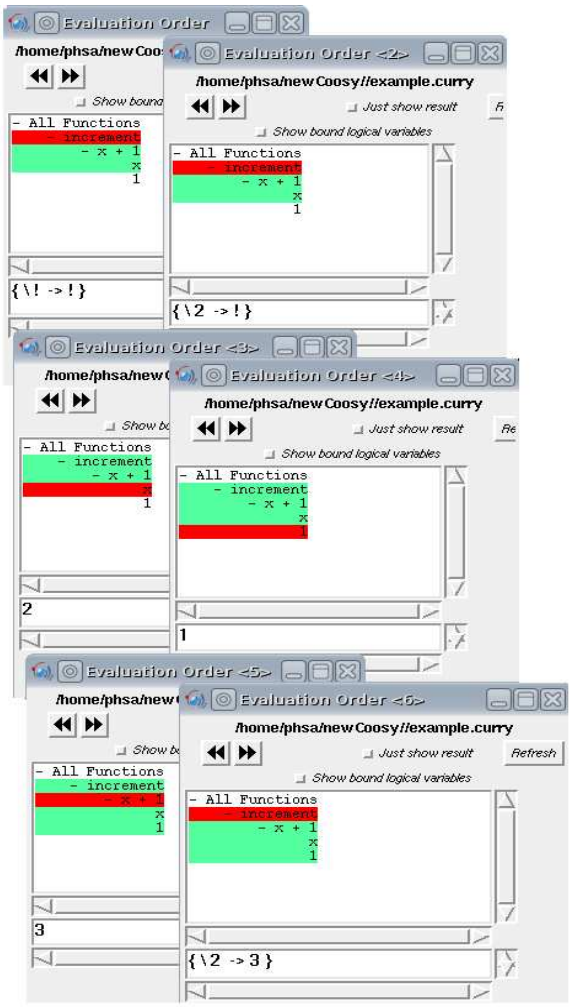


Figure 8.11: Tracing the evaluation order of the function increment

`reverse`. Figure 8.13 represents two sequence steps of the evaluations when the function `reverse` is being executed. The first application of this function in the program has offered a list with one element: `[2]`. In the second call of this function, the evaluation of the first list element is started `(!:_)` (see viewer 2). After completing the execution of the first element, it is evaluated to `2` (see viewer 3). The bug is located. The second call of the function `reverse` should be an application to the given list ([1]), but it is applied to a list with `2` as an element (see Figure 8.13).

Following the evaluation order of program expressions is useful not only for locating bugs in simple programs but also for understanding the semantics of functional logic programs. To illustrate this ability, let us consider the following program by using the data type `Nat` for natural numbers:

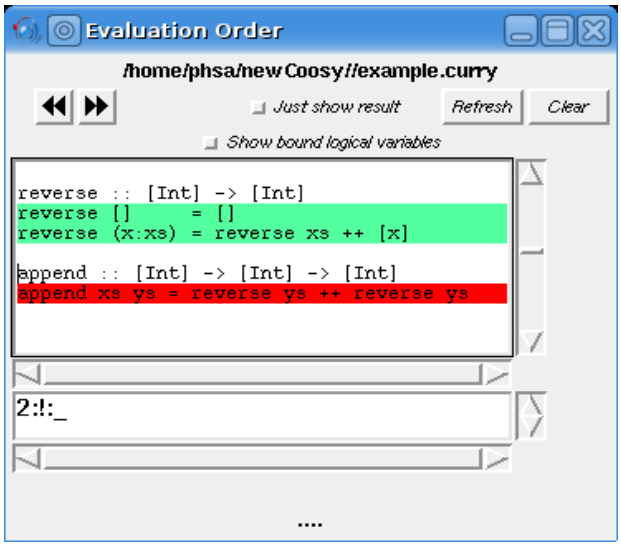


Figure 8.12: One intermediate step of the evaluation when `append` is executed

```
data Nat = O | S Nat

coin :: Nat
coin = O
coin = S O

plus :: Nat -> Nat -> Nat
plus O x     = x
plus (S x) y = S (plus x y)

main :: Nat
main = fst pair  where pair :: (Nat,Nat)
                       pair = let c = S coin ? coin
                                  p = plus O c in (c,p)
```

`coin` is defined non-deterministically and offers two results - zero and one. `plus` takes two arguments of type `Nat` and yields an addition between these two arguments. We would like to observe the lazy semantics and non-deterministic computations of the program. `main` returns the first element of `pair`. The local pattern `pair` uses both defined functions `coin` and `plus` in its right-hand side. By following the evaluation order of function definitions, we can see that the second element of the pair will not be evaluated.

Let us begin executing the program by calling `main`. After 11 evaluation steps, the first

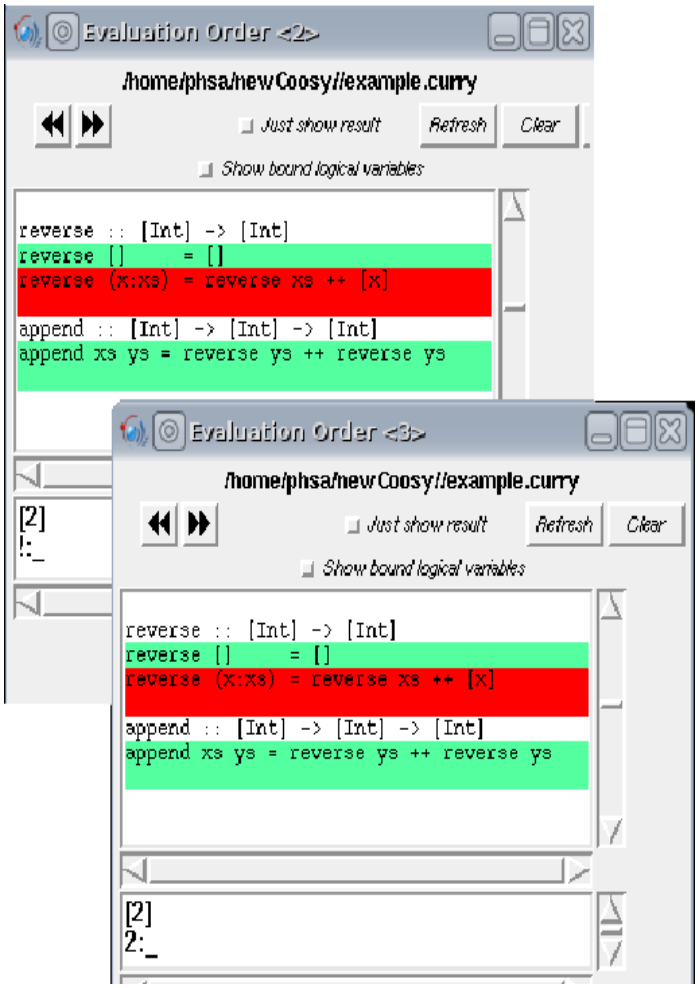


Figure 8.13: Two intermediate steps of the evaluation when `reverse` is executed

result is represented in a PAKCS-Shell: `S O`. The first view of Figure 8.14 represents the evaluated parts of the program (for deduction purposes the trace produces are represented in a sequence of viewers). The red color shows the last evaluated function. Note that both rules of `plus`, and also the second rule of `coin`, are not evaluated. We request the next result by pressing a key (e.g., ";"). The result is returned in the PAKCS-Shell after 10 steps: `S (S O)`. View 2 of Figure 8.14 displays a part of the second result in a lighter color. This is because of the non-deterministic definition of `coin`. In this step, the second rule of `coin` is evaluated and highlighted. The third result is represented after 4 evaluation steps (see View 3 of Figure 8.14): `O`. The last result is returned after 10 evaluation steps (see View 4 of Figure 8.14): `S O`. The last view shows that: (1) no rules of the function `plus` are executed, (2) the second result of `main` is computed non-deterministically and (3) `main` is called four times.

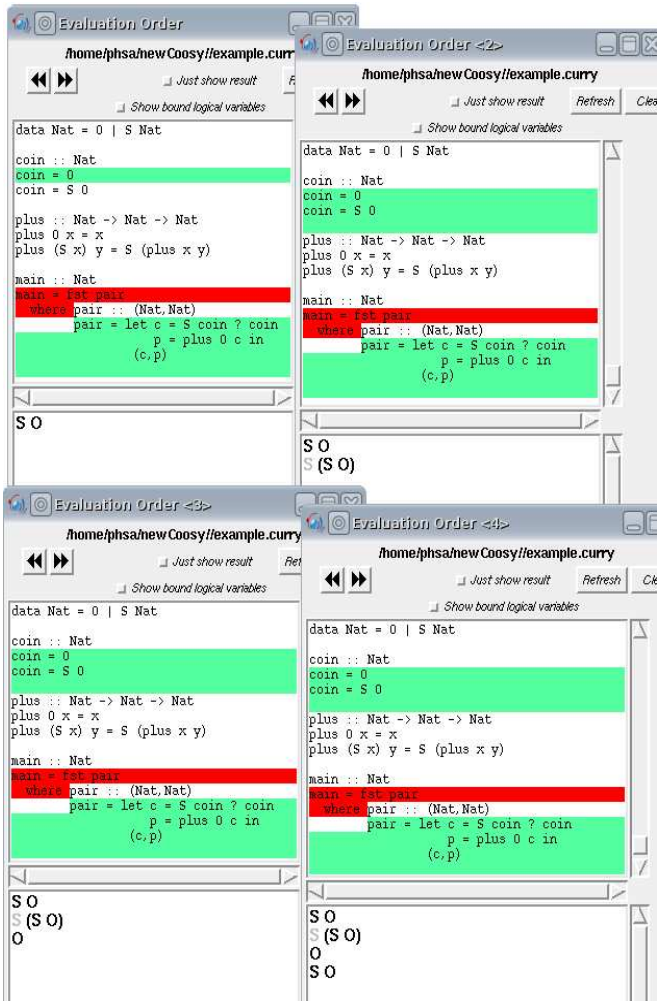


Figure 8.14: Tracing the evaluation order of function rules

By pressing the backward button, we have access to the result of the last evaluated expression `pair` (see Figure 8.15). Underlines appears in all four results of `pair` because, due to the lazy behavior of computations, the second elements of the pairs are not evaluated.

8.3 *Summary*

We have developed a source-code-level observation tool to test and trace execution generations. Figure 8.16 shows a diagram of the various parts of the tool. The browser and viewers are written in Curry using Tcl/Tk [36]. In other words, all of the code is implemented in Curry.

We use the meta-programming library of Curry to translate an original source code

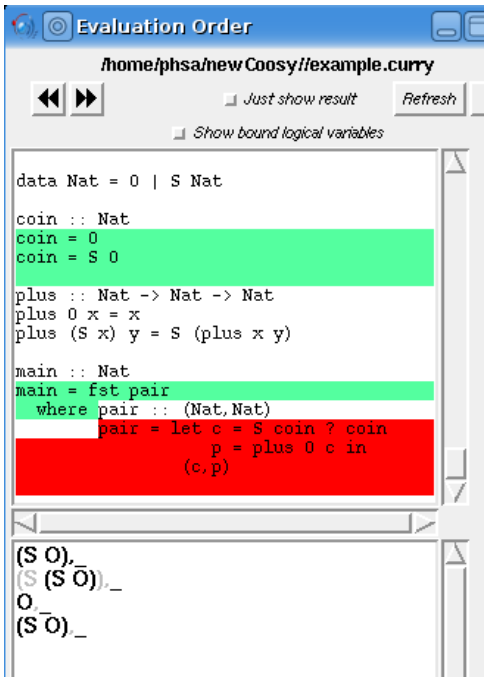


Figure 8.15: The result of the local pattern `pair`

to an abstract syntax tree (`AST`) format. Arbitrarily selected expressions from the user are annotated in `AST`. These annotations are added to `AST` with respect to the activated sub-tool from the user. A sub-tool can be:

- Single expression observer
- Unit tester
- Unit tracer

Using the meta-programming library of Curry reduces the complexity of the transformations, and allows the sub-tools to share the same `AST` for an observation of an activated program. The only difference is in arguments of annotations (identity functions). For the observer, the value of the evaluated expression is required. For the tester, the position of the executed expression in the program and the tracer needs both of the mentioned arguments. Therefore, Figure 8.16 shows the structure of all three implemented sub-tools in our observation system.

An annotated `AST` should be un-parsed to the original format of a Curry program. After un-parsing, the program is run in the PAKCS system. Note the diagram shows two different architecture designs:

- Design-1: This is implemented in the distributed architecture, to implement a run-time tester and tracer. In some cases, the user wants to see the last information of computations, showing that the order of executions in a single-step mode is not necessary. Using the tool in Design-2 is proposed in this case. It could be selected by an option in the browser.

  The execution of an annotated program in Design-1 establishes a socket connection between different modules (windows) of the tool. These modules are related to:

  – PAKCS system

  – Stepping window

  – Viewers

  By executing expressions, information about evaluations is represented in viewing tools.The representation in viewers is delayed until a message is returned to the PAKCS system from the stepper. It allows the user to observe the execution in single-step mode.

- Design-2: In this implementation we do not use the distributed architecture. In this model, information about evaluated expressions is recorded in an intermediate file. By pressing a forward button, it can be represented to the user in a viewing tool.

The viewers represent different information to the user:

- In the observer; viewers are provided for every single observed expression and display values of evaluated expressions.

- In the tester; viewers are provided for different modules (i.e., a main module and imported modules that have already been observed). They represent executed parts of programs by highlighting expressions and also display the number of function calls.

- In the tracer; like the tester, viewers are provided for different modules. Each viewer not only highlights executed expressions, but also displays values of an executed expression that is currently evaluated. We have also implemented a text handler in each viewer so that a double-click on each highlighted expression that is not currently evaluated shows values of this expression.

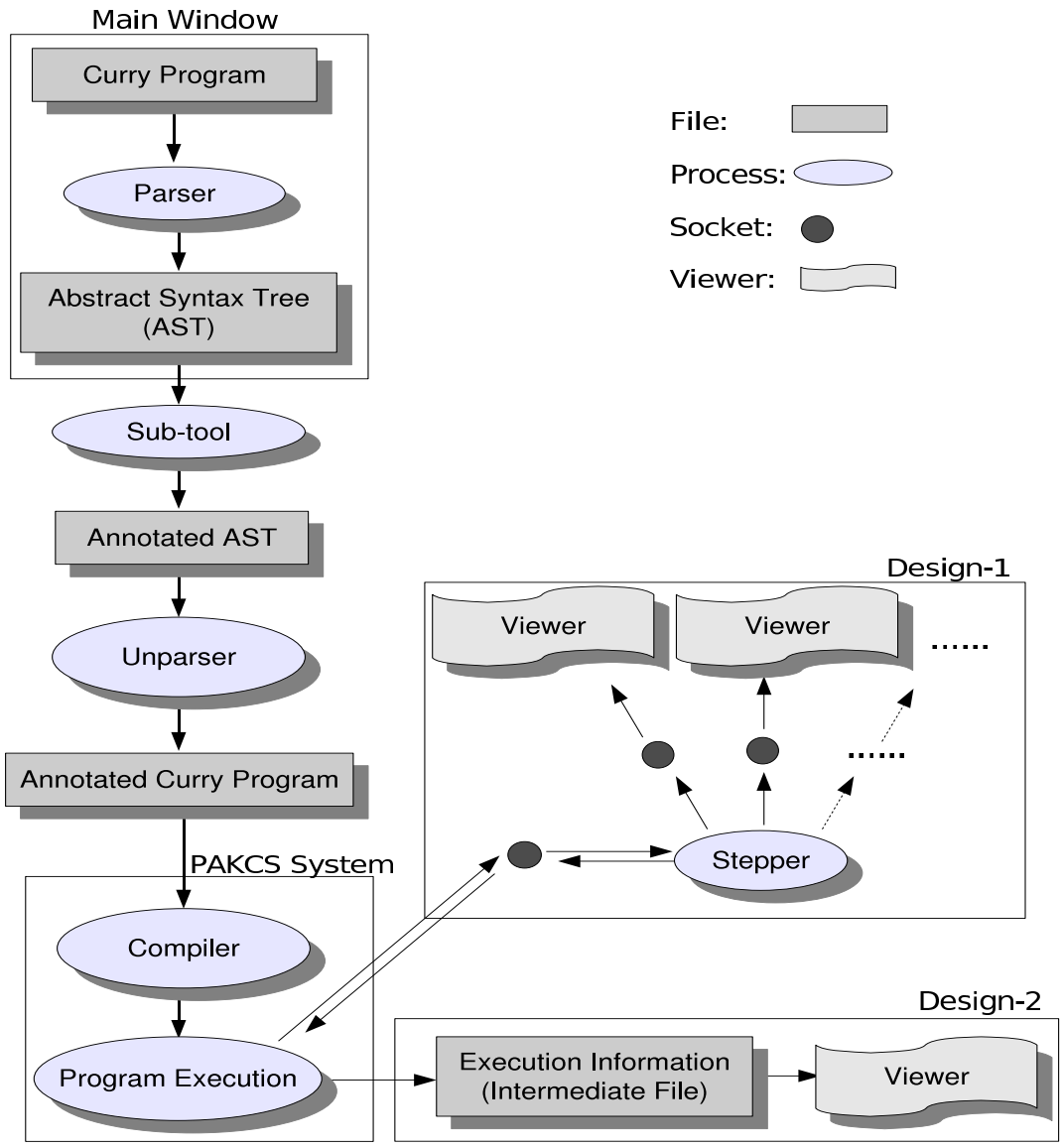


Figure 8.16: A diagram of the tool process

Chapter 9

Conclusion and Related Works

"There is nothing good or bad, but thinking makes it so."

William Shakespeare

Debuggers are magnifying glasses that a programmer can use to examine programs to determine why a program does not behave correctly. A well implemented debugger allows the programmer to follow the flow of the program evaluation and stop the execution at any desired point to inspect generated values in order to discover a reason for a failure.

This thesis offers methods to test and trace programs written in functional logic languages like Curry. These methods are intended to:

- analyze programs that allows to:
  - represent program expressions in tree structures
  - enable the user to select and de-select arbitrary program expressions
- represent executed parts (slices) of a program by a program coverage that helps to:
  - propose test cases for testing un-executed parts that have not been executed, but may be executable
  - recognize possibly dead code
  - understand lazy behavior of programs
- use a distributed architecture that allows to:

  - avoid storage of excessive information in files
  - avoid slow-down of program executions
  - perform run-time slicing
  - understand non-deterministic and lazy semantics of programs

- collect run-time values in a tree structure by a top-down constructing algorithm for observation.

These ideas are implemented in the interactive tool iCODE with different sub-tools, each of which supports an aspect of program behavior for locating bugs and understanding programs.

All of the program expressions are represented to the user in a GUI as a browser. The tool provides an ability to select one unit, a number of integrated units or an entire program or even a sub-expression for observation during a program execution. The programmer has different views of a program's behavior:

- *Observing* the evaluation order of selected expressions in separated viewers.
- *Testing* selected units by showing marked and unmarked parts that represent executed and un-executed expressions of the units.
- *Tracing* selected units by following the evaluation order of unit expressions in a tree representation of units or in source code.

`HOOD` [20] is a system that had an important influence on the design of our debugger iCODE. To use `HOOD`, programmers annotate expressions whose values they wish to observe. These applicative annotations act as identities. If the evaluation of an annotated expression is demanded by lazy evaluation, this expression reduces to a value and this value is recorded into a file. The recording of values in the file can be played back in a way that reveals the order in which their sub-expressions were demanded. `HOOD` is implemented for Haskell programs. An extension of this system for non-deterministic computations and logical variables is also provided in COOSy [10]. In contrast to `HOOD` and COOSy that programmers annotate program expressions manually, our debugging system does all required program manipulations automatically. Furthermore, we do not record the information of evaluations in a file. In our design, this information are constructed in a data term with a tree structure.

There are different tools available to locate failures in functional programs. `Hat` [71] is one of them that is implemented as a debugger for `Haskell` programs and has different components like `Hat-Explore` [14]. In contrast to `Hat-Explore` that highlights reductions

or equations of computations after program execution, our tracer highlights program expressions during an execution. In `Hat-Explore`, the programmer should navigate on an `Evaluation Dependency Tree` [47] to find a failure, but sometimes can lose orientation. Our implemented tracer automatically follows the evaluation order of related expressions in a tree representation of programs or within source code. The automatic navigation helps the programmer to find a run-time error or faulty output generated by a program. Another difference between `Hat` and our implemented tracer is that `Hat` represents a sequence of eager evaluations of the function calls, whereas our tracer shows the real lazy evaluation steps. We believe that representing lazy evaluation steps not only can be useful in locating bugs in simple programs, but also can be used to assist beginners and students in learning the non-deterministic behavior of lazy functional logic programs.

Another important component of `Hat` is `Hat-Observe` [71]. This tool shows how top-level functions are used. Meaning that, all arguments of function applications and also their results are represented to the user as a list of equational observations. In contrast to `Hat-Observe` that represents the equational observations after program execution, our interactive observer COOiSY [57] represents this information during an execution. This ability helps the user to follow the interesting aspects of program behaviour during a pattern matching. Furthermore, our tool provides a mean to observe every sub-expression of program as well as top-level functions.

As a debugger for Curry programs, `TeaBag` [5] displays information of evaluations to the programmer by providing a view of a tree structure of a computation. It is shown separately from the structure of a source program.

Following the evaluation of program expressions within lines of a source code can provide a good view to programmers for finding bugs. This is implemented in the tracer of iCODE (C-Ace). Values of expressions with a textual visualization form are represented to the programmer, while evaluated expressions are highlighted within a source program.

Program slicing in [62, 49] by removing non-executed parts of programs minimizes code to concentrate the programmer on small parts of source code in order to locate bugs. By showing marked (executed) and unmarked (un-executed) expressions, the tester of iCODE (CUTTER) [57] displays a surrounding area that might contain failures, either in executed parts that are highlighted or in non-executed parts that are not marked and should be tested by other test cases. A program coverage by highlighting program expressions is also used in `HPC` [21] for Haskell programs. [19] uses program-coverage technique not for highlighting expressions in an interactive manner but for an automatic generation of test cases.

When using `HUnit` [31] as a tester for Haskell programs, the user can write different

test expressions in programs. The tool checks whether expected results that are written in test expressions are the same as generated results from programs. The tool does not represent which expressions of programs are not executed. Because of the special test cases, sometimes some parts of programs will not be executed, although they are executable. Beside testing programs in CUTTER, highlighting expressions can be useful if we compare our tool to `HUnit`. Additionally, CUTTER represents the number of function calls not only for terminated computations, but also for non-terminated ones. In other words, if one of the generated test cases produces a run-time error, the result of the last executed expression is offered to the user.

Furthermore, iCODE is an easy-to-use tool with a source-code-level representation for programmers.

Bibliography

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Functional Logic Languages. In M. Comini and M. Falaschi, editors, *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier Science Publishers, 2002.

[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

[3] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, July 2000.

[4] S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

[5] S. Antoy and S. Johnson. TeaBag: A Functional Logic Language Debugger. In *Proc. of the 13th International Workshop on Functional and (constraint) Logic Programming (WFLP'04)*, pages 4–18, Aachen, Germany, June 2004.

[6] Sergio Antoy. Optimal Non-Deterministic Functional Logic Computations. In *ALP/HOA*, pages 16–30, 1997.

[7] Sergio Antoy. Constructor-Based Conditional Narrowing. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 199–206, New York, NY, USA, 2001. ACM.

[8] Sergio Antoy. Evaluation Strategies for Functional Logic Programming. *Electronic Notes in Theoretical Computer Science*, 57, 2001.

[9] Daniel Brand. A Software Falsifier. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, page 174, Washington, DC, USA, 2000. IEEE Computer Society.

[10] B. Braßel, O. Chitil, M. Hanus, and F. Huch. Observing Functional Logic Computations. In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 193–208. Springer LNCS 3057, 2004.

[11] B. Braßel and M. Hanus. Nondeterminism Analysis of Functional Logic Programs. In *Proceedings of the International Conference on Logic Programming (ICLP'05)*, pages 265–279. Springer LNCS 3668, 2005.

[12] 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.

[13] Rafael Caballero and Francisco Javier López-Fraguas. A Functional-Logic Perspective on Parsing. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 85–99, London, UK, 1999. Springer-Verlag.

[14] O. Chitil. Source-Based Trace Exploration. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, March 2005.

[15] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[16] Simon Peyton Jones CV Hall, K Hammond and PL Wadler. Type classes in Haskell. In *European Symposium On Programming*, number LNCS 788, pages 241–256, April 1994.

[17] M. Hanus et. al. PAKCS: The Portland Aachen Kiel Curry System, 2008.

[18] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 38(2/3):258–287, 1999.

[19] Sebastian Fischer and Herbert Kuchen. Systematic Generation of Glass-Box Test Cases for Functional Logic Programs. In *PPDP*, pages 63–74, 2007.

[20] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electr. Notes Theor. Comput. Sci.*, 41(1), 2000.

[21] Andy Gill and Colin Runciman. Haskell Program Coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 1–12, New York, NY, USA, 2007. ACM.

[22] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type Classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[23] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Principles and Practice of Declarative Programming*, pages 188–205, 1999.

[24] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *PADL '00: Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, pages 47–62, London, UK, 2000. Springer-Verlag.

[25] M. Hanus. Curry: An Integrated Functional Logic Language, 2006.

[26] M. Hanus. Multi-Paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

[27] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[28] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. volume 9, pages 33–75, 1999.

[29] Michael Hanus. Lazy Unification with Simplification. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 272–286, London, UK, 1994. Springer-Verlag.

[30] Michael Hanus and Johannes Koj. CIDER: An Integrated Development Environment for Curry.

[31] D. Herington. HUnit: Unit Testing with HUnit, 2002.

[32] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[33] Graham Hutton. *Programming in Haskell*. Cambridge Univ. Press, 2007.

[34] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 71–84, 1993.

[35] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[36] B. Welch K. Jones. *Practical Programming in Tcl and Tk*. Prentice-Hall, 2003.

[37] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[38] J. W. Klop, Marc Bezem, and R. C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001.

[39] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. GAST: Generic automated software testing. In *Implementation of Functional Languages (IFL'02)*, volume 2670 of *LNCS*. Springer, 2003.

[40] J. L. Krivine. *Lambda-Calculus, Types and Models*. Ellis Horwood, Upper Saddle River, NJ, USA, 1993.

[41] J. Launchbury. Lazy Imperative Programming. In *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, DK, SIPL '92*, pages 46–56, 1993.

[42] John Launchbury. A Natural Semantics for Lazy Evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, 1993.

[43] W. Lux. The Muenster Curry Compiler, 2004.

[44] Brian Marick. *The Craft of Software Testing: Subsystem Testing including Object-Based and Object-Oriented Testing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[45] Glenford J. Myers. *The Art of Software Testing*. John Wiley &Sons, 1979.

[46] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linkoping University, April 1998.

[47] H. Nilsson. Tracing Piece by Piece: Affordable Debugging for Lazy Functional Languages. In *ICFP '99: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, New York, NY, USA, 1999. ACM Press.

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

[49] C. Ochoa, J. Silva, and G. Vidal. Lightweight Program Specialization via Dynamic Slicing. In *Proc. of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1–7. ACM Press, 2005.

[50] Colin Runciman Olaf Chitil and Malcolm Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 176–193, September 2000.

[51] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley Longman,Inc., 1998.

[52] Thomas Plum and Dan Saks. *C++ Programming Guidelines*. Plum Hall Inc., Kamuela, HI, USA, 1991.

[53] C. Reinke. GHood – Graphical Visualisation and Animation of Haskell Object Observations. In Ralf Hinze, editor, *ACM SIGPLAN Haskell Workshop, Firenze, Italy*, volume 59 of *Electronic Notes in Theoretical Computer Science*, page 29. Elsevier Science, September 2001. Preliminary Proceedings have appeared as Technical Report UU-CS-2001-23, Institute of Information and Computing Sciences, Utrecht University. Final proceedings to appear in ENTCS.

[54] G. Revesz. *Lambda-Calculus Combinators and Functional Programming*. Oxford University Press, Inc., New York, NY, USA, 1988.

[55] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[56] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[57] Parissa Sadeghi and Frank Huch. The Interactive Curry Observation Debugger iCODE. *Electr. Notes Theor. Comput. Sci.*, 177:107–122, 2007.

[58] Norman F. Schneidewind. Body of Knowledge for Software Quality Measurement. *Computer*, 35(2):77–83, 2002.

[59] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

[60] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[61] J. Silva and O. Chitil. Combining Algorithmic Debugging and Program Slicing. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and practice of declarative programming*, pages 157–166, New York, NY, USA, 2006. ACM Press.

[62] J. Silva and G. Vidal. Forward Slicing of Functional Logic Programs by Partial Evaluation. *Theory and Practice of Logic Programming*, 7(1-2):215–247, 2007.

[63] M. T. Skinner. *The Advanced C++ Book*. Prentice Hall Professional Technical Reference, 1992.

[64] Z. Somogyi, F. Henderson, and T. Conway. Mercury: an Efficient Purely Declarative Logic Programming Language, 1995.

[65] J. Sparud. A Transformational Approach to Debugging Lazy Functional Programs, 1996.

[66] Jan Sparud and Colin Runciman. Complete and Partial Redex Trails of Functional Computations. *Lecture Notes in Computer Science*, 1467, 1998.

[67] Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd edition*. The MIT Press, 1994.

[68] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Christian Schulte, and Martin Henz. Logic Programming in the Ccontext of Multiparadigm Programming: the Oz Experience. *Theory Pract. Log. Program.*, 3(6):717–763, 2003.

[69] W. Zhou W. Jia. *Distributed Network Systems: From Concepts to Implementations*. Springer, 2004.

[70] P. L. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[71] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final proceedings to appear in ENTCS 59(2).

[72] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[73] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.