Combining Memoisation and Change Propagation for Automatic Incremental Evaluation of Haskell Arrow Programs

Alessandro Vermeulen

August 26, 2013

Abstract

Implementing re-use of previously computed values is hard and is therefore often dismissed by programmers: a missed opportunity for improved computational efficiency. Memoisation and change propagation are techniques for implementing the re-use of previously computed values. Using existing libraries either destroys the conciseness of the code or is too limited for a real-world application where fine-grained control over the amount of memory used is necessary. We combine both memoisation and change propagation and provide clean interfaces through the ubiquitous applicative, monad, and arrow interfaces.

Contents

Chapter 1

Preamble

1.1 Introduction

We unnecessarily waste energy and decrease performance by repeatedly recomputing values. Re-using previously computed values increases performance and thereby reaction time as less computations areas the processor is less active and when it is active it is for shorter periods of time. Long reaction times have a negative impact on user experience and on the energy efficiency of the computer.

1. **User experience** - The user has to wait longer for results and in some cases desired features, such as automatically updating the table of contents of a large Word document, are not available because they would cost too much performance.

2. **Energy efficiency** - Processors are active unnecessarily frequent and for unnecessary long periods of time.

Performing the same computation repeatedly is equal to not re-using the results of previously performed computations. Re-using results from previously computed results is known as incremental evaluation.

incremental evaluation

Because of the absence of side-effects in purely functional languages, programs written in such languages are easier to incrementalise than in stateful, imperative, languages: when a previously evaluated expression is encountered we know that re-evaluating it will yield the same result; therefore, it is safe to re-use the result of the previous evaluation.

The foremost reason for the lack of the re-use of previously computed values is the additional programming effort it requires. The implementation of an algorithm in a programming language is less concise due to implementation de-tails, thus the implementation does not resemble the original algorithm strongly. Adding the re-use of previously computed values is another implementation detail and completely destroys any residual conciseness and resemblance to the original algorithm, thus further reducing the readability and maintainability of the program. Besides the initial effort required to add the re-use of previously computed results, the addition also increases the effort necessary for future updates. Additionally, real- life applications require the possibility of fine- tuning of how and when to re- use values which is absent in current solutions for the re-use of previously computed results. In this thesis we show how to provide for this fine- tuning.

It is necessary to know whether a function is side-effect-free to safely assume that the result of a previous computation can be re-used without changing the observable behaviour of the application. In order to not make the problem more complicated than necessary this thesis is embedded in the context of the purely functional language Haskell (S. Peyton Jones, 2003).

In particular, there are two techniques for implementing the re-use of previously computed results: memoisation and change propagation. Memoisation has been investigated extensively in many languages, including Haskell, and is known as caching in imperative languages. Little is known, however, on implementing change propagation in Haskell. Change propagation has primarily been research in the context of Attribute Grammars (AG) (Reps et al., 1983) and Standard ML (U. A. Acar, Blelloch, and Harper, 2002). Both memoisation and change propagation are techniques that can be used to achieve incremental evaluation.

incremental evaluation

In this thesis we present a library for easily adding support for the re-use of previously computed values without drastically changing the layout of the program. We evaluate our implementation by running some benchmarks that show that the desired effect is indeed achieved.

The remainder of this introductory chapter consists of the explanation of the technical terms and techniques used in this paper: memoisation is explained in subsection 1.2.1 and change propagation in subsection 1.2.2, followed by their differences in subsection 1.2.3. We present our goal in more detail in section 1.3. The final research question and the sub research questions are presented in section 1.4, followed by our research methods in section 1.5. We conclude with an overview of the remainder of this thesis in section 1.7.

1.2 Memoisation, Change Propagation en their differences

Memoisation and change propagation are at the core of the solution presented in this thesis and we explain them in the following sections, starting with memoisation, followed by change propagation and concluding with the differences between the two techniques.

1.2.1 What is memoisation?

The concept of memoisation is well known. Hughes describes memo-functions as follows: “Memo-functions are functions that remember the arguments they are applied to, together with the results computed from them.” (Hughes, 1985). This means that a memo-function can lookup the result for a certain input if it has seen the input before, instead of having to recompute the result again. The process that creates a memo-function out of an ordinary function is called memoisation.

We further explain what memoisation is on the basis of the textbook example of memoisation: the Fibonacci sequence. The function *fib* n is the function that calculates the n^{th} Fibonacci number. The default implementation of *fib* in Haskell reads as follows:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

This definition of *fib* is not efficient because for every call of *fib* two additional calls to *fib* occur and each path to *fib* 0 or *fib* 1 is at maximum n calls long; thus, this function has a run-time complexity of $O(2^n)$. We can improve this to a complexity of $O(n^2)$ by storing the results of *fib* in a Haskell list. The lookup of *fib* n has a complexity of $O(n)$ but we only perform one lookup for every n; therefore, *mfib* has a final run-time complexity of $O(n^2)$.[1]

In the following implementation we use a standard Haskell list to store our results and use the operator !! to lookup our result in the list. We will see that there are more methods for storing our inputs and results in section 3.6 and subsection 6.1.1. The advantage of using a list is that it does not require additional memory to store the input of our function, in contrast to the commonly used technique of storing input and value together in a list or map. The principle used here is the same principle as used in implementing memo-tries as described in section 6.1.1.

```
mfib = (map fib [0 ..] !! )
  where
    fib 0 = 0
    fib 1 = 1
    fib n = mfib (n - 2) + mfib (n - 1)
```

[1] In section 7.1, we see that a linear implementation of *fib* can also be given.

This implementation has the aforementioned complexity because the result of $map\ fib\ [0\,..]$ is shared between the recursive calls to $mfib$ in the definition of fib; therefore we call $fib\ n$ only once for every n. Additionally, due to Haskell's lazy-evaluation the list is only evaluated up to the n^{th} element; therefore, this function terminates in a finite amount of time.

Side-effects Note that we can only perform memoisation on a pure function. We cannot re- use the previously computed results of a function that has a side-effect without risking to change the observable behaviour, and therefore the semantics, of our program; moreover, the result of the new function call could be different due to possibly different global state.

Levels of sharing There are different levels of sharing. In the example above the list is shared between all calls to fib within the initial call to fib; however, the list is not shared between the two different calls to fib in an expression like $fib\ n + fib\ n$. This means that evaluating this expression takes twice the time to calculate $fib\ n$ and a constant for the summation. If the list was shared, the evaluation would only take as long to calculate $fib\ n$ once plus the time it takes to lookup the result in the list. Therefore, we can discern at least three levels of sharing.

1. Memoisation on the outer call
2. Memoisation on the outer call and contained calls
3. Memoisation between outer calls and contained calls

In the remainder of this thesis we refer to the last level of sharing as Third-level Sharing. A sub goal for our implementation of memoisation, as described in section 1.3, is to achieve this third level of sharing.

Third-level Sharing

1.2.2 What is change propagation?

We call a program *reactive* if it responds to changes in its input by using change propagation. A reactive program is the sum of two parts:

1. The Reactive model, or *model*, is the run-time representation of an expression by variables and the dependencies between them, also known as the data-dependency graph. There are input variables containing input of the calculation, and output variables that contain the result.
2. The *mutator* (U. A. Acar, Blelloch, Blume, et al., 2009) that updates the input variables and propagates the changes from the input variables to the output variables.

Reactive model

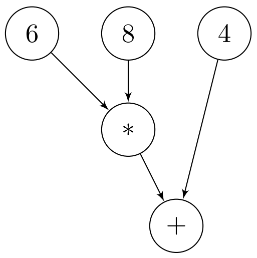


Figure 1.1: Graph representation of the calculation $xyz\ 4\ 6\ 8$. The arrows indicate the direction of the data-flow. The arguments of the operators are not depicted.

We define change propagation as the process of updating the result of a calculation by computing the effect of a change in a value of an argument to a calculation on its result where only those parts of the calculation that depend on the changed input are recomputed.

We illustrate this process by comparing a simple calculation and its reactive counterpart. We use the expression $x + y * z$, in which x, y, and z are input variables, which can be modelled in Haskell with the function xyz:

$$xyz :: Int \to Int \to Int \to Int$$
$$xyz\ x\ y\ z = x + y * z$$

If we apply the function xyz to the values 4, 6 and 8, we get the calculation $4 + 6 * 8$ as illustrated in Figure 1.1.

Now we proceed by applying the function to the values 10, 6 and 8 we get the calculation $10 + 6 * 8$. Note, however, that we again evaluate the, unchanged, expression $y * z$ which translates to the computation $6 * 8$ as illustrated in Figure 1.2. We want to avoid the evaluation of unchanged expressions and thereby the computation of unchanged values.

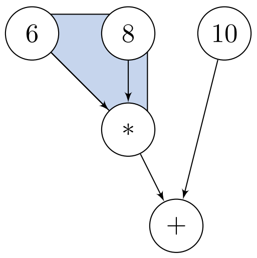


Figure 1.2: The calculation of $10 + 6 * 8$ with the repeated calculation of $6 * 8$ highlighted with a blue background.

Change propagation prevents the repeated evaluation of (sub)expressions by locally storing the values of each (sub)expression and by only recomputing the value of an expression if the value of a sub-expression has changed.

We can implement the remembering of values by associating a variable to each expression. A variable represents the location of changeable value in the memory. We introduce the type $Var\ \alpha$ for a variable of type α and we lift the operators $+$ and $*$ to work on variables of integers. We do the same with the function xyz leaving only the types changed:

$$\begin{aligned}
(+) &:: Var\ Int \to Var\ Int \to Var\ Int\\
(*) &:: Var\ Int \to Var\ Int \to Var\ Int\\
xyz &:: Var\ Int \to Var\ Int \to Var\ Int \to Var\ Int
\end{aligned}$$

The function $+$ yields a variable containing the result of the summation of the values of the two arguments and similarly for $*$.

Assume that we have three variables x, y and z of type $Var\ Int$ with the values 4, 6, and 8, respectively. Applying the function xyz to these variables results in a structure of variables with dependencies between the variables. The structure is illustrated in Figure 1.3. The squares are the variables representing the (sub)expression on the left hand side and the value of the variable is displayed inside the square.

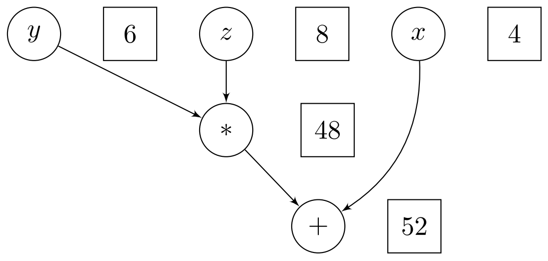


Figure 1.3: The calculation $4 + 6 * 8$ with the variables illustrated as rectangles to the right of the associated expression.

We now proceed by illustrating how change propagation works. We change the value of the variable x to 10; hereby we have created a situation where the value of the summation is invalid as illustrated in Figure 1.4. An invalid variable is a variable with an invalid value and is represented by a grey square.

By creating an invalid variable we have put our reactive model in an invalid state and this has to be fixed. We return the model to a valid state by re- evaluating every expression that is a dependant of x; i.e., we *propagate* a change in the value of x to all children of the expression x in the graph. In this case, we only have to perform a single update. We do this by recomputing the result of $+$ and storing

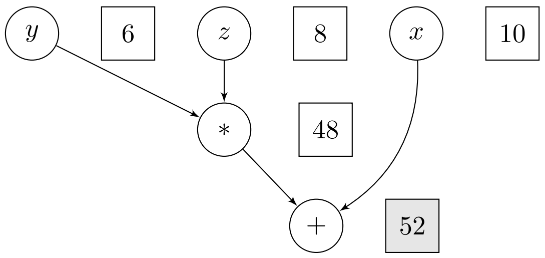


Figure 1.4: The situation of 1.3 after changing the value of x to 10. The value of the summation is now invalid as the value of x has changed.

the result in the associated variable as illustrated in Figure 1.5. We now have prevented the recomputation of $6 * 8$.

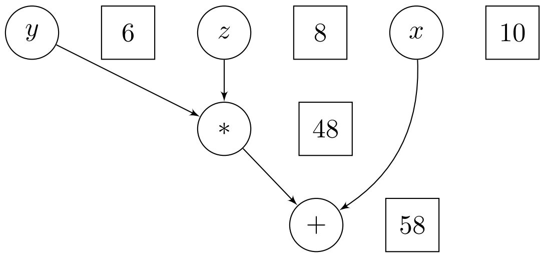


Figure 1.5: The situation after propagating the change in the value of x as illustrated Figure 1.4.

The previous example was a simple case. We now repeat the process for a more complex situation. This time we change to value of y to 2 creating the situation as depicted in Figure 1.6 where the value of $y * z$ has become invalid.

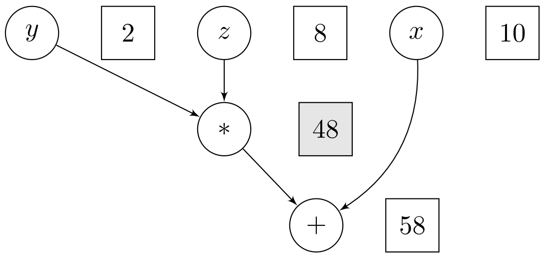


Figure 1.6: The situation as in Figure 1.5 after changing the value of y to 2.

We correct this invalid state by recomputing $y * z$ and storing the result in the variable associated with $y * z$. While now the sub-graph representing $y * z$ is now valid the graph representing $x + y * z$ is still invalid as the value of the summation has become invalid as depicted in Figure 1.7.

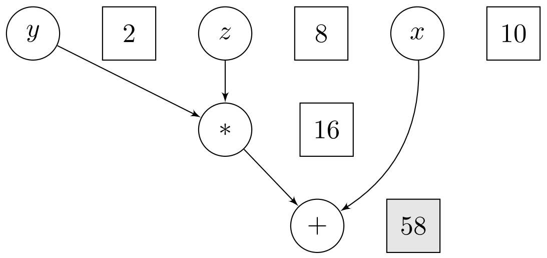


Figure 1.7: After propagating the change in Figure 1.6. The result of the addition has become invalid, depicted with the grey square.

By recomputing the value of the summation we bring the model to a valid state again as illustrated in Figure 1.8.

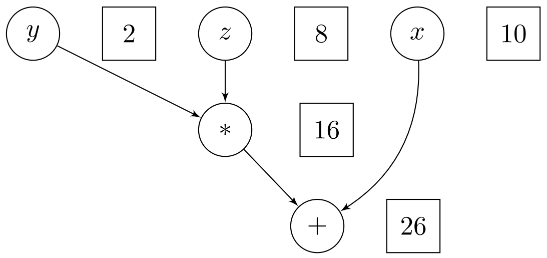


Figure 1.8: The final situation after updating the value of the addition from Figure 1.7.

This process of propagating changes through the model is change propagation.

A downside of remembering the value of each sub-expression is that it can be disadvantageous because it can cost a lot of memory. By not remembering the value of every sub-expression, we can perform computations with very large sub-expressions without risking a memory shortage.

Some might argue that change propagation is equivalent to memoisation with a table size of 1. However, with change propagation both results of applying $+$ twice are remembered by the variables created by calling $+$, whereas only one result would be remembered when using memoisation with a memo-table size of 1.

Outside the context of Haskell, there are good implementations for evaluating an Attribute Grammar (Reps et al., 1983) (AG) with change propagation as well as for the programming language Standard ML (Milner et al., 1997) (ML) as we will see in Related Work (chapter 6) and specifically in subsection 6.1.2.

1.2.3 Differences between memoisation and change propagation

As we have seen, both memoisation and change propagation can be used to prevent the repeated recomputation of the same calculation. Why are we interested in their combined application? The answer is simple: by combining the two techniques we expect to prevent more recomputations.

Conceptually the two techniques are different as well. Memoisation prevents recomputation if a function is called; whereas change propagation prevents the function from being called. Memoisation looks at inputs and results of functions where change propagation uses the values of expressions. Expressing this in terms of a graph: with every node a variable, change propagation happens on the nodes and memoisation happens on the edges between nodes.

There are situations where memoisation prevents recomputation better than change propagation does and vice versa. We present a scenario for each situation, starting with a scenario where memoisation is better suited for preventing recomputation than change propagation.

If the input of a reactive program oscillates between two values the program efficiently calculates the new result of each change; however, on a larger scale it repeatedly performs the same calculations. A memoised function would compute and then store the two results; therefore, subsequent calls would only involve a lookup. As we will see later, by combining the two techniques we can efficiently compute the result associated with the second value by using change propagation and then store it.

The scenario that shows that change propagation can be more effective than memoisation is more complicated. The following scenario is adopted from the work of Acar (U. A. Acar, Blelloch, Blume, et al., 2009).

The calculation in this scenario is the *quicksort* sorting algorithm for lists. This algorithm takes the first element of the list, the *pivot*, and divides the remainder of the list in two lists: a list with elements smaller than the pivot and a list with elements greater than or equal to the pivot. Both lists are then sorted by recursively applying $quicksort$ and the resulting sorted lists are concatenated with the pivot in between.

The state of the reactive model after sorting the list $[11, 7, 4, 17, 5, 19, 9]$ is illustrated in Figure 1.9. The functions qs and sp are the functions for *quicksort* and the splitting of the list based on the pivot respectively. The expression $qs\ (11)$ is used for denoting a call to qs with the list that starts with 11. The control-flow is depicted with the thin arrows; i.e., the call $qs\ (11)$ applies the function sp on the list starting with 7.

Inserting the element 8 between elements 17 and 5 changes the list from element

17 and further. The whole list now reads $[11, 7, 4, 17, 8, 5, 19, 9]$ and we have not applied $quicksort$ to this list before; therefore, memoisation has no effect and without change propagation we would have to sort the entire list again.

With change propagation we have to redo the first partitioning from element 8 and further. This results in a change in the list with elements smaller than 11 so we have to sort that one again. This time the list with elements greater or equal than the pivot has changed. We sort this list again. All sorted lists are now again concatenated an we end up with a sorted list.

In Figure 1.9 the function calls that are avoided with change propagation and which would have been performed if we had used only memoisation are highlighted with the grey background.

1.3 Goals

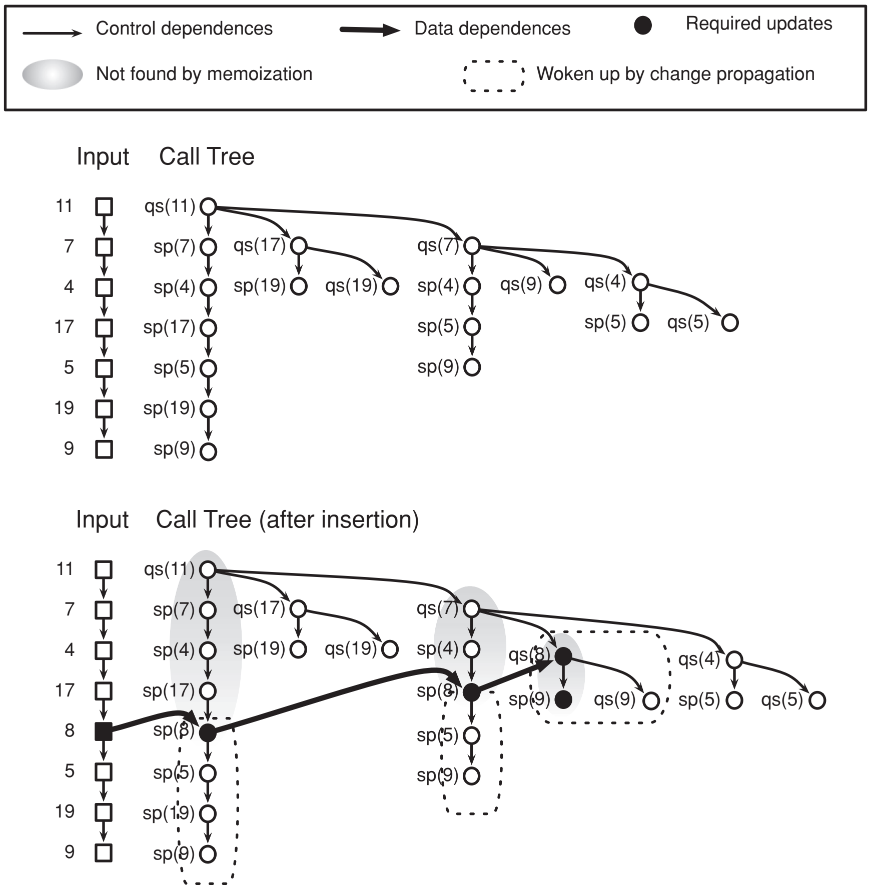
We have seen what memoisation is (subsection 1.2.1), what change propagation is (subsection 1.2.2) and we have determined that they are two intrinsically different techniques that serve their own purpose in preventing recomputation (subsection 1.2.3).

Sadly they are often not used; firstly, due to the difficulty to implement them or because the available libraries are lacking in the availability of fine-tuning necessary for real-world applications; secondly, because the provided libraries tend to reduce readability, with the notable exception of the latest contribution by Chen et al. (Chen et al., 2012) and several purely functional $memo$ functions; and thirdly, because none of the existing libraries implement the combination of memoisation and change propagation.

Our primary goal is therefore to apply the *combination* of the techniques memoisation and change propagation to improve the performance of a reactive Haskell program when it reacts to changes in input *and* give the programmer access to the behaviour of the memoisation and change propagation, *without* changing the compiler or the language, in an easy to use format that does not alter the appearance of the code drastically.

Furthermore, where memoisation is applied it should use the third level of sharing where multiple calls to the same memoised function use the same memo-table because this further increases the re-use of previously computed results.

We illustrate how we want the combination of memoisation and change propagation to behave by applying the combination to the *fib* function from subsection 1.2.1. We depict calculations as a graph as depicted in Figure 1.10: The values are represented by the nodes and the functions are represented by the



```
fun qs(nil,r)   = r
  | qs(h::t,r) = let (l,g) = sp(h,t)
      in qs(l,h::qs(g,r))
and sp(p,nil)   = (nil,nil)
  | sp(p,h::t) = let (l,g) = sp(p,t)
      in if h<p then (h::l,g) else (l,h::g)
```

Figure 1.9: Example of a scenario(U. A. Acar, Blelloch, Blume, et al., 2009) where change propagation prevents work in contrast to memoisation.

edges. We can read the graph as follows: the value of variable y depends on the value of variable x and is obtained by applying f to x.

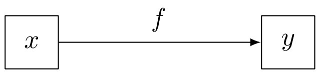


Figure 1.10: The value of variable y depends on the value of variable x and is calculated by applying f to the value of x.

We now show how we expect the combination of change propagation and memoisation to behave using the *fib* example. If we start with calculating *fib* 2 two additional calls to *fib* take place: *fib* 0 and *fib* 1 resulting in the situation illustrated in Figure 1.11.

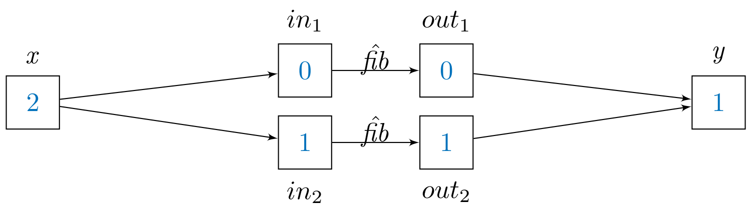


Figure 1.11: State of the reactive model after calling *fib* with a variable with value 2. The squares represent a variable and the edges functions. Two converging edges means $+$.

Change propagation causes the values of the variables for the $(n-2)$ and for the $(n-1)$ branches to change when the value of n changes; i.e., changing the value of n to 3 changes the value of in_1 to 1 and the output, out_1, becomes 1. The value of in_2 becomes 2. As we saw before, for *fib* 2 two additional calls to *fib* take place and they create two additional pairs of variables: (in_3, out_3) and (in_4, out_4). Subsequently, the value of out_2 will be updated to the sum of the values of out_3 and out_4 and becomes 1. The value of y is updated to the sum of out_1 and out_2 and becomes 2. The current situation is illustrated in Figure 1.12.

Note that the sub-graph highlighted by the blue background, representing the calculation of *fib* 2 is identical to the graph in Figure 1.11 up to renaming of variables. To prevent the recomputation of *fib* 2 we use memoisation. The memoised *fib* function is $\hat{fibm}$. Memoising a function over variables is analogous to memoising a function over values: As soon as a result is computed it is stored in the memo-table together with its input. When the input variable changes, the new value is first looked up in the memo-table and if it exists the corresponding result value is assigned to the result variable. Note that we choose to store the values and not the variables. Storing the variables could potentially be more efficient but it also makes the implementation more difficult. The situation illustrated in Figure 1.13 depicts the same situation as Figure 1.12 but with the

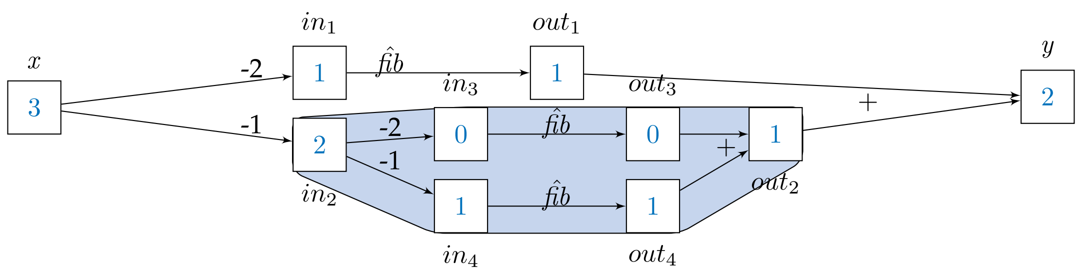


Figure 1.12: The reactive model after changing the value of variable x to 3 without applying memoisation.

addition of memoisation.

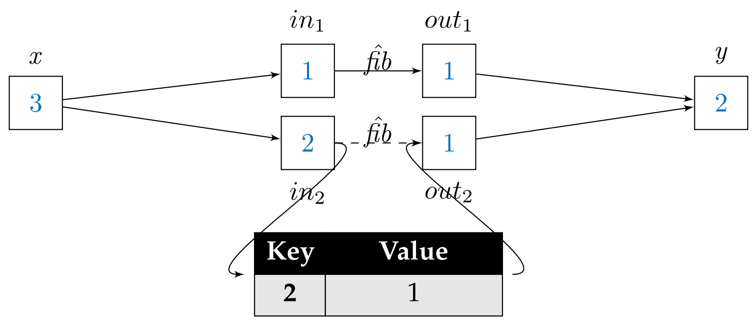
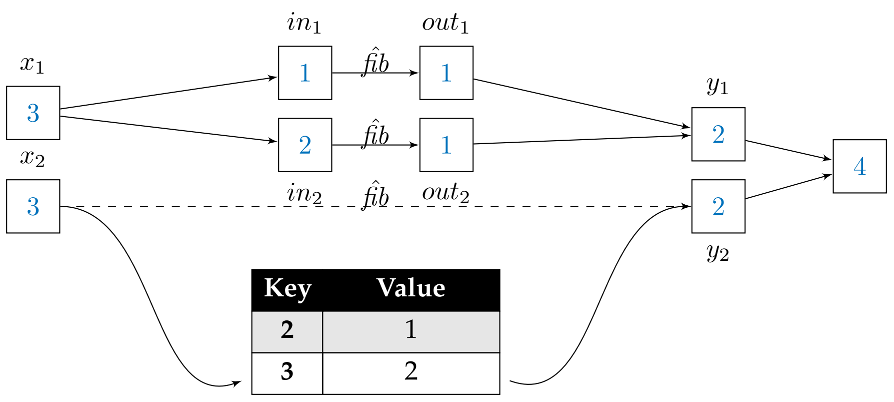


Figure 1.13: The sitation of the reactive model after changing the value of x to 3 when applying memoisation together with change propagation.

Now imagine the calculation *fib* $3 +$ *fib* 3 as we gave at the end of the explanation of memoisation (subsection 1.2.1) with $n = 3$. Our goal is to calculate the value of *fib* 3 only once and that the result of the second call of *fib* 3 is obtained by a lookup in a shared memo- table. This means that the there has to be sharing of the memo-table between the two calls to *fib*. We have illustrated a possible execution in Figure 1.14, note that we do not impose an order in the evaluation of the arguments to $+$.

1.4 Research question

How can we apply the *combination* of the techniques memoisation and change propagation to improve the performance of a reactive Haskell program when it reacts to changes in input *and* give the programmer access to the behaviour of the memoisation and change propagation, *without* changing the compiler or



Key	Value
2	1
3	2

Figure 1.14: A possible state of the reactive model after *fib* n + *fib* n with $n = 3$ has been calculated when sharing the memo-tables.

the language, in an easy to use format that does not alter the appearance of the code drastically?

1.4.1 Sub questions

1. **Stateless memoisation with fine-tuning of the memoisation** - How can we add memoisation to our pure functions and give the programmer control over the memoisation, without making the functions themselves side-effectful?
2. **Stateless change propagation** - How can we describe and consequently use change propagation in pure functions where we describe the changes in input and translate these to changes in output?
3. **Combination of Memoisation and Change Propagation** How can we combine the two intrinsically different techniques memoisation and change propagation into one system?
4. **Easy-to-use** How can we package this in an interface that is easy to use for the programmer and has a minimal effect on the conciseness of the program code?
5. **Third-level sharing** How do we achieve Third-level Sharing of memo-tables when applying memoisation?

Third-level Sharing

1.5 Research method

We choose to create a library instead of creating a compiler-extension, because a library is easier to maintain, and to show that Haskell is powerful enough to extend the possibilities of the language with powerful features without changing the compiler or the language.

During this thesis we encounter the following types of Haskell functions.

1. non-recursive functions on non-recursive datatypes;
2. recursive functions on non-recursive datatypes;
3. recursive functions on recursive datatypes;

The program $x + y * z$ is used to illustrate the simple programs. We then extend the simple program with recursion to obtain fib and finally we use $quicksort$ and Maximum Segment Sum (MSS) as examples of recursive programs over recursive data-types.

We show that using the methods presented in this paper increases the re-use of previously computed values. We compare our methods with native Haskell.

1.6 The chosen approach

During my thesis I started out with a straightforward implementation of change propagation and tried to add memoisation in a later stage. This proved to be difficult; therefore, I attempted a second approach based on memoisation in the first place. Both implementations have their advantages and disadvantages as we will see in the remainder of this thesis.

1.7 Overview of the remainder of this thesis

In the next chapter, chapter 2, we describe the first system we designed followed by the second system in chapter 3. We then relate both implementation to each other in chapter 4 followed by the benchmarks of the implementations and other solutions in chapter 5. Subsequently, we describe all related work in chapter 6 and we conclude in the last chapter, chapter 7.

Chapter 2

Change Propagation first

Change propagation and memoisation are intrinsically different techniques, as we explained in the introduction. In this chapter we approach the combination of both techniques by first implementing change propagation and adding memoisation afterwards. We will see that adding memoisation proves to be difficult.

We first introduce a low-level library for building a reactive program. Followed by the implementation of the first approach and the issues with this approach.

We intent to be able to write the functions *fib*, *quicksort*, and *mss* in the following form at the end of this chapter:

```
$fib\ n = \mathbf{do}$
  $l \leftarrow fib\ (n - 2)$
  $\rho \leftarrow fib\ (n - 1)$
  $return\ \$\ l + \rho$


$quicksort\ [] = return\ []$
$quicksort\ (x : xs) = \mathbf{do}$
  $(l, g) \leftarrow split\ x\ xs$
  $ls \leftarrow quicksort\ l$
  $gs \leftarrow quicksort\ g$
  $return\ (ls +\!\!+ [x] +\!\!+ gs)$
$split\ p\ [] = return\ ([], [])$
$split\ p\ (x : xs) = \mathbf{do}$
  $(l, g) \leftarrow split\ p\ xs$
  $\mathbf{if}\ x < p\ \mathbf{then}$
      $return\ (x : l, g)$
```

```
else
  return $(l, x : g)$
```

2.1 Overview of the reactive library

This library is a straightforward implementation of the idea change propagation. The library is based on the concept of a *variable*. A variable has a changeable value and every variable has a collection of variables that use the variable as source, the dependants or reactors. When a value of a variable changes the reactors of the variable are notified of the new value together with the old value. Every variable maintains his own record of reactors. reactors

The implementation of this system consists of two parts: the first part contains the basic functions for creating variables and for creating a dependency between two variables called *linking*, and the second part contains higher level functions modelled after the interfaces of the $Applicative$, $Functor$, and $Monad$ classes.

Because the aforementioned functions are stateful the functions run in IO. For simplicity reasons we have chosen to forgo any abstraction to other monads that support state such as the St monad. Because the functions run in a monad, the use of the functions is more cumbersome than the use of the operators they were derived from.

To fix this, we introduce a datatype that contains a constructor for each function in the interface that we model. We use these constructors as our Domain-Specific-Language (DSL). We can then interpret this DSL to obtain the same result as whether we wrote the stateful functions by hand rendering the code more concise and, therefore, cleaner and easier to read.

The library further has the following two properties.

1. **Lazy evaluation** The value of the variables is only evaluated when the value of the result variable is evaluated.
2. **Garbage collection** The library ensures that any variables that are dead can be garbage collected.

The following sections describe how the creating and linking of variables works. After that we introduce the stateful abstractions bases on the aforementioned classes. Subsequently, we describe how to abstract over the stateful abstraction by applying a technique called deep-embedding and how to interpret the abstraction.

2.1.1 The reactive library - basics

As implementation of a variable we use the standard Haskell type *IORef*. A variable has the type *RVar* α; i.e., it is a variable containing a value of type α. Additionally, the variable contains meta-data such as the collection of reactors.

```
type RVar α = IORef (RVal α)
data RVal α = RVal {
    rMeta  :: RMeta α
  , rValue :: α
  }
```

The meta-data consists of the reactors, the sources of the variable, the *IO* () actions that should be executed when the value of the variable changes, and debug information.

A new variable with an initial value is created with the function *newRVar*.

```
newRVar :: α → IO (RVar α)
```

We can lookup the current value of a variable with the function *getRVal* and assign it a new value with the function *setRVar*. For convenience, there *printRVal* function that prints the variable to the standard output exists.

```
getRVal   :: RVar α      → IO α
setRVal   :: RVar α → α → IO (RVar α)
printRVal :: RVar α      → IO ()
```

Two variables can be linked by the function *linkRVar*. The first argument becomes the reactor to the second argument. The third argument to *linkRVar* is the function that should be executed when the value of the source variable changes.

```
type OnValSet α = α → α → IO ()
linkRVal :: RVar β → RVar α → OnValSet α → IO HandlerLink
```

The typical use of the third argument is to set the value of the of the reactor by calling *setRVal*.

2.1.2 Stateful abstraction

We introduce a set of functions based on the interface of *Functor*, *Applicative*, and *Monad* to provide a familiar interface for the programmer. The functions *rValFMap* and *rValStar* are the counterparts of *fmap* and <*> respectively. The function $rValBind$ is the counterpart of $\gg=$ and is implemented in terms of $joinRVal$ and $rValFMap$.

```
rValFMap :: (α → β)       → RVar α        → IO (RVar β)
rValStar :: RVar (α → β)  → RVar α        → IO (RVar β)
joinRVal :: RVar (RVar α)                 → IO (RVar α)
rValBind :: RVar α        → (α → RVar β)  → IO (RVar β)
```

Compared to their counterparts the functions run in *IO*. We would like to write our programs in applicative or monad style but the extra *IO* makes this impossible because we cannot use these functions to define the instances for the three classes. We first show why this is a problem, followed by a solution in subsection 2.2.1.

Remember the calculation $x + y * z$ from the introduction. Written in Applicative style it, with *I* the identity functor, would read as follows.

```
xyz :: Integer → Integer → Integer → I Integer
xyz x y z = (+) <$> pure x <*> ((*) <$> pure y <*> pure z)
```

However, as we do not support this style yet we have to write this by using the associated functions from our library which would be done as follows.

```
xyz x y z = do
  ytz   ← do yt ← rValFMap (*) y
               rValStar yt z
  xp    ← rValFMap (+) x
  xpytz ← rValStar xp ytz
  return xpytz
```

Obviously, this is less concise than our original calculation, $x + y * z$, and less concise than our applicative implementation and therefore less readable. We can now use the created reactive model by wrapping it in a mutator as follows.

```
main :: IO ()
main = do
  x ← newRVar 4
  y ← newRVar 6
  z ← newRVar 8
  out ← xyz x y z
  printRVal out   -- Prints "52"
  setRVal x 10
  printRVal out   -- Prints "58"
```

We create the input variables first and then construct the reactive model by applying xyz to the variables. The first time that we request the result of xyz, the value of variable out, it will be equal to 52. When we change the value of x to 10, as in our example from the introduction, we will see that the value of out is 58.

In the following sections we show how to use this library to achieve change propagation without writing the whole program in IO.

2.2 Implementation

We want our library to be user-friendly and therefore writing functions with it should be easy. The goal is to keep the advantage that Haskell code closely resembles the original algorithm. Besides reduced conciseness, we have now introduced IO in a calculation that is essentially side-effect- free, which is undesirable.

In the next section we show how we can eliminate the IO in order to be able to write our functions in applicative style.

2.2.1 The Representation

In this section we introduce the stateless representation of our programs and translate the representation to a Reactive model.

Reactive model

Note that the only difference between the types of our functions and the types of the functions from our model classes is that our functions run in IO. We only use IO as our implementation monad and it does not effect the semantics of the structure we are trying to express so can we somehow leave it out? Yes we can. We create a DSL, essentially delaying the IO actions until the interpretation phase.

We create a datatype with a constructor for each method we want to support in our DSL. This technique is called Deep Embedding and is common when working with DSLs. A deep embedding captures the *semantics* of a language and enables multiple interpretations of the model, whereas a shallow embedding allows for a single interpretation only. This embedding is the *representation* of our *reactive model*, *representation* for short.

Deep Embedding

Our datatype is a Generalised Abstract DataType (GADT) called R. We first introduce the support for the $Applicative$ class and we add the monadic interface in subsection 2.2.3 afterwards.

```
data R α where
  FMap ::    (α → β) → R α → R β
```

$$\begin{array}{lll} \mathit{Star} & :: R\ (\alpha \to \beta) \to R\ \alpha & \to R\ \beta \\ \mathit{Pure} & :: \alpha & \to R\ \alpha \\ \mathit{Input} & :: \mathit{RVar}\ \alpha & \to R\ \alpha \end{array}$$

The instances of $\mathit{Functor}\ R$ and $\mathit{Applicative}\ R$ are trivial, each function from the class maps to its corresponding constructor and, therefore, omitted from this text. In essence this datastructure represents the calculation the programmer wrote, however, delaying the actual creation of the run-time variables.

We can now use this embedding to write xyz in applicative style.

$$\begin{array}{l} \mathit{xyz} :: R\ \mathit{Integer} \to R\ \mathit{Integer} \to R\ \mathit{Integer} \to R\ \mathit{Integer} \\ \mathit{xyz}\ x\ y\ z = (+) \mathbin{<\$>} x \mathbin{<*>} ((*) \mathbin{<\$>} y \mathbin{<*>} z) \end{array}$$

Note that the parametrisation of the model happens at the Haskell level. The model only describes how to create a variable of type α, lacking the concept of input. Concretely, the function xyz is a Haskell function that takes three representations of values and creates a new representation of value and it is not a representation of a function that takes three values and constructs a new value.

Now that we have this representation of our reactive model we want to create the reactive model in order to actually create our values. In the following section we describe how we can interpret this representation to obtain our model.

2.2.2 Interpreting the Representation

In this section, we show how to interpret the DSL in order to actually perform the stateful operations the programmer had in mind. The implementation of the interpretation follows directly from the datatype and the functions we have available from our reactive library. The function $\mathit{install}$ creates the reactive model.

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{install} :: R\ v \to \mathit{IO}\ (\mathit{RVar}\ v)} \\ \mathit{install}\ (\mathit{FMap}\ f\ \mathit{rv}) & = \mathbf{do} & v \leftarrow \mathit{install}\ \mathit{rv} \\ & & f\ \text{`}\mathit{rValFMap}\text{`}\ v \\ \mathit{install}\ (\mathit{Star}\ \mathit{rf}\ \mathit{rv}) & = \mathbf{do} & v \leftarrow \mathit{install}\ \mathit{rv} \\ & & f \leftarrow \mathit{install}\ \mathit{rf} \\ & & f\ \text{`}\mathit{rValStar}\text{`}\ v \\ \mathit{install}\ (\mathit{Pure}\ x) & = \mathit{newRVar}\ x & \\ \multicolumn{3}{l}{\mathit{install}\ (\mathit{Input}\ \rho) = \mathit{return}\ \rho} \end{array}$$

For FMap and Star we recursively call $\mathit{install}$ on the children of the constructors and combine the results with the associated operators. For a pure value we create a new variable and we just return the embedded variable in the case for Input.

With this *install* function and the modification we made to *xyz* our mutator remains almost the same. The only difference is that we embed the variables in our DSL and that we call *install* to create the model. We avoid having to write our calculation with our IO functions by adding an extra layer of indirection.

```
main :: IO ()
main = do
  x ← newRVar 4
  y ← newRVar 6
  z ← newRVar 8
  out ← install $ xyz (Input x) (Input y) (Input z)
  printRVal out   -- Prints "52"
  setRVal x 10
  printRVal out   -- Prints "58"
```

2.2.3 Reactive Recursive Datatypes

reactive recursive datatypes

In this section we add support for reactive recursive datatypes. We start with how we represent reactive recursive datatypes and how they are stored at run time followed by how we interpret the representation and how we can mutate the reactive recursive datatypes. Weillustrate this by using reactive lists; i.e., lists that are reactive in their tail. This means that expressions can automatically be recomputed when the *structure* of the list changes.

2.2.4 Representation

To represent all reactive recursive datatypes we view recursive datatypes as fix-point and we express this with the newtype[1] RFix. We add the counterpart $\mathit{RVarFix}$ to represent a datatype with run-time variable at the recursive points.

```
newtype RFix    f = RIn    {out    :: R    (f (RFix f))}
newtype RVarFix f = RVarIn {valout :: RVar (f (RVarFix f))}
```

We can now define the type of our reactive list with RFix as follows.

```
data ListF α f    = Nil | Cons α f deriving Show
type RList α      = RFix    (ListF α)
type RListVar α   = RVarFix (ListF α)
```

[1]We would rather have a type synonym; however, the Haskell type-checker doesn't agree with the infinite type.

The ubiquitous function *foldr* is used to abstract away the recursion over Haskell lists. We introduce a similar function for our reactive lists. We will see that we need to make our representation an instance of a monad in order to write this function. The, incomplete, definition of *foldr* is as follows.

```
$foldRList :: (\rho, \alpha \rightarrow \rho \rightarrow \rho) \rightarrow RList\ \alpha \rightarrow R\ \rho$
$foldRList\ \alpha@(nil, cons)\ (RIn\ rxs) =$
  ...
  case $rxs$ of
    $Nil$          $\rightarrow$ ...$Nil$
    $Cons\ y\ ys \rightarrow Cons\ y$ <$> $foldRList\ \alpha\ ys$
```

Note, the type of rxs is $R\ (ListF\ \alpha\ (RFix\ (ListF\ \alpha)))$ so it is not a value of type $ListF\ \alpha$ that we can use for our pattern match: we first need to unwrap the R. This exactly models the concept that the variable needs to be read before we can pattern match on the value. The case depends on the value of our variable and this sounds as an $Monad$; therefore, we apply this idea to our definition and get the following implementation where we use R as a monad. The implementation of R as a monad follows shortly.

```
$foldRList :: (\rho, \alpha \rightarrow \rho \rightarrow \rho) \rightarrow RList\ \alpha \rightarrow R\ \rho$
$foldRList\ \alpha@(nil, cons)\ (RIn\ rxs) =$ do
  $xs \leftarrow rxs$
  case $xs$ of
    $Nil$          $\rightarrow return\ Nil$
    $Cons\ y\ ys \rightarrow Cons\ y$ <$> $foldRList\ \alpha\ ys$
```

We first unwrap the fix-constructor by pattern matching on RIn, then we bind the *value* of the variable, our list, rxs to xs and finally, we pattern match on the list.

The astute reader has noticed that the type of our case-branches is not correct. We return a $ListF\ \alpha\ f$ instead of a $RList\ \alpha$. To ease the construction of values of type $RList\ \alpha$ we introduce the following *smart-constructors*.

```
$nil :: RList\ \alpha$
$nil = RIn\ (return\ Nil)$

$cons :: \alpha \rightarrow RList\ \alpha \rightarrow RList\ \alpha$
$cons\ v = RIn \circ return \circ Cons\ v$
```

And now we come to our final, correct, definition of $foldRList$.

```
$foldRList :: (\rho, \alpha \rightarrow \rho \rightarrow \rho) \rightarrow RList\ \alpha \rightarrow R\ \rho$
$foldRList\ \alpha@(nil, cons)\ (RIn\ rxs) =$ do
```

```
    xs ← rxs
    case xs of
      Nil         → return nil
      Cons y ys → cons y <$> foldRList α ys
```

We now use the *Monad* class in our representation; therefore, we need to make our DSL a Monad. Essentially, we create a *Bind* constructor for the ≫= operator. In our *install* function we map the *Bind* constructor to the function *rValBindIO*. The function *rValBind* behaves the same as ≫= where the continuation of *rValBindIO* runs in *IO*. We need the argument to run in *IO* because we will have to install the model that results from the function argument of the bind.

```
rValBindIO :: RVar α → (α → IO (RVar β)) → IO (RVar β)
```

We extend *R* with the constructors *Bind* and *Return* and add the cases to the interpretor *install*.

```
data R v where
    ...
   Bind   :: R α → (α → R β) → R β
  Return :: α                      → R α
```

It is now easy to define an instance *Monad R*.

```
instance Monad R where
  return = Return
  v ≫= f = Bind v f
```

2.2.5 Interpretation

We first implement the cases for Monads in our interpreter and subsequently we add the interpretation of an *RFix* representation to a *RVarFix* model, analogous to the interpretation of *R* to *RVar*.

Monads We need to add support for Monads to our interpreter in addition to supporting them in our model. This is done easily by adding the following two cases to our interpreter where we use *rValBindIO* from above. For *Bind* we first translate the representation to the model and for the callback of the bind we apply the function *f* to the value and install the resulting representation.

```
install (Bind v f)  = do α ← install v
                         let fx α = install (f α)
```

```
                    rValBindIO α fx
install (Return α) = newRVar α
```

Note, that the Monad instance we provided in the previous section does not satisfy the monad laws. We, realistically, assume that they do hold under *install*; i.e., we assume that the following rules hold.

```
install (return α >>= f)   ≡ install (f α)
install (m >>= return)     ≡ install m
install ((m >>= f) >>= g)  ≡ install (m >>= (λx → f x >>= g))
```

Recursive representations We can use our *quicksort* function as in the code below which would give an undesired result. It would yield us a variable containing a representation of the sorted list instead of the actual run-time value. Although this might be the desired result in some cases, we would like to see the sorted list using run-time variables in this case.

To achieve this we rename the previous *install* function to *installr* and create a class *Reactive* α β that contains the *install* function which interprets a value of type α to a value of type β in *IO*. We obtain the original behaviour by defining the instance *Reactive* $(R\ \alpha)$ $(RVar\ \alpha)$.

```
class Reactive α β where
  install :: α → IO β
instance Reactive (R α) (RVar α) where
  install = installr
```

We then add our case for the reactive lists by defining an instance for *Reactive* $(RList\ \alpha)$ $(RListVar\ \alpha)$ analogous to *Reactive* $(R\ \alpha)$ $(RVar\ \alpha)$. We first unwrap the recursive position. Then we install the first item of the list which yields a variable containing a *ListF* α with an *RFix* at the recursive position and wee need to translate this representation to a run- time variable as well. We do this by mapping the function that applies *install* to the recursive point on the variable.

```
instance Reactive (RList α) (RListVar α) where
  install (RIn ρ) = do
    out ← install ρ
    fmap RVarIn $ flip rValFMapIO out $ λxs → case xs of
      Nil        → return Nil
      Cons x rxs → do
        rvxs ← install rxs
        return $ Cons x rvxs
```

This method creates an additional variable per list item which is not very efficient. Optimising this is left as future work.

2.2.6 Mutation and Usage

We now have enough material to write our *quicksort* function. The definition differs slightly from the definition we showed in the introduction of this chapter as illustrated in Figure 2.1.

The difference with the original function is the extra pattern match on RIn because of the $RFix$ newtype, and the extra lines for the monadic unwrapping. We could prevent the latter by using the function $withFix :: (f\ \ (RFix\ \ f) \rightarrow R\ \beta) \rightarrow RFix\ f \rightarrow R\ \beta$.

```
split p = withFix $ λxs →
  case xs of
    ...
```

We can obtain a sorted list by constructing the representation below. We create a $RList$ with the *fromList* function, we sort it with the *quicksort* function and translate it back to a Haskell list. Printing the result shows the Haskell list $[1, 2, 3, 4, 5]$.

```
main = do
  let m :: R [Int]
      m = do let l = fromList [5, 4 .. 1]
             toList $ quicksort l
  out ← install m
  printRVar out
```

Fibonacci We also have enough to create our Fibonacci function. Again the code is similar to the original code but certainly less concise. We have now shown that we support all three kinds of functions that we introduced in the introduction.

```
fib :: R Int → R Int
fib rn = do
  n ← rn
  case n of
    0 → return 0
    1 → return 1
    n → do l ← fib (return $ n − 2)
           ρ ← fib (return $ n − 1)
           return (l + ρ)
```

```
quicksort   :: Ord α
               ⇒ RList α → RList α
quicksort (RIn rxs) = RIn $ do
  xs ← rxs
  case xs of
    Nil → return $ Nil
    Cons y ys → do
      (l, g) ← split y ys
      let ls = quicksort l
          gs = quicksort g
      out $
        ls         `app`
        singleton y `app`
        gs
split :: Ord α
     ⇒ α → RList α
     → R (RList α, RList α)
split p (RIn rxs) = do
  xs      ← rxs
  case xs of
    Nil       → return (nil, nil)
    Cons y ys → do
      (l, g) ← split p ys
      if y < p then
          return (y `cons` l, g)
        else
          return (l, y `cons` g)
```

(a) Reactive *quicksort*

```
quicksort [] = return []
quicksort (x : xs) = do
  (l, g) ← split x xs
  ls ← quicksort l
  gs ← quicksort g
  return (ls ++ [x] ++ gs)
split p [] = return ([], [])
split p (x : xs) = do
  (l, g) ← split p xs
  if x < p then
      return (x : l, g)
    else
      return (l, x : g)
```

(b) Monadic *quicksort*

Figure 2.1: Comparison of the reactive and monadic versions of *quicksort*.

2.2.7 Returning multiple values

Currently we can only create top-level reactive values such as a reactive Int, or a variable that contains a list, or a reactive list. We can also create a variable that contains a tuple. But if the tuple contains an representation R it is not translated to a variable. To counter this we add the following instance for $Reactive$ where we apply $install$ to both elements of the tuple. The function $doFirst$ applies an IO function to the first element of a tuple and lifts the IO outside of the tuple and the function $doSecond$ does the same for the second element.

```
instance (Reactive α α', Reactive β β')
   ⇒     Reactive (R (α, β)) (RVar (α', β')) where
  install ρ = do
    out  ← (install :: R x → IO (RVar x)) ρ
    out' ← rValFMapIO (doFirst install) out
    rValFMapIO (doSecond installr) out'
doFirst :: (α → IO α') → (α, β) → IO (α', β)
doFirst f inp = do
  let (iores, right) = first f inp
  res ← iores
  return (res, right)
doSecond :: (β → IO β') → (α, β) → IO (α, β')
doSecond f inp = do
  let (left, iores) = second f inp
  res ← iores
  return (left, res)
```

There is a catch to this however, we can now write a program that behaves different from what we expect as we illustrate with the following example.

We create a representation that takes a literal Haskell list, and that yields a tuple containing a reactive list created from the Haskell list and the sorted reactive list. We construct the model by applying $install$ to the representation.

```
main = do
  let m xs = do
    let list = fromList xs
    return (list, quicksort list)
  out ← install (m [5, 4 . . 1])
  ...
```

We define a function $insert$ that takes the position where the new item has to be inserted, the value of the item, and the list in which the item has to be inserted. The definition follows later. We use this function to insert an element in reactive list and expect the sorted list to be updated as well because it depends on the *list*. The function $toListRVar :: RListVar\ \alpha \rightarrow IO\ (R\ [\alpha])$ takes a reactive list and creates a variable containing the corresponding Haskell list. We use this function to make the printing of the variables easier.

```
...
(list, sortedlist)   ← getRVal out
haskellList          ← toListRVar list
sortedHaskellList ← toListRVar sortedlist
printRVar haskellList
printRVar sortedHaskellList
insert 2 10 list
printRVar haskellList
printRVar sortedHaskellList
```

We would expect the output of this function to be as follows: the unsorted list, the sorted list, the unsorted list with the inserted element and the sorted list with the inserted element at the correct position.

```
[5, 4, 3, 2, 1]      { list }
[1, 2, 3, 4, 5]      { sorted list }
[5, 4, 10, 3, 2, 1]{ updated list }
[1, 2, 3, 4, 5, 10]{ updated sorted list }
```

Instead the sorted list is not updated.

```
[5, 4, 3, 2, 1]      { list }
[1, 2, 3, 4, 5]      { sorted list }
[5, 4, 10, 3, 2, 1]{ updated list }
[1, 2, 3, 4, 5]      { not updated sorted list }
```

This is because the representation of the list is copied and the same representation is installed twice: once for the first element of the tuple and once when installing the representation of the sorted list. Therefore, the reactive list in the first element of the tuple is a different list than is used by the *quicksort* function.

2.3 Memoisation

As we discussed in subsection 1.2.1, memoisation is a process that reasons over functions; i.e., something with an input and an output. The current representation models *values* and not *functions*. Remember that the function *memo* takes a function and returns a memoised function.

Ideally we would want to use this concept in our DSL; therefore, we should include it in our embedding. However, we are missing the concept of *input* in our DSL. Leaving the concept of input out of $memo$ would result in the following embedding.

$$\begin{aligned}&\textbf{data}\ R\ v\ \textbf{where}\\&\quad Memo :: R\ v \to R\ v\end{aligned}$$

How could we interpret this constructor in our interpretor? Conceptually, we could traverse the representation in the first argument of $Memo$ to look for $Input$ nodes and then construct a mapping of these inputs to the value. Additionally, we would have to prevent propagation of a change in a value in one of the inputs if we already have seen the new set of values in the inputs.

Actually implementing this idea would be even harder than comprehending all possible implications of this idea; Moreover, we, falsely, assume that we can traverse the model which is not possible because our model is not inspectable due to Haskell function in the second argument of $Bind$.

We could still apply memoisation to increase the performance of our change propagation by memoising the function argument to the bind. This would mean that the dynamically constructed model is reused if the input of the bind was seen before. We leave the implementation for future work.

$$\begin{aligned}&\textbf{data}\ R\ v\ \textbf{where}\\&\qquad \ldots\\&\quad MBind :: Ord\ \alpha\\&\qquad\qquad \Rightarrow R\ \alpha \to (\alpha \to R\ \beta) \to R\ \beta\end{aligned}$$

We forgo the implementation of memoisation in this system because memoisation is easily added to our second system as we will see in chapter 3.

2.4 Summary

In this chapter we have shown how to create a reactive model manually using the reactive library. We have shown how to abstract over the functions in the reactive library to avoid having to program in IO. Additionally we showed that while it easy to obtain change propagation it is difficult to add memoisation. Also, as seen in subsection 2.2.7, the programmer may encounter unexpected behaviour.

We have seen that this implementation supports all three kinds of functions from the introduction.

A minor issue is that the model is not inspectable because it can only be determined by interpreting it at run-time. This prevents the printing and optimisation of the representation which would have been nice to have.

Chapter 3

Memoisation first

3.1 Overview

In this chapter we model our DSL after *calculations*, in contrast to chapter 2 where we used *values* as foundation for the DSL. Subsequently, we add memoisation and change propagation.

We use the concept of a calculation, a process with an input and an output, and therefore the DSL in this chapter is modelled after the *Category* and *Arrow* classes instead of the *Functor*, *Applicative*, and *Monad* classes. The *Category* class represents a calculation and provides functions for the identity calculation and the composition of two calculations. The *Arrow* class adds the lifting of Haskell functions and operators for controlling data-flow.

Remember that we want to preserve the conciseness of the program when the programmer uses our library; therefore, we use the Arrow- syntax (Paterson, 2001; Hughes, 2000; GHC, 2013) in the remainder of this chapter. We refer the reader to *Arrows: A General Interface to Computation* (Paterson, 2010) for a concise explanation of the arrow operators and arrow syntax.

We will also see that we can inspect the DSL given in this chapter which enables the printing and the optimisation of the DSL.

In the remainder of this chapter we first introduce a new DSL based on the *Arrow*-class (section 3.2) and we give an recursive interpretation function that interprets the DSL to an Haskell function in section 3.3. We show that recursive models, such as *fib*, give rise to issues and we provide a solution in section 3.4. Subsequently, we create a new interpretor function that creates a reactive model instead of an Haskell function in section 3.5. After this we proceed to add support for memoisation to the DSL and the interpretor (section 3.6) and show that with this implementation we have reached our goal (section 1.3), and we revisit change propagation and the issues with our current solution in section 3.7. Next we show that in order to calculate the MSS we need higher-order models and add support for this in section 3.8. We conclude with possible optimisation algorithms (section 3.9) and a summary of this chapter in section 3.10.

3.2 Representation: $\cdot \rightsquigarrow \cdot$

We, again, create a deep-embedding but now for the *Category* and *Arrow* classes. The definition of datatype follows directly from the definitions of the classes.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \alpha \rightsquigarrow \beta\ \textbf{where}} \\
\quad \mathit{Id} & :: \alpha \rightsquigarrow \alpha & \\
\quad \mathit{Comp} & :: \beta \rightsquigarrow \gamma \to \alpha \rightsquigarrow \beta \to \alpha \rightsquigarrow \gamma & \\
\quad \mathit{Arr} & :: (\alpha \to \beta) \to \alpha \rightsquigarrow \beta & \\
\quad \mathit{Split} & :: \beta \rightsquigarrow \gamma \to \beta' \rightsquigarrow \gamma' & \to (\beta,\beta') \rightsquigarrow (\gamma,\gamma') \\
\quad \mathit{First} & :: \beta \rightsquigarrow \gamma & \to (\beta,\delta) \rightsquigarrow (\gamma,\delta) \\
\quad \mathit{Second} & :: \beta \rightsquigarrow \gamma & \to (\delta,\beta) \rightsquigarrow (\delta,\gamma) \\
\quad \mathit{Choice} & :: \beta \rightsquigarrow \gamma \to \beta' \rightsquigarrow \gamma' & \to (\mathit{Either}\ \beta\ \beta') \rightsquigarrow (\mathit{Either}\ \gamma\ \gamma')
\end{array}
$$

The instances *Category* $\cdot \rightsquigarrow \cdot$, *Arrow* $\cdot \rightsquigarrow \cdot$, and *ArrowChoice* $\cdot \rightsquigarrow \cdot$ are trivial to implement. These instances allow us to use the arrow-syntax; therefore, we can write the functions xyz and fib as follows.

$$
\begin{array}{ll}
\mathit{xyz}\ x\ y\ z = x + y * z & \mathit{xyx} = \mathit{proc}\ (x,y,z) \to \textbf{do} \\
& \quad \mathit{yz} \leftarrow \mathit{returnA} \prec y * z \\
& \quad \mathit{returnA} \prec x + \mathit{yz}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{fib}\ 0 = 0 & \mathit{fib} = \mathit{proc}\ n \to \textbf{case}\ n\ \textbf{of} \\
\mathit{fib}\ 1 = 1 & \quad 0 \to \mathit{returnA} \prec 0 \\
\mathit{fib}\ n = \mathit{fib}\ (n-2) + \mathit{fib}\ (n-1) & \quad 1 \to \mathit{returnA} \prec 1 \\
& \quad n \to \textbf{do}\ l \leftarrow \mathit{fib} \prec (n-2) \\
& \qquad\qquad \rho \leftarrow \mathit{fib} \prec (n-1) \\
& \qquad\qquad \mathit{returnA} \prec (l + \rho)
\end{array}
$$

This **do**-notation for arrows is desugared to a combination of the operators from the *Arrow* class. We desugar the function fib' to show what happens. A **case**-expression is translated to two parts. The first parts is a lifted Haskell function that performs the pattern matching and encodes the chosen path with *Either*-constructors. The second part uses the choice-operator, $(\ |||\ )$, to traverse the given path and uses the left- hand arrow when it encounters a *Left* and the right-hand otherwise.

```
fib :: Integer ⇝ Integer
fib
  = (arr
      (λn →
         case n of
           0 → Left ()
           1 → Right (Left ())
           n′ → Right (Right n′))
      ⋙
      (arr (λ() → 0) |||
         (arr (λ() → 1) |||
            (arr (λn′ → (n′, n′)) ⋙
               (first (arr (λn′ → (n′ − 2)) ⋙ fib) ⋙
                  arr (λ(l, n′) → (n′, l)))
                  ⋙
                  (first (arr (λn′ → (n′ − 1)) ⋙ fib) ⋙
                     arr (λ(ρ, l) → (l + ρ)))))))
```

Further on we will see that we need to add constrains to the types in the constructors of our datatype. This means that we cannot write programs in our DSL by using the arrow operators any more; therefore, we introduce our own functions with the same name as the functions in *Category* and *Arrow* in combination with the `RebindableSyntax` extension in order to keep the arrow-syntax.[1]

3.3 Interpretation, recursively

As we did before, we can interpret our DSL by applying our interpretor at each recursive point and combine the results. Interpreting to a Haskell function is achieved by simply replacing the constructors with their associated operator from the *Arrow* class as → is an arrow.

```
runR :: α ⇝ β → α → β
runR (Id _)           = C.id
runR (Comp _ g f) = runR g ∘ runR f
runR (Arr _ f)        = f
runR (Split _ f g)   = runR f *** runR g
runR (First _ f)      = first (runR f)
runR (Second _ f)   = second (runR f)
runR (Choice _ l ρ) = runR l +++ runR ρ
```

[1] Actually, we are forced to the translation manually or with the `arrowp` package as the combination of arrow-syntax and `RebindableSyntax` is broken.

With this interpretor we can map the xyz and fib representations to Haskell functions. Note, however, that our definition of fib is recursive and thus our representation of fib is an infinite datastructure. Because of lazy evaluation we can obtain the Haskell function representation of fib. However, we cannot print or optimise our representation in finite time because printing and optimisation inspect the whole structure.

In the next section we show how to transform this tree-representation into a graph-representation of our representation of the reactive model.

3.4 Graph representation of $\cdot \rightsquigarrow \cdot$

In this section we show why we want to store the representation of the reactive model differently and introduce the new way to store the representation.

There are two reasons why we want to move away from the infinite representations.

1. If we want to add change propagation we have to lift our functions to IO thereby losing lazy evaluation and thus we are unable to interpret finite representations such as fib. We loose lazy evaluation as IO is a monad and monads impose an order in the evaluation of the value.

2. It is impossible to print or optimise infinite datastructures which is something we want.

We, therefore, make the recursion explicit by storing our representation as a graph instead of a tree. The graph is stored as a map of node identifiers to node constructors and each recursive point is replaced with a pointer to a node. Each node is represented by the datatype R'.

```
data R' α β where
  Id'     :: ...
          ⇒ R' α α
  Comp'   :: ...
          ⇒ R' β γ → R' α β → R' α γ
  Arr'    :: ...
          ⇒ Int → (α → β) → R' α β
  Split'  :: ...
          ⇒ R' β γ → R' β' γ' → R' (β, β') (γ, γ')
  First'  :: ...
          ⇒ R' β γ → R' (β, δ) (γ, δ)
  Second' :: ...
          ⇒ R' β γ → R' (δ, β) (δ, γ)
  Choice' :: ...
```

```
          ⇒ $R' \; \beta \; \gamma \to R' \; \beta' \; \gamma' \to R' \; (Either \; \beta \; \beta') \; (Either \; \gamma \; \gamma')$
$Ptr$     :: ...
          ⇒ $Int \to R' \; \alpha \; \beta$
```

We still aim to use the arrow-syntax but the types of the arrow combinators do not allow to create the desired structure directly. Finding a cyclic representation of a through recursion created infinite datastructure is equivalent to finding sharing in a tree structure; therefore we want to detect sharing in our datastructure of type $\cdot \rightsquigarrow \cdot$.

One way to find sharing in a tree-structure is readily available as a Hackage package (Gill, 2009; Gill, 2011) and we will therefore use this package. The package detects the sharing that GHC introduces by inspecting the memory address of values as opposed to creating the sharing itself.

By reifying our tree-structure we essentially find the binding of variables which is generally hard to do directly with deep-embedding. Another, type- safe, solution for observing bindings is the work presented by Baars et al. (Baars et al., 2009). We refrain to use their method here as it is not compatible with the Arrow operators. Ultimately we would like to be able to reify any Haskell function as that would enable us to perform the transformation we now perform on our DSL on any Haskell function.

Our model R' contains representations of functions with different types and therefore the constructor of R' have type parameters. We store our representation as a graph in a map only stores values of the same type. We introduce $SomeR$ as the fully existentially qualified, or Anonymised datatype, type of R'. Because we need to lookup values and retrieve the type parameters for use later on we need $Typeable$ constraints on the type parameters so we change our R' to read as follows.

Anonymised datatype

```
data $SomeR$ where
  $Wrap :: (Typeable \; \alpha, Typeable \; \beta) \Rightarrow R' \; \alpha \; \beta \to SomeR$
data $R' \; \alpha \; \beta$ where
  $Id'$     :: $(Typeable \; \alpha)$
            ⇒ $R' \; \alpha \; \alpha$
  $Comp'$   :: $(Typeable \; \alpha, Typeable \; \beta, Typeable \; \gamma)$
            ⇒ $R' \; \beta \; \gamma \to R' \; \alpha \; \beta \to R' \; \alpha \; \gamma$
  $Arr'$    :: $(Typeable \; \alpha, Typeable \; \beta)$
            ⇒ $(\alpha \to \beta) \to R' \; \alpha \; \beta$
  $Split'$  :: $(Typeable \; \beta, Typeable \; \beta', Typeable \; \gamma, Typeable \; \gamma')$
            ⇒ $R' \; \beta \; \gamma \to R' \; \beta' \; \gamma' \to R' \; (\beta, \beta') \; (\gamma, \gamma')$
  $First'$  :: $(Typeable \; \beta, Typeable \; \gamma, Typeable \; \delta)$
            ⇒ $R' \; \beta \; \gamma \to R' \; (\beta, \delta) \; (\gamma, \delta)$
  $Second'$ :: $(Typeable \; \beta, Typeable \; \gamma, Typeable \; \delta)$
```

$$
\begin{array}{ll}
 & \Rightarrow R'\ \beta\ \gamma \to R'\ (\delta, \beta)\ (\delta, \gamma) \\
\mathit{Choice}' & :: (\mathit{Typeable}\ \beta, \mathit{Typeable}\ \beta', \mathit{Typeable}\ \gamma, \mathit{Typeable}\ \gamma') \\
 & \Rightarrow R'\ \beta\ \gamma \to R'\ \beta'\ \gamma' \to R'\ (\mathit{Either}\ \beta\ \beta')\ (\mathit{Either}\ \gamma\ \gamma') \\
\mathit{Ptr} & :: (\mathit{Typeable}\ \alpha, \mathit{Typeable}\ \beta) \\
 & \Rightarrow \mathit{Int} \to R'\ \alpha\ \beta
\end{array}
$$

We create a type Model that contains the start of the graph, conceptually the main function in our model, and the representation as a graph.

$$\textbf{newtype}\ \mathit{Model} = \mathit{Graph}\ (\mathit{Int}, \mathit{Map}\ \mathit{Int}\ \mathit{SomeR})$$

Because we have the $\mathit{Typeable}$ constraints in R' and we create R' from $\cdot \rightsquigarrow \cdot$ we need to add the $\mathit{Typeable}$ constraints on the constructors of $\cdot \rightsquigarrow \cdot$ and to our smart-constructors as well. This is why we defined our own functions and use the language extension `RebindableSyntax`. This extension lets GHC use user defined functions with the same name as the functions from Arrow class.

We can obtain our graph out of a representation by using the function $\mathit{reify} :: \alpha \rightsquigarrow \beta \to \mathit{IO}\ \mathit{Model}$.[2]

It is interesting to note that we mentioned in our introduction that the functions in a reactive model can be represented by edges and that here we have nodes for our functions.

In the next paragraph we show how we can define an interpretor for this graph representation.

3.5 Interpretation, again

We now show how we can interpret the obtained graph representation. We first show how to translate the graph to a Haskell function, similar to runR, to get an idea how the mechanism works. Subsequently, we change the interpreter to create a Reactive model.

Reactive model

3.5.1 Interpretating to an Haskell function

We now show how to define an interpretor, $\mathit{installFunction}$, that creates an Haskell function from the graph representation. The only reason why this function is in IO is because the reification from $\cdot \rightsquigarrow \cdot$ to R' happens in IO.

$$\mathit{installFunction} :: (\mathit{Typeable}\ \alpha, \mathit{Typeable}\ \beta) \Rightarrow \alpha \rightsquigarrow \beta \to \mathit{IO}\ (\alpha \to \beta)$$

[2]The actual function reify from the package has a more general type and returns a list instead of a map.

Globally this function works as follows:

1. We translate the tree to a graph;
2. Subsequently we map a *translate* function over the nodes to translate each node to a Haskell function, the result is a map of identifiers to Haskell functions;
3. We look up the root of the graph and yield that function as a result.

We can translate the representation into an Haskell function in a finite amount of time as because the graph is finite and we translate each node only once. The implementation of the algorithm above reads as follows.

```
installFunction :: (Typeable α, Typeable β)
                ⇒ α ⇝ β → IO (α → β)
installFunction e = do
  Graph root m ← reify e
  let functions = Data.Map.mapWithKey (translate functions) m
  fromError "letf" $ lookupF functions root
```

The *translate* function is similar to *runR* but instead of a recursive call to *runR* it performs a lookup in the map of Haskell functions. The function *lookupF* performs a lookup in the map *functions* and casts the result to the correct function type. We need an anonymized type for storing our functions with a different argument and result types in a map.

```
data SomeFunction where
  SomeFunction :: (Typeable α, Typeable β)
               ⇒ (α → β) → SomeFunction
type Error α = Either String α
lookupF :: (Typeable α, Typeable β)
        ⇒ Map Int SomeFunction → Int → Error (α → β)
lookupF m n = case m ! n of
  SomeFunction f → let castf = cast f in
    case castf of
      Just x → Right x
      Nothing → ...
```

The implementation of *translate* follows naturally and follows the same pattern as *runR*. The, partial, implementation reads as follows.

```
translate :: Map Int SomeFunction → SomeR Unique
          → SomeFunction
translate functions (Wrap node) = case node of
  Id'     → SomeFunction Prelude.id
```

```
    Arr' f → SomeFunction f
    Comp' (Var g :: R' y z Unique) (Var f :: R' x y Unique) →
      SomeFunction ∘ fromError $ do
        g' ← lookupF functions g
        f' ← lookupF functions f
        return $ (g' :: y → z) ∘ (f' :: x → y)
    ...
```

3.5.2 Interpreting to a reactive model

We now construct a reactive model instead of an Haskell function by using our reactive library. We use the same algorithm as before but now with variables. This function is called *installCP* and the body is almost identical to that of *installFunction*.

```
installCP :: (Typeable α, Typeable β) ⇒ α ⇝ β → IO (RVar α, RVar β)
installCP e = do
  Graph n m ← reify e
  let mods = Data.Map.mapWithKey (translate tables mods) m
  let f = fromError "letf" $ lookupMod mods n :: ModFunction x y
  inp ← newRVar (error "Uninitialised")
  out ← f inp
  return (inp, out)
```

We do almost the same as in *installFunction*: we transform the tree structure to a graph and we lookup the root of the model. But instead of returning the function we apply the function to a new input variable and we yield the resulting output variable together with the input variable in a pair. The function *lookupMod* is analogous to the function *lookupF*.

The function *translate* now produces an anonimised *ModFunction* and this is reflected in the types as follows.

```
type ModFunction α β = RVar α → IO (RVar β)
data SomeModFunction where
  WrapModFunction :: (Typeable α, Typeable β)
                           ⇒ ModFunction α β → SomeModFunction
translate :: Map Unique SomeModFunction → SomeR Unique
          → SomeModFunction
```

The implementation of *translate* for *installCP* follows the same pattern as the function *install* from subsection 2.2.2, just as the *translate* function from *installFunction* follows the *runR* function.

```
translate mods (Wrap node) = case node of
  Id'      → WrapModFunction $ λinp → return (inp :: RVar α)
  Arr' _ f → WrapModFunction $ rValFMap f
  Comp' (Var g :: R' β γ Unique) (Var f :: R' α β Unique) →
    let (gF, fF) = fromError $ do
          g :: ModFunction β γ ← lookupMod mods g
          f :: ModFunction α β ← lookupMod mods f
          return (g, f)
    in WrapModFunction $ fF >=> gF
```

The implementation of the *Choice'* case is interesting as we have to preserve the semantics of a case, namely that only the chosen branch is executed in contrast to executing each branch and choosing which result to take later. As **case**-statement is translated into a nested set of *Choice'* constructors this means that we have to install either the left or the right branch.

The only way we can do this is by delaying the construction of the sub reactive model until the input of the *Choice'* constructor is set. At that point we know which sub graph to pick. We then have to link the variables in such a way that if the value of the *Choice'* constructor changes the input of the sub-graph changes as well. We can perform the choice of the right sub-graph and its construction at every change in input but we can optimise this as well by keeping the previous sub-graph intact and only change it when the *direction* of the input has changed as well; i.e., it has become a *Left* instead of a *Right x* or vice-versa.

```
type GraphPair α β = (RVar α, RVar β)
type ChoiceStore β β' γ γ' = IORef
  (Maybe (Either (GraphPair β β') (GraphPair γ γ')))
translate mods (Wrap node) = case node of
  ...
  (Choice' (Var l :: R' β β' Unique) (Var ρ :: R' γ γ' Unique)) =
    let (lF, rF) = fromError $ do
          l :: ModFunction β β' ← lookupMod mods l
          ρ :: ModFunction γ γ' ← lookupMod mods ρ
          return (l, ρ)
        -- switchSubGraph would also be a good name
      getSubGraphF :: ChoiceStore β β' γ γ' → RVar (Either β' γ')
                     → Either () ()
                     → IO (Either (GraphPair β β')
                                  (GraphPair γ γ'))
      getSubGraphF store out side = do
        stored ← readIORef store
        case (stored, side) of
          (Just (Left s),   Left    _) → return (Left s)
      (Just (Right s), Right _)   → return (Right s)
      (x,              Left  _)  → do
        case x of
          (Just (Right s)) → unlink out
          _                → return ()
        inL ← newRVar $ error "Choice.inL"
        outL ← lF inL
        linkRVal out outL $ λx _ → void $ setRVal out (Left x)
        let res = Left (inL, outL)
        writeIORef store (Just res)
        return res
          -- Create new pair and store in store
      (x,              Right _) → do
        case x of
          (Just (Left (inL, outL))) → unlink out
          _                         → return ()
        inR ← newRVar $ error "Choice.inR"
        outR ← rF inR
        linkRVal out outR $ λx _ → void $ setRVal out (Right x)
        let res = Right (inR, outR)
        writeIORef store (Just res)
        return res
in WrapModFunction $ λinp → do
  store ← newIORef    Nothing                  :: IO (ChoiceStore β β′ γ γ′)
  out    ← newRVar $ error "Choice.out" :: IO (RVar (Either β′ γ′))
  let getSubGraph = getSubGraphF store out
  let setIn α _ = debugLn "in choice" ≫ case α of
    Left                x → do
                debugLn "Choice.Left"
                Left (inL, outL) ← getSubGraph (Left ())
                setRVal inL x
                return ()
    Right x → do
                debugLn "Choice.Right"
                Right (inR, outR) ← getSubGraph (Right ())
                setRVal inR x
                return ()
addEffect inp setIn
return out
```

The implementations of the cases for the *First′* and *Second′* constructors are less natural and straightforward than the implementations in the *runR* function.

We only show the implementation of the case for $First'$ as the case for $Second'$ is analogous to the one of $First'$. The implementation reads as follows [3]:

```
(First' (Var l :: R' β γ Unique) :: R' (β, δ) (γ, δ) Unique) →
  let (lF) = fromError $ do
    l :: ModFunction β γ ← lookupMod mods l
    return l
    in WrapModFunction $ λ(inp :: RVar (β, δ)) → do
      { Project }
      secondv ← fmap snd $ getRVal inp
      proj ← rValFMap fst inp
      { Get individual result }
      outProj ← lF proj
      { Combine }
      out ← newRVar (error $ "Uninitialized First :: "
                          ++ show (typeOf (⊥ :: γ))
                    , secondv)
      linkRVal out inp      (λx _ → void $
        modifyRVar out (Arrow.second (const $ snd x)))
      linkRVal out outProj (λx _ → void $
        modifyRVar out (Arrow.first (const x)))
      return out
```

We create an output variable containing the second value of the input pair and an $\perp$ first element. We can use $\perp$ here because no $getRVar$ occurs before a $setRVar$ has occured.

3.6 Adding memoisation

Now we go ahead and add support for memoisation to obtain a fast *fib* function. We first explain how to achieve this with hard-coded HashTables and subsequently we abstract the creation of the memoisation table, and the lookup and insert functions out.

The idea of the memoisation constructor is that at the point of memoisation we store the calculated values together with the input in the memo table, just as *memo* does for normal Haskell functions. For every call a unique variable is generated and the value it obtained from the memo-table if possible, otherwise it is computed.

[3]The actual implementation has the case statement split over two case statements due to a bug in ghc.

First we add an additional $MemoHash$ constructor to $\cdot \rightsquigarrow \cdot$ and its corresponding constructor to R'.

$$
\begin{aligned}
&\textbf{data}\ \alpha \rightsquigarrow \beta\ \textbf{where}\\
&\qquad\qquad \ldots\\
&\quad MemoHash :: (Typeable\ \alpha, Typeable\ \beta, Hashable\ \alpha, Eq\ \alpha)\\
&\qquad\qquad\qquad\quad \rightarrow \alpha \rightsquigarrow \beta \rightarrow \alpha \rightsquigarrow \beta
\end{aligned}
$$

Now we have to change both the *install* function and the *translate* function. As an additional step in the *install* function we will create a memo-table for each $MemoHash$ constructor in our graph and we will then pass these generated tables along to the *translate* function. This way the creation of the memo table is taken out of the IO action of the *ModFunction* and therefore does not happen whenever the mod function is called. Creating the table inside the translation step would make *translate* run in IO which would make the translation infinite again for recursive functions.

Recall that the $MemoHash'$ constructor points to the model that has to be memoised and as that node is unique we can re-use the pointer to the model as the key for the memo-table. The function $isSomeMemoHash'$ checks whether the value is an anonymized $MemoHash'$ constructor. We filter out the $MemoHash'$ constructors and create a new table for each one by mapping the function *createSomeTable* over the list of $MemoHash'$ constructors.

$$
\begin{aligned}
&installCP =\\
&\quad \ldots\\
&\quad \textbf{let}\ memoConstructors = filter\ isSomeMemoHash'\ (elems\ m)\\
&\quad tables \leftarrow fmap\ fromList\ \$\ mapM\ createSomeTable\ memoConstructors\\
&\quad \textbf{let}\ mods = Data.Map.mapWithKey\ (translate\ tables\ mods)\ m\\
&\quad \ldots
\end{aligned}
$$

We add the tables argument to the *translate* function and we define a case for the $MemoHash'$ constructor. In the function *translate* we perform a lookup in the memo table with the input value as key each time the input of the $MemoHash'$ constructor is set. If the value was in the table the output variable of the $MemoHash'$ constructor is set to the result value, otherwise, the value of the input is propagated to the input variable of the inner reactive model. When the output value of the inner model is set, we store the current values of the input variable and the inner output variable in the memo-table and we set the output variable of the constructor to the result value.

$$
\begin{aligned}
&MemoHash'\ (Var\ m :: R'\ \alpha\ \beta\ Unique) \rightarrow\\
&\quad \textbf{let}\ f :: ModFunction\ \alpha\ \beta = fromError\ \$\ lookupMod\ mods\ m\\
&\qquad table :: Reactive.Implementation.HashTable\ \alpha\ \beta =
\end{aligned}
$$

```
      fromError "memohash.table:" $ getMemoTable tables m
  in WrapModFunction $ λinp → do
    out ← newRVar $ error "out"
    innerInp ← newRVar $ error "innerIn"
    innerOut ← f innerInp
    linkRVal innerInp inp $ λx _ → do
      res ← H.lookup table x
      case res of
        Just x' → void $ setRVal out x'
        Nothing → void $ setRVal innerInp x
    linkRVal out innerInp (λ_ _ → return ())
    linkRVal out innerOut $ λv _ → void $ do
      k ← getRVal inp
      setRVal out v
      H.insert table k v[4]
    return out
```

The astute reader has noticed that the output is linked by a void update function. We could have chosen to perform the lookup twice and to move the setting of the output variable to the currently void update function. However, for reasons of performance and avoiding code duplication we have merged the two functions.

As we will see in the benchmarks this gives us a performance increase when we use the $MemoHash$ constructor in the *fib* function.

3.6.1 Sharing of the memo-tables

As we pointed out earlier, there is exactly one memo table for each $MemoHash'$ constructor. If we translate the function $\lambda n \to fib\ n + fib\ n$ to arrow form we get the following code:

```
dup ⋙ fib &&& fib &&& arr (+)
```

If we translate this to the graph we see that the two arguments of &&& point to the same constructor, the top constructor of *fib* in our case the $MemoHash'$ constructor. This, combined with the implementation of the $MemoHash'$ constructor above, gives us exactly the behaviour as we set out to achieve in section 1.3.

[4]Actually due to a bug that I haven't solved yet this line is wrapped in an exception handler. For some reason the setter function is called while the k and or v are still ⊥ which should not happen.

3.6.2 Abstraction of the memoisation

Remember that our goal is to provide the programmer with the ability to control how memoisation is performed. To that end we factor out the functions for creating the memo-table, and inserting and looking up in the memo-table to the *MemoTable* constructor. This allows, among other things, the programmer to control the size of the table and which values are stored.

We add the following constructor to our representation. The implementation of the interpretor is identical to the case for *MemoHash′* with the functions specialized for *HashTables insert* and *lookup* replaced with the generic versions stored in the constructor.

$$
\begin{array}{l}
\textbf{data}\ \alpha \rightsquigarrow \beta\ \textbf{where} \\
\qquad \ldots \\
\quad \mathit{MemoTable} :: (\mathit{Typeable}\ \alpha, \mathit{Typeable}\ \beta, \mathit{Typeable2}\ t) \\
\qquad\qquad\qquad \Rightarrow \mathit{IO}\ (t\ \alpha\ \beta) \to (t\ \alpha\ \beta \to \alpha \to \mathit{IO}\ (\mathit{Maybe}\ \beta)) \\
\qquad\qquad\qquad \to (t\ \alpha\ \beta \to \alpha \to \beta \to \mathit{IO}\ ()) \\
\qquad\qquad\qquad \to \alpha \rightsquigarrow \beta \to \alpha \rightsquigarrow \beta
\end{array}
$$

Commutativity The ability to define custom functions for the *insert* and *lookup* operators gives the programmer the ability to take the commutativity of an operator into account. When an operator is commutative the order of the arguments does not matter for the result of the operator. The programmer can use this attribute to either make the memo- table more compact by only storing the given combination and at lookup try each permutation of the values, or save on the lookup time and add the same value for each permutation at the cost of increased memory usage.

3.7 Issues with the change propagation implementation

The astute reader has noticed from the above that the way change propagation is implemented in this method does not work as we claimed it would in subsection 1.2.2. In fact, it does not prevent unnecessary recomputation at all and in its current form it is just a convenience to the programmer as the output of the model is updated automatically when its input changes. In this section we describe why the change propagation does not work as desired and how we can work around it.

In the previous chapter we knew for certain that when a variable was changed that it was the only variable that had changed. Therefore we knew that we could update its dependencies and leave other computations untouched. However, in this implementation each variable contains all the input of the calculation because all arguments to an are tupled into a single argument. This means that if one of the arguments changes the value of the input variable, the tuple, changes but we have lost the information which argument, which element of the tuple, actually changed.

Adding an equality constraint or function would go some way to solve this as the changes would stop propagation when the arguments are passed further along through the model to where they are the only arguments at which point it can be decided whether it actually did change. However, in addition to that this would only work if the model has methods with just one argument this also imposes a significant performance hit as we have to do work at every point to check whether the arguments have changed opposed to, as opposed to intrinsically knowing that we could update a variable because it was the only one that had changed. Therefore, where we could add the equality check to *increase* performance in the previous implementation, it is required to prevent any recomputation in this implementation.

We could limit the equality checks to the *first* and *second* operators thus limiting the amounts of equality checks performed. However, we recognise at least two problems with this. The first is that performing equality checks is a serious performance hit when working with recursive datastructures, such as lists, and the second is that requiring equality limits the flexibility of the programmer to choose an other criteria on whether to propagate or not. Therefore we choose for a solution similar to the predicates used by Acar (U. Acar et al., 2006).

Following this idea we introduce another construct to our DSL, namely, the *Cache* constructor. It contains a predicate function that is used to determine whether the change in the input of the *Cache* constructor should propagate to its *output*-variable.

Please note that it is not possible to add *Cache* ($\equiv$) nodes at a point after compilation, e.g. a runtime optimisation step, due to parametricity. We cannot determine at runtime whether a certain value as an associated *Eq* instance.

We implement the case for the *Cache′* constructor as follows. We create a new output variable and we link it to the input variable with a link function that only sets the output variable with the input variable if the predicate p holds.

```
(Cache′ (p :: α → α → Bool)) → WrapModFunction $ λinp → do
  out ← newRVar (error "Uninitialized Cache")
  let setter n = void $ setRVal out n
  let set n o = handle (λ(_ :: ErrorCall) → setter n) $
    unless (p n o) (setter n)
  linkRVal out inp set
```

```
    return out
```

Note that the implementation of the case for the $Cache'$ constructor is a bit hairy as we have a situation where we can't use the *previous* argument as it is $\bot$. I have made an attempt at wrapping the value of an $RVar$ in a $Maybe$ however the extra pattern matching made things to strict in the underlying reactive library. Therefore we choose to do some exception-handling at this point at the cost of readability and efficiency.

We now have support for almost all our desired functions. We will now show how to calculate the maximum segment sum of a list.

3.8 Maximum Segment Sum

In this section we introduce the model of the calculation of the MSS. In order to define this model we need to introduce a new concept, namely, that of higher-order models. We first describe the calculation of MSS and show why we need higher-order models and then proceed with how to add support for this in our interpretor.

3.8.1 Background

```
mss :: [Int] ⇝ Int
mss = tails ⋙ map inits ⋙ concat ⋙ map sum ⋙ maximum
```

Each of the functions in the code above are the arrow implementation of their Prelude equivalents. We omit the definition of the functions *tail*, *concat* and *maximum* and focus on the functions *map* and *inits*. We have two different versions of the *map* function. One that accepts a model as an Haskell level argument and the other that models that accepts the function as an argument on the model level. The definition of the first one, *map*, is straightforward and follows the definition of its Prelude equivalent.

```
map :: α ⇝ β → [α] ⇝ [β]
map f = proc xs →
  case xs of
    []     → returnA −≺ []
    x : xs → do mxs ← map f −≺ xs
                fx ← f −≺ x
                returnA −≺ (fx : mxs)
```

The *inits* function in normal Haskell code reads as follows.

```
inits    :: [α] → [[α]]
inits xs = [] : case xs of
                  []      → []
                  x : xs′ → map (x : ) (inits xs′)
```

When translated to arrow-syntax the function reads as follows.

```
inits :: [α] ⇝ [[α]]
inits = proc xs →
  do t ← case xs of
       []      → returnA −< []
       x : xs′ → do ins ← inits −< xs′
                    (map (arr (x : ))) −≪ ins
     returnA −< ([] : t)
```

Pay particular attention to the line $(map\ (arr\ (x:)))$−≪ ins. We have introduced a new operator, namely, −≪. This operator stands for the *app* function from the *ArrowApply*-class. The *app* function takes an arrow and an input value for the arrow and applies the arrow to the input.

```
class ArrowApply ρ where
  app :: ρ (ρ α β, α) β
```

We need this functionality because during the evaluation of the *inits* function we create a new representation, namely, $map\ (arr\ (x:))$. We then apply the obtained representation to the inits of the remainder of the list with the −≪ operator. In the interpretation of this representation we create the reactive model and apply it to the inits.

However, the current implementation is not semantically clean: we do not mean to say that we create a new *map* function that we will apply to a list but we mean to say that there is a part of the model, namely, the $arr\ (x:)$ that can be only constructed during runtime as it is only then that the x is know. Therefore we create a variant of *map*, namely, *mapR*.

```
mapR :: (α ⇝ β, [α]) ⇝ [β]
mapR = proc (f, xs) →
  case xs of
    []     → returnA −< []
    x : xs → do mxs ← mapR −< (f, xs)
                fx ← f −≪ x
                returnA −< (fx : mxs)
```

This *mapR* function performs the application of f to x and this means that we can change the line from *inits* we discussed above to read as follows.

```
...
  mapR −< (arr (x :), ins)
```

3.8.2 Implementation

We now have all the models we need to define *mss* but we still need to support the concept of *app* into our interpretor. Therefore, we add the constructor *Apply* to our model.

```
data α ⇝ β where
  ...
  Apply :: (α ⇝ β, α) ⇝ β
```

The implementation of the case for this constructor is where we are going to have to make interesting choices. We have to answer the following questions:

1. **Sharing memo-tables**

   When applying a model, the argument model is the inner model and the model where the application happens is the context. Do we aim for the sharing of memo-tables between the inner model and the context? That means, if we have a memo-table for *fib* and we use fib in the inner model and in the context, should their memo-tables be shared?

2. **Prevent double work** As the goal of this thesis is to prevent the repeated recomputation of work it would be odd if we would repeat the whole process of translating the model to a graph and translating the graph to a reactive model each time we call apply. Do we want to prevent this? If so, how do we prevent this?

The implementation is straightforward if the answers to these questions is “no". We show this implementation. Furthermore we discuss the changes necessary to support the sharing of the memo-tables and to support the recomputation of the model.

The naive implementation of the case for *Apply′* is extremely straightforward. We create a new output variable and we link it to the input as follows. If the input changes we take the provided representation of the reactive model, install it by calling *installCP*, setting the input variable of the installed model to the provided value and read the resulting value and storing it in the output variable.

```
x@Apply′ → WrapModFunction $ λinp → do
  out ← newRVar $ error "Apply.out"
  linkRVal out inp $ λ(ρ, x) _ → void $ do
    (innerInp, innerOut) ← installCP ρ
    setRVal innerInp x
    rx ← getRVal innerOut
    setRVal out rx
  return out
```

Sharing memo-tables In order to be able to share the memo-tables between the context and the inner model we need to be able to uniquely identify the $Memo'$ constructors. In the implementation of memoisation in section 3.6 we used the value of the pointer to the memoised expression as the key for our map of memo-tables. However, we cannot continue to do this if we want to share the memo-tables because the identities of two equal nodes in two separate representations are not necessarily equal as well. Therefore we could an additional field to the $Memo$ constructor for the id of the constructor and use this field as key as it will be the same in both the inner representation and the representation of the context. The question is then how to come up with this id. We can either do this manually, which is *error-prone* and thus undesirable, or by using a global counter. Using a global counter would mean using $unsafePerformIO$ as arrows don't support IO.

Prevent double work In order to prevent to repeated installation of the same reactive model we need a mechanism to detect whether the model has changed. However, a simple equality check on a representation of type $\cdot \rightsquigarrow \cdot$ is a potentially infinite computation. We see at least two options around this, firstly to perform an equality check on the translated representation R', and, secondly, to provide a sound shallow equality. The first option is easy to implement but has the disadvantage that is computationally expensive because it performs the translation regardless whether the representation was changed or not subsequently performs an expensive equality check. As far as we know the only way to provide a sound shallow equality is by giving each constructor an unique identifier, similar to the identifiers for $Memo$ in the previous paragraph, and to check whether the identifiers are equal. However, this breaks referential transparency as equal values are not considered equal. However, in the event that we would subsequently apply the same representation to different values it is probable that the shallow equality holds. We delegate the decision on which, if any, optimisation to future work (section 7.2).

We now conclude the implementation of our interpretor. In the following sections we will provide algorithms for optimising the representation and we give a summary of this chapter. The benchmarks of the code can be found in chapter 5.

3.9 Optimisations

In this section we describe three optimisations that can improve the performance of the reactive model. These three optimisations use the fact that we can inspect and change the representation of the reactive model. We describe the ideas of these optimisations but leave the implementation as future work, section 7.2.

3.9.1 Lift to Haskell functions

As we saw before in section 3.7, applying change propagation at every node does not prevent recomputation and in fact only adds additional overhead. Therefore it would we beneficial if we could reduce the number of variables in our reactive model. We can achieve this in a two-pass optimisation. We first lift all nodes to their equivalent lifted Haskell function and subsequently compose the lifted Haskell functions. We can only lift a node to a Haskell function if its children can also be lifted to a Haskell function. We can lift any node to its Haskell equivalent except the $Cache$ and $Memo$ nodes. We illustrate this with the following two examples.

$$\begin{aligned}
&\mathit{dup} \ggg \mathit{arr}\ (+2)\ \&\&\&\ \mathit{arr}\ (+1) \ggg \mathit{arr}\ (+)\\
&\quad \equiv \{\ \text{lifting of } \&\&\&\ \}\\
&\mathit{dup} \ggg \mathit{arr}\ ((+2)\ \&\&\&\ (+1)) \ggg \mathit{arr}\ (+)\\
&\quad \equiv \{\ \text{def. of } \mathit{dup}\ \}\\
&\mathit{arr}\ (\lambda x \to (x, x) \ggg (+2)\ \&\&\&\ (+1) \ggg (+))
\end{aligned}$$

Notice that we can use the same operators inside the lifted Haskell functions as the Haskell function is also an instance of $Arrow$. We can lift the $\&\&\&$ to an Haskell function because none of its children contain a $Cache$ or $Memo$ constructor. The functions translated from $Cache$ and $Memo$ need to run in IO and thus cannot be lifted to a pure Haskell function. Therefore, we cannot directly optimise the following function.

$$\mathit{dup} \ggg (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2))\ \&\&\&\ \mathit{arr}\ (+1) \ggg \mathit{arr}\ (+)$$

However, recall that $f\ \&\&\&\ s \equiv \mathit{first}\ f \ggg \mathit{second}\ s$. So we can first use this rule to expand the $\&\&\&$ operator and subsequently optimise the result.

$$\begin{aligned}
&\mathit{dup} \ggg (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2))\ \&\&\&\ \mathit{arr}\ (+1) \ggg \mathit{arr}\ (+)\\
&\quad \equiv \{\ \text{expansion of } \&\&\&\ \}\\
&\mathit{dup} \ggg (\mathit{first}\ (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2)) \ggg \mathit{second}\ (\mathit{arr}\ (+1))) \ggg \mathit{arr}\ (+)\\
&\quad \equiv \{\ \text{assoc. of } \ggg\ \}
\end{aligned}$$

$$
\begin{aligned}
& \mathit{dup} \ggg (\mathit{first}\ (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2))) \ggg \mathit{second}\ (\mathit{arr}\ (+1)) \ggg \mathit{arr}\ (+) \\
& \quad \equiv \{ \text{lifting of } \mathit{second} \} \\
& \mathit{dup} \ggg (\mathit{first}\ (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2))) \ggg \mathit{arr}\ (\mathit{second}\ (+1)) \ggg \mathit{arr}\ (+) \\
& \quad \equiv \{ \text{comp. of} \ggg \text{and } \mathit{arr} \} \\
& \mathit{dup} \ggg (\mathit{first}\ (\mathit{Cache}\ (\equiv) \ggg \mathit{arr}\ (+2))) \ggg \mathit{arr}\ (\mathit{second}\ (+1) \ggg \mathit{uncurry}\ (+))
\end{aligned}
$$

3.9.2 Common Subexpression Elimination

Common Subexpression Elimination (Cocke, 1970) factors out the identical parts from a set of expressions. We illustrate this idea with the following example. In this example the expression $y * z$ is shared between the calculations of α and β. Therefore, we can first calculate the value of $y * z$ and then use it in the calculation of α and β. This prevents a second computation of $y * z$ and is therefore similar to memoising the $*$ operator.

$$
\begin{aligned}
& \alpha = t + y * z \\
& \beta = u + y * z \\
& \ \equiv \\
& e = y * z \\
& \alpha = t + e \\
& \beta = u + e
\end{aligned}
$$

By eliminating the amount of expressions in our reactive model we further reduce the overhead imposed by the change propagation, we reduce the need for memoisation, and we reduce the amount of lookups in the memo-table in the event that we factor out memoised expressions, all in one go.

3.9.3 Memo re-routing

Imagine the following situation where we define the function *fib* as we did in section 3.2.

```
fib = proc n → case n of
  0 → returnA −< 0
  1 → returnA −< 1
  n → do l ← fib′ −< (n − 2)
         ρ ← fib′ −< (n − 1)
         returnA −< (l + ρ)
```

If we would now define a function $\mathit{foo} = \mathit{memo}\ \mathit{fib}$. We would only memoise the outer calls to *fibm* and the runtime of *fib* would still be bad. An idea is to rewrite the recursive calls to point to $fibm$ instead of fib. If there is only one such case, where we add a memo constructor, then it is evident that we can perform the rewriting. In the case that we have another function $bar = memoHash\ fib$ that uses a different memoisation technique we can choose to either not perform the optimisation or to copy the implementation of fib and rewrite the recursive calls to their respective memoised versions.

3.10 Summary

We have seen that it is easy to add memoisation when approaching the problem from the perspective of a calculation. Because we create the reactive model in IO we prevent using $unsafePerformIO$ and pure memoisation functions that do not provide the programmer with the desired control over the memoisation.

The addition of change propagation makes some the implementation of some cases in the interpretor less intuitive than their Haskell function counterparts. However, this is invisible to the programmer.

Despite the lack of support of modelling recursively reactive values, e.g. reactive lists, algorithms over recursive datatypes benefit from memoisation and the change propagation.

In chapter 5 we will see how much performance we gain by using memoisation and change propagation.

Chapter 4

Reflection

We have seen two different methods of representing a calculation. The first DSL, chapter 2, was based on the concept of an expression, or value. This made the process of implementing change propagation natural, however, implementing memoisation was difficult. The second DSL, chapter 3, was based on the concept of a calculation which made implementing memoisation natural and implementing change propagation difficult, see section 3.7. The first implementation supports reactive recursive datatypes, whereas the second does not.We have tabulated this summary in Table 4.1.

Both approaches have the problem that in general we cannot provide a predicate by which we can control whether to propagate a change or not. However, for the first approach this is means a missed chance for an additional optimisation in the change propagation process, whereas for the second approach it means that the change propagation becomes useless in terms of performance.

The approach to define an embedded DSL has the advantage that it supports a platform for easy experimentation with optimisation strategies as well as support optimisation of calculations constructed during- runtime.

Features	Implementation 1	Implementation 2
Recursive functions	✓	✓
Recursive reactive data types	✓	✗
Memoisation	✗	✓
Sharing of memo tables	n/a	✓
Change Propagation	✓	✓, but limited

Table 4.1: Overview of the capabilities of Implementation 1 and Implementation 2

Chapter 5

Benchmarks

In this chapter we describe three benchmarks performed to validate our methods. The first benchmark shows that memoisation works. The second benchmark shows that we have obtained Third-level Sharing, The third example shows that change propagation works. The functions used for these benchmarks are $\mathit{fib}\ n$, $\mathit{fib}\ n + \mathit{fib}\ n$, and $x + \mathit{fib}\ n$, respectively.

Third-level Sharing

Environment The benchmarks are performed by criterion (O'Sullivan, 2013) on a Intel i7-2600K processor with 24GB of RAM running OS X 10.8.4. A new Haskell runtime is started for each test to prevent unwanted sharing by a smart GHC. Every benchmark is run by starting the same executable to ensure the executable is in memory for each test. The overhead of starting a new executable and starting the Haskell runtime is $4.5ms$. For additional speed we compiled the hashtables package (Collins, 2012) with `SSE4.1` optimisation *on*. The results of each calculation are printed to force full evaluation.

Legend In our benchmarks results we used several variants of the same calculation. We will now explain them.

`Native` This is either native Haskell code or the DSL from implemention 1 or 2 translated to a Haskell function.

`Native+Memo` The same as above with native memoisation using a list as a table as seen in subsection 1.2.1.

`Linear` The linear implementation of fib using a list as we will see in section 7.1.

`LinRec` The linear recursion implementation as found in B.

`CP` The interpretation of the DSL with change propagation enabled.

`UnsafeMemo` The interpretation of the DSL with memoisation in IO performed inside an $unsafePerformIO$.

`MemoIO` The interpretation of the DSL creating an IO function that cleanly uses the memoisation.

`WeakMemoIO` As above but using weak memo-tables.

Remarks We have chosen inputs for our functions that cause significant and comparable run times. When comparing the run-time results we noticed that sometimes a calculation that should take longer is in fact faster as we can see in Table 5.3 with the last two results for `Native+memo` and the results for `Native`, `Linear`, and `LinRec` in Table 5.1. We can only attribute this to inaccuracies in the measurement.

Memoisation To show that memoisation works we have used the calculations $fib\ n$ and $fib\ n + fib\ n$. We use $fib\ n$ to verify that memoisation works. We tested the speed of the memoised function for different n and tabulated the results in Table 5.1. We see that fib performs better if we add memoisation; however, it also shows that using a different algorithm yields even better results. The run times for the `Linear` versions and `LinRec` versions are better than when using memoisation. The comparison might not be entirely fair due to that the native Haskell versions are heavily optimised by GHC whereas the interpreted DSL receives no such optimisation.

The benchmark $fib\ \ n + fib\ \ n$ is used to check whether we have Third-level Sharing. The results are show in Table 5.2. The column header 500, 500 denotes that first 500 is used as an input and that the input is set to 500 a second time. When using memoisation the run time should not double. We see indeed that writing the calculation $fib\ n$ twice does not cause the calculation to occur twice as well.

Third-level Sharing

Change propagation In order to test whether change propagation works we use the calculation $x + fib\ n$ where n is a fixed number. The value of n depends on the used interpretation and is choosen so that $fib\ n$ takes the majority of the time. The results are shown in Table 5.4. We see that in all cases except the native variants, without change propagation or memoisation, the run time of changing the value of x does not cause the recalculation of $fib\ n$ as the run time is not doubled.

fib n	0 (overhead) (s)	35 (s)	500 (s)	5000 (s)
		Native		
Native	$4.79e^{-3}$	$8.60e^{-2}$	∞	∞
Native+memo	$4.77e^{-3}$	$4.73e^{-3}$	$5.83e^{-3}$	$8.17e^{-2}$
Linear	$4.72e^{-3}$	$4.79e^{-3}$	$4.81e^{-3}$	$6.35e^{-3}$
LinRec	$4.82e^{-3}$	$4.78e^{-3}$	$4.75e^{-3}$	$4.92e^{-3}$
		System1		
Native	$5.51e^{-3}$	2.36	∞	∞
CP	$5.07e^{-3}$	∞	∞	∞
		System2		
Native	$4.74e^{-3}$	5.43	∞	∞
UnsafeMemo	$4.89e^{-3}$	$5.93e^{-3}$	$5.93e^{-3}$	$1.58e^{-2}$
MemoIO	$5.08e^{-3}$	$5.28e^{-3}$	$5.73e^{-3}$	$1.17e^{-2}$
WeakMemoIO	$5.11e^{-3}$	$5.66e^{-3}$	$6.19e^{-3}$	$1.18e^{-2}$
MemoIO+CP	$5.88e^{-3}$	$8.89e^{-3}$	0.772	±360

Table 5.1: Benchmark of *fib* n for a single calculation for different n.

Summary We have shown that the use of memoisation and change propagation trough our platform increases the run-time efficiency of Haskell programs.

fib n	500 (s)	500, 500 (s)	500, 500, 500, 500
		Native	
Native+memo	$5.83e^{-3}$	$8.20e^{-2}$	$8.26e^{-2}$
Linear	$4.81e^{-3}$	$7.76e^{-3}$	$9.67e^{-3}$
Linrec	$4.75e^{-3}$	$5.35e^{-3}$	$6.18e^{-3}$
		System2	
UnsafeMemo	$5.93e^{-3}$	$1.51e^{-2}$	$1.53e^{-2}$
MemoIO	$5.73e^{-3}$	$1.27e^{-2}$	$1.28e^{-2}$
WeakMemoIO	$6.19e^{-3}$	$1.39e^{-2}$	$1.40e^{-2}$
MemoIO+CP	0.772	0.765	0.771

Table 5.2: Benchmark of *fib* n for a sequence of calculations for different n.

$fib\ n + fib\ n$	0 (s)	35	5000	5000, 5000
Native				
Native	$5.52e^{-3}$			
Native+memo	$4.99e^{-3}$	$5.05e^{-3}$	$8.52e-e2$	$8.43e^{-3}$
Linear	$4.93e^{-3}$	$5.04e^{-3}$	$7.76e-e3$	$1.09e^{-2}$
LinRec	$5.01e^{-3}$	$5.11e^{-3}$	$5.43e-e3$	$7.20e^{-3}$
System2				
Native	0.498	0.997	∞	∞
UnsafeMemo	$5.94e^{-3}$	$5.63e^{-3}$	$1.57e-e2$	$1.71e^{-2}$
MemoIO	$5.30e^{-3}$	$5.51e^{-3}$	$1.28e-e2$	$1.47e^{-2}$
WeakMemoIO	$5.30e^{-3}$	$5.73e^{-3}$	$1.31e-e2$	$1.47e^{-2}$
MemoIO+CP	$6.92e^{-3}$	$1.05e^{-2}$	∞	∞

Table 5.3: Benchmarks indicating the presence of Third-level Sharing.

$x + fib\ n$	0 (overhead) (s)	0,1 (s)
Native		
Native	$1.42e^{-2}$	$2.35e^{-2}$
Native+memo	$8.34e^{-2}$	$8.36e^{-2}$
Linear	$9.31e^{-3}$	$1.12e^{-2}$
LinRec	$7.09e^{-3}$	$8.55e^{-3}$
System1		
Native	0.228	0.408
CP	0.363	0.374
System2		
Native	0.498	0.997
UnsafeMemo	$1.61e^{-2}$	$1.90e^{-2}$
MemoIO	$1.42e^{-2}$	$1.88e^{-2}$
WeakMemoIO	$1.42e^{-2}$	$1.88e^{-2}$
MemoIO+CP	$1.06e^{-2}$	$1.04e^{-2}$

Table 5.4: Benchmark of $x + fib\ n$ for a single calculation for different n.

Chapter 6

Related work

In this thesis we presented a platform for achieving incremental evaluation by combining memoisation and change propagation and in this chapter we first describe the fundamental contributions to both memoisation and change propagation in section 6.1 and, subsequently, we relate our platform to existing platforms for achieving incremental evaluation in section 6.2.

In this chapter we review work related to this thesis and relate them to the concept of incremental evaluation. As we have not found a single definition of incremental computation we give a definition that captures the essences of the found definitions.

We define the evaluation of an expression as an incremental evaluation if the evaluation of said expression reuses the results from previously evaluated expressions for which it is known that re-evaluating them would not yield a different value. incremental evaluation

6.1 Foundations

6.1.1 Memoisation

Hughes was the first to describe (Hughes, 1985) memoisation in functional languages and he introduced the following four important concepts which we describe afterwards.

1. The definition of a memo-function;
2. *Lazy* memo-functions;

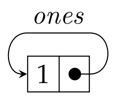


Figure 6.1: Cyclic representation of *ones*

3. Infinite structures can be represented as cyclic structures by applying memoisation;
4. The concept of weak memo-tables and their necessity.

Memo-function A memo-function is defined as follows: “a memo-function is like an ordinary function, but it remembers all the arguments it is applied to, together with the results computed from them."

Lazy memo-functions Lazy memo-functions use reference equality for non-atomic values instead of structured equality. This prevents the complete evaluation of the value and, therefore, does not introduce strictness in the memoised function, in contrast to normal memo-functions that create a strict memo- function. Using reference equality has the disadvantage that equal values are not recognized as such if they are stored in different locations in the memory. When two equal values are not recognized as such, the calculation is repeated and the result is stored in the memo-table for a second time. To prevent this, we ensure that each value exists in the memory only once by memoising the constructor functions.

Cyclic structures By applying memoisation, infinite structures can be represented as cyclic structures. We illustrate this with the textbook example of mapping $(*2)$ over a list of ones. We define the list of ones as the following recursive Haskell function.

$$ones = 1 : ones$$

The function ones can be represented as the cyclic structure in Figure 6.1.

We can obtain a list of twos by mapping the function $*2$ over the list of ones. However, the list of doubles will be an infinite structure as opposed to the cyclic structure of *ones* as we explain shortly.

$$\begin{aligned}
&twos = map\ (*2)\ ones \\
&map\ f\ []\qquad\quad = [] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}$$

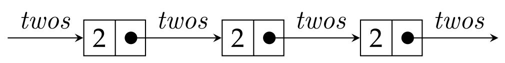


Figure 6.2: Infinite representation of *twos*

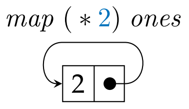


Figure 6.3: Cyclic representation of *twos*

By its definition, map is unable to distinguish between infinite lists and lists and cyclic lists. The result of map can only be cyclic if the input was a cyclic list and. Therefore, the result of map is always an infinite list. We illustrate this with showing the evaluation of twos in equational as follows and graphically in Figure 6.2

$$\begin{aligned}
& map\ (*\ 2)\ ones \\
\equiv & \ \{ \text{definition of } ones \} \\
& map\ (*\ 2)\ (1 : ones) \\
\equiv & \ \{ \text{definition of } map \} \\
& (*\ 2)\ 1 : map\ (*\ 2)\ ones \\
\equiv & \ \{ \text{application of } (*\ 2) \} \\
& 2 : map\ (*\ 2)\ ones
\end{aligned}$$

By memoising map we create the cyclic structure as illustrated in Figure 6.3. Because the arguments to map are equal the memoisation returns the reference to the result. The arguments are equal because the reference to $(*\ 2)$ is equal and the list is equal due to the cyclic structure of $ones$.

Weak memo-tables We need weak memo-tables to ensure that we do not unnecessarily store results in our memo-tables. It is unnecessary to store results when the associated key if the memo-function will not be applied to the key. It is certain that the memo-function will not be applied to the key if the key is, or has become, inaccessible, from the program. Whether a key is accessible, or *live*, is determined by whether another live object has a reference to it. If no live object has a reference to an object it is dead and it can be garbage collected. The problem with normal memo-tables is that they keep a reference to all keys and results, thereby keeping them alive and preventing their garbage collection causing unnecessary memory usage. Weak memo-tables solve this by using weak references instead of normal references. Weak references are ignored when determining the liveness of an object. This allows the keys and results to be garbage collected when necessary.

Weak memo-tables in Haskell

Peyton Jones and Marlow implement memoisation using weak memo- tables (S. L. Peyton Jones et al., 2000) and show how to construct the weak memo-tables. Weak memoisation adds additional overhead to the insert and lookup operations as for each insert additionally a weak pointer has to be created and for each successful lookup the weak reference has to be traversed to check whether the value was garbage collected. We verified the existence of some overhead with our own benchmark in chapter 5 and found it to be marginal. It could also be argued that this is the price one has to pay for automatic garbage collection.

In contrast, we allow the programmer to make the choice whether he wants to memoise in the style of lazy memo-functions and or whether he wants to use weak memo-tables.

Side-effect free memoisation

In contrast to the aforementioned stateful implementation of memoisation (S. L. Peyton Jones et al., 2000) the following two contributions are purely functional and rely on lazy evaluation.

Memo-tries The idea behind memo-tries (Hinze, 2000) is to memoise a function $f\ x$ by mapping the function f over all possible values of x. This is the same approach as we saw in the $mfib$ example in the introduction (subsection 1.2.1). With memo-tries, however, f is mapped over generic representations of the values in the case of memo-tries.

There is another difference with the version we introduced in our introduction as we show with the following implementation of $mfib$. This uses the implementation of memo-tries: the `MemoTrie` package (Elliot, 2012).

$$mfib = memo\ fib$$

$$\begin{aligned} &fib\ 0 = 0 \\ &fib\ 1 = 1 \\ &fib\ n = mfib\ (n-2) + mfib\ (n-1) \end{aligned}$$

The difference between this version and the version from subsection 1.2.1 is that the map structure is shared between calls. Therefore, $mfib\ n + mfib\ n$ shows the behaviour we set out to achieve in our goal (section 1.3).

Although this solution is a textbook example of functional programming it doesn't provide the programmer with control over the memoisation process and trade-off for the obtained sharing is that the whole computed trie remains in memory until the program ends.

Memoising catamorphisms The previous contributions provide a solution for automatic memoisation, requiring only the addition of *memo*, either as keyword or function, whereas the following contribution requires manual effort.

The solution presented by Leather et al. (Leather et al., 2010) *annotates* a datatype with attributes in order to prevent their recalculation.

The *depth* of a tree is the maximum distance to one of its root nodes and is easily calculated with a recursive function. Observe that the use of a recursive function inside another recursive function is not efficient as illustrated by the following example (Hughes, 1985). The function *depth* determines the depth of a binary tree and traverses the tree once. The function *deepest* determines the values of the deepest leafs by using *depth* at each recursive step.

```
data Tree α = Leaf α | Bin (Tree α) (Tree α)
depth :: Tree α → Int
depth (Leaf _)  = 0
depth (Bin l ρ) = 1 + (depth l ‘max‘ depth ρ)
deepest :: Tree α → [α]
deepest (Leaf α)  = [Leaf α]
deepest (Bin l ρ) = case depth l ‘comp‘ depth ρ of
   LT → deepest l
   GT → deepest ρ
   EQ → deepest l ++ deepest ρ
```

Because *depth* is called for each level of the tree it is calculated n times for each node where n is the depth of the node. The authors observe (Leather et al., 2010) that *depth* is actually an *attribute* of the Tree and they annotate the datatype at the recursive point with a wrapper and fix-point notation so that a tree is representated as follows.

$$Bin^{1}\ Leaf^{0}\ Leaf^{0}$$
$$Bin^{2}\ (Bin^{1}\ Leaf^{0}\ Leaf^{0})\ Leaf^{0}$$

The functions *depth* and *deepest* are examples of functions that can be rewritten to use catamorphisms. Catamorphisms abstract away recursive traversal and take a set of instructions on what should be done at the recursive points, called an algebra, and the actual datatype to transform. The ubiquitous function *foldr* is a catamorphism for lists. Leather et al. lift the observation above to catamorphisms. The only effort required by the programmer is the annotation of the datatype with attributes that should be memoised and referencing it from the algebra.

Use-cases Memoisation has been used for implementing incremental evaluation of attribute grammars (W. Pugh and Teitelbaum, 1989; Kuiper and

Saraiva, 1998) as well as for incremental evaluation of *higher-order* attribute grammars (Saraiva et al., 2000). Another use case is the creation of incremental parser combinators (Cook and Launchbury, 1997; Frost et al., 2007).

6.1.2 Change propagation

Change propagation has mainly been researched in the context of incremental evaluation of AGs and by and in the context of ML. We first discuss change propagation in ML, followed by the work on Attribute Grammars.

6.1.3 Adaptive Programs

Both the implementation of change propagation used in this contribution and the implementation by Acar (U. A. Acar, 2005) in the language ML are based on dynamically constructing the Data-Dependency Graph (DDG). The solution presented in this contribution is based on the approach taken by Acar. We describe the similarities and the main difference between the two approaches.

What we in this thesis defined as *reactive* is called *adaptive* by Acar and What we call a *variable* is called a *modifiable*. Two additional types are introduced by Acar: *changeable* and *destination*. A destination is a modifiable that can only be written to and a changeable is stored inside a modifiable.

Both approaches provide a set of operators to model the reactive model. Acar introduces the functions *mod*, *read* and *write*. The function *mod* accepts a function that creates a changeable from a destination and creates a modifiable and a predicate that determines whether propagation should take place. The function *read* accepts a modifiable and a function that given a value will create a changeable and results in a changeable. The function *write* takes a destination and value and yields a new destination. The types of these function can be represented by the following Haskell code.

```
data Modifiable α
data Destination α
data Changeable
mod   :: (α → α → Bool) → (Destination α → Changeable) → Modifiable α
read  :: Modifiable α   → (α → Changeable)             → Changeable
write :: Destination α  → α                            → Changeable
```

The propagation of changes is split up in two parts. The function *change* takes a modifiable and actually assigns a given value to the modifiable in contrast to the function *write* that models an assignment. The propagation of the changes in values is started by calling the function *propagate*.

The counterpart of *mod* from our system is *newRVar* and *read* and *write* are combined in the *linkRVar* function. Similarly, the functions *change* and *propagate* are combined in *setRVar*. This means that Acar can change several variables before performing the propagation which is potentially more efficient as it potentially reduces the amount of re computations.

The main difference between the two systems is how they store the DDG. In the adaptive model they are stored separately from the modifiables; whereas in our system the variables themselves know their dependencies. The advantage of storing the graph separately is that it is easy to implement algorithms the whole graph. The disadvantage, however, is that efficiently maintaining an efficient representation of the graph is difficult. Several of the improvements made over time to the algorithms involved the maintainance of the graph (U. Acar et al., 2006; Ley-Wild et al., 2009; Chen et al., 2012).

6.1.4 Finite differencing

In addition, there is a third technique for achieving incremental evaluation, namely, finite differencing (Paige and Koenig, 1982). Finite differencing is the application of algebraic knowledge of the operators and datatypes used in an expression in order to prevent the re- evaluation. Finite differencing can be applied statically, as illustrated by the CSE optimisation as shown in section 3.9, or dynamically as shown in section 6.2.

6.2 Platforms for incremental evaluation

Attribute Grammars According to Acar (U. A. Acar, Blelloch, and Harper, 2006) the idea to use dependency graphs for incremental updates was first introduced by Demers et al. (Demers et al., 1981) followed by a working implementation by Reps and Titlebaum (Reps et al., 1983). They create a static dependency graph between attributes at compile-time; therefore, higher-order attribute grammars are not supported (Vogt et al., 1989). Their system is similar to the approach as described in the introduction subsection 1.2.2. In order to determine the order in which attributes can be calculated the AG compiler needs to determine the dependencies between the attributes. Therefore, the dependencies are known statically and this can be used to hard-code which attributes should be updated if an attribute changes. In contrast, in our system and the system we describe in the following section the dependencies are constructed at run- time.

Incremental evaluation is obtained for free, without any additional effort from the programmer.

Higher-order Attribute Grammars Equality on functions is hard and Haskell provides no mechanism to determine equality of functions; therefore, memoising higher-order functions is difficult. There are no frameworks that support memoising higher-order functions. By using our system however the programmer can implement equality check on functions based on pointer equality which may or may not suit his particular situation.

However, by using the platform presented by Saraiva et al. (Saraiva et al., 2000) one *can* achieve incremental evaluation of higher-order Attribute Grammars which are as powerful as higher-order functions. Saraiva uses a purely functional representation of Attribute Grammars, and the built-in memoisation features of LRC.

ML programs The work presented by Chen et al. (Chen et al., 2012) drastically reduces the amount of work required by the programmer to obtain incremental evaluation through change propagation. The only effort required by the programmer is the annotation of the type; subsequently, the compiler transforms the code written by the programmer to a form using the operators from the library as described in subsection 6.1.3. In contrast, the solution presented in this thesis requires the programmer to rewrite their functions in applicative or monad style for method 1 or to arrow style for method 2.

Finite differencing in the Views system The Views System (Meertens and Pemberton, 1992) models computer programs by describing the constraints between values in the program. Ganzevoort shows that these constraints can be maintained incrementally (Ganzevoort, 1992). His implementation uses the Observer pattern (Gamma, 1995) to notify the dependencies of an object which method was called and which what arguments. Each dependency then updates it own internal state, possibly triggering subsequent notifications.

We can implement this in pseudo object oriented code:

```
class List {
  ...
  void push(value) {
    // update internal representation
    observers.all $ \ o -> o.onPush(value);
  }

  Sum sum() {
    Sum res = new Sum(sumvalue);
    observers.add res;
    return res;
  }
}

class Sum {
  int sum;
  function onPush(value) {
    self.sum += value;
  }
}
```

The advantage of this method is that we can incorporate algebraic knowledge on the operations and the datatypes in the observers to efficiently update the state of the observer. Essentially this is performing finite differencing at run-time.

However, there are at least two problems with this approach.

1. **Error prone**

   The semantics of a method f in class α are encoded in at least two places: in the implementation of f and in each class that handles notifications for objects of class α.

2. **Efficiency**

   Sometimes the information provided by the method call is not enough.

We illustrate this last point with an example on lists. Each list has the methods $push\ (value)$ that pushes a value on the list, $pop\ ()$ that deletes the value , and $sum\ ()$ that creates a sum object that acts as an observer to the list. When a value is pushed on the list the Sum object recieves the notification that a $add\ (value)$ has happened; therefore, it can add the pushed value to its internal value. When a value is popped from the list, however, the message $pop\ ()$ contains no information on the value of the popped element and the whole array has to be traversed again to calculate the sum.

6.3 Overview

We have summarized the information in this chapter in Table 6.1.

Contribution	Control	Implementation	Syntax	Language	Evaluation
		Change Propagation			
U. A. Acar (2005)	Fine	Library	Verbose	ML	Strict
U. A. Acar, Blelloch, Blume, et al. (2009)*	Fine	Library	Verbose	ML	Strict
Carlsson (2002)	Fine	Library	Monads	Haskell	Strict
Chen et al. (2012)*	None	Compiler	Type annotation	(T)ML	Strict
Reps et al. (1983)	None	Compiler	Equal	AG	?
W. W. Pugh (1988)	None	Compiler	Equal	AG	?
Ganzevoort (1992)	Fine	Paradigm	Very verbose	OOP	Strict
		Memoisation			
Leather et al. (2010)	Fine	Manual/Library	More verbose	Haskell	Lazy
S. L. Peyton Jones et al. (2000)	None	Library	Equal	Haskell	Strict
Hinze (2000)	None	Library	Equal	Haskell	Lazy
Hughes (1985)	None	Compiler	Equal	FP	Lazy
		Memoisation + Change Propagation			
This paper	Fine	Library	Slightly more verbose	Haskell	Lazy

* Memo is used for implementing CP efficiently

Table 6.1: Overview of techniques for memoisation and memoisation

Chapter 7

Wrapping up

7.1 Discussion

Better algorithms We have proposed the use of memoisation and change propagation to improve the performance of Haskell programs; however, there are other methods to improve performance. The use of a better algorithm even further lowers the run-time complexity of an algorithm as we will illustrate with the following implementation of *fib* and as we saw in chapter 5.

As we saw, it is possible to obtain a run-time complexity of $\mathrm{O}(n^2)$ by using memoisation; however, we can even obtain a linear run-time complexity.

$$\begin{array}{l} \mathit{fib}\ n = \mathit{fibs}\ !!\ n \\ \mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs}) \end{array}$$

The value *fibs* is an infinite list that will be evaluated as far as necessary due to lazy evaluation. Furthermore, *fibs* is shared between all calls to *fib* thereby creating Third-level Sharing.

Third-level Sharing

Better data structures Obtaining better performance without using memoisation or change propagation can also be achieved by changing the data structure itself; e.g. adding a constructor for the concatenation of two lists improves the efficiency of the concatenation while a fold over the list can still be performed in linear time. Using a different datatype could be beneficial when using change propagation as well. When using only change propagation pushing an element on the front of a list is expensive as the whole sub list has to be traversed again, using a snoclist is more efficient in that case.

Devising a faster algorithm or data structure is non-trivial for complicated real-world scenarios, especially for the average programmer; therefore, a general solution such as memoisation, change propagation or the combination, might be a better choice in terms of effort.

Convenience of change propagation Even though the overhead added by change propagation might decrease the performance of a program it might be convenient to use for a programmer as it will let the program react automatically to changes in the input without much effort by the programmer.

7.2 Future work

Optimisations for change propagation We have proposed several optimisations to generate a better performing reactive model. Further research can be done to figure out which optimisations would benefit change propagation. In theory one wants to let the propagation stop as early as possible, but maybe it is better to lift more functions to Haskell functions and thus reduce the overhead created by change propagation. The optimal division between change propagation and pure Haskell functions is currently not known and is likely to be different for each use-case.

Reactive recursive datatypes Support reactive recursive datatypes through a generic diff algorithm in our second system similar to Proxima 2 (Schrage, 2010). If we can find a way to transform a $\delta\alpha$ into a $\delta\beta$ we can eliminate the use of IO in our programs.

Memoisation to improve change propagation Implement the use of memoisation to improve the performance of change propagation as mentioned in section 2.3.

Embed in compiler Perform the interpretation of the model at compile-time to reduce the overhead of interpreting parts of the model that are statically known.

7.3 Conclusion

Stateless memoisation and change propagation We have added memoisation and change propagation to pure functions by describing the functions in a pure language, the DSL introduced in chapter 3, and subsequently interpreting the language inside the *IO* monad.

Fine-tuning of the memoisation, Third-level sharing We offer the programmer control over the memoisation through the memoisation constructor which delegates the insert and lookup functions to the programmer allowing for complete control over the amount of memoisation performed. Additionally, the system

Combination of Memoisation and Change Propagation We provided a basic library for implementing change propagation and we used it to create a platform that combines both change propagation and memoisation.

Easy-to-use We offer the programmer an easy-to-use interface via the `arrow-syntax`. Functions written in the arrow-syntax resemble their non-arrow versions.

We have also shown that using memoisation and change propagation with this system can increase the performance of an Haskell program and that there are situations where the use of a better algorithm or data structure deliver higher performance increases.

Bibliography

Acar, Umut A. (2005). “Self-adjusting computation”. AAI3166271. PhD thesis. Pittsburgh, PA, USA. ISBN: 0-542-01547-1.

Acar, Umut A., Guy E. Blelloch, Matthias Blume, et al. (Nov. 2009). “An experimental analysis of self-adjusting computation”. In: *ACM Trans. Program. Lang. Syst.* 32.1, 3:1–3:53. ISSN: 0164-0925. DOI: 10.1145/1596527.1596530. URL: http://doi.acm.org/10.1145/1596527.1596530.

Acar, Umut A., Guy E. Blelloch, and Robert Harper (2002). “Adaptive functional programming”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’02. Portland, Oregon: ACM, pp. 247–259. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503296. URL: http://doi.acm.org/10.1145/503272.503296.

– (Nov. 2006). “Adaptive functional programming”. In: *ACM Trans. Program. Lang. Syst.* 28.6, pp. 990–1034. ISSN: 0164-0925. DOI: 10.1145/1186632.1186634. URL: http://doi.acm.org/10.1145/1186632.1186634.

Acar, Umut et al. (Mar. 2006). “A Library for Self-Adjusting Computation”. In: *Electron. Notes Theor. Comput. Sci.* 148.2, pp. 127–154. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.11.043. URL: http://dx.doi.org/10.1016/j.entcs.2005.11.043.

Baars, Arthur I. et al. (2009). “Typed transformations of typed abstract syntax”. In: *Proceedings of the 4th international workshop on Types in language design and implementation*. TLDI ’09. Savannah, GA, USA: ACM, pp. 15–26. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481865. URL: http://doi.acm.org/10.1145/1481861.1481865.

Carlsson, Magnus (2002). “Monads for incremental computing”. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ICFP ’02. Pittsburgh, PA, USA: ACM, pp. 26–35. ISBN: 1-58113-487-8. DOI: 10.1145/581478.581482. URL: http://doi.acm.org/10.1145/581478.581482.

Chen, Yan et al. (2012). “Type-directed automatic incrementalization”. In: *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. PLDI ’12. Beijing, China: ACM, pp. 299–310. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254100. URL: http://doi.acm.org/10.1145/2254064.2254100.

Cocke, John (July 1970). "Global common subexpression elimination". In: *SIGPLAN Not.* 5.7, pp. 20–24. ISSN: 0362-1340. DOI: 10.1145/390013.808480. URL: http://doi.acm.org/10.1145/390013.808480.

Collins, Gregory (2012). *hashtables*. [Online, accessed 10-july-2013]. URL: http://hackage.haskell.org/package/hashtables.

Cook, Byron and John Launchbury (1997). "Disposable Memo Functions". In: *In Proceedings of the 1997 Haskell Workshop*.

Demers, Alan et al. (1981). "Incremental evaluation for attribute grammars with application to syntax-directed editors". In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '81. Williamsburg, Virginia: ACM, pp. 105–116. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567544. URL: http://doi.acm.org/10.1145/567532.567544.

Elliot, Conal (2012). *MemoTrie*. URL: http://hackage.haskell.org/package/MemoTrie.

Frost, Richard A. et al. (2007). "Modular and efficient top-down parsing for ambiguous left-recursive grammars". In: *Proceedings of the 10th International Conference on Parsing Technologies*. IWPT '07. Prague, Czech REpublic: Association for Computational Linguistics, pp. 109–120. ISBN: 978-1-932432-90-9. URL: http://dl.acm.org/citation.cfm?id=1621410.1621425.

Gamma, E. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley. ISBN: 9780201633610. URL: http://books.google.nl/books?id=iyIvGGp2550C.

Ganzevoort, Job (1992). *Maintaining presentation invariants in the Views system.* Tech. rep. Amsterdam, The Netherlands, The Netherlands.

GHC (2013). *7.15. Arrow notation*. [Online, accessed 4-february-2013]. URL: http://www.haskell.org/ghc/docs/latest/html/users_guide/arrow-notation.html.

Gill, Andy (2009). "Type-safe observable sharing in Haskell". In: *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. Haskell '09. Edinburgh, Scotland: ACM, pp. 117–128. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596653. URL: http://doi.acm.org/10.1145/1596638.1596653.

– (2011). *data-reify*. URL: http://hackage.haskell.org/package/data-reify.

Hinze, Ralf (2000). "Memo Functions, Polytypically!" In: *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*, pp. 17–32.

Hughes, John (1985). "Lazy memo-functions". In: *Proc. of a conference on Functional programming languages and computer architecture*. Nancy, France: Springer-Verlag New York, Inc., pp. 129–146. ISBN: 3-387-15975-4. URL: http://dl.acm.org/citation.cfm?id=5280.5289.

– (May 2000). "Generalising monads to arrows". In: *Sci. Comput. Program.* 37.1-3, pp. 67–111. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(99)00023-4. URL: http://dx.doi.org/10.1016/S0167-6423(99)00023-4.

Kuiper, Matthijs F. and João Saraiva (1998). "Lrc - A Generator for Incremental Language-Oriented Tools". In: *Proceedings of the 7th International Conference on*

Compiler Construction. CC '98. London, UK, UK: Springer-Verlag, pp. 298–301. ISBN: 3-540-64304-4. URL: http://dl.acm.org/citation.cfm?id=647474.727599.

Leather, Sean et al. (2010). "Pull-ups, push-downs, and passing it around". In: *Proceedings of the 21st international conference on Implementation and application of functional languages*. IFL'09. South Orange, NJ, USA: Springer-Verlag, pp. 159–178. ISBN: 3-642-16477-3, 978-3-642-16477-4. URL: http://dl.acm.org/citation.cfm?id=1929087.1929097.

Ley-Wild, Ruy et al. (2009). "A cost semantics for self-adjusting computation". In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '09. Savannah, GA, USA: ACM, pp. 186–199. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480907. URL: http://doi.acm.org/10.1145/1480881.1480907.

Meertens, Lambert G.L.T. and Steven Pemberton (1992). *The ergonomics of computer interfaces – Designing a system for human use.* Tech. rep. Amsterdam, The Netherlands, The Netherlands.

Milner, Robin et al. (May 15, 1997). *The Definition of Standard ML - Revised*. Rev Sub. The MIT Press. ISBN: 0262631814. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20%5C&path=ASIN/0262631814.

O'Sullivan, Brian (2013). *Criterion*. [Online, accessed 10-july-2013]. URL: http://hackage.haskell.org/package/criterion.

Ozaki, Kohei (2011). *hubigraph*. [Online, accessed 30-july-2013]. URL: http://hackage.haskell.org/package/hubigraph.

Paige, Robert and Shaye Koenig (July 1982). "Finite Differencing of Computable Expressions". In: *ACM Trans. Program. Lang. Syst.* 4.3, pp. 402–454. ISSN: 0164-0925. DOI: 10.1145/357172.357177. URL: http://doi.acm.org/10.1145/357172.357177.

Paterson, Ross (2001). "A new notation for arrows". In: *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. ICFP '01. Florence, Italy: ACM, pp. 229–240. ISBN: 1-58113-415-0. DOI: 10.1145/507635.507664. URL: http://doi.acm.org/10.1145/507635.507664.

– (2010). *Arrows: A General Interface to Computation*. [Online, accessed 9-july-2013]. URL: http://www.haskell.org/arrows/.

Peyton Jones, Simon (Jan. 2003). "Haskell 98". In: *J. Funct. Program.* 13.1, pp. .1–255. ISSN: 0956-7968. DOI: 10.1017/S0956796803000315. URL: http://dx.doi.org/10.1017/S0956796803000315.

Peyton Jones, Simon L. et al. (2000). "Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell". In: *Selected Papers from the 11th International Workshop on Implementation of Functional Languages*. IFL '99. London, UK, UK: Springer-Verlag, pp. 37–58. ISBN: 3-540-67864-6. URL: http://dl.acm.org/citation.cfm?id=647978.743369.

Pugh, W. and T. Teitelbaum (1989). "Incremental computation via function caching". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Prin-*

ciples of programming languages. POPL '89. Austin, Texas, USA: ACM, pp. 315–328. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75305. URL: http://doi.acm.org/10.1145/75277.75305.

Pugh, William W. (1988). *Incremental Computation and the Incremental Evaluation of Functional Programs*. Tech. rep. Ithaca, NY, USA.

Reps, Thomas et al. (July 1983). "Incremental Context-Dependent Analysis for Language-Based Editors". In: *ACM Trans. Program. Lang. Syst.* 5.3, pp. 449–477. ISSN: 0164-0925. DOI: 10.1145/2166.357218. URL: http://doi.acm.org/10.1145/2166.357218.

Saraiva, João et al. (2000). "Functional Incremental Attribute Evaluation". In: *Proceedings of the 9th International Conference on Compiler Construction*. CC '00. London, UK, UK: Springer-Verlag, pp. 279–294. ISBN: 3-540-67263-X. URL: http://dl.acm.org/citation.cfm?id=647476.727632.

Schrage, Martijn (2010). *Proxima 2.0, WYSIWYG generic editing for the Web*.

ubietylab.net (n.d.). *Ubigraph*. [Online, accessed 30-july-2013]. URL: http://ubietylab.net/ubigraph/.

Vermeulen, Alessandro (July 2013). *The Difference Between Shallow and Deep Embedding*. [Online, accessed 28-july-2013]. URL: http://alessandrovermeulen.me/2013/07/13/the-difference-between-shallow-and-deep-embedding/.

Visser, Sebastiaan et al. (2013). *fclabels*. URL: http://hackage.haskell.org/package/fclabels.

Vogt, H. H. et al. (June 1989). "Higher order attribute grammars". In: *SIGPLAN Not.* 24.7, pp. 131–145. ISSN: 0362-1340. DOI: 10.1145/74818.74830. URL: http://doi.acm.org/10.1145/74818.74830.

Appendices

Appendix A

The reactive library

We created our own implementation of change propagation instead of using the existing `Adaptive` (Carlsson, 2002) library because it doesn't allow for the memoisation structure as we used in section 3.6.

A.1 The basic functions

The library we present here is a straightforward implementation of the concept of change propagation: A variable is a changeable piece of memory, and it potentially has other variables that depend on it, e.g., there are other variables that need to be updated when it changes.

As we saw before we have an $RVar$ containing the value and the meta-data. Note that the record fields are prefixed with an underscore. This is because we generate lenses by using `fclabels` (Visser et al., 2013).

```
type RVar α = IORef (RVal α)
data RVal α = RVal {
    _rMeta  :: RMeta α
  , _rValue :: α
  }
```

The meta-data consists of the reactors and the sources of the variable, the $IO\ ()$ actions that should be executed when the value of the variable changes, and debug information. This is stored as follows:

```
type OnValSet α = α → α → IO ()
data SomeRVar where
```

```
  SomeRVar :: RVar α → SomeRVar
data RMeta α = RMeta {
   _mReactors :: [(HandlerLink, OnValSet α, (Int, String))]
  , _mSource    :: [(SomeRVar, HandlerLink)]
  , _mEffects   :: [α → α → IO ()]
  , _mDebugID :: Int
  , _mDebugName :: String
}
```

A new variable with an initial value is created with the function *newRVar*.

```
newRVar :: α → IO (RVar α)
```

We can lookup the current value of a variable with the function *getRVal* and assign it a new value with the function *setRVar*. For convenience there is *printRVal* function that prints the variable to the standard output. The functions *getRVal* and *printRVar* simply access the value of the variable. The function *setRVal* is where most of the magic happens. When a value is set the *setRVal* function updates the value in the *RVar* and then performs all the associated effects and finally it notifies the reactors of the variable that its value has changed. reactors

```
getRVal :: RVar α        → IO α
getRVal = fmap (get rValue) ∘ readIORef
printRVar :: Show α ⇒ RVar α → IO ()
printRVar v = getRVal v ≫= print
setRVal :: RVar α → α → IO (RVar α)
setRVal this val = do
  this′  ← readIORef this
  setValue this val
  let oldValue = get rValue this′
  let thisID = get (mDebugID ∘ rMeta) this′
  let reactors = get (mReactors ∘ rMeta) this′
  { Notify listeners }
  let update f (rid, name) = do Viz.setRVar thisID rid
    f val oldValue
  mapM_ (λf → f val oldValue) (get (mEffects ∘ rMeta) this′)
  mapM_ (λ(_, f, n) → update f n) reactors
  return this
  where setValue :: RVar α → α → IO ()
    setValue rVal val = modifyIORef rVal (modify rValue (const val))
```

Two variables can be linked by the function *linkRVar*. The first argument becomes the reactor to the second argument. The third argument to *linkRVar* is the function that should be executed when the value of the source variable changes. This function is available but its main purpose is the use in the stateful abstraction functions. The function $modify'$ is the strict version of $modify$ which modifies the value at the provided path.

```
linkRVal :: RVar β → RVar α → OnValSet α → IO HandlerLink
linkRVal this that onSet = do
    link ← newLink this
    thisName ← fmap (get (mDebugName ∘ rMeta)) (readIORef this)
    thisID ← fmap (get (mDebugID ∘ rMeta)) (readIORef this)
    modifyIORef' that $!bindReact      link thisName thisID
    modifyIORef' this $!bindSource     link
      -- Visualization
    thisID ← fmap (get (mDebugID ∘ rMeta)) (readIORef this)
    thatID ← fmap (get (mDebugID ∘ rMeta)) (readIORef that)
    Viz.newLink thisID thatID
    return link
  where
    bindReact   link name rid = modify' (mReactors ∘ rMeta)
                                        ((link, onSet, (rid, name)) :)
    bindSource link            = modify' (mSource ∘ rMeta)
                                        ((SomeRVar that, link) :)
    removeHandler link         = filter  (λ(ehl, _, _) → ehl ≢ link)
```

The unlinking of a variable is done with the function $unlink$. The unlinking makes sure that the variable is not holding any references to other variables to ensure garbage collection. The variable is removed from the list of the reactors of the source variables and it removes any references to its sources.

```
unlink :: RVar α → IO ()
unlink rv = do
    v ← readIORef rv
    mapM_ p (get (mSource ∘ rMeta) v)
    modifyIORef rv (modify' (mSource ∘ rMeta) (const []))
  where
    p (SomeRVar rv, link) = modifyIORef' rv
      (modify' (mReactors ∘ rMeta)
        (filter (λ(l, _, _) → l ≢ link)))
```

A.2 The applicative and monadic functions

We will now explain the functions that expose the applicative and monadic programming style starting with *joinRVal* the counterpart of *join*.

We express bind in terms of *join* and *fmap*. The *join* function is the most complicated function in the library as it has to all of the administrative work. The case where the inner variable changes is easy as we can change the value of the result variable directly. The case where the value of the outer variable changes, i.e. we get a new inner variable, is more difficult. We need to unlink the result variable from the inner variable for which we need to know the link variable so we store that in the *IORef* called *store*. After this we read the value of the new inner variable and link the result to inner variable.

```
joinRVal :: RVar (RVar α) → IO (RVar α)
joinRVal outerR = do
  outerName   ← fmap (get (mDebugName ∘ rMeta)) (readIORef outerR)
  innerR      ← fmap (get rValue) (readIORef outerR)
  innerRValue ← fmap (get rValue) (readIORef innerR)
  innerName   ← fmap (get (mDebugName ∘ rMeta)) (readIORef innerR)
  res ← newNamedRVar ("join: ( " ++ outerName ++ " (" ++ innerName ++ "))")
                     innerRValue
    -- Link to innerR
  let onInnerSet n o = void $ setRVal res n
  innerLink ← linkRVal res innerR onInnerSet
  store ← newIORef innerLink
    -- Link to outerR
  let onOuterSet newInner o = do
    innerLink ← readIORef store
    modifyIORef' o (modify'   (mReactors ∘ rMeta)
                              (const []))
    modifyIORef' res (modify' (mSource ∘ rMeta)
                              (filter (λ(_, l) → l ≢ innerLink)))
    newLink ← linkRVal res newInner onInnerSet
    writeIORef store newLink
    newValue ← fmap (get rValue) (readIORef newInner)
    setRVal res newValue
    return ()
  linkRVal res outerR onOuterSet
  return res
```

The following function *rValFMap* is the counterpart of *fmap* and is really simple. We read the value of the source variable, we create the new variable with the result of the application of the function to the value of the source variable as its value and we link the resulting variable so that the function is applied for every set of the source variable.

```
rValFMap :: (α → β) → RVar α → IO (RVar β)
rValFMap f source = do
  sourceValue ← getRVal source                      { init }
  res         ← newRVar (f sourceValue)
  linkRVal res source (const ∘ void ∘ setRVal res ∘ f){ connect }
  return res
```

The implementation of the counterpart to *bind* is now simply the combination of *rValFMap* and *joinRVal*.

```
rValBind :: RVar α → (α → RVar β) → IO (RVar β)
rValBind ra f = rValFMap f ra >>= joinRVal
```

The implementation of the counterpart to <*>, *rValStar*, is a bit more involved as there are two variables that can change. The function in *rValFMap* was constant so we could embed it in the update function. Here we have to maintain the current state ourselves. We create an *IORef* that contains a tuple with the function and the value. When either variable changes the store is updated and we update the value of the result variable by calling *setVal*.

```
rValStar :: RVar (α → β) → RVar α → IO (RVar β)
rValStar rf source = do
  f ← fmap (get rValue) (readIORef rf)
  val ← fmap (get rValue) (readIORef source)
  res ← newRVar (f val)
  store ← newIORef (f, val)
  let setVal :: IO ()
      setVal = do
        debugLn "setVal"
        f   ←        fmap fst (readIORef store)
        val ←        fmap snd (readIORef store)
        void $ setRVal res (f val)
  let onUpdateF f oF = do
        modifyIORef store (λ(_, val) → (f, val))
        setVal
  let onUpdateSource val oval = do
        modifyIORef store (λ(f, _) → (f, val))
        setVal
  linkRVal res rf        onUpdateF
```

```
    linkRVal res source onUpdateSource
    return res
```

We also create two versions that accept a function argument that runs in *IO* for the benefit of our interpreters. De difference is the monadic bind used in the local function *g* of *rValFMapIO* instead of normal function application. And again, the counterpart to bind is again the combination.

```
rValFMapIO :: (α → IO β) → RVar α → IO (RVar β)
    thatValue ← fmap (get rValue) (readIORef that)
    val ← f thatValue
    res ← newRVar val
    let g v _ = setRVal res =≪ f v
    linkRVal res that g
    return res
rValBindIO :: RVar α → (α → IO (RVar β)) → IO (RVar β)
rValBindIO ra f = rValFMapIO f ra ≫= joinRVal
```

A.3 Visualizing the reactive model

In order to visualize how the reactive model grows each function above is supplemented with a call to the visualization library, which we describe in this section.

Additionally each link created by the *linkRVar* function adds a finalizer to the link so that when it is collected by the garbage collector it calls a function in the visualizer.

The visualizer uses the `ubigraph` graph visualising tool(ubietylab.net, n.d.) and the `hubigraph` package(Ozaki, 2011). For each action: the creation of a variable or a link, the command is relayed to `ubigraph` which draws the new network on the screen as illustrated by Figure A.1. The binary of the server is found in the `bin` directory and the code should be compiled with the `ENABLE_VISUALIZATION` flag on. A video of a example run is available at `http://www.youtube.com/watch?v=EpI3l_2E_UI`.

- We kunnen het reactive model visualiseren. Daarnaast is het ook mogelijk om de assignments te zien propageren.
- Aangeven dat er een link is met Vacuum etc.
- Plaatje van de nodes

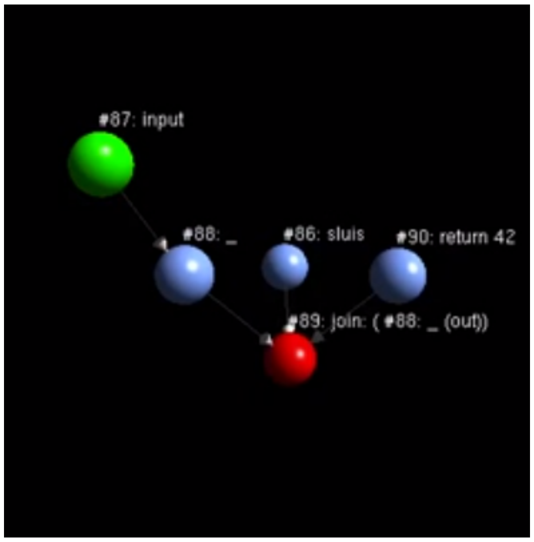


Figure A.1: Example visualization of running a test case.

Appendix B

LinRec implementation of *fib*

Source code listing for the linear recursion implementation of *fib*.

```
  -- | return scalar product of two vectors (given as lists)
(.*.) :: Num a => [a] -> [a] -> a
a .*. b = sum DOLLAR zipWith (*) a b

-- | apply the recurrence exactly once to a prefix, yielding the next prefix
--
-- @
-- step [1,1] [2,3]
-- @
--
-- results in [3,5]
step :: Num a => [a] -> [a] -> [a]
step rec pre = tail DOLLAR pre ++ [rec .*. pre]

-- | generate the sequence with prefix pre.
--
-- @
-- generate [1,1] [0,1]
-- @
--
-- results in the Fibonacci sequence [0,1,1,2,...]
generate :: Num a => [a] -> [a] -> [a]
generate rec pre = let res = pre ++ map (rec .*.) (tails res) in res

-- | Get the k-th element of the sequence with prefix pre.
-- @get@ obeys the law
```

```
--
-- @
-- get rec pre k == generate rec pre !! k
-- @
--
-- but takes O(n^2 * log(k)) @Num@ operations, where n is the order of
-- the recurrence relation, while @generate@ takes O(n*k) operations.
--
-- @
-- get [1,1] [0,1] 200000
-- @
--
-- calculates the 200,000th Fibonacci number reasonably quickly.
get :: Num a => [a] -> [a] -> Int -> a
get rec pre k = u0inv [] (u k) .*. pre where
   -- Let
   --       /  0   1  ...   0     \
   --   T = |  .   .        .     |
   --       |  0   0  ...   1     |
   --       \ b_0 b_1 ... b_{n-1} /
   --
   -- The code below is largely concerned with calculating T^k efficiently.
   --
   -- Let
   --   u_0 = ... = u_{k-2} = 0, u_{k-1} = 1,
   --   u_n+k = <defined by linear recurrence>
   --
   -- and define
   --         /   u_k     u_{k+1} ... u_{k+n-1}  \
   --   U_k = |    .         .            .      |
   --         \ u_{k+n-1} u_{k+n} ... u_{k+2n-2} /
   --
   -- Then, U_k = T^k U_0, and T^k = U_k U_0^{-1}.
   --
   -- Each U_k can be extrapolated easily from the first row by applying
   -- the linear recurrence, and U_0 was picked such that it’s unitarian,
   -- and thus can be inverted without using divisions.

   n = length rec

   -- u_0 is the first row of U_0, u_n the first row of U_n.
   u_0 = iterate (0:) [1] !! (n-1)
   u_n = take n DOLLAR drop n DOLLAR generate rec u_0
```

```
    -- u k calculates the first row of U_k,
    -- using U_{2k} = U_k U_0^{-1} U_k
    u 0 = u_0
    u k | odd k     = step rec DOLLAR u (k-1)
        | otherwise = let u' = u (k `div` 2)
                          u'' = u0inv [] u'
                          extrapolate xs = take n DOLLAR tails DOLLAR generate rec xs
                      in map (u'' .*.) DOLLAR extrapolate u'

    -- u0inv [] x calculates the first row of  X U_0^{-1}, where X is the
    -- matrix obtained by extrapolating the first row, x, by using the linear
    -- recurrence. It's a streamlined version of Gaussian elimination on U_0.
    u0inv acc []     = acc
    u0inv acc (x:xs) = u0inv (x:acc) DOLLAR zipWith (-) xs DOLLAR map (x*) u_n

fibLinRec = get [1,1] [0,1]
```

Appendix C

Acronyms and Definitions

AG Attribute Grammar (Reps et al., 1983)

CPS Continuation Passing Style

DDG Data-Dependency Graph

DSL Domain-Specific-Language

eDSL embedded DSL

GADT Generalised Abstract DataType

ML Standard ML (Milner et al., 1997)

MSS Maximum Segment Sum

Definitions

Anonymised datatype An anonymised datatype is a completely existantially qualified type. 38

Deep Embedding The term deep-embedding comes from the domain of Domain Specific Languages. Domain Specific Languages can be embedded into an host language, such as Haskell. We distinguish between two variants of embedding: shallow and deep embedding. In a shalllow embedding of a DSL the *semantics* of a language are directly encoded in the functions used to represent the language, allowing for at a single interpretation of the language. In a deep embedding the functions to represent the language construct an *inspectable* representation of the language. This allows for multiple interpretations of the language. For more information and examples we refer the reader to *The Difference Between Shallow and Deep Embedding* (Vermeulen, 2013). 22

incremental evaluation We define the evaluation of an expression as an incremental evaluation if the evaluation of said expression reuses the results from previously evaluated expressions for which it is known that re-evaluating them would not yield a different value. 3, 4, 61

Reactive model The reactive model, or model, is the run-time representation of an expression by variables and the dependencies between them. There are input variables containing input of the calculation, and output variables that contain the result. 6, 22, 39

reactive recursive datatypes Reactive recursive datatypes are datatypes that contain a reactive value in one or more fields of a constructor. 24

reactors Each variable x has a collection of variables for which their value depends on the value of variable x. 19, 80

Third-level Sharing We distinguish three levels of sharing:

1. Memoisation on the outer call
2. Memoisation on the outer call and contained calls
3. Memoisation between outer calls and contained calls

The last level of sharing is referred to as third-level sharing. 6, 16, 57, 58, 60, 71