Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Debugging Non-strict Programs by Strict Evaluation

Holger Siegel

October 29, 2008

Institute of Computer Science and Applied Mathematics
Programming Languages and Compiler Construction

Supervised by:
Prof. Dr. Michael Hanus
Dr. phil. Bernd Braßel

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den _____

Contents

List of Figures

Chapter 1

Introduction

The debugging of programs written in non-strict functional languages requires sophisticated techniques that go beyond simple observation of program states, as it is common practice in the debugging of imperative programs. One approach, called *declarative debugging*, has been successfully taken in several debuggers for declarative programming languages: The dependencies between intermediate values are displayed, so that the origin of wrong results can be located.

Often declarative debuggers collect data by tracing all computations, allowing the collected data to be analyzed after the program has terminated. But memorizing all intermediate results can lead to big data structures. As an example, the Haskell debugger HAT [Sparud and Runciman, 1997] often saves megabytes of data to temporary disk files, which can slow down debugging noticably.

In [Braßel et al., 2007], a different approach is developed: Instead of collecting all intermediate values, it is only recorded which intermediate values have to be evaluated and which of them can be skipped. For the resulting data structure the name *oracle* has been coined.

With this information at hand, the non-strict program can be evaluated a second time in strict evaluation order. But strict evaluation order resembles the dependency between intermediate values very closely: Declarative debugging techniques can be utilised without the need to collect intermediate values beforehand.

This thesis contributes a new formal model which leads to more efficient implementations than the one from [Braßel et al., 2007]. Based on this model, it describes how the creation of oracles as well as the oracle-directed debugging of functional programs can be implemented by program transformations. It also treats the yet-unresolved issue of how circular declarations fit into this context.

1.1 Declarative Debugging

In this chapter the following small Haskell program will be used to demonstrate some aspects of debugging declarative programs:

$$
\begin{aligned}
&\mathit{length}\ (\_ : \mathit{xs}) = \mathit{length}\ \mathit{xs}\\
&\mathit{length}\ [\,] \qquad\quad = 0\\
&\mathit{take}\ n\ (x : \mathit{xs}) = x : \mathit{take}\ (n-1)\ \mathit{xs}\\
&\mathit{take}\ 0\ \_ \qquad\quad = [\,]\\
&\mathit{main} = \mathit{length}\ (\mathit{take}\ 2\ [1,2,3])
\end{aligned}
$$

When *main* is evaluated, the value 0 is returned, although one would expect the value 2. In this example, the error is easy to spot: In the definition of function *length* the first alternative should be

$$\mathit{length}\ (\_ : \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}$$

In order to find errors in a larger program the user may be required to observe the evaluation of the program. One way of tracing the execution of a program is to print out every function call together with its arguments at the time when it is evaluated.

By tracing the non-strict evaluation of *main* this way, the following list of function calls is displayed:

$$
\begin{aligned}
&\mathit{length}\ \_\\
&\mathit{take}\ 2\ [1,2,3]\\
&\mathit{length}\ \_\\
&\mathit{take}\ 1\ [2,3]\\
&\mathit{length}\ \_\\
&\mathit{take}\ 0\ [3]\\
&\mathit{length}\ [\,]
\end{aligned}
$$

In current implementations of functional languages there is usually no way to reconstruct the source code of yet unevaluated expressions, so-called *thunks*, from their compiled representation. Therefore, some placeholder has to be displayed instead of an unevaluated thunk. Here the convention to represent unevaluated thunks by an underscore "_" is followed.

When function *length* is called, it forces the evaluation of its argument, so that in the next step the expression *take* 2 $[1,2,3]$ is evaluated. Due to the lazy order of evaluation only the first node of the resulting list is computed, whereas the tail of the list is returned as a suspended evaluation of the expression *take* 1 $[2,3]$. Then function *length* is called recursively with the unevaluated list tail as its argument, again forcing a call to function *take*. After returning from function *take*, function *length* is called again, forcing a third call to function *take*.

Tracing the actual order of evaluation can give valuable insight into the run-time behavior of a non-strict functional program. It can also help in finding the cause of run-time errors. For Haskell programs the GHCi Debugger, which is part of current versions of the Haskell interpreter GHCi, is able to visualize the non-strict evaluation of Haskell programs [Marlow et al., 2007].

However, when one tries to find out why some program returns a wrong result, this approach shows two major disadvantages:

- The call sequence does not reflect the structure of the source code, as can be seen from the example: Conceptually, the value of the expression *take* 2 $[1, 2, 3]$ is computed, and then function *length* is applied to this value. But in the call sequence the focus of attention repeatedly switches between the functions *length* and *take*.

- Often functions are applied to unevaluated arguments, whose evaluation will be forced when the function is applied. As a consequence, these arguments show up as unevaluated thunks in the execution trace, although the resulting value depends on their values: In the example there is no way to distinguish between the different calls to function *length*.

The technique of *declarative debugging* (also called *algorithmic debugging*) does not show these disadvantages. Recent declarative debuggers for non-strict functional languages collect data while the program is evaluated and produce an *evaluation dependence tree* (EDT) from the collected data. This EDT can then be analyzed by debugging tools after the program has terminated [Nilsson and Sparud, 1997].

In an EDT every node corresponds to a function call. It contains the function name, the arguments and the resulting value of the function call. When another function is called in the evaluation of that function, the corresponding EDT node will be a child node of the current EDT node.

Using EDTs for declarative debugging is an *offline* debugging technique: First a modified version of the program is run. It behaves like the unmodified program, but in the background it collects information about the function calls and their results. Then the user can search the resulting EDT for the function call that has caused the error.

Evaluating the example program results in the EDT shown in figure 1.1 on the following page. By inspecting it in a systematic way the error can easily be found:

1. The function call and the resulting value of the root node are displayed to the user. The expression *length* (*take* 2 $[1, 2, 3]$) should evaluate to 2, but the actual result is 0. Therefore, either the root node or some of its children contain a bug.

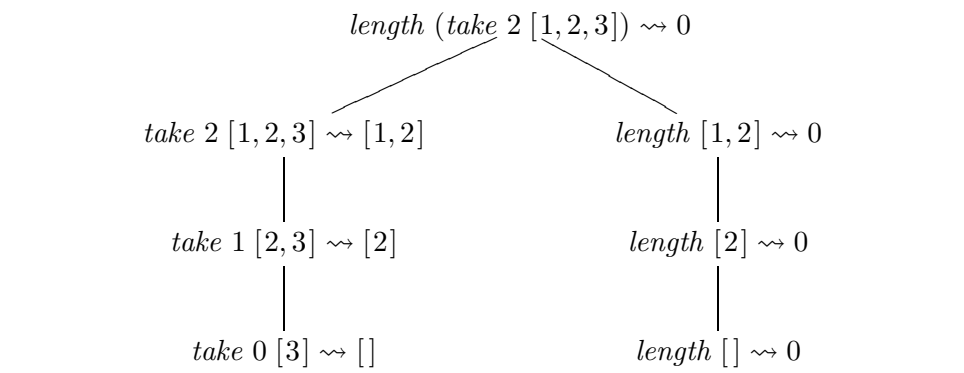


Figure 1.1: Evaluation dependence tree

2. In order to decide whether the root node or one of its children is the source of the program error, the computation represented by the first child node is inspected; it corresponds to the expression *take* 2 $[1, 2, 3]$ together with the correct result $[1, 2]$. Thus, the program error cannot be caused by this function call.

3. The other child node is inspected: The value of *length* $[1, 2]$ should be 2, but it is actually 0. Therefore, either this node or some of its children is the source of the program error.

4. Again, we move to the child node of the current EDT node in order to decide whether the current node or some of its children give rise to the error: The function call *length* $[2]$ incorrectly evaluates to 0.

5. The only child node of the current nodecorresponds to the expression *length* $[\,]$, which yields the correct result 0.

6. Therefore, we go back to the node corresponding to the expression *length* $[2]$. We have excluded every other possibility, and so we know that the error came directly from the program rule

$$length\ (\_ : xs) = length\ xs,$$

which has determined the value of *length* $[2]$.

This algorithm has been implemented in the Haskell debugger Freja. The Haskell debugger HAT uses a similar approach. It records the whole evaluation history in a so-called *redex trail*, from which the EDT can be reconstructed [Sparud and Runciman, 1997]. For a brief introduction to Freja and HAT see [Chitil et al., 2001].

All these implementations have in common that they need to record every function application, including all intermediate results. The collected

data can amount to megabytes or even gigabytes of information for larger programs.

1.2 Declarative Debugging by Strict Evaluation

In [Braßel et al., 2007] another approach is taken: Declarative programs are executed in strict evaluation order, overcoming the need to record large intermediate data structures.

Evaluating the example program expression in strict order gives the following trace:

take 2 $[1,2,3]$
  take 1 $[2,3]$
    take 0 $[3]$
length $[1,2]$
  length $[1]$
    length $[\,]$

In this trace the calls to subcomputations are indented, reflecting the structure of the function call graph. One can see that the disadvantages of tracing non-strict program executions don't arise here:

- The trace resembles the Evaluation Dependency Tree much closer than the non-strict evaluation trace.
- When a function is called, its arguments are already evaluated.

Using strict evaluation, Declarative Debugging works similar to the EDT-based approach described above. The only difference is that some subexpressions have to be re-evaluated, because the results of subfunction calls are not cached: After the user has rated the result of function *main* as wrong, both the computation of *take* 2 $[1,2,3]$ and *length* $[1,2]$ has to be repeated, so that the user can decide whether one of them has caused the error: Time is traded for space by replacing the need to create huge intermediate data structures with the need to re-evaluate parts of the program.[1]

Switching from lazy to strict evaluation also makes it easier to utilise other debugging techniques. For example, stack traces that resemble the call structure of a program can easily be derived from the system state; under lazy evaluation this requires additional bookkeeping.

[1] Obviously, improvents of the run-time behaviour can be achieved by skipping the debugging of trusted subfunctions (e.g. library functions), or by memorizing the results of selected subcomputations. These options require only minor changes to the debugging algorithm, so we won't discuss them here.

The Problem

There are programs that can be reduced to normal forms under a non-strict evaluation order, but not under a strict evaluation order. The sources of this problem can be divided into four groups:

Laziness. It is well-known that there are expressions that can be reduced to a normal form under a non-strict evaluation regime, but not under strict evaluation: There are expressions that have no normal form at all, because their evaluation does not terminate or provokes a run-time error. If such an expression is passed as parameter or bound to an identifier, but not used in the following evaluation, then strict evaluation will fail, although non-strict evaluation might have led to a normal form.

Infinite data structures. In non-strict functional languages infinite data structures like the Haskell list $[1..]$ can be defined; only the parts that are actually needed will be created at run-time. In the context of this work infinite data structures can be seen as a special case of *laziness*: At some point unfolding the data structure has to stop in order to avoid non-termination.

Unspecified order of evaluation. In non-strict functional languages the order of evaluation is not specified, so switching to strict evaluation order will not influence the resulting value. But in some cases it is not possible to evaluate an expression in strict order: For mutually recursive declarations it is not defined what a "strict" order of evaluation would be. Single recursion can also require non-strict evaluation, as the Haskell expression

$$\textbf{let } x = [1, 2, \mathit{length}\ x] \textbf{ in } \mathit{sum}\ x$$

witnesses. It will turn out that recursive declarations are the only source of this problem.

Cyclic data structures. In non-strict programming languages recursive declarations can be used to create cyclic data structures. For example, the Haskell expression

$$\textbf{let } \mathit{ones} = 1 : \mathit{ones} \textbf{ in } \mathit{ones}$$

will be evaluated to a cyclic list in many implementations of the language Haskell. From the user's point of view this cyclic list can be seen as an infinite list where every list node contains the number one. In

strict functional languages cyclic data structures can not be defined;[2] they can only be approximated by finite lists of arbitrary size.

Chapter 2 discusses how strict evaluation can be made possible even in the presence of recursive declarations. A solution of the problem of unneccessary evaluation is sketched in the next section.

1.3 Lazy Call-By-Value Evaluation

Under a non-strict evaluation order, the evaluation of subexpressions may be skipped if the value of the whole expression does not depend on them. If a subexpression has no normal form, then it even has to be skipped; otherwise the whole evaluation will fail.

Under a lazy evaluation strategy, unneeded subexpressions are skipped automatically. But when a strict evaluation strategy is used, it must be known beforehand whether the evaluation of a subexpression will terminate. Because of the well-known fact that the termination of a program cannot always be determined a priori, the only general way to find out which subexpressions have to be skipped is to evaluate the whole expression in non-strict order.

In [Braßel et al., 2007] it is shown how this approach can be put into practice. In their work, a declarative program is executed twice:

1. The program is evaluated under a lazy evaluation order. In doing so, a list of truthvalues, the so-called *oracle*, is created. Every entry relates to a redex that occurs in the evaluation of the main expression. Its value shows whether the redex has to be evaluated or not.

   The list entries are ordered with respect to the strict evaluation order, so that the entries can be accessed efficiently when the program is executed in strict order.

2. The program is executed in strict order, consuming the oracle list elementwise: depending on the value of the current entry, the next redex is either evaluated or replaced by a placeholder value.

The list of oracle entries is implemented as a list of natural numbers, using a simple form of run-length encoding: an entry n stands for n entries of value *true*, followed by one entry of value *false*. In the following example only the values of a and b are needed to calculate the value of the whole expression:

[2] Strict functional languages like Scheme or SML offer imperative extensions that can be used to create cyclic structures by modifying the program state. In this work we refrain from introducing stateful operations into otherwise purely functional programs — with the exception of a small implementation specific detail in chapter 4.

```
let a = 1 + 1          -- true
in let b = a + 2       -- true
   in let c = c + 3    -- false
      in b
```

The resulting oracle is encoded as a one-elemented list [2] that encodes the list of Boolean values *true*, *true*, *false*.

This way the problem of *laziness* and *infinite data structures* is solved. What remains to be solved is the problem of *unspecified evaluation order* and *cyclic data structures*.

1.4 The Structure of This Work

Chapter 2 discusses how recursive declarations can be dealt with in the context of Lazy Call-by-Value evaluation.

In Chapter 3 a natural semantics for Lazy Call-by-Value-Evaluation as well as for the non-strict creation of oracles is presented.

Chapter 4 discusses how the semantics of non-strict oracle creation can be implemented in an efficient way.

In Chapter 5 it is investigated how non-strict declarative programs have to be transformed into strict programs, so that they can be run by Lazy Call-by-Value evaluation.

Chapter 6 gives a summary of the practical experiences that have been made in the progress of this work and summarizes the achieved results.

Chapter 2

On Recursion

2.1 Recursive Declarations and Evaluation Order

One difference between strict and non-strict evaluation is that under a non-strict reduction strategy the order of evaluation is determined at run-time, whereas under a strict reduction strategy the order of evaluation is fixed. The following Haskell expression can be reduced to a normal form under a non-strict evaluation strategy, but not under a strict evaluation strategy:

$$\begin{array}{l}\textbf{let } x = \textbf{let } y = \mathit{length}\ x \\ \qquad\qquad \textbf{in } [1, 2, y] \\ \textbf{in } \mathit{sum}\ x\end{array}$$

A non-strict evaluator will defer the evaluation of y and build up the list $[1, 2, y]$ first. Then it will determine the value of y by counting the elements of this list without touching the actual value of its elements. After calculating the length of the list it will use the values of the list elements to calculate their sum.

A strict evaluator will fail to evaluate this expression: Since function *length* is strict in its argument, the value of x is needed in order to determine the value of y. But strict evaluation order requires that the value of y is computed before the list $[1, 2, y]$ is returned as a result.

The problem of unspecified evaluation order can only arise from recursive declarations. The reason is that in the absence of recursive declarations, the values of the declarations an expression refers to cannot refer to this expression itself. Thus, their values can always be obtained before the expression is evaluated.

Nevertheless the example above can be transformed to an equivalent expression that may be evaluated in sequential order: using a least fixed point combinator *fix* of type $(a \to a) \to a$, it can be rewritten as

$$\textbf{let } x = \mathit{fix}\ ((\lambda y \to [1, 2, y]) \circ \mathit{length})\ \textbf{in } \mathit{sum}\ x$$

In the untyped lambda-calculus, the fixed point combinator *fix* can be defined by a non-recursive function, so that this expression is perfectly non-recursive. But it is still not possible to evaluate it in strict order: the application of *fix* will not terminate, trying to build up an infinite data structure. In fact, we have replaced a cyclic data structure with an infinite data structure.

In the context of this work, this is not problematic: Lazy Call-by-Value evaluation makes it possible to evaluate programs that contain infinite data structures in strict evaluation order.

2.2 The Non-locality of Non-sequentiality

Non-sequentiality can only arise from recursive declarations, so one might hope for some evaluation strategy that allows non-recursive functions to be evaluated sequentially, delaying only right hand sides of recursive declarations.

But the following example[1] shows that such a strategy does not exist; the non-sequentiality introduced by a recursive declaration is propagated to the functions which are called in that declaration:

$$
\begin{aligned}
&\mathit{two} :: \mathit{Int} \to \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\
&\mathit{two}\ x\ y = (x * x, y + y) \\
&g\ z = \mathbf{let}\ (a, b) = \mathit{two}\ z\ a\ \mathbf{in}\ b \\
&h\ z = \mathbf{let}\ (a, b) = \mathit{two}\ b\ z\ \mathbf{in}\ a
\end{aligned}
$$

In both functions g and h the function *two* is part of a circular depenency:

- In function g the second argument of the call to function *two* depends on the first component of the result of this call. Therefore, the first component has to be evaluated first and then fed back to the function as its second argument.

- In function h the first argument depends on the second component of the result. Therefore, the second component has to be evaluated first and then fed back to the function as its first argument.

This shows that in the presence of recursive declarations even seemingly sequential functions need to be evaluated in non-strict evaluation order.

Nevertheless, for each of the functions g and h a specific order in which function *two* has to be evaluated can be found. The following example shows that this is not possible in general. It is a variation of the previous example, in which the order of evaluation is determined by the argument p of function kt, so that the evaluation of *two* has to be scheduled dynamically:

[1] The examples in this section are taken from [Schauser and Goldstein, 1995].

$$
\begin{aligned}
&kt\ p\ z = \mathbf{let}\ (a,b,c) = \mathbf{if}\ p\ \mathbf{then}\ (y,z,x)\ \mathbf{else}\ (z,x,y)\\
&\qquad\qquad\ \ (y,x)\ \ \ = two\ b\ a\\
&\qquad\ \mathbf{in}\ c\\
&g\ z = kt\ True\ z\\
&h\ z = kt\ False\ z
\end{aligned}
$$

In the following variation, non-sequentiality arises from the use of higher-order functions; there is not even an explicit conditional expression that indicates the need for dynamic scheduling:

$$
\begin{aligned}
&f1\ x\ y\ z = (y,z,x)\\
&f2\ x\ y\ z = (z,x,y)\\
&kt2\ f\ z = \mathbf{let}\ (a,b,c) = f\ x\ y\ z\\
&\qquad\qquad\ \ (y,x)\ \ \ = two\ b\ a\\
&\qquad\ \mathbf{in}\ c\\
&g\ z = kt2\ f1\ z\\
&h\ z = kt2\ f2\ z
\end{aligned}
$$

These examples show that

1. even the presence of one single recursive declaration can influence the evaluation order of other, seemingly unrelated parts of a program, and
2. there is no general way to determine the order of evaluation at compile time.

Therefore, Lazy Call-by-Value evaluation requires that recursive declarations are replaced by non-recursive declarations.

2.3 Cyclic references

It is possible to replace recursive declarations by non-recursive declarations, for example by using a least fixed point operator *fix*. But for two reasons it is not satisfying to model every recursively defined value by the application of a fixed point combinator: First, for every recursively defined value the strict evaluator must know how deep the combinator *fix* has to be unfolded until the reapplication may stop. This information has to be provided beforehand. Second, unfolding these declarations costs a lot of computing resources.

As a consequence, we allow cyclic references under two circumstances:

1. The value that is recursively referred to is used only in a non-strict way, so that the whole expression can be reduced to a weak head normal form before the actual value of the reference is known.[2]

   One example of this is the expression

[2] This is similar to the `letrec` construct of the programming language Scheme and to the way the operator *mfix* allows to introduce monadic value recursion in Haskell.

$$\textbf{let } x = [1, 2, \mathit{length}\ x] \textbf{ in } x$$

2. The value that is recursively referred to does not need to be evaluated, because it is already in weak head normal form. This particularly holds for definitions of constructor terms and λ-abstractions.

   Examples of this are the recursive data declaration

   $$\textbf{let } \mathit{ones} = 1 : \mathit{ones} \textbf{ in } \mathit{ones}$$

   and the recursive function declaration

   $$\textbf{let } \mathit{fac}\ n = \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fac}\ (n-1) \textbf{ in} ...$$

This restricted form of cyclic references does not introduce the problem of unspecified evaluation order. The reason is that a recursively referred to variable does not impose restrictions on the evaluation order

- when it is used only in a non-strict way, because it does not need to be evaluated in order to calculate the whole expression, or
- when it is in weak head normal form, because it is already evaluated.

In this work the term *strict evaluation* is also used for this modified form of strict evaluation.

2.4 Removing Recursion

According to [Jones, 1987], the semantics of recursive declarations can be explained by the equation

$$\textbf{let } x = e_1 \textbf{ in } e_2 \equiv \textbf{let } x = \mathit{fix}\ (\lambda x \rightarrow e_1) \textbf{ in } e_2$$

where *fix* is a least fixed point combinator, i.e. it obeys the rule

$$\mathit{fix}\ f \equiv f\ (\mathit{fix}\ f).$$

Since we allow for recursive function declarations in our extended version of strict evaluation, *fix* can easily be defined by

$$\mathit{fix}\ f = f\ (\mathit{fix}\ f)$$

It is obvious that this definition of *fix* satisfies the above rule. Therefore, any single recursive declaration can be transformed into an equivalent non-recursive declaration. But in Haskell a recursive declaration may define more than one variable, so that the values of these variables mutually depend on each other. A mutually recursive declaration has the form:

$$
\begin{array}{l}
\textbf{let } x_1 = e_1 \\
\quad\quad x_2 = e_2 \\
\quad\quad ... \\
\quad\quad x_n = e_n \\
\textbf{in } e
\end{array}
$$

The expressions $e_1, ..., e_n$ may refer to the variables $x_1, ..., x_n$, so that the declaration cannot easily be broken into smaller declarations. . An easy way to remove mutual recursion is to make the definition of one variable local to the definitions of the other variables and the body of the **let**-expression:

$$
\begin{array}{l}
\textbf{let } x_2 = \textbf{let } x_1 = e_1 \textbf{ in } e_2 \\
\quad\quad ... \\
\quad\quad x_n = \textbf{let } x_1 = e_1 \textbf{ in } e_n \\
\textbf{in let } x_1 = e_1 \\
\quad \textbf{in } e
\end{array}
$$

By subsequently applying this transformation one can replace every mutually recursive declaration by a single recursive declaration, which can be evaluated the way described above. Therefore, the presence of mutually recursion does not prohibit a program from being evaluated in strict order.

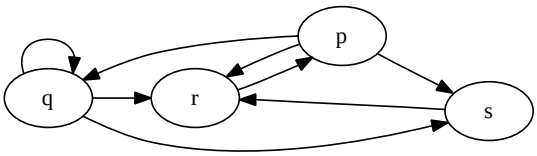
This transformation introduces inefficiencies by duplicating code, so in practice other ways of turning mutual recursive declarations into single recursive declarations are desirable.

Efficient removal of mutually recursive declarations

In order to survey different methods of transforming mutual recursion into single recursion, the following Haskell expression is used as an example:

$$
\begin{array}{l}
\textbf{let } p = \mathit{length}\ r \\
\quad\quad q = p : q \\
\quad\quad r = [p, \mathit{head}\ q, s] \\
\quad\quad s = \mathit{head}\ q + p \\
\textbf{in } \mathit{sum}\ r
\end{array}
$$

The *dependency graph* of this expression shows that each of the variables p, q, r and s depends on the other variables recursively, so it is impossible to break the declaration into two or more nested declarations:



In this graph an arrow from a node a to a node b means that variable a is referred to by the definition of variable b.

In [Braßel et al., 2007], a more efficient method of turning a mutually recursive declaration into an equivalent declaration of a single variable is sketched: instead of defining n variables, a tuple with n components is defined, so that each component of that tuple contains the value of one of the variables. Then the values of the variables can be accessed by decomposing the tuple via a **case**-expression. Applying this transformation to the example expression results in the following single recursive expression:

$$
\begin{array}{l}
\textbf{let } \mathit{rec} = \textbf{let} \quad p' = \textbf{case } \mathit{rec} \textbf{ of } (p, \_, \_, \_) \to p \\
\qquad\qquad \textbf{in let } q' = \textbf{case } \mathit{rec} \textbf{ of } (\_, q, \_, \_) \to q \\
\qquad\qquad \textbf{in let } r' = \textbf{case } \mathit{rec} \textbf{ of } (\_, \_, r, \_) \to r \\
\qquad\qquad \textbf{in let } s' = \textbf{case } \mathit{rec} \textbf{ of } (\_, \_, \_, s) \to s \\
\qquad\qquad\quad \textbf{in let } p = \mathit{length}\ r' \\
\qquad\qquad\quad \textbf{in let } q = p' : q' \\
\qquad\qquad\quad \textbf{in let } r = [p', \mathit{head}\ q', s'] \\
\qquad\qquad\quad \textbf{in let } s = \mathit{head}\ q' + p' \\
\qquad\qquad\qquad \textbf{in } (p, q, r, s) \\
\textbf{in let } r = \textbf{case } \mathit{rec} \textbf{ of } (\_, \_, r, \_) \to r \\
\quad \textbf{in } \mathit{sum}\ r
\end{array}
$$

In this expression one can spot at least two sources of inefficiency:

- In the declaration of r, the right hand side refers to the variables p' and q', although r could have been defined in terms of the variables p and q. In the presence of recursive definitions this does not influence the run-time behavior. But if recursive declarations are modelled as least fixed points that are approximated by successively unfolding some function, then accessing the values of p' or q' will force the evaluation of *rec*, which in turn will force another step of unfolding.

  This one step of unfolding is saved when the right hand side of the definition of r is changed to

  $$[p, \mathit{head}\ q, s']$$

- The definitions of q and s refer to p', although they could as well refer to p. This will not avoid any unfolding steps of *rec*, because *rec* still has to be scrutinized in order to retrieve the value of q'.

  But after changing the definitions of p, q and r, the first component of tuple *rec* will not be needed at all!

By applying these optimizations the following expression is obtained:

$$
\begin{array}{l}
\textbf{let } \mathit{rec} = \textbf{let } q' = \textbf{case } \mathit{rec} \textbf{ of } (q, \_, \_) \to q \\
\quad \textbf{in} \qquad \textbf{let } r' = \textbf{case } \mathit{rec} \textbf{ of } (\_, r, \_) \to r
\end{array}
$$

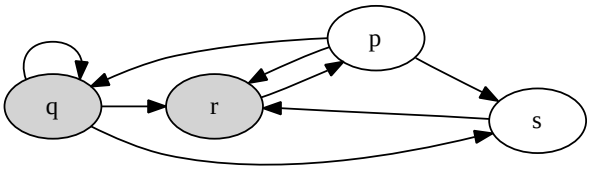
```
in      let s' = case rec of (_, _, s) → s
        in let p = length r'
        in let q = p : q'
        in let r = [p, head q, s']
        in let s = head q + p
           in (q, r, s)
in let r = case rec of (_, r, _) → r
   in sum r
```

But this is still not optimal: If we had initially chosen to define s before r, then for the same reason the third component of *rec* would not be needed, too. This demonstrates that there is a dependency between the applicability of these optimizations and the order in which the variables are defined.

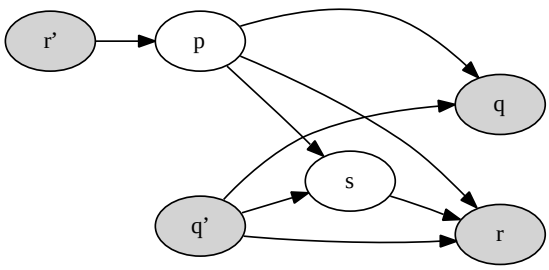
Instead of trying to find an order that allows for maximal optimization, we look for a minimal set of variables that has to be *fed back* to the recursive declaration via the tuple *rec*. In other words, we have to find a set of vertices that can be removed from the dependency graph of a recursive declaration, so that the remaining graph contains no cycles. In graph theory such a set of vertices is called a *feedback vertex set*, and the problem of finding small feedback vertex sets is well investigated [Festa et al., 1999].

Definition (Feedback vertex set)**.** *W is called a* feedback vertex set *of a directed graph (V, E), if $W \subseteq V$ and $(V \setminus W, E \cap (V \setminus W \times V \setminus W))$ acyclic.*

In the above example $\{q, r\}$ is a feedback vertex set:



Splitting q and r into "sources" q', r' and "sinks" q, r leads to a cycle-free dependency graph:



Now the example expression can be transformed to the following code:

$$
\begin{array}{l}
\textbf{let } rec = \textbf{let } q' = \textbf{case } rec \textbf{ of } (q, \_) \rightarrow q \\
\quad \textbf{in} \quad\;\; \textbf{let } r' = \textbf{case } rec \textbf{ of } (\_, r) \rightarrow r \\
\qquad\quad\;\; \textbf{in let } p = length\ r' \\
\qquad\quad\;\; \textbf{in let } q = p : q' \\
\qquad\quad\;\; \textbf{in let } s = head\ q + p \\
\qquad\quad\;\; \textbf{in let } r = [p, head\ q, s] \\
\qquad\quad\;\;\; \textbf{in } (q, r) \\
\textbf{in let } r = \textbf{case } rec \textbf{ of } (\_, r) \rightarrow r \\
\quad \textbf{in } sum\ r
\end{array}
$$

Finding Good Feedback Sets

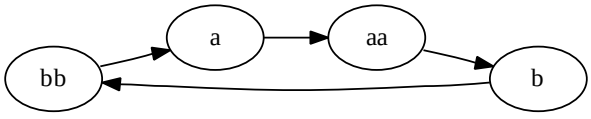
Unfortunately, finding a minimal feedback vertex set is an NP-hard problem. But in addition to the size of feedback sets there are other criterions that influence the efficiency of the transformed program.

If an identifier that is not part of the feedback set is used in the body of the declaration, then it has to be declared twice: once on the right hand side and once in the body of the transformed declaration. Therefore, identifiers that the body of the declaration refers to should be selected for feedback sets with higher prioritiy than others.

The following example demonstrates that the efficiency of this transformation also depends on whether the defined values refer to each other in a strict or non-strict way:[3]

$$
\begin{array}{l}
conditional\_example\ p \\
\quad = \textbf{let } a \;\; = \textbf{if } p \textbf{ then } bb \textbf{ else } 3 \\
\qquad\;\; b \;\; = \textbf{if } p \textbf{ then } 4 \textbf{ else } aa \\
\qquad\;\; aa = a + 5 \\
\qquad\;\; bb = b + 6 \\
\qquad \textbf{in } a + b
\end{array}
$$

The dependency graph of this declaration forms a circle; without knowing the value of p it is not possible to decide whether the value of a depends on the value of bb or whether the value of b depends on the value of a. Thus, the dependency graph of this declaration does not give any hint which identifiers to choose for the feedback set:



The only observable difference between the identifiers is that a and b are referred to by the body of the declaration. Therefore, a reasonable heuristics

[3]This example is taken from [Traub, 1991]

that decides only on the basis of the dependency graph will select either $\{a\}$ or $\{b\}$ as a feedback set. If it decides in favor of $\{a\}$, then the expression is translated to the following code:[4]

```
conditional_example p
  = let a = let aa = a + 5
            in let b = if p then 4 else aa
               in let bb = b + 6
                  in if p then bb else 3
    in let aa = a + 5
       in let b = if p then 4 else aa
          in a + b
```

Independent from the value of p, the value of a is always needed, so that the relatively big recursive declaration of a will always be evaluated. But when selecting $\{aa\}$ as the feedback set, the expression may be transformed to the following code:

```
conditional_example p
  = let aa = let b = if p then 4 else aa
             in let bb = b + 6
                in let a = if p then bb else 3
                   in a + 5
    in let b = if p then 4 else aa
       in let bb = b + 6
          in let a = if p then bb else 3
             in a + b
```

Now it depends on p whether the big recursive declaration of aa is evaluated: it is only needed when p has the value *True*. Thus, the second version is more efficient when p has the value *False*.

The example shows that it can be a bad choice to select identifiers that are strictly referred to by other declarations as members of feedback sets; knowing the strictness of the right hand sides may lead to more efficient translations.

Finding Feedback Vertex Sets: Algorithm

Deriving strictness information from programs or implementing sophisticated graph algorithms is beyond the scope of this work, so we restrict ourselves to a simple heuristics. We start by selecting one identifier that has the highest priority with respect to the following list of criterions:

1. If the right hand side of a declaration depends recursively on itself, then it is added to the feedback set with highest priority. It will be

[4]The results for feedback set $\{b\}$ are symmetrical.

added to the feedback set in any case; by adding it early it gets the chance to break other circles before they cause other identifers to be added to the feedback set.

2. If a vertex is not part of any circle, then it will never be selected as a feedback vertex. If the corresponding identifier is referred to by the body of the declaration, then this choice will lead to unnecessary code duplication.

   Another option would be to add identifiers that are referred to by the body to the feedback set, regardless of whether they are part of any circles. It has to be tested which option leads to more efficient programs.

3. From the remaining declarations the one with the highest number of references to other identifiers is selected. This is the heuristic part of the algorithm: The more references a declaration has, the more circles it can be expected to be a part of.

4. If two vertices have the same number of references, then the ones that are used by the body of the declaration have priority over the others: When we don't know how to minimize the feedback set further, we try to minimize code duplication.

After an identifier is selected, the corresponding vertex is removed from the dependdency graph. If the selected identifer is referred to by the body of the declaration, then the declaration is added to the body. After that we continue to select the next vertex from the remaining graph.

An implementation of this algorithm can be found in the reference implementation in Appendix B.1 on page 96.

2.5 Summary

Recursive declarations have to be avoided, because they cannot be evaluated in strict order. There are two exceptions from this rule: (mutually) recursive declarations whose right hand sides are already in weak head normal form, and recursive declarations that refer to themselves in a non-strict way.

Before a program can be debugged by Lazy Call-by-Value evaluation, all declarations have to be transformed into non-recursive declarations and — possibly mutually recursive — function declarations and constructor terms.

With the technique described above, mutually recursive declarations can be transformed into single recursive ones. Then the recursion can be eliminated, for example by the use of a least fixed point combinator.

Chapter 3

Natural Semantics of Lazy Call-by-Value Evaluation

In this chapter a natural semantics for Lazy Call-by-Value evaluation as well as for the non-strict calculation of oracles is developed. On the basis of John Launchbury's natural semantics for lazy evaluation it is discussed which pieces of information have to be collected in oracles. Then two variants of the semantics for lazy evaluation are presented — one for the calculation of oracles and one for oracle-directed strict evaluation —, and it is proved that the latter will always be able to calculate the same result as the former.

3.1 A Natural Semantics for Lazy Evaluation

In [Launchbury, 1993] a natural semantics for lazy evaluation is presented. It provides an accurate model for the sharing of program terms, so that it is well suited to explain the actual run-time behaviour of lazy functional languages. In [Sestoft, 1997], this semantics is modified in order to make it more suitable for mechanical evaluation. This modified semantics will serve as a reference which assures that the semantics for oracle creation models lazy evaluation properly.

Identifiers. There are two disjoint sets of identifiers: *Program variables* may occur in expressions provided by the user, whereas *heap pointers* are only introduced by the evaluation machinery. The set of program variables is denoted by $\mathbb{X}$; the set of heap pointers is denoted by $\mathbb{V}$. At this time no assumptions about the sets $\mathbb{X}$ and $\mathbb{V}$ are made except that they are disjoint and contain an infinite number of elements.

Expressions. The set $\mathbb{E}$ of expressions is inductively defined, so that an *expression* is

- an identifier $v \in \mathbb{X} \cup \mathbb{V}$,
- an abstraction $\lambda x.e$, where $x \in \mathbb{X}$ and $e \in \mathbb{E}$,
- an application $e\,v$, where $e \in \mathbb{E}$ and $v \in \mathbb{X} \cup \mathbb{V}$,
- a potentially mutually recursive declaration $\textbf{let } \overline{x_n = e_n} \textbf{ in } e$, where $e \in \mathbb{E}$ and $x_i \in \mathbb{X}$, $e_i \in \mathbb{E}$ for every $i \in \{1, \ldots, n\}$,
- a constructor term $C\,v_1 \ldots v_n$, where C is a constructor name, $n \in \mathbb{N}$ is the arity of the constructor, and $v_1, \ldots, v_n \in \mathbb{X} \cup \mathbb{V}$, or
- a case expression $\textbf{case } e \textbf{ of } \overline{C_n\,\overline{x_{n,k_n}} \mapsto a_n}$ with $a_i \in \mathbb{E}$, different constructor names $C_1, \ldots, C_n$ and $x_{i,1}, \ldots, x_{i,k_i} \in \mathbb{X}$ for every $i \in \{1, \ldots, n\}$.[1]

We will use parentheses to disambiguate between the abstraction $\lambda x.e\,v$, which is equivalent to $\lambda x.(e\,v)$, and the application $(\lambda x.e)\,v$.[2],

Here and in the following, we abbreviate sequences $\alpha_1, \ldots, \alpha_n$ by the notation $\overline{\alpha_n}$.

Weak head normal forms. An expression is said to be in *weak head normal form* iff it is either a lambda abstraction or a constructor term. Whether an expression e is in weak head normal form is denoted by the predicate $\mathrm{whnf}(e)$.

Used heap pointers. The function $\mathcal{V} : \mathbb{E} \to 2^{\mathbb{V}}$ indicates, which heap pointers occur in an expression. It is defined by:

$$
\begin{array}{ll}
\mathcal{V}(v) & \equiv \begin{cases} \{v\} & \text{if } v \in \mathbb{V} \\ \emptyset & \text{if } v \in \mathbb{X} \end{cases} \\
\mathcal{V}(\lambda x.e) & \equiv \mathcal{V}(e) \\
\mathcal{V}(e\,v) & \equiv \mathcal{V}(e) \cup \mathcal{V}(v) \\
\mathcal{V}(\textbf{let } \overline{x_n = e_n} \textbf{ in } e) & \equiv \mathcal{V}(e) \cup \bigcup_{i=1}^{n} \mathcal{V}(e_i) \\
\mathcal{V}(C\,v_1 \ldots v_n) & \equiv \bigcup_{i=1}^{n} \mathcal{V}(v_i) \\
\mathcal{V}(\textbf{case } e \textbf{ of } \overline{C_n\,\overline{x_{n,k_n}} \mapsto a_n}) & \equiv \mathcal{V}(e) \cup \bigcup_{i=1}^{n} \mathcal{V}(a_i)
\end{array}
$$

[1] The constructor names must be different in order to avoid non-determinism; if two patterns would have the same constructor name and the same number of arguments, then it would be unclear which of them has to be selected.

[2] The semantics will rule out the application of constructor terms, so we do not need to disambiguate between a constructor term $C\,v_1 \ldots v_{n+1}$ and an application $(C\,v_1 \ldots v_n)\,v_{n+1}$

Substitution. As in [Sestoft, 1997], the substitution $e[v/x]$ of a heap pointer v for all free occurrences of a program identifier x in an expression e is defined as follows:

$$
\begin{aligned}
y[v/x] &\equiv \begin{cases} v & \text{if } x = y \\ y & \text{otherwise} \end{cases} \quad \text{for any identifier } y \in \mathbb{X} \cup \mathbb{V} \\
(\lambda y.e)[v/x] &\equiv \begin{cases} \lambda x.e & \text{if } x = y \\ \lambda y.e[v/x] & \text{otherwise} \end{cases} \\
(e\ w)[v/x] &\equiv e[v/x]\ w[v/x] \\
(\mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e)[v/x] &\equiv \begin{cases} \mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e & \text{if } \exists i :\ x = x_i \\ \mathbf{let}\ \overline{x_n = e_n[v/x]}\ \mathbf{in}\ e[v/x] & \text{otherwise} \end{cases} \\
C\ v_1 \ldots v_n &\equiv C\ v_1[v/x] \ldots v_n[v/x] \\
(\mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n})[v/x] & \\
&\equiv \mathbf{case}\ e[v/x]\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a'_n} \\
\text{where } a'_i &= \begin{cases} a_i & \text{if } \exists j :\ x = x_{i,j} \\ a_i[v/x] & \text{otherwise} \end{cases} \quad \text{for every } i \in \{1, \ldots, n\}
\end{aligned}
$$

We abbreviate the simultaneous substitution $e[v_1/x_1][v_2/x_2] \ldots [v_n/x_n]$ of distinct program identifiers $x_1, \ldots, x_n$ with heap pointers $v_1, \ldots, v_n$ by writing $e\overline{[v_n/x_n]}$.

Heaps. A *heap* is a mapping from heap pointers to expressions. The set of heaps is defined as the set of partial functions from $\mathbb{V}$ to $\mathbb{E}$:

$$\mathbb{H} := \{\Gamma \mid \Gamma \subseteq \mathbb{V} \to \mathbb{E}\}$$

Configurations. A *configuration* is a tuple $(\Gamma,\ e)$ that consists of a heap $\Gamma \in \mathbb{H}$ and an expression $e \in \mathbb{E}$. The heap identifiers that occur in the expression e are interpreted as pointers to values bound on heap Γ.[3]

Judgments. For every $A \subseteq \mathbb{V}$ the relation $\Downarrow_A \subseteq (\mathbb{H} \times \mathbb{E}) \times (\mathbb{H} \times \mathbb{E})$ is defined in terms of the inference rules given in Figure 3.1 on the following page; the *judgment*

$$(\Gamma,\ e)\ \Downarrow_A\ (\Delta,\ e')$$

says that evaluating expression e in a heap Γ will yield the expression e', together with a heap Δ.

The parameter A is only used to decide whether a variable name is fresh in the current context; if an identifier $v \in \mathbb{V}$ occurs in the parameter A of some judgment, then v is considered as non-fresh in the derivation tree of that judgment.

[3] Allowing heap pointers to occur on the left hand sides of declarations, a configuration $(\{\overline{(v_n, e_n)}\})$ can be seen as equivalent to an expression $\mathbf{let}\ \overline{v_n = e_n}\ \mathbf{in}\ e$.

$$\textit{Const} \qquad \frac{\text{whnf}(e)}{(\Gamma,\, e) \Downarrow_A (\Gamma,\, e)}$$

$$\textit{App} \qquad \frac{(\Gamma,\, f) \Downarrow_A (\Delta,\, \lambda x.e) \;\wedge\; (\Delta,\, e[v/x]) \Downarrow_A R}{(\Gamma,\, f\; v) \Downarrow_A R}$$

$$\textit{Var} \qquad \frac{(v,\, e) \in \Gamma \;\wedge\; (\Gamma \setminus \{(v,\, e)\},\, e) \Downarrow_{A \cup \{v\}} (\Delta,\, e')}{(\Gamma,\, v) \Downarrow_A (\Delta \cup \{(v,\, f)\},\, e')}$$

$$\textit{Case} \qquad \frac{\begin{array}{c} i \in \{1, \ldots, n\} \;\wedge \\ (\Gamma,\, e) \Downarrow_A (\Delta,\, C_i\, \overline{v_{k_i}}) \;\wedge\; (\Delta,\, a_i[\overline{v_{k_i}/x_{i,k_i}}]) \Downarrow_A R \end{array}}{(\Gamma,\, \textbf{case}\; e\; \textbf{of}\; \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}) \Downarrow_A R}$$

$$\textit{Let}(*) \qquad \frac{\begin{array}{c} \forall j \in \{1, \ldots, n\} :\; e'_j = e_j[\overline{v_n/x_n}] \\ (\Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e[\overline{v_n/x_n}]) \Downarrow_A R \end{array}}{(\Gamma,\, \textbf{let}\; \overline{x_n = e_n}\; \textbf{in}\; e) \Downarrow_A R}$$

(*) In rule Let, the variables $v1, \ldots, v_n$ must be fresh and distinct heap pointers: $v_i \in \mathbb{V} \setminus (A \cup \operatorname{dom}\Gamma)$ must hold for every $i \in \{1, \ldots, n\}$.

Figure 3.1: Natural semantics for lazy evaluation

To *evaluate an expression e to a configuration* $(\Delta,\, e')$ means to prove that

$$(\emptyset,\, e) \Downarrow_\emptyset (\Delta,\, e').$$

Semantic Rules for Lazy Evaluation

The *revised semantics* presented in [Sestoft, 1997] differs from the semantics of [Launchbury, 1993] in two respects:

- In Launchbury's semantics, fresh identifiers are introduced when an expression is copied from the heap, whereas in Sestoft's revised semantics a fresh identifier is introduced whenever a new binding is created on the heap.
- Sestoft's semantics also formalizes when a variable can be considered as *fresh*.

The rules for lazy evaluation shown in Figure 3.1 are taken from the *revised semantics for lazy evaluation* of [Sestoft, 1997], with rules for constructor terms and case-expressions added as suggested in [Launchbury, 1993]:

Const. This rule combines the rule *Lam* of the revised semantics — which is identical to rule *Lambda* of Launchbury's semantics — with rule *Constructors* from Launchbury's extended semantics. It states that a constant value, i.e. a value that is already in weak head normal form, is returned immediately without changing the heap.

App. This rule is identical to rule *Application* of Launchbury's semantics and rule *App* of Sestoft's semantics. The evaluation of an application $f\ v$ starts by evaluating function f in heap Γ to a head normal form, obtaining a modified heap Δ. This evaluation must return an abstraction as its result. In the body of that abstraction, the formal parameter is then replaced by the argument. The resulting expression is evaluated in the context of the modified heap, producing a configuration R, which will be returned as result.

Var. This rule follows Sestoft's revised semantics. In Launchbury's semantics this rule was the place to introduce new identifiers, but in Sestoft's semantics new identifiers are introduced when values are bound on the heap.

When an identifer v is evaluated, the corresponding value is fetched from heap Γ. Then the binding (v, e) is removed from Γ, so that self-referential computations will fail instead of looping forever. Now this value is evaluated in the heap $\Gamma \setminus \{(v, e)\}$, returning a configuration $(\Delta,\ e')$. Finally, the resulting value e' is bound in the resulting heap Δ, and e' together with heap $\Delta \cup \{v, e'\}$ is returned as result.

In the evaluation of e, the identifier v is added to set A in order to prohibit subcompotations from re-introducing the temporarily removed variable v into the heap.

Case. This rule is taken from Launchbury's extended semantics. The evaluation of an expression $\mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n}$ requires two subcomputations: First, the expression e is evaluated to a constructor term $C\ v_1 \ldots v_n$. Then a pattern $C_i\ x_1 \ldots x_n$ is selected from the list of alternatives that matches the resulting constructor term. Then the identifiers $x_1 \ldots x_n$ are replaced by $v_1 \ldots v_n$ in the right hand side of this alternative, and finally the resulting expression is evaluated to a configuration that is returned as the overall result.

Let. This rule follows Sestoft's revised semantics. When a mutually recursive binding $\mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e$ is evaluated, the identifiers $x_1 \ldots x_n$ are replaced by fresh and distinct heap pointers $v_1, \ldots, v_n$ in the expressions $e_1, \ldots, e_n$ as well as in the body expression e. Then the body expression is evaluated in a heap that consists of the original heap together with the renamed expressions $e_1, \ldots, e_n$ bound to the identifiers $v_1, \ldots, v_n$.

An identifier is considered as fresh if it does not occur in A and no value is bound to it in the current heap.

3.2 What Oracles Must Contain

An oracle contains information about which redexes are needed and which redexes have to be skipped. It controls the runtime behavior by deciding whether the next redex will be evaluated or replaced by some placeholder value. This costs computing resources at run-time, so it is preferable to reduce the number of oracle entries to a minimum.

An inspection of the semantics presented in [Braßel et al., 2007] reveals some inefficiencies: When a case expression **case** e **of**... is evaluated, an oracle entry for expression e is created, indicating whether the value of e is needed. But at this point it is already clear that e must be evaluated in order to select an alternative. This inefficiency comes from their choice to create oracle entries for every redex.

In this work we follow another approach by asking: *If the value of an expression is needed, then which of its subexpressions have to be evaluated?*

If it is known that the value of a subexpression is always required, then its evaluation can always be forced; only subexpressions that are not always needed have to be controlled by the oracle.

Identifiers. Whether the value bound to a heap pointer is needed or not is controlled at the place where it is introduced by a **let**-declaration; keeping track of the use of heap pointers would duplicate this work. Therefore, no oracle entries are needed for identifers.

Abstractions. The subexpression e of an abstraction $\lambda x.e$ is not evaluated at all; when the abstraction is applied to an argument, a fresh copy of e is evaluated in place of the application. Thus, there are no subexpressions whose evaluation needs to be supervised by the oracle.

Applications. When an application $f\ v$ is evaluated, the value of function f is always needed. The argument v may be needed or not, depending on f, but the evaluation of the expression bound to v is controlled at the place where the heap pointer v is introduced. Thus, there are no optional steps in the evaluation of abstractions; no oracle entries are needed for it.

Constructor terms. All components of a constructor term are identifiers, whose evaluation is already tracked at the place where they are defined. Therefore, no oracle entries are needed to control the evaluation of constructor terms.

Case expressions. In an expression **case** e **of** $\overline{C_n\,\overline{x_{n,k_n}} \mapsto e_n}$ there are no optional evaluation steps: First e has to be evaluated, so that an

alternative i can be chosen. Then only the ith right hand side e_i has to be evaluated. Thus, there are no optional evaluation steps and no oracle entries are needed to control the evaluation.

Let bindings. Four cases have to be distinguished:

1. An expression **let** $\overline{x_n = e_n}$ **in** e where all right hand sides e_i, $i \in \{1, \ldots, n\}$, are in weak head normal form. Then the only subexpression that has to be evaluated is the body expression e. But the value of e is needed in any case, so there are no optional evaluation steps.
2. An expression **let** $x = y$ **in** e, where y is an identifier with $x \neq y$. Since y is an identifier, there are no oracle entries needed to control the evaluation.[4]
3. An expression **let** $x = e_1$ **in** e_2 where e_1 does not refer to x, is not an identifier and not in weak head normal form. The value bound to x may be needed or not in the evaluation of e_2, so an oracle entry is needed to control whether e_1 has to be evaluated. The value of e_2 is alway needed, so no further oracle entry is required.
4. As discussed in Chapter 2, all other declarations have to be transformed to one of the former cases.

These considerations lead to the following strategy:

- If the right hand side of a single non-recursive declaration is not in weak head normal form, then it needs to be controlled by an oracle entry.
- Declarations that define only constants do not need to be controlled by the oracle.
- Other declarations have to be transformed to one of the former types.

3.3 Encoding the Program Structure in Identifiers

We develop the semantics for non-strict oracle creation in two steps. In the first step we specify the way new identifiers are invented. Then in Section 3.4 we add the facility to create oracles to the semantics.

We fix heap pointers to be sequences of letters from an infinite alphabet $\mathbb{A}$:

$$\mathbb{V} := \mathbb{A}^*$$

[4] Another justification is that this expression is equivalent to the expression $((\lambda x.e)\ y)$, for which no oracle entries are needed.

$$\textit{Const} \qquad \frac{\mathrm{whnf}(e)}{(I,\,\Gamma,\,e)\ \Downarrow_p\ (I,\,\Gamma,\,e)}$$

$$\textit{App} \qquad \frac{(I,\,\Gamma,\,f)\ \Downarrow_p\ (J,\,\Delta,\,\lambda x.e)\ \wedge\ (J,\,\Delta,\,e[v/x])\ \Downarrow_p\ R}{(I,\,\Gamma,\,f\ v)\ \Downarrow_p\ R}$$

$$\textit{Var} \qquad \frac{(v,\,e) \in \Gamma\ \wedge\ (\emptyset,\,\Gamma \setminus \{(v,\,e)\},\,e)\ \Downarrow_v\ (J,\,\Delta,\,e')}{(I,\,\Gamma,\,v)\ \Downarrow_p\ (I,\,\Delta \cup \{(v,\,e')\},\,e')}$$

$$\textit{Case} \qquad \frac{\begin{array}{c} i \in \{1,\ldots,n\}\ \wedge \\ (I,\,\Gamma,\,e)\ \Downarrow_p\ (J,\,\Delta,\,C_i\,\overline{v_{k_i}})\ \wedge\ (J,\,\Delta,\,a_i\overline{[v_n/x_n]})\ \Downarrow_p\ R \end{array}}{(I,\,\Gamma,\,\mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\,\overline{x_{n,k_n}} \mapsto a_n})\ \Downarrow_p\ R}$$

$$\textit{Let}(*) \qquad \frac{\begin{array}{c} \forall j \in \{1,\ldots,n\} :\ e'_j = e_j\overline{[p \cdot v_n/x_n]} \\ (I \cup \{\overline{v_n}\},\,\Gamma \cup \{\overline{(p \cdot v_n,\,e'_n)}\},\,e\overline{[v_n/x_n]})\ \Downarrow_p\ R \end{array}}{(I,\,\Gamma,\,\mathbf{let}\ \overline{x_n = e_n}\ \mathbf{in}\ e)\ \Downarrow_p\ R}$$

(*) In rule Let, the variables $v1,\ldots,v_n$ must be distinct elements of $\mathbb{A}$ that do not occur in I.

Figure 3.2: Modified natural semantics for lazy evaluation

In contrast to the semantics of lazy evaluation, a configuration is a 3-tuple $(I,\,\Gamma,\,e)$. Like before, it contains a heap Γ and an expression e. As an additional element, it contains a set $I \subseteq \mathbb{A}$ of the letters that already occur as suffixes of heap pointers.

The relation $\Downarrow_A$ is replaced by a relation $\Downarrow_p$ that takes a sequence $p \in \mathbb{A}^*$ as its parameter. It is defined in terms of the inference rules given in Figure 3.2. Parameter p is called the *thunk prefix*. When a new identifiers is needed in the evaluation of a judgment $(I,\,\Gamma,\,e)\ \Downarrow_p\ (J,\,\Delta,\,e')$, it is composed of the thunk prefix p and an element of $\mathbb{A}$ that is not an element of I. To evaluate an expression e to a configuration $(I,\,\Delta,\,e')$ means to prove that

$$(\emptyset,\,\emptyset,\,e)\ \Downarrow_\epsilon\ (I,\,\Delta,\,e').$$

As an example, consider the following expression:

$$\begin{array}{l} \mathbf{let}\ p = \mathbf{let}\ q = \mathbf{let}\ r = \mathbf{case}\ p\ \mathbf{of}\ \mathit{Cons}\ a\ b \to b \\ \qquad\qquad\qquad\quad \mathbf{in}\ \mathit{Cons}\ r\ s \\ \qquad\qquad\quad s = \mathit{True} \\ \qquad\qquad \mathbf{in}\ \mathit{Cons}\ q\ s \\ \mathbf{in\ case}\ p\ \mathbf{of}\ \mathit{Cons}\ a\ b \to a \end{array}$$

$$
\begin{array}{l}
\mathrm{Let}_{\epsilon}\Big[\ (\emptyset,\ \emptyset,\ \textbf{let}\ ...\ \textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to a) \\
\quad \mathrm{Case}_{\epsilon}\Big[\ (\{\tilde{p}\},\ P,\ \textbf{case}\ \tilde{p}\ \textbf{of}\ \mathit{Cons}\ a\ b \to a) \\
\qquad \mathrm{Var}_{\epsilon}\Big[\ (\{\tilde{p}\},\ P,\ \tilde{p}) \\
\qquad\quad \mathrm{Let}_{\tilde{p}}\Big[\ (\emptyset,\ \emptyset,\ \textbf{let}\ ...\ \textbf{in}\ \mathit{Cons}\ q\ s) \\
\qquad\qquad \mathrm{Const}_{\tilde{p}}\big[\ (\{\tilde{q},\tilde{s}\},\ Q\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\ \tilde{p}\tilde{s}) \\
\qquad\qquad (\{\tilde{q},\tilde{s}\},\ Q\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\ \tilde{p}\tilde{s}) \\
\qquad (\{\tilde{p}\},\ Q\ \cup\ P'\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\ \tilde{p}\tilde{s}) \\
\qquad (\{\tilde{p}\},\ Q\ \cup\ P'\ \cup\ S,\ \tilde{p}\tilde{q}) \\
\qquad \mathrm{Var}_{\epsilon}\Big[\ \mathrm{Let}_{\tilde{p}\tilde{q}}\Big[\ (\emptyset,\ P'\ \cup\ S,\ \textbf{let}\ ...\ \textbf{in}\ \mathit{Cons}\ r\ \tilde{p}\tilde{s}) \\
\qquad\qquad \mathrm{Const}_{\tilde{p}\tilde{q}}\big[\ (\{\tilde{r}\},\ P'\cup R\cup S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s}) \\
\qquad\qquad (\{\tilde{r}\},\ P'\cup R\cup S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s}) \\
\qquad (\{\tilde{p}\},\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s}) \\
\quad (\{\tilde{p}\},\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s}) \\
(\{\tilde{p}\},\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s})
\end{array}
$$

$$
\begin{array}{ll}
P = \{(\tilde{p}, \textbf{let}\ ...\ \textbf{in}\ \mathit{Cons}\ q\ s)\} & P' = \{(\tilde{p}, \mathit{Cons}\ \tilde{p}\tilde{q}\ \tilde{p}\tilde{s})\} \\
Q = \{(\tilde{p}\tilde{q}, \textbf{let}\ ...\ \textbf{in}\ \mathit{Cons}\ r\ s)\} & Q' = \{(\tilde{p}\tilde{q}, \mathit{Cons}\ \tilde{p}\tilde{q}\tilde{r}\ \tilde{p}\tilde{s})\} \\
R = \{(\tilde{p}\tilde{q}\tilde{r}, \textbf{case}\ \tilde{p}\ \textbf{of}\ \mathit{Cons}\ a\ b \to b)\} & \\
S = \{(\tilde{p}\tilde{s}, \mathit{True})\} &
\end{array}
$$

Figure 3.3: Example of lazy evaluation

For the purpose of demonstration, we fix set $\mathbb{A}$:

$$\mathbb{A} := \{\tilde{x} \mid x \in \mathbb{X}\}$$

Figure 3.3 shows the evaluation of this expression. It starts with the initial configuration $(\emptyset,\ \emptyset,\ \textbf{let}\ ...\ \textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to a)$. First, rule *Let* is applied to it. A new heap pointer $\tilde{p}$ is invented and substituted for p in the whole expression. Then the right hand side **let** ... **in** *Cons* q s is bound to $\tilde{p}$, and $\tilde{p}$ is added to the set of used letters.

Then as a subcompution the body expression **case** $\tilde{p}$ **of** *Cons* a $b \to a$ is evaluated by rule *Case*. This in turn has two subcomputations: The first one evaluates the expression $\tilde{p}$, the second one evaluates the selected alternative.

Both subcomputations are applications of rule *Var*. The first one fetches the expression bound to $\tilde{p}$ from the heap and evaluates it by applying rule *Let*. In this evaluation, the binding of $\tilde{p}$ is removed from the heap, the thunk prefix consists of the variable name $\tilde{p}$, and the set of used letters is set to $\emptyset$.

A new heap pointer has to be invented by appending an unused letter to the thunk prefix. We choose the letter $\tilde{pq}$ in order to keep the result similar to the initial expression, yielding a new heap pointer $\tilde{pq}$. Then this pointer is substituted for q in the right hand sides as well as in the body of the declaration. In the same way, a new heap pointer $\tilde{ps}$ is invented and substituted for s, and the right hand sides are bound to $\tilde{pq}$ and $\tilde{ps}$ on the heap.

Now the body expression $Cons\ \tilde{pq}\ \tilde{ps}$ is evaluated by rule *Const*, which simply returns the unchanged configuration as its result, so that it is also returned by rule *Let*. Rule *Var* restores the values of the set of used letters and binds the resulting expression of its subcomputation to $\tilde{p}$.

After that, one alternative of the case expression is selected, the identifiers a and b are substituted with $\tilde{pq}$ and $\tilde{ps}$ in the right hand side of the alternative, and the resulting expression $\tilde{pq}$ is evaluated, again by an application of rule *Var*.

As result, the evaluation returns the expression $Cons\ \tilde{pqr}\ \tilde{ps}$, together with the following heap:

$$\begin{array}{ll}\{(\tilde{p}, & Cons\ \tilde{pq}\ \tilde{ps}),\\ (\tilde{pq}, & Cons\ \tilde{pqr}\ \tilde{ps}),\\ (\tilde{pqr}, & \textbf{case}\ \tilde{p}\ \textbf{of}\ Cons\ a\ b \to b),\\ (\tilde{ps}, & True)\}\end{array}$$

By grouping heap entries whose names have a common prefix, an expression that resembles the structure of the original expression can easily be reconstructed:[5]

$$\begin{array}{l}\textbf{let}\ p = \textbf{let}\ q = \textbf{let}\ r = \textbf{case}\ \tilde{p}\ \textbf{of}\ Cons\ a\ b \to b\\ \qquad\qquad\qquad\quad \textbf{in}\ Cons\ (\tilde{pqr})\ (\tilde{ps})\\ \qquad\qquad\quad s = True\\ \qquad\qquad \textbf{in}\ Cons\ (\tilde{pq})\ (\tilde{ps})\\ \textbf{in}\ Cons\ (\tilde{pqr})\ (\tilde{ps})\end{array}$$

3.4 Language $\lambda^{?!}$

The considerations in Section 3.2 have shown that strict declarations of functions and constants have to be evaluated in a different way than non-strict value declarations. Therefore, we modify the expression syntax, so that declarations of the form

$$\textbf{let}\ \overline{x_n = e_n}\ \textbf{in}\ e$$

[5] This reconstructed expression contains heap ointers. Therefore, it cannot be fed back into the interpreter; for that the invention and substitution of identifiers would have to be defined in a slightly different way.

are replaced by two more restricted forms of declarations:

Strict declarations. A mutually recursive let binding $\mathbf{let}^{!}\ \overline{x_n = e_n}\ \mathbf{in}\ e$, where all $e_1, \ldots, e_n$ are in weak head normal form.

Non-strict declarations A non-strict, non-recursive let binding $\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2$.[6]

This modification does not reduce expressiveness: As shown in Chapter 2, every declaration $\mathbf{let}\,\overline{x_n = e_n}\,\mathbf{in}\,e$ can be expressed by the newly introduced strict and non-strict declarations.

The set $\mathbb{V}$ of heap pointers is now defined as the union of two disjoint sets:

- The set $\mathbb{N}^*$ of *thunk identifiers*. A thunk identifier is a finite sequence of natural numbers. It refers to an unevaluated thunk or to the result of its evaluation.
- The set $\mathbb{V}^C$ of *constant identifiers*. They may only refer to constant values, i. e. to constructor terms or abstractions. A constant identifier $x_{t:c}$ consists of a program identifier $x \in \mathbb{X}$, a thunk identifier $t \in \mathbb{N}^*$ and an additional index $c \in \mathbb{N}$. The set $\mathbb{V}^C$ is defined by

$$\mathbb{V}^C := \{x_{t:c} \mid t \in \mathbb{N}^*,\, c \in \mathbb{N}\}.$$

As before, the sets $\mathbb{X}$ and $\mathbb{V}$ are assumed to be disjoint.

The resulting set of expressions is called $\mathbb{E}$, and the resulting language is referred to as $\lambda^{?!}$. Figure 3.4 on the following page summarizes language $\lambda^{?!}$.

Substitution. Additional rules for substitution and for function $\mathcal{V}$ have to be provided for the newly introduced expressions:

$$(\mathbf{let}^{?}\ y = e_1\ \mathbf{in}\ e_2)[v/x] \equiv \begin{cases} \mathbf{let}^{?}\ y = e_1\ \mathbf{in}\ e_2[v/x] & \text{if } x = y \\ \mathbf{let}^{?}\ y = e_1[v/x]\ \mathbf{in}\ e_2[v/x] & \text{otherwise} \end{cases}$$

$$(\mathbf{let}^{!}\ \overline{x_n = e_n}\ \mathbf{in}\ e)[v/x] \equiv \begin{cases} \mathbf{let}^{!}\ \overline{x_n = e_n}\ \mathbf{in}\ e & \text{if } \exists i:\ x = x_i \\ \mathbf{let}^{!}\ \overline{x_n = e_n[v/x]}\ \mathbf{in}\ e[v/x] & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathcal{V}(\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2) &\equiv \mathcal{V}(e_1) \cup \mathcal{V}(e_2) \\ \mathcal{V}(\mathbf{let}^{!}\ \overline{x_n = e_n}\ \mathbf{in}\ e) &\equiv \mathcal{V}(e) \cup \textstyle\bigcup_{i=1}^{n} \mathcal{V}(e_i) \end{aligned}$$

[6] Since non-strict declarations are non-recursive, replacing an expression $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ naïvely with $\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2$ may lead to errors if x occurs free in e_1. Instead, the recursive declaration has to be transformed into a function declaration or into a non-recursive data declaration, for example via the techniques suggested in Section 2.4

Identifiers

Program identifiers	$\mathbb{X}$
Heap pointers	$\mathbb{V} := \mathbb{N}^* \cup \mathbb{V}^C$
Constant identifiers	$\mathbb{V}^C := \{x_{t:c} \mid t \in \mathbb{N}^*, c \in \mathbb{N}\}$.
Thunk identifiers	$\mathbb{N}^*$

$\mathbb{X}$ and $\mathbb{V}$ are pairwise disjoint

Expressions

Variable	x
Abstraction	$\lambda x.e$
Application	$e\ v$
Strict declaration*	$\mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e$
Non-strict declaration	$\mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2$
Constructor term	$C\ v_1 \ldots v_n$
Case expression	$\mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto e_n}$

(*) where all $e_1 \ldots, e_n$ are in weak head normal form

Figure 3.4: Syntax of language $\lambda^{?!}$

We call an expression that may be provided by the user an *initial expression*. The only required property of initial expressions is that they do not contain heap pointers. Otherwise these heap pointers might collide with heap pointers that are introduced by the evaluation machinery.

Definition (Initial expression)**.** *An expression $e \in \mathbb{E}$ is called an* initial expression*, iff $\mathcal{V}(e) = \emptyset$.*

3.5 Non-strict Oracle Creation

Oracles. An *oracle* is a set of thunk pointers. It contains pointers to the thunks whose values were needed in an evaluation.

Configurations. In contrast to the last section, a configuration is now a 4-tuple $(t, c, \Gamma, e) \in \mathbb{N} \times \mathbb{N} \times \mathbb{H} \times \mathbb{E}$. Like before, it contains a heap Γ and an expression e. The set that previously indicated the used variable names is now replaced by two counters: The *thunk counter* t is used to create unique thunk identifiers, and the *constant counter* c is used to create unique constant identifiers. This way the non-deterministic choice of variable names is replaced by a *completely deterministic* mechanism.

Judgments. Like before, a judgment $L \Downarrow_{p,\gamma} R$ relates a configuration L to the result of its evaluation R. It is parameterized with a *thunk prefix* $p \in \mathbb{N}^*$ and an oracle $\gamma \subseteq \mathbb{N}^*$. In the derivation tree a judgment inherits the thunk prefix from its parent judgments. Together with the thunk counter of the left configuration the thunk prefix is used to create unique thunk identifiers. The oracle is synthesized from the child judgments. It collects the names of the thunks that were needed in the evaluation.

To evaluate an expression e to a configuration (t', c', Δ, e') means to prove that

$$(0, 0, \emptyset, e) \Downarrow_{\epsilon,\omega} (t', c', \Delta, e')$$

for a set $\omega \subseteq \mathbb{N}^*$. Then the set ω is the *oracle* that can be used to direct the Lazy Call-by-Value evaluation of e.

For a configuration $C = (t, c, \Gamma, e)$, we write $\mathcal{H}(C)$ for the heap Γ and $\mathcal{E}(C)$ for the expression e. Also, for a judgment $L \Downarrow_{p,\gamma} R$, we write $\mathcal{O}(L \Downarrow_{p,\gamma} R)$ for the oracle γ.

Figure 3.5 on the next page shows the rules for non-strict oracle creation. Due to the newly introduced distinction between strict and non-strict bindings, rules *Var* and *Let* of the semantics of lazy evaluation are each split into two rules. Rule *Var* is now split into the rules $\mathit{VarStrict}^L$ and $\mathit{VarLazy}^L$, whereas rule *Let* is split into the rules $\mathit{LetStrict}^L$ and $\mathit{LetLazy}^L$.

Const^L. Like the previos rule *Const*, this rule does not change the configuration. It returns the empty set as its oracle, because it does not evaluate any heap-bound thunks.

App^L, Case^L. In these rules both subcomputations may change the thunk counter. The resulting oracle is the union of the oracles of both subcomputations.

$\mathit{VarStrict}^L$. In contrast to the previous rule *Var*, the value fetched from the heap is known to be in weak head normal form. This value is simply returned; no other components of the configuration are changed, and no oracle entries are emitted.

$\mathit{VarLazy}^L$. Like the previous rule *Var*, this rule handles the evaluation of non-strict heap entries. In contrast to rule *Var*, it adds the name of the currently evaluated variable to the oracle.

The evaluation of the heap entry takes place with the thunk prefix set to the thunk pointer v and the thunk counter and constant counter set to 0; this way the heap pointers created in the evaluation appear next to v when the heap entries are sorted in lexicographic order.

$\mathit{LetStrict}^L$. This rule binds constant values on the heap. It synthesizes new constant pointers from the current constant counter, heap counter and thunk prefix. Then it increments the constant couunter and enhances the

$$Const^L \quad \frac{\text{whnf}(e)}{(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\emptyset} (t,\, c,\, \Gamma,\, e)}$$

$$App^L \quad \frac{(t,\, c,\, \Gamma,\, f) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, \lambda x.e) \;\wedge\; (t',\, c',\, \Delta,\, e[v/x]) \Downarrow_{p,\delta} R}{(t,\, c,\, \Gamma,\, f\ v) \Downarrow_{p,\gamma\cup\delta} R}$$

$$VarStrict^L \quad \frac{v \in \mathbb{V}^C \;\wedge\; (v,\, e) \in \Gamma}{(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\emptyset} (t,\, c,\, \Gamma,\, e)}$$

$$VarLazy^L \quad \frac{v \in \mathbb{N}^* \;\wedge\; (v,\, e) \in \Gamma \;\wedge\; (0,\, 0,\, \Gamma \setminus \{(v,\, e)\},\, e) \Downarrow_{v,\gamma} (t',\, c',\, \Delta,\, e')}{(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\gamma\cup\{v\}} (t,\, c,\, \Delta \cup \{(v,\, e')\},\, e')}$$

$$Case^L \quad \frac{1 \leqslant i \leqslant n \;\wedge\; (t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, C_i\, \overline{v_{k_i}}) \;\wedge\; (t',\, c',\, \Delta,\, a_i\overline{[v_{k_i}/x_{i,k_i}]}) \Downarrow_{p,\delta} R}{(t,\, c,\, \Gamma,\, \textbf{case}\ e\ \textbf{of}\ \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}) \Downarrow_{p,\gamma\cup\delta} R}$$

$$LetStrict^L \quad \frac{\forall i \in \{1,\ldots,n\} :\; v_i = (x_i)_{p\cdot t:(c+1)} \;\wedge\; e_i' = e_i\overline{[v_n/x_n]} \;\wedge\; (t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e_n')}\},\, e\overline{[v_n/x_n]}) \Downarrow_{p,\gamma} R}{(t,\, c,\, \Gamma,\, \textbf{let}^!\ \overline{x_n = e_n}\ \textbf{in}\ e) \Downarrow_{p,\gamma} R}$$

$$LetLazy^L \quad \frac{(t+1,\, 0,\, (\Gamma \cup \{(p\cdot t,\, e_1)\},\, e_2[p\cdot t/x]) \Downarrow_{p,\gamma} R}{(t,\, c,\, \Gamma,\, \textbf{let}^?\ x = e_1\ \textbf{in}\ e_2) \Downarrow_{p,\gamma} R}$$

Figure 3.5: Rules for non-strict oracle creation

heap by binding the right hand sides to these constant pointers, allowing for recursive references like the previous rule *Let*. Finally, it evaluates the body of the declaration.

LetLazy^L. This rule handles non-strict declarations. Like rule *LetStrict*^{L}, it synthesizes a new unique name for the variable and binds the expression to that identifier on the heap. It substitutes all free occurrences of x with v in expression e_2 and evaluates the resulting expression to a configuration that is returned as the result of rule *LetLazy*^{L}.

The main difference to rule *LetStrict*^{L} is that the newly created identifier is a thunk pointer, and that the bound value cannot refer to itself.

The example from page 26 has to be translated to the following $\lambda^{?!}$

$$
\begin{array}{l}
\mathit{LetLazy}^{L}\ {}^{\epsilon}_{\{0,00\}} \Big[\ (0,\ 0,\ \emptyset,\ \textbf{let}^{?}\ \ldots\ \textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to a) \\
\qquad \mathit{Case}^{L}\ {}^{\epsilon}_{\{0,00\}} \Big[\ (1,\ 0,\ P,\ \textbf{case}\ 0\ \textbf{of}\ \mathit{Cons}\ a\ b \to a) \\
\qquad\qquad \mathit{VarLazy}^{L}\ {}^{\epsilon}_{\{0\}} \Big[\ (1,\ 0,\ P,\ 0) \\
\qquad\qquad\qquad \mathit{LetStrict}^{L}\ {}^{0}_{\emptyset} \Big[\ (0,\ 0,\ \emptyset,\ \textbf{let}^{!}\ \ldots\ \textbf{in let}^{?}\ \ldots\ \textbf{in}\ \mathit{Cons}\ q\ s) \\
\qquad\qquad\qquad\qquad \mathit{LetLazy}^{L}\ {}^{0}_{\emptyset} \Big[\ (0,\ 1,\ S,\ \textbf{let}^{?}\ \ldots\ \textbf{in}\ \mathit{Cons}\ q\ s) \\
\qquad\qquad\qquad\qquad\qquad \mathit{Const}^{L}{}_{0,\emptyset} \Big[\ (1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
\qquad\qquad\qquad\qquad\qquad\qquad (1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
\qquad\qquad\qquad\qquad (1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
\qquad\qquad\qquad (1,\ 0,\ Q\ \cup\ P'\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
\qquad\qquad \mathit{VarLazy}^{L}\ {}^{\epsilon}_{\{00\}} \Big[\ (1,\ 0,\ Q\ \cup\ P'\ \cup\ S,\ 00) \\
\qquad\qquad\qquad \mathit{LetLazy}^{L}\ {}^{00}_{\emptyset} \Big[\ (0,\ 0,\ P'\ \cup\ S,\ \textbf{let}^{?}\ \ldots\ \textbf{in}\ \mathit{Cons}\ r\ s_{00:1}) \\
\qquad\qquad\qquad\qquad \mathit{Const}^{L}{}_{00,\emptyset} \Big[\ (0,\ 0,\ P'\cup R\cup S,\ \mathit{Cons}\ 000\ s_{00:1}) \\
\qquad\qquad\qquad\qquad\qquad (0,\ 0,\ P'\cup R\cup S,\ \mathit{Cons}\ 000\ s_{00:1}) \\
\qquad\qquad\qquad (1,\ 0,\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1}) \\
\qquad\qquad (1,\ 0,\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1}) \\
\qquad (1,\ 0,\ P'\ \cup\ Q'\ \cup\ R\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1})
\end{array}
$$

$P = \{(0, \textbf{let}^{!}\ \ldots\ \textbf{in let}^{?}\ \ldots\ \textbf{in}\ \mathit{Cons}\ q\ s)\}$ $\quad P' = \{(0, \mathit{Cons}\ 00\ s_{00:1})\}$
$Q = \{(00, \textbf{let}^{?}\ \ldots\ \textbf{in}\ \mathit{Cons}\ r\ s)\}$ $\quad Q' = \{(00, \mathit{Cons}\ 000\ s_{00:1})\}$
$R = \{(000, \textbf{case}\ 0\ \textbf{of}\ \mathit{Cons}\ a\ b \to b)\}$
$S = \{(s_{00:1}, \mathit{True})\}$

Figure 3.6: Example of non-strict oracle creation

expression, before an oracle can be calculated for it:

$$
\begin{array}{l}
\textbf{let}^{?}\ p = \textbf{let}^{!}\ s = \mathit{True} \\
\qquad\qquad \textbf{in let}^{?}\ q = \textbf{let}^{?}\ r = \textbf{case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to b \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathit{Cons}\ r\ s \\
\qquad\qquad\quad \textbf{in}\ \mathit{Cons}\ q\ s \\
\textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to a
\end{array}
$$

Figure 3.6 shows the non-strict oracle creation for this expression. As its result it returns the expression $\mathit{Cons}\ 000\ s_{00:1}$ together with the the oracle set $\{0, 00\}$ and the following heap:

$$\{(0, \quad \mathit{Cons}\ 00\ s_{00:1}),$$

$$\begin{array}{ll}(00, & \mathit{Cons}\ 000\ s_{00:1}),\\ (000, & \textbf{case}\ 0\ \textbf{of}\ \mathit{Cons}\ a\ b \to b),\\ (s_{00:1}, & \mathit{True})\}\end{array}$$

In order to relate the subcomputations of non-strict oracle creation to the subcomputations of Lazy Call-by-Value evaluation, the notion of *current positions* is introduced. On these positions the order $\preccurlyeq$ is defined. In Observation 8 on page 41 we will see that the current position is monotonically increasing in the course of a Lazy Call-by-Value evaluation. For every thunk pointer p and every constant counter c the set $\mathbb{V}_{p:c}$ denotes those identifiers that Lazy Call-by-Value evaluation may have bound on the heap when the current position (p, c) is reached.

Definition. *The* current position *of a judgment* $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\omega} C$ *is the tuple* $(p \cdot t, c)$*. Analogously, for a judgment* $(t,\, c,\, \Gamma,\, e) \downarrow_{p,\omega} D$ *the current position is the tuple* $(p \cdot t, c)$*.*

Definition. *The relation* $\preccurlyeq$ *is the lexicographic order on* $\mathbb{N}^* \times \mathbb{N}$*, i.e. for every* $s, t \in \mathbb{N}^*$*,* $b, c \in \mathbb{N}$ *there is*

$$(s, b) \preccurlyeq (t, c) \ :\Longleftrightarrow\ s <_{\mathbb{N}^*} t \vee (s = t \wedge b \leqslant c)$$

where $<_{\mathbb{N}^*}$ *is the strict lexicographic order on* $\mathbb{N}^*$*.*

For every $t \in \mathbb{N}^*$*,* $c \in \mathbb{N}$ *the set* $\mathbb{V}_{t:c}$ *is defined by*

$$\mathbb{V}_{t:c} := \{x_{s:b} \in \mathbb{V}^C \mid (s, b) \preccurlyeq (t, c)\} \cup \{s \in \mathbb{N}^* \mid s <_{\mathbb{N}^*} t\}$$

For every identifier $v \in \mathbb{N}^*$ *there is* $\mathbb{V}_v := \mathbb{V}_{t:\infty}$*, and for every identifier* $w = x_{t:c} \in \mathbb{V}^C$ *there is* $\mathbb{V}_w := \mathbb{V}_{t:c}$*.*

It can be seen that the final thunk counter of a computation is always greater or equal to its initial thunk counter, and that the constant counter is only decreased when the thunk counter is increased:

Observation 1. *If the judgment* $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e')$ *holds, then*

1. $(p \cdot t, c) \preccurlyeq (p \cdot t', c')$*,*
2. *if* e *is a strict or non-strict declaration, then* $(p \cdot t, c) \neq (p \cdot t', c')$*, and*
3. $\mathbb{V}_{p \cdot t:c} \subseteq \mathbb{V}_{p \cdot t':c'}$*.*

Proof. The first observation can easily be verified by induction over the derivation of $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e')$.

Rule *LetStrict*^{L} as well as rule *LetLazy*^{L} advances the current position before its subcomputation. The first observation holds for this subcomputation, so the second observation follows.

The third observation follows directly from the first. □

Although lazy evaluation is a deterministic evaluation strategy, the semantics for lazy evaluation of Figure 3.1 as well as of Figure 3.2 are non-deterministic in the choice of variable names. In non-strict oracle creation, identifiers are created in a deterministic way; the result of a computation as well as the intermediate steps are determined by the initial configuration:

Observation 2.

1. *For every judgment* $P \Downarrow_{p,\gamma} Q$ *there is* at most one *derivation tree.*
2. *For every configuration* P *there is* at most one *configuration* Q*, such that* $P \Downarrow_{p,\gamma} Q$*.*

Proof. There is no configuration for which two different rules are applicable, and none of the rules allows for non-deterministic choice. □

We will prove properties of judgments by induction over the structure of their derivation trees. For this we define a partial order on judgments that resembles their causal dependency:

Definition (Derivation tree)**.** *A binary relation* $\Subset$ *over judgments is defined, such that for all judgments* I, J *the statement* $I \Subset J$ *holds iff the derivation tree of* I *is a strict subtree of the derivation tree of* J*. If* $I \Subset J$ *then* I *is called a* subcomputation *of* J*.*

For every judgment I*, the set* $\mathcal{D}(I)$ *is defined to be the set of* I*'s subcomputations:*

$$\mathcal{D}(I) := \{J \mid J \Subset I\}$$

For every initial expression a *with* $(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\omega} C$ *for some oracle* ω *and some configuration* C*, there is*

$$\mathcal{D}(a) := \mathcal{D}((0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\omega} C)$$

The partial order $(\mathcal{D}(I), \Subset)$ *is called the* derivation tree *of* I*. The dual order* $(\mathcal{D}(a), \Supset)$ *of* $(\mathcal{D}(I), \Subset)$ *is called the* reversed derivation tree *of* I*.*

The following observation formalizes the fact that the oracle of a judgment contains the oracle entries that were collected by the subcomputations of that judgment:

Observation 3. *For judgments* I*,* J *with* $J \Subset I$ *there is* $\mathcal{O}(J) \subseteq \mathcal{O}(I)$*.*

Proof. By induction over the reversed deriavation tree $(\mathcal{D}(I), \Supset)$.

Base case: Trivial, since $\mathcal{O}(I) \subseteq \mathcal{O}(I)$.

Inductive step: If the proposition holds for some judgment $J \in \mathcal{D}(I)$, then the oracles of its premises are subsets of $\mathcal{O}(J)$ and thereby of $\mathcal{O}(I)$.

□

Another important property of the semantics is that every returned expression is in weak head normal form, and that expressions which already are in weak head normal form are not changed in the evaluation:

Observation 4. *For every judgment* $L \Downarrow_{p,\gamma} R$ *there is*

1. $\mathrm{whnf}(\mathcal{E}(R))$.
2. $\mathrm{whnf}(\mathcal{E}(L)) \implies \mathcal{E}(L) = \mathcal{E}(R)$.

Proof.

1. By induction over the derivation tree $(\mathcal{D}(L \Downarrow_{p,\gamma} R), \Subset)$: For rule Const^L the resulting expression is obviously in weak head normal form. For rule $\mathit{VarStrict}^L$ the resulting expression is also in weak head normal form; only expressions that are in weak head normal form are bound to constant pointers, so that the value fetched from the heap must be in weak head normal form. Every other rule returns an expression that is the result of a subcomputation.

2. If $\mathcal{E}(L)$ is in weak head normal form, then the only applicable rule is rule Const^L. But if $L \Downarrow_{p,\gamma} R$ holds by rule Const^L, then $\mathcal{E}(L) = \mathcal{E}(R)$ also holds.

□

The following observation states that every oracle entry corresponds to the evaluation of a thunk:

Observation 5. *If the judgment* $S \Downarrow_{p,\gamma} T$ *holds, then for every* $v \in \gamma$ *the derivation tree of* $S \Downarrow_{p,\gamma} T$ *contains a judgment* $(0, 0, L, l) \Downarrow_{v,\delta} (t, c, R, r)$ *such that* $(v, r) \in \mathcal{H}(T)$.

Proof. By induction over the derivation tree $(\mathcal{D}(S \Downarrow_{p,\gamma} T), \Subset)$, distinguishing the rules by which the judgment holds:

Rule Const^L, $\mathit{VarStrict}^L$. Trivial, since $\gamma = \emptyset$.

Rules App^L, Case^L, $\mathit{LetStrict}^L$, $\mathit{LetLazy}^L$. The oracle γ is the union of the oracles of the current judgment's subcomputations. Therefore, every element of γ is contained in the oracle set of at least one subcomputation. The induction hypothesis states that the derivation tree of this subcomputation contains a judgment $(0, 0, L, l) \Downarrow_{v,\delta} (t, c, R, r)$, which is also contained in the derivation tree of $S \Downarrow_{p,\gamma} T$.

Rule *VarLazy*^{L}. Every element of γ is either contained in the oracle of the subcomputation — then the proposition holds by induction —, or it is equal to $\mathcal{E}(S)$. Then the judgment $(0,\, 0,\, L,\, l) \Downarrow_{v,\,\delta} (t,\, c,\, R,\, r)$ is the premise of the current judgment.

□

If there are no right hand sides of non-strict declaration that are already in weak head normal forms, then this observation together with Observation 4 implies that the oracle of a computation consists of the thunk pointers that refer to weak head normal forms on the resulting heap. Formally, if an expression a contains no subexpression $\mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2$ with $\mathrm{whnf}(e_1)$, then there is

$$(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\,\omega} C \implies \omega = \{v \mid (v, e) \in \mathcal{H}(C) \wedge \mathrm{whnf}(e)\}$$

If the non-strict declaration of constants is ruled out, then there is no need to calculate the oracle explicitly; it can be reconstructed from the resulting heap. But in practice it can be useful to declare constructor terms in a non-strict way, because then the user can see which parts of a data structure are actually needed.

If a thunk pointer occurs in a computation, then all its prefixes — except ϵ — will also occur in the resulting heap. The following observation formalizes this:

Observation 6. *Let a be an initial expression with $(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\,\omega} Z$ for some configuration Z. Then for every $v \in \mathrm{dom}\mathcal{H}(Z)$ there is*

$$\{p \in \mathbb{N}^+ \mid q \in \mathbb{N}^* \wedge p \cdot q = \mathcal{T}(v)\} \subseteq \mathrm{dom}\mathcal{H}(Z)$$

with $\mathcal{T}(v) = \begin{cases} v & \text{if } v \in \mathbb{N}^* \\ p & \text{if } v = x_{p:c} \in \mathbb{V}^C \end{cases}$ *for every* $v \in \mathbb{V}$

Proof. If $\mathcal{H}(Z)$ contains an entry (v, e'), then an entry (v, e) must have been added by a judgment $L \Downarrow_{p,\,\gamma} R$ that holds by rule *LetLazy*^{L} or by rule *LetStrict*^{L}, such that there is $i \in \mathbb{N}$ with $p \cdot i = \mathcal{T}(v)$. Now there is either $p = \epsilon$, or the thunk prefix has been set to p in the evaluation of the identifier p by another application of rule *VarLazy*^{L}. But then p will also be contained in $\mathrm{dom}\mathcal{H}(Z)$. From this the proposition follows by induction. □

The following observation states that every judgment that evaluates a declaration has a distict current position.

Observation 7. *For an initial expression a, every judgments $L \Downarrow_{p,\,\gamma} R \in \mathcal{D}(a)$, for which $\mathcal{E}(L)$ is a strict or non-strict declaration, has a distinct current positions.*

Proof. By induction over the derivation tree $(\mathcal{D}(a), \Subset)$, distinguishing the rules by which the judgment hold:

Rules $Const^L$, $VarStrict^L$. These rules have no subcomputations, their left expression is not a declaration, and they leave the current position unchanged.

Rules App^L, $Case^L$. Each rule has two subcomputations. If the left hand side of the first subcomputation contains *no* declaration, then the observation holds by induction for the first subcomputation. Otherwise, Observation 1 implies that the current position of the second subcomputation is greater that the current position of the first, so that the observation holds by induction.

Rule $VarLazy^L$. This rule evaluates a thunk bound to a thunk pointer p. In its subcomputation it sets the current position to $(p \cdot 0, 0)$. The thunk cannot refer to itself, and it is updated with a result, for which Observation 4 states that it is in weak head normal form, so that the current position can only be set to $(p \cdot 0, 0)$ once.

All its subcomputations have either a current position (q, c), so that p is a prefix of q, or they are subcomputations of another application of rule $VarLazy^L$.

Rules $LetStrict^L$, $LetLazy^L$. These rules advance the current position, so that subsequent declarations have another position.

□

3.6 Lazy Call-by-Value Evaluation

The semantics for Lazy Call-by-Value evaluation is kept as similar as possible to the semantics of non-strict oracle creation. The main difference is that no unevaluated thunks are created on the heap. Instead, all declarations are evaluated *at binding time*. For this the oracle is inherited from the root of the derivation tree, and the right hand sides of non-strict declarations are either evaluated or discarded at binding-time, depending on the oracle.

Configurations. Like before, a configuration is a 4-tuple $(t,\, c,\, \Gamma,\, e) \in \mathbb{N} \times \mathbb{N} \times \mathbb{H} \times \mathbb{E}$ that consists of a thunk counter t, a const counter c, a heap Γ and an expression e.

Judgments. A judgment $(t,\, c,\, \Gamma,\, e) \downarrow_{p,\omega} (t',\, c',\, \Delta,\, e')$ relates a configuration $(t,\, c,\, \Gamma,\, e)$ to the result of its evaluation $(t',\, c',\, \Delta,\, e')$. Like the judgments of non-strict oracle creation, it contains a *thunk prefix* $p \in \mathbb{N}^*$

$$Const^S \quad \frac{\text{whnf}(e)}{(t,\, c,\, \Gamma,\, e) \;\downarrow_{p,\omega}\; (t,\, c,\, \Gamma,\, e)}$$

$$App^S \quad \frac{(t,\, c,\, \Gamma,\, f) \;\downarrow_{p,\omega}\; (t',\, c',\, \Delta,\, \lambda x.e) \;\wedge\; (t',\, c',\, \Delta,\, e[v/x]) \;\downarrow_{p,\omega}\; R}{(t,\, c,\, \Gamma,\, f\; v) \;\downarrow_{p,\omega}\; R}$$

$$Var^S \quad \frac{(v,\, e) \in \Gamma}{(t,\, c,\, \Gamma,\, v) \;\downarrow_{p,\omega}\; (t,\, c,\, \Gamma,\, e)}$$

$$Case^S \quad \frac{i \in \{1,\ldots,n\} \;\wedge\; (t,\, c,\, \Gamma,\, e) \;\downarrow_{p,\omega}\; (t',\, c',\, \Delta,\, C_i\, \overline{v_{k_i}}) \;\wedge\; (t',\, c',\, \Delta,\, a_i\overline{[v_n/x_n]}) \;\downarrow_{p,\omega}\; R}{(t,\, c,\, \Gamma,\, \mathbf{case}\; e\; \mathbf{of}\; \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}) \;\downarrow_{p,\omega}\; R}$$

$$LetStrict^S \quad \frac{\forall i \in \{1,\ldots,n\} :\; v_i = (x_i)_{p\cdot t:(c+1)} \;\wedge\; e_i' = e_i\overline{[v_n/x_n]} \;\wedge\; (t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e_n')}\},\, e\overline{[v_n/x_n]}) \;\downarrow_{p,\omega}\; R}{(t,\, c,\, \Gamma,\, \mathbf{let}^!\; \overline{x_n = e_n}\; \mathbf{in}\; e) \;\downarrow_{p,\omega}\; R}$$

$$LetEval^S \quad \frac{p\cdot t \in \omega \wedge\; (0,\, 0,\, \Gamma,\, e_1) \;\downarrow_{p\cdot t,\omega}\; (t',\, c',\, \Delta,\, e_1') \;\wedge\; (t+1,\, 0,\, (\Delta \cup \{(p\cdot t,\, e_1')\},\, e_2[p\cdot t/x]) \;\downarrow_{p,\omega}\; R}{(t,\, c,\, \Gamma,\, \mathbf{let}^?\; x = e_1\; \mathbf{in}\; e_2) \;\downarrow_{p,\omega}\; R}$$

$$LetSkip^S \quad \frac{p\cdot t \notin \omega \;\wedge\; (t+1,\, 0,\, \Gamma,\, e_2[p\cdot t/x]) \;\downarrow_{p,\omega}\; R}{(t,\, c,\, \Gamma,\, \mathbf{let}^?\; x = e_1\; \mathbf{in}\; e_2) \;\downarrow_{p,\omega}\; R}$$

Figure 3.7: Rules for Lazy Call-by-Value evaluation

and an oracle $\omega \subseteq \mathbb{N}^*$. The difference is that both the thunk prefix p and the oracle γ are inherited from the parent judgment.

To evaluate a program p to a configuration $(t',\, c',\, \Delta,\, q)$ directed by an oracle ω means to prove that

$$(0,\, 0,\, \emptyset,\, p) \;\downarrow_{\epsilon,\omega}\; (t',\, c',\, \Delta,\, q).$$

Figure 3.7 shows the derivation rules for Lazy Call-by-Value evaluation:

Const^S, *App*^{S}, *Case*^{S},*LetStrict*^{S}. These rules are the same as the respective rules for non-strict oracle creation. The only difference is that the oracle is not synthesized from the subcomputations, but inherited from the root of the derivation tree.

$$
\begin{array}{l}
\mathit{LetEval}^S_{\epsilon} \left[\begin{array}{l}
(0,\ 0,\ \emptyset,\ \textbf{let}^?\ \ldots\ \textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \rightarrow a) \\
\mathit{LetStrict}^S_0 \left[\begin{array}{l}
(0,\ 0,\ \emptyset,\ \textbf{let}^!\ \ldots\ \textbf{in let}^?\ \ldots\ \textbf{in}\ \mathit{Cons}\ q\ s) \\
\mathit{LetEval}^S_0 \left[\begin{array}{l}
(0,\ 1,\ S,\ \textbf{let}^?\ \ldots\ \textbf{in}\ \mathit{Cons}\ q\ s) \\
\mathit{LetSkip}^S_{00} \left[\begin{array}{l}
(0,\ 1,\ S,\ \textbf{let}^?\ \ldots\ \textbf{in}\ \mathit{Cons}\ r\ s_{00:1}) \\
\mathit{Const}^L_{00} \left[\begin{array}{l}
(1,\ 0,\ S,\ \mathit{Cons}\ 000\ s_{00:1}) \\
(1,\ 0,\ S,\ \mathit{Cons}\ 000\ s_{00:1})
\end{array} \right.
\end{array} \right. \\
\mathit{Const}^S_0 \left[\begin{array}{l}
(1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
(1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1})
\end{array} \right.
\end{array} \right. \\
(1,\ 0,\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1}) \\
\mathit{Case}^S_{\epsilon} \left[\begin{array}{l}
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ \textbf{case}\ 0\ \textbf{of}\ \mathit{Cons}\ a\ b \rightarrow a) \\
\mathit{Var}^S_{\epsilon} \left[\begin{array}{l}
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ 0) \\
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ \mathit{Cons}\ 00\ s_{00:1})
\end{array} \right. \\
\mathit{Var}^S_{\epsilon} \left[\begin{array}{l}
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ 00) \\
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1})
\end{array} \right. \\
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1})
\end{array} \right. \\
(1,\ 0,\ P\ \cup\ Q\ \cup\ S,\ \mathit{Cons}\ 000\ s_{00:1})
\end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
P = \{(0, \mathit{Cons}\ 00\ s_{00:1})\} \\
Q = \{(00, \mathit{Cons}\ 000\ s_{00:1})\} \\
S = \{(s_{00:1}, \mathit{True})\}
\end{array}
$$

Figure 3.8: Example of Lazy Call-by-Value evaluation

Var^S. In this rule, rules *VarStrict*^{L} and *VarLazy*^{L} of the non-strict oracle creation are merged. Since only weak head normal forms are bound on the heap, there is no need to evaluate the expressions that are fetched from the heap.

LetEval^S. This rule handles variable bindings that will be needed by subsequent computations. For this it uses the information collected in the inherited oracle. If the oracle contains the name of the identifier, then it is known that the binding is needed.

LetSkip^S. This rule handles variable bindings that are not needed by subsequent computations. It simply increments the thunk counter i and evaluates e_2 in the unchanged heap Γ.

Figure 3.8 shows the Lazy Call-by-Value evaluation of the example ex-

pression from page 33, together with the oracle $\{0, 00\}$. As in non-strict oracle creation, the expression *Cons* 000 $s_{00:1}$ is returned, but this time the heap does not contain the unevaluated thunk bound to pointer 000:

$$\begin{array}{ll}\{(0, & \mathit{Cons}\ 00\ s_{00:1}),\\ (00, & \mathit{Cons}\ 000\ s_{00:1}),\\ (s_{00:1}, & \mathit{True})\}\end{array}$$

In the example evaluation it can be seen how the order of evaluation differs from non-strict oracle creation: Now all right hand sides are evaluated before they are bound on the heap, so that the current position increases monotonically w.r.t. order $\preccurlyeq$ in the course of the evaluation. We fix this in the following observation:

Observation 8. *In the derivation of a judgment* $(t, c, \Gamma, e) \downarrow_{p,\omega} C$*, the current position is monotonically increasing with respect to order* $\preccurlyeq$*.*

Proof. Rules Const^S and Var^S have no subcomputations and leave the thunk counter and constant counter unchanged. For every other rule, the inequality

$$(p \cdot t, c) \preccurlyeq (q \cdot s, b) \preccurlyeq (q \cdot s', b') \qquad (*)$$

holds for every subcomputation $(s, b, F, f) \downarrow_{q,\omega} (s', b', G, g)$ by induction over the derivation tree.

Rule $\mathit{LetStrict}^S$ as well as rulle $\mathit{LetSkip}^S$ has exactly one judgment as its premise, so that the observation follows directly from (*).

Only the premises of rules App^S, Case^S and $\mathit{LetEval}^S$ contain more than one subcompution. In all three rules, the resulting thunk counter and constant counter of the first subcompution are equal to the initial thunk counter and constant counter of the second subcompution. Thus, the observation follows from (*). □

3.7 Proof of Correctness

It remains to be shown that for every non-strict oracle creation there is a Lazy Call-by-Value evaluation that computes the same result. Since only needed values are bound on the heap, we can not expect the resulting heaps to be the same; in the above example, the heap entry 000 was missing in the resulting heap of the Lazy Call-by-Value evaluation. Therefore, we consider both heaps as sufficiently equal if they contain the same bindings for strictly-bound identifiers in $\mathbb{V}^C$ and for thunk identifiers that were needed to evaluate the program. Therefore, we regard the resulting configurations as equivalent if

1. they contain the same expression, and

2. the heap resulting from Lazy Call-by-Value evaluation is identical to the heap resulting from non-strict oracle creation, restricted to $\mathbb{V}^C \cup \omega$.

The following diagram shows the correspondence that has to be shown:

$$\begin{array}{ccc} (0,\, 0,\, \emptyset,\, a) & \xRightarrow[\epsilon,\omega]{} & (t,\, c,\, H,\, z) \\ \vdots & & \vdots \\ (0,\, 0,\, \emptyset,\, a) & \xrightarrow[\epsilon,\omega]{} & (t,\, c,\, H|_{\mathbb{V}^C \cup \omega},\, z) \end{array}$$

First, we show that non-strict oracle creation does not remove or change thunks after they have been evaluated. Then we show that non-strict oracle creation does not introduce heap pointers to values that are not yet bound on the heap. From this we conclude that expressions may be evaluated as soon as they are bound on the heap.

We start by introducing the notion of *pre-heaps*. If a heap Γ is a pre-heap of a heap Δ, then the evaluated thunks contained in Γ are also contained in Δ. As it turns out, every heap that occurs in the non-strict oracle creation is a pre-heap of the resulting heap.

Definition (Pre-heap)**.** *The relation* $\trianglelefteq :\subseteq \mathbb{H} \times \mathbb{H}$ *is defined by*

$$\Gamma \trianglelefteq \Delta \;:\Longleftrightarrow\; \forall (v, e) \in \Gamma :\; \text{whnf}(e) \Rightarrow (v, e) \in \Delta$$

for all Γ, $\Delta \in \mathbb{H}$. *If* $\Gamma \trianglelefteq \Delta$, *then* Γ *is called a* pre-heap *of* Δ.

Lemma 1. *The relation* $\trianglelefteq$ *is a preorder on* $\mathbb{H}$.

Proof. We have to show that $\trianglelefteq$ is reflexive and transitive.

1. *Reflexivity.* Trivial, since for every heap Γ and every $(v, e) \in \Gamma$ there is $\text{whnf}(e) \Rightarrow (v, e) \in \Gamma$

2. *Transitivity.* Let Γ, Δ and Θ be heaps with $\Gamma \trianglelefteq \Delta$ and $\Delta \trianglelefteq \Theta$. If no weak head normal forms are bound in Γ, then $\Gamma \trianglelefteq \Theta$ holds trivially. Otherwise choose $(v, e) \in \Gamma$ with $\text{whnf}(e)$. Then $\Gamma \trianglelefteq \Delta$ implies that $(v, e) \in \Delta$. Since $\text{whnf}(e)$ holds, $\Delta \trianglelefteq \Theta$ implies that $(v, e) \in \Theta$.

□

The following two lemmata state that in a judgment $L \Downarrow_{p,\gamma} R$ the left heap $\mathcal{H}(L)$ is always a pre-heap of the right heap $\mathcal{H}(R)$, and the resulting heap of every subcomputation is a pre-heap of the right heap $\mathcal{H}(R)$. This implies that in a computation every intermediate heap is a pre-heap of the final heap.

Lemma 2. *If* $L \Downarrow_{p,\gamma} R$, *then* $\mathcal{H}(L) \trianglelefteq \mathcal{H}(R)$.

Proof. By induction over the derivation tree $(\mathcal{D}(L \Downarrow_{p,\gamma} R), \Subset)$, distinguishing between the rules by which the judgment holds:

Rules $Const^L$, $VarStrict^L$. Since $\trianglelefteq$ is reflexive, $\mathcal{H}(L) \trianglelefteq \mathcal{H}(R)$ follows from $\mathcal{H}(L) = \mathcal{H}(R)$.

Rules App^L, $Case^L$. After applying the induction hypothesis to the subcomputations, $\mathcal{H}(L) \trianglelefteq \mathcal{H}(R)$ follows from the transitivity of $\trianglelefteq$.

Rule $VarLazy^L$ Assume that $(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\gamma\cup\{v\}} (t,\, c,\, \Delta \cup \{(v,\, e')\},\, e')$ holds by rule $VarLazy^L$. Then there is an expression e, such that $(v, e) \in \Gamma$ and the judgment $(0,\, 0,\, \Gamma \setminus \{(v, e)\},\, e) \Downarrow_{v,\gamma} (t',\, c',\, \Delta,\, e')$ is a premise of the current judgment.

Now $\Gamma \setminus \{(v, e)\} \trianglelefteq \Delta$ holds by the induction hypothesis.

Since the subcomputation can only introduce fresh identifiers, v is not bound in Δ, so that $\Delta \trianglelefteq \Delta \cup \{(v, e')\}$ also holds. It follows that $\Gamma \setminus \{(v, e)\} \trianglelefteq \Delta \cup \{(v, e')\}$ holds by the transitivity of $\trianglelefteq$.

Observation 4 states that $e = e'$ or not whnf(e), so that $\Gamma \trianglelefteq \Delta \cup \{(v, e')\}$ also holds.

Rule $LetStrict^L$. If the judgment $(t,\, c,\, \Gamma,\, \mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e) \Downarrow_{p,\gamma} T$ holds by rule $LetStrict^L$, then $(t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e[\overline{v_n/x_n}]) \Downarrow_{p,\gamma} T$ holds with $v_i = (x_i)_{p\cdot t:(c+1)}$ and $e'_i = e_i[\overline{v_n/x_n}]$ for every $i \in \{1, \ldots, n\}$.

Then $\Gamma \trianglelefteq \Gamma \cup \{\overline{(v_n, e'_n)}\}$ holds, because all $v_1, \ldots, v_n$ are fresh identifiers, , and $\Gamma \cup \{\overline{(v_n, e'_n)}\} \trianglelefteq \mathcal{H}(T)$ holds by the induction hypothesis, so that $\Gamma \trianglelefteq \mathcal{H}(T)$ follows from the transitivity of $\trianglelefteq$.

Rule $LetLazy^L$. If $(t,\, c,\, \Gamma,\, \mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2) \Downarrow_{p,\gamma} T$ holds by rule $LetLazy^L$, then $(t+1,\, 0,\, \Gamma \cup \{(p \cdot t,\, e_1)\},\, e_2[p \cdot t/x]) \Downarrow_{p,\gamma} T$ holds.

$\Gamma \trianglelefteq \Gamma \cup \{(p \cdot t, e_1)\}$ holds, because $p \cdot t$ is a fresh identifier, and $\Gamma \cup \{(p \cdot t, e_1)\} \trianglelefteq \mathcal{H}(T)$ holds by the induction hypothesis, so that $\Gamma \trianglelefteq \mathcal{H}(T)$ follows from the transitivity of $\trianglelefteq$.

□

The next lemma shows that the resulting heap of a subcomputation is always a pre-heap of the overall computation. Intuitively, this is clear, every rule except rule $VarLazy^L$ returns the resulting heap of a subcomputation, and rule $VarLazy^L$ only adds an entry to it.

Lemma 3. *Assume that the judgment $A \Downarrow_{q,\delta} Z$ holds. Then for every judgment $L \Downarrow_{p,\gamma} R$ in its derivation tree there is $\mathcal{H}(R) \trianglelefteq \mathcal{H}(Z)$.*

Proof. By induction over the reversed derivation tree $(\mathcal{D}(A \Downarrow_{\epsilon,\omega} Z), \ni)$.

Base case: If $L \Downarrow_{p,\gamma} R$ is the root of the derivation tree, then $\mathcal{H}(R) = \mathcal{H}(Z)$, so that the proposition follows by the reflexivity of $\trianglelefteq$.

Inductive step: Assuming that $\mathcal{H}(R) \trianglelefteq \mathcal{H}(Z)$ holds for a judgment $L \Downarrow_{p,\gamma} R$, it is shown that the proposition also holds for the judgments that occcur in its premise. We distinguish the rules by which $L \Downarrow_{p,\gamma} R$ holds:

Rules $Const^L$, $VarStrict^L$ The premise of these rules contain no judgments.

Rules App^L, $Case^L$ Each rule contains two judgments $P \Downarrow_{p,\alpha} Q$ and $S \Downarrow_{p,\beta} T$ as its premise, for which $\mathcal{H}(T) = \mathcal{H}(R)$ and $\mathcal{H}(Q) = \mathcal{H}(S)$ holds. Therefore, the proposition holds for the second premise, and $\mathcal{H}(S) \trianglelefteq \mathcal{H}(Z)$ follows by Lemma 2.

Rule $VarLazy^L$ This rule has one judgment $P \Downarrow_{v,\alpha} Q$ as its premise. There is $\mathcal{H}(Q) \subseteq \mathcal{H}(R)$, so that $\mathcal{H}(Q) \trianglelefteq \mathcal{H}(R)$. Since $\trianglelefteq$ is transitive, there is also $\mathcal{H}(Q) \trianglelefteq \mathcal{H}(Z)$.

Rules $LetStrict^L$, $LetLazy^L$ Each rule has one judgment $P \Downarrow_{v,\alpha} Q$ as its premise, for which $\mathcal{H}(Q) = \mathcal{H}(R)$ holds. Therefore, $\mathcal{H}(Q) \trianglelefteq \mathcal{H}(Z)$ holds by the induction hypothesis.

□

Now it is easy to see that every heap that occurs in a computation is a pre-heap of that computation's final heap:

Corollary 4. *Let a be an initial expression, with $(0, 0, \emptyset, a) \Downarrow_{\epsilon,\omega} C$ for some configuration C. Then for every judgment $L \Downarrow_{p,\gamma} R \in \mathcal{D}(a)$ in its derivation tree there is*

1. $\mathcal{H}(R) \trianglelefteq \mathcal{H}(C)$.
2. $\mathcal{H}(L) \trianglelefteq \mathcal{H}(C)$.

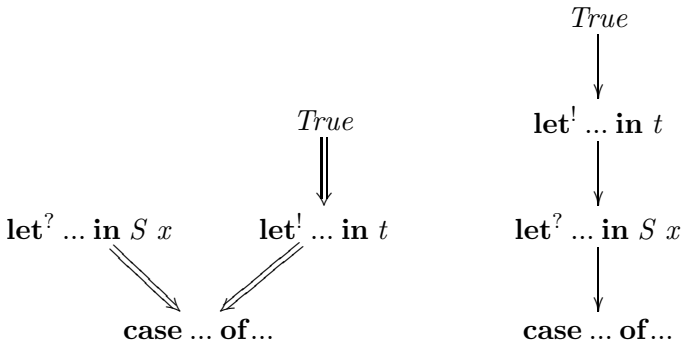
Proof. Direct consequence of Lemma 3 and Lemma 2. □

Relation $\sqsubset$ draws the connection between the derivation trees of a non-strict oracle creation and the corresponding Lazy Call-by-Value evaluation. Consider the following expression:

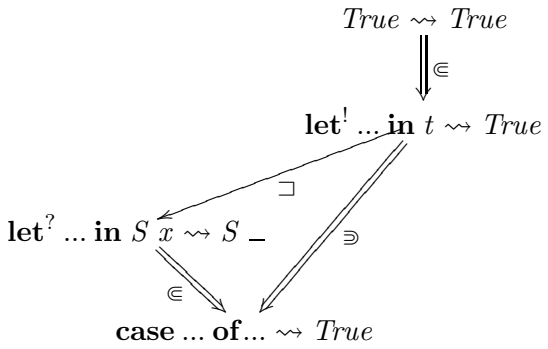
$$\textbf{case}\ (\textbf{let}^?\ x = (\textbf{let}^!\ t = \mathit{True}\ \textbf{in}\ t)\ \textbf{in}\ S\ x)\ \textbf{of}\ S\ y \to y$$

Under lazy evaluation, the value of x is forced after the alternative of the case expression has been selected; under Lazy Call-by-Value evaluation the

value of x is calculated before the body $S\ x$ is returned:



Under Lazy Call-by-Value evaluation, the evaluation of x is a subcomputation of its declaration. Relation $\sqsubset$ adds this connection to relation $\Subset$, so that their union $\lessdot$ combines the structure of the lazy derivation with the structure of the Lazy Call-by-Value derivation:



This makes it possible to relate the two evaluation strategies to each other in the central Lemma 10.

Definition. *The relations $\sqsubset$ and $\lessdot$ over judgments are defined, such that for all judgments $P \Downarrow_{p\cdot i,\,\delta} Q$ and $R \Downarrow_{p,\,\gamma} S$ with $i \in \mathbb{N}$ the statement*

$$P \Downarrow_{p\cdot i,\,\delta} Q \sqsubset R \Downarrow_{p,\,\gamma} S$$

holds iff $\mathcal{E}(R)$ not in weak head normal form, and for all judgments I and J there is

$$I \lessdot J :\Longleftrightarrow I \Subset J \vee I \sqsubset J.$$

Lemma 5. *Let a be an initial expression with $(0,\,0,\,\emptyset,\,a) \Downarrow_{\epsilon,\,\omega} C$ for some oracle ω and some configuration C.*

Then $\lessdot$ is a well-founded relation over $\mathcal{D}(a)$.

Proof. On $\mathcal{D}(a)$ the relations $\Subset$ and $\sqsubset$ are finite and circle-free. Therefore, we have to show that their union $\lessdot$ is also circle-free on $\mathcal{D}(a)$. Since $\Subset$ is

transitive on $\mathcal{D}(a)$, it is sufficient to show that there are no judgments I, $J \in \mathcal{D}(a)$ with $J \Subset I$ and $I \sqsubset J$. This is shown by contradiction.

Assume that there are judgments $I := P \Downarrow_{p \cdot i,\, \delta} Q$, $J := R \Downarrow_{p,\, \gamma} S$ in $\mathcal{D}(a)$ with $i \in \mathbb{N}$, such that $J \Subset I$ and $I \sqsubset J$.

The only rule that can change the thunk prefix is rule $\mathit{VarLazy}^L$. In this rulethe expression to be evaluated is a thunk pointer; this thunk pointer will be used as thunk prefix in the rule' premise. Therefore, that judgment I has thunk prefix $p \cdot i$ implies that there is an application of rule $\mathit{VarLazy}^L$ $A \Downarrow_{b,\, \alpha} B$ with $I \Subset A \Downarrow_{b,\, \alpha} B$ and $\mathcal{E}(A) = p \cdot i$, which sets the thunk prefix to $p \cdot i$ in its subcomputation.

It is impossible that $p = \epsilon$, because no rule can set the thunk prefix of its subcomputation to ϵ if the rule itself does not have the thunk prefix ϵ. But there is $J \Subset I$ with $\mathcal{O}(I) \neq \epsilon$. Therefore, judgment J can only have thunk prefix p if there is a judgment $C \Downarrow_{d,\, \beta} D$ with $J \Subset C \Downarrow_{d,\, \beta} D$ and $\mathcal{E}(C) = p$. Thus, there is:

$$J \Subset C \Downarrow_{d,\, \beta} D \Subset I \Subset A \Downarrow_{b,\, \alpha} B$$

We have to distinguish two cases:

1. There is $(p, e) \in \mathcal{H}(P)$ with $\mathrm{whnf}(e)$. Then J holds by rule Const^L and is a direct premise of $C \Downarrow_{d,\, \beta} D$. But there is $\mathrm{whnf}(\mathcal{H}(R))$, which contradicts the assumption $I \sqsubset J$.

2. There is no $(p, e) \in \mathcal{H}(P)$ with $\mathrm{whnf}(e)$. Since $\mathcal{E}(A) = p \cdot i$, $\mathcal{H}(A)$ contains a binding of $p \cdot i$. Thus, a judgment K with thunk prefix p must have defined $p \cdot i$ before the judgment $A \Downarrow_{b,\, \alpha} B$ could be derived. For that, another judgment must already have started the evaluation of p, so that p could become the thunk prefix of K.

   Since there is no $(p, e) \in \mathcal{H}(P)$ with $\mathrm{whnf}(e)$ there is

   - judgment I is a subcomputation of the evaluation of p, and
   - judgment $C \Downarrow_{d,\, \beta} D$ with $\mathcal{E}(C) = p$ implies that p is defined a second time by a subcomputation of I.

   This is a contradiction, because every heap pointer can only be introduced once.

□

The predicate $\mathcal{B}$ is needed to assure that a configuration contains only backwards references, i.e. that the expression only refers to identifiers that are already defined in the heap, and that every heap entry v refers only to identifiers in $\mathbb{V}_v$:

Definition. *For every thunk prefix $p \in \mathbb{N}^*$ the predicate $\mathcal{B}_p$ on configurations is defined by*

$$\mathcal{B}_p((t,\, c,\, \Gamma,\, e)) \;:\Longleftrightarrow\; \mathcal{V}(e) \subseteq \mathbb{V}_{p\cdot t:c} \wedge \forall (v,g) \in \Gamma : \mathcal{V}(g) \subseteq \mathbb{V}_v$$

Lemma 6. *For every judgment $S \Downarrow_{p,\,\gamma} T$ there is*

$$\mathcal{B}_p(S) \implies \mathcal{B}_p(T)$$

Proof. By induction over the derivation tree $(\mathcal{D}(S \Downarrow_{p,\,\gamma} T), \Subset)$, distinguishing the rules by which the judgment holds:

Rule Const^L. Since $S = T$, the proposition holds trivially.

Rule App^L. If the judgment $(t,\, c,\, \Gamma,\, f\, v) \Downarrow_{p,\,\gamma\cup\delta} T$ holds by rule App^L, then $(t,\, c,\, \Gamma,\, f) \Downarrow_{p,\,\gamma} (t',\, c',\, \Delta,\, \lambda x.e)$ and $(t',\, c',\, \Delta,\, e[v/x]) \Downarrow_{p,\,\delta} T$.

If $\mathcal{B}_p((t,\, c,\, \Gamma,\, f\ v))$, then $\mathcal{V}(f) \subseteq \mathcal{V}(f\ v)$ implies $\mathcal{B}_p((t,\, c,\, \Gamma,\, f))$, and $\mathcal{B}_p((t',\, c',\, \Delta,\, \lambda x.e))$ follows by the induction hypothesis.

Observation 1 on page 34 states that $\mathbb{V}_{p\cdot t:c} \subseteq \mathbb{V}_{p\cdot t':c'}$, so there is $v \in \mathbb{V}_{p\cdot t:c} \subseteq \mathbb{V}_{p\cdot t':c'}$, there is $\mathcal{V}(e[v/x]) \subseteq \mathcal{V}(e) \cup \{v\} \subseteq \mathbb{V}_{p\cdot t':c'}$.

Therefore, $\mathcal{B}_p((t',\, c',\, \Delta,\, e[v/x]))$ also holds, and $\mathcal{B}_p(T)$ follows by the induction hypothesis.

Rule $\mathit{VarStrict}^L$. If $(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\,\gamma} (t,\, c,\, \Gamma,\, e)$ holds by rule $\mathit{VarStrict}^L$, then $(v, e) \in \Gamma$. If $\mathcal{B}_p((t,\, c,\, \Gamma,\, v))$, then $v \in \mathbb{V}_{\mathbb{N}^*:c}$ implies that $\mathcal{V}(e) \subseteq \mathbb{V}_v \subseteq \mathbb{V}_{p\cdot t:c}$, and therefore $\mathcal{B}_p((t,\, c,\, \Gamma,\, e))$ holds.

Rule $\mathit{VarLazy}^L$. If $I := (t,\, c,\, \Gamma,\, v) \Downarrow_{p,\,\gamma\cup\{v\}} (t',\, c',\, \Delta \cup \{(v, e')\},\, e')$ holds by rule $\mathit{VarLazy}^L$, then $(v, e) \in \Gamma$ and $J := (0,\, 0,\, \Gamma \setminus \{(v, e)\},\, e) \Downarrow_{v,\,\gamma} (t',\, c',\, \Delta,\, e')$ holds as its premise.

If $\mathcal{B}_p((t,\, c,\, \Gamma,\, v))$, then $v \in \mathbb{V}_{\mathbb{N}^*:c}$ implies that $\mathcal{V}(e) \subseteq \mathbb{V}_v \subseteq \mathbb{V}_{v\cdot 0:0}$, and therefore $\mathcal{B}_v((0,\, 0,\, \Gamma \setminus \{(v, e)\},\, e))$. With the induction hypothesis $\mathcal{B}_v((t',\, c',\, \Delta,\, e'))$ follows.

It remains to show that $\mathbb{V}_{v\cdot t':c'} \subseteq \mathbb{V}_{p\cdot t:c}$.

Since $v \in \mathbb{V}_{p\cdot t:c}$, there must be $v <_{\mathbb{N}^*} p \cdot t$.

Assume that $\mathbb{V}_{v\cdot t':c'} \not\subseteq \mathbb{V}_{p\cdot t:c}$. Then there must be $(p \cdot t, c) \preccurlyeq (v \cdot t', c')$. This together with $v <_{\mathbb{N}^*} p \cdot t$ implies that there is an $i \in \mathbb{N}$, such that $v \cdot i = p$ and $i < t'$. But then $I \sqsubset J$ holds. Since the latter is a subcomputation of the former, there is $I \lessdot J$ and $J \lessdot I$, which contradicts Lemma 5.

Rule Case^L. If the judgment $(t,\, c,\, \Gamma,\, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}) \Downarrow_{p,\,\gamma\cup\delta} T$ holds by rule Case^L, then there is $i \in \{1, \ldots, n\}$, such that the judgments $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\,\gamma} (t',\, c',\, \Delta,\, C_i\, \overline{v_{k_i}})$ and $(t',\, c',\, \Delta,\, a_i[\overline{v_n/x_n}]) \Downarrow_{p,\,\delta} T$ hold.

If $\mathcal{B}_p((t,\, c,\, \Gamma,\, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}))$, then $\mathcal{B}_p((t,\, c,\, \Gamma,\, e))$ and by the induction hypothesis it follows that $\mathcal{B}_p((t',\, c',\, \Delta,\, C_i\, \overline{v_{k_i}}))$.

Since $\mathcal{V}(a_i) \subseteq \mathbb{V}_{p\cdot t:c} \subseteq \mathbb{V}_{p\cdot t':c'}$ and $\{v_1, \ldots, v_{k_i}\} \subseteq \mathbb{V}_{p\cdot t':c'}$, there is $\mathcal{V}(a_i\overline{[v_n/x_n]}) \subseteq \mathbb{V}_{p\cdot t':c'}$. It follows that $\mathcal{B}_p((t',\, c',\, \Delta,\, a_i\overline{[v_n/x_n]}))$, so that $\mathcal{B}_p(T)$ follows by the induction hypothesis.

Rule $\mathit{LetStrict}^L$. If the judgment $(t,\, c,\, \Gamma,\, \mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e) \Downarrow_{p,\gamma} T$ holds by rule $\mathit{LetStrict}^L$, then $(t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]}) \Downarrow_{p,\gamma} T$ holds with $v_i = (x_i)_{p\cdot t:(c+1)}$ and $e'_i = e_i\overline{[v_n/x_n]}$ for every $i \in \{1, \ldots, n\}$.

If $\mathcal{B}_p((t,\, c,\, \Gamma,\, \mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e))$, then $\{v_1, \ldots, v_n\} \subseteq \mathbb{V}_{p\cdot t:(c+1)}$, so that

$$\mathcal{V}(e'_i) \subseteq \mathbb{V}_{p\cdot t:c} \cup \{v_1, \ldots, v_n\} \subseteq \mathbb{V}_{p\cdot t:(c+1)}$$

for every $i \in \{1, \ldots, n\}$. This implies

$$\mathcal{B}_p((t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]})),$$

so that $\mathcal{B}_p(T)$ follows by the induction hypothesis.

Rule $\mathit{LetLazy}^L$. If $(t,\, c,\, \Gamma,\, \mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2) \Downarrow_{p,\gamma} T$ holds by rule $\mathit{LetLazy}^L$, then $(t+1,\, 0,\, \Gamma \cup \{(p\cdot t,\, e_1)\},\, e_2[p\cdot t/x]) \Downarrow_{p,\gamma} T$ holds.

If $\mathcal{B}_p((t,\, c,\, \Gamma,\, \mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2))$, then $p\cdot t \in \mathbb{V}_{p\cdot(t+1):0}$ implies that $\mathcal{V}(e_2[p\cdot t/x]) \subseteq \mathbb{V}_{p\cdot(t+1):0}$.

It follows that $\mathcal{B}_p((t+1,\, 0,\, \Gamma \cup \{(p\cdot t,\, e_1)\},\, e_2[p\cdot t/x]))$, so that $\mathcal{B}_p(T)$ follows by the induction hypothesis.

□

Lemma 7. *Assume that the judgment $A \Downarrow_{q,\delta} Z$ holds with $\mathcal{B}_p(A)$. Then for every judgment $L \Downarrow_{p,\gamma} R$ in its derivation tree there is $\mathcal{B}_p(L)$.*

Proof. By induction over the reversed derivation tree $(\mathcal{D}(A \Downarrow_{q,\delta} Z), \ni)$.

Base case: For the judgment $A \Downarrow_{q,\delta} Z$ the proposition holds by assumption.

Inductive step: Assuming that the proposition holds for a judgment, it has to be shown that it also holds for the judgments that occcur in its premise.

Rules Const^L, $\mathit{VarStrict}^L$ There are no judgments in the premises of these rules.

Rule App^L. If the judgment $(t,\, c,\, \Gamma,\, f\ v) \Downarrow_{p,\gamma\cup\delta} R$ holds by rule App^L, then $(t,\, c,\, \Gamma,\, f) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, \lambda x.e)$ and $(t',\, c',\, \Delta,\, e[v/x]) \Downarrow_{p,\delta} R$ hold as its premise.

By the induction hypothesis there is $\mathcal{B}_p((t,\, c,\, \Gamma,\, f\ v))$. This implies $\mathcal{V}(f) \subseteq \mathbb{V}_{p\cdot t:c}$, so that $\mathcal{B}_p((t,\, c,\, \Gamma,\, f))$ holds. By Lemma 6 it follows that $\mathcal{B}_p((t',\, c',\, \Delta,\, \lambda x.e))$. Since $v \in \mathbb{V}_{p\cdot t:c} \subseteq \mathbb{V}_{p\cdot t':c'}$, there is $\mathcal{V}(e[v/x]) \subseteq \mathcal{V}(e) \cup \{v\} \subseteq \mathbb{V}_{p\cdot t':c'}$, so that $\mathcal{B}_p((t',\, c',\, \Delta,\, e[v/x]))$ also holds.

Rule *VarLazy*^{L}. If the judgment $(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\,\gamma\cup\{v\}} R$ holds by rule *VarLazy*^{L}, then there is $(v, e) \in \Gamma$, such that the judgment $(0,\, 0,\, \Gamma \setminus \{(v,\, e)\},\, e) \Downarrow_{v,\,\gamma} U$ holds as its premise for some configuration U.
By the induction hypothesis there is $\mathcal{B}_p((t,\, c,\, \Gamma,\, v))$. Since $\mathcal{V}(e) \subseteq \mathbb{V}_v \subseteq \mathbb{V}_{v\cdot 0:0}$, there is also $\mathcal{B}_p((0,\, 0,\, \Gamma \setminus \{(v,\, e)\},\, e))$.

Rule *LetStrict*^{L}. If the judgment $(t,\, c,\, \Gamma,\, \mathbf{let}^!\, \overline{x_n = e_n}\, \mathbf{in}\, e) \Downarrow_{p,\,\gamma} R$ holds by rule *LetStrict*^{L}, then as its premise the judgment $(t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]}) \Downarrow_{p,\,\gamma} R$ holds with $v_i = (x_i)_{p\cdot t:(c+1)}$ and $e'_i = e_i\overline{[v_n/x_n]}$ for every $i \in \{1, \ldots, n\}$. Now $\mathcal{B}_p((t,\, c,\, \Gamma,\, \mathbf{let}^!\, \overline{x_n = e_n}\, \mathbf{in}\, e))$ follows by the induction hypothesis.
Since $v_i \in \mathbb{V}_{p\cdot t:(c+1)}$ and $\mathcal{V}(e'_i) \subseteq \mathbb{V}_{p\cdot t:(c+1)}$ for every $i \in \{1, \ldots, n\}$, there is $\mathcal{B}_p((t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]}))$.

Rule *LetLazy*^{L}. If the judgment $(t,\, c,\, \Gamma,\, \mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2) \Downarrow_{p,\,\gamma} R$ holds by rule *LetLazy*^{L}, then the judgment $(t+1,\, 0,\, (\Gamma \cup \{(p\cdot t,\, e_1)\},\, e_2[p\cdot t/x]) \Downarrow_{p,\,\gamma} R$ holds as its premise. By the induction hypothesis $\mathcal{B}_p((t,\, c,\, \Gamma,\, \mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2))$ follows.
Since $p\cdot t \in \mathbb{V}_{p\cdot(t+1):0}$ and $\mathcal{V}(e_1) \subseteq \mathbb{V}_{p\cdot(t+1):0}$, there is $\mathcal{B}_p((t+1,\, 0,\, \Gamma \cup \{(p\cdot t, e_1)\},\, e_2[p\cdot t/x]))$.

□

Corollary 8. *Let a be an initial expression with $(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\,\omega} Z$ for some configuration Z and oracle ω. Then for every judgment $L \Downarrow_{p,\,\gamma} R$ in $\mathcal{D}(a)$ there is $\mathcal{B}_p(L)$ and $\mathcal{B}_p(R)$.*

Proof. There is $\mathcal{V}(a) = \emptyset$, so that $\mathcal{B}_\epsilon((0,\, 0,\, \emptyset,\, a))$ holds vacuously, and the proposition is a direct consequence of Lemma 6 and Lemma 7. □

The following lemma states that no evaluation can introduce "forward-references" to heap pointers that are not yet defined:

Lemma 9. *For every judgment $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\,\gamma} (t',\, c',\, \Delta,\, e')$ there is*

$$\mathcal{B}_p((t,\, c,\, \Gamma,\, e)) \implies \mathrm{dom}\Delta \setminus \mathrm{dom}\Gamma \subseteq \mathbb{V}_{p\cdot t':c'}$$

Proof. By induction over the derivation tree $(\mathcal{D}(a), \Subset)$.

Rule *Const*^{L}, *VarStrict*^{L} Since $\Delta \setminus \Gamma = \emptyset$, the proposition holds trivially.

Rule *App*^{L}, *Case*^{L}. With Observation 1 and Lemma 6 the proposition follows directly from induction hypothesis.

Rule *VarLazy*^{L}. If the judgment

$$I := (t,\, c,\, \Gamma,\, v) \Downarrow_{p,\,\gamma\cup\{v\}} (t,\, c,\, \Delta \cup \{(v, e')\},\, e')$$

holds by rule *VarLazy*^{L}, then $(v, e) \in \Gamma$ and the judgment

$$J := (0,\, 0,\, \Gamma \setminus \{(v, e)\},\, e) \Downarrow_{v,\,\gamma} (t',\, c',\, \Delta,\, e')$$

holds as its premise.

The induction hypothesis states that $\mathrm{dom}\Delta \setminus \mathrm{dom}(\Gamma \setminus \{(v, e)\}) \subseteq \mathbb{V}_{v\cdot t':c'}$. Since $v \in \mathbb{V}_{v\cdot t':c'}$, there is also $\mathrm{dom}(\Delta \cup \{(v, e')\}) \setminus \mathrm{dom}\Gamma \subseteq \mathbb{V}_{v\cdot t':c'}$

Lemma 7 together with Lemma 6 states that $\mathcal{B}_v((t',\, c',\, \Delta,\, e'))$, and so $v \in \mathbb{V}_{p\cdot t:c}$.

There is no $i \in \mathbb{N}$, such that $v \cdot i = p$, because then $J \Subset I$ and $I \sqsubset J$ would hold simultaneously, which would contradict Lemma 5.

Therefore, $(v \cdot t', c') \preccurlyeq (p \cdot t, c)$ holds, which implies $\mathbb{V}_{v\cdot t':c'} \subseteq \mathbb{V}_{p\cdot t:c}$, so that the second proposition holds.

Rule *LetStrict*^{L}. If $(t,\, c,\, \Gamma,\, \mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e) \Downarrow_{p,\,\gamma} (t',\, c',\, \Delta,\, e')$ holds by rule *LetStrict*^{L}, then $(t,\, c{+}1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e[\overline{v_n/x_n}]) \Downarrow_{p,\,\gamma} (t',\, c',\, \Delta,\, e')$ holds with $v_i = (x_i)_{p\cdot t:(c+1)}$ and $e'_i = e_i[\overline{v_n/x_n}]$ for all $i \in \{1, \ldots, n\}$.

The induction hypothesis states that $\mathrm{dom}\Delta \setminus \mathrm{dom}(\Gamma \cup \{\overline{(v_n,\, e'_n)}\}) \subseteq \mathbb{V}_{p\cdot t':c'}$, and so the proposition follows from $\{v_1, \ldots, v_n\} \subseteq \mathbb{V}_{p\cdot t':c'}$.

Rule *LetLazy*^{L}. If $(t,\, c,\, \Gamma,\, \mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2) \Downarrow_{p,\,\gamma} T$ holds by rule *LetLazy*^{L}, then $(t + 1,\, 0,\, \Gamma \cup \{(p \cdot t,\, e_1)\},\, e_2[p \cdot t/x]) \Downarrow_{p,\,\gamma} T$ holds.

The induction hypothesis states that $\mathrm{dom}\Delta \setminus \mathrm{dom}(\Gamma \cup \{(p \cdot t,\, e_1)\}) \subseteq \mathbb{V}_{p\cdot t':c'}$, and so the proposition follows from $p \cdot t \in \mathbb{V}_{p\cdot t':c'}$.

□

For a heap H, an oracle ω and a current position $(p \cdot t, c)$ the heap $H^\omega_{p:c}$ contains those definitions $(v, e) \in H$, for which

1. $v \in \mathbb{V}_{p:c}$ holds, so that they can be defined by Lazy Call-by-Value evaluation before the current position $(p \cdot t, c)$ is reached, and
2. v is a constant pointer or v is contained in the oracle ω, so that $H^\omega_{p:c}$ does not contain unneeded, unevaluated thunks.

Definition. *For every heap $H \in \mathbb{H}$, every $p \in \mathbb{N}^*$, $c \in \mathbb{N}$ and every oracle $\omega \subseteq \mathbb{N}^*$ there is*

$$H^\omega_{p:c} := \left\{(v, e) \mid (v, e) \in H \ \wedge\ v \in \omega \cup \mathbb{V}^C \ \wedge\ v \in \mathbb{V}_{p:c}\right\}$$

With this definition, we can formulate the central theorem that relates non-strict oracle creation to Lazy Call-by-Value evaluation. It states that every judgment in a non-strict evaluation has a corresponding judgment in the respective Lazy Call-by-Value evaluation. The following diagram shows this correspondence:

$$\begin{array}{ccc} (t,\, c,\, \Gamma,\, e) & \xRightarrow[p,\gamma]{} & (t',\, c',\, \Delta,\, e') \\ \vdots & & \vdots \\ (t,\, c,\, H^{\omega}_{p\cdot t:c},\, e) & \xrightarrow[p,\omega]{} & (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, e') \end{array}$$

Lemma 10. *Let a be an initial expression with $(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\omega} Z$ for some configuration Z with $\mathcal{H}(Z) = H$. Then for all judgments $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e') \in \mathcal{D}(a)$ the judgment $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, e) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, e')$ also holds.*

Proof. By induction over the well-founded relation $\lessdot$, distinguishing the rules by which the judgment holds:

Rule $Const^L$. If $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t,\, c,\, \Gamma,\, e)$ holds by rule $Const^L$, then $\mathrm{whnf}(e)$ and $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, e) \downarrow_{p,\omega} (t,\, c,\, H^{\omega}_{p\cdot t:c},\, e)$ trivially holds by rule $Const^S$.

Rule App^L. If $(t,\, c,\, \Gamma,\, f\, v) \Downarrow_{p,\alpha} (t'',\, c'',\, \Theta,\, g)$ holds by rule App^L, then the judgments $(t,\, c,\, \Gamma,\, f) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, \lambda x.e)$ and $(t',\, c',\, \Delta,\, e[v/x]) \Downarrow_{p,\delta} (t'',\, c'',\, \Theta,\, g)$ hold as its premise for some oracles γ and δ with $\alpha = \gamma \cup \delta$.

Applying the induction hypothesis to these judgments yields $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, f) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, \lambda x.e)$ as well as $(t',\, c',\, H^{\omega}_{p\cdot t':c'},\, e[v/x]) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g)$. From these $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, f\, v) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g)$ follows by rule App^S.

Rule $VarStrict^L$. If $(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\gamma} (t,\, c,\, \Gamma,\, e)$ holds by rule $VarStrict^L$, then $v \in \mathbb{V}^C$ and $(v, e) \in \Gamma$.

Corollary 4 states that $\Gamma \trianglelefteq H$, so that $(v, e) \in H$ follows from $v \in \mathbb{V}^C$. Corollary 8 implies that $v \in \mathbb{V}_{p\cdot t:c}$, so that $(v, e) \in H^{\omega}_{p\cdot t:c}$ also holds. Therefore, the judgment $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, v) \downarrow_{p,\omega} (t,\, c,\, H^{\omega}_{p\cdot t:c},\, e)$ follows by rule Var^S.

Rule $VarLazy^L$. If $(t,\, c,\, \Gamma,\, v) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e)$ holds by rule $VarLazy^L$, then $v \in \mathbb{N}^*$, $v \in \gamma$ and $(v, e) \in \Delta$.

Corollary 4 states that $\Delta \trianglelefteq H$ holds, and $\mathrm{whnf}(e)$ follows from Observation 4; therefore $(v, e) \in H$ holds.

Observation 3 states that $v \in \omega$, and Lemma 7 states that $v \in \mathbb{V}_{p\cdot t:c}$. Therefore, $(v, e) \in H^{\omega}_{p\cdot t:c}$ also holds, so that $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, v) \downarrow_{p,\omega} (t,\, c,\, H^{\omega}_{p\cdot t:c},\, e)$ follows by rule Var^S.

Rule $Case^L$. If $(t,\, c,\, \Gamma,\, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\, \overline{x_{n,k_n}} \mapsto a_n}) \Downarrow_{p,\alpha} (t'',\, c'',\, \Theta,\, a')$ holds by rule $Case^L$, then there exists $i \in \{1, \ldots, n\}$, such that $(t,\, c,\, \Gamma,\, e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, C_i\, \overline{v_{k_i}})$ and $(t',\, c',\, \Delta,\, a_i\overline{[v_n/x_n]}) \Downarrow_{p,\delta} (t'',\, c'',\, \Theta,\, g)$ hold as its premise for some oracles γ and δ with $\alpha = \gamma \cup \delta$.

Applying the induction hypothesis to these judgments yields $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, e) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, C_i\, \overline{v_{k_i}})$ and $(t',\, c',\, H^{\omega}_{p\cdot t':c'},\, a_i\overline{[v_n/x_n]}) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, a')$. The judgment $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, \mathbf{case}\ e\ \mathbf{of}\ \overline{C_n\, \overline{x_{n,k_n}} \mapsto e_n}) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, a')$ follows by rule $Case^S$.

Rule $LetStrict^L$. If $(t,\, c,\, \Gamma,\, \mathbf{let}^!\, \overline{x_n = e_n}\, \mathbf{in} e) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e')$ holds by rule $LetStrict^L$, then $(t,\, c+1,\, \Gamma \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]}) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e')$ holds as its premise, where $v_i = (x_i)_{p\cdot t:c}$ and $e'_i = e_i\overline{[v_n/x_n]}$ for every $i \in \{1, \ldots, n\}$.

Applying the induction hypothesis yields $(t,\, c+1,\, H^{\omega}_{p\cdot t:c+1},\, e\overline{[v_n/x_n]}) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, e')$. Corollary 4 implies that $\Gamma \cup \{\overline{(v_n,\, e'_n)}\} \trianglelefteq H$, so there is $H^{\omega}_{p\cdot t:c} \cup \{\overline{(v_n,\, e'_n)}\} \subseteq H^{\omega}_{p\cdot t:(c+1)}$.

If there were other heap entries in $H^{\omega}_{p\cdot t:(c+1)} \setminus H^{\omega}_{p\cdot t:c}$ than $v_1, \ldots, v_n$, then there would be another jugdment with the same current position. This is ruled out by Observation 7, so all constant identifiers with suffix $p\cdot t{:}c$ that are bound in H come from the current judgment.

Therefore, $H^{\omega}_{p\cdot t:(c+1)} = H^{\omega}_{p\cdot t:c} \cup \{\overline{(v_n,\, e'_n)}\}$ holds. With that the judgment $(t,\, c+1,\, H^{\omega}_{p\cdot t:c} \cup \{\overline{(v_n,\, e'_n)}\},\, e\overline{[v_n/x_n]}) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, e')$ holds, and $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, \mathbf{let}^!\, \overline{x_n = e_n}\, \mathbf{in}\, e) \downarrow_{p,\omega} (t',\, c',\, H^{\omega}_{p\cdot t':c'},\, f)$ follows by rule $LetStrict^S$.

Rule $LetLazy^L$. If $I := (t,\, c,\, \Gamma,\, \mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2) \Downarrow_{p,\gamma} (t'',\, c'',\, \Theta,\, g)$ holds by rule $LetLazy^L$, then $(t+1,\, 0,\, \Gamma,\, e_2[p\cdot t/x]) \Downarrow_{p,\gamma} (t'',\, c'',\, \Theta,\, g)$ holds as its premise.

Applying the induction hypothesis yields

$$(t+1,\, 0,\, H^{\omega}_{p\cdot(t+1):c},\, e_2[p\cdot t/x]) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g) \qquad (*)$$

Now two cases have to be distinguished:

1. $p\cdot t \notin \omega$. By Observation 6, heap H contains no thunk pointers with prefix $p\cdot t$ and no constant pointers whose subscript has prefix $p\cdot t$. Also, Observation 7 implies that all constant identifiers with suffix $p\cdot t{:}c$ that are bound in H come from the current judgment. Therefore, $H^{\omega}_{p\cdot t:c} = H^{\omega}_{p\cdot(t+1):0}$ holds,so that (*) implies that $(t{+}1,\, 0,\, H^{\omega}_{p\cdot t:c},\, e_2[p\cdot t/x]) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g)$ holds.

   From this $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, \mathbf{let}^?\, x = e_1\, \mathbf{in}\, e_2) \downarrow_{p,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g)$ follows by rule LetSkip.

2. $p \cdot t \in \omega$. Then Observation 5 states that there is a judgment $J := (0,\, 0,\, G,\, e_1) \Downarrow_{p\cdot t,\,\delta} (t',\, c',\, D,\, f) \in \mathcal{D}(a)$. If whnf$(e_1)$, then the judgment

$$(0,\, 0,\, H^{\omega}_{p\cdot t\cdot 0:0},\, e_1) \downarrow_{p\cdot t,\,\omega} (t',\, c',\, H^{\omega}_{p\cdot t\cdot t':c'},\, f) \qquad (**)$$

holds by rule $Const^S$. Otherwise there is $J \sqsubset I$ and therefore $J \lessdot I$, so that (**) holds by the induction hypothesis.
Corollary 8 states that $\mathcal{V}(e_1) \subseteq \mathbb{V}_{p\cdot t:c}$ and no entry of $H^{\omega}_{p\cdot t:c}$ refers to $H^{\omega}_{p\cdot t\cdot 0:0} \setminus H^{\omega}_{p\cdot t:c}$. Therefore,

$$(0,\, 0,\, H^{\omega}_{p\cdot t\cdot 0:c},\, e_1) \downarrow_{p\cdot t,\,\omega} (t\cdot t',\, c',\, \Lambda,\, f) \qquad (***)$$

with $\Lambda = H^{\omega}_{p\cdot t\cdot t':c'} \setminus (H^{\omega}_{p\cdot t\cdot 0:0} \setminus H^{\omega}_{p\cdot t:c})$.
Since $H^{\omega}_{p\cdot t\cdot 0:0} \setminus H^{\omega}_{p\cdot t:c} = \{(p\cdot t, f)\}$ and $H^{\omega}_{p\cdot t\cdot t':c'} \cup \{(p\cdot t, f)\} = H^{\omega}_{p\cdot (t+1):0}$, there is $\Lambda \cup \{(p\cdot t, f)\} = H^{\omega}_{p\cdot (t+1):0}$.
Thus, (*) states that

$$(t+1,\, 0,\, H^{\omega}_{p\cdot t\cdot t':c'} \cup \{(p\cdot t, f)\},\, e_2[p\cdot t/x]) \downarrow_{p,\,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g) \qquad (****)$$

Finally, $(t,\, c,\, H^{\omega}_{p\cdot t:c},\, \mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2) \downarrow_{p,\,\omega} (t'',\, c'',\, H^{\omega}_{p\cdot t'':c''},\, g)$ follows from (***) and (****) by rule $LetEval^S$

□

Now we have everything at hand to prove the correctness property from the beginning of this section.

Theorem 11 (Correctness)**.** *For every initial expression a there is:*

$$(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\,\omega} (t,\, c,\, H,\, z) \implies (0,\, 0,\, \emptyset,\, a) \downarrow_{\epsilon,\,\omega} (t,\, c,\, H|_{\mathbb{V}^C \cup \omega},\, z)$$

Proof. Assume that $(0,\, 0,\, \emptyset,\, a) \Downarrow_{\epsilon,\,\omega} (t,\, c,\, H,\, z)$ holds. Then Lemma 10 states that $(0,\, 0,\, \emptyset,\, a) \downarrow_{\epsilon,\,\omega} (t,\, c,\, H^{\omega}_{p\cdot t:c},\, z)$ also holds. Lemma 9 implies that $H|_{\mathbb{V}^C \cup \omega} \subseteq \mathbb{V}_{p\cdot t:c}$, and so $H^{\omega}_{p\cdot t:c} = H|_{\mathbb{V}^C \cup \omega}$. Therefore, $(0,\, 0,\, \emptyset,\, a) \downarrow_{\epsilon,\,\omega} (t,\, c,\, H|_{\mathbb{V}^C \cup \omega},\, z)$ holds. □

Chapter 4

Implementation of Oracle Generation

4.1 Implementing Oracles as Lists

In the semantic model presented in the last chapter, oracles are represented by sets of sequences of natural numbers. A naïve implementation that actually uses such sets as oracles would be far too inefficient for the evaluation of real-world programs. Therefore, another representation for oracles is needed.

In [Braßel et al., 2007] oracles are implemented as lists that contain oracle entries in the same order in which they are consumed by the strict evaluator; it only needs to inspect the first list entry in order to decide whether the next redex shall be skipped. Additionally, these lists are compressed, so that they contain natural numbers instead of Boolean values: a number n means that the next n redexes are needed, followed by one redex that may be skipped.[1]

We will use the same representation for oracles, but due to the different semantic model we have to find another way of creating oracle list.

First, we have to make sure, that the oracle sets of the semantic model can indeed be represented as lists of Boolean values. For this we need the notion of the *current position*, as defined in the last chapter. When oracles are represented as sets, it represents the current oracle entry; when oracles are represented by lists, it serves as a pointer to the current list node. For an initial expression a with $(0,\, 0,\, \emptyset,\, a) \Downarrow_{p,\omega} Z$ there is:

- Every element $v \in \omega$ is a thunk pointer in $\mathrm{dom}\mathcal{H}(Z)$: Only rule $\mathit{VarLazy}^L$ can add an entry to the oracle, but then it also updates

[1] This compression scheme is efficient as long as the number of needed redexes is much higher than the number of unneeded redexes. In comparison to [Braßel et al., 2007], the semantics presented here can be expected to create oracles with fewer positive entries (cf. Section 3.2 on page 24). Further investigation is needed to decide whether switching to another compression scheme could gain some efficiency.

the corresponding thunk in heap $\mathcal{H}(Z)$.

- Every thunk pointer in dom$\mathcal{H}(Z)$ is introduced by an application of rule *LetLazy*^{L}, for which Theorem 10 on page 51 states that there is a corresponding application of rule *LetEval*^{S}or *LetSkip*^{S}in the Lazy Call-by-Value derivation $(0,\ 0,\ \emptyset,\ a)\ \downarrow_{p,\omega}\ Z'$. Thus, for every thunk pointer $p \in \text{dom}\mathcal{H}(Z)$ the Lazy Call-by-Value evaluator checks whether p is also an element of ω.

- Observation 8 states that in the Lazy Call-by-Value evaluation the current position is constantly increasing with respect to $\preccurlyeq$, so that the thunk pointers in dom$\mathcal{H}(Z)$ are checked in lexicographic order.

Thus, the oracle can be represented by a list of Boolean values. The nodes of this list correspond to the thunk pointers in dom$\mathcal{H}(Z)$, sorted in lexicographic order. The value of a node that corresponds to a thunk pointer p corresponds to the validity of the statement $p \in \gamma$.

Next, concrete operations that create and modify oracle lists have to be derived from the semantics of non-strict oracle creation. An inspection of the rules in figure 3.5 on page 32 reveals that:

1. Every judgment $(t,\ c,\ \Gamma,\ e)\ \Downarrow_{p,\gamma}\ (t',\ c',\ \Delta,\ e')$ allows its subcomputations to contribute entries to the resulting oracle set, but not to remove any entries. Therefore, it is not necessary to synthesize an oracle set by joining the oracle sets of the current judgments subcomputation. Instead, an oracle list can be calculated by letting the subcomputations modify an initial oracle list.

2. Rule *LetLazy*^{L} binds one thunk on the heap and leaves the oracle set unchanged; the name of that binding is equal to the current position of the judgment proven by rule *LetLazy*^{L}.

   Thus, whenever rule *LetLazy*^{L} is applied, one entry has to be inserted in the oracle list at the current position. This entry must contain the Boolean value *false*, because at this time the oracle set does not contain any entry that corresponds to the newly bound thunk.

   No other rule adds bindings to the heap, so no other rule may add entries to the oracle list.

3. Rule *VarLazy*^{L} adds an entry to the oracle, so the corresponding entry of the oracle list has to be set to the value *true*. Since no other rule adds entries to the oracle set, no other rule may change entries of the oracle list.

   The element added to the oracle set is equal to the variable that is being evaluated, which in turn is equal to the current position of the judgment in the premise of rule *VarLazy*^{L}.

Thus, when the evaluation of a variable v starts the current position is changed to v. Then the value of the current oracle entry is set to *true* and the expression bound to v is evaluated. After that the current position is set back to its previous value, and the result is returned.

4.2 Instrumenting Haskell Code

In order to implement this behavior as a source-to-source transformation of Haskell programs, we have to make some design decisions:

- Oracle lists are implemented as doubly linked lists, so that nodes can be efficiently inserted and removed at arbitrary positions.
- Oracle lists are represented by circular lists that contain an additional *start node*. The end of an oracle list is linked to its start node, so that no special treatment of list heads and tails needs to be implemented.

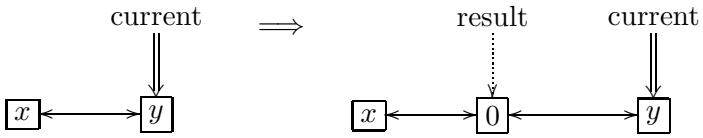
  When a non-strict evaluation starts, the oracle consists of the start node that is linked to itself. A consequence is that every oracle list will contain an additional entry. It turns out that this entry is always located at the end of the oracle list, where it can silently be ignored.
- The start node and the pointer to the current position of the oracle list are implemented as mutable global states. This overcomes the need to thread a pointer to the current node through the program by handing it over as an argument. The downside is that special care has to be taken when the instrumented program is optimized, or when multiple instances of an instrumented program that are running in parallel share the same set of global variables.[2]
- We let the current node pointer refer to the successor node of the current position, so that new entries can be inserted without changing the current position.
- Oracle lists are lists of natural numbers, where a number n stands for n needed redexes, followed by one redex that may be skipped.

Interface

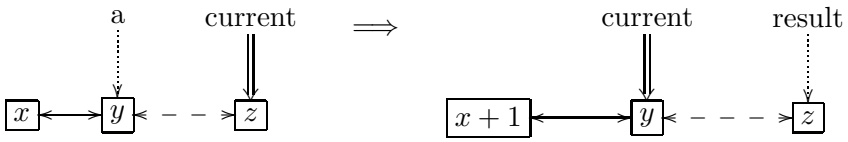
This functionality can be implemented in three side-effecting functions:

[2] In chapter 5 it is described how a program can be transformed into a monadic version that maintains global state in a purely functional way. When the restrictions imposed by the use of global variables become an issue, that method can easily be adapted to non-strict oracle creation.

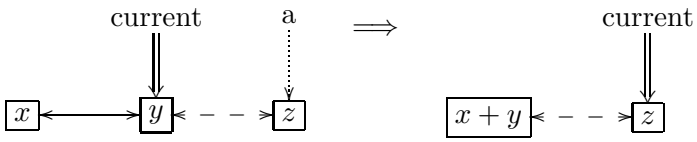
- A parameterless function *newOracleEntry* that returns a pointer to an oracle node. As a side effect it allocates a new oracle node and inserts it into the oracle list between the current oracle node and its predecessor. The returned value is a pointer to the newly created node:



- A function *enterRhs* of type $OraclePtr \rightarrow OraclePtr$. It sets the current node pointer to the argument node a, adds 1 to the value of the current node's predecessor, and returns the previos value of the current node pointer as its result:



- A function *leaveRhs* that takes an oracle node pointer as argument and returns the unit expression () as result. It unlinks the node pointed to by the current oracle pointer from the oracle list and adds its value to its predecessor. Then it sets the current oracle pointer to the argument node a:



Additionally, a function *initOracle* is needed to initialize the start node and the current oracle pointer, and a function *opOracle* is needed that reads the values of the created oracle nodes, so that they can be used in the Lazy Call-by-Value evaluation.

Transformations

Haskell programs can be instrumented with these functions, so that they generate oracle lists as a side effect. Comparing the rules for lazy evaluation given in Figure **??** on page ?? to the rules for non-strict oracle creation given in Figure 3.1 on page 22 reveals that they differ only in the evaluation of let-bound values and their declarations: The rules for non-strict oracle creation

add oracle entries and increment the current position, but they compute the same results as the rules for lazy evaluation. The other rules for the non-strict oracle creation are identical to the rules for lazy evaluation, except that they maintain oracle entries and the values that are used to invent new identifiers. Thus, only variable declarations have to be instrumented and all other program constructs can be left unchanged.

Strict declarations. For a declaration **let** $x_1 = e_1 \,...\, x_n = e_n$ **in** e with all $e_1 \ldots e_n$ in weak head normal form, the lazy evaluation rule is identical to the Lazy Call-by-Value rule for the corresponding $\lambda^{?!}$ declaration $\mathbf{let}^{!}$ $x_1 = e_1 \,...\, x_n = e_n$ **in** e. Therefore, there is no need to transform this declaration; we simply keep the original Haskell declaration, instrumenting only the right hand sides $e_1 \ldots e_n$ and the body expression e.

Non-strict declarations. A non-strict, non-recursive Haskell declaration

$$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$$

where x does not occur free in e_1 is translated to the $\lambda^{?!}$ expression

$$\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2.$$

When e_1 is bound on the heap, an oracle entry has to be created by function *newOracleEntry*. When the value bound to x is requested, the current position is set to this oracle entry by function *enterRhs*, and the previous value is saved as o_{save}. Then expression e_1 is evaluated. Finally, the current position is set back to o_{save} by function *leaveRhs*, and the result is returned. This can be implemented by transforming the above declaration into the following Haskell code:[3]

```
let ! o = newOracleEntry
in let x = let ! o_save = enterRhs o
           in let ! result = e1
              in leaveRhs o_save `seq` result
   in e2
```

Applications. In language $\lambda^{?!}$, functions can only be applied to identifiers. But in Haskell functions can be applied to all kinds of expressions. Thus, before an application $e_1\ e_2$ can be instrumented, it must be translated to an expression

$$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ x$$

[3]We use *bang patterns* to indicate that the bound value e_1 has to be reduced to weak head normal forms, before the body e_2 is evaluated; the expression $\mathbf{let}\,!\,x = e_1$ **in** e_2 is an abbreviation for the Haskell expression **let** $x = e_1$ **in** x `` `seq` `` e_2.

with a fresh identifier x. After this expression is instrumented, the definition of x can be inlined, so that the newly introduced identifier x is not needed anymore:

$$\begin{array}{l}\textbf{let}\ !\ o = \mathit{newOracleEntry}\\ \textbf{in}\ e_2\ (\textbf{let}\ !\ o_{save} = \mathit{enterRhs}\ o\\ \qquad \textbf{in let}\ !\ \mathit{result} = e_1\\ \qquad\quad \textbf{in}\ \mathit{leaveRhs}\ o_{save}\ `\mathit{seq}`\ \mathit{result})\end{array}$$

When applications are directly transformed from Haskell to this code, there is no need to invent new identifiers.

4.3 Transforming Cyclic Declarations

In Chapter 2 it has been shown that there are situations where recursive declarations have to be transformed into non-recursive declarations.

This section discusses several ways of transforming a Haskell expression

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

in which expression e_1 refers to variable x into an expression without recursive references to x.

Using a fixed point combinator

As mentioned in Chapter 2, a recursive declaration $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ is equivalent to the expression $\textbf{let}\ x = \mathit{fix}\ (\lambda x \rightarrow e_1)\ \textbf{in}\ e_2$, where *fix* is a least fixed point combinator:

$$\mathit{fix}\ f = f\ (\mathit{fix}\ f)$$

One disadvantage of this approach is that a call to function *fix* is introduced for every recursive declaration. Another disadvantantage is that there are two declarations that have to be instrumented: one in function *fix* and one in the transformed declaration.

Using self-application

A way to model recursive declarations without the help of a fixed point combinator is to apply a function to itself; this way a recursive declaration $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ can be transformed into an entirely non-recursive declaration:

$$\begin{array}{l}\textbf{let}\ x' = \lambda x' \rightarrow \textbf{let}\ x = x'\ x'\ \textbf{in}\ e_1\\ \textbf{in}\ \ \textbf{let}\ x = x'\ x'\\ \qquad \textbf{in}\ e_2\end{array}$$

This way the call to function *fix* is omitted, but now there are two declarations that have to be instrumented *per declaration*.

There is also a pitfall: When a Haskell program is optimized, definitions are inlined at the call site. In order to ensure termination of the compilation process, the compiler has to know when to stop inlining recursively defined functions inside their own definition. In Haskell, recursion always comes from recursive **let**-expressions, because the type system of Haskell rules out self-application of functions.

When a transformation like this introduces self-application at some compilation stage, then recursion is no longer explicit in the subsequent optimisation process; if no special care is taken, the optimizer may run into an endless loop. For example, the inlining mechanism of the Glasgow Haskell Compiler as described in [Jones and Marlow, 1999] requires recursion to be explicit.

Using a dummy parameter

Since we allow for recursive function declarations, there is no need to remove recursive declarations entirely: it is sufficient to turn recursive data declarations into recursive function declarations. This can be achieved by adding a "dummy" parameter to the function definition. By using Haskell's unit expression () as a dummy parameter, a recursive declaration **let** $x = e_1$ **in** e_2 can be transformed to the following expression:

$$\begin{array}{l}\textbf{let } x' = \lambda() \rightarrow \textbf{let } x = x' \ () \\ \qquad\qquad\quad \textbf{in } e_1 \\ \textbf{in let } x = x' \ () \\ \quad \textbf{in } e_2\end{array}$$

Under some circumstances this approach turns out to be too simple: an optimizing Haskell compiler may remove the unneeded argument (), changing the transformed expression back to the original form.[4] Thus, special care has to be taken when optimisations are applied after this transformation step.

Using oracles as parameter

It turns out that there is a simple way to replace dummy parameters by parameters that are actually needed. Instrumenting the above expression results in the following code:

$$\begin{array}{l}\textbf{let } x' = \lambda() \rightarrow \textbf{let } ! \ o = \mathit{newOracleEntry} \\ \qquad\qquad\quad \textbf{in let } x = \textbf{let } ! \ o_{\mathit{save}} = \mathit{enterRhs} \ o \\ \qquad\qquad\qquad\quad \textbf{in let } ! \ \mathit{result} = x' \ ()\end{array}$$

[4] This has been observed with GHC 6.8.2 with option `-O2` turned on.

```
                                    in leaveRhs o_save `seq` result
                        in e_1
in let ! o = newOracleEntry
   in let x = let ! o_save = enterRhs o
              in let ! result = x' ()
                 in leaveRhs o_save `seq` result
      in e_2
```

It is easy to see that both declarations of variable x have identical right hand sides; they can be abstracted out into a function x'' that takes an oracle entry o as parameter. Doing so gives the following code:

```
let x'' = λo → let ! o_save = enterRhs o
               in let ! result = x' ()
                  in leaveRhs o_save `seq` result
    x' = λ() → let ! o = newOracleEntry
               in let x = x'' o
                  in e_1
in let ! o = newOracleEntry
   in let x = x'' o
      in e_2
```

Now function x' is referred to only once, so the obvious thing to do is to inline x' at its call site:

```
let x'' = λo → let ! o_save = enterRhs o
               in let ! result = let ! o = newOracleEntry
                                 in let x = x'' o
                                    in e_1
                  in leaveRhs o_save `seq` result
in let ! o = newOracleEntry
   in let x = x'' o
      in e_2
```

By two small refactorings we have overcome the need to invent dummy parameters. Now every application of the recursive function uses distinct oracle entries as its parameter, so that they cannot be turned back into a recursive declaration. Compared to the previous attempts there is also less code duplication: the only function that is called twice per declaration is function *newOracleEntry*.

4.4 A Small Example

Consider the following Haskell expression:

$$
\begin{array}{l}
\textbf{let } succ\ x = S\ x \\
\quad\ \ pred\ (S\ y) = y \\
\quad\ \ x = Z \\
\quad\ \ y = succ\ x \\
\textbf{in } pred\ (succ\ x)
\end{array}
$$

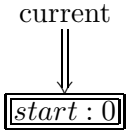
By breaking up the recursive declaration as described in Chapter 2 and transforming the resulting expression to language $\lambda^{?!}$ the following expression is obtained:

$$
\begin{array}{l}
\textbf{let}^{!}\ succ = \lambda x.S\ x \\
\quad\ \ pred = \lambda x.\textbf{case } x \textbf{ of } S\ y.y \\
\textbf{in let}^{!}\ x = Z \\
\quad \textbf{in let}^{?}\ y = succ\ x \\
\qquad \textbf{in } pred\ (succ\ x)
\end{array}
$$

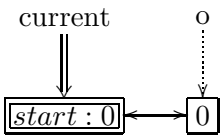
Applying the transformation scheme described above results in the following Haskell code:

$$
\begin{array}{l}
\textbf{let } !\ succ = \lambda x \rightarrow S\ x \\
\quad\ \ !\ pred = \lambda x \rightarrow \textbf{case } x \textbf{ of } S\ y \rightarrow y \\
\textbf{in let } !\ x = Z \\
\quad \textbf{in let } !\ o = newOracleEntry \\
\qquad \textbf{in let } y = \textbf{let } !\ o' = enterRhs\ o \\
\qquad\qquad\qquad \textbf{in let } !\ result = succ\ x \\
\qquad\qquad\qquad\quad \textbf{in } leaveRhs\ o'\ `seq`\ result \\
\qquad\quad \textbf{in let } !\ p = newOracleEntry \\
\qquad\qquad \textbf{in } pred\ (\textbf{let } !\ p' = enterRhs\ p \\
\qquad\qquad\qquad\quad \textbf{in let } !\ result = succ\ x \\
\qquad\qquad\qquad\qquad \textbf{in } leaveRhs\ p'\ `seq`\ result)
\end{array}
$$

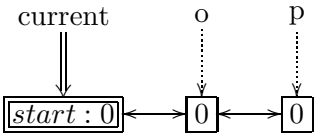
When the evaluation starts, the oracle contains only one node, the start node. Initially, the current node pointer points to the start node:



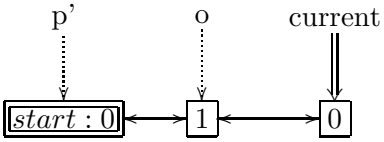
Before y is bound on the heap the first call to *newOracleEntry* occurs. It inserts one new node on the left side of the current node:[5]



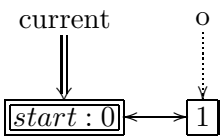
After y has been bound on the heap, function *newOracleEntry* is called a second time. It creates an oracle entry that will keep track of the evaluation of the argument of function *pred*. A pointer to that node is bound to p:



Then function *pred* is called. It forces the evaluation of its argument, so that function *enterRhs* is called with the node pointer p as its argument. It makes p the current oracle node, increments the value of its left neighbour and returns a pointer to the previously current oracle. which is bound to p':



Finally, the expression *leaveRhs* p' is evaluated. It removes the current list node and restores the previus value of the current oracle pointer:



[5]It may seem odd that the new node appears on the right side of the current node, but in this situation the right side of the singular start node is also its left side, because oracles are implemented as circular list.

The resulting oracle list has two entries. They encode three Boolean values, of which the last is an unused artefact coming from the additional start node:

$$[0, 1] \cong [\mathit{False}, \mathit{True}, \mathit{False}]$$

An implementation of the stateful oracle-manipulating functions can be found in appendix B.3 on page 106.

Chapter 5

Debugging by Asking the Oracle

5.1 Lazy Call-by-Value Evaluation

In the non-strict oracle creation described in the last chapter a global state is used to hold a pointer to the current oracle entry. But declarative debugging by Lazy Call-by-Value evaluation requires that subexpressions can be evaluated more than once, and the presence of mutable state may complicate this. Therefore, it is preferable to avoid the use of global variables.

In Haskell it has become common practice to use monads for modelling mutable state in a purely declarative way. A variant of this monad that takes lists of integers as its state is formed by an operator ST on types together with two functions *return* and *bind*:

$$\begin{aligned}
&\textbf{type } ST\ a = [Integer] \to ([Integer], a)\\
&return :: a \to ST\ a\\
&return\ x = \lambda s \to (s, x)\\
&(\gg\!=) :: ST\ a \to (a \to ST\ a) \to ST\ a\\
&x \gg\!= f = \lambda s \to \textbf{case } x\ s \textbf{ of } (s', v) \to f\ v\ s'
\end{aligned}$$

In this monad, a value that has type a is represented by a function that has type $[Integer] \to ([Integer], a)$; this function takes a state as parameter and returns a tuple containing the updated state and the value of type a.[1]

Two of the three *monad laws* will turn out to be useful for the purpose of this section. They say that the following equations hold for all expressions f and x:

$$\begin{array}{rcll}
return\ x \gg\!= f & = & f\ x & \text{Left identity law}\\
(m \gg\!= f) \gg\!= g & = & m \gg\!= (\lambda x \to f\ x \gg\!= g) & \text{Associativity law}
\end{array}$$

[1] We use this explicit definition of a state monad, so that we can explore the potential for program optimisations. In Haskell, one might simply reuse the existing implementation in module *Control.Monad.State.Strict*.

Monads can also be used to enforce a particular order of evaluation. In [Wadler, 1990], a translation scheme is given that transforms an expression into a monadic version that is evaluated in strict order. We will use this translation scheme to enforce strict evaluation order.

For an expression e of type E we denote the result of "lifting" e into the state monad by e^*. The lifted expression e^* has type $ST\ E^*$, where the transformation of types is defined by the following rules:

$$\begin{array}{ll} K^* & \equiv K \quad \text{if } K \text{ is a base type} \\ (U \to V)^* & \equiv U^* \to ST\ V^* \\ (U, V)^* & \equiv (U^*, V^*) \end{array}$$

These rules transform an expression of type E into a monadic expression of type $ST\ E^*$:[2]

$$\begin{array}{ll} x^* & \equiv \mathit{return}\ x \quad \text{if } x \text{ is an identifier} \\ (\lambda x \to e)^* & \equiv \mathit{return}\ (\lambda x \to e^*) \\ (t\ u)^* & \equiv t^* \gg\!= \lambda f \to u^* \gg\!= f \\ (u, v)^* & \equiv u^* \gg\!= \lambda x \to v^* \gg\!= \lambda y \to \mathit{return}\ (x, y) \end{array}$$

They can be used directly to transform identifiers and abstractions of language $\lambda^{?!}$ into monadic Haskell expressions. But we still have to give rules for strict and non-strict declarations, constructor terms and case expressions:

Applications. The translation rule for applications can be simplified: in language $\lambda^{?!}$ functions are only applied to identifiers, so the left identity law can always be applied:

$$\begin{array}{ll} (e\ x)^* & \equiv e^* \gg\!= \lambda f \to \mathit{return}\ x \gg\!= f \\ & = e^* \gg\!= \lambda f \to f\ x \end{array}$$

This rule still contains an inefficiency: a newly introduced abstraction $\lambda f \to f\ x$ is handed over to the operator $\gg\!=$. In order to avoid this, a function *funApp* is introduced:

$$\begin{array}{l} \mathit{funApp} :: ST\ (a \to ST\ b) \to a \to ST\ b \\ \mathit{funApp}\ e\ x = \lambda s \to \mathbf{case}\ e\ s\ \mathbf{of}\ (s', v) \to v\ x\ s' \end{array}$$

Now the transformation rule can be simplified to:

$$(e\ x)^* \equiv \mathit{funApp}\ e^*\ x$$

In some situations function *funApp* can even be replaced by plain function application: If $e^* = \mathit{return}\ f$ for some expression f, then the

[2] In the original presentation, monad comprehensions the right hand sides are formulated in terms of monad comprehensions.

left identity law can be applied, yielding a simplified version of the transformation rule:

$$\begin{aligned}(e\ x)^* &\equiv \mathit{return}\ f \gg\!= \lambda f \to f\ x \\ &= f\ x\end{aligned}$$

Strict declarations. In a declaration $\mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e$ all right hand sides are in weak head normal form, i.e. they are constructor terms or abstractions. Thus, for every e_i there is a v_i, such that $e_i^* = \mathit{return}\ v_i$. We simply translate the above declaration to

$$\mathbf{let}\ \overline{x_n = v_n}\ \mathbf{in}\ e^*$$

When this transformation is applied, the types of the right hand sides change in the same way as the types of the variables that refer to them, so that well-typedness is preserved. Strict evaluation order is also preserved, because the defined values are already in weak head normal form, so that only the body of the declaration needs to be evaluated.

If $e^* = \mathit{return}\ v$ for some expression v, then this rule can be modified, giving opportunities for simplifications in other rules:[3]

$$(\mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e)^* \equiv \mathit{return}\ (\mathbf{let}\ \overline{x_n = e_n^*}\ \mathbf{in}\ v)$$

Non-strict declarations. A declaration $\mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2$ is equivalent to the expression $(\lambda x \to e_2)\ e_1$, whose lifted form is

$$e_1^* \gg\!= \lambda x \to e_2^*$$

Lazy Call-by-Value evaluation lets the oracle decide whether non-strict declarations bind a value on the heap or not: If the current oracle entry is a positive number, then this number is decreased by one, the value to be bound is calculated and bound on the heap, and finally the body of the declaration is evaluated (cp. section 4.1). If the current oracle entry has value zero, then this entry is removed from the oracle, the value to be bound is not calculated but replaced by some placeholder value, and finally the body of the declaration is evaluated.

[3] When the modified rule is applied to the right hand side of a non-strict declaration, then the resulting expression will create an unevaluated thunk at run-time. Creating unevalated thunks violates the principle of strict evaluation. But this thunk will contain only nested strict let-bindings around a constructor term, abstraction or identifier. Thus, we make the — somewhat questionable — choice not to consider the binding of constant values as "real work" and allow this optimisation nevertheless.

Function *optional* inspects and modifies the current oracle entry and decides whether its argument is evaluated or replaced by a placeholder value *underscore*:

$$
\begin{aligned}
&\mathit{underscore} = \bot\\
&\mathit{optional} :: \mathit{ST}\ a \to \mathit{ST}\ a\\
&\mathit{optional}\ \_\ (0 : \mathit{orc}) = \mathit{return}\ \mathit{underscore}\ \mathit{orc}\\
&\mathit{optional}\ x\ (n : \mathit{orc}) = x\ (n-1 : \mathit{orc})
\end{aligned}
$$

This function is used to instrument the lifted versions of non-strict declarations, so that they follow the Lazy Call-by-Value semantics:

$$(\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2\ )^* \equiv \mathit{optional}\ e_1^* \gg\!= \lambda x \to e_2^*$$

If $e_1^* = \mathit{return}\ v$ holds for some expression v, then there is an opportunity to apply the left identity law,[4] yielding an optimized translation rule. For this we introduce a function *optionalApp* that takes values of type a and $a \to \mathit{ST}\ a$ as its parameter:

$$
\begin{aligned}
&\mathit{optionalApp} :: a \to (a \to \mathit{ST}\ b) \to \mathit{ST}\ b\\
&\mathit{optionalApp}\ \_\ q\ (0 : \mathit{orc}) = q\ \mathit{underscore}\ \mathit{orc}\\
&\mathit{optionalApp}\ p\ q\ (n : \mathit{orc}) = q\ p\ (n-1 : \mathit{orc})
\end{aligned}
$$

The function call $\mathit{optionalApp}\ \ p\ \ q$ is equivalent to the expression $\mathit{optional}\ (\mathit{return}\ \ p) \gg\!= q$, so that an optimized transformation rule can be given:

$$(\mathbf{let}^{?}\ x = e_1\ \mathbf{in}\ e_2\ )^* \equiv \mathit{optionalApp}\ v\ (\lambda x \to e_2^*)$$

Constructor terms. The rule for constructor terms is a simplified variant of the rule for tuples. If u and v are indentifiers, then lifting the tuple (u, v) leads to the expression $\mathit{return}\ u \gg\!= \lambda x \to \mathit{return}\ v \gg\!= \lambda y \to \mathit{return}\ (x, y)$. By applying the left identity rule twice, this expression is transformed to the expression $\mathit{return}\ (u, v)$. In language $\lambda^{?!}$, all components of constructor terms are identifiers, so we translate arbitrary constructor terms the same way:

$$(C\ v_1 \ldots v_n)^* \equiv \mathit{return}\ (C\ v_1 \ldots v_n)$$

Case expressions. The rule for case expressions is straightforward:

$$(\mathbf{case}\ u\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n})^* \equiv u^* \gg\!= \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n^*}$$

[4] This opportunity is rare, because we have decided not to allow constructors, abstractions and identifiers on the right hand side of non-strict declarations; it will only occur after the optimized rule for strict declarations has been applied to e_1.

$$
\begin{array}{lcl}
x^* & \equiv & \textit{return}\ x \text{ if } x \text{ is an identifier} \\
(\lambda x.e)^* & \equiv & \textit{return}\ (\lambda x \to e^*) \\
(t\ u)^* & \equiv & \begin{cases} v\ u & \text{if } t^* = \textit{return}\ v \\ \textit{funApp}\ t^*\ u & \text{otherwise} \end{cases} \\
(\mathbf{let}^!\ \overline{x_n = e_n}\ \mathbf{in}\ e)^* & \equiv & \begin{cases} \textit{return}\ (\mathbf{let}\ \overline{x_n = v_n}\ \mathbf{in}\ v^*) & \text{if } e^* = v \\ \mathbf{let}\ \overline{x_n = v_n}\ \mathbf{in}\ e^* & \text{otherwise} \end{cases} \\
 & & \text{where } \overline{e_n^* = \textit{return}\ v_n} \\
(\mathbf{let}^?\ x = e_1\ \mathbf{in}\ e_2)^* & \equiv & \begin{cases} \textit{optionalApp}\ v\ (\lambda x \to e_2^*) & \text{if } e_1^* = \textit{return}\ v \\ \textit{optional}\ e_1^* \gg\!= e_2^* & \text{otherwise} \end{cases} \\
(C\ v_1 \ldots v_n)^* & \equiv & \textit{return}\ (C\ v_1 \ldots v_n) \\
(\mathbf{case}\ u\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n})^* & & \\
\qquad \equiv \begin{cases} \mathbf{case}\ v\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n^*} & \text{if } u^* = v \\ u^* \gg\!= \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n^*} & \text{otherwise} \end{cases} &&
\end{array}
$$

Figure 5.1: Translation scheme for Lazy Call-by-Value evaluation

If $u^* = \textit{return}\ v$ for some expression v, then this rule can be simplified by applying the left identity law:

$$
\begin{array}{l}
(\mathbf{case}\ u\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n})^* \\
\quad \equiv \textit{return}\ v \gg\!= \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n^*} \\
\quad = \mathbf{case}\ v\ \mathbf{of}\ \overline{C_n\ \overline{x_{n,k_n}} \mapsto a_n^*}
\end{array}
$$

If all right hand sides are identifiers, abstractions or constructor terms, then a similar optimisation rule as the one for strict declarations can be introduced. We refrain from that, because this optimisation may cause thunks to be created that contain unevaluated case-expressions.

Figure 5.1 shows the translation scheme for monadic Lazy Call-by-Value evaluation.

5.2 Transforming Cyclic Declarations

In the implementation of non-strict oracle creation, cyclic data declarations were eliminated by turning them into cyclic function declarations; then these functions were instrumented with the capability to create oracle lists (cp. section 4.3 on page 60). When implementing Lazy Call-by-Value evaluation, care must be taken that cyclic data declarations are transformed the same way; otherwise creation and consumption of oracle list may get out of sync.

Therefore, an expression of the form **let** $x = e_1$ **in** e_2, with x free in e_1 and e_1 not in weak head normal form, is first transformed into the following expression:

$$\begin{array}{l}\textbf{let}^{!}\ x' = \lambda() \to \textbf{let}^{?}\ x = x'\ ()\ \textbf{in}\ e_1\\ \textbf{in let}^{?}\ x = x'\ ()\ \textbf{in}\ e_2\end{array}$$

Then the transformation scheme is applied to this expression:

$$\begin{array}{l}\textbf{let}\ x' = \lambda() \to \mathit{optional}\ (x'\ ()) \gg\!= (\lambda x \to e_1^*)\\ \textbf{in}\ \mathit{optional}\ (x'\ ()) \gg\!= (\lambda x \to e_2^*)\end{array}$$

Code duplication is removed by abstracting out $\mathit{optional}\ (x'\ ())$ into a function r and inlining x' there:

$$\begin{array}{l}\textbf{let}\ r = \mathit{optional}\ (r \gg\!= (\lambda x \to e_1^*))\\ \textbf{in}\ r \gg\!= (\lambda x \to e_2^*)\end{array}$$

Inlining function *optional* and turning the abstraction $\lambda x \to e_1^*$ into a locally defined function yields the following expression:

$$\begin{array}{l}\textbf{let}\ r\ (0 : \mathit{orc}) = \mathit{return}\ \mathit{underscore}\ \mathit{orc}\\ \quad r\ (n : \mathit{orc}) = (r \gg\!= f)\ (n-1 : \mathit{orc})\\ \quad f\ x = e_1^*\\ \textbf{in}\ r \gg\!= (\lambda x \to e_2^*)\end{array}$$

Now it can be seen that the function call $r\ (n : \mathit{orc})$ is equivalent to the expression

$$(((\mathit{return}\ \mathit{underscore} \gg\!= f) \gg\!= ...f) \gg\!= f)\ \mathit{orc}$$

where f is applied n times to *return underscore*. By repeated application of the associativity law this expression is transformed to

$$(\mathit{return}\ \mathit{underscore} \gg\!= (\lambda x \to f\ x \gg\!= ...(\lambda x \to f\ x \gg\!= f)))\ \mathit{orc}$$

Using this insight, the expression can be rewritten as follows:

$$\begin{array}{l}\textbf{let}\ r\ 0\ x = \mathit{return}\ x\\ \quad r\ n\ x = f\ x \gg\!= r\ (n-1)\\ \quad f\ x = e_1^*\\ \textbf{in}\lambda(n : \mathit{orc}) \to (r\ n\ \mathit{underscore} \gg\!= \lambda x \to e_2^*)\ \mathit{orc}\end{array}$$

Finally, inlining $\gg\!=$ yields an end-recursive expression:

$$\begin{array}{l}\textbf{let}\ r\ 0\ x\ s = (s, x)\\ \quad r\ n\ x\ s = \textbf{case}\ f\ x\ s\ \textbf{of}\ (s', v) \to r\ (n-1)\ v\ s'\end{array}$$

$$
\begin{array}{l}
\quad f\ x = e_1^* \\
\mathbf{in} \lambda(n : orc) \rightarrow (r\ n\ underscore \gg\!= \lambda x \rightarrow e_2^*)\ orc
\end{array}
$$

We abstract out a function *fixBind* that takes the expressions $\lambda x \rightarrow e_1^*$ and $\lambda x \rightarrow e_2^*$ as parameters:

$$
\begin{array}{l}
fixBind :: (a \rightarrow ST\ a) \rightarrow (a \rightarrow ST\ b) \rightarrow ST\ b \\
fixBind\ p\ q\ (n : orc) \\
\quad = \mathbf{let}\ r\ 0\ x\ s = (s, x) \\
\qquad\quad r\ n\ x\ s = \mathbf{case}\ p\ x\ s\ \mathbf{of}\ (s', v) \rightarrow r\ (n-1)\ v\ s' \\
\qquad \mathbf{in}\ (r\ n\ underscore \gg\!= q)\ orc
\end{array}
$$

With the help of this function the translated non-strict recursive declaration becomes considerably smaller:

$$fixBind\ (\lambda x \rightarrow e_1^*)\ (\lambda x \rightarrow e_2^*)$$

5.3 Adding Debugging Functionality

This section describes how declarative debugging facilities can be added to a program that has been translated to the monadic form shown above. It is based on previous work published in [Braßel and Siegel, 2007].

In order to use the monadic implementation of Lazy Call-by-Value evaluation for declarative debugging, four modifications have to be made to the transformed program:

1. The monad has to be extended, so that it is able to hold user-provided information about the correctness of subexpressions and to cancel the evaluation when the bug is located.

2. Function declarations have to be instrumented, so that the user can inspect the values resulting from their application.

3. Every data type has to be enhanced by a value *underscore*, that is used as a placeholder for skipped evaluations.

4. In order to display function arguments and results, functions that display program values in a user-friendly way have to be derived from the datatypes of the program to be debugged.

Implementation of Placeholder Values

Unneeded values whose evaluation is skipped are replaced by a placeholder value "_" (*underscore*). Since the evaluation of expressions of any type may be skipped, a way of extending arbitrary data types with the special value *underscore* has to be found. How this can be done depends on several circumstances

- In the Curry compiler `kics` every algebraic data type is extended with an *Or*-constructor that models nondeterministic values. This mechanism is easy to extend, so that it also adds a nullary constructor that represents the value *underscore*.

  The drawback of this method is that every data type has to changed; all functions that use primitive data types like — for example — integers have to be changed.

- In a bytecode interpreter, the expression *error* `"underscore"` can be used for the value *underscore*. Under Lazy Call-by-Value evaluation no unevaluated thunks are created, so no unevaluted values except *underscore* will occur. A primitive operator has been added to the bytecode interpreter that checks whether an expression is evaluated or not. It can be used to distinguish between *underscore* and other values.

  The drawback of this method is that it depends on low-level details of a specific interpreter, so that it can not be used with other interpreters or compilers.

  This has been implemeneted in the bytecode interpreter of the Essential Haskell Compiler (EHC).

- In [Braßel and Siegel, 2007], *underscore* is an expression whose evaluation throws a run-time exception. Since only unneeded values that are never accessed are replaced by *underscore*, the run-time exception is only thrown when the debugger tries to print out a value. Then the printing routine can catch the exception and display an underscore instead of the value.

- In the reference implementation in Appendix B.1 the value $\perp$ is used for *underscore*. It is evaluated to the value *Blackhole* that stands for self-referential definitions. This value can easily be detected by the printing routine.

Displaying Data

In the debugging of a program, arguments and results of function application must be displayed to the user. To use Haskell's type class *Show* for this has two disadvantages:

- It rules out implementing the placeholder value *underscore* by an exception-throwing expression, so that a special constructor has to be provided for every data type. But this is not possible for primitive data types.

- More flexible ways of displaying data may be needed, for example for pruning big or even circular data structures.

Therefore, a data type *Term* and a type class *ShowTerm* are introduced:

```
data Term = Term String [Term] | Underscore
class ShowTerm a where
  showCons :: a → Term
```

The constructor *Term* represents constructor terms, whereas the constructor *Underscore* represents the placeholder value that stands for unused expressions. Function *showCons* converts an expression into a printable *Term*. Now every data type can be made an instance of a class *ShowTerm*. For the data type *Maybe* the instance declaration would be:

```
instance ShowTerm a ⇒ ShowTerm Maybe a where
  showCons Nothing = Term "Nothing" []
  showCons (Just x) = Term "Just" [showCons x]
```

The drawback of this method is that an instance declaration has to be introduced for every data type and that the types of all function arguments have to be constrained to the type class *ShowTerm*.

For the EHC bytecode interpreter the need to introduce a type class *ShowTerm* has been circumvented by implementing function *showCons* as a primitive operator of the bytecode interpreter. The drawback of this method is that the constructor names have to be derived from the constructor numbers used in the interpreter. It turned out that this was not possible, because the constructors are enumerated *per type*, so that additional type information would be needed to distinguish between the constructors. But providing such information for polymorphic function arguments is exactly what the type class *ShowTerm* does. As a consequence, function *showCons* only displays constructor numbers instead of names.

A Debugging Monad

In the following it is assumed that the state monad maintaining the oracle list and the function that are used to lift programs into it are defined by through Haskell standard library functions:

```
type DebugM a = State [Integer] a
optional :: DebugM a → DebugM a
optional x = do (n : orc) ← get
                if n ≡ 0 then put orc ≫ return underscore
                         else put (n − 1 : orc) ≫ x
optionalApp :: DebugM a → (a → DebugM b) → DebugM b
```

$$
\begin{aligned}
&\mathit{optionalApp}\ x\ f = \mathit{optional}\ (\mathit{return}\ x) \ggg\!= f\\
&\mathit{fixBind} :: (a \to \mathit{DebugM}\ a) \to (a \to \mathit{DebugM}\ b) \to \mathit{DebugM}\ b\\
&\mathit{fixBind}\ p\ q = \mathbf{do}\ n : \mathit{orc} \leftarrow \mathit{get}\\
&\qquad\qquad\qquad\quad \mathit{put}\ \mathit{orc}\\
&\qquad\qquad\qquad\quad r\ n\ \mathit{underscore} \ggg\!= q\\
&\quad \mathbf{where}\ r\ 0\ x = \mathit{return}\ x\\
&\qquad\qquad r\ n\ x = p\ x \ggg\!= r\ (n-1)
\end{aligned}
$$

The definition of *DebugM* can easily be exchanged by the following definition. This makes it possible to execute *IO* actions or to cancel the debugging session, yielding a value of type *BugReport* as result:

$$\mathbf{type}\ \mathit{DebugM}\ a = \mathit{StateT}\ [\mathit{Integer}]\ (\mathit{ErrorT}\ \mathit{BugReport}\ \mathit{IO})\ a$$

We don't define the type *BugReport* here. It is intended to hold the same kind of information that nodes of an Evaluation Dependence Tree hold.

Instrumenting Functions

Function *traceFunCall* implements the debugging functionality. It has the following type signature:

$$\mathit{traceFunCall} :: \mathit{ShowTerm}\ a \Rightarrow \mathit{Term} \to \mathit{DebugM}\ a \to \mathit{DebugM}\ a$$

The first argument is a displayable constructor term that represents the function name and parameters. The second argument is the lifted function body. When a bug is found, this function returns a value of type *BugReport*; otherwise it returns the result of the function call together with the rest of the oracle list.

In order to instrument a program with debugging functionality, the lifted right hand side of every function declaration is wrapped in a call to function *traceFunCall*. For example, a function declaration

$$
\begin{aligned}
&\mathit{twice} :: (a \to a) \to a \to a\\
&\mathit{twice} = \lambda f \to \lambda x \to f\ (f\ x)
\end{aligned}
$$

is transformed into the following function declaration:

$$
\begin{aligned}
&\mathit{twice} :: (a \to \mathit{DebugM}\ a) \to \mathit{DebugM}\ (a \to \mathit{DebugM}\ a)\\
&\mathit{twice} = \lambda f \to \mathit{return}\\
&\qquad\qquad (\lambda x \to \mathit{traceFunCall}\\
&\qquad\qquad\qquad\quad (\mathit{Term}\ \texttt{"twice"}\ [\mathit{showTerm}\ f, \mathit{showTerm}\ x])\\
&\qquad\qquad\qquad\quad (\mathit{optional}\ (f\ x) \ggg\!= f))
\end{aligned}
$$

In Haskell, an n-ary function can be seen as a unary function that returns an $(n-1)$-ary function as result. Thus, function *twice* could also be transformed to this function:

$$
\begin{aligned}
&\mathit{twice} :: (a \to \mathit{DebugM}\ a) \to \mathit{DebugM}\ (a \to \mathit{DebugM}\ a)\\
&\mathit{twice} = \lambda f \to \mathit{traceFunCall}\\
&\qquad\qquad (\mathit{Term}\ \texttt{"twice"}\ [\mathit{showTerm}\ f])\\
&\qquad\qquad (\mathit{return}\ (\lambda x \to \mathit{optional}\ (f\ x) \gg\!= f))
\end{aligned}
$$

But then the resulting value would always be a function that cannot be displayed by the debugger. Therefore, the call to *traceFunCall* should be 'pushed' inside as many abstractions as possible.

It is yet unclear how nested function declarations like the following should be treated:

$$
\begin{aligned}
&f :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&f = \lambda x \to \mathbf{let}\ y = x + x\\
&\qquad\quad \mathbf{in} \lambda z \to y + z
\end{aligned}
$$

If *traceFunCall* is pushed inside the inner abstraction, then the computation of y appears to be outside of function f; if *traceFunCall* is only pushed inside the outer abstraction, then a function is displayed as the result of function f, which is less informative than the resulting integer value.

A pragmatic solution is to lambda-lift the inner abstraction to the outer level:

$$
\begin{aligned}
&f :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}\\
&f = \lambda x \to \mathbf{let}\ y = x + x\\
&\qquad\quad \mathbf{in}\ f'\ y\\
&f' = \lambda y \to \lambda z \to x + y
\end{aligned}
$$

In this work, the practical experiences were made with the intermediate core language of the York Haskell Compiler. It allows for lambda abstractions, but in the compilation from Haskell to YHC Core all abstractions are automatically lifted to the top level, so that the problem does not occur.

5.4 Summary

With the techniques presented in this chapter, a declarative debugger that uses the principle of Lazy Call-by-Value evaluation can be implemented. A formal translation scheme for the monadic implementation of this evaluation strategy is provided; it is based on a provably correct translation scheme published in [Wadler, 1990], and it transforms cyclic declarations into end-recursive functions, which can be evaluated efficiently.

It is described how monadic Lazy Call-by-Value programs can be instrumented with declarative debugging facilities. This has previously been implemented in the Kiel Curry System (KiCS). In [Braßel and Siegel, 2007], the implementation is described in more detail.

Chapter 6

Conclusion

6.1 Practical Experiences

In the first two months of my work on this diploma thesis I have worked at the Department of Information and Computing Sciences of the Utrecht University. My goal was to implement a debugger that is based on the concept of Lazy Call-by-Value evaluation for the Essential Haskell Compiler (EHC), that is being developed there. I succeeded only partly, due to the following problems:

First, some design decisions, that were taken with efficient compilation of Haskell programs in mind, turned out to be disadvantageous for the task of writing a debugger frontend. One example is the enumeration scheme for constructors, that made it difficult to relate the binary representation of constructor terms to their textual representation.

Second, much effort had gone into the development of the type-checking frontend and the code-generating backend of EHC, whereas the intermediate representation — a core language similar to an applied lambda calculus — was only used to translate code to the next compilation stage; I was the first one who actually implemented transformations for the intermediate representation. In doing so, I stumbled over problems that could not be foreseen: For example, adding new function declarations to the intermediate core representation turned out to be too difficult.

Finally, reacting to those problems would have required more time and a more mature concept of the implementation than I had at that time.

Implementation of Natural Semantics

A prototype implementation of the semantics developed in Chapter 3 has been developed. It uses the algorithm developed in Chapter 2 for removing recursive declarations. It is a straightforward implementation of the semantic rules from Figure 3.5 on page 32 and Figure 3.7 on page 39.

Thereby it has been shown that the semantic model is suitable to calculate oracle sets for arbitrary programs and that the resulting oracle sets can indeed be used to direct Lazy Call-by-Value evaluation.

Reference Implementation

A reference implementation of non-strict oracle creation and Lazy Call-by-Value evaluation has been developed. It is based on the core language of the York Haskell Compiler (YHC) [Golubovsky et al., 2007]: Haskell programs are compiled to YHC Core programs by the compiler YHC. Then the resulting Core program is read in by the reference implementation.

The implementation of non-strict oracle creation follows Chapter 4 on page 55. YHC Core programs are instrumented with calls to the low-level functions *newOracleEntry*, *enterRhs* and *leaveRhs* that manipulate oracle lists as described there. The resulting programs are interpreted by a simple interpreter that uses the semantics of lazy evaluation presented in [Sestoft, 1997]. The low-level functions are implemented in the language C and called by the interpreter via Haskell's foreign function interface.

The implementation of Lazy Call-by-Value evaluation follows Chapter 5 on page 67. The loaded Core program is transformed into a monadic version as described there. The implementation of the monad is then linked to the transformed program, and the reslting program is evaluated by the same interpreter as the program that has created the oracle.

The reference implementation is written in the language Haskell. It can be found in Appendix B on page 91.

Efficiency

In most cases, the semantics presented in this work calculates smaller oracle lists that the semantics of [Braßel et al., 2007]. As an example, consider the program in Figure 6.1 on the facing page. When it is evaluated using their semantics, the oracle list

$$[3, 0, 1, 0, 0, 18]$$

is obtained. This means that under Lazy Call-by-Value evaluation the oracle will have to be checked 24 times. But with the semantics presented in this thesis, the resulting oracle list is

$$[0, 5, 0, 0, 6],$$

so that the oracle is only checked 16 times: For this example, the run-time overhead is reduced by one third.

```
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)
length Nil = Z
length (Cons _ b) = S (length b)
take Z _          = Nil
take (S x) (Cons a b) = Cons a (take x b)
take (S _) Nil = Nil
fib = fib
fibs x = Cons (fib x) (fibs (S x))
mainexpr = isTwo (length (take (S (S Z)) (fibs Z)))
isTwo (S (S Z)) = True
isTwo _         = False
```

Figure 6.1: Example program

6.2 Results

This thesis makes several contributions that go beyond the work published in [Braßel et al., 2007] and [Braßel and Siegel, 2007]:

- A novel way of formalizing the semantics of Lazy Call-by-Value evaluation has been developed. It leads to smaller oracle lists than the previous approach of [Braßel et al., 2007].
- This semantic model has been proved correct and verified by implementing it directly in Haskell.
- It has been shown that recursive declarations have to be ruled out in order to evaluate programs by the Lazy Call-by-Value strategy.
- Also, a method to eliminate recursive declarations in an efficient way has been suggested. It has also been implemented for the core languages of the Utrecht Haskell Compiler and for the core language of the York Haskell Compiler.
- Non-strict oracle creation has been implemented by a program transformation that adds calls to primitive stateful operations to a declarative program. Due to the simplified semantic model, the primitive functions turned out to be smaller and simpler than the functions needed in [Braßel et al., 2007].
- Lazy Call-by-Value evaluation has been implemented by a program transformation that transforms a declarative program into a monadic

declarative program. This monadic program is evaluated in strict order and maintains a global state in a purely functional way.

For this transformation formal rules were systematically derived from transformation rules given in [Wadler, 1990].

- The implementation of elimination of mutual recursion, non-strict oracle creation and Lazy Call-by-Value evaluation has been integrated in a prototype interpreter, showing that they fit together.

6.3 Future Work

- The reference implementation is merely a proof of concept. It is not designed for efficiency, and it cannot handle primitive data types; a debugger that applies the results of this paper to a sufficient subset of Haskell still has to be written.

- This work is only concerned with debugging purely functional programs. The debugger described in [Braßel and Siegel, 2007] is able to debug interactive programs by providing a virtual I/O environment in which input from the user or from the file system is simulated: When the oracle is calculated all interactions are recorded, so that they can be replayed at debugging time.

  It is desirable for a debugging tool to support I/O actions, so a similar approach could be taken here.

- The suggested method of removing mutually recursive declarations has to be tested for efficiency. Due to the shortcomings of the reference implementation no meaningful results can be given yet.

- The semantic model can be extended with strict and recursive declarations, like the `letrec`-construct of the programming languages Lisp and Scheme. An extension of the semantic model is suggested in Appendix A.

- As shown in Section 2.4, the algorithm that removes mutual recursiven could benefit from the availability of strictness information. Also, strictness information could be used to replace non-strict declarations by the `letrec`-like construct sketched in Appendix A, so that the size of oracle lists is reduced further.

  It still has to be investigated, how much efficiency can be gained by deriving strictness information from the program to be debugged.

The debugger for the functional logic programming language Curry presented in [Braßel and Siegel, 2007] has been integrated into the Kiel Curry

System. Since then it has turned out to be useful in the debugging of functional logic programs.

Probably the easiest way to put the results of this work into practice is to apply them to this existing debugger.

Bibliography

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

[Braßel and Siegel, 2007] Braßel, B. and Siegel, H. (2007). Debugging lazy functional programs by asking the oracle. In *IFL*, pages 183–200.

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

[Festa et al., 1999] Festa, P., Pardalos, P. M., Mauricio, and Resende, G. C. (1999). Feedback set problems. In *Handbook of Combinatorial Optimization*, pages 209–258. Kluwer Academic Publishers.

[Golubovsky et al., 2007] Golubovsky, D., Mitchell, N., and Naylor, M. (2007). Yhc.Core - from Haskell to Core. *The Monad.Reader*, 7:236–243.

[Jones, 1987] Jones, S. L. P. (1987). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Jones and Marlow, 1999] Jones, S. P. and Marlow, S. (1999). Secrets of the Glasgow Haskell compiler inliner. In *Journal of Functional Programming*.

[Launchbury, 1993] Launchbury, J. (1993). 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.

[Marlow et al., 2007] Marlow, S., Iborra, J., Pope, B., and Gill, A. (2007). A lightweight interactive debugger for Haskell. Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop.

[Nilsson and Sparud, 1997] Nilsson, H. and Sparud, J. (1997). The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150.

[Schauser and Goldstein, 1995] Schauser, K. E. and Goldstein, S. C. (1995). How much non-strictness do lenient programs require? In *Conference on Functional Programming Languages and Computer Architecture*, La Jolla, CA.

[Sestoft, 1997] Sestoft, P. (1997). Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264.

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

[Traub, 1991] Traub, K. R. (1991). *Implementation of non-strict functional programming languages*. MIT Press, Cambridge, MA, USA.

[Wadler, 1990] Wadler, P. (1990). Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78.

Appendix A

Extending the Semantics with Lisp-like Declarations

The programming languages Lisp and Scheme extend the strict evaluation model with recursive declarations as described in Section 2.3: The right hand sides of a mutually recursive declaration are evaluated before the body of the declaration. They may refer to each other, but in their evaluation they are not bound on the heap, so they cannot access the concrete values of the variables that are being defined.

Modified Semantics

This kind of delarations can be supported by modifying the semantics for non-strict oracle creation and Lazy Call-by-Value evaluation: First, the requirement that in a strict declaration $\mathbf{let}^! \ \overline{x_n = e_n}$ **in** e all right hand sides $e_1, \ldots, e_n$ must be in weak head normal form is dropped. Then rule *LetStrict*^{L} is replaced by the following rule:

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,n\} : \ v_i = x_{t:(c+1)} \\ \forall i \in \{1,\ldots,n\} : \ (t_{i-1},\, c_{i-1},\, \Gamma_{i-1},\, e_i\overline{[v_n/x_n]}) \Downarrow_{p,\gamma_i} (t_i,\, c_i,\, \Gamma_i,\, e_i') \\ (t_n,\, c_n,\, \Gamma_n \cup \{\overline{(v_n,\, e_n')}\},\, e\overline{[v_n/x_n]}) \Downarrow_{p,\gamma} (t',\, c',\, \Delta,\, e')\end{array}}{(t_0,\, c_0,\, \Gamma_0,\, \mathbf{let}^! \ \overline{x_n = e_n} \ \mathbf{in} \ e) \Downarrow_{p,\bigcup_{j=1}^{n}\gamma_j \cup \gamma} (t',\, c',\, \Delta,\, e')}$$

It differs from the previous rule *LetStrict*^{L} in that it evaluates the right hand sides, before it binds them on the heap. Rule *LetStrict*^{S} has to be modified analogously:

$$\frac{\begin{array}{c}\forall i \in \{1,\ldots,n\} : \ v_i = x_{t:(c+1)} \\ \forall i \in \{1,\ldots,n\} : \ (t_{i-1},\, c_{i-1},\, \Gamma_{i-1},\, e_i\overline{[v_n/x_n]}) \downarrow_{p,\omega} (t_i,\, c_i,\, \Gamma_i,\, e_i') \\ (t_n,\, c_n,\, \Gamma_n \cup \{\overline{(v_n,\, e_n')}\},\, e\overline{[v_n/x_n]}) \downarrow_{p,\omega} (t',\, c',\, \Delta,\, e')\end{array}}{(t_0,\, c_0,\, \Gamma_0,\, \mathbf{let}^! \ \overline{x_n = e_n} \ \mathbf{in} \ e) \downarrow_{p,\omega} (t',\, c',\, \Delta,\, e')}$$

Efficiency

Reconsider the example expression from page 33. It contains three non-strict declarations. In the Lazy Call-by-Value evaluation of that expression, one non-strict declaration is skipped by rule $LetSkip^S$, and two are evaluated by rule $LetEval^S$.

If strictness information is available for this expression, then two non-strict declarations can be turned into strict ones: In order to reduce the expression to a weak head normal form, the value of p as well as the value of q will always be required. Thus, they do not need to be controlled by the oracle, so that their declarations can be replaced by the newly-introduced Lisp-like declarations:

$$\begin{array}{l}\textbf{let}^{!}\ p = \textbf{let}^{!}\ s = \mathit{True}\\ \qquad\qquad \textbf{in let}^{!}\ q = \textbf{let}^{?}\ r = \textbf{case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to b\\ \qquad\qquad\qquad\qquad \textbf{in}\ \mathit{Cons}\ r\ s\\ \qquad\qquad\quad \textbf{in}\ \mathit{Cons}\ q\ s\\ \textbf{in case}\ p\ \textbf{of}\ \mathit{Cons}\ a\ b \to a\end{array}$$

Now, only one oracle entry needs to be checked in the evaluation of this expression; instead of the oracle list $[2]$, the oracle list $[0]$ is computed.

Cyclic Declarations

In Chapter 2 we were not able to give a strict evaluation strategy that makes it possible to evaluate the following Haskell expression in sequential evaluation order:

$$\begin{array}{l}\textbf{let}\ x = \textbf{let}\ y = \mathit{length}\ x\\ \qquad\qquad \textbf{in}\ [1, 2, y]\\ \textbf{in}\ \mathit{sum}\ x\end{array}$$

We had to remove the cyclic data dependence entirely; either by introducing a least fixed point combinator, or by turning the data declaration into a function declaration. But now Lisp-like declarations allow us to transform this expression into an equivalent expression that can be evaluated in strict order, although it contains a cyclic data declaration.

It is still not possible to evaluate the above expression directly, because x refers to itself in a strict way. To overcome this problem, we transform the expression into the following, equivalent expression:

$$\textbf{let}\ x = \mathit{fix}\ ((\lambda y \to [1, 2, y]) \circ \mathit{length})\ \textbf{in}\ \mathit{sum}\ x$$

By 'unrolling' the fixed point, this expression can again be transformed into an equivalent expression:

$$\textbf{let}\ x = (\lambda y \to [1, 2, y])\ (\mathit{fix}\ (\mathit{length} \circ (\lambda y \to [1, 2, y])))\ \textbf{in}\ \mathit{sum}\ x$$

Now this expression can be rewritten as a recursive declaration, so that the fixed point combinator is no longer needed:

$$\begin{array}{l}\textbf{let } y = \textbf{let } x = [1, 2, y] \\ \qquad\quad \textbf{in } \mathit{length}\ x \\ \textbf{in } \mathit{sum}\ [1, 2, y]\end{array}$$

Assuming that function *length* is defined and there is also support for integers, the Haskell expression can be translated to the following $\lambda^{?!}$ expression:

$$\begin{array}{l}\textbf{let}^{!}\ y = \textbf{let}^{!}\ x = [1, 2, y] \\ \qquad\quad \textbf{in } \mathit{length}\ x \\ \textbf{in } \mathit{sum}\ [1, 2, y]\end{array}$$

At the cost of some code duplication, we have obtained an expression that can be evaluated in sequential order: The elements of list $[1, 2, y]$ can be counted without knowing the value of y. When the length of the list has been calculated and bound to the identifier y, another list $[1, 2, y]$ can be created and summed up.

In short, we have rewritten a recursive declaration as the least fixed point $\mu_{\phi\circ\psi}$ of a composed function $\phi \circ \psi$, where ϕ is non-strict. Then we have unrolled this fixed point, exploiting the fact that $\mu_{\phi\circ\psi} = \phi(\mu_{\psi\circ\phi})$ for monotonous functions ϕ, ψ. Since ϕ is not strict in its argument, $\psi \circ \phi$ is also non-strict, and the fixed point $\mu_{\psi\circ\phi}$ can be rewritten as a recursive declaration.

Conclusion

In order to gain efficiency from Lisp-like declarations, strictness information must be available. Otherwise the transformations sketched above can not be applied safely. Also, the strictness of a function cannot be determined beforehand in general, so that there will remain cases where recursive data declarations will have to be eliminated as described in Chapter 2.

Nevertheless, deriving strictness information from declarative programs is common practice, and if it is known statically that the value of a non-strict declaration is needed, then an oracle entry can be saved by turning the declaration into Lisp-like declaration. Therefore, enhancing the semantics with Lisp-like declarations can increase run-time efficiency even in the presence of simple forms of strictness analysis.

Appendix B

Reference Implementation

B.1 Interpreter

```
{-# LANGUAGE ForeignFunctionInterface #-}
module Main (main) where

import Foreign.Ptr
import System.Environment (getArgs)
import Control.Monad.Fix (mfix)
import Control.Monad
import Data.IORef
import Data.Maybe
import Yhc.Core
import Data.Function (on)
import Data.Maybe
import Data.Graph
import Data.List
```

The functions for manipulating oracles are implemented in the language C (see section B.3 on page 106). They are called via Haskell's foreign function interface. Since oracles are not accessed directly by the interpreter, they are simply defined as values of type *Ptr* ().

```
type Oracle = Ptr ()
foreign import ccall initOracle :: IO ()
foreign import ccall newOracleEntry :: IO Oracle
foreign import ccall enterRhs :: Oracle → IO Oracle
foreign import ccall leaveRhs :: Oracle → IO ()
foreign import ccall popOracle :: IO Int
```

Function *mapSnd* applies a function to the second component of every list element.

```
mapSnd :: (a → b) → [(c, a)] → [(c, b)]
mapSnd f = map (λ(l, r) → (l, f r))
```

Simplified Core Expression Syntax

Data type *Expr* implements a core language that is even simpler than the core language of the York Haskell Compiler. The differences are that there are only unary applications and abstractions, that there is no distinction between function names and the names of local variables, and that constructor terms are always saturated:

```
type Ident  = String
type CIdent = String

type Bind = (Ident, Expr)

data Expr = Var{varId :: Ident}
          | Abs Ident Expr
          | App Expr Ident
          | Let [Bind] Expr
          | Con CIdent [Ident]
          | Case Expr [(CorePat, Expr)]
          | Literal CoreLit
```

Function *isVar* tests whether an expression is an identifier. Function *whnf* follows the definition of the predicate whnf in chapter 3. Function *allWhnf* tests if all right hand sides in a list of bindings are in weak head normal form.

```
isVar :: Expr → Bool
isVar (Var _) = True
isVar _       = False

whnf :: Expr → Bool
whnf (Con _ _) = True
whnf (Abs _ _) = True
whnf _         = False

allWhnf :: [Bind] → Bool
allWhnf = all (whnf ∘ snd)
```

Function *fvs* returns a list containing the identifiers that occur free in an expression.

```
fvs :: Expr → [Ident]
fvs expr = case expr of
```

```
  Var v       → [v]
  Abs x e     → delete x (fvs e)
  App f x     → union (fvs f) [x]
  Let bs e    → foldr letFvs (fvs e) bs \\ map fst bs
  Con _ xs    → nub xs
  Case e as   → foldr altFvs (fvs e) as
  Literal _   → []
  where
  letFvs (_, e) vs        = union (fvs e) vs
  altFvs (p, e) vs        = (fvs e \\ pvars p) `union` vs
  pvars (PatCon _ vs) = vs
  pvars _             = []
```

Function *isRecursive* checks whether the right hand side of a binding refers to that binding in a recursive way.

```
isRecursive :: Bind → Bool
isRecursive (x, e) = x ∈ fvs e
```

Traversal of Expression Trees

Function *mapExpr* applies a function f to the direct subexpressions of *expr*. Function *postOrder* uses *mapExpr* to apply a function f to all subexpressions (including the whole expression) in post-order.

```
mapExpr :: (Expr → Expr) → Expr → Expr
mapExpr f expr
  = case expr of
    Abs x e    → Abs x (f e)
    App e x    → App (f e) x
    Let bs e   → Let (mapSnd f bs) (f e)
    Case e as  → Case (f e) (mapSnd f as)
    _          → expr
postOrder :: (Expr → Expr) → Expr → Expr
postOrder fun expr = fun (mapExpr (postOrder fun) expr)
```

Loading Core files

Function *loadCoreExpr* loads a `.yca-` or `.ycr`-file and returns a declaration that defines all top-level declarations of the loaded file and has *mainexpr* as its body.

```
loadCoreExpr :: FilePath → Expr → IO Expr
loadCoreExpr filename mainexpr
```

```
  = do core ← loadCore filename
       let binds = [(x, foldr Abs (toExpr body) args)
                   | CoreFunc x args body ← coreFuncs core]
       return (Let binds mainexpr)
```

Function *toExpr* transform an expression of type *CoreExpr* into an expression of type *Expr*.

It is assumed that all constructor terms are saturated,i.e. that every constructor of arity n is wrapped in an application of n arguments. Core files created by `yhc` seem to satisfy this condition.

```
toExpr :: CoreExpr → Expr
toExpr expr
  = case expr of
    CoreCon c → Con c []
    CoreVar v → Var v
    CoreFun v → Var v
    CoreApp (CoreCon c) as
      | null bs'   → con
      | otherwise → Let bs' con
      where
        con = Con c (map fst bs)
        bs = zipWith foo [1..] (map toExpr as)
        bs' = filter (¬ ∘ isVar ∘ snd) bs
        foo :: Int → Expr → (Ident, Expr)
        foo _ (Var v) = (v, Var v)
        foo n e = ("_v" ++ show n, e)
    CoreApp f as  → mkApp (toExpr f) (map toExpr as)
    CoreLam xs e  → foldr Abs (toExpr e) xs
    CoreCase e as → Case (toExpr e) (mapSnd toExpr as)
    CorePos _ e   → toExpr e
    CoreLet bs e  → Let (mapSnd toExpr bs) (toExpr e)
    CoreLit l     → Literal l
```

Helper Functions

Function *mkApp* creates nested applications from a list of arguments. It transforms the arguments of applications into **let**-bound values if necessary.

```
mkApp :: Expr → [Expr] → Expr
mkApp f as = foldl app f as
  where
  app e (Var x) = App e x
  app e x       = Let [("_x", x)] (App e "_x")
```

The function call *mkSeq e1 e2* creates an expression that is equivalent to the Haskell expression *e1* `seq` *e2*. It (ab)uses Case expressions to force the evaluatiuon of *e1*.

```
mkSeq :: Expr → Expr → Expr
mkSeq e1 e2 = Case e1 [(PatDefault, e2)]
```

Some self-explaining helper functions:

```
mkFunapp :: Ident → [Expr] → Expr
mkFunapp f xs = mkApp (Var f) xs
mkLet, mkLetstrict :: String → Expr → Expr → Expr
mkLet x e1 e2 = Let [(x, e1)] e2
mkLetstrict x e1 e2 = mkLet x e1 (mkSeq (Var x) e2)
```

The type *FvsNode* stands for a single node in a dependency graph. It contains the binding, i.e. the identifier and the right hand side, as well as a list of the identifiers the right hand side refers to.

Function *depGraph* turns a list of bindings into a dependency graph.

Function *sortSccs* calculates a list of strongly connected components with the help of the library function *stronglyConnCompR*. In contrast to the list of SCCs returned from this function, the list of SCCs returned by *sortSccs* is in reversed order. This is required, beacuase we start to process nested declarations at the innermost binding.

```
type FvsNode = (Bind, Ident, [Ident])
depGraph :: [Bind] → [FvsNode]
depGraph = map (λ(x, e) → ((x, e), x, fvs e))
sortSccs :: [FvsNode] → [SCC FvsNode]
sortSccs = reverse ∘ stronglyConnCompR
```

Breaking Declarations into Binding Groups

Function *splitLets* breaks a declaration into their smallest binding groups.

```
splitLets :: Expr → Expr
splitLets expr
  = case mapExpr splitLets expr of
    Let bs e
      → let fvsGraph = depGraph bs
        sccs         = stronglyConnComp fvsGraph
        bs'          = map flattenSCC sccs
        in foldr Let e bs'
    e  → e
```

Remove mutual recursion

Function *rmMutualRecursion* turns a mutually recursive data declaration (a letrec with more than one binding that contain at least one non-whnf right hand side) into a single recursive declaration.

```
rmMutualRecursion :: Expr → Expr
rmMutualRecursion (Let bs body)
  | ¬ (allWhnf bs) ∧ length bs > 1
  = let fvsGraph = depGraph bs
    sccs         = sortSccs fvsGraph
    (body', bound, fbs)
      = partitionBinds (fvs body) sccs (body, mkTuple fbs, [])
    in mkSingleLet body' bound fbs
rmMutualRecursion e = e
```

Function *partitionBinds* takes the following arguments: A list of identifier that occur in the body of the declaration, a sorted list of strongy connected components, a 3-tuple consising of the body of the declaration, a tuple expression that contains the feedback variables, and the list of identifiers that are already added to the feedback set. It returns an updated version of that 3-tuple, in which the body expression is 'surrounded' by declarations of identifiers that the body refers to, the tuple expression is 'surrounded' by declarations that are needed to define the feedback vriables, and the set of feedback identifiers is the complete feedback set:

```
partitionBinds :: [Ident] → [SCC FvsNode]
  → (Expr, Expr, [Ident])
  → (Expr, Expr, [Ident])
```

When there is no binding left in a strongly connected component, then move to the next SCC:

```
partitionBinds pull (CyclicSCC [] : ds) part
  = partitionBinds pull ds part
```

If the next SCC is cyclic, then pick the best candidate for the feedback set and remove it from the SCC. The rest of the SCC breaks into smaller SCCs that are sorted and added to the remaining list of SCCs. The selected candidate is added to the feedback set, and its declaration is added to the tuple expression:

```
partitionBinds pull (CyclicSCC d : ds) (body, bound, fbs)
  = let (b, d') = pickFbNode pull d
    sccs = sortSccs d' ++ ds
    in partitionBinds pull sccs (body, Let [b] bound, fst b : fbs)
```

If the next SCC is acyclic, then it is not added to the feedback set. Instead, its declaration is added to the tuple expression. Depending on whether it is needed in the body expression, its declaration is also added to the body expression:

$$
\begin{aligned}
&\mathit{partitionBinds}\ \mathit{pull}\ (\mathit{AcyclicSCC}\ ((x, e), \_, r) : \mathit{ds})\ (\mathit{body}, \mathit{bound}, \mathit{fbs}) \\
&\quad = \mathit{partitionBinds}\ \mathit{pull}'\ \mathit{ds}\ (\mathit{body}', \mathit{mkLet}\ x\ e\ \mathit{bound}, \mathit{fbs}) \\
&\quad \textbf{where}\ (\mathit{body}', \mathit{pull}') \\
&\qquad \mid x \in \mathit{pull} = (\mathit{mkLet}\ x\ e\ \mathit{body}, r\ `\mathit{union}`\ \mathit{pull}) \\
&\qquad \mid \mathit{otherwise} = (\mathit{body}, \mathit{pull})
\end{aligned}
$$

When there are no more declarations to be processed, the 3-tuple is returned as result:

$$
\begin{aligned}
&\mathit{partitionBinds}\ \_\mathit{pull}\ [\,]\ \mathit{part} \\
&\quad = \mathit{part}
\end{aligned}
$$

Function *pickFbNode* picks the best candidate from a SCC. Irs choice depends not only on the SCC, but also on whether the candidate is referred to by the body expression:

$$
\begin{aligned}
&\mathit{pickFbNode} :: [\mathit{Ident}] \to [\mathit{FvsNode}] \to (\mathit{Bind}, [\mathit{FvsNode}]) \\
&\mathit{pickFbNode}\ \mathit{pull}\ \mathit{defs} = (b, d) \\
&\quad \textbf{where} \\
&\quad \mathit{ds} \qquad = [x \mid (\_, x, \_) \leftarrow \mathit{defs}] \\
&\quad (b, y, \_) = \mathit{maximumBy}\ (\mathit{compare}\ `\mathit{on}`\ \mathit{weight}\ \mathit{pull}\ \mathit{ds})\ \mathit{defs} \\
&\quad d \qquad = [n \mid n@(\_, x, \_) \leftarrow \mathit{defs}, x \not\equiv y]
\end{aligned}
$$

Function *weight* estimates the usefulness odf adding an identifier to the feedback set. It uses the fact, that tuples are sorted in exicographic order by default. An identifier is rated on whether it

1. has a recursive reference to itself,
2. has a high number of references to other identifiers in the same SCC, or
3. is referred to by the body expression.

$$
\begin{aligned}
&\mathit{weight} :: [\mathit{Ident}] \to [\mathit{Ident}] \to \mathit{FvsNode} \to (\mathit{Bool}, \mathit{Int}, \mathit{Bool}) \\
&\mathit{weight}\ \mathit{pull}\ \mathit{defs}\ (\_, x, \mathit{fv}) = (\mathit{recursive}, \mathit{length}\ \mathit{incoming}, \mathit{pulled}) \\
&\quad \textbf{where}\ \mathit{recursive} = x \in \mathit{fv} \\
&\qquad\qquad \mathit{incoming} = \mathit{fv}\ `\mathit{intersect}`\ \mathit{defs} \\
&\qquad\qquad \mathit{pulled} \quad = x \in \mathit{pull}
\end{aligned}
$$

Function *mkSingleLet* takes the body expression, the tuple expression and the list of feedback idnetifiers. It creates a single recursive declaration from them. For example, the function call

$$mkSingkeLet\ body\ bound\ [\texttt{"x"}, \texttt{"y"}]$$

returns a declaration

$$\begin{aligned}
&\textbf{let } rec = \textbf{let } x = \textbf{case } rec \textbf{ of } Tuple2\ x\ y \to x \\
&\qquad\qquad \textbf{in let } y = \textbf{case } rec \textbf{ of } Tuple2\ x\ y \to y \\
&\qquad\qquad\quad \textbf{in } bound \\
&\textbf{in let } x = \textbf{case } rec \textbf{ of } Tuple2\ x\ y \to x \\
&\qquad\qquad \textbf{in let } y = \textbf{case } rec \textbf{ of } Tuple2\ x\ y \to y \\
&\qquad\qquad\quad \textbf{in } body
\end{aligned}$$

$$\begin{aligned}
&mkSingleLet :: Expr \to Expr \to [Ident] \to Expr \\
&mkSingleLet\ body\ bound\ [v] \\
&\quad = mkLet\ v\ bound\ body \\
&mkSingleLet\ body\ bound\ fbs \\
&\quad = mkLet\ recname\ bound'\ body' \\
&\quad \textbf{where} \\
&\quad body' \ = mkFbSelectors\ body \\
&\quad bound' = mkFbSelectors\ bound \\
&\quad mkFbSelectors\ b = foldr\ mkSelector\ b\ fbs \\
&\quad mkSelector\ v\ b \ = mkLet\ v\ (mkSel\ (Var\ recname)\ v\ fbs)\ b \\
&\quad recname = \texttt{"rec"}
\end{aligned}$$

$$\begin{aligned}
&mkTuple :: [Ident] \to Expr \\
&mkTuple\ [e] = Var\ e \\
&mkTuple\ es \ = Con\ (mkTupleConstr\ \$\ length\ es)\ es
\end{aligned}$$

$$\begin{aligned}
&mkTupleConstr :: Int \to CIdent \\
&mkTupleConstr\ arity = \texttt{"Tuple"} +\!\!+ show\ arity
\end{aligned}$$

$$\begin{aligned}
&mkSel :: Expr \to Ident \to [Ident] \to Expr \\
&mkSel\ e\ v\ vs = Case\ e\ [(pat, Var\ v)] \\
&\quad \textbf{where } pat \ = PatCon\ tcon\ vs \\
&\qquad\quad tcon = mkTupleConstr\ (length\ vs)
\end{aligned}$$

Instrument Code for Oracle Creation

It is assumed that the right hand sides of declarations with more than one binding are in weak head normal form.

```
instrument :: Expr → Expr
instrument e@(Let [(x, e1)] e2)
  | whnf e1 = e
  | isRecursive (x, e1) = instrRecLet x e1 e2
  | isVar e1 = e
  | otherwise           = instrLet x e1 e2
instrument e = e

instrLet :: String → Expr → Expr → Expr
instrLet x e1 e2
  = withNewEntry (mkLet x (withOracle e1) e2)

instrRecLet :: String → Expr → Expr → Expr
instrRecLet x e1 e2
  = mkLetstrict x' body $ withNewEntry $ letXXo e2
  where x' = x ++ "'"
        body = Abs "o" (withOracle (withNewEntry (letXXo e1)))
        letXXo = mkLet x (mkFunapp x' [Var "o"])
```

Functions *withNewEntry* and *withOracle* add calls to the primitive functions *newOracleEntry*, *enterRhs* and *leaveRhs* to an expression:

$$
\begin{aligned}
\mathit{withNewEntry}\ e &\equiv \textbf{let}\ !\, o = \mathit{newOracleEntry} \\
&\quad\ \ \textbf{in}\ e \\
\mathit{withOracle}\ e &\equiv \textbf{let}\ !\, o_{save} = \mathit{enterRhs}\ o \\
&\quad\ \ \textbf{in let}\ !\, \mathit{result} = e \\
&\quad\quad\ \ \textbf{in}\ \mathit{leaveRhs}\ `\mathit{seq}`\ \mathit{result}
\end{aligned}
$$

```
withNewEntry :: Expr → Expr
withNewEntry = mkLetstrict "o" (Var "newOracleEntry")

withOracle :: Expr → Expr
withOracle r = enterAndSave strictLet
  where
  strictLet      = mkLetstrict "result" r leaveWithResult
  leaveWithResult = leaveDef `mkSeq` Var "result"
  leaveDef       = mkFunapp "leaveRhs" [Var "o_save"]
  enterAndSave   = mkLetstrict "o_save" callRhs
  callRhs        = mkFunapp "enterRhs" [Var "o"]
```

Lift code into state monad

Function *liftToST* implements the lifting of Core expression into the Lazy Call-by-Value monad via the translation scheme from figure 5.1 on page 71.

In order to avoid name-clashes with Prelude functions, the function *return* is called *unit* here, and the operator $\gg\!=$ is now a function *bind*:

```
liftToMonad :: Expr → Expr
liftToMonad = either mkRet id ∘ liftToST
  where mkRet e = mkFunapp "Lcbv;unit" [e]
```

The data type *Either* is used to distinguish between values that are returned by function *bind* and monadic expressions that are already lifted into the state monad. The expression *Left x* stands for *unit x*, and the expression *Right x* stands for *x*:

```
liftToST :: Expr → Either Expr Expr
liftToST (Var v) = Left (Var v)
liftToST (Abs x e)
   = Left (Abs x (liftToMonad e))
liftToST (App f x)
   = case liftToST f of
     Left f'  → Right (App f' x)
     Right f' → Right (mkFunapp "Lcbv;funApp" [f', Var x])
liftToST (Let bs e)
   | allWhnf bs
   = case liftToST e of
     Left e'  → Left (Let bs' e')
     Right e' → Right (Let bs' e')
     where bs' = mapSnd liftToVal bs
liftToST (Let [(x, r)] b)
   | isRecursive (x, r)
   = Right (mkFunapp "Lcbv;fixBind" [Abs x $ liftToMonad r, b'])
   | isVar r
   = Right (App b' (varId r))
   | otherwise
   = case liftToST r of
     Left r'  → Right (mkFunapp "Lcbv;optionalApp" [r', b'])
     Right r' → Right (mkFunapp "Lcbv;optionalBind" [r', b'])
  where
  b' = Abs x (liftToMonad b)
liftToST (Let _ _)
   = error $ "internal error: mutual recursion in liftToST"
liftToST (Con c xs)
   = Left (Con c xs)
liftToST (Case e as)
   = let as' = mapSnd liftToMonad as
     in case liftToST e of
        Left e' → Right (Case e' as')
```

```
        Right e' → let args = [e', Abs "e" (Case (Var "e") as')]
                   in Right (mkFunapp "Lcbv;bind" args)
liftToST (Literal l)
  = Left (Literal l)


liftToVal :: Expr → Expr
liftToVal expr
  = case liftToST expr of
    Left e  → e
    Right _ → error "internal error in liftToVal"
```

Interpreter

Data type *Value* models closures and the possible results of their evaluation. The constructor *Blackhole* stands for the value $\bot$. It results from the evaluation of self-referential expressions like **let** $x = x$ **in** x:

```
data Value = Closure Env Expr
           | Constr String [Thunk]
           | Oracle{ getOracle :: Oracle }
           | Lit{ getLit :: CoreLit }
           | Blackhole
```

Environments are simple lists of pairs:

```
type Env = [(String, Thunk)]
```

A thunk is an updateable pointer to a result value:

```
type Thunk = IORef Value
```

Function *evaluate* evaluates an expression in an empty environment and return a *Value* as result. The definition of *Prelude*; $\bot$ has to be added here, because every identifier that occurs as a parameter is looked u, even if the corresponding thunk is never evaluated:

```
evaluate :: [Thunk] → Expr → IO Value
evaluate args expr
  = do undef ← newIORef Blackhole
       eval args [("Prelude;undefined", undef)] expr
```

Function *eval* takes a argument stack, an environment and an expression and returns the result of its evaluation:

```
eval :: [Thunk] → Env → Expr → IO Value
eval stack env expr
```

```
= case expr of
  Con c xs
    → return (Constr c (map (lookupVar env) xs ++ stack))
  Var vn          → evalVar stack vn
  App c a         → eval (lookupVar env a : stack) env c
  Abs x c → case stack of
              []       → return (Closure env $ Abs x c)
              (s : ss) → eval ss ((x, s) : env) c
  Case c alts → eval [] env c >>= match alts
  Let bs c → do env' ← bindAll bs
                eval stack env' c
  Literal l
    | null stack  → return (Lit l)
    | otherwise   → error ("tried to apply literal" ++ show l)
where
evalVar [t] "Prelude;Prelude.Int;Prelude.Enum;pred"
  = evalThunk t >>= return ∘ Lit ∘ decrement ∘ getLit
evalVar [] "newOracleEntry"
  = fmap Oracle newOracleEntry
evalVar [t] "enterRhs"
  = do v ← evalThunk t
       o ← enterRhs (getOracle v)
       return (Oracle o)
evalVar [t] "leaveRhs"
  = do v ← evalThunk t
       leaveRhs (getOracle v)
       return (Constr "()" [])
evalVar s x = evalThunk (lookupVar env x) >>= evalValue s
lookupVar e x = fromMaybe
                  (error $ "free variable: " ++ x)
                  (x `lookup` e)
bindAll bs = mfix (λenv' → foldM (insertRef env') env bs)
  where
  insertRef env' env'' (x, e)
    = do r ← newIORef (Closure env' e)
         return ((x, r) : env'')
match ((PatDefault, rhs) : _) _
  = eval stack env rhs
match ((PatCon pn xs, rhs) : _) (Constr cn as)
  | cn ≡ pn ∧ length as ≡ length xs
  = eval stack (zip xs as ++ env) rhs
match ((PatLit l, rhs) : _) (Lit l')
  | l ≡ l'
```

```
    = eval stack env rhs
  match (_ : alts) v
    = match alts v
  match [] _
    = error "no match"
  decrement (CoreInt n) = CoreInt (n - 1)
  decrement v = error $ "cannot decrement" ++ show v
```

Function *evalThunk* evaluates the contents of a thunk and updates it with the result:

```
evalThunk :: Thunk -> IO Value
evalThunk thunk = do val <- readIORef thunk
                     writeIORef thunk Blackhole
                     val' <- evalValue [] val
                     writeIORef thunk val'
                     return val'
```

Function *evalValue* takes an argument stack and a *Value*. If the value is aclosure, then this closure is evaluated, and the result is applied to the arguments from the argument stack. If the value is a constructor term, then the arguments are appended to it as additional components. If argument stack is ampty and the value is neither a closure nor a constructor, then the value is returned unchanged. Otherwise the evaluation will stop with a runtime error.

```
evalValue :: [Thunk] -> Value -> IO Value
evalValue s (Closure e c) = eval s e c
evalValue s (Constr c as) = return (Constr c $ as ++ s)
evalValue [] v = return v
evalValue _ _ = error "tried to apply non-function"
```

Function *valueToString* turns a *Value* into a textual representation.

```
valueToString :: Int -> Value -> IO String
valueToString level val
  = case val of
    Closure _ (Abs _ _)
      -> return "<FUN>"
    Closure _ _
      -> return ("<THUNK>")
    Constr "Prelude.:" [car, cdr]
      -> do car' <- (readIORef >=> valueToString 1) car
            cdr' <-       (readIORef >=> valueToString 1) cdr
            return (car' ++ " : " ++ cdr')
```

```
    Constr c []
      → return (stripPrelude c)
    Constr c as
      → do as' ← mapM (readIORef >=> valueToString 2) as
           return $ mkBraces 1 $ ppConstr (stripPrelude c) as'
    Oracle _
      → return "<ORACLE>"
    Blackhole
      → return "undefined"
    Lit l → return (show $ CoreLit l)
    where
    ppConstr c as
      = intercalate " " (c : as)
    stripPrelude c = fromMaybe c (stripPrefix "Prelude;" c)
    mkBraces l s
      | l < level  = '(' : s ++ ")"
      | otherwise = s
```

Function *popList* turns the current oracle into a list, so that t can be used for Lazy Call-by-Value evaluation.

```
popList :: Thunk → IO Thunk
popList t = popOracle >>= consOracle
  where consOracle (−1) = return t
    consOracle n = do
      v ← newIORef (Lit (CoreInt n))
      l ← newIORef (Constr "Prelude;:" [v, t])
      popList l
```

Function *main* loads a program and evaluates it.

```
main :: IO ()
main = do
  args ← getArgs
  let (corePath, mainexpr)
    = case args of
      [f, e] | ';' ∈ e   → (f, e)
             | otherwise → (f, "Main;" ++ e)
      _ → error "usage: lcbv-inter file.yca mainexpr"
  coreexpr ← loadCoreExpr corePath (Var mainexpr)
  let withSplitLets   = postOrder splitLets coreexpr
      withoutMutual = postOrder rmMutualRecursion withSplitLets
      transformed     = postOrder instrument withoutMutual
  putStrLn "\n{- ** Result of lazy evaluation ** -}\n"
  initOracle
```

```
val ← evaluate [] transformed
valstring ← valueToString 0 val
putStrLn valstring
let lcbvexpr = liftToMonad withoutMutual
putStrLn "\n{- ** Oracle ** -}\n"
nil ← newIORef (Constr "Prelude;[]" [])
oracle ← popList nil
orcstr ← readIORef oracle >>= valueToString 0
putStrLn orcstr
let runLcbvApp = (mkFunapp "Lcbv;runLcbv" [lcbvexpr])
lcbvexpr' ← loadCoreExpr "Lcbv.ycr" runLcbvApp
putStrLn "\n{- ** Result of LCBV evaluation ** -}\n"
lcbvval ← evaluate [oracle] lcbvexpr'
lcbvvalstring ← valueToString 0 lcbvval
putStrLn lcbvvalstring
```

B.2 Implementation of State Monad

```
module Lcbv where

type ST a = [Int] → ([Int], a)

unit :: a → ST a
unit x s = (s, x)

bind :: ST a → (a → ST b) → ST b
bind x f s = case x s of
  (s', v) → f v s'

funApp :: ST (a → ST b) → a → ST b
funApp t u s = case t s of
  (s', v) → v u s'

optionalBind :: ST a → (a → ST b) → ST b
optionalBind p q orc
  = case orc of
    0 : o → q ⊥ o
    n : o → case p (pred n : o) of
              (s', v) → q v s'
    _ → ⊥
```

```
optionalApp :: a → (a → ST b) → ST b
optionalApp p q orc
  = case orc of
    0 : o → q ⊥ o
    n : o → q p (pred n : o)
    _ → ⊥


fixBind :: (a → ST a) → (a → ST b) → ST b
fixBind p q orc
  = case orc of
    n : o → let r :: Int → a → [Int] → ([Int], a)
              r 0 x s = (s, x)
              r n x s = case p x s of
                (s', v) → r (pred n) v s'
            in (bind (r n ⊥) q) o
    _ → ⊥


runLcbv :: ST a → [Int] → a
runLcbv e o = case e o of
  (_, e') → e'
```

B.3 Oracle Creation

The following C code implements the functions for oracle creation described in section 4.2 on page 57.

```
#include <stdlib.h>

typedef struct Oracle {
  struct Oracle *last,*next;
  int count;
} *OraclePtr;

static struct Oracle startNode;
static OraclePtr current;

void initOracle(void) {
  current = startNode.next
    = startNode.last = &startNode;
  startNode.count = 0;
}
```

```
OraclePtr newOracleEntry(void) {
  OraclePtr o;
  if ((o = malloc(sizeof * o)) == 0)
      exit(EXIT_FAILURE);
  o -> last  = current->last;
  o -> next  = current;
  o -> count = 0;
  current->last->next = o;
  current->last = o;
  return o;
}

OraclePtr enterRhs(OraclePtr o) {
  OraclePtr saved = current;
  o->last->count++;
  current = o;
  return saved;
}

void leaveRhs(OraclePtr saved) {
  OraclePtr o = current;
  current = saved;
  o->last->count += o->count;
  o->last->next = o->next;
  o->next->last = o->last;
  free(o);
}

int popOracle(void) {
  OraclePtr l = startNode.last;
  int c = l->count;
  if (l == &startNode) {
    startNode.count = -1;
  } else {
    startNode.last = l -> last;
    l->last->next = &startNode;
    free(l);
  }
  return c;
}
```

Appendix C

Contents of the Supplemental CD-ROM

The supplemental CD-ROM contains the the reference implementation as well as the source files of this thesis:

File `README`
: A short explanation of the disk's contents.

File `DiplomaThesis.ps`
: This thesis in PostScript format.

Directory `TeXSource`
: The source files of this document in LaTeX format. To create the document `DiplomaThesis` from it, enter the directory and type `make` at the command prompt. The following tools are required:

  - The preprocessor `lhs2TeX` is used to format embedded Haskell code.
  - GNU `make` and the LaTeX building system `rubber` are used to coordinate the calls to `latex`, `bibtex` etc..
  - The graph layouting program `dot` and the PostScript tool `ps2eps` are used for some illustrations.

Directory `Implementation`
: The Haskell source files of the reference implementations. To create the executable programs `lcbv-interpreter` and `core-interpreter` from it, enter the directory and type `make` at the command prompt. The Haskell build system `cabal` is used to coordinate the build process, and the Haskell library `yhccore` is required. It has been tested with GHC-6.8.3 and `yhccore-0.9`.

File `lcbv-interpreter`
: The implementation of the semantics from Chapter 3 as an executable

program. It is compiled by GHC-6.8.3 on an x86 Linux system with `glibc-2.7`.

File `core-interpreter`
: The implementation from Appendix B as an executable program. It is compiled by GHC-6.8.3 on an x86 Linux system with `glibc-2.7`.

Directory `Implementation/Examples`
: Some small examples that can be used to test the interpreters. For an example program `Test.hs`, typing `make Test.yca` on the command line lets the York Haskell Compiler create a YHC Core file from it.

  Then the YHC Core file can be evaluates by typing `core-interpreter Test.yca mainexpr` or `lcbv-interpreter Test.yca mainexpr`, where `mainexpr` is the name of the declaration to be evaluated.