

# Generic Program Transformation

Oege de Moor and Ganesh Sittampalam

Programming Research Group, Oxford University Computing Laboratory, Wolfson  
Building, Parks Road, OX1 3QD, United Kingdom

## 1 Introduction

When writing a program, especially in a high level language such as Haskell, the programmer is faced with a tension between abstraction and efficiency. A program that is easy to understand often fails to be efficient, while a more efficient solution often compromises clarity.

Fortunately Haskell permits us to reason about programs, so that we can start out with a program that is clear but inefficient, and transform it into a program that is efficient, but perhaps less readable. Indeed, such a *transformational* style of programming is as old as the subject of functional programming itself.

Programs developed in this style continue to suffer from a lack of readability, however: typically a functional programmer will develop his program on the back of an envelope, and only record the final result in his code. Of course he could document his ideas in comments, but as we all know, this is rarely done. Furthermore, when the programmer finds himself in a similar situation, using the same technique to develop a new piece of code, there is no way he can reuse the development recorded as a comment.

We claim that there is a handful of techniques that functional programmers routinely use to transform their programs, and that these techniques can themselves be coded as *meta programs*, allowing one to reuse the same optimisation technique on different pieces of code. In these lectures we shall explore this claim, and ways in which such meta programs might be implemented.

The structure of these notes is as follows. We first discuss three motivating examples, to clarify what type of optimisation we have in mind, and how an inefficient program might be annotated with transformations that effect the optimisation. Next, we discuss how the application of transformations can be mechanised. Our main design decision at this point is that transformations are never applied backwards. These ideas are then put to practice in a series of practical assignments, with a toy transformation system especially developed to accompany these notes. Finally, we discuss the matching problem in some detail, and explain how we have chosen to circumvent the undecidability inherent in matching of  $\lambda$ -terms.

Throughout, we shall take a cavalier attitude towards semantics. In particular, we have chosen to ignore all issues of strictness: some of our transformation rules ought to state side conditions about strictness. While it is admittedly incorrect to ignore such side conditions, they would clutter the presentation and detract from the main thesis of these lectures.

## 2 Abstraction versus efficiency

For concreteness, let us first examine a number of examples of the type of optimisation that we wish to capture, and the kind of programs on which they operate. This will give us a specific aim when developing the machinery for automating the process, and a yardstick for evaluating our results.

### 2.1 Minimum depth of a tree

Consider the data type of leaf labelled binary trees:

$$\text{data } Btree\ a = Leaf\ a \mid Bin\ (Btree\ a)\ (Btree\ a)$$

The minimum depth of such a tree is returned by the function  $mindepth :: Btree\ a \rightarrow Int$ :

$$\begin{aligned} mindepth\ (Leaf\ a) &= 0 \\ mindepth\ (Bin\ s\ t) &= \min(mindepth\ s)\ (mindepth\ t) + 1 \end{aligned}$$

This program is clear, but rather inefficient. It traverses the whole tree, regardless of leaves that may occur at a small depth. A better program would keep track of the ‘minimum depth so far’, and never explore subtrees beyond that current best solution. One possible implementation of that idea is

$$\begin{aligned} mindepth\ t &= md\ t\ 0\ \infty \\ md\ (Leaf\ a)\ d\ m &= \min\ d\ m \\ md\ (Bin\ s\ t)\ d\ m &= \text{if } d' \geq m \\ &\quad \text{then } m \\ &\quad \text{else } md\ s\ d'\ (md\ t\ d'\ m) \\ &\quad \text{where } d' = 1 + d \end{aligned}$$

The second parameter of  $md$  is called  $d$ , and it represents the current depth. The third parameter, called  $m$ , is the ‘minimum depth so far’. When computing the minimum depth of a leaf, we simply take the minimum of  $m$  and  $d$ . To compute the minimum depth of a composite tree, we first see whether there is any point in exploring beyond this node: if the new depth  $d' = 1 + d$  is greater or equal to  $m$ , we can cut the search. Otherwise, we first compute the minimum depth of the right subtree  $t$ , and using that as the new ‘minimum depth so far’ we explore the left subtree as well.

Our purpose in these lectures is to explore whether we could annotate the original, clear program for  $mindepth$  with the optimisations needed to obtain the second, efficient program. We could then leave the generation of the efficient code to a computer. Here, to obtain the efficient program, we need to know the definition of  $md$  in terms of the original function  $mindepth$ :

$$md\ t\ d\ m = \min(mindepth\ t + d)\ m$$

as well as some facts about minimum and addition: ( $a$ ,  $b$  and  $c$  are assumed to be natural numbers)

$$\begin{aligned}
 0 + a &= a \\
 (a + b) + c &= a + (b + c) \\
 \min(\min a b) c &= \min a (\min b c) \\
 \min a b + c &= \min(a + c)(b + c) \\
 \min(a + b) c &= \text{if } b \geq c \text{ then } c \text{ else } \min(a + b) c
 \end{aligned}$$

The full annotated program, as presented to a transformation system, might thus read as follows (the syntax is ad hoc, and won't be used in the practical exercises):

$$\begin{aligned}
 \text{mindepth}(\text{Leaf } a) &= 0 \\
 \text{mindepth}(\text{Bin } s t) &= \min(\text{mindepth } s)(\text{mindepth } t) + 1 \\
 \\
 \text{transform} \quad \text{mindepth } t &= \text{mdt } 0 \infty \\
 \text{where} \quad \text{mdt } d m &= \min(\text{mindepth } t + d) m \\
 \text{with} \quad 0 + a &= a \\
 &(a + b) + c = a + (b + c) \\
 &\min(\min a b) c = \min a (\min b c) \\
 &\min a b + c = \min(a + c)(b + c) \\
 &\min(a + b) c = \text{if } b \geq c \text{ then } c \text{ else } \min(a + b) c
 \end{aligned}$$

Would it not be much preferable to write the program in this form, and have the efficient program automatically generated? Obviously a system that allows one to do so would have to offer an inspection of the generated code, so that the programmer can be sure that he has indeed specified the efficient program he wished to write in the first place.

Some readers may object that we have left out an important transformation rule necessary to obtain the efficient program, namely some form of induction. As shown in the lectures by Backhouse and Jeuring at this summer school, such induction principles (and their associated program structures, known as *folds* or *catamorphisms*) can be deduced from the type definition of trees. For this reason, we do not need to state them in annotating a program for optimisation. Some practical restrictions apply, however, and we shall return to this point below.

## 2.2 Decorating a tree

We stay with leaf labelled binary trees, and consider their decoration with a list. That is, we wish to implement the function  $\text{decorate} :: \text{Btree } a \rightarrow [b] \rightarrow \text{Btree}(a, b)$  defined by

$$\begin{aligned}
 \text{decorate}(\text{Leaf } a) bs &= \text{Leaf}(a, \text{head } bs) \\
 \text{decorate}(\text{Bin } s t) bs &= \text{Bin}(\text{decorate } s bs)(\text{decorate } t(\text{drop}(\text{size } s) bs))
 \end{aligned}$$

Here the function *size s* returns the number of leaves in *s*, and *drop n bs* leaves off the first *n* elements of *bs*. Clearly the definition of *decorate t bs* makes sense only if *size t* ≤ *length bs*, but we shall not need that assumption below.

Consider a left skewed tree *t* of size *n*, where every right hand child of *Bin* is a leaf. To evaluate *decorate t bs*, we will have to compute each of *drop (n - 1) bs*, *drop (n - 2) bs*, and so on, until *drop 1 bs*. Because *drop i bs* takes about *i* steps to evaluate, it follows that the decoration takes  $\theta(n^2)$  time: the above program is clear, but of unacceptable inefficiency.

Here is the program that a skilled Haskell programmer would write:

```

decorate t bs = fst (dect bs)
dec (Leaf a) bs = (Leaf (a, head bs), drop 1 bs)
dec (Bin s t) bs = (Bin s' t', bs'')
                    where (s', bs') = dec s bs
                          (t', bs'') = dect bs'

```

The function *dect bs* decorates *t*, and also returns what is left over of *bs* after the decoration. When we decorate a leaf, we have to drop one element from the supply *bs* of decorations. To decorate a composite tree, we first decorate the left subtree, and then, with what is left over of the decorations (here called *bs'*) we decorate the right subtree. What is left over after decorating the right subtree (in the above program *bs''*), is the net result of the whole decoration. It is immediate from the above program that at each node of the tree, we spend only constant time, and therefore its time complexity is linear in the size of the tree.

How is the efficient program obtained from the first definition of *decorate*? What is the reasoning that the skilled Haskell programmer implicitly applied in writing *dec*? First of all, the original function *decorate* is related to *dec* by:

$$dect\ bs = (decorate\ t\ bs, drop\ (size\ t)\ bs)$$

Furthermore, the function *drop* satisfies the identity

$$drop\ (n + m)\ x = drop\ m\ (drop\ n\ x)$$

No further facts are needed. It does of course take a bit of a specialist to know that only these rules are needed to make the program efficient: but it is such knowledge that turns programming from a craft (where the rules are acquired by each programmer through experience) into a science (where the rules are made explicit, and communicated between programmers).

### 2.3 Partitioning a list

Finally, we consider a rather mundane example, which is representative of a large class of functions that occur frequently. In the standard *List* library of Haskell, there exists a function called *partition* :: (*a* → *Bool*) → [*a*] → ([*a*], [*a*]). It takes a predicate *p* and a list *x*, and it splits the list *x* into two subsequences: one

that contains those elements of  $x$  that do satisfy the predicate  $p$ , and the other containing those elements of  $x$  that fail to satisfy  $p$ . A neat program for *partition* might read

$$\text{partition } px = (\text{filter } px, \text{filter } (\text{not} \cdot p) x)$$

The function *filter*  $qx$  retains those elements of  $x$  that satisfy the predicate  $q$ . Using *filter* to code *partition* is pretty, but it requires two passes over the list  $x$ , and for each element of  $x$ , the predicate  $p$  will be evaluated twice.

A more efficient program (and indeed the definition found in the library) is

$$\text{partition } p [] = ([], [])$$

$$\text{partition } p (a : x) = \text{if } p a \text{ then } (a : y, z) \text{ else } (y, a : z)$$

$$\text{where } (y, z) = \text{partition } px$$

The only way to split the empty list is into two copies of itself. Furthermore, if we have partitioned  $x$  into  $(y, z)$ , and  $a$  satisfies the predicate  $p$ , we append it to  $y$ . If  $a$  fails to satisfy  $p$  we append it to  $z$ .

To develop the more efficient program, we need to know the following facts about *if-then-else*:

$$\text{if } \text{not } c \text{ then } e_1 \text{ else } e_2 = \text{if } c \text{ then } e_2 \text{ else } e_1$$

$$\text{if } c \text{ then } (\text{if } c \text{ then } e_1 \text{ else } e_2) \text{ else } e_3 = \text{if } c \text{ then } e_1 \text{ else } e_3$$

$$\text{if } c \text{ then } e_1 \text{ else } (\text{if } c \text{ then } e_2 \text{ else } e_3) = \text{if } c \text{ then } e_1 \text{ else } e_3$$

$$f (\text{if } c \text{ then } e_1 \text{ else } e_2) = \text{if } c \text{ then } f e_1 \text{ else } f e_2$$

Of course these rules are not particular to the problem in hand. In fact, they are themselves instances of more general transformations involving *case* expressions [19].

## Exercises

**2.1** Can the computation of the *maximum* depth of a tree be made more efficient in the same way as the computation of the minimum depth? If not, which property of *min* fails for *max*?

**2.2** A leaf labelled binary tree is said to be *perfectly balanced* if for every subtree  $t$ , the size of the left hand child is precisely *size*  $t$  ‘*div*’ 2. Given a list  $x$ , the function *build*  $x$  produces a perfectly balanced tree whose inorder traversal is  $x$ . First give a naive program for computing *build*, and then show how it can be made efficient.

**2.3** Are there circumstances where the original definition of *partition* is as efficient as the ‘improved’ version?

## 3 Automating the transition: fusion and higher order rewriting

The time has come to be more precise about the mechanical application of transformation rules. The main principle in designing a transformation system

is that its operation should be transparent: the programmer must be able to predict its results without running experiments. For that reason, we reject any form of artificial intelligence, and also sophisticated procedures from automated theorem proving. Approaches based on such sophisticated techniques have been tried, and shown to fail. Few transformation systems have been used outside the laboratory of their creators, and we believe this is due to the lack of transparency in their operation.

The most obvious strategy is therefore a simple process of rewriting, where the rules are applied from left to right. That still leaves some freedom in specifying the order in which rules are tried, and on which subexpressions. It does seem rather restrictive, however, for it does not allow for the application of rules in reverse direction. In particular, program development by means of the well known unfold/fold strategy [7] is not applicable: we can use a function definition in its evaluating direction, but never backwards, matching the right hand side and producing an instance of the left hand side.

Instead, all inductive arguments must be carried out through so-called *fusion* rules, which encapsulate the induction in a higher order function called a *fold* (or, by residents of the Netherlands, a *catamorphism*). There is precisely one *fold* and an accompanying fusion rule for each data type.

To illustrate, consider the data type of lists. Its *fold* operator *foldr* has type  $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$ , and is defined by

$$\begin{aligned} \textit{foldr step } e [] &= e \\ \textit{foldr step } e (a : x) &= \textit{step } a (\textit{foldr step } e x) \end{aligned}$$

If we write  $a \oplus y$  for  $\textit{step } a y$ , we can visualise the computation of *foldr* by

$$\textit{foldr } (\oplus) e (a_0 : (a_1 : (a_2 : \dots (a_n : [])))) = a_0 \oplus (a_1 \oplus (a_2 \oplus \dots (a_n \oplus e)))$$

From this equation, it is apparent that *foldr step e* replaces the cons operator ( $:$ ) by *step*, and the empty list by *e*. This is the general pattern of a *fold*: it replaces the constructor functions of a given type by new functions.

Many functions on lists can be expressed as instances of *foldr*. For example,  $\textit{concat} = \textit{foldr } (++) []$  is the function that concatenates the components of a list whose elements are lists themselves, and

$$\begin{aligned} \textit{filter } p &= \textit{foldr step } [] \\ \textit{where } \textit{step } a y &= \textit{if } p a \textit{ then } a : y \textit{ else } y \end{aligned}$$

The advantage of expressing a function through *foldr* is that the definition becomes nonrecursive, and therefore much easier to manipulate. In particular, we get a simple way of doing induction that only requires us to apply rewrite rules in one direction.

Suppose that we apply a function *f* to the result of *foldr*:

$$f(a_0 \oplus (a_1 \oplus (a_2 \oplus \dots (a_n \oplus e))))$$

In a sense this is a two pass computation: first we compute the sum of the  $a_i$ , and then we apply *f* to that sum. If we know that  $f(a \oplus z) = a \otimes f z$ , (for all  $a$

and  $z$ ) we can distribute  $f$  through the sum, to get a single pass computation:

$$\begin{aligned} f(a_0 \oplus (a_1 \oplus (a_2 \oplus \dots (a_n \oplus e)))) &= \\ a_0 \otimes f(a_1 \oplus (a_2 \oplus \dots (a_n \oplus e))) &= \\ a_0 \otimes (a_1 \otimes f(a_2 \oplus \dots (a_n \oplus e))) &= \\ &\dots \\ a_0 \otimes (a_1 \otimes (a_2 \otimes \dots (a_n \otimes f e))) & \end{aligned}$$

That last line is of course nothing but an application of  $foldr(\otimes)(f e)$ . This principle can be formally stated as the *fusion* rule:

$$f(foldr(\oplus)ex) = foldr(\otimes)cx, \text{ if } fe = c \quad \text{and} \\ \lambda az \rightarrow f(a \oplus z) = \lambda az \rightarrow a \otimes fz$$

As remarked above, the distributivity condition should hold for *all*  $a$  and  $x$ : here we have encoded that by abstracting over these variables, and requiring the two functions to be equal.

As a simple example of the fusion rule, consider  $sum(foldr(+))[]x$ . Because  $sum[] = 0$  and  $sum(a + z) = sum a + sum z$ , we have

$$sum(foldr(+))[]x = foldr(\lambda ay \rightarrow sum a + y)0x$$

In the mechanical application of the fusion rule, we first match the pattern  $f(foldr(\oplus)ex)$  with  $sum(foldr(+))[]x$ . That match succeeds, with  $f = sum$ ,  $(\oplus) = (+)$ , and  $e = []$ . Next, we attempt to prove the provisos, starting with the second equation. We rewrite its left hand side until no more rules apply:

$$\lambda az \rightarrow sum(a + z) = \lambda az \rightarrow sum a + sum z$$

The result of the rewriting process (here  $\lambda az \rightarrow sum a + sum z$ ) is then matched with the pattern  $\lambda az \rightarrow a \otimes sum z$  in the right hand side of the condition. That match yields a definition for the operator  $(\otimes)$ , namely  $a \otimes y = sum a + y$ . This completes the verification of the distributivity condition, and next we prove  $sum[] = 0$ : this only involves rewriting.

It is worthwhile to reflect on the mechanical application of fusion before proceeding further. Most of the work goes into the matching process: in particular, the definition of  $(\otimes)$  is ‘invented’ by matching the right hand side of the distributivity condition against a fully rewritten version of the left hand side. A matching algorithm that performs the synthesis of function definitions is said to be a *higher order* matching algorithm. The observation that higher order matching should be a key ingredient of any program transformation tool was first made by Huet and Lang [13].

While fusion is elegant and amenable to mechanisation, it may appear that it only applies to programs that were written in terms of *foldr* to start with. What if we wish to improve a program that uses explicit recursion? The answer is that the identity function is itself an instance of *foldr*:

$$id = foldr(\cdot)[]$$

Therefore, if we wish to write a function  $f$  in terms of  $foldr$ , we just apply fusion to the expression

$$\lambda x \rightarrow f(foldr\ (\cdot)\ []\ x)$$

The reader may care to work out for herself how this can be used to write  $sum$  (the function that returns the sum of a list) as an instance of  $foldr$ .

Our insistence on the principle that rules are never applied backwards, combined with a matching procedure that is unable to synthesise function definitions that involve pattern matching, sometimes forces us to use multiple variants of the fusion rule. When doing program derivation by hand, a single rule suffices, but in mechanised program synthesis, multiple variants are required. Fortunately, these variants are again dictated by type considerations, just as the fusion rule itself can be deduced from the type to which it applies.

To appreciate why multiple variants of fusion are needed, recall the function  $partition$ , defined by

$$partition\ p\ x = (filter\ p\ x, filter\ (not\ \cdot\ p)\ x)$$

Now let us try and write  $partition\ p$  as an instance of  $foldr$ , by applying fusion to

$$partition\ p\ (foldr\ (\cdot)\ []\ x)$$

We start by rewriting

$$\begin{aligned} & partition\ p\ (a : x) \\ = & \{definition\ of\ partition\} \\ & (filter\ p\ (a : x), filter\ (not\ \cdot\ p)\ (a : x)) \\ = & \{definition\ of\ filter\ (twice)\} \\ & (if\ p\ a\ then\ a : filter\ p\ x\ else\ filter\ p\ x, \\ & \quad if\ not\ (p\ a)\ then\ a : filter\ (not\ \cdot\ p)\ x\ else\ filter\ (not\ \cdot\ p)\ x) \\ = & \{various\ facts\ about\ if\} \\ & if\ p\ a\ then\ (a : filter\ p\ x, filter\ (not\ \cdot\ p)\ x) \\ & \quad else\ (filter\ p\ x, a : filter\ (not\ \cdot\ p)\ x) \end{aligned}$$

Our procedure for the mechanical application of fusion requires that we match the expression derived above,

$$\lambda a\ x \rightarrow if\ p\ a\ then\ (a : filter\ p\ x, filter\ (not\ \cdot\ p)\ x) \\ \quad else\ (filter\ p\ x, a : filter\ (not\ \cdot\ p)\ x)$$

against the pattern

$$\lambda a\ x \rightarrow a \otimes partition\ p\ x$$

This clearly fails, because we do not rewrite the right hand side of the pattern before matching, as that would go against our design principle that no rule is applied backwards. Nevertheless, let us assume for the moment that the pattern

is rewritten, which results in the definition of *partition* to be expanded. We then have to find  $(\otimes)$  so that

$$\begin{aligned} \lambda a x \rightarrow & \text{if } p a \text{ then } (a : \text{filter } p x, \text{filter } (\text{not} \cdot p) x) \\ & \text{else } (\text{filter } p x, a : \text{filter } (\text{not} \cdot p) x) \\ = & \lambda a x \rightarrow a \otimes (\text{filter } p x, \text{filter } (\text{not} \cdot p) x) \end{aligned}$$

The only solution is

$$a \otimes (y, z) = \text{if } p a \text{ then } (a : y, z) \text{ else } (y, a : z)$$

Now observe that this requires  $(\otimes)$  to perform pattern matching on its right hand argument. To make the matching algorithm synthesise such definitions that involve pattern bindings is rather complicated: to be consistent, it would have to cope not only with tuples, but also with types that have multiple constructor functions.

The conclusion is that to make the above example work with standard fusion, we would have to compromise our guiding design principle (no backward steps), as well as the matching algorithm. Clearly it is preferable to have a specialised fusion rule for dealing with examples such as *partition*. Define the function *split* ::  $(a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)$  by

$$\text{split } f g a = (f a, g a)$$

This higher order operator captures the idea of two separate functions applied to the same argument. For example, we have

$$\text{partition } p = \text{split } (\text{filter } p) (\text{filter } (\text{not} \cdot p))$$

We can now formulate a specialised fusion rule that applies to instances of *split*:

$$\text{split } f g (\text{foldr } \text{step } e x) = \text{foldr } (\lambda a \rightarrow \text{uncurry } (h a)) c x$$

provided

$$\begin{aligned} & \text{split } f g e = c, \quad \text{and} \\ & \lambda a y \rightarrow \text{split } f g (\text{step } a y) = \lambda a y \rightarrow h a (f y) (g y) \end{aligned}$$

The function *uncurry* is defined by

$$\text{uncurry } k(a, b) = k a b$$

It explicitly introduces the pattern binding that our matching algorithm cannot synthesise.

Cognoscenti will recognise the above variant of fusion as the *tupling* principle, a well studied technique in transformational program development. You will have a chance to explore its applications in the next section, which consists entirely of practical assignments.

## Exercises

**3.1** Define the *fold* operator on natural numbers. Its type is

$$\text{foldn} :: (a \rightarrow a) \rightarrow a \rightarrow \text{Int} \rightarrow a$$

You could think of the first argument as replacing the successor function (+1) and the second argument as replacing 0. If there were a data type *Nat* of natural numbers in Haskell, the type of the third argument would be *Nat* and not *Int*. What is the fusion rule for *foldn*?

**3.2** Define the *fold* operator on leaf labelled binary trees. Its type is

$$\text{foldbtree} :: (a \rightarrow a \rightarrow a) \rightarrow (b \rightarrow a) \rightarrow \text{Btree } b \rightarrow a$$

What is its fusion rule?

## 4 The MAG system

Before we proceed to explore the intricacies of pattern matching with function variables, it is important to gain some experience with the concepts introduced so far. We have produced a little system for this purpose, called MAG, after the place where it was written, Magdalen College, Oxford. The name also reminds us of its overriding design principle, famously coined by a former British prime minister, Margaret Thatcher: *the lady is not for turning!* In MAG, transformation rules are applied in one direction only.

MAG is not a serious research tool, it is just a little program rather hastily written for these lectures. It is also an experiment in the use of the pretty printing, parsing and attribute grammar libraries provided by Swierstra and his colleagues at Utrecht University [20]: by using their libraries, we aimed to produce a system that is truly light weight and easy to modify. Finally, we took inspiration from an unpublished term rewriting system by Mike Spivey. An elegant exposition of Spivey's program, with a number of important improvements, can be found in [5]. Bird also shows how Spivey's program can be used as a tool in program verification. A much more advanced tool for transforming Haskell programs is the HYLO system [16]. Other systems that perform the type of transformation considered here are described in [2, 14].

### 4.1 Getting acquainted

MAG takes two kinds of input file: programs and theories. A *program* is just a program in a functional programming language that is a small subset of Haskell. A *theory* is a set of conditional equations that are applied in the rewriting process. The program file merely exists to infer the types of all the constants in a theory. If you wish the definitions to be used as rewrite rules, they have to be repeated in the theory.

**Getting started** If you use MAG as a precompiled binary, simply invoke *mag*. If you use MAG from Hugs (this may be slow!) start up Hugs with a heap of at least 1M, load the file *Main.hs*, and evaluate the function *main*. The system will respond with the message:

MAG calculator

Type program name <filename>.p:

Respond by typing *program*, followed by a carriage return. This loads a program from the file *program.p*. The system will now ask you to specify a theory:

Type theory name <filename>.eq:

Respond by typing *sumsq*, followed by a carriage return. This loads a set of equations from the file *sumsq.eq*. The system is ready to transform an expression, and it prompts:

Type expression:

Type *sumsq*, followed by a carriage return. In response, the system applies rewrite rules from *sumsq.eq* to *sumsq*, until no more apply. It will show the result of each rewrite step, as follows:

```
sumsq
= { sumsq }
  sum . map sq
= { compose }
  \a -> sum (map sq a)
= { sum }
  \a -> foldr (+) 0 (map sq a)
= { map }
  \a -> foldr (+) 0 (foldr (:) . sq) [] a)
= { compose }
  \a -> foldr (+) 0 (foldr (\d -> (:) (sq d)) [] a)
= { fusion

      foldr (+) 0 []
    = { foldr0 }
      0

      \b c -> foldr (+) 0 (sq b : c)
    = { foldr1 }
      \b c -> sq b + foldr (+) 0 c
  }
  foldr (\b -> (+) (sq b)) 0
```

There are a couple of things worth noting here. First of all, in order to apply the definition of function composition, the system introduces a new argument called *a*. Furthermore, when *fusion* is applied, it does two nested calculations,

one for each of the applicability conditions of fusion. In the resulting expression, the argument `a` is no longer needed, and therefore it is not displayed.

After the calculation is completed, MAG asks you again for a theory file, and then an expression. When you enter an empty name for the theory file (just a carriage return) it stops executing. It will also halt when certain errors occur; we discuss error reporting in more detail below.

**Program files** Have a look at the contents of `program.p`. It contains definitions of all functions that you need to carry out the exercises below: it follows that in principle, all can be done in a single session with MAG, only giving `program.p` as the program file. It should not be necessary to modify `program.p`. A typical fragment of `program.p` is

```
{
map f [] = [];
map f (a : x) = f a : map f x
}

{
foldr f e [] = e;
foldr f e (x : xs) = f x (foldr f e xs)
}

{
data Tree a = Node a [Tree a]
}

{
foldtree f step e (Node a ts) = f a (foldr (step . r) e ts)
      where r = foldtree f step e
}
```

As you can see, it looks a lot like Haskell, but each definition or family of mutually recursive definitions have to be grouped together in curly braces. Also, there is no offside rule, so all definitions within such a group have to be separated by semicolons. Finally, you can only use functions that have been defined earlier in the file: it would be incorrect to define `foldr` after `foldtree`, for example. These restrictions are necessary because MAG does not perform dependency analysis on program files.

**Theory files** A theory file contains a set of conditional rewrite rules. If you wish to use definitions from a program file as rewrite rules, you have to repeat them in the theory file. The syntax for rewrite rules is fairly obvious, except that local definitions (*let* and *where*) are not allowed: the matching algorithm does not know how to cope with them.

As an example, here are the contents of `sumsq.eq`:

```

sumsq: sumsq = sum . map sq;

compose: (f . g) x = f (g x);

sum: sum = foldr (+) 0;

map: map f = foldr ((:).f) [];

foldr0: foldr step e [] = e;
foldr1: foldr step e (a:x) = step a (foldr step e x);

fusion: f (foldr step e x) = foldr g c x,
        if {f e = c;
            \ a y -> f (step a y) = \ a y -> g a (f y)}

```

Each rule starts with an identifier that is the name of the rule, followed by a colon, followed by an equation. Equations are separated by semicolons. When an equation has side conditions (in the above example, only `fusion` has side conditions), they are introduced by a comma, followed by `if`, followed by the conditions in braces, separated by semicolons. For reasons of efficiency, it is advisable to put the conditions in order of increasing complexity, so that the most stringent condition (the last in the list) is tried first. Equations should never introduce variables on the right hand side of equations that do not occur on the left.

**Expressions for transformation** Expressions to be transformed should not contain any local definitions, and no free variables at all. If you want to transform an expression with variables, bind them with  $\lambda$ . There is no way of transforming local definitions, except by writing them as  $\lambda$ -abstractions.

**Error reporting** Error reporting from Swierstra's parsing combinators is excellent, so you should get fairly comprehensible messages about syntax errors: these occur when the files are read in. Due to lazy evaluation, there is no telling when a type inference error might occur, and the messages do not give any indication which input line is at fault. Should this occur, eyeball your code, and if there is no obvious mistake, call one of the authors for help.

## 4.2 Accumulation parameters

The first set of practical exercises is about *accumulation parameters*. This is a technique for improving functional programs where the original, inefficient definition is generalised by introducing an extra parameter. The parameter is used to *accumulate* additional information during a computation — hence the name. The technique was studied in [4]; see also [11].

**Fast reverse** A naive definition of the function *reverse* is

$$\begin{aligned} \text{reverse} [] &= [] \\ \text{reverse} (a : x) &= \text{reverse } x \text{ ++ } [a] \end{aligned}$$

Write down the recursive definition of (++) , and estimate how much time it takes to evaluate *reverse x* when *x* has length *n*.

The time complexity of *reverse* can be improved by defining:

$$\text{fastrev } x \ y = \text{reverse } x \text{ ++ } y$$

Why is this a generalisation of *reverse*? Using the fact that (++) is associative

$$(x \text{ ++ } y) \text{ ++ } z = x \text{ ++ } (y \text{ ++ } z)$$

one can synthesise an efficient program for *fastrev*.

The above definitions, and the associativity of (++) , have been recorded in the theory file `reverse.eq`. There is one peculiarity, however: the definition of *fastrev* reads

$$\text{fastrev } x \ y = \text{reverse} (\text{foldr } (:) [] x) \text{ ++ } y$$

Why is the instance of *foldr* there? Confirm your answer by loading the theory file `reverse.eq` in MAG, and transforming *fastrev*. Estimate the time complexity of the resulting program.

**Postorder traversal** The example of fast reverse is in fact representative of a much larger class of programs, where the concatenation operator is eliminated by introducing an extra parameter, and exploiting the fact that concatenation (++) is associative.

Consider the data type of rose trees, defined by

$$\text{data Tree } a = \text{Node } a [\text{Tree } a]$$

The fold operator on this type of tree is

$$\begin{aligned} \text{foldtree } f \ \text{step } e (\text{Node } a \ ts) &= f \ a (\text{foldr } (\text{step} \cdot r) \ e \ ts) \\ \text{where } r &= \text{foldtree } f \ \text{step } e \end{aligned}$$

In particular, the identity function on rose trees is given by *foldtree Node* (:) [] .

The *postorder traversal* of a rose tree lists all the elements in the descendants of a node, followed by the label of that node itself:

$$\text{postorder} (\text{Node } a \ ts) = \text{concat} (\text{map } \text{postorder } ts) \text{ ++ } [a]$$

This definition is inefficient, because of the concatenation operator. It is your task to produce an efficient definition through the use of MAG.

The theory file `postorder.eq` contains the relevant fusion law, and the definitions of the functions involved. In analogy with the preceding exercise, add the definition of *fastpost* :: *Tree a* → [*a*] → [*a*] (that is *postorder* with an extra

parameter), and also the associativity of concatenation. Load the theory file, and if there are no error messages, transform *fastpost*.

Now consult Section 7.3 of Bird's text *Introduction to functional programming in Haskell*. (If you do not have a copy of this book, we warmly recommend you get one. It teaches you functional programming with *taste*.) In particular, read section 7.3.2. Congratulations! With the help of MAG, you have beaten an Oxford professor at his own game! Your program solves a slightly more complicated problem (postorder instead of preorder), is much simpler than his, and obtained in exactly the same way as fast reverse.

**Breadth first traversal** It would be wrong to suggest that the technique of accumulating parameters only applies to examples that involve simple concatenation. One can exploit accumulation parameters in almost every program where a tree is traversed using an associative operator.

Consider, for example, the problem of listing the elements of a rose tree in breadth first order. This can be achieved by listing the elements level by level, and then concatenating the result:

$$\text{breadthfirst} = \text{concat} \cdot \text{levels}$$

The function  $\text{levels} :: \text{Tree } a \rightarrow [[a]]$  first gives the elements that occur at depth 0, then the elements at depth 1, depth 2 and so on. It is defined by

$$\text{levels} (\text{Node } a \text{ } ts) = [a] : \text{glues} (\text{map levels } ts)$$

$$\begin{aligned} \text{glues } [] &= [] \\ \text{glues } (x : xs) &= \text{lzc } x (\text{glues } xs) \end{aligned}$$

$$\begin{aligned} \text{lzc } [] x &= x \\ \text{lzc } (a : x) y &= \text{if } \text{null } y \\ &\quad \text{then } a : \text{lzc } x [] \\ &\quad \text{else } (a \text{ ++ } \text{head } y) : \text{lzc } x (\text{tail } y) \end{aligned}$$

Here *lzc* stands for *long zip with concatenation*. What is the difference between *lzc* and *zipWith*(++)? Is *lzc* associative? What is the time complexity of *levels*?

You know what has to be done: all the definitions can be found in the theory file `levels.eq`. Produce an efficient program with MAG. This program and its derivation were first discovered and presented by [10], and Jeremy Gibbons suggested it as an exercise in the use of MAG. His paper also presents another, arguably more elegant, method of computing the breadth first traversal.

**Minimum depth** Let us now return to the problem discussed at the beginning of these lectures, namely computing the minimum depth of a leaf labelled binary tree. It was claimed there that apart from the definitions, all we need to produce

an efficient program is the following set of rules:

$$\begin{aligned}0 + a &= a \\(a + b) + c &= a + (b + c) \\ \min(\min a b) c &= \min a (\min b c) \\ \min a b + c &= \min (a + c)(b + c) \\ \min (a + b) c &= \text{if } b \geq c \text{ then } c \text{ else } \min (a + b) c\end{aligned}$$

Unfortunately, there is a problem when this set of equations is implemented by rewriting. Why?

In this particular example, the most elegant solution might be to single out rules that should be applied at most once. However, we wish to avoid any ad hoc features in the code of MAG itself, so instead we modify the rules. One can take the last two rules together in a single, composite rule:

$$\begin{aligned}\min (\min a b + c) d &= \text{if } c \geq d \\ &\quad \text{then } d \\ &\quad \text{else } \min (\min (a + c)(b + c)) d\end{aligned}$$

Even with this fix, however, we cannot directly generate an efficient program. The *mindepth* example is different from those that went before in that there are *two* accumulation parameters. Due to certain limitations in our matching algorithm (which will be discussed in depth later), we have to slightly adapt the fusion rule to cope with the two parameters at once.

These modifications have been installed for you in the file `mindepth.eq`, and transforming the expression `md` will produce the efficient program we discussed above. Because of the large number of free variables in the fusion rule, its generation can take a while: on a Pentium 90 running Linux and Hugs 1.4, it took almost 28 minutes. Which variables in the fusion rule could be bound to improve efficiency of the transformation process? Make that change in `mindepth.eq`, and generate an efficient program for *mindepth*.

### 4.3 Tupling

The next set of exercises is about tupling: improving the efficiency of a program by computing several values at once. Tupling is in a sense the dual of accumulation parameters: in the first case, we generalise the *range* of a function, and in the second case the *domain*. There is quite a large body of literature on tupling, e.g. [3, 6, 8, 12, 17]. In the exercises below, you will have the opportunity to explore its versatility: many programs can be improved through tupling. One could even say that the attribute grammar system presented by Swierstra at this summer school is a specialised tool for tupling.

**Fibonacci** The standard example of tupling, found in any introductory text on programming, is the Fibonacci function. This function, which is amazingly

ubiquitous in computer science, is defined by

$$\begin{aligned}fib0 &= 0 \\fib1 &= 1 \\fib(n + 2) &= fib(n + 1) + fib n\end{aligned}$$

This program is inefficient because there are many duplicated calls to *fib*.

To improve the efficiency, we compute *fib n* simultaneously with *fib(n + 1)*. That is, we aim to apply the tupling transformation to

$$fastfib = split fib(\lambda n \rightarrow fib(n + 1))$$

Of course one also needs some arithmetic to do that transformation. Which two facts about addition are needed? Install these equations in the file `fib.eq`, and generate the efficient program.

As an aside, we remark that there exists a much better program still, which exploits the identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} fib(n + 1) \\ fib n \end{pmatrix}$$

together with the fact that exponentiation can be performed in logarithmic time.

**Steep sequences** A list of numbers is said to be *steep* if each element is greater than the sum of the elements that follow it. We might define the predicate *steep* :: [Int] → Bool by

$$\begin{aligned}steep[] &= True \\steep(a : x) &= a > sum x \wedge steep x\end{aligned}$$

Estimate the time complexity of *steep*.

How would you improve the efficiency of *steep*? The file `steep.eq` contains the rules you need to apply tupling. Add a definition of

$$faststeep :: [Int] \rightarrow (Bool, Int)$$

note the type carefully!) and generate an efficient program for *faststeep*. In this case you need no rules besides the definitions.

**Partition** Need we say more? Just do it, and imagine how tedious and error prone it would be to do a derivation by hand! All you need is in `partition.eq`. In this example, the ordering of the rules in the theory file is rather important: in the rewriting process, they are tried in the order they occur. Why does the rule about distributing a function over a conditional come last?

**Decorate** Finally, recall the problem of decorating a leaf labelled binary tree. The original, inefficient program is

$$\begin{aligned} \text{decorate}(\text{Leaf } a) \text{ } bs &= \text{Leaf}(a, \text{head } bs) \\ \text{decorate}(\text{Bin } s \ t) \text{ } bs &= \text{Bin}(\text{decorate } s \ bs) (\text{decorate } t \ (\text{drop}(\text{size } s) \ bs)) \end{aligned}$$

We aim to improve *decorate* by defining a new function

$$\text{dect } bs = (\text{decorate } t \ bs, \text{drop}(\text{size } t) \ bs)$$

This does not conform exactly to the pattern in our previous examples of tupling, because *dec* is not an instance of *split*. We do have

$$\text{dect} = \text{split}(\text{decorate } t) (\text{drop} \cdot \text{size } t)$$

but this does not make the repeated use of *t* explicit — and that use must be explicit to obtain a single pass algorithm. Instead, what we need here is

$$\text{split2 } f \ g \ a \ b = (f \ a \ b, g \ a \ b)$$

We then have

$$\text{dec} = \text{split2 } \text{decorate} (\lambda t \ bs \rightarrow \text{drop}(\text{size } t) \ bs)$$

Furthermore, associated with *split2*, one obtains the *double tupling* rule

$$\begin{aligned} &\text{split2 } f \ g \ (\text{foldbtree } \text{join } \text{start } x) \\ = & \\ &\text{foldbtree}(\text{uncurry2 } h) \ k \ x, \\ &\text{if } \lambda a \rightarrow \text{split2 } f \ g \ (\text{start } a) = \lambda a \rightarrow k \ a \\ &\lambda x \ y \rightarrow \text{split2 } f \ g \ (\text{join } x \ y) = \lambda x \ y \rightarrow h \ (f \ x) \ (g \ x) \ (f \ y) \ (g \ y) \end{aligned}$$

The function *uncurry2* in the conclusion of this rule is defined by

$$\text{uncurry2 } e \ f \ g \ h \ k = e \ (fst \cdot fg) \ (snd \cdot fg) \ (fst \cdot hk) \ (snd \cdot hk)$$

While this may all seem a bit complicated, once *split2* is defined, the rest follows purely from type considerations.

Returning to the problem of optimising *decorate*, the above machinery can be found in the file `decorate.eq`. Run MAG on *dec*. Is the result identical to the program discussed earlier in the lectures? What further transformation needs to be performed?

#### 4.4 Carrying on

In the above examples, we have illustrated the use of MAG in only two kinds of optimisation, namely accumulation parameters and tupling. There is no reason why MAG could not apply more sophisticated transformations. Indeed, one exercise that any transformation tool must tackle is the derivation of the so-called *maximum segment sum* problem. The file `mss.eq` shows how that is done. You now have sufficient knowledge to attempt such exercises yourself. The authors would be pleased to hear of your experience.

## 5 Matching typed $\lambda$ -expressions

The choice of matching algorithm (for matching the left hand side of rewrite rules to expressions) determines the scope of applicability of transformation rules such as fusion. If we had been able to implement an ‘ideal’ matching algorithm, all programming exercises in these notes would have been applications of a single formulation of fusion. Put another way, it is the matching algorithm which determines how *generic* our program transformations are.

In the light of this observation, it is perhaps unsurprising that no such ideal algorithm exists: in its full generality, the matching problem is not computable. To cope with that incomputability, we have to somehow restrict the matching problem.

In the context of program transformation, the most popular restriction is to *second-order* matching: roughly speaking, this restricts pattern variables to be of base type (such as *Int*, *Bool* or  $[Int]$ ), or functions between base types. For example, the KORSO transformation system, developed at Bremen, uses second-order matching [9].

Unfortunately the restriction to second-order variables is not a natural one when transforming Haskell programs. Consider, for example, our synthesis of a fast program for *reverse*: this would be beyond the scope of a second-order matching algorithm. We therefore investigate a different restriction of the matching problem, which seems better suited to the applications we have in mind.

We start by reviewing matching of types, both because this will refresh the elementary definitions in the reader’s mind, and because we shall need these operations on types later on. We then turn to the definitions on expressions that include  $\lambda$ -bound variables. After these elementary definitions, we introduce substitutions and some related operations. Finally, we discuss a number of variants of the matching problem. We conclude with a rigorous specification of our restricted matching problem, which pins down the algorithm that is implemented in MAG.

This section admittedly requires a little bit more background knowledge than the preceding material: a good introduction to the concepts used here can be found in [18].

### 5.1 Types

Expressions are typed in the Hindley-Milner discipline, without any modern extensions such as type classes [15]. More precisely, we shall be working with *type schemes*, as defined by the declaration

$$\text{data Type} = \text{Tid Tname} \mid \text{Tcon Cname [Type]} \quad \text{deriving Eq}$$

That is, a type scheme is a variable (a type identifier), or a constructor applied to a list of argument types. Equality on types is straightforward structural equality.

*Substitution.* A *type substitution* is a mapping from type variables to types:

$$\text{type } Tsub = Tname \rightarrow Type$$

The function  $tapply :: Tsub \rightarrow Type \rightarrow Type$  applies a substitution to a type in the obvious way.

*Generality.* We say that  $\tau$  is *less general* than  $\sigma$  if there exists another substitution  $\phi$  so that

$$tapply \tau = tapply \phi \cdot tapply \sigma$$

*Matching.* Consider the problem of matching one type against another. Given types  $p$  and  $t$  a *match* is a substitution  $\tau$  such that

$$tapply \tau p = t$$

If there exists a match at all, there exists a most general match. The function  $tmatch p t$  returns a singleton list  $[\sigma]$  of that most general match  $\sigma$ , or (if no matches exist) it returns the empty list. Here, *most general* means that  $\sigma$  leaves any variable not occurring in  $p$  unchanged.

## 5.2 Expressions

An expression is a constant, a variable, an application, or a  $\lambda$ -abstraction. Furthermore, each expression is decorated with its type:

$$\begin{array}{l} \text{data } Exp = \text{Con Constant Type} \mid \\ \quad \text{Var Name Type} \mid \\ \quad \text{Ap Exp Exp Type} \mid \\ \quad \text{Lam Name Exp Type} \end{array}$$

This representation is highly redundant, because much of the type information is duplicated. In our experience, however, this redundancy is worth the overhead when programming a system such as MAG: in any form of meta programming, it is very easy to produce terms that are ill-typed. Carrying around the types everywhere makes it easier to track such errors down. In the examples below, we shall often leave the type information implicit to improve readability.

*$\alpha$ -conversion* In  $\lambda$ -expressions, the names of bound variables do not matter. We have, for example, the identity

$$\lambda x \rightarrow x + z = \lambda y \rightarrow y + z$$

Note, however, that

$$\lambda x \rightarrow x + z \neq \lambda z \rightarrow z + z$$

because  $z$  is a free variable on the left hand side. This principle, that bound variables can be renamed to any identifier that is not free in the body of a  $\lambda$ -abstraction, is called  *$\alpha$ -conversion*. The function

$$alphaconvertible :: Exp \rightarrow Exp \rightarrow Bool$$

tests whether two expressions are equal up to renaming of bound variables.

*$\eta$ -conversion* If you do not use a bound variable in an abstraction body, there is no need to mention it. We have, for instance,

$$\text{map} = \lambda f \rightarrow \text{map} f$$

Again we have to be careful with free variables, however:

$$e = \lambda x \rightarrow e x$$

only if  $x$  does not occur free in  $e$ . This principle is called  *$\eta$ -conversion*. One can write a function

$$\text{etacontract} :: \text{Exp} \rightarrow \text{Exp}$$

that systematically applies  *$\eta$ -conversion* to eliminate all redundant abstractions. Conversely, it is also possible to *expand* a term so that every expression of function type is an abstraction, or applied to an argument. Expansion is a bit trickier to code in Haskell than contraction, however, because of the need to generate fresh identifiers — a notorious difficulty in a purely functional setting [1].

*Equality*. We define two expressions to be *equal* if one can be obtained from the other through repeated application of  $\alpha$ - and  $\eta$ -conversion. That test could be coded in Haskell as follows:

$$\begin{aligned} &\text{instance Eq Exp where} \\ &e_1 == e_2 = \text{alphaconvertible} (\text{etacontract } e_1) (\text{etacontract } e_2) \end{aligned}$$

*$\beta$ -reduction*. It is important to realise that  $e_1 == e_2$  is *not* the same as saying that  $e_1$  and  $e_2$  represent the same value. That is because our notion of equality does not entail any notion of evaluation. For example

$$(\lambda x \rightarrow x + 1)2 \neq 2 + 1$$

The fundamental evaluation rule for expressions is

$$(\lambda x \rightarrow e_1) e_2 = \text{subst } x e_2 e_1$$

That is, all free occurrences of  $x$  in  $e_1$  are replaced by  $e_2$ . The substitution function *subst* is defined in the usual way (we assume that all naming conflicts are resolved by appropriate renaming). This evaluation rule is called  *$\beta$ -reduction*.

The function *betareduce* :: *Exp* → *Exp* exhaustively applies the  *$\beta$ -reduction* rule to all subexpressions of its argument. It is a fact that *betareduce* will always terminate, thanks to the type system that we employ (provided we ignore user-defined types). Furthermore, it does not matter in what order the reductions are carried out: the result is always the same.

If we ignore the semantics of constants, two expressions  $e_1$  and  $e_2$  represent the same value iff *betareduce*  $e_1 == \text{betareduce } e_2$ . Because *betareduce* is terminating, we could have taken this as our definition of equality, but we want to vary the notion of  *$\beta$ -reduction* in our discussion of matching below.

*One step reduction.* In particular, we shall consider the function *betastep*, which carries out one pass over an expression, applying  $\beta$ -reduction where possible:

$$\begin{aligned}
 \textit{betastep}(\textit{Var } x t) &= \textit{Var } x t \\
 \textit{betastep}(\textit{Con } ct) &= \textit{Con } ct \\
 \textit{betastep}(\textit{Lam } x et) &= \textit{Lam } x (\textit{betastep } e) t \\
 \textit{betastep}(\textit{Ap } e_1 e_2 t) &= \textit{case } e'_1 \textit{ of} \\
 &\quad \begin{array}{l} \textit{Lam } x b \_ \rightarrow \textit{subst } x e'_2 b \\ \_ \rightarrow \textit{Ap } e'_1 e'_2 t \end{array} \\
 &\quad \text{where } e'_1 = \textit{betastep } e_1 \\
 &\quad \quad e'_2 = \textit{betastep } e_2
 \end{aligned}$$

Its name derives from the fact that it captures the notion of one *parallel* reduction step. To appreciate the difference between *betareduce* and *betastep*, consider the expression

$$e = (\lambda f a \rightarrow 1 + f a)(\lambda b \rightarrow b + 2)$$

We have

$$\textit{betareduce } e = \lambda a \rightarrow 1 + (a + 2)$$

However, application of *betastep* yields

$$\textit{betastep } e = \lambda a \rightarrow 1 + (\lambda b \rightarrow b + 2) a$$

because the result of applying a substitution is not reduced again. It is not the case, therefore, that  $\textit{betareduce } e == \textit{betastep } e$ . Note that (for the particular  $e$  defined above) we do have

$$\textit{betastep}(\textit{betastep } e) == \textit{betareduce } e$$

More generally, for any expression  $e$ , there exists a natural number  $n$  so that

$$\textit{betastep}^n e == \textit{betareduce } e$$

In this sense *betastep* is an approximation of *betareduce*. As we shall see shortly, the importance of *betastep* is that it can be undone rather easily: we can enumerate its inverse image.

## Exercises

- 5.1 Write a program for *alphaconvertible*.
- 5.2 Write a program for *etacontract*.
- 5.3 Write a program for *subst*, assuming that there are no name conflicts.
- 5.4 Write a program for *betareduce*.

### 5.3 Substitutions

An *expression substitution* is a mapping from variables to expressions:

$$\text{type } Esub = Name \rightarrow Type$$

Note that an expression substitution applies only to variables in the expression, not to type identifiers that might occur in the type attributes of an expression.

A *substitution* is a pair of an expression substitution and a type substitution

$$\text{type } Sub = (Esub, Tsub)$$

The application of such a substitution  $(\epsilon, \tau)$  applies  $\epsilon$  to the variables in an expression, and  $\tau$  to all the type attributes ( $\epsilon$  and  $\tau$  should be consistent with each other so that the result of applying a substitution is well-typed):

$$\begin{aligned} \text{apply}(\epsilon, \tau)(Var\ x\ t) &= \epsilon\ x \\ \text{apply}(\epsilon, \tau)(Con\ ct) &= Con\ c(\text{tapply}\ \tau\ t) \\ \text{apply}(\epsilon, \tau)(Ap\ e_1\ e_2\ t) &= Ap\ (\text{apply}(\epsilon, \tau)\ e_1)\ (\text{apply}(\epsilon, \tau)\ e_2)\ (\text{tapply}\ \tau\ t) \\ \text{apply}(\epsilon, \tau)(Lam\ x\ et) &= Lam\ x\ (\text{apply}(\epsilon, \tau)\ e)\ (\text{tapply}\ \tau\ t) \end{aligned}$$

In the last clause of this definition, we tacitly assumed that there are no name clashes:  $\epsilon$  does not substitute for  $x$ , and  $x$  does not occur in the range of  $\epsilon$ . If this cannot be guaranteed, the variable  $x$  has to be removed from the domain of  $\epsilon$  before processing the body of  $Lam\ x\ et$ , and the bound occurrence of  $x$  has to be renamed.

A substitution  $(\epsilon, \tau)$  is *closed* if all variables substituted for by  $\epsilon$  are mapped to closed  $\lambda$ -terms (since closed  $\lambda$ -terms can contain free type variables, we cannot impose this restriction on the type substitution  $\tau$  too).

Generality of substitutions is defined the same as for type substitutions. A substitution  $\Psi$  is said to be *less general* than  $\Phi$  if there exists another substitution  $\Delta$  so that

$$\text{apply}\ \Psi\ e == \text{apply}\ \Delta(\text{apply}\ \Phi\ e) , \text{ for all } e.$$

We write  $\Psi \leq \Phi$  when  $\Psi$  is less general than  $\Phi$ . Two substitutions are *incomparable* if neither is more general than the other.

### 5.4 Matching

**Simple matching** We are now in a position to define what we mean exactly by *simple matching* of  $\lambda$ -expressions. Given a pattern  $p$  and a closed  $\lambda$ -term  $e$ , a *simple match* is a closed substitution  $\Phi$  which satisfies the additional restriction that variables are only mapped to beta-normal forms (i.e. fully beta-reduced expressions), such that

$$\text{apply}\ \Phi\ p == e$$

As in the case of types, if there exists a simple match, there exists a most general simple match.

It is fairly easy to extend standard matching algorithms to cope with bound variables and  $\eta$ -conversion. Let

$$\text{simplematch} :: \text{Exp} \rightarrow \text{Exp} \rightarrow [\text{Sub}]$$

be the function so that  $\text{simplematch } p \ e$  returns a singleton containing the most general match if a match exists at all, and the empty list otherwise.

**Ideal matching** Now suppose that we modify the condition in the above definition of simple matching to read

$$\text{betareduce}(\text{apply } \Phi \ p) == \text{betareduce } e$$

A substitution  $\Phi$  that satisfies this equation is said to be an *ideal match*. Ideal matches may be incomparable in the generality order, and not have a common generalisation. To see this, let

$$p = f \ x \ \text{and} \ e = 0$$

then both

$$\left\{ \begin{array}{l} f := \lambda a \rightarrow a \\ x := 0 \end{array} \right\} \text{ and } \left\{ f := \lambda a \rightarrow 0 \right\}$$

are ideal matches, the two are incomparable, and there exists no ideal match that generalises both.

It follows that we need to modify the concept of a ‘most general match’. Given  $p$  and  $e$ , the *ideal match set* is a set  $X$  of ideal matches for  $p$  and  $e$  such that

- for each ideal match  $\Phi$  there exists  $\Psi$  in  $X$  so that  $\Phi \leq \Psi$
- the elements of  $X$  are pairwise incomparable.

The first clause is a completeness condition: it says that every ideal match is represented in  $X$ . The second clause says that there are no redundant elements in  $X$ .

Unfortunately, it is not even decidable whether an ideal match set is empty or not [13]. It follows that we cannot hope to solve the matching problem in its full generality. Intuitively, this is unsurprising because the inverse image of  $\text{betareduce}$  is potentially infinite. As remarked before, however, the inverse image of  $\text{betastep}$  can be enumerated without too much difficulty.

**One-step matching** We are thus led to the following definition. Given a pattern  $p$  and an expression  $e$ , a *one-step match* is a substitution  $\Phi$  such that

$$\text{betastep}(\text{apply } \Phi \ p) == \text{betareduce } e$$

The definition of a *one-step match set* is analogous to that of an ideal match set. The matching algorithm in MAG computes this one-step match set.

To understand the behaviour of MAG, it is important that the reader develops an intuition for one-step matching, so that she can adapt the transformation rules (as we have done in the *minddepth* exercise). As a simple but typical example, consider the pattern and term

$$\lambda a x \rightarrow f a ((+)x) \text{ and } \lambda a x y \rightarrow x ++ (a : y)$$

The one-step match set of this pair is a singleton, namely

$$\{ f := \lambda a b c \rightarrow b(a : c) \}$$

To illustrate the difference between one-step matching and ideal matching, consider  $f x$  and  $\lambda a \rightarrow 1 + (a + 2)$ . The match

$$\left\{ \begin{array}{l} f := \lambda g a \rightarrow 1 + g a \\ x := \lambda b \rightarrow b + 2 \end{array} \right\}$$

is in the ideal match set, but since

$$\text{betastep}((\lambda g a \rightarrow 1 + g a)(\lambda b \rightarrow b + 2)) = \lambda a \rightarrow 1 + (\lambda b \rightarrow b + 2) a$$

which is not the same as  $\lambda a \rightarrow 1 + (a + 2)$ , it is not in the one-step match set.

It is beyond the scope of these notes to go into the details of an algorithm for computing one-step match sets. Essentially it proceeds by repeatedly extracting subexpressions. Such an algorithm, its proof of correctness and a performance comparison with similar algorithms are the subject of a forthcoming paper.

## 6 Concluding remarks

In these lectures, we have attempted to demonstrate that the fusion transformation is itself a generic program, whose parameters are the distributivity conditions needed in its application. The scope of its applicability is marred only by the limitations of the matching algorithm used to implement rewriting.

We have proposed the *one-step matching* algorithm for typed  $\lambda$ -expressions, which appears not to be commonly known. Compared to more traditional matching algorithms (which restrict the order of variables), this algorithm greatly enhances the applicability of transformations such as fusion. It is however still necessary to state special cases of fusion as separate rules, most notably for tupling.

These ideas were illustrated by a cheap-and-cheerful Haskell program, the MAG system. We found the libraries for pretty printing and parsing from Utrecht University an invaluable tool. The attribute grammar system from Utrecht made it easy to experiment with different versions of the type checker.

## Acknowledgements

Mike Spivey's rewriting system was the starting point of these investigations. He also commented on a draft of these notes, suggesting several corrections

and improvements. Richard Bird generously shared insights gained while implementing his own Functional Calculator, and also commented on a draft. We are grateful for his advice and friendship. Jeremy Gibbons suggested many examples — unfortunately, time and space did not allow us to include them all. Doaitse Swierstra provided all three tools used in building MAG: the pretty-printing and parsing libraries, as well as his attribute grammar system. He furthermore provided encouragement and many suggestions during a visit to Oxford at the beginning of August. Ivan Sanabria pointed out a number of mistakes in an early draft.

## References

1. L. Augustsson, M. Rittri, and D. Synek. Functional pearl: On generating unique names. *Journal of Functional Programming*, 4:117–123, 1994.
2. F. Bellegarde. A transformation system combining partial evaluation with term rewriting. In *Higher-Order Algebra, Logic and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 40–55. Springer-Verlag, 1994.
3. R. S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, December 1980.
4. R. S Bird. The promotion and accumulation strategies in functional programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984.
5. R. S Bird. *Introduction to Functional Programming in Haskell*. International Series in Computer Science. Prentice Hall, 1998.
6. E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18(2):139–179, 1992.
7. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
8. W. N. Chin. Fusion and tupling transformations: Synergies and conflicts (invited paper). In *Fuji International Workshop on Functional and Logic Programming*, pages 176–195. World Scientific, 1995.
9. R. Curien, Z. Qian, and H. Shi. Efficient second-order matching. In *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 317–331. Springer Verlag, 1996.
10. J. Gibbons and G. Jones. The under-appreciated unfold. In *3rd ACM SIGPLAN International Conference on Functional Programming 1998*, 1998.
11. Z. Hu, H. Iwasaki, and M. Takeichi. Calculating accumulations. Technical Report METR 96-0-3, Department of Mathematical Engineering, University of Tokyo, Japan, 1996. Available from URL: <http://www.ipl.t.u-tokyo.ac.jp/~hu/pub/tech.html>.
12. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 164–175, 1997.
13. G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
14. J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Functional Programming Languages and Computer Architecture*, pages 314–323. Association for Computing Machinery, 1995.
15. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

16. Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system hyl. In R. S. Bird and L. Meertens, editors, *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997.
17. A. Pettorossi. Methodologies for transformations and memoing in applicative languages. Ph.D. thesis CST-29-84, University of Edinburgh, Scotland, 1984.
18. S. L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Foundations of Computing Series. Prentice-Hall International, 1994.
19. S. L. Peyton-Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *Science of Computer Programming*, 32(1–3):3–48, 1998.
20. S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Second International Summer School on Advanced Functional Programming*, volume 1126 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.

# Generic Program Transformation

## Answers to exercises

### 1 Introduction

No exercises.

### 2 Abstraction versus efficiency

#### Exercises

**2.1** Can the computation of the *maximum* depth of a tree be made more efficient in the same way as the computation of the minimum depth? If not, which property of *min* fails for *max*?

**Answer:** The property  $\max(a + b) c = \text{if } b \leq c \text{ then } c \text{ else } \max(a + b) c$  fails to hold. It is thus not possible to cut the search for a maximum depth leaf in the same way as we cut the search for a minimum depth leaf.

**2.2** A leaf labelled binary tree is said to be *perfectly balanced* if for every subtree  $t$ , the size of the left hand child is precisely  $\text{size } t \text{ div } 2$ . Given a list  $x$ , the function *build*  $x$  produces a perfectly balanced tree whose inorder traversal is  $x$ . First give a naive program for computing *build*, and then show how it can be made efficient.

**Answer:** The obvious algorithm follows the divide-and-conquer strategy, splitting the input into half at each step, and recursing on each half. That may be coded as follows

$$\begin{aligned} \text{build}[a] &= \text{Leaf } a \\ \text{build } x &= \text{Bin}(\text{build } y)(\text{build } z) \\ &\quad \text{where } (y, z) = \text{splitAt } n \ x \\ &\quad \quad n = \text{length } x \text{ div } 2 \end{aligned}$$

This is not efficient, because *splitAt*  $n \ x$  takes  $n$  steps to evaluate; the cost adds up to  $\Omega(n \log n)$  for the evaluation of *build*  $x$  where  $n$  is the length of  $x$ . To make this more efficient, we compute

$$\text{build}' \ n \ x = (\text{build}(\text{take } n \ x), \text{drop } n \ x)$$

Note that this is very similar to the way we optimised *decorate*. The efficient program for *build'* reads

$$\begin{aligned} \text{build}' \ 1 \ (a : x) &= (\text{Leaf } a, x) \\ \text{build}' \ n \ x &= (\text{Bin } s \ t, z) \\ &\quad \text{where } (s, y) = \text{build}' \ (n \text{ div } 2) \ x \\ &\quad \quad (t, z) = \text{build}' \ (n - n \text{ div } 2) \ y \end{aligned}$$

**2.3** Are there circumstances where the original definition of *partition* is as efficient as the 'improved' version?

**Answer:** Yes, for instance when we select only one of the two components of its result.

### 3 Automating the transition: fusion and higher order rewriting

#### Exercises

**3.1** Define the *fold* operator on natural numbers. Its type is

$$\text{foldn} :: (a \rightarrow a) \rightarrow a \rightarrow \text{Int} \rightarrow a$$

You could think of the first argument as replacing the successor function (+1) and the second argument as replacing 0. If there were a data type *Nat* of natural numbers in Haskell, the type of the third argument would be *Nat* and not *Int*. What is the fusion rule for *foldn*?

**Answer:** The definition of *foldn* is

$$\begin{aligned} \text{foldn step start } 0 &= \text{start} \\ \text{foldn step start } (n + 1) &= \text{step}(\text{foldn step start } n) \end{aligned}$$

The fusion rule is what allows you to push you another operator through this computation, in the following intuitive fashion:

$$\begin{aligned} f(\text{step}(\text{step}(\text{step} \dots (\text{step start})))) &= \\ g(f(\text{step}(\text{step} \dots (\text{step start})))) &= \\ g(g(f(\text{step} \dots (\text{step start})))) &= \\ g(g(g(f \dots (\text{step start})))) &= \\ \dots & \\ g(g(g \dots f(\text{step start}))) &= \\ g(g(g \dots g(f start))) &= \\ g(g(g \dots g e)) & \end{aligned}$$

Formally, we have

$$f(\text{foldn step start } x) = \text{foldn } g e x,$$

provided we have

$$\begin{aligned} f(\text{step } n) &= g(f n) \quad \text{all } n, \text{ and} \\ f \text{ start} &= e \end{aligned}$$

**3.2** Define the *fold* operator on leaf labelled binary trees. Its type is

$$\text{foldbtree} :: (a \rightarrow a \rightarrow a) \rightarrow (b \rightarrow a) \rightarrow \text{Btree } b \rightarrow a$$

What is its fusion rule?

**Answer:** The fold operator is

$$\begin{aligned} \text{foldbtree bin leaf } (\text{Leaf } a) &= \text{leaf } a \\ \text{foldbtree bin leaf } (\text{Bin } s t) &= \text{bin } (\text{foldbtree bin leaf } s) (\text{foldbtree bin leaf } t) \end{aligned}$$

The fusion rule says that

$$f(\text{foldbtree } \text{bin leaf } x) = \text{foldbtree } g h x$$

provided we have

$$\begin{aligned} f(\text{bin } s t) &= g(f s)(f t), \quad \text{all } s \text{ and } t \\ f(\text{leaf } a) &= h a, \quad \text{all } a \end{aligned}$$

## 4 The MAG system

Model solutions to the practical assignments of this section are distributed with the MAG system. The MAG system is available from URL:

<http://www.comlab.ox.ac.uk/oucl/groups/progtools/mag.htm>

Below we answer the in-lined questions only.

### 4.1 Getting acquainted

No exercises.

### 4.2 Accumulation Parameters

**Fast reverse** A naive definition of the function *reverse* is

$$\begin{aligned} \text{reverse}[] &= [] \\ \text{reverse}(a : x) &= \text{reverse } x \text{ ++ } [a] \end{aligned}$$

Write down the recursive definition of (++), and estimate how much time it takes to evaluate *reverse* *x* when *x* has length *n*.

**Answer:** The recursive definition of (++) is:

$$\begin{aligned} [] \text{ ++ } y s &= y s \\ (x : x s) \text{ ++ } y s &= x : (x s \text{ ++ } y s) \end{aligned}$$

The time to evaluate a concatenation is thus proportional to the length of the left-hand argument. It follows that *reverse* takes quadratic time. **End of answer.** The time complexity of *reverse* can be improved by defining:

$$\text{fastrev } x y = \text{reverse } x \text{ ++ } y$$

Why is this a generalisation of *reverse*?

**Answer:** Because

$$\text{fastrev } x [] = \text{reverse } x \text{ ++ } [] = \text{reverse } x$$

**End of answer.**

Using the fact that  $(++)$  is associative

$$(x ++ y) ++ z = x ++ (y ++ z)$$

one can synthesise an efficient program for *fastrev*.

The above definitions, and the associativity of  $(++)$ , have been recorded in the theory file `reverse.eq`. There is one peculiarity, however: the definition of *fastrev* reads

$$\text{fastrev } xy = \text{reverse}(\text{foldr } (:) [] x) ++ y$$

Why is the instance of *foldr* there?

**Answer:** It acts as a seed for the use of fusion: if there is no *foldr* in the program, fusion cannot be applied. **End of answer.**

Confirm your answer by loading the theory file `reverse.eq` in MAG, and transforming *fastrev*. Estimate the time complexity of the resulting program.

**Answer:** The result is an instance of *foldr* where each of the operators takes constant time to evaluate, so the total time complexity is linear. **End of answer.**

**Postorder traversal** No in-lined exercises.

**Breadth first traversal** ...

The function  $\text{levels} :: \text{Tree } a \rightarrow [[a]]$  first gives the elements that occur at depth 0, then the elements at depth 1, depth 2 and so on. It is defined by

$$\text{levels}(\text{Node } a \text{ } ts) = [a] : \text{glues}(\text{map } \text{levels } ts)$$

$$\begin{aligned} \text{glues } [] &= [] \\ \text{glues } (x : xs) &= \text{lzc } x (\text{glues } xs) \end{aligned}$$

$$\begin{aligned} \text{lzc } [] x &= x \\ \text{lzc } (a : x) y &= \text{if } \text{null } y \\ &\quad \text{then } a : \text{lzc } x [] \\ &\quad \text{else } (a ++ \text{head } y) : \text{lzc } x (\text{tail } y) \end{aligned}$$

Here *lzc* stands for *long zip with concatenation*. What is the difference between *lzc* and *zipWith*( $++$ )? Is *lzc* associative? What is the time complexity of *levels*?

**Answer:** *zipWith* truncates its result to the length of the shortest argument. By contrast, the length of the result of *lzc* is the maximum of the lengths of its arguments. Yes, *lzc* is associative. The time complexity of *levels* is at least quadratic. **End of answer.**

**Minimum depth** Let us now return to the problem discussed at the beginning of these lectures, namely computing the minimum depth of a leaf labelled binary

tree. It was claimed there that apart from the definitions, all we need to produce an efficient program is the following set of rules:

$$\begin{aligned}0 + a &= a \\(a + b) + c &= a + (b + c) \\ \min(\min a b) c &= \min a (\min b c) \\ \min a b + c &= \min (a + c)(b + c) \\ \min(a + b) c &= \text{if } b \geq c \text{ then } c \text{ else } \min(a + b) c\end{aligned}$$

Unfortunately, there is a problem when this set of equations is implemented by rewriting. Why?

**Answer:** The last rule can be applied to its own result, so application of this set of rules does not terminate. **End of Answer.**

### 4.3 Tupling

**Fibonacci** No in-lined exercises.

**Steep sequences** A list of numbers is said to be *steep* if each element is greater than the sum of the elements that follow it. We might define the predicate  $steep :: [Int] \rightarrow Bool$  by

$$\begin{aligned}steep [] &= True \\ steep(a : x) &= a > sum x \wedge steep x\end{aligned}$$

Estimate the time complexity of *steep*.

**Answer:** The time complexity of *sum* is linear. We call *sum* on each suffix of the argument of *steep*, so that makes for a quadratic program. **End of answer.**

**Partition** In this example, the ordering of the rules in the theory file is rather important: in the rewriting process, they are tried in the order they occur. Why does the rule about distributing a function over a conditional come last?

**Answer:** When applied in favour of *if* contraction (to a nested *if* statement) this rule is applicable to its own result. So it should come after *if* contraction. **End of answer.**

**Decorate . . .**

Returning to the problem of optimising *decorate*, the above machinery can be found in the file `decorate.eq`. Run MAG on *dec*. Is the result identical to the program discussed earlier in the lectures? What further transformation needs to be performed?

**Answer:** We need to perform common sub-expression elimination. That is not easily expressed as a rewrite rule, and it is typical of the kind of transformation that MAG cannot do. **End of answer.**

#### 4.4 Carrying on

No exercises.

### 5 Matching typed $\lambda$ -expressions

#### Exercises

5.1 Write a program for *alphaconvertible*.

**Answer:**

```
data Binding = Free Name | Bound Int
    deriving Eq
```

```
alphaconvertible e1 e2 = ac [[]] e1 e2
```

```
where ac xs ys (Var x s) (Var y t) = (s == t)  $\wedge$ 
    getbinding xs x == getbinding ys y
```

```
    where getbinding as a | a `elem` as = Bound (getpos a as)
        | otherwise = Free a
```

```
        getpos a (b : bs) = if (a == b) then 0 else 1 + getpos a bs
```

```
ac xs ys (Con x s) (Con y t) = (s == t)  $\wedge$  (x == y)
```

```
ac xs ys (Ap f1 a1 s) (Ap f2 a2 t) = (s == t)  $\wedge$  ac xs ys f1 f2  $\wedge$  ac xs ys a1 a2
```

```
ac xs ys (Lam p1 e1 s) (Lam p2 e2 t) = (s == t)  $\wedge$  ac xs' ys' e1 e2
```

```
    where xs' = p1 : xs
```

```
        ys' = p2 : ys
```

```
ac _ _ _ _ = False
```

5.2 Write a program for *etacontract*.

**Answer:**

```
etacontract exp@(Lam v1 (Ap e (Var v2 _) _) _) =
```

```
if (v1 == v2  $\wedge$  and (map (v1/=) (freevars e)))
```

```
then (etacontract e)
```

```
else exp
```

```
etacontract (Lam pat bod t) = Lam pat (etacontract bod) t
```

```
etacontract (Ap f a t) = Ap (etacontract f) (etacontract a) t
```

```
etacontract (Var x t) = Var x t
```

```
etacontract (Con c t) = Con c t
```

**5.3** Write a program for *subst*, assuming that there are no name conflicts.

**Answer:**

$$\begin{aligned} \text{subst } x \ e \ (\text{Var } y \ t) &= \text{if } x == y \ \text{then } e \ \text{else } \text{Var } y \ t \\ \text{subst } x \ e \ (\text{Con } ct) &= \text{Con } ct \\ \text{subst } x \ e \ (\text{Ap } e_1 \ e_2 \ t) &= \text{Ap} (\text{subst } x \ e \ e_1) (\text{subst } x \ e \ e_2) \ t \\ \text{subst } x \ e \ (\text{Lam } y \ b \ t) &= \text{if } x == y \ \text{then } (\text{Lam } y \ b \ t) \\ &\quad \text{else } \text{Lam } y \ (\text{subst } x \ e \ b) \ t \end{aligned}$$

**5.4** Write a program for *betareduce*.

**Answer:**

$$\begin{aligned} \text{betareduce} (\text{Var } x \ t) &= \text{Var } x \ t \\ \text{betareduce} (\text{Con } ct) &= \text{Con } ct \\ \text{betareduce} (\text{Lam } x \ e \ t) &= \text{Lam } x \ (\text{betareduce } e) \ t \\ \text{betareduce} (\text{Ap } e_1 \ e_2 \ t) &= \text{case } e'_1 \ \text{of} \\ &\quad \text{Lam } x \ b \ _ \rightarrow \text{betareduce} (\text{subst } x \ e'_2 \ b) \\ &\quad \_ \rightarrow \text{Ap } e'_1 \ e'_2 \ t \\ \text{where } e'_1 &= \text{betareduce } e_1 \\ e'_2 &= \text{betareduce } e_2 \end{aligned}$$