Higher-order matching for program transformation

Ganesh Sittampalam
Magdalen College

D.Phil. Thesis
Trinity Term, 2001

Higher-order matching for program transformation

Ganesh Sittampalam, Magdalen College

D.Phil. Thesis
Trinity Term, 2001

Abstract

The tension between abstraction and efficiency in programming can be eased by *program transformation*; programs which are highly abstract and readable, but inefficient, are automatically converted to efficient forms, allowing the programmer to gain the advantages of having easy to understand and maintainable source whilst also obtaining reasonable performance from the compiled code.

Many transformations cannot be detected and applied completely automatically. *Active source* seeks to circumvent this problem by allowing the programmer to have some control over what transformations are applied and how this is done, by *annotating* the source code with just enough information to allow the transformation to be mechanised.

Some complex transformations can be expressed as conditional higher-order rewrite rules. Mechanically applying such rules requires the use of *higher-order matching*, which is undecidable and infinitary in general for any type system more complex than the simply-typed lambda calculus. This thesis is concerned with algorithms for a suitably restricted version of this problem for use in applying such transformations.

Our contributions are to present two novel algorithms, the *one-step* and the *two-step* algorithms, which depart from previous work in the field by not being dependent on any particular type system and by not being specified in terms of the "order" of the results generated. Instead we restrict the notion of β-equality so as to make the problem tractable. Our algorithms have clear specifications, so that if they do not apply in any particular situation it is easy to understand why.

We also show how our algorithms have been implemented in the functional programming language Haskell as part of the prototype program transformation system MAG, and give various examples of applying them to transformation problems.

Acknowledgments

There are many people who have contributed to this thesis in one way or another, and I would like to express my gratitude to each of them. First and foremost, my supervisor, Oege de Moor, who has collaborated with me on the work presented here, showed me how research should be done, and helped me to learn the essential arts of speaking and writing. He has always been available to help when required, but has also given me the space I needed to learn to survive on my own.

The Programming Tools Group here at Oxford, of which I am a member, has provided invaluable support, both through regular meetings which provided a forum for immediate review of work in progress, and by providing a network of mutual support and assistance. In particular, Eric Van Wyk read this entire document with very little notice and made many valuable suggestions, and Yorck Hünke read some of the chapters and found various errors and points which required further elucidation. My officemates Kevin Backhouse and Iván Sanabria-Piretti provided much help with LaTeXproblems.

A large group of friends in Oxford made an invisible contribution simply by being there, especially when work was not going so well. In addition, Ian Lynagh proofread much of this thesis and pointed out many obvious and not-so-obvious errors, and Ben Kremer proofread an earlier progress report, some of which material has ended up in here in modified form.

I very much appreciate the effort put in by my examiners, Richard Bird and John Hughes, and the many helpful comments they made during my oral examination. At an earlier progress review, Richard Bird also suggested making use of higher-order functions to allow the common elements of each matching algorithm to be elegantly shared; this idea made a big difference to the elegance of the presentation.

Microsoft Research provided generous financial support for the entire period of my doctorate, and the Intentional Programming team were hosts for two enjoyable internships at Microsoft in Redmond.

Finally, I would like to thank all the people involved in the rather unusual circumstances in which I did my undergraduate degree which laid the foundations for this doctorate, in particular my parents and Donald Keedwell from the University of Surrey. I am also very grateful for the encouragement from my parents of my early interest in computers which eventually led me into this field of research.

Contents

Chapter 1

Introduction

1.1 Motivation

Program transformation is the process of converting a piece of code from one form to another whilst preserving its essential meaning. Most often, the aim is to improve time or space efficiency, though other goals such as making an abstract specification executable also fall within this field.

There is an enormous variety of potential transformations, many of which are relatively simple and can be easily applied automatically by a computer; such transformations are often implemented as part of modern optimising compilers and programmers can enjoy the benefits they bring in terms of the performance of the compiled code without needing any knowledge of their details. However, many others are rather more complicated and require sophisticated analysis or specialised knowledge to apply, which makes it infeasible to search for and apply them as a routine part of the compilation process. In other cases, a transformation can be automatically detected and applied but it is hard for the programmer to predict when writing code whether or not the transformation will successfully apply to that code.

The alternative to automatic transformation is for the programmer to simply apply the relevant change manually when a guarantee of the extra performance it will provide is required. This process might be completely implicit; an experienced programmer would just write the optimised program directly, perhaps without even thinking about the less optimal version that could have been written instead. However, the gain in efficiency comes at a price; less optimal code is often also simpler and more readable and thus easier to maintain and less likely to contain bugs.

It is this clash that motivates the concept of *active source*. Recognising that many programs can be written both in an inefficient simple form and in an efficient complex form, and that these forms often cannot easily be linked by completely automatic transformations, we seek a compromise. We write the program in the first form, but add extra annotations that give enough direction to the compiler for the second form to be automatically derived.

This thesis seeks to advance this concept by providing practical techniques to support it. In particular, we focus on *source-to-source* transformations, that is those which return a program in the same source language as the input program. Such transformations have the property that it should be easy to inspect the resulting code to verify whether the transformation had the desired result or not. This possibility is likely to be particularly important to programmers who have gone to the extra effort of annotating their program in the first place.

One of the key elements of mechanising the application of complex program transformations is *higher-order matching*. We develop two algorithms to carry this out that represent an advance on existing theory, provide practical implementations of these algorithms, and show how they can be applied to mechanise some rather involved manual *promotion* derivations from the literature. One of the key features of the algorithms is that they have clear specifications of what they do and do not do. Thus, if a transformation fails because our higher-order matching algorithms fail, users can see from the specification exactly why this was.

For simplicity, we choose to transform *functional* programs. One key feature that these have is *referential transparency* – it is always possible to replace an expression with its value without changing the meaning of the program, making it straightforward to apply transformations based on *equational reasoning* to our programs. Some might consider this choice a bit limiting, but we see no intrinsic reason why our algorithms could not also be used with imperative programs, given enough time to deal with the usual complexities of transforming languages that allow side-effects. Of course, others would argue that functional programs are actually superior to imperative programs in any case – languages such as Haskell offer significant advantages, particularly in terms of ease of use and modularity [48].

The rest of this thesis is structured as follows:

- In the remainder of this chapter we provide a gentle introduction to promotion (Section 1.2) and explain how we shall mechanise this transformation (Section 1.2.1). We then discuss related work in the broader field of *fusion* transformations (Section 1.3) and go into more depth for the *deforestation* (Section 1.3.1) and *short-cut deforestation* (Section 1.3.2) transformations which are most closely related to our work.
- In Chapter 2 we introduce the problem of higher-order matching and lay a framework within which our matching algorithms shall be developed.
- In Chapters 3 and 4 we develop the *one-step* and *two-step* algorithms respectively and discuss in detail some related matching algorithms from the literature (Sections 3.4 and 4.4).
- In Chapter 5 we show how these algorithms can be implemented in the functional programming language Haskell as part of MAG, a toy program transformation system.
- Chapter 6 gives some examples of transformations that have been successfully mechanised using MAG.
- Finally, we conclude in Chapter 7 by discussing the value of our work and possible future developments.

- Appendix A provides a quick reference for the notation and important definitions in this thesis.
- Appendix B gives the exact details of the examples discussed in Chapter 6.

Readers purely interested in using MAG are advised to read the beginning of Chapter 5 and Section 5.1, and then Chapter 6 with reference to the actual input and output to MAG given in Appendix B. Those interested in using our matching algorithms in their own projects should read Chapters 2, 3 and 4, along with Chapter 5 from Section 5.2 onwards. It may be advisable to skip the proofs of the lemmas that occasionally appear, and the proofs of correctness in Sections 2.5, 3.3 and 4.3. Of course, those interested in verifying the correctness of our algorithms for themselves should check these proofs carefully!

1.2 Promotion

Consider the obvious program to calculate the reverse of a list:

$$\begin{aligned} &\mathit{reverse}\,[\,] = [\,] \\ &\mathit{reverse}\,(x : xs) = \mathit{reverse}\ xs \mathbin{+\!\!+} [x] \end{aligned}$$

This program is well known in the functional programming community as the canonical example of inefficient use of $(+\!\!+)$, the operator that joins two lists together. The problem is that $(+\!\!+)$ takes running time proportional to the length of its first argument. Thus, programs such as *reverse* that build up a result by repeatedly adding to the right-hand end of a list often end up taking time that is quadratic in the size of the result.

The well-known improvement to this program and others like it [97] is known as *cat-elimination*. It takes advantage of the associativity of $(+\!\!+)$ to reorganise the calculation so that new elements are added to the left-hand side of the list being constructed. This requires elements to be added in the opposite order, which can be achieved by the addition of an *accumulating parameter* to the program definition; this parameter is used to build up the result starting with the rightmost element instead of the leftmost. An auxiliary function *fastreverse* that takes an extra parameter *ys* is defined, and *reverse* is redefined to invoke *fastreverse*, initialising *ys* to the empty list $[\,]$:

$$\begin{aligned}&\mathit{reverse}\ \mathit{xs}\ =\ \mathit{fastreverse}\ \mathit{xs}\ [\,]\\&\mathit{fastreverse}\ [\,]\ \mathit{ys}\ =\ \mathit{ys}\\&\mathit{fastreverse}\ (x : \mathit{xs})\ \mathit{ys}\ =\ \mathit{fastreverse}\ \mathit{xs}\ (x : \mathit{ys})\end{aligned}$$

Proving that the original and improved programs for *reverse* are equal is a straightforward induction argument; however what we really want to do is automatically *generate* the second program. If this is carried out by a process of applying correct transformation rules, this will of course simultaneously provide a proof of correctness.

The key detail that links these two programs is the following specification for the *fastreverse* function:

$$\mathit{fastreverse}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{reverse}\ \mathit{xs}\ +\!\!+\ \mathit{ys}$$

This specification encodes in a formal manner the insight that we stated in words above, namely that the optimised version can be obtained by adding an accumulating parameter to *reverse*. Of course, it still remains to show how this leads to the improved program. If we were to do this manually, the derivation would look something like the following:

$$\begin{aligned}&\quad \mathit{fastreverse}\ [\,]\ \mathit{ys}\\=&\quad\quad \{\text{specification of } \mathit{fastreverse}\}\\&\quad \mathit{reverse}\ [\,] +\!\!+\ \mathit{ys}\\=&\quad\quad \{\text{definition of } \mathit{reverse}\}\\&\quad [\,]\ +\!\!+\ \mathit{ys}\\=&\quad\quad \{\text{definition of } (+\!\!+)\}\\&\quad \mathit{ys}\end{aligned}$$

$$
\begin{aligned}
& \mathit{fastreverse}\ (x : \mathit{xs})\ \mathit{ys} \\
= \quad & \{\text{specification of } \mathit{fastreverse}\} \\
& \mathit{reverse}\ (x : \mathit{xs}) \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\text{definition of } \mathit{reverse}\} \\
& (\mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\text{associativity of } (\mathbin{+\!\!+})\} \\
& \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} \mathit{ys}) \\
= \quad & \{\text{definition of } (\mathbin{+\!\!+})\} \\
& \mathit{reverse}\ \mathit{xs} \mathbin{+\!\!+} (x : \mathit{ys}) \\
= \quad & \{\text{specification of } \mathit{fastreverse}\} \\
& \mathit{fastreverse}\ \mathit{xs}\ (x : \mathit{ys})
\end{aligned}
$$

This derivation gives us the efficient program for *fastreverse*; examination of the specification also shows us how we can define *reverse* in terms of *fastreverse*.

For another example of a transformation between an obvious inefficient program and a more subtle efficient one, consider the following program for calculating the minimum depth of a leaf-labelled binary tree:

$$
\begin{aligned}
& \mathbf{data}\ \mathit{Tree}\ \alpha \ = \ \mathit{Leaf}\ \alpha \mid \mathit{Bin}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
& \mathit{mindepth}\ (\mathit{Leaf}\ x) \ = \ 0 \\
& \mathit{mindepth}\ (\mathit{Bin}\ t_1\ t_2) \ = \ \mathit{min}\ (\mathit{mindepth}\ t_1)\ (\mathit{mindepth}\ t_2) + 1
\end{aligned}
$$

This program will examine every node of the tree, which is often not necessary. For example, the left-leaning tree in Figure 1.1 has a minimum depth of 1; this can be established quickly by noticing that the right branch of the root node is a leaf and that it is therefore not necessary to check the left branch. Thus, one possible optimisation would traverse the tree recursively as in the program above, but cut off the search on any particular branch if it reached a depth that was greater than the minimum depth already found

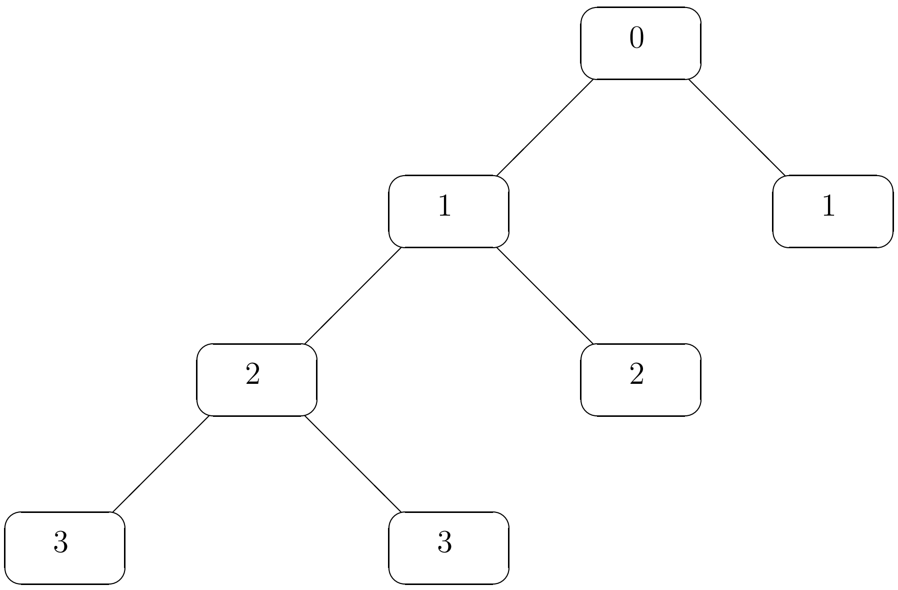

Figure 1.1: A left-leaning binary tree with depths labelled

(another possibility that we shall not explore here would be to conduct a breadth-first search).

To implement this optimisation, two accumulating parameters are needed. One, which we call d, stores the current depth we have reached in the tree being traversed, and another, m, that stores the minimum depth that has been seen so far. If the first parameter reaches the value of the second on any particular branch of the tree, then the search can be terminated for that branch. We can specify this optimisation as before:

$$\begin{aligned} &\mathit{md}\ t\ d\ m \ = \ \mathit{min}\ (\mathit{mindepth}\ t\ +\ d)\ m \\ &\mathit{mindepth}\ t \ = \ \mathit{md}\ t\ 0\ \infty \end{aligned}$$

The following calculation then gives us an efficient program for md:

$$\begin{aligned} & \mathit{md}\ (\mathit{Leaf}\ x)\ d\ m \\ = \quad & \{\text{specification of } \mathit{md}\} \\ & \mathit{min}\ (\mathit{mindepth}\ (\mathit{Leaf}\ x)\ +\ d)\ m \end{aligned}$$

$=$ {definition of *mindepth*}

$min\ (0 + d)\ m$

$=$ {definition of $+$}

$min\ d\ m$

$md\ (Bin\ t_1\ t_2)\ d\ m$

$=$ {specification of *md*}

$min\ (mindepth\ (Bin\ t_1\ t_2)\ +\ d)\ m$

$=$ {definition of *mindepth*}

$min\ ((min\ (mindepth\ t_1)\ (mindepth\ t_2)\ +\ 1)\ +\ d)\ m$

$=$ {associativity of $+$}

$min\ ((min\ (mindepth\ t_1)\ (mindepth\ t_2))\ +\ (1 + d))\ m$

$=$ {the result of *mindepth* is non-negative}

```
if $1 + d \geq m$ then $m$
              else $min\ (min\ (mindepth\ t_1)\ (mindepth\ t_2)$
                          $+\ (1 + d))$
                   $m$
```

$=$ {$+$ distributes over *min*}

```
if $1 + d \geq m$ then $m$
              else $min\ (min\ (mindepth\ t_1\ +\ (1 + d))$
                              $(mindepth\ t_2\ +\ (1 + d)))$
                   $m$
```

$=$ {associativity of *min*}

```
if $1 + d \geq m$ then $m$
              else $min\ (mindepth\ t_1\ +\ (1 + d))$
                   $(min\ (mindepth\ t_2\ +\ (1 + d))\ m)$
```

$=$ {specification of *md*}

```
if $1 + d \geq m$ then $m$
              else $md\ t_1\ (1 + d)\ (md\ t_2\ (1 + d)\ m)$
```

Thus, we are left with this program:

$$
\begin{array}{lcl}
\mathit{mindepth}\ t & = & \mathit{md}\ t\ 0\ \infty \\
\mathit{md}\ (\mathit{Leaf}\ x)\ d\ m & = & \mathit{min}\ d\ m \\
\mathit{md}\ (\mathit{Bin}\ t_1\ t_2)\ d\ m & = & \textbf{if}\ 1 + d \geq m \\
 & & \quad \textbf{then}\ m \\
 & & \quad \textbf{else}\ \ \mathit{md}\ t_1\ (1 + d)\ (\mathit{md}\ t_2\ (1 + d)\ m)
\end{array}
$$

The two derivations that we have described, for fast reverse and minimum depth, represent quite different optimisations. However, they do have some features in common; in particular, in both cases the initial and final programs are recursive traversals of an inductive datatype. Both derivations proceed by the following pattern, a strategy first put forward by Bird in 1984 under the name *promotion* [9]. Here we present the steps in the manner of Burstall and Darlington's famous *fold/unfold* transformations [15]. For simplicity, we neglect the issue of mutually recursive datatypes, which can also be handled by this strategy with a little more care.

- Define original program $\mathit{orig}\ x$, where x is a variable of type T.
- Specify optimised program $\mathit{opt}\ x\ \underline{r}$ (where $\underline{r}$ is a set of additional parameters) in the form $\mathit{opt}\ x\ \underline{r}\ =\ f\ (\mathit{orig}\ x)\ \underline{r}$
- For each constructor C in the datatype T, instantiate x to a generic instance of C and:
  - Unfold the specification of $\mathit{opt}\ x\ \underline{r}$
  - Unfold the resulting instance of $\mathit{orig}\ x$
  - Do some calculation on the result using equational laws to give a form suitable for the final step:
  - For each parameter y of C of type T, fold all occurrences of y into the form $\mathit{opt}\ y\ \underline{r'}$ for some $\underline{r'}$.

From the point of view of an automatic program transformation system, the difficulty with this pattern is that it is necessary to apply the equation specifying the optimisation in both directions; forwards in the initial unfolding step and backwards in the final folding step. A naïve system which tries to apply every rule it knows wherever possible will inevitably loop if given rules of this nature, which is a well-known problem with Burstall and Darlington's transformations.

Fortunately, category theory provides an elegant formalism within which this strategy can be encapsulated. We mentioned that the initial and final programs are recursive traversals of inductive datatypes; formally, this means that they can be expressed as *folds* (otherwise known as *catamorphisms*). Each datatype has precisely one fold, which can be derived from the datatype in a straightforward fashion; any recursive traversal of the datatype can be uniquely expressed in terms of this fold. A formal law which is also known as promotion [10, 11, 61, 62], or fusion in [63], shows how a fold followed by another function can be expressed just as a fold provided that certain conditions are met. For example, consider the datatype of lists:

$$\mathbf{data}\,[\alpha] \;=\; [\,] \mid \alpha : [\alpha]$$

This datatype has constructors $[\,]$ and $(:)$. The fold function over lists is named *foldr* in Haskell; $\mathit{foldr}\ (\oplus)\ e\ \mathit{xs}$ replaces all occurrences of $[\,]$ and $(:)$ in xs with e and $(\oplus)$ respectively:

$$\begin{aligned}\mathit{foldr}\ (\oplus)\ e\ [\,] \;&=\; e\\ \mathit{foldr}\ (\oplus)\ e\ (x : \mathit{xs}) \;&=\; x \;\oplus\; (\mathit{foldr}\ (\oplus)\ e\ \mathit{xs})\end{aligned}$$

So for example,

$$\mathit{foldr}\ (\oplus)\ e\ (1 : (2 : (3 : [\,]))) \;=\; 1 \;\oplus\; (2 \;\oplus\; (3 \;\oplus\; e))$$

(where $1 : (2 : (3 : [\,]))$ is the list $[1, 2, 3]$ written out in long form).

Now, consider the expression $f\ (\mathit{foldr}\ (\oplus)\ e\ [1, 2, 3])$. To express this directly as a fold over the list $[1, 2, 3]$, we need to find $(\otimes)$ and e' such that:

$$f\ (1\ \oplus\ (2\ \oplus\ (3\ \oplus\ e)))\ =\ 1\ \otimes\ (2\ \otimes\ (3\ \otimes\ e'))$$

One way that this might be true is if we have the following:

$$\begin{aligned} \forall x, y. f\ (x \oplus y) &= x \otimes (f\ y) \\ f\ e &= e' \end{aligned}$$

The first of these conditions allows us to "push" f through the series of $\oplus$s, and the second allows us to replace the resulting $f\ e$ with e', giving this derivation:

$$\begin{aligned} & f\ (1\ \oplus\ (2\ \oplus\ (3\ \oplus\ e))) \\ = & \quad \{\text{first condition}\} \\ & 1\ \otimes\ f\ (2\ \oplus\ (3\ \oplus\ e)) \\ = & \quad \{\text{first condition}\} \\ & 1\ \otimes\ (2\ \otimes\ f\ (3\ \oplus\ e)) \\ = & \quad \{\text{first condition}\} \\ & 1\ \otimes\ (2\ \otimes\ (3\ \otimes\ f\ e)) \\ = & \quad \{\text{second condition}\} \\ & 1\ \otimes\ (2\ \otimes\ (3\ \otimes\ e')) \end{aligned}$$

There is nothing special about the values 1, 2 and 3 or indeed about the length of the list; thus we have a general promotion law for lists (which can easily be proved by induction):

$$\begin{aligned} f\ (\mathit{foldr}\ (\oplus)\ e\ \mathit{xs}) &= \mathit{foldr}\ (\otimes)\ e'\ \mathit{xs} \\ \text{if} \quad & f \text{ strict} \\ & f\ e = e' \\ & \forall x, y\ :\ f\ (x \oplus y) = x \otimes (f\ y) \end{aligned}$$

The strictness condition on f is required to make this law correct for lazy languages such as Haskell in the case where *xs* is an infinite list. Otherwise, we might transform a program that did terminate (because the *foldr* on the left-hand side of the rule was never evaluated) into one that did not terminate (because the *foldr* on the right-hand side will always be evaluated and it is strict in its list argument). We can express *reverse* in terms of *foldr*:

$$\mathit{reverse}\ \mathit{xs}\ =\ \mathit{foldr}\ (\lambda\ a\ \mathit{as}\ .\ \mathit{as}\ +\!\!+\ [a])\ [\,]\ \mathit{xs}$$

The same is true for the efficient definition of *fastreverse*, but the result is a little more complicated; in this case *foldr* will return a function which is then applied to the accumulating parameter *ys*:

$$\mathit{fastreverse}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{foldr}\ (\lambda\ a\ f\ \mathit{as}\ .\ f\ (a : \mathit{as}))\ (\lambda\ a\ .\ a)\ \mathit{xs}\ \mathit{ys}$$

Now, recall the specification of *fastreverse*:

$$\mathit{fastreverse}\ \mathit{xs}\ \mathit{ys}\ =\ \mathit{reverse}\ \mathit{xs}\ +\!\!+\ \mathit{ys}$$

If we express *reverse* as a *foldr* as shown above, then since *fastreverse* can be expressed as a function applied to *reverse* we should be able to apply promotion to derive the form above.

Of course, manually expressing *reverse* as a *foldr* would be rather inconvenient, especially since the process should be essentially mechanical. Conveniently, promotion comes to our aid once more. Consider the expression *foldr* (:) [ ] *xs*; this expression evaluates to *xs* with all occurrences of (:) replaced by (:) and [ ] replaced by [ ]. In other words, *foldr* (:) [ ] is equivalent to the identity function on lists. Thus, *reverse xs* can equivalently be written as *reverse* (*foldr* (:) [ ] *xs*), and the promotion law can then be applied to derive a definition for *reverse* directly in terms of *foldr*.

In fact, this application of promotion can be merged with the use that derives the efficient form for *fastreverse* which we have already discussed – we show exactly how this works in Section 1.2.1, when we describe the mechanisation of this process.

Thus, deriving the efficient version of *fastreverse* is reduced to the problem of applying promotion to the expression *fastreverse* (*foldr* (:) [ ] *xs*) *ys*, making use of various definitions and the associativity of the ($+\!\!+$) operator. Indeed, we could define *reverse* in the following style:

$$\begin{array}{ll}
\multicolumn{2}{l}{\textit{reverse}\ [\,] \;=\; [\,]} \\
\multicolumn{2}{l}{\textit{reverse}\ (x : \textit{xs}) \;=\; \textit{reverse xs} \mathbin{+\!\!+} [x]} \\
\quad\textbf{transform} & \textit{reverse xs} \;=\; \textit{fastreverse xs}\ [\,] \\
\quad\textbf{where} & \textit{fastreverse xs ys} \;=\; \textit{reverse xs} \mathbin{+\!\!+} \textit{ys} \\
\quad\textbf{with} & \text{list promotion} \\
 & \text{definition of } \textit{reverse} \\
 & \text{definition of } (\mathbin{+\!\!+}) \\
 & \forall\, \textit{xs},\ \textit{ys},\ \textit{zs}\ :\ (\textit{xs} \mathbin{+\!\!+} \textit{ys}) \mathbin{+\!\!+} \textit{zs} \;=\; \textit{xs} \mathbin{+\!\!+} (\textit{ys} \mathbin{+\!\!+} \textit{zs})
\end{array}$$

This program is of course significantly longer than simply writing the optimised form of *fastreverse* directly, but it has its design steps explicitly documented. Cat-elimination is a "well-known" transformation, but the knowledge of how to apply it is generally communicated in an ad-hoc fashion. Here we have formalised the required information in a form which communicates it both to a programmer unfamiliar with cat-elimination and to the compiler. In the case of rather more complicated or specialised transformations such as that required for the *mindepth* example, the advantages of this form should be even more significant.

The correctness of each part can easily be verified – the original definition of *reverse* is obvious, and individual rules in the **with** clause are simple and can be checked independently of each other. Combining the new definition of *reverse* in terms of *fastreverse* and the specification of *fastreverse* immediately shows that together they leave the behaviour of *reverse* unaltered. This contrasts with the need to use some deep intuition or detailed reasoning to check the directly optimised program.

It is the mechanisation of the necessary transformation for this example and others like it that we shall address in the remainder of this thesis. In the following section, we explain the process of fully automatically transforming a program given in this way into the desired efficient version, and thus motivate the development of higher-order matching algorithms in Chapters 2, 3 and 4. Chapter 5 gives some details of how we have implemented these ideas, and in Chapter 6 we provide some rather more substantial examples of optimisations which can be given in this way and mechanised using our techniques.

1.2.1 Mechanisation

Automatically applying promotion can be done with *term rewriting*, a process of applying equationally correct laws to an expression. Consider for example the expression $1 + \mathit{square}\, 2$ and the law $\mathit{square}\, x = x * x$. Clearly we can use this law to write the expression as $1 + 2 * 2$, but consider the exact process required to do this. First, we search for a subexpression of $1 + \mathit{square}\, 2$ to which the rule applies. To make use of the rule, we must find an appropriate subexpression and a value for x, the only free variable in the rule. To do this, we *match* the left-hand side of the rule – $\mathit{square}\, x$ in this case – against each subexpression of $1 + \mathit{square}\, 2$. Matching two expressions is the process of searching for a substitution that when applied to the first expression produces exactly the second expression. Thus, in this case we will find the subexpression $\mathit{square}\, 2$ and the substitution $x := 2$. To apply the rule, we simply apply the substitution we found to the right-hand side of the rule, giving $2 * 2$, and replace the original subexpression with this expression to give $1 + 2 * 2$.

This procedure is known as *first-order* term rewriting. A good introduction to the theory behind it can be found in [4]. However, it is not adequate for our purposes, for two reasons.

Firstly, promotion rules have *side conditions* which must be verified. Thus, the above procedure must be extended; after matching the left-hand side of the rule against an appropriate subexpression, we first check all the side conditions before replacing the subexpression with the right-hand side of the rule (appropriately instantiated). Indeed, as with the promotion rule, sometimes the right-hand side will contain some free variables that are not on the left-hand side, values for which have to be found during the process of verifying the side conditions. Recall the promotion rule for lists we gave above:

$$\begin{array}{rcl} f\ (\mathit{foldr}\ (\oplus)\ e\ \mathit{xs}) & = & \mathit{foldr}\ (\otimes)\ e'\ \mathit{xs} \\ \text{if} & & f \text{ strict} \\ & & f\ e = e' \\ & & \forall x, y. f\ (x \oplus y) = x \otimes (f\ y) \end{array}$$

The variables $(\otimes)$ and e' occur on the right-hand side of the main rule but not the left-hand side, and assignments for these variables will be generated during verification of the second and third side conditions respectively.

The second complication is the nature of the free variables in promotion rules. In the example above, x was a variable that took on a value of type *Int*, a *base* type. However, the promotion rule is rather more complicated – it contains various free variables of *function* type. In particular, to generate an appropriate substitution for $(\otimes)$ it turns out that we will have to invent a completely new function.

It is this last difficulty that forms the basis for the work in this thesis. The process of inventing completely new functions during matching is known as *higher-order* matching, and we give two new algorithms for this that represent an advance on existing work in the field and are particularly useful for applying promotion rules. As we described above, the fast reverse derivation can be achieved using promotion by starting with the expression *fastreverse* (*foldr* (:) [ ] *xs*) *ys*. We shall now sketch the procedure a mechanical system would use to do this; we give more details of our implementation of this in Chapter 5.

Recalling once more the promotion rule for lists, the first step is to find an appropriate subexpression of our original expression, and match the left-hand side of the promotion rule against it. It turns out that the correct subexpression to choose is *fastreverse* (*foldr* (:) [ ] *xs*); here we are making use of *currying* by giving the *fastreverse* function only some of its arguments. A mechanical system can find that this is the subexpression to use by exhaustive search.

Matching then gives us the assignments ($f \ := \ \mathit{fastreverse}$), ($(\oplus) \ := \ (:)$) and ($e \ := \ [\,]$). We now verify the side conditions in order. Strictness of *fastreverse* follows from strictness of *reverse*. For the second condition, we instantiate the left-hand side of the condition using the substitution we have already calculated, giving $\mathit{fastreverse}\ [\,]$. Since the right-hand side of the rule is just e', we could use this value to give an appropriate substitution for it, but we can do better than this – we can apply term rewriting to this expression, using rules obtained from the specification of *fastreverse*, the definition of *reverse* and other auxiliary functions such as $(+\!\!+)$, and any other rules the user might supply, to give a simplified form. In this instance, we carry out the following derivation. Notice that our use of expressions in curried form means that we on occasion need to η-expand by adding in a missing argument; in this case the expression $(+\!\!+)\ [\,]$ is expanded to the equivalent $\lambda zs.[\,] \ +\!\!+ \ zs$ in the final step of the derivation.

$$\begin{array}{ll}
 & \mathit{fastreverse}\ [\,] \\
= & \quad \{\text{specification of } \mathit{fastreverse}\} \\
 & (+\!\!+)\ (\mathit{reverse}\ [\,]) \\
= & \quad \{\text{definition of } \mathit{reverse}\} \\
 & (+\!\!+)\ [\,] \\
= & \quad \{\eta\text{-expand by adding an argument } zs\text{, definition of } (+\!\!+)\} \\
 & \lambda zs\ .\ zs
\end{array}$$

Having reached a value which cannot be simplified further, we match the right-hand side of the rule against this value to give ($e' \ := \ \lambda zs\ .\ zs$), and we extend the substitution we have already obtained with this assignment.

Moving on to the third condition, we first notice that we can express the universal quantification over the variables x and y in terms of equality of functions. Thus, this side condition is equivalent to

$$\lambda x\ y.f\ (x \ \oplus \ y) \ = \ \lambda x\ y.x \ \otimes \ (f\ y)$$

This useful simplification saves us from having to treat universal quantifiers in a special way.

It is here that we find that our policy of rewriting the left-hand side after instantiating it is crucial. Examining the right-hand side, the one unknown quantity appearing there is the function $(\otimes)$. Unfortunately, it will not be straightforward to obtain a *definition* for $(\otimes)$. It appears with two arguments, one of which is the bound variable x and the other is the expression $f\ y$. Since y is also a bound variable, it cannot appear in any substitution for $(\otimes)$, and so to give a definition of $(\otimes)$, we will need to find a form for the left-hand side that only makes use of y in the form $f\ y$.

To do this, we apply term rewriting as follows, renaming y to the more appropriate xs for the purposes of presentation here. Notice the use of the associativity of $(+\!\!+)$; this is the crucial fact that any cat-elimination transformation makes use of, and it is thus natural that it should be one of our rewriting rules.

$$\begin{array}{ll}
& \lambda x\ xs\ .\ \mathit{fastreverse}\ (x : xs) \\
= & \quad \{\text{specification of } \mathit{fastreverse}\} \\
& \lambda x\ xs\ .\ (+\!\!+)\ (\mathit{reverse}\ (x : xs)) \\
= & \quad \{\text{definition of } \mathit{reverse}\} \\
& \lambda x\ xs\ .\ (+\!\!+)\ (\mathit{reverse}\ xs\ +\!\!+\ [x]) \\
= & \quad \{\eta\text{-expand, associativity of } (+\!\!+)\} \\
& \lambda x\ xs\ zs\ .\ \mathit{reverse}\ xs\ +\!\!+\ ([x]\ +\!\!+\ zs) \\
= & \quad \{\text{definition of } +\!\!+\} \\
& \lambda x\ xs\ zs\ .\ \mathit{reverse}\ xs\ +\!\!+\ (x : zs)
\end{array}$$

The occurrence of xs is not quite in the form of *fastreverse* xs that we might expect we need from looking at the right-hand side of the side condition. However, we can instantiate the occurrence of f in the right-hand side and rewrite this also:

$$\begin{array}{ll} & \lambda x\ xs\ .\ x \otimes (\mathit{fastreverse}\ xs) \\ = & \quad \{\text{specification of } \mathit{fastreverse}\} \\ & \lambda x\ xs\ .\ x \otimes (+\!\!+ (\mathit{reverse}\ xs)) \end{array}$$

We are now left with the problem of matching the expression

$$\lambda x\ xs\ .\ x \otimes (+\!\!+ (\mathit{reverse}\ xs))$$

against

$$\lambda x\ xs\ zs\ .\ \mathit{reverse}\ xs +\!\!+ (x : zs)$$

Inspecting this matching problem, we find that $(\otimes) := \lambda a\ b\ ys\ .\ b\ (a : ys)$ is the required assignment for $(\otimes)$, the only free variable in the first expression. It is the problem of finding such assignments automatically that this thesis will address.

We can combine all the assignments we have found and substitute back into the original rule to give a new subexpression:

$$\mathit{foldr}\ (\lambda a\ b\ zs\ .\ b\ (a : zs))\ (\lambda zs\ .\ zs)$$

Finally, using this to replace the original subexpression in the expression we started from gives us the required:

$$\mathit{foldr}\ (\lambda a\ b\ zs\ .\ b\ (a : zs))\ (\lambda zs\ .\ zs)\ ys$$

Crucially, note that it was not necessary to apply any equations or definitions in more than one direction. Thus, promotion offers us a safe way of mechanising the fast reverse derivation; by eliminating the need for a folding step, it allows the derivation to be accomplished using a set of rewrite rules that will never produce an infinite loop in our transformer. The same is true for other similar derivations, such as minimum depth; we give several examples in Chapter 6.

1.3 Fusion

Promotion is just one example of *fusion*, the process of merging the definitions of two or more functions to provide a definition of a single function that is equivalent to the sequential composition of those functions. The hope is that this single function will be somehow more efficient than applying the original functions. It should be noted that there is some ambiguity in the literature about these terms; what we describe as promotion has sometimes itself been referred to as fusion. However, it appears that the majority of work assumes the more general definition for fusion that we give, and since we wish to clearly disambiguate the two we have also followed this approach. Below we survey previous work on fusion and then focus on deforestation and short-cut deforestation, two techniques that are particularly relevant to this thesis.

Most commonly, a more efficient function can be derived by eliminating the production of intermediate data structures; if one function produces a structure which a second function then takes apart, a definition that acts on the elements of the structure as it is produced without ever explicitly allocating or deallocating the storage necessary will provide a performance benefit. Automatic techniques for eliminating intermediate lists were first put forward by Wadler [94, 95] and later generalised to handle all algebraic data structures and named *deforestation* [99].

In order to guarantee the termination of deforestation, some quite tight syntax restrictions were imposed, which Chin loosened in [17, 18] and then in [19] gave a means of syntactically preprocessing programs to further extend the (safe) applicability of deforestation. An alternative relaxation was given by Sørenson with a semantic based analysis [85].

Gill et al. [37] give a modified approach known as *short-cut* deforestation which has been implemented in the Glasgow Haskell Compiler (GHC). Their method removes Wadler's guarantee of removing all intermediate lists in exchange for a simpler algorithm that is applicable to all programs. Only parts of the program that produce lists using a higher-order function known as *build* and consume them using *foldr* are deforested, but the entire prelude (the standard library for Haskell) is written in this style wherever possible to make this quite common. One other disadvantage (which is shared with other approaches) is that function definitions must be inlined to make programs susceptible to deforestation, and this is inconvenient to do across module boundaries. In [20], Chitil gives a type-based analysis that allows some functions to be split into *worker* and *wrapper* parts, with only the wrapper section needing to be inlined to allow deforestation to be applied.

Since people do not generally write their programs in terms of *foldr* and *build*, short-cut deforestation is unlikely to be particularly helpful at optimising user-supplied functions. Launchbury and Sheard [59] give an algorithm for *warm* fusion, which uses the promotion theorem on an identity fold as we described earlier to automatically derive appropriate forms from recursive programs. They use a heuristic to determine appropriate places in the program to try this. This has been implemented by Johann and Visser [52] in the *Stratego* system [93], and by Nemeth [68] as an optimising pass for GHC.

Another method of automatically applying fusion is described by Onoue et al. in [71]. Recursive functions are first expressed in terms of *hylomorphisms* [63], then manipulated using the *Acid Rain Theorems* [88], which generalise the rule used in short-cut deforestation, and a "shifting" law which allows computations to be moved around within the definition of a particular hylomorphism. This technique has been implemented as an optimising pass to GHC, with encouraging results.

Other work involving promotion has been carried out by Sheard and Fegaras in [82], which gives a *normalisation* algorithm to automatically improve programs based on folds, and in [34] Fegaras et al. extend this work by giving an extended version of folds for programs which recurse over multiple data structures simultaneously, and provide a generalised promotion theorem for fusing these extended folds.

Deforestation and promotion are distinct transformations; in general deforestation applies to expressions where the outer function consumes a data structure produced by the inner function, whereas promotion requires that the inner function be a fold and makes use of subtle properties that show how the computation in outer function can be broken up and merged with the individual operations of the fold. They happen to overlap in some cases, such as the example of calculating the sum of the squares of the elements of a list that we shall explain in the following section when we discuss deforestation in more detail, but in general this is not true – the *mindepth* example from Section 1.2 is an example of promotion but not deforestation, whereas the *foldr-build* fusion of short-cut deforestation is not an instance of promotion.

Supercompilation [90] is a very powerful technique (but currently with few practical implementations) which generalises both deforestation and *partial evaluation* [54]. Partial evaluation focuses on specialising a general program once specific values have been given for some of its input; supercompilation takes this a stage further, generating an entirely new program by symbolically tracing the execution of the original one.

1.3.1 Deforestation

Consider the program

$$sumsq\ xs \;=\; sum\ (map\ square\ xs)$$

The expression *map square xs* generates a list of results which is immediately consumed by *sum*. More efficient would be the following:

$$\begin{aligned} &sumsq\ [\,] \;=\; 0 \\ &sumsq\ (x : xs) \;=\; square\ x \;+\; sumsq\ xs \end{aligned}$$

Fusing the definitions of *sum* and *square* has allowed the construction and destruction of the intermediate list to be completely eliminated; as each new element is generated, it is immediately used and thus there is no need to explicitly create and destroy the constructors that "glue" the list together. Applying this transformation across a large program can produce quite large speedups, because the natural style employed by many functional programmers involves writing in a modular fashion and passing numerous data structures between various parts of the program.

The standard method of applying deforestation is to unfold the expression to be deforested using the relevant definitions. If an intermediate data structure in that expression can be eliminated, this unfolding should expose the point at which it generated and then immediately consumed, allowing these operations to be removed.

For example, consider the above program. For simplicity, assume that the definitions of *sum* and *map* are given as **case** expressions:

$$\begin{aligned} \mathit{sum}\ \mathit{xs} \ = \ & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & \quad [\,] \qquad\quad \to 0 \\ & \quad (x : \mathit{xs}) \ \to x \ + \ \mathit{sum}\ \mathit{xs} \end{aligned}$$

$$\begin{aligned} \mathit{map}\ f\ \mathit{xs} \ = \ & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & \quad [\,] \qquad\quad \to [\,] \\ & \quad (x : \mathit{xs}) \ \to f\ x \ : \ \mathit{map}\ f\ \mathit{xs} \end{aligned}$$

From this, we obtain the following derivation:

$$\begin{aligned} & \mathit{sumsq}\ \mathit{xs} \\ = \quad & \{\text{definition of } \mathit{sumsq}\} \\ & \mathit{sum}\ (\mathit{map}\ \mathit{square}\ \mathit{xs}) \\ = \quad & \{\text{definition of } \mathit{map}\} \\ & \mathit{sum}\ (\ \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & \qquad\quad [\,] \qquad\quad \to [\,] \\ & \qquad\quad (x : \mathit{xs}) \ \to \mathit{square}\ x \ : \ \mathit{map}\ \mathit{square}\ \mathit{xs}\ ) \\ = \quad & \{\text{(strict) functions distribute through } \textbf{case}\} \\ & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & \quad [\,] \qquad\quad \to \mathit{sum}\ [\,] \\ & \quad (x : \mathit{xs}) \ \to \mathit{sum}\ (\mathit{square}\ x \ : \ \mathit{map}\ \mathit{square}\ \mathit{xs}) \end{aligned}$$

$$
\begin{array}{ll}
= & \{\text{definition of } \mathit{sum}\} \\
& \textbf{case } \mathit{xs} \textbf{ of} \\
& \quad [\,] \quad \rightarrow \textbf{case } [\,] \textbf{ of} \\
& \qquad\qquad [\,] \qquad \rightarrow 0 \\
& \qquad\qquad (y : \mathit{ys}) \rightarrow y + \mathit{sum\ ys} \\
& \quad (x : \mathit{xs}) \rightarrow \textbf{case } (\mathit{square\ x} : \mathit{map\ square\ xs}) \textbf{ of} \\
& \qquad\qquad [\,] \qquad \rightarrow 0 \\
& \qquad\qquad (y : \mathit{ys}) \rightarrow y + \mathit{sum\ ys} \\
= & \{\text{definition of } \textbf{case}\} \\
& \textbf{case } \mathit{xs} \textbf{ of} \\
& \quad [\,] \qquad \rightarrow 0 \\
& \quad (x : \mathit{xs}) \rightarrow \mathit{square\ x} + \mathit{sum}\ (\mathit{map\ square\ xs}) \\
= & \{\text{definition of } \mathit{sumsq}\} \\
& \textbf{case } \mathit{xs} \textbf{ of} \\
& \quad [\,] \qquad \rightarrow 0 \\
& \quad (x : \mathit{xs}) \rightarrow \mathit{square\ x} + \mathit{sumsq\ xs}
\end{array}
$$

It is the last two steps of this derivation that illustrate the most important aspects of deforestation. The first of these, which applies the definition of **case** by making use of the known structure of the argument to select one particular result, is what deforestation is about – it is here that the intermediate list is actually eliminated from our program. The last of these steps applies the definition of *sumsq* in reverse, and thus prevents the unfolding from continuing infinitely. In general, ensuring termination in such a way can be difficult, and it is this issue that tends to make automatic deforestation quite complex.

1.3.2 Short-cut deforestation

The idea behind short-cut deforestation is that although in general deforestation can be difficult to apply in a safe manner, it should be easy to remove intermediate data structures from certain parts of our program by writing those parts in terms of specific predefined higher-order functions. The compiler can be given special knowledge of these functions, allowing it to safely and quickly deforest them.

In [37], this idea is applied only to lists, the most common intermediate structure found in Haskell programs. The higher-order functions in question are *foldr*, which we have already described in Section 1.2, and *build*, which we shall define shortly. The idea is that functions which create lists should be defined in terms of *build*, and those which consume lists should be defined in terms of *foldr*. For those which do both, such as *map*, or those which consume two lists at once, such as *zip*, we shall have to choose one option, which will limit the applicability of short-cut deforestation.

Just as *foldr* encapsulates the pattern of (recursive) list consumption, *build* encapsulates the pattern of list generation. Its definition is deceptively simple:

$$\mathit{build}\ g \ = \ g\ (:)\ [\,]$$

In other words, g should be a function that constructs its output list using its first two parameters for the constructors of that list. Recall that *foldr* replaces the constructors of a list with the functions supplied to it; thus, the following should hold:

$$\mathit{foldr}\ f\ e\ (\mathit{build}\ g) \ = \ g\ f\ e$$

In other words, instead of first constructing the list with (:) and [ ] and then replacing these constructors with f and e, we can directly construct it using f and e.

There is a slight complication; we have no guarantee that the argument g to *build* will actually construct its output list using the supplied parameters. It could simply make direct use of (:) and [ ], which are after all globally accessible constructors. However, *foldr* does not “care” where the constructors in the list it consumes came from, and thus if g does not behave as expected the above equation will not hold.

We refer the reader to [37] for the precise details of how this is solved; essentially *build* is given a carefully constructed polymorphic type which ensures that g is forced to use its arguments to construct its output; the above law follows as a consequence of the appropriate *free theorem* [98]. The definition of *build* then becomes the following:

$$\begin{aligned} &\mathit{build} :: \forall\, \alpha\, .\, (\forall\, \beta\, .\, (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha] \\ &\mathit{build}\ g \ =\ g\ (:)\ [\,] \end{aligned}$$

Informally, the universal quantification of the type variable β in the type of the argument g is what provides the necessary guarantee; the body of g cannot "know" anything about the structure of β and is thus forced to use the supplied arguments to construct values of this type.

For example, consider the *sumsq* example from above. We can write *sum* as a fold and map as a build:

$$\begin{aligned} &\mathit{sum}\ \mathit{xs}\ =\ \mathit{foldr}\ (+)\ 0\ \mathit{xs} \\ &\mathit{map}\ f\ \mathit{xs}\ =\ \mathit{build}\ g \\ &\quad \textbf{where}\ g\ \mathit{cons}\ \mathit{nil}\ =\ h\ \mathit{cons}\ \mathit{nil}\ \mathit{xs} \\ &\qquad\qquad h\ \mathit{cons}\ \mathit{nil}\ [\,]\ =\ \mathit{nil} \\ &\qquad\qquad h\ \mathit{cons}\ \mathit{nil}\ (y : \mathit{ys})\ =\ \mathit{cons}\ (f\ y)\ (h\ \mathit{cons}\ \mathit{nil}\ \mathit{ys}) \end{aligned}$$

Applying the *foldr*-*build* rule from above to *sumsq* then immediately gives us the optimised program.

Short-cut deforestation was originally implemented completely internally to GHC, both because there was no way for the user to specify optimisations, and because the "internal" universal quantifier in the definition of *build* could not be expressed in the standard Hindley-Milner [66] type system that Haskell was then based on. Functions in the prelude were defined in terms of *foldr* and *build* wherever possible, so user programs written in terms of them could be deforested, but the user could not define his or her own functions using *build*, and user-defined datatypes could not be deforested at all.

Since then, the type system of Haskell has become richer, and recently it has become possible to define simple rewrite rules for GHC in Haskell source files by adding appropriate annotations [77], which are quite similar (albeit rather less sophisticated) to those we proposed on page 17. As a result, it is now possible for short-cut deforestation to be completely defined using mechanisms available to the user, and appropriate folds and builds can be defined (along with the necessary rewrite rule) to allow user-defined datatypes to be deforested.

Chapter 2

Preliminaries

2.1 The matching problem

Pattern matching has been widely studied in a variety of contexts, most notably with regard to strings [1, 57], trees [41], and graphs [40]. Here, we are concerned with a specific variation of tree pattern matching that applies specifically to λ-expressions. The *higher-order* nature of this pattern matching arises from the fact that we wish to synthesise new functions as part of the process. To avoid confusion with a distinct definition of *higher-order patterns* that can be found in the literature (which we discuss later), we shall henceforth refer to pattern matching as simply "matching".

At its most general, the matching problem we are interested in can be stated as follows. Given arbitrary λ-expressions P and T, known as the *pattern* and *term* respectively, find a substitution ϕ such that ϕP is equivalent to T.

Here, "is equivalent to" has intentionally not been precisely defined; λ-expressions can be equated to each other using three types of equivalence:

- α-conversion. Bound variables can be freely renamed; for example, $\lambda x.x + x = \lambda y.y + y$.

- β-reduction. λ-bound variables can be replaced by the appropriate argument to the λ-abstraction. For example, $(\lambda x.x + x)\,1 = 1 + 1$. This rule can also be applied in reverse.

- η-conversion. Superfluous combinations of λ-abstraction and function application can be removed; $\lambda x.E\,x$ is equivalent to E, so long as x is not free in E. For example, $\lambda x.f\,x = f$, but $\lambda x.g\,x\,x \neq g\,x$.

Thus, the fully general matching problem requires us to find all substitutions ϕ such that ϕP and T are equivalent under all three of the above rules. Although this is significantly easier than the related problem of *unification*, which involves finding all ϕ such that ϕP and ϕT are equivalent, it is still a hard problem, partly because the set of substitutions is potentially infinite. For example, consider the pattern $p\,q$ and the term 0; then $p := \lambda y.y^n\,0$, $q := \lambda z.z$ is a valid substitution for any n greater than or equal to 0.

The usual approach taken in the literature is to choose a particular type discipline, often from Barendregt's λ-cube [7] and investigate solutions of the matching problem within that discipline. Usually, the problem is then further restricted by *order*; terms of ground type such as *Int* or *Char* (or a base type variable in the case of polymorphic type systems) are known as *first-order*, functions which take first-order terms as parameters are *second-order*, and so on. This notion is extended to matching by defining that the order of a substitution is the maximum order of the terms in its range; then *nth-order matching* is the problem of finding all appropriate substitutions of order n or less.

For the simply-typed λ-calculus, the decision problem – finding whether or not matches exist, without actually enumerating them – was posed in 1976 by Gerard Huet [45], but is as yet unsolved (in contrast, general unification is known to be undecidable [39, 43]).

Wolfram [102] gives a procedure is given for finding solutions for the simply-typed case; it is shown to be sound and terminating (infinite branches of the search tree which would give sets of solutions similar to those described above are detected and truncated); it is also conjectured to be complete (modulo this truncation of infinite sets of results), and if this conjecture were to be proved then the matching problem would also be decidable. It was claimed in 1984 that higher-order matching is decidable [55], but no proof was provided, and the existence of infinite sets of matches would seem to contradict the claim made there that the matching process converges.

First-order matching is decidable, and an algorithm for finding all solutions (of which there is at most one for any given pattern and term) is easy to implement. Second-order matching is also decidable, with a finite set of solutions; an algorithm for finding these was given by Huet and Lang in 1978 [47]. Third- and fourth-order matching are both decidable [31, 72] but the set of solutions is potentially infinite – algorithms to find a *representation* of this set have been given [22]. Finally, a restricted version of the fifth-order problem is decidable [81].

Going beyond simple typing, matching is undecidable in all the other systems of the λ-cube [32]. However, there is still an algorithm for generating all solutions to the second-order problem in the presence of polymorphic and dependent types, and type constructors [30, 69]. Third-order matching is also decidable for polymorphically-typed terms [86] and those with type constructors [87], but undecidable in the presence of dependent types [29].

Another restriction that has been studied is limiting both pattern and term to be *linear λ-terms*. These are closely related to Girard's *linear logic* [38]; they restrict all abstractions $\lambda x.B$ contained in the expression so that B contains precisely one occurrence of x. The matching problem for such expressions is known to be decidable and an algorithm exists for calculating them [25].

Finally, the notion of *higher-order patterns* has been explored in the context of unification. These restrict expressions such that every free variable should appear together with a list of distinct bound variables as parameters. General unification for these expressions is known to be decidable with a single most general solution [65], and a practical implementation has been given for finding these [70]. These results are also applicable to matching by simply treating all free variables in the term as constants; this will also have the consequence that any term will trivially satisfy the restrictions.

For our purposes, a practical program transformation system that makes use of term rewriting requires a terminating matching algorithm with a finite set of results. Second-order results turn out not to be adequate for many of the problems we would like to be able to solve, and likewise the restrictions imposed by either linearity and higher-order patterns rule out many of these problems. Since third-order matching can produce an infinite set of results, this thesis takes the approach of investigating a completely new restriction on the problem.

Consider the equivalence rules for λ-expressions stated above; it is the β-reduction rule that introduces difficulties to the matching process. If we ignore it entirely, it is then straightforward to give a sound and complete *simple* matching algorithm; apply exhaustive η-reduction (a procedure that is guaranteed to terminate) to both pattern and term, and then compare the two by structural recursion over each of them simultaneously, renaming bound variables and creating substitutions for free variables as necessary. In Section 2.6, we give an implementation of this algorithm as an example of using the framework we shall present shortly.

Since the gap that remains between general matching and this "simple" matching is the difference between allowing full β-reduction and not permitting it at all, it makes sense to explore restricted forms of this rule instead, and it is this idea that forms the main thrust of this thesis. We defer discussion of the exact nature of the restrictions we will impose until after we have introduced some notation.

The other way our algorithms will differ from those in the literature is that we do not impose any typing discipline on expressions. Thus, our matching algorithms will be specified and implemented by essentially syntactic manipulation. They manipulate terms in η-contracted form rather than the more common η-long form; η-contraction is naturally bounded whereas η-

expansion requires type-based restrictions to enforce termination. The benefit we gain from this is that our algorithms are more widely applicable; users of our algorithms are not constrained to languages that use an appropriate type system. This is likely to be particularly valuable when transforming languages such as Haskell, for which new extensions for the type system are regularly suggested and sometimes implemented. It would be rather inconvenient if it was necessary to recheck the correctness of the matching algorithm each time this happened. In addition, the use of η-contracted normal forms means that if using a complex type system we are spared the complexities that η-expansion brings [33, 36].

2.2 Notation

2.2.1 Expressions

Expressions are built recursively from variables, constants, λ-abstractions and function applications. Variables are either *local*, indicating that they refer to a binding λ and can only be substituted when being replaced by the parameter of that λ, or *pattern*, indicating that they can be freely substituted for. We use local and pattern instead of the more traditional *bound* and *free* because we shall diverge slightly from standard usage; In particular, when recursively traversing expressions, a variable shall continue to be considered local even if the corresponding binding λ is no longer present. Thus, in the body $x + x$ of the λ-abstraction $\lambda x.x + x$, x is a local variable.

To make the presentation simpler, we shall use notation to make distinctions between kinds of expressions:

- a, b, c will represent constants.
- p, q, r will represent pattern variables.
- x, y, z will represent local variables.
- Capital letters will represent arbitrary expressions.

This simplification saves the clutter that using de Bruijn notation [24] or explicit environments would introduce.

Manipulating λ-expressions often introduces the problem of *variable capture*, in which a local variable is substituted into a term which already contains a λ-abstraction involving a variable of the same name. Thus, we assume that fresh identifiers are introduced as needed or existing variables implicitly renamed.

Function application is a binary operator, and is written as a space. Applications with multiple parameters are expressed by repeated uses of this operator; it is defined to be left-associative and thus $F\,E_1\,E_2$ represents the function F applied to the arguments E_1 and E_2 in that order. This expression is also an example of the use of *currying*; the expression $F\,E_1$ gives a function which is then applied to E_2. λ-abstraction is written $\lambda x.B$ to represent the function that when applied to an argument E returns B with all occurrences of x replaced by E.

The equality operator "$=$" is defined to be equality modulo α-conversion only. The operator "$\simeq$" is used to represent equality modulo α and η-conversion.

A *β-redex* is an expression of the form $(\lambda x.B)\,E$; the *β-contractum* of such an expression is the term B with all occurrences of x replaced by E. Similarly, an *η-redex* is one of the form $\lambda x.E\,x$, where E does not contain x, with *η-contractum* E.

An expression is *β-normal* if it contains no β-redexes and *η-normal* if it contains no η-redexes. It is *$\beta\eta$-normal* (sometimes referred to as just *normal*) if it contains neither. An expression is *closed* if it contains no pattern variables.

We write $E \xrightarrow{\beta} E'$ to signify that E' was obtained by reducing a single β-redex in E. The relation $\xrightarrow{\beta}^{*}$ is the reflexive transitive closure of $\xrightarrow{\beta}$.

The *head* of an expression is the expression itself if the expression is not an application, or the head of the function part if it is an application. An expression is said to be *flexible* if its head is either a pattern variable or a

λ-abstraction (we shall not make use of these notions until Section 3.4).

We shall sometimes comment on properties of our algorithms when the user does choose to use the simply-typed λ-calculus. In this context, the *order* of an expression E is the order of its type, defined as follows:

- A base type (*Int*, *Char* etc.) is of order 1.
- A function type $a \to b$ is of order $\mathit{max}(1 + \mathit{order}(a), \mathit{order}(b))$.

2.2.2 Subexpressions

An expression S is a *subexpression* of another expression E if S occurs somewhere within E; this is written $S \trianglelefteq E$. This also leads to a convenient notation for stating that S does not occur somewhere within E, which we write $S \ntrianglelefteq E$. A *direction* is either *Func*, representing the function part of an application, *Arg* to represent the argument part, or *Body* to represent the body of a λ-expression. A *location* is a sequence of directions used to indicate the position of a subexpression in the expression it was taken from, written using the syntax $\langle \mathit{Func}, \mathit{Func}, \mathit{Arg}, \mathit{Body} \rangle$. The empty sequence $\langle\rangle$ is the location representing the root of an expression, and $\frown$ is the operator which joins two sequences one after the other. For convenience we use $\mathit{dir}; \mathit{loc}$ as shorthand for $\langle \mathit{dir} \rangle \frown \mathit{loc}$.

2.2.3 Substitutions

We define *substitutions* as partial functions from pattern variables to closed, normal expressions. They are denoted by Greek identifiers; if we wish to give an explicit substitution this will be done by listing the assignments to variables in the domain of the substitution; so for example

$$\phi \quad = \quad (p := \lambda y. b\; y\; x)$$

makes the indicated assignment to p, but leaves all other variables unchanged. The identity substitution *idSubst* is defined as the substitution with an empty domain.

Substitutions are applied to expressions by replacing all instances of each pattern variable in their domain with the image of that pattern variable under the substitution.

Composition of substitutions ϕ and ψ is defined by first applying ψ and then ϕ:

$$(\phi \circ \psi)\, E \quad = \quad \phi\,(\psi E)$$

Composition is associative, and since expressions in the range of a substitution must be closed, if two substitutions have disjoint domains they will commute.

We say that one substitution ϕ is *more general* than another substitution ψ if there exists a third substitution δ such that

$$\psi \quad = \quad \delta \circ \phi$$

We write $\phi \leq \psi$ to indicate that ϕ is more general than ψ. Intuitively, when $\phi \leq \psi$, the larger substitution ψ substitutes for variables that ϕ leaves unchanged; if we viewed substitutions as sets of assignments to pattern variables, then $\leq$ would simply be set inclusion. For example, with ϕ as above and ψ specified by

$$\psi \quad = \quad (p := \lambda y.b\, y\, x,\;\, q := a\,(b\; c))$$

we have $\phi \leq \psi$ because $\psi = \delta \circ \phi$ where

$$\delta \quad = \quad (q := a\,(b\; c))$$

Lemma 2.1. *If two substitutions ϕ and ψ are equally general, they are identical.*

Proof. As we remarked above, we can view $\leq$ as set inclusion on sets of assignments, and set inclusion is antisymmetric. $\square$

In the special case of the application of β-reduction, we shall extend the substitution notation to allow local variables in the domain; thus we write the β-contractum of $(\lambda x.B)\, E$ as $(x := E)\, B$.

Within the context of the simply-typed λ-calculus, the order of a substitution is the maximum order of the terms in its range.

2.2.4 Meta-programs

We will express the specification and implementation of matching algorithms as meta-programs acting on expressions. We will use syntax similar to that of Haskell for these meta-programs, with the addition of the substitution notation described above. In particular, various definitions shall be given by *pattern matching*, in which a function is defined by a list of clauses, each of which may or may not apply to a particular call to the function. The right-hand side of the first one that does apply for any individual set of arguments is used to calculate the result of the function in that case. So for example, the following program to return the sign of a number treats 0 as a special case:

$$\begin{aligned} \mathit{sign}\, 0 &= 0 \\ \mathit{sign}\, x &= x \div \mathit{abs}\, x \end{aligned}$$

We will use the "$\equiv$" operator to signify equivalence of meta-programs (as well as of logical statements).

2.2.5 Rules

We now introduce the concept of *rules* to represent matching problems. A rule is a pair of expressions, written $P \rightarrow T$, where P is η-normal and T is closed and normal. We call P the *pattern* and T the *term* of the rule; the goal of matching will be to find a substition that makes pattern and term equal.

Rules are denoted by variables X, Y and Z, and sets of rules (also known as *rule sets*) by Xs, Ys and Zs. Our notation for rules bears some resemblance to that used for rewriting or transition systems, but should not be taken as such.

The *measure* of a rule is a pair of numbers: the first component is the number of distinct pattern variables in the pattern, and the second component is the total number of symbols in the pattern (where λ and the space representing function application are taken to be symbols, but brackets are not). The measure of a set of rules is also a pair of numbers, which are defined by summing the first and second components respectively of the measures of its elements. When Xs and Ys are sets of rules, we shall write $Xs \ll Ys$ to indicate that in the lexicographic comparison of pairs, the measure of Xs is strictly less than the measure of Ys. Note that $\ll$ is a well-founded transitive relation.

A substitution ϕ is said to be *pertinent* to a rule $(P \rightarrow T)$ if all variables in its domain are contained in P. Similarly, a substitution is pertinent to a set of rules if each variable in its domain is contained in the pattern of one of the rules (but not necessarily all in the same rule).

The application of a substitution to a rule is defined by $\sigma(P \rightarrow T) = \sigma P \rightarrow T$ (since T is closed applying a substitution to it would have no effect). The obvious extension of this definition to a set of rules applies.

The reason that we do not require β-normalness of the pattern in a rule is that substitution does not preserve this property; we wish to be able to apply substitutions to rules and have an immediate guarantee that the result is also a rule.

2.3 Specification

2.3.1 Beta-reduction

We remarked earlier that we would specify our algorithms by restricting the form of β-reduction allowed in our equality law. If we did allow full β-reduction, we would define that ϕ *satisfies* a rule $P \rightarrow T$ iff there existed β-conversions of ϕP and T that were related by $\simeq$. Since T is defined to be normal, and it is known that if repeated application of β-reduction on an expression terminates then it always terminates with the same result [6], we can define a function *betanormalise* that returns this result if it exists, and *undefined* if not, and recast this definition as:

$$\mathit{betanormalise}\,(\phi P) \quad \simeq \quad T$$

It is this form of the definition that we will vary, by replacing *betanormalise* with other functions that only partially apply β-reduction.

We can define the *betanormalise* function as a recursive treewalk (making use of our notational conventions that make a, b, c represent constants, x, y, z local variables, p, q, r pattern variables and capital letters any expression):

$$\begin{array}{rcl}
\mathit{betanormalise}\, c & = & c \\
\mathit{betanormalise}\, x & = & x \\
\mathit{betanormalise}\, p & = & p \\
\mathit{betanormalise}\, (\lambda x.E) & = & \lambda x.(\mathit{betanormalise}\, E) \\
\mathit{betanormalise}\, (E_1\, E_2) & = & \text{case } E_1' \text{ of} \\
 & & \quad (\lambda x.B) \;\rightarrow\; \mathit{betanormalise}\, ((x := E_2')B) \\
 & & \quad \_ \qquad\quad\; \rightarrow\; E_1'\, E_2' \\
 & & \text{where } E_1' \;=\; \mathit{betanormalise}\, E_1 \\
 & & \qquad\quad\;\; E_2' \;=\; \mathit{betanormalise}\, E_2
\end{array}$$

The function that applies no β-reduction at all is the identity function *id*. We can write it in a recursive form (entirely redundantly), following the same pattern as above:

$$\begin{array}{rcl}
\mathit{id}\, c & = & c \\
\mathit{id}\, x & = & x \\
\mathit{id}\, p & = & p \\
\mathit{id}\, (\lambda x.E) & = & \lambda x.(\mathit{id}\, E) \\
\mathit{id}\, (E_1\, E_2) & = & (\mathit{id}\, E_1)\, (\mathit{id}\, E_2)
\end{array}$$

Writing the two functions in this form allows us to make explicit the differences between them, namely their result in the case that the expression they are being applied to is a function application. We can parametrize this difference in the following definition of *reduce*, a higher-order function:

$$\begin{array}{rcl}
\mathit{reduce}\ \mathit{app}\ c & = & c \\
\mathit{reduce}\ \mathit{app}\ x & = & x \\
\mathit{reduce}\ \mathit{app}\ p & = & p \\
\mathit{reduce}\ \mathit{app}\ (\lambda x.E) & = & \lambda x.(\mathit{reduce}\ \mathit{app}\ E) \\
\mathit{reduce}\ \mathit{app}\ (E_1\ E_2) & = & \mathit{app}\ (\mathit{reduce}\ \mathit{app}\ E_1)\ (\mathit{reduce}\ \mathit{app}\ E_2)
\end{array}$$

The parameter *app* is a function that defines what happens to applications; for example, $\mathit{betanormalise} = \mathit{reduce}\,\mathit{full}$ and $\mathit{id} = \mathit{reduce}\,\mathit{none}$, where:

$$\begin{array}{rcl}
\mathit{full}\ (\lambda x.B)\ E & = & \mathit{reduce}\,\mathit{full}\ ((x := E)B) \\
\mathit{full}\ F\ E & = & F\ E \\
\mathit{none}\ F\ E & = & F\ E
\end{array}$$

We shall give more instances of *app* functions later in the context of the one-step and two-step matching algorithms. It should be noted that *app* should be a function whose role is to perform some (or maybe none) β-reduction and nothing else. Thus, in general we require that:

$$F\ E \overset{\beta}{\rightarrow}{}^{*}\ \mathit{app}\ F\ E$$

A consequence of this is that for any expression T, we have that:

$$T \overset{\beta}{\rightarrow}{}^{*}\ \mathit{reduce}\ \mathit{app}\ T$$

We can say something about β-redexes that *reduce* generates, namely that they can only be produced by *app*. This fact will be of use later when discussing properties of specific instances of *reduce*.

Lemma 2.2. *If*

$$(\lambda x.B)\ E \trianglelefteq \mathit{reduce}\ \mathit{app}\ T$$

then

$$\exists\, T_0, T_1\ :\ T_0\ T_1 \trianglelefteq T \wedge (\lambda x.B)\ E \trianglelefteq \mathit{app}\ (\mathit{reduce}\ \mathit{app}\ T_0)\ (\mathit{reduce}\ \mathit{app}\ T_1)$$

Proof. By case analysis on the definition of *reduce* and induction on the structure of T. □

2.3.2 Matches

For a given function *app*, a substitution ϕ *satisfies*, or *is a match for*, the rule $P \to T$ *with respect to app* iff

$$\mathit{reduce\ app}\,(\phi P) \quad \simeq \quad T$$

We write this as $\phi \vdash_{\mathit{app}} P \to T$, thus

$$\phi \vdash_{\mathit{app}} P \to T \equiv \mathit{reduce\ app}\,(\phi P) \simeq T$$

We also refer to substitutions as satisfying and being matches for sets of rules. We say that ϕ is a *general match* for the rule $P \to T$ if ϕ is a match for this rule with respect to *full* (recall that *betanormalise* = *reduce full*).

Lemma 2.3.

$$\phi \circ \psi \vdash_{\mathit{app}} P \to T \quad \equiv \quad \phi \vdash_{\mathit{app}} \psi(P \to T)$$

Proof.

$$\begin{array}{ll}
 & \phi \circ \psi \vdash_{\mathit{app}} P \to T \\
\equiv & \quad \{\text{definition of } \vdash_{\mathit{app}}\} \\
 & \mathit{reduce\ app}\,((\phi \circ \psi)P) \simeq T \\
\equiv & \quad \{\text{definition of } \circ\} \\
 & \mathit{reduce\ app}\,(\phi\,(\psi P)) \simeq T \\
\equiv & \quad \{\text{definition of } \vdash_{\mathit{app}}\} \\
 & \phi \vdash_{\mathit{app}} \psi P \to T \\
\equiv & \quad \{\text{definition of substitution on rules}\} \\
 & \phi \vdash_{\mathit{app}} \psi(P \to T)
\end{array}$$

□

Corollary 2.4.

$$\phi \circ \psi \vdash_{\mathit{app}} Xs \quad \equiv \quad \phi \vdash_{\mathit{app}} \psi Xs$$

Lemma 2.5. *If $\phi \leq \psi$, then*

$$\phi \vdash_{\mathit{app}} P \to T \quad \Rightarrow \quad \psi \vdash_{\mathit{app}} P \to T$$

Proof. Suppose that $\delta \circ \phi = \psi$. We argue as follows:

$$\begin{array}{ll} & \phi \vdash_{\mathit{app}} P \to T \\ \equiv & \quad \{\text{definition of } \vdash_{\mathit{app}}\} \\ & \mathit{reduce\ app}\,(\phi P) \simeq T \\ \Rightarrow & \quad \{\text{claim}\} \\ & \mathit{reduce\ app}\,(\delta\,(\phi P)) \simeq T \\ \equiv & \quad \{\text{value of } \psi\} \\ & \mathit{reduce\ app}\,(\psi P) \simeq T \\ \equiv & \quad \{\text{definition of } \vdash_{\mathit{app}}\} \\ & \psi \vdash_{\mathit{app}} P \to T \end{array}$$

□

We made a claim that if *reduce app* $(\phi P) \simeq T$, then we also have that *reduce app* $(\delta\,(\phi P)) \simeq T$. This claim is true because T contains no pattern variables, and so neither does *reduce app* (ϕP). Thus, if any pattern variables do appear in the expression ϕP, their value does not affect the result of applying *reduce app* to it, and so any substitution δ can freely substitute for them without changing this value.

Corollary 2.6. *If $\phi \leq \psi$, then*

$$\phi \vdash_{\mathit{app}} Xs \quad \Rightarrow \quad \psi \vdash_{\mathit{app}} Xs$$

2.3.3 Match sets

If Xs is a set of rules, then we define the *match set with respect to app* of Xs to be a set of substitutions $\mathcal{M}$ such that:

- For all ϕ: $\phi \vdash_{\mathit{app}} Xs$ iff there exists $\psi \in \mathcal{M}$ such that $\psi \leq \phi$.

- For all $\phi_1, \phi_2 \in \mathcal{M}$: if $\phi_1 \leq \phi_2$, then $\phi_1 = \phi_2$.

The first condition is a soundness and completeness property. The backwards direction is soundness; it says that all substitutions in a match set satisfy the rules. The forwards implication is completeness; it says that every match is represented, either by itself or by a more general match. The second condition states that there are no redundant elements in a match set.

It should be noted that this defines the *unique* match set for a set of rules; given the (infinite) set of all possible matches we could cut this down to the match set by removing all elements for which a more general match also existed.

With these definitions done, we note we will give the specifications of our algorithms by defining an appropriate *app* function and giving conditions under which they produce match sets with respect to this *app* function.

2.4 Implementation

We shall also follow a common pattern in implementing our algorithms; we take the approach of progressively breaking down a set of rules into smaller rules, until none remain.

In the interests of producing algorithms that can be efficiently implemented, we shall generally represent collections as *bags*. As a result, it will be incumbent on us to show that these bags do not contain duplicate elements where necessary for the purposes of verifying non-redundancy conditions. Bags are denoted by the brackets $\llbracket$ and $\rrbracket$, bag union by the operator $+$, and membership by $\in$ as with sets.

Our algorithms all start with the function *matches*, which takes a set of rules and returns a set of matches. As with our specification, we parameterise this function by *app*:

$$\begin{aligned}
\mathit{matches\ app} &:: [\![\, \mathit{Rule} \,]\!] \to [\![\, \mathit{Subst} \,]\!] \\
\mathit{matches\ app}\, [\![\,]\!] &= [\![\, \mathit{idSubst} \,]\!] \\
\mathit{matches\ app}\, ([\![X]\!] + \mathit{Xs}) &= [\![\, (\phi \circ \sigma) \mid \ (\sigma, \mathit{Ys}) \in \mathit{resolve\ app}\, X, \\
&\qquad\qquad \phi \in \mathit{matches\ app}\, (\sigma\, (\mathit{Xs} + \mathit{Ys}))\ ]\!]
\end{aligned}$$

The empty set of rules gives a singleton match set containing the identity substitution. For a non-empty set of rules $[\![X]\!] + \mathit{Xs}$, X is broken down (in zero or more ways) into a set of smaller rules Ys along with a substitution σ that makes Ys equivalent to X and which represents the “result so far”. The new set Ys is combined with the old set Xs, and the substitution σ is applied to the result of the combination to give a new set of rules to pass to a recursive call to *matches*. Finally, σ is composed with each of the results of the recursive call to return a match.

The key part to note in the above definition is the function *resolve*, which corresponds closely to the logic programming concept of “resolution”. It is in this function that the crucial work of breaking a rule down into a substitution and a set of smaller rules is done. In this thesis, the implementation of the matching algorithms presented are all based around the above *matches* function, differing only in specific parts of the definition of *resolve*.

We can think of the matching process described by *matches* as building a tree whose nodes are rule sets and whose edges are substitutions. The tree is rooted at a node consisting of the original rule set; to grow the tree, a previously unconsidered node with a non-empty rule set is chosen. This rule set is represented by the parameter $[\![X]\!] + \mathit{Xs}$ in the definition of *matches* above. The order in which unconsidered nodes are chosen does not affect the final result, since the procedure is guaranteed to terminate with a result that is the unique match set (as we shall show later).

The rule X is then selected from the rule set. Applying *resolve* to X gives a (possibly empty) list of (substitution, rule set) pairs. For each pair (σ, Ys), a new node is created, linked to the original node by an edge labelled with the substitution σ; the node itself is labelled with the residual rule set $\sigma\,(Xs + Ys)$, where Xs is the rule set from the original node with X removed.

This procedure is repeated until no unconsidered nodes remain. At this point, the nodes at the leaves of the tree that are labelled with empty rule sets represent successful matches – they correspond to the first clause in the definition of *matches* above. For each of these leaves the match itself can be recovered by traversing the tree from the leaf to the root, composing the substitutions found on edges. Since substitutions are applied to the residual rule sets as they are generated, all the substitutions will have disjoint domains, and so it does not matter in what order they are composed.

We hope that considering the matching process in terms of such trees will provide an intuitive means of understanding it. We shall provide an example of one for each specific definition of *resolve* that we present, in Sections 2.6, 3.2 and 4.2 – for an immediate example, see page 61.

2.4.1 Specification of *resolve*

For a particular *app* function, suppose that

$$\mathit{resolve}\ \mathit{app}\ X \quad = \quad [\![\, (\sigma_0,\, Ys_0), (\sigma_1,\, Ys_1), \ldots, (\sigma_k,\, Ys_k) \,]\!]$$

We require that

(1) For all substitutions ϕ:

$$(\phi \vdash_{app} X) \quad \equiv \quad \bigvee_i (\phi \vdash_{app} Ys_i \wedge \sigma_i \leq \phi)$$

(2) For all substitutions ϕ and indices i and j:

$$(\phi \vdash_{app} Ys_i) \wedge (\phi \vdash_{app} Ys_j) \quad \Rightarrow \quad i = j$$

(3) For each index i, σ_i is pertinent to X.

(4) The pattern variables in Ys_i are contained in the pattern variables of X.

(5) For each index i:

$$Ys_i \ll X$$

(1) is a soundness and completeness condition: it says that all relevant matches can be reached via *resolve*, and that *resolve* stays true to the original set of rules. (2) states that *resolve* should not return any superfluous results. It should be noted that this condition does not mention σ_i or σ_j, so it is possible that a reasonable definition of *resolve* might exist which violates this specification by producing two (σ, Ys) pairs in which the two values for Ys were the same but the two σs were different. However, the definition that we shall give does not have this property.

(3) and (4) are technical requirements we need to prove the non-redundancy of *matches*. Finally, (5) states that we make progress by applying *resolve*; *i.e.* that the process of breaking down the set of rules will eventually terminate. Note that the statement that Ys_i is a set of rules imposes an implicit condition on each member; the pattern is η-normal and the term normal.

2.4.2 Implementing *resolve*

Table 2.1 gives a definition for *resolve*, which once again is parametrised by *app*. This definition defers the case where the pattern is a function application to the function *appresolve*; the main work of this thesis will be to give appropriate definitions of *appresolve* for different *app* functions.

The first clause says that two local variables match only if they are equal. If they are, the identity substitution is returned along with an empty set of rules to indicate that no more rules need to be solved to make the expressions equal. Similarly, the second clause states the same principle for constants.

The third clause says that we can solve a rule $(p \to T)$ where the pattern is a pattern variable by making an appropriate substitution. Such a substitution can only be made, however, if T does not contain any local variables occurring without their enclosing λ; otherwise it would move these variables out of scope.

X	*resolve app* X
$x \to y$	$[\![\, (\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $x = y$ $[\![\,]\!]$, otherwise
$a \to b$	$[\![\, (\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $a = b$ $[\![\,]\!]$, otherwise
$p \to T$	$[\![\, (p := T, [\![\,]\!])\,]\!]$, if T does not contain unbound local variables $[\![\,]\!]$, otherwise
$(\lambda x.P) \to (\lambda x.T)$	$[\![\, (\mathit{idSubst}, [\![\, P \to T\,]\!])\,]\!]$
$(\lambda x.P) \to T$	$[\![\, (\mathit{idSubst}, [\![\, P \to (T\, x)\,]\!])\,]\!]$
$(F\ E) \to T$	$[\![\, (\mathit{idSubst}, \mathit{Ys}) \mid \mathit{Ys} \in \mathit{appresolve\ app}\ F\ E\ T\,]\!]$
$P \to T$	$[\![\,]\!]$

Table 2.1: Definition of *resolve*

Next, we consider matching of λ-abstractions $(\lambda x.P)$ and $(\lambda x.T)$. Here it is assumed that the clauses are applied modulo renaming, so that the bound variable on both sides is the same, namely x. To match the λ-abstractions is to match their bodies; note that we have already dealt with the problem of variable capture above.

Since our definition of equality applies modulo η-conversion, we have to consider the possibility that P is a λ-abstraction but T is not (if the reverse is true, one of the other clauses will apply instead). In this case we simply replace T by its equivalent $(\lambda x.T\,x)$ and continue; for the purposes of satisfying the progress requirement on *resolve* we also immediately apply the previous rule.

If the pattern is a function application then we choose the identity substitution, and delegate the task of generating new sets of rules to *appresolve*. Finally, if none of the other rules apply then no matches exist.

2.5 Proof of correctness

Our next task is to prove that the definitions above meet our specification, namely that *matches* does return a match set. To do this, we first show that this is the case given a definition of *resolve* that satisfies the specification in Section 2.4.1. We then prove that, given appropriate conditions on *appresolve*, the definition of *resolve* given in Section 2.4.2 is correct.

2.5.1 Correctness of *matches*

We have to verify two properties:

- For each substitution ϕ and set of rules Xs:

$$(\phi \vdash_{app} Xs) \quad \equiv \quad \exists \psi \in matches\ app\ Xs\ :\ \psi \leq \phi$$

- For all substitutions ϕ and ψ and sets of rules Xs:

$$(\phi, \psi \in matches\ app\ Xs \wedge \phi \leq \psi) \quad \Rightarrow \quad (\phi = \psi)$$

For the first condition (soundness and completeness), we proceed by induction on $\ll$. For the empty set of rules,

$$\begin{array}{ll}
 & \phi \vdash_{app} [\![\,]\!] \\
\equiv & \quad \{\text{definition of } \vdash_{app} \text{ on a set of rules}\} \\
 & True \\
\equiv & \quad \{\text{definition of } idSubst,\ \leq\} \\
 & idSubst \leq \phi \\
\equiv & \quad \{\text{definition of } matches\ app\ [\![\,]\!]\} \\
 & \exists \psi \in matches\ app\ [\![\,]\!]. \psi \leq \phi
\end{array}$$

Next, we consider the case of a non-empty set of rules $[\![X]\!] + Xs$. Suppose

$$resolve\ app\ X \quad = \quad [\![\,(\sigma_0,\ Ys_0), (\sigma_1,\ Ys_1), \ldots, (\sigma_k,\ Ys_k)\,]\!]$$

Then,

$$
\begin{array}{ll}
& \phi \vdash_{app} (\llbracket X \rrbracket + Xs) \\
\equiv & \{\text{definition of } \vdash_{app} \text{ on set of rules}\} \\
& \phi \vdash_{app} X \wedge \phi \vdash_{app} Xs \\
\equiv & \{\text{soundness and completeness of } \mathit{resolve}\} \\
& (\bigvee_i \phi \vdash_{app} Ys_i \wedge \sigma_i \leq \phi) \wedge \phi \vdash_{app} Xs \\
\equiv & \{(\wedge) \text{ distributes over } (\vee)\} \\
& \bigvee_i (\phi \vdash_{app} Ys_i \wedge \sigma_i \leq \phi \wedge \phi \vdash_{app} Xs\,) \\
\equiv & \{\text{definition of generality } (\leq)\} \\
& \bigvee_i (\phi \vdash_{app} Ys_i \wedge (\exists \delta . \phi = \delta \circ \sigma_i) \wedge \phi \vdash_{app} Xs\,) \\
\equiv & \{\text{predicate logic}\} \\
& \bigvee_i (\exists \delta . \phi \vdash_{app} Ys_i \wedge \phi = \delta \circ \sigma_i \wedge \phi \vdash_{app} Xs\,) \\
\equiv & \{\delta \circ \sigma_i \vdash_{app} Xs \equiv \delta \vdash_{app} (\sigma_i\, Xs) \text{ (Corollary 2.4)}\} \\
& \bigvee_i (\exists \delta . \phi = \delta \circ \sigma_i \wedge \delta \vdash_{app} (\sigma_i\, Ys_i) \wedge \delta \vdash_{app} (\sigma_i\, Xs),\,) \\
\equiv & \{\text{definition of } \vdash_{app} \text{ on list of rules}\} \\
& \bigvee_i (\exists \delta . \phi = \delta \circ \sigma_i \wedge \delta \vdash_{app} (\sigma_i\, (Xs + Ys_i))\,) \\
\equiv & \{\text{progress of } \mathit{resolve} \Rightarrow Xs + Ys_i \ll \llbracket X \rrbracket + Xs\text{; induction}\} \\
& \bigvee_i (\exists \delta . \phi = \delta \circ \sigma_i \wedge (\exists \chi \in \mathit{matches}\ \mathit{app}\, (\sigma_i\, (Xs + Ys_i)) \,:\, \chi \leq \delta)\ ) \\
\equiv & \{\text{predicate logic}\} \\
& \bigvee_i (\exists \chi \in \mathit{matches}\ \mathit{app}\, (\sigma_i\, (Xs + Ys_i)) \,:\, \exists \delta \,:\, \phi = \delta \circ \sigma_i \wedge \chi \leq \delta\ ) \\
\equiv & \{\text{definition of } \leq\} \\
& \bigvee_i (\exists \chi \in \mathit{matches}\ \mathit{app}\, (\sigma_i\, (Xs + Ys_i)) \,:\, \exists \delta, \gamma \,:\, \phi = \delta \circ \sigma_i \wedge \gamma \circ \chi = \delta\ ) \\
\equiv & \{\text{backwards, choose } \delta = \gamma \circ \chi\} \\
& \bigvee_i (\exists \chi \in \mathit{matches}\ \mathit{app}\, (\sigma_i\, (Xs + Ys_i)) \,:\, \exists \gamma . \gamma \circ \chi \circ \sigma_i = \phi\,) \\
\equiv & \{\text{definition of } \leq\} \\
& \bigvee_i (\exists \chi \in \mathit{matches}\ \mathit{app}\, (\sigma_i\, (Xs + Ys_i)) \,:\, \chi \circ \sigma_i \leq \phi\,) \\
\equiv & \{\text{definition of } \mathit{matches}\text{, take } \psi = \chi \circ \sigma_i\}
\end{array}
$$

$$\exists \psi \in \textit{matches app}\,(\llbracket X \rrbracket + \textit{Xs}) \;:\; \psi \leq \phi$$

It remains to prove non-redundancy of *matches app Xs*; we first prove by induction over the measure of a rule set that the substitutions returned by *matches app Xs* are closed and pertinent to *Xs*. Clearly the identity substitution is pertinent to any rule set, so the base case for the empty rule set is satisfied. For the case of *matches app* $(\llbracket X \rrbracket + \textit{Xs})$, we know that the substitution σ returned by *resolve* is pertinent to X, and by the induction hypothesis ϕ is pertinent to $\sigma(\textit{Xs} + \textit{Ys})$.

Since any pattern variables in *Ys* are also in X, $(\phi \circ \sigma)$ is pertinent to $\llbracket X \rrbracket + \textit{Xs}$, thus completing the proof.

Let us now move on to the proof of non-redundancy, which also proceeds by induction on the measure of a rule set. The base case is trivially satisfied since we only return a single substitution for the empty rule set. For the step case, let ϕ and ψ be elements of *matches app* $\llbracket X \rrbracket + \textit{Xs}$. Furthermore, assume that $\phi \leq \psi$. From the definition of *matches*, there exist $(\sigma_i, \textit{Ys}_i)$, $(\sigma_j, \textit{Ys}_j)$ in *resolve app X* such that:

$$\begin{aligned}
\exists \phi' \in \textit{matches app}\,(\sigma_i\,(\textit{Xs} + \textit{Ys}_i)) \quad &: \quad \phi = \phi' \circ \sigma_i \\
\exists \psi' \in \textit{matches app}\,(\sigma_j\,(\textit{Xs} + \textit{Ys}_j)) \quad &: \quad \psi = \psi' \circ \sigma_j
\end{aligned}$$

From the soundness of *resolve*, we have that

$$\phi \vdash_{\textit{app}} \textit{Ys}_i \wedge \psi \vdash_{\textit{app}} \textit{Ys}_j$$

Since $\phi \leq \psi$, we have by Lemma 2.6 that:

$$\psi \vdash_{\textit{app}} \textit{Ys}_i \wedge \psi \vdash_{\textit{app}} \textit{Ys}_j$$

Thus, by the non-redundancy of *resolve*, we have that $i = j$.

Now, since $\phi \leq \psi$, there exists δ such that $\delta \circ \phi = \psi$. Therefore, $\delta \circ \phi' \circ \sigma_i = \psi' \circ \sigma_i$. We know that ϕ' and ψ' are pertinent to $\sigma_i(\textit{Xs} + \textit{Ys})$, so they cannot make any changes to variables which are changed by σ_i.

Construct δ' by restricting the domain of δ to variables not changed by σ_i. Now consider a pattern variable p. If p is changed by σ_i, it cannot be changed by ϕ', ψ' or δ', and so $(\delta' \circ \phi')p = \psi' p$. If p is not changed by σ_i, then $\sigma_i p = p$ and so $(\delta' \circ \phi')p = \psi' p$.

Therefore $\delta' \circ \phi'$ and ψ' are equal on all pattern variables, and so $\delta' \circ \phi' = \psi'$ and thus $\phi' \leq \psi'$. We can now apply the induction hypothesis to give $\phi' = \psi'$ and so $\phi = \psi$.

2.5.2 Correctness of *resolve*

In order to prove this definition of *resolve* correct, we have to satisfy the five proof obligations listed in Section 2.4.1, and verify the validity of each rule generated.

In fact, all of these obligations save the first are easy to check for each case except that where the pattern is a function application. Since each rule set has at most one member the non-redundancy condition (2) is obviously true; a non-identity substitution is only generated when matching a pattern variable against a closed term, so (3) is met, and it is similarly easy to verify (4) and (5), and that each generated rule is valid, by inspection. This just leaves the soundness and completeness condition (1).

2.5.2.1 Simplifying the proof obligation

The definition of $\vdash_{app}$ means that it will frequently be necessary to reason about $(\simeq)$, that is equality modulo η-reduction. This can sometimes be rather complicated, so we first show how $(\simeq)$ can be eliminated from the definition. Recall that *reduce* was defined by:

$$\begin{array}{rcl}
\mathit{reduce}\ \mathit{app}\ c & = & c \\
\mathit{reduce}\ \mathit{app}\ x & = & x \\
\mathit{reduce}\ \mathit{app}\ p & = & p \\
\mathit{reduce}\ \mathit{app}\ (\lambda x.E) & = & \lambda x.(\mathit{reduce}\ \mathit{app}\ E) \\
\mathit{reduce}\ \mathit{app}\ (E_1\ E_2) & = & \mathit{app}\ (\mathit{reduce}\ \mathit{app}\ E_1)\ (\mathit{reduce}\ \mathit{app}\ E_2)
\end{array}$$

We now define *reduce′* as follows:

$$\begin{array}{rcl}
\mathit{reduce}'\,\mathit{app}\;c & = & c \\
\mathit{reduce}'\,\mathit{app}\;x & = & x \\
\mathit{reduce}'\,\mathit{app}\;p & = & p \\
\mathit{reduce}'\,\mathit{app}\;(\lambda x.E) & = & \mathit{etared}\;(\lambda x.(\mathit{reduce}'\,\mathit{app}\;E)) \\
\mathit{reduce}'\,\mathit{app}\;(E_1\,E_2) & = & \mathit{etanormalise} \\
 & & \qquad (\mathit{app}\;(\mathit{reduce}'\,\mathit{app}\;E_1)\;(\mathit{reduce}'\,\mathit{app}\;E_2))
\end{array}$$

where *etared* is defined as the function that strips off a single *outer* η-redex from an expression if one exists, and otherwise leaves it unchanged:

$$\begin{array}{rcll}
\mathit{etared}\;E & = & F & \text{if } E = \lambda x.F\,x,\; x \not\trianglelefteq F \\
 & & E & \text{otherwise}
\end{array}$$

Now, since *reduce′* removes η-redexes at any point where it might otherwise have generated them, we have that $\mathit{etanormalise}\,(\mathit{reduce}\;\mathit{app}\;E) = \mathit{reduce}'\,\mathit{app}\;E$. Although this fact is true for all expressions E, we will later want to simplify the $\mathit{etanormalise}\,(\mathit{app}\,(\mathit{reduce}\;\mathit{app}\;E_1)\,(\mathit{reduce}\;\mathit{app}\;E_2))$ element in the definition of *reduce′* for particular versions of *app*, and we therefore restrict *reduce′* to being applied to η-normal arguments only.

Now, given a substitution ϕ and a rule $P \to T$, we have the following:

$$\begin{array}{ll}
 & \phi \vdash_{\mathit{app}} P \to T \\
\equiv & \quad \{\text{definition of } \vdash_{\mathit{app}}\} \\
 & \mathit{reduce}\;\mathit{app}\;(\phi P) \simeq T \\
\equiv & \quad \{T \text{ normal}\} \\
 & \mathit{etanormalise}\;(\mathit{reduce}\;\mathit{app}\;(\phi P))) = T \\
\equiv & \quad \{\phi \text{ and } P\ \eta\text{-normal guarantees } \phi P\ \eta\text{-normal}\} \\
 & \mathit{reduce}'\,\mathit{app}\;(\phi P) = T
\end{array}$$

This equivalent form for the definition of $\vdash_{\mathit{app}}$ will prove useful in the following proofs. In each of these proofs, the goal is to prove that a specific clause of the implementation of *resolve* satisfies the soundness and completeness condition (1) of Section 2.4.1, which for convenience we rephrase here as:

$$\phi \vdash_{app} X \quad \equiv \quad \exists (Ys, \sigma) \in \mathit{resolve}\ \mathit{app}\ X \ : \ \phi \vdash_{app} Ys \wedge \sigma \leq \phi$$

2.5.2.2 Matching local variables

X	$\mathit{resolve}\ \mathit{app}\ X$
$x \rightarrow y$	$[\![\, (\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $x = y$ $[\![\,]\!]$, otherwise

$$\begin{array}{ll}
 & \phi \vdash_{app} x \rightarrow y \\
\equiv & \quad \{\text{property of } \vdash_{app}\} \\
 & \mathit{reduce}'\ \mathit{app}\ (\phi x) = y \\
\equiv & \quad \{x \text{ is a local variable and cannot be substituted}\} \\
 & \mathit{reduce}'\ \mathit{app}\ x = y \\
\equiv & \quad \{\text{definition of } \mathit{reduce}'\} \\
 & x = y \\
\equiv & \quad \{\text{predicate logic, vacuous truths about } \vdash_{app} \text{ and } \leq\} \\
 & \phi \vdash_{app} [\![\,]\!] \wedge \mathit{idSubst} \leq \phi, \ \text{if } x = y \\
 & \mathit{False}, \ \text{otherwise}
\end{array}$$

2.5.2.3 Matching constants

X	$\mathit{resolve}\ \mathit{app}\ X$
$a \rightarrow b$	$[\![\, (\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $a = b$ $[\![\,]\!]$, otherwise

$$\begin{array}{ll}
 & \phi \vdash_{app} a \rightarrow b \\
\equiv & \quad \{\text{property of } \vdash_{app}\} \\
 & \mathit{reduce}'\ \mathit{app}\ (\phi a) = b
\end{array}$$

$\equiv \quad \{a \text{ is a constant and cannot be substituted}\}$

$\mathit{reduce}'\ \mathit{app}\ a = b$

$\equiv \quad \{\text{definition of } \mathit{reduce}'\}$

$a = b$

$\equiv \quad \{\text{predicate logic, vacuous truths about } \vdash_{\mathit{app}} \text{ and } \leq\}$

$\phi \vdash_{\mathit{app}} [\![\,]\!] \wedge \mathit{idSubst} \leq \phi, \text{ if } a = b$
$\mathit{False}, \text{ otherwise}$

2.5.2.4 Matching against a pattern variable

X	$\mathit{resolve}\ \mathit{app}\ X$
$p \to T$	$[\![\,(p := T, [\![\,]\!])\,]\!]$, if T does not contain unbound local variables $[\![\,]\!]$, otherwise

$\phi \vdash_{\mathit{app}} p \to T$

$\equiv \quad \{\text{property of } \vdash_{\mathit{app}}\}$

$\mathit{reduce}'\ \mathit{app}\ (\phi p) = T$

$\equiv \quad \{\text{expressions in the range of a substitution are normal}\}$

$\phi p = T$

$\equiv \quad \{\text{property of substitution}\}$

$((p := T) \leq \phi) \wedge T$ does not contain unbound local variables

$\equiv \quad \{\text{predicate logic, vacuous truth about } \vdash_{\mathit{app}}\}$

$\phi \vdash_{\mathit{app}} [\![\,]\!] \wedge (p := T) \leq \phi,$ if T does not contain unbound local variables
$\mathit{False}, \text{ otherwise}$

2.5.2.5 Matching λ-abstractions

X	$\mathit{resolve}\ \mathit{app}\ X$
$(\lambda x.P) \to (\lambda x.T)$	$[\![\,(\mathit{idSubst}, [\![\, P \to T \,]\!])\,]\!]$

$$
\begin{array}{ll}
& \phi \vdash_{app} (\lambda x.P) \to (\lambda x.T) \\
\equiv & \{\text{definition of } \vdash_{app}\} \\
& \mathit{reduce\ app}\,(\phi(\lambda x.P)) \simeq \lambda x.T \\
\equiv & \{\text{property of substitution}\} \\
& \mathit{reduce\ app}\,(\lambda x.(\phi P)) \simeq \lambda x.T \\
\equiv & \{\text{definition of } \mathit{reduce}\} \\
& \lambda x.\mathit{reduce\ app}\,(\phi P) \simeq \lambda x.T \\
\equiv & \{\text{property of } \simeq\} \\
& \mathit{reduce\ app}\,(\phi P) \simeq T \\
\equiv & \{\text{definition of } \vdash_{app}, \text{ vacuous truth about } \leq\} \\
& \phi \vdash_{app} P \to T \wedge \mathit{idSubst} \leq \phi
\end{array}
$$

2.5.2.6 Matching against a λ-abstraction

X	*resolve app* X
$(\lambda x.P) \to T$	$[\![\, (\mathit{idSubst}, [\![\, P \to (T\,x) \,]\!]) \,]\!]$

Here we make use of the fact that clauses are applied in order to assert that T is a normal expression that is *not* a λ-abstraction. This is important to guarantee that $T\,x$ is normal.

$$
\begin{array}{ll}
& \phi \vdash_{app} (\lambda x.P) \to T \\
\equiv & \{\text{definition of } \vdash_{app}\} \\
& \mathit{reduce\ app}\,(\phi(\lambda x.P)) \simeq T \\
\equiv & \{\text{property of substitution}\} \\
& \mathit{reduce\ app}\,(\lambda x.(\phi P)) \simeq T \\
\equiv & \{\text{definition of } \mathit{reduce}\} \\
& \lambda x.\mathit{reduce\ app}\,(\phi P) \simeq T \\
\equiv & \{\text{property of } \simeq\} \\
& \lambda x.\mathit{reduce\ app}\,(\phi P) \simeq \lambda x.T\,x
\end{array}
$$

$$\begin{array}{ll}
\equiv & \{\text{property of } \simeq\} \\
& \mathit{reduce}\ \mathit{app}\ (\phi P) \simeq T\ x \\
\equiv & \{\text{definition of } \vdash_{\mathit{app}}, \text{vacuous truth about } \leq\} \\
& \phi \vdash_{\mathit{app}} P \rightarrow T\ x \wedge \mathit{idSubst} \leq \phi
\end{array}$$

2.5.2.7 Failure to match

X	*resolve app X*
$P \rightarrow T$	$[\![\,]\!]$

$$\begin{array}{ll}
& \phi \vdash_{\mathit{app}} P \rightarrow T \\
\equiv & \{\text{property of } \vdash_{\mathit{app}}\} \\
& \mathit{reduce}'\ \mathit{app}\ (\phi P) = T \\
\Rightarrow & \{\text{definition of } \mathit{reduce}'\} \\
& \quad (\exists c.\phi P = c \wedge T = c) \\
& \vee\ (\exists x.\phi P = x \wedge T = x) \\
& \vee\ (\exists B.\phi P = \lambda x.B) \\
& \vee\ (\exists E_1, E_2 : \phi P = E_1\ E_2) \\
\Rightarrow & \{\text{property of substitution}\} \\
& \quad (\exists p.P = p) \\
& \vee\ (\exists c.P = c \wedge T = c) \\
& \vee\ (\exists x.P = x \wedge T = x) \\
& \vee\ (\exists B.P = \lambda x.B) \\
& \vee\ (\exists E_1, E_2.P = E_1\ E_2)
\end{array}$$

We again make use of the fact that clauses of *resolve* are applied in order; since each of the cases in this condition has been covered by previous clauses, it evaluates to *False* in this context, which is equivalent to saying that *resolve app* $(P \rightarrow T)$ has no members, as required. The reverse implication follows trivially since *False* can imply anything.

2.5.2.8 Matching against an application

Thus, we have shown that *resolve* satisfies its specification for all clauses except that where the pattern is a function application. To complete the proof, we derive appropriate conditions on *appresolve*. For the soundness and completeness condition:

$$
\begin{array}{ll}
 & \phi \vdash_{app} F\,E \to T \\
\equiv & \quad \{\text{property of } \vdash_{app}\} \\
 & \mathit{reduce'}\;\mathit{app}\;(\phi(F\,E)) = T \\
\equiv & \quad \{\text{property of substitution}\} \\
 & \mathit{reduce'}\;\mathit{app}\;((\phi F)\,(\phi E)) = T \\
\equiv & \quad \{\text{definition of } \mathit{reduce'}\} \\
 & \mathit{etanormalise}\;(\mathit{app}\;(\mathit{reduce'}\;\mathit{app}\;(\phi F))\;(\mathit{reduce'}\;\mathit{app}\;(\phi E))) = T \\
\equiv & \quad \{\text{condition on } \mathit{appresolve}\;\mathit{app}\} \\
 & \exists\, Ys \in \mathit{appresolve}\;\mathit{app}\;F\;E\;T . \phi \vdash_{app} Ys
\end{array}
$$

Similarly, conditions (2), (4) and (5) give rise to analogous conditions on *appresolve app*. Condition (3) is trivially satisfied since the substitution returned is *idSubst*. Thus, we are left with the following conditions on *appresolve app*:

Suppose that $\mathit{appresolve}\;\mathit{app}\;F\;E\;T = [\![\, Ys_1, ..., Ys_k \,]\!]$. Then we require that:

{1} For all substitutions ϕ:

$$
\begin{gathered}
\mathit{etanormalise}\;(\mathit{app}\;(\mathit{reduce'}\;\mathit{app}\;(\phi F))\;(\mathit{reduce'}\;\mathit{app}\;(\phi E))) = T \\
\equiv \\
\bigvee_i \phi \vdash_{app} Ys_i
\end{gathered}
$$

{2} For all substitutions ϕ:

$$(\phi \vdash_{app} Ys_i) \wedge (\phi \vdash_{app} Ys_j) \;\;\Rightarrow\;\; i = j\ .$$

{3} The pattern variables in Ys_i are contained in the pattern variables of $F\ E$.

{4} For each index i:

$$Ys_i \ll (F\ E \to T)\ .$$

2.6 Example : Simple matching

In order to provide a simple demonstration of the above framework, we present an algorithm which implements *simple matching.* The *app* function for this case is the *none* function we described earlier:

$$\mathit{none}\ F\ E \quad = \quad F\ E$$

It is straightforward to define *appresolve none*:

$$\begin{aligned} \mathit{appresolve\ none}\ F\ E\ (T_0\ T_1) \quad &= \quad [\![\,[\![\, F \to T_0, E \to T_1 \,]\!]\,]\!] \\ \mathit{appresolve\ none}\ F\ E\ T \quad &= \quad [\![\,]\!], \text{ if } T \neq T_0\ T_1 \end{aligned}$$

If the term is also a function application, then we simply match F against the function part and E against the argument part. Otherwise, there are no matches.

We show a matching tree constructed in the manner described on page 46 for the matching problem $p\ q\ q \to 1 + 1$ in Figure 2.1. Note that for each node, we select the first rule in the rule set for applying *resolve*.

The proof of correctness is quite simple. For condition {1}:

$$\begin{aligned} & \mathit{etanormalise}\ (\mathit{none}\ (\mathit{reduce}'\ \mathit{none}\ (\phi F))\ (\mathit{reduce}'\ \mathit{none}\ (\phi E))) = T \\ \equiv \quad & \{\text{definition of } \mathit{none}, \phi, F, E\ \eta\text{-normal}\} \\ & (\mathit{reduce}'\ \mathit{none}\ (\phi F))\ (\mathit{reduce}'\ \mathit{none}\ (\phi E)) = T \\ \equiv \quad & \{\text{definition of } =\} \\ & T = T_0\, T_1 \wedge \mathit{reduce}'\ \mathit{none}\ (\phi F) = T_0 \wedge \mathit{reduce}'\ \mathit{none}\ (\phi E) = T_1 \end{aligned}$$

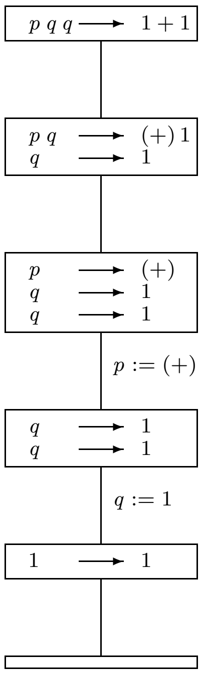

Figure 2.1: A simple matching tree for $p\ q\ q \rightarrow 1 + 1$

$$\equiv \quad \{\text{definition of } \vdash_{none}\}$$

$$T = T_0\,T_1 \wedge \phi \vdash_{none} \{\, F \rightarrow T_0, E \rightarrow T_1 \,\}$$

$$\equiv \quad \{\text{definition of } \mathit{appresolve\ none}\}$$

$$\exists\, Ys \in \mathit{appresolve\ none}\ F\ E\ T . \phi \vdash_{none}\ Ys$$

Condition {2} follows automatically since *appresolve none* returns at most one result; {3} is obviously satisfied since the only new rules generated have F and E as patterns, and finally {4} is satisfied since breaking down the pattern $F\ E$ into individual rules in F and E removes one operator.

Note that simple match sets will contain at most one match, since the definition of *appresolve none* returns at most one set of rules, and each of the clauses in *resolve* also either return one set of rules or at most one match.

This means that the specification for simple matching can be simplified somewhat from the general specification given in Section 2.3.3, since the non-redundancy condition is trivially satisfied. Also, since P is η-normal,

$\mathit{reduce}'\ \mathit{none}\ P = P$, and so the specification becomes

$$\phi \in \mathit{matches}\ \mathit{none}\,[\![\, P \rightarrow T \,]\!] \;\; \equiv \;\; \phi P = T \wedge \forall \psi : \psi P = T \Rightarrow \phi \leq \psi$$

We can say something about the relationship between simple matching and general matching:

Lemma 2.7. *If ϕ is a first-order general match for the rule $P \rightarrow T$, with P β-normal, then ϕ is also a simple match for this rule.*

Proof. First, suppose ϕP contains a β-redex $(\lambda x.B)\, E$. Since P is β-normal, the redex was not present in P, and since expressions in the range of ϕ are normal, it was not present in ϕ. Therefore, P contained an expression of the form $p\, E'$, with $\phi p = \lambda x.B$ and $\phi E' = E$. But ϕ is a first-order match, and $\lambda x.B$ is a second-order expression. Thus, ϕP is β-normal.

Therefore,

$$\begin{array}{ll} & \mathit{reduce}\ \mathit{none}\ (\phi P) \\ = & \quad \{\text{definition of } \mathit{reduce}, \mathit{none}\} \\ & \phi P \\ = & \quad \{\phi P\ \beta\text{-normal}\} \\ & \mathit{betanormalise}(\phi P) \\ \simeq & \quad \{\phi \text{ is a general match}\} \\ & T \end{array}$$

Therefore ϕ is a simple match between P and T. □

The simple matching algorithm also sometimes returns results that are not first-order. For example, consider the pattern $p\,1\,1$ and the term $1+1$. Then simple matching gives the substitution $p := (+)$, a second-order match. This is a pattern that we shall see repeated with the matching algorithms in this thesis – for each, we shall prove that they return all matches of a certain order or below, but also show that they return some extra results.

Chapter 3

One-step matching

In this chapter, we introduce the first of the matching algorithms that will follow the framework we have laid out. It is named *one-step* matching because the *reduce* function we shall define for it carries out one parallel β-reduction step; we shall elaborate on this informal description later. This work has previously been published in [27] and will be published in journal form in [28].

The standard matching algorithm used in program transformation systems such as KORSO [58] is the algorithm given by Huet and Lang in 1978 [47]; this algorithm is complete for all second-order matches in the simply-typed λ-calculus, but this is inadequate for even some quite simple transformations, such as the cat-elimination required for fast reverse and others. The one-step algorithm returns all second-order results, but also gives some extra results which in many cases overcome this inadequacy.

3.1 Specification

We shall now define *once*, the *app* function that shall specify the one-step algorithm:

$$\begin{aligned} \mathit{once}\,(\lambda x.B)\,E &= (x := E)B \\ \mathit{once}\,F\,E &= F\,E \end{aligned}$$

For convenience, we shall define *step* = *reduce once*. Recall that we can write *betanormalise* as *reduce full*, where:

$$\begin{aligned} \mathit{full}\,(\lambda x.B)\,E &= \mathit{reduce\,full}\,((x := E)B) \\ \mathit{full}\,F\,E &= F\,E \end{aligned}$$

Thus, *step* differs from *betanormalise* in that once *step* has reduced a β-redex by substituting an argument for a formal, it does nothing more to the result, whereas *betanormalise* would continue to reduce it if necessary. For example,

$$\mathit{step}\,((\lambda x.x\,1)\,(\lambda y.y + y)) \quad = \quad (\lambda y.y + y)\,1$$

whereas

$$\mathit{betanormalise}\,((\lambda x.x\,1)\,(\lambda y.y + y)) \quad = \quad 1 + 1$$

Intuitively, *step* can be thought of as applying one *parallel* reduction step to its argument; it conducts a bottom-up sweep, reducing redexes as it finds them. However, it should be noted that *step* does not correspond to similar notions of parallel β-reduction in the literature; a more common approach is that of *finite developments* [6]. A single (complete) finite development first marks all β-redexes in the original expression, then reduces just the marked redexes. In contrast, *step* will also reduce β-redexes that appear when the left-hand side of an application reduces to a λ-abstraction by a recursive call to *step*, thus:

$$\mathit{step}\,(((\lambda x.x)\,(\lambda x.x))\,((\lambda x.x)\,(\lambda x.x))) \quad = \quad \lambda x.x$$

In contrast, a single complete finite development of the same term would produce $(\lambda x.x)\,(\lambda x.x)$, but it would require two complete finite developments to completely reduce this expression.

An interesting property of *step* is that if applied enough times, it will always produce the same result as *betanormalise*:

Lemma 3.1. *If betanormalise* E *exists, then*

$$\exists n \;:\; \mathit{step}^n\,E = \mathit{betanormalise}\,E$$

Proof. If E is *strongly normalising*, *i.e.* has no infinite reduction sequences, then this is immediate (as is the existence of *betanormalise* E) by the fact that *step* will always reduce at least one β-redex if any exist in its parameter. In particular all terms which can be given types in any system of Barendregt's λ-cube satisfy this property [7].

In the general case, it is necessary to appeal to some theory from the untyped λ-calculus, which can be found in Section 13.2 of [6]. We sketch a proof by contradiction: suppose that there does not exist n such that $\mathit{step}^n\, E = \mathit{betanormalise}\, E$. Then for all n, $\mathit{step}^n\, E$ contains a β-redex and so $\mathit{step}^n\, E \neq \mathit{step}^{n+1}\, E$.

Now, each application of *step* can be expressed as a sequence of reductions of individual β-redexes; for example the evaluation of $\mathit{step}\,((\lambda xy.x + y)\, 0\, 1)$ is:

$$(\lambda xy.x + y)\, 0\, 1 \xrightarrow{\beta} (\lambda y.0 + y)\, 1 \xrightarrow{\beta} 0 + 1$$

The λs in a term can be ordered by their left-to-right position when the term is written out; this induces an ordering on the β-redexes of the term. If $\mathit{step}\, E \neq E$, then at least one of the elements of the reduction sequence from E to $\mathit{step}\, E$ will involve reducing the *leftmost* β-redex of the current term. Thus, there is an infinite reduction sequence starting from E containing an infinite number of reductions of leftmost β-redexes. Such a sequence is known as a *infinite quasi leftmost reduction sequence*, and by Theorem 13.2.6 of [6], its existence implies that E has no β-normal form and thus that *betanormalise* E cannot exist. □

In particular, if E does not contain a λ-abstraction applied to a term that reduces to another λ-abstraction then $n = 1$. This claim will form the basis of our proof that one-step matching returns at least as many matches as Huet and Lang's algorithm.

As remarked earlier, the one-step algorithm operates on untyped terms and does not depend on a particular typing discipline for its correctness. However, if we use the simply-typed lambda calculus (and run the algorithm ignoring the type information), the algorithm does return all matches of second-order or lower, so long as the pattern does not contain any β-redexes. However, it is not limited to second-order matches – in Section 3.1.1, we give an example of a third-order match which satisfies the specification of one-step matching.

To show that our algorithm returns all matches of second-order or lower, consider a rule $P \rightarrow E$, where P does not contain any β-redexes. (Recall that in a rule, the term E is always normal and therefore free of β-redexes.) Let ϕ be a general match between P and E. Furthermore, assume that ϕ does not contain any terms of order greater than 2. We aim to show that ϕ is in the match set of $P \rightarrow E$; the proof is by contradiction.

Suppose that ϕ is not represented in the match set. Then by completeness, we have $\mathit{step}(\phi P) \not\simeq E$. Since ϕ is a general match between P and E, $\mathit{betanormalise}\,(\phi P) \simeq E$, and so $\mathit{betanormalise}\,(\mathit{step}\,(\phi P)) \simeq \mathit{betanormalise}\,(\phi P) \not\simeq \mathit{step}\,(\phi P)$. Therefore $\mathit{step}(\phi P)$ contains a β-redex, the left-hand side of which is of at least second-order (since a first-order term cannot occur on the left-hand side of a function application). By Lemma 3.4 below, ϕP contains a β-redex with left-hand side of at least third-order. Since both P and the expressions in the range of ϕ are β-normal, P contains a subexpression of the form $p\,E$, where p is mapped to a λ-abstraction of at least third-order by ϕ. But this contradicts our assumptions on ϕ.

Before stating and proving Lemma 3.4, we prove the following technical lemma and corollary. Readers who are not interested in the details should skip to Section 3.1.1.

Lemma 3.2. *Suppose* $T \xrightarrow{\beta} S$.

(1) If $S = \lambda x.B$ *where* $\lambda x.B$ *is of order* n *then either* T *is a* λ*-abstraction of order* n *or it contains a* β*-redex whose left-hand side is of order* n *or higher.*

(2) If $(\lambda x.B)\,E \trianglelefteq S$ *where* $\lambda x.B$ *is of order* n*, then* T *contains a beta-redex whose left-hand side is of order* n *or higher.*

Proof.

(1) If $T = \lambda x.B'$ with $B' \xrightarrow{\beta} B$, the result is immediate.

Otherwise, the definition of $\xrightarrow{\beta}$ implies that $T = (\lambda y.C)\,F$, with $(y := F)\;C = \lambda x.B$. Then either:

- $C = y$ with $F = \lambda x.B$. Then since F is an expression of order n, $\lambda y.C$ is a expression of order $n+1$ and T is a β-redex as required.
- $C = \lambda x.B'$ with $(y := F)B' = B$. Then B' is of the same order as B and so T is a β-redex whose left-hand side is of order n or higher.

(2) This part is by induction on the size of T. Suppose that for all expressions smaller than T, property (2) holds. Now, either:

- $T = T_0\,T_1$, $S = S_0\,T_1$, and $T_0 \xrightarrow{\beta} S_0$. If $(\lambda x.B)\,E \trianglelefteq T_1$ the result is trivial. If $(\lambda x.B)\,E \trianglelefteq S_0$ then it follows from the induction hypothesis. Otherwise $S_0 = \lambda x.B$, $T_1 = E$. By (1), either T_0 is a λ-abstraction of order n and so T is a β-redex as required, or T_0 contains the required β-redex.
- $T = T_0\,T_1$, $S = T_0\,S_1$, and $T_1 \xrightarrow{\beta} S_1$. If $(\lambda x.B)\,E \trianglelefteq T_0$ the result is trivial. If $(\lambda x.B)\,E \trianglelefteq S_1$ then it follows from the induction hypothesis. Otherwise $T_0 = \lambda x.B$ and so T is the required β-redex.
- $T = \lambda y.T'$, $S = \lambda y.S'$, $T' \xrightarrow{\beta} S'$. The result follows from the induction hypothesis.
- $T = (\lambda y.C)\,F$, $S = (y := F)\;C$. Either:
  - $(\lambda x.B)\,E \trianglelefteq F$. The result is immediate.
  - $(\lambda x.B')\,E' \trianglelefteq C$ where $\lambda x.B'$ is of order n. The result is immediate.
  - $y\,E' \trianglelefteq C$ and $F = \lambda x.B$. Then $\lambda y.C$ is a λ-abstraction of order $n+1$.

□

Corollary 3.3. *(1) and (2) of Lemma 3.2 are also true if* $T \xrightarrow{\beta}{}^{*} S$.

Proof. This is trivially true for $T = S$, and so is true for any T and S by induction on the length of the reduction sequence $T \xrightarrow{\beta}{}^{*} S$. □

Lemma 3.4. *If step* T *contains a* β*-redex* $(\lambda x.B)\, E$*, where* $\lambda x.B$ *is an expression of order* n*, then* T *contains a* β*-redex* $(\lambda y.C)\, F$*, where* $\lambda y.C$ *is an expression of order* $n+1$ *or higher.*

Proof. The proof is by induction on the size of T. Suppose that the result is true for all expressions smaller than T. By Lemma 2.2,

$$\exists (T_0\; T_1) \trianglelefteq T \;:\; (\lambda x.B)\, E \trianglelefteq \mathit{once}\,(\mathit{step}\ T_0)\,(\mathit{step}\ T_1)$$

If *step* T_0 is not a λ-abstraction, then

$$\mathit{once}\,(\mathit{step}\ T_0)\,(\mathit{step}\ T_1) = (\mathit{step}\ T_0)\,(\mathit{step}\ T_1)$$

Therefore, $(\lambda x.B)\, E \trianglelefteq \mathit{step}\ T_0$ or $(\lambda x.B)\, E \trianglelefteq \mathit{step}\ T_1$, and the result holds by the induction hypothesis.

Otherwise, $\mathit{step}\ T_0 = \lambda y.C$, and therefore

$$\mathit{once}\,(\mathit{step}\ T_0)\,(\mathit{step}\ T_1) = (y := \mathit{step}\ T_1)\ C$$

Either:

- $(\lambda x.B)\, E \trianglelefteq \mathit{step}\ T_1$ (and y occurs in C). The result holds by the induction hypothesis.
- $(\lambda x.B')\, E' \trianglelefteq C$, with $(y := \mathit{step}\ T_1)B' = B$ and $(y := \mathit{step}\ T_1)E' = E$. Since substituting for an unbound local variable does not affect the order of an expression, we can again apply the induction hypothesis.
- $\mathit{step}\ T_1 = \lambda x.B$ and $y\, E' \trianglelefteq C$, with $(y := \mathit{step}\ T_1)E' = E$. Thus, $(\mathit{step}\ T_0)\,(\mathit{step}\ T_1) = (\lambda y.C)\,(\lambda x.B)$. Since $\lambda x.B$ is of order n, $\lambda y.C$ is of order $n+1$. By (1) of Lemma 3.2 either T_0 is a λ-abstraction of order $n+1$ whence $(T_0\; T_1)$ is the required β-redex, or T_0 contains the required β-redex.

□

3.1.1 Example : fast reverse

Recall the fast reverse example from Section 1.2. To verify the second side condition of the promotion rule, we had to match

$$P \quad = \quad \lambda x\, xs.(\otimes)\, x\, ((+\!\!+)\, (\mathit{reverse}\, xs))$$

against

$$T \quad = \quad \lambda x\, xs\, ys.(+\!\!+)\, (\mathit{reverse}\, xs)\, (x : ys)$$

giving the substitution

$$\phi \quad = \quad \{\, (\otimes) := \lambda a\, b\, ys.b\, (a : ys)\, \}$$

Firstly, it should be noted that since the parameter b is itself a function, this is a third-order match and thus Huet and Lang's algorithm would be inadequate. Secondly,

$$\phi P \quad = \quad \lambda x\, xs.(\lambda a\, b\, ys.b\, (a : ys))\, x\, ((+\!\!+)\, (\mathit{reverse}\, xs))$$

Furthermore, $\mathit{step}\, (\phi P) \; = \; T$ and so the one-step algorithm does find the appropriate match.

3.2 Algorithm

The following is the definition of *appresolve once*:

$$\begin{array}{rcl} \mathit{appresolve\ once}\, F\; E\; T & = & [\![\, [\![\, (F \to T_0), (E \to T_1)\,]\!] \mid (T_0\; T_1) = T\,]\!] \\ & + & [\![\, [\![\, (F \to T_0), (E \to T_1)\,]\!] \mid (T_0, T_1) \leftarrow \mathit{apps}\; T\,]\!] \\ & + & [\![\, [\![\, F \to (\lambda x.T)\,]\!] \mid \; x \text{ fresh}\,]\!] \end{array}$$

This definition allows any expression as the term T, and tries to write it as an application, so that the parts of the application can be matched against the pattern's function F and argument E respectively. There are three different ways *appresolve once* tries to do this. Firstly, T might already be an application $(T_0\; T_1)$, in which case we match F against T_0 and E against T_1.

Alternatively, there might be some pair of expressions (T_0, T_1) such that $(T_0\ T_1)$ is a β-redex which reduces to T. This can be formalized in the following specification of the function *apps*; *apps* T is the function that returns all pairs of normal expressions (T_0, T_1) such that

$$\begin{array}{lll} \exists B : (& & T_0 = \lambda x.B \\ & \wedge & (x := T_1)B = T \\ & \wedge & x \text{ occurs in } B,\ x \text{ fresh}) \end{array}$$

For example, a correct implementation of *apps* would return

$$\begin{array}{lll} \mathit{apps}\,(a + a) = & [\![& (\lambda x.x + x,\, a), \\ & & (\lambda x.x + a,\, a), \\ & & (\lambda x.x,\, a + a), \\ & & (\lambda x.x\ a, (+)\ a), \\ & & (\lambda x.x\ a\ a, (+)) \quad]\!] \end{array}$$

Note that the pair $(\lambda x.a + x, a)$ is *not* in the set of results. This is because $\lambda x.a + x$ is not η-normal. However, recall that if the expression T can be directly expressed as an application $(T_0\ T_1)$, then this is treated as a special case. The expression $a + a$ is the application $(a+)\ a$, and $(a+)$ is the η-normal form of $\lambda x.a + x$.

Finally, note the requirement in the specification of *apps* that x should occur in B. The reason for this is that if x did not occur in B, any expression would be a valid choice for T_1 and so *apps* would give an infinite set of results. We therefore deal with this case separately; if x does not occur in B then $B = T$ and so $T_0 = \lambda x.T$, thus we simply match F against $\lambda x.T$ and do not match E against anything, which indicates that it can take on any value.

In Figure 3.1, we give a matching tree for the one-step algorithm for the rule $p\ q \rightarrow 1 + 1$. Reading the results from the tree, we find the following

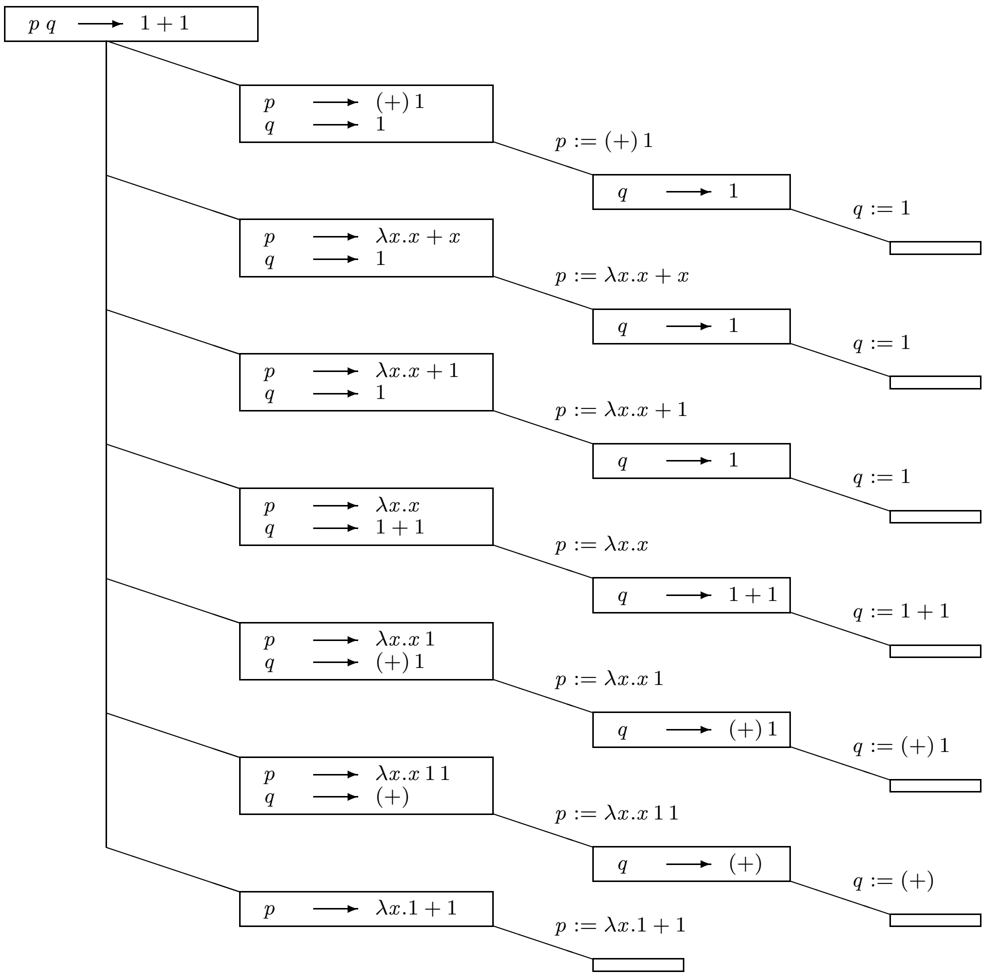

Figure 3.1: A one-step matching tree for $p\,q \to 1+1$

matches:

$$(p := (+)\,1, q := 1)$$
$$(p := \lambda x.x + x, q := 1)$$
$$(p := \lambda x.x + 1, q := 1)$$
$$(p := \lambda x.x, q := 1 + 1)$$
$$(p := \lambda x.x\,1, q := (+)\,1)$$
$$(p := \lambda x.x\,1\,1, q := (+))$$
$$(p := \lambda x.1 + 1)$$

3.2.1 Defining *apps*

The implementation of *apps* is relatively simple. The specification tells us that T_0 is a λ-abstraction $\lambda x.B$ where x appears in B. Therefore, since $(x := T_1)B = T$, any value we choose for T_1 must be subexpression of T. Therefore, *apps* T first finds all subexpressions of T and sets T_1 to be each subexpression in turn; T_0 will be the expression $\lambda x.B$. We construct B by taking T and selectively replacing occurrences of T_1 in T with the bound variable x. Care must be taken to ensure that at least one occurrence of T_1 is replaced, or B will not contain x.

We must also take care to avoid problems of variable capture. The specification of *apps* states that T_1 will be substituted for x in B, and therefore T_1 should not contain any unbound local variables whose binding λ occurs in B. This would happen if a subexpression was chosen that contained unbound local variables that were not also unbound in T.

$$\begin{array}{rcll} \mathit{apps}\ T & = & [\![\,(\lambda x.B, S) \mid & (S, \mathit{locs}) \leftarrow \mathit{collect}\,(\mathit{subexps}\ T), \\ &&& \mathit{unboundlocals}\ S \subseteq \mathit{unboundlocals}\ T, \\ &&& \mathit{locs}' \subseteq \mathit{locs}, \\ &&& \mathit{locs}' \neq \{\,\}, \\ &&& B = \mathit{replaces}\ T\ \mathit{locs}'\ x, \\ &&& \lambda x.B \text{ normal}, \\ &&& x \text{ fresh} \quad]\!] \end{array}$$

The *subexps* function returns all subexpressions of a given expression:

$$\begin{array}{rcl} \mathit{subexps}\ v & = & [\![\,(v, \langle\rangle)\,]\!] \\ \mathit{subexps}\ c & = & [\![\,(c, \langle\rangle)\,]\!] \\ \mathit{subexps}\,(\lambda x.E) & = & [\![\,((\lambda x.E), \langle\rangle)\,]\!] \\ & + & [\![\,(S, \mathit{Body}; \mathit{loc}) \mid (S, \mathit{loc}) \in \mathit{subexps}\ E\,]\!] \\ \mathit{subexps}\,(E_1\ E_2) & = & [\![\,(E_1\ E_2, \langle\rangle)\,]\!] \\ & + & [\![\,(S, \mathit{Func}; \mathit{loc}) \mid (S, \mathit{loc}) \in \mathit{subexps}\ E_1\,]\!] \\ & + & [\![\,(S, \mathit{Arg}; \mathit{loc}) \mid (S, \mathit{loc}) \in \mathit{subexps}\ E_2\,]\!] \end{array}$$

The *collect* function gathers together all subexpressions that have equal terms; this allows *apps* to identify common subexpressions. Its implementation is straightforward but somewhat intricate, so we define it here by the following specification:

$$\begin{array}{rcl} \exists \mathit{locs} : ((S, \mathit{locs}) \in \mathit{collect}\ \mathit{Ss} \wedge \mathit{loc} \in \mathit{locs}) & \equiv & (S, \mathit{loc}) \in \mathit{Ss} \\ \left(\begin{array}{cl} & \mathit{locs}_1 \neq \mathit{locs}_2 \\ \wedge & (S_1, \mathit{locs}_1) \in \mathit{collect}\ \mathit{Ss} \\ \wedge & (S_2, \mathit{locs}_2) \in \mathit{collect}\ \mathit{Ss} \end{array}\right) & \Rightarrow & S_1 \neq S_2 \end{array}$$

For example, we have that:

$$\begin{array}{c} \mathit{collect}\,[\![\,(0, \langle \mathit{Func}, \mathit{Arg}\rangle), (0, \langle \mathit{Arg}, \mathit{Func}, \mathit{Arg}\rangle), (1, \langle \mathit{Arg}, \mathit{Arg}\rangle)\,]\!] \\ = \\ [\![\,(0, [\![\,\langle \mathit{Func}, \mathit{Arg}\rangle, \langle \mathit{Arg}, \mathit{Func}, \mathit{Arg}\rangle\,]\!]), (1, [\![\,\langle \mathit{Arg}, \mathit{Arg}\rangle\,]\!])\,]\!] \end{array}$$

Calling *replace loc S E* replaces the subexpression at position *loc* in E with S:

$$\begin{array}{rcl}
\mathit{replace}\ \langle\rangle\ S\ E & = & S \\
\mathit{replace}\ (\mathit{Body}; \mathit{loc})\ S\ (\lambda x.E) & = & \lambda x.(\mathit{replace}\ \mathit{loc}\ S\ E) \\
\mathit{replace}\ (\mathit{Func}; \mathit{loc})\ S\ (E_1\ E_2) & = & (\mathit{replace}\ \mathit{loc}\ S\ E_1)\ E_2 \\
\mathit{replace}\ (\mathit{Arg}; \mathit{loc})\ S\ (E_1, E_2) & = & E_1\ (\mathit{replace}\ \mathit{loc}\ S\ E_2)
\end{array}$$

The *replaces* function iterates the application of *replace* over a set of locations:

$$\begin{array}{rcl}
\mathit{replaces}\ T\ [\![\,]\!]\ S & = & T \\
\mathit{replaces}\ T\ ([\![\mathit{loc}]\!] + \mathit{locs})\ S & = & \mathit{replace}\ \mathit{loc}\ S\ (\mathit{replaces}\ T\ \mathit{locs}\ S)
\end{array}$$

3.3 Proof of correctness

To prove the correctness of this algorithm, we need to show that our definition of *appresolve once* satisfies the following conditions (repeated from Section 2.5.2.8). We first prove this assuming the above specification of *apps*, and then show that the implementation we have given for *apps* satisfies this specification.

Suppose that $\mathit{appresolve}\ \mathit{once}\ F\ E\ T\ =\ [\![\ Ys_1, ..., Ys_k\ ]\!]$. Then we require that:

{1} For all substitutions ϕ:

$$\begin{array}{c}
\mathit{etanormalise}\ (\mathit{once}\ (\mathit{reduce}'\ \mathit{once}\ (\phi F))\ (\mathit{reduce}'\ \mathit{once}\ (\phi E)))\ =\ T \\
\equiv \\
\bigvee_i \phi \vdash_{\mathit{once}} Ys_i
\end{array}$$

{2} For all substitutions ϕ:

$$(\phi \vdash_{\mathit{once}}\ Ys_i) \wedge (\phi \vdash_{\mathit{once}}\ Ys_j)\ \Rightarrow\ i = j\ .$$

$\{3\}$ The pattern variables in Ys_i are contained in the pattern variables of $F\ E$.

$\{4\}$ For each index i:

$$Ys_i \ll (F\ E \to T)\ .$$

In the same way as we write *step* for *reduce once*, we can abbreviate *reduce′ once* with *step′*. First, note that if F and E are η-normal, then so is *once* $F\ E$, since substitution cannot introduce η-redexes. Thus, we have that

$$\mathit{etanormalise}\,(\mathit{once}\,(\mathit{step}'\,F)\,(\mathit{step}'\,E))$$

is equal to

$$\mathit{once}\,(\mathit{step}'\,F)\,(\mathit{step}'\,E)$$

As a result, our proof obligation for condition $\{1\}$ is as follows:

$$\begin{array}{c} \mathit{once}\,(\mathit{step}'\,(\phi F)\,(\mathit{step}'\,(\phi E)) = T \\ \equiv \\ \exists\, Ys \in \mathit{appresolve}\ \mathit{once}\ F\ E\ T\ :\ \phi \vdash_{\mathit{once}} Ys \end{array}$$

Now,

$$\begin{array}{ll} & \mathit{once}\,(\mathit{step}'\,(\phi F)\,(\mathit{step}'\,(\phi E)) = T \\ \equiv & \quad \{\text{definition of } \mathit{once}\text{, let } F' = \mathit{step}'\,(\phi F) \text{ and } E' = \mathit{step}'\,(\phi E)\} \\ & \qquad (\exists B : F' = \lambda x.B \wedge (x := E')B = T) \\ & \quad \vee \quad (\neg(\exists B : F' = \lambda x.B) \wedge (F'\,E' = T)) \end{array}$$

We continue with the two disjuncts separately.

$$\begin{array}{ll} & \neg(\exists B : F' = \lambda x.B) \wedge (F'\,E' = T) \\ \equiv & \quad \{T \text{ is normal}\} \\ & F'\,E' = T \\ \equiv & \quad \{\text{definition of } \vdash_{\mathit{once}}\} \\ & \exists\, T_0, T_1\ :\ T_0\ T_1 = T \wedge \phi \vdash_{\mathit{once}} [\![\, F \to T_0, E \to T_1 \,]\!] \end{array}$$

For the other disjunct, we argue

$$
\begin{array}{ll}
& \exists B : F' = \lambda x.B \wedge (x := E')B = T \\
\equiv & \{\text{predicate logic}\} \\
& \exists B_0, B_1 : \quad F' = \lambda x.B_0 \\
& \qquad\qquad \wedge \; B_1 = E' \\
& \qquad\qquad \wedge \; (x := B_1)B_0 = T \\
\equiv & \{\text{property of substitution}\} \\
& \quad (\exists B_0, B_1 : \quad F' = \lambda x.B_0 \\
& \qquad\qquad \wedge \; B_1 = E' \\
& \qquad\qquad \wedge \; (x := B_1)B_0 = T \\
& \qquad\qquad \wedge \; x \text{ occurs in } B_0 \\
& \vee \;\; F' = \lambda x.T) \\
\equiv & \{\text{definitions of } F',\, E',\, \mathit{apps} \text{ and } \vdash_{\mathit{once}}\} \\
& \quad \exists (T_0, T_1) \in \mathit{apps}\ T \;:\; \phi \vdash_{\mathit{once}} [\![\, F \to T_0, E \to T_1 \,]\!] \\
& \vee \;\; \phi \vdash_{\mathit{once}} (F \to \lambda x.T)
\end{array}
$$

In the forward implication of the last step, we need to know that B_1 and $\lambda x.B_0$ are normal in order to apply the definition of *apps*. Normalness of B_1 follows from normalness of T (as B_1 is a subexpression of T). Similarly, normalness of B_0 follows from normalness of T. Because B_0 is normal, the abstraction $\lambda x.B_0$ can only fail to be normal by being an η-redex; but F' is not an η-redex (because it results from *step*$'$), and $\lambda x.B_0 = F'$.

Therefore, we have that

$$
\begin{array}{ll}
& \quad (\exists B : F' = \lambda x.B \wedge (x := E')B = T) \\
& \vee \;\; (\neg(\exists B : F' = \lambda x.B) \wedge (F'\,E' = T)) \\
\equiv & \{\text{combining the two derivations above}\} \\
& \quad (\exists T_0, T_1 : (T_0\ T_1) = T \wedge \phi \vdash_{\mathit{once}} [\![\, F \to T_0, E \to T_1 \,]\!]) \\
& \vee \;\; (\exists (T_0, T_1) \in \mathit{apps}(T) \;:\; \phi \vdash_{\mathit{once}} [\![\, F \to T_0, E \to T_1 \,]\!]) \\
& \vee \;\; (\phi \vdash_{\mathit{once}} [\![\, F \to \lambda x.T \,]\!]) \\
\equiv & \{\text{definition of } \mathit{appresolve}\}
\end{array}
$$

$$\exists\, Ys \in \mathit{appresolve\ once}\ F\ E\ T\ :\ \phi \vdash_{\mathit{once}} Ys$$

This completes the proof of soundness and completeness of *appresolve none*. The progress condition {4} is clearly satisfied since "F" and "E" together have one less symbol than "$F\ E$" (recall that the space of function application is considered a symbol). Condition {3} is obvious from the definition of *appresolve*, so it remains to prove condition {2}, that is that

$$\mathit{appresolve\ once}\ F\ E\ T \;=\; [\![\, Ys_0,\, Ys_1, \ldots,\, Ys_k \,]\!]$$

does not contain any redundant elements. We need to prove that

$$\phi \vdash_{\mathit{once}} Ys_i \wedge \phi \vdash_{\mathit{once}} Ys_j \;\Rightarrow\; i = j$$

This implication follows because each Ys_i contains a rule whose left-hand side is F:

$$(F \rightarrow E_i) \in Ys_i$$

Now observe that

$$\begin{array}{ll} & \phi \vdash_{\mathit{once}} Ys_i \wedge \phi \vdash_{\mathit{once}} Ys_j \\ \Rightarrow & \quad \{\text{since } (F \rightarrow E_i) \in Ys_i\} \\ & \phi \vdash_{\mathit{once}} (F \rightarrow E_i) \;\wedge\; \phi \vdash_{\mathit{once}} (F \rightarrow E_j) \\ \equiv & \quad \{\text{definition of } \vdash_{\mathit{once}}\} \\ & E_i = \mathit{step}'\,(\phi F) = E_j \end{array}$$

However, all the E_i are distinct, so $i = j$. To see that the E_i are distinct, recall that there are three cases to consider:

$$\begin{array}{ll} & (E_i\,,\, Y) \in \mathit{apps}\ T \\ \text{or} & E_j = \lambda x.T \quad \text{where } x \text{ is fresh} \\ \text{or} & E_k\ T_1 = T \end{array}$$

To show that these three cases are mutually exclusive, we argue:

$i \neq j$**:** Note that $E_i = \lambda x.B_x$ for some B_x, with x occurring in B_x. It follows that $E_i = \lambda x.B_x \neq \lambda x.T = E_j$.

$j \neq k$**:** If $E_j = E_k$, we have $E_j = \lambda x.T = \lambda x.E_k\ T_1 = \lambda x.(E_j\ T_1)$. An expression cannot occur inside itself, so this is a contradiction.

$i \neq k$**:** If $E_i = E_k$, then $E_i = (\lambda x.B_x) = E_k$ for some B_x. But this implies that $T = E_k\ T_1 = (\lambda x.B_x)\ T_1$ contains a β-redex. That contradicts the normalness of T.

3.3.1 Correctness of *apps*

The specification of *apps* is:

$$\begin{array}{rlll} (T_0, T_1) \in \mathit{apps}\ T \equiv & & T_0, T_1 \text{ normal} & \\ & \wedge & \exists B : (& T_0 = \lambda x.B \\ & & \wedge & (x := T_1)B = T \\ & & \wedge & x \text{ occurs in } B,\ x \text{ fresh}) \end{array}$$

Thus, to prove the correctness of our implementation, we need to prove the following equivalence:

$$\left(\begin{array}{ll} & \lambda x.B, T_1 \text{ normal} \\ \wedge & (x := T_1)B = T \\ \wedge & x \text{ occurs in } B,\ x \text{ fresh} \end{array}\right)$$
$$\equiv$$
$$\exists \mathit{locs}, \mathit{locs}' : \left(\begin{array}{ll} & (T_1, \mathit{locs}) \in \mathit{collect}\ (\mathit{subexps}\ T) \\ \wedge & \mathit{unboundlocals}\ T_1 \subseteq \mathit{unboundlocals}\ T \\ \wedge & \mathit{locs}' \subseteq \mathit{locs} \\ \wedge & \mathit{locs}' \neq \{\,\} \\ \wedge & B = \mathit{replaces}\ T\ \mathit{locs}'\ x \\ \wedge & \lambda x.B \text{ normal} \\ \wedge & x \text{ fresh} \end{array}\right)$$

The backwards implication is straightforward from the definitions of *subexps*, *collect* and *replaces*; note that the normalness of T guarantees the normalness of T_1. Forwards, the definition of substitution combined with the demand that x occurs in B ensures that T_1 is a subexpression of T. The rest follows by detailed consideration of the definition of substitution, *collect* and *replaces*, which is straightforward but rather tedious, and hence we omit it.

3.4 Related work : Second-order matching

The standard higher-order matching algorithm used in program transformation systems was first described in 1978 by Huet and Lang [47]. It operates on simply-typed λ-terms, and returns all second order matches (with the restriction that constants in the term can be of order no higher than three).

The algorithm centres around two procedures, *SIMPL* and *MATCH* which together perform a similar function to *resolve*. Essentially, the MATCH procedure deals with the situation where the pattern is an application with a flexible head or a pattern variable, and the SIMPL procedure deals with the other cases.

The underlying structure of the language on which it operates is somewhat different. All expressions are fully η-expanded and β-normal simply-typed λ-terms; thus substitution cannot introduce β-redexes, but must instead be defined to automatically apply β-reduction.

Here, we present Huet and Lang's algorithm in the same style as our own matching algorithms, with appropriate modifications. We start by defining the relevant notation and then present the algorithm.

3.4.1 Notation

There is a finite set of *ground* types, *e.g.* *Int*, *Char*, etc. These types are defined as having *order* 1. The function, or *derived* types are defined as follows: If $\alpha_1, \alpha_2 \ldots \alpha_n$ are either ground or derived types and β is a ground type, then $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \beta)$ is a derived type. Furthermore, its order is $1 + \max\{O(\alpha_i)\}$, where $O(\alpha)$ is the order of α.

The set of *expressions* is made up from *atoms*, *applications* and *abstractions*. The order of a expression is defined as the order of its type.

Atoms consist of *constant* symbols which we write a, b and c, *pattern* (free) variables which we write p, q and r, and *local* (bound) variables which we write x, y and z. As before, we take the slightly questionable step of using notation to make the distinction between each kind of atom, in the interests of simplicity.

If F is an atom of type $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \beta)$, and $T_1, T_2, \ldots, T_n$ are terms of type $\alpha_1, \alpha_2, \ldots, \alpha_n$ respectively, then the application denoted by $F(T_1, T_2, \ldots, T_n)$ is a expression of type β. The *head* of $F(T_1, T_2, \ldots, T_n)$ is F.

If T is an expression of type β, where β is a ground type, and $x_1, x_2, \ldots, x_n$ are local variables of type $\alpha_1, \alpha_2, \ldots, \alpha_n$ respectively, then the abstraction $\lambda x_1 x_2 \ldots x_n.T$ is a expression of type $(\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \beta)$.

Substitutions are partial functions from pattern variables to expressions; expressions in the range of substitutions need not be closed. When applying a substitution to an expression, it is necessary to apply β-reduction to ensure that the result is also a term.

Rules are denoted $P \rightarrow T$ as before. Since we are working with typed terms, the expressions P and T are required to have the same type. Since neither β- nor η-reduction can be expressed in the term language we are using, no normalness conditions are imposed on P and T.

3.4.2 Algorithm

We copy the *matches* function from Section 2.4 to form the basis of our presentation of Huet and Lang's algorithm; the only changes are to remove the parameter *app* and rename the *matches* and *resolve* functions appropriately:

$$\begin{aligned}
\mathit{huetmatches} &:: [\![\mathit{Rule}]\!] \to [\![\mathit{Subst}]\!] \\
\mathit{huetmatches}\,[\![\,]\!] &= [\![\mathit{idSubst}]\!] \\
\mathit{huetmatches}\,([\![X]\!] + \mathit{Xs}) &= [\![(\phi \circ \sigma) \mid (\sigma, \mathit{Ys}) \in \mathit{huetresolve}\, X, \\
&\qquad\qquad \phi \in \mathit{huetmatches}\,(\sigma\,(\mathit{Xs} + \mathit{Ys}))]\!]
\end{aligned}$$

The major differences between Huet and Lang's algorithm and our own algorithms become apparent with the definition of the *huetresolve* function in Table 3.1. Firstly, in keeping with the structure of the language which does not allow currying, functions are considered with all their arguments at once. Secondly, our algorithms only generate assignments for variables in one particular case, that where the pattern of a rule is a pattern variable. In contrast, Huet and Lang's algorithm also generates assignments whilst considering patterns that are function applications; these assignments in some senses do *part* of the work of calculating a value for the head of the pattern, whereas with our algorithms generated assignments give completed values for the appropriate pattern variable.

The first clause simply states that we can strip away the enclosing λs from both the pattern and the term. Note that the clauses are applied modulo renaming of local variables, and so the fact that the pattern and the term are type compatible and fully η-expanded means that λ-abstractions will match up exactly.

The second clause states that two constants match if they are the same and their arguments match. Type compatibility guarantees that if $c = d$ then $n = m$. The third clause says the same thing about local variables; again, if $y = z$ then $n = m$.

The fourth clause tells us that if we are matching a pattern variable p of ground type (which will therefore have no arguments attached) against a term T, then we can simply generate the obvious substitution for p. However we can only do this if T does not contain any local variables occurring without their enclosing λ; otherwise the substitution would move these variables out of scope.

X	$\mathit{huetresolve}\ X$
$(\lambda x_1 \ldots x_n.P) \rightarrow (\lambda x_1 \ldots x_n.T)$	$(\mathit{idSubst}, \llbracket\, P \rightarrow T \,\rrbracket)$
$c(T_1 \ldots T_n) \rightarrow d(T'_1 \ldots T'_m)$	$\llbracket\, (\mathit{idSubst}, \llbracket\, T_1 \rightarrow T'_1 \ldots T_n \rightarrow T'_n \,\rrbracket) \,\rrbracket$, if $c = d$ $\llbracket\,\rrbracket$, otherwise
$y(T_1 \ldots T_n) \rightarrow z(T'_1 \ldots T'_m)$	$\llbracket\, (\mathit{idSubst}, \llbracket\, T_1 \rightarrow T'_1 \ldots T_n \rightarrow T'_n \,\rrbracket) \,\rrbracket$, if $y = z$ $\llbracket\,\rrbracket$, otherwise
$p \rightarrow T$	$\llbracket\, (p := T, \llbracket\,\rrbracket) \,\rrbracket$, if T does not contain unbound local variables $\llbracket\,\rrbracket$, otherwise
$p(T_1 \ldots T_n) \rightarrow z(T'_1 \ldots T'_m)$	$\llbracket\, (p := \lambda x_1 \ldots x_n.x_i,$ $p(T_1 \ldots T_n) \rightarrow z(T'_1 \ldots T'_m))$ $\mid\ 1 \leq i \leq n;$ $T_i, z(T'_1 \ldots T'_m)$ type compatible $\rrbracket$
$p(T_1 \ldots T_n) \rightarrow c(T'_1 \ldots T'_m)$	$\llbracket\, (p := \lambda x_1 \ldots x_n.x_i,$ $p(T_1 \ldots T_n) \rightarrow c(T'_1 \ldots T'_m))$ $\mid\ 1 \leq i \leq n;$ $T_i, c(T'_1 \ldots T'_m)$ type compatible $\rrbracket$ $+\llbracket\, ((p := \mathit{imitate}_n\ c(T'_1 \ldots T'_m)),$ $p(T_1 \ldots T_n) \rightarrow c(T'_1 \ldots T'_m)) \,\rrbracket$

Table 3.1: Definition of *huetresolve*

The fifth clause states that when matching a second order pattern variable against a local variable, we generate all the *projections*, that is the functions which select one of their arguments, so long as they are type compatible. Generating just these functions suffices because the procedure is only aiming to return second-order matches, and anything more complex would lead to a result whose order was higher than two. Note that the rule is left unchanged; application by *huetmatches* of the substitution generated will lead to it later being broken down by the second clause.

Finally, the sixth clause tells us that when matching a second order pattern variable p against a constant c, we again generate all the type compatible projections, and in addition we generate an *imitation*, that is the most general match for p that contains c. Again, the rule itself is left unchanged in each case.

$$\mathit{imitate}_n\ c(T'_1, \ldots , T'_m) \quad = \quad \lambda x_1 \ldots x_n.c(S_1, \ldots , S_m)$$

The S_i values are calculated as follows; suppose that the type of T'_i is:

$$\alpha_1 \times \ldots \times \alpha_l \rightarrow \beta$$

Then we make q_i be a fresh free variable and define:

$$S_i \quad = \quad \lambda y_1 \ldots y_l.q_i(x_1, \ldots , x_n, y_1, \ldots , y_l)$$

3.4.3 Example

An example matching tree for the problem $p(q) \rightarrow 1 + 1$ is shown in Figure 3.2. In the tree, we have written $a + b$ as $+(a, b)$ to make it clearer which clause in the definition of *huetresolve* we are applying. As before, we always work on the first rule in any rule set. Since expressions in the range of a substitution need not be closed, we must be careful in which order substitutions are composed when calculating the results from a matching tree; substitutions are composed from the top of the tree downwards. For this

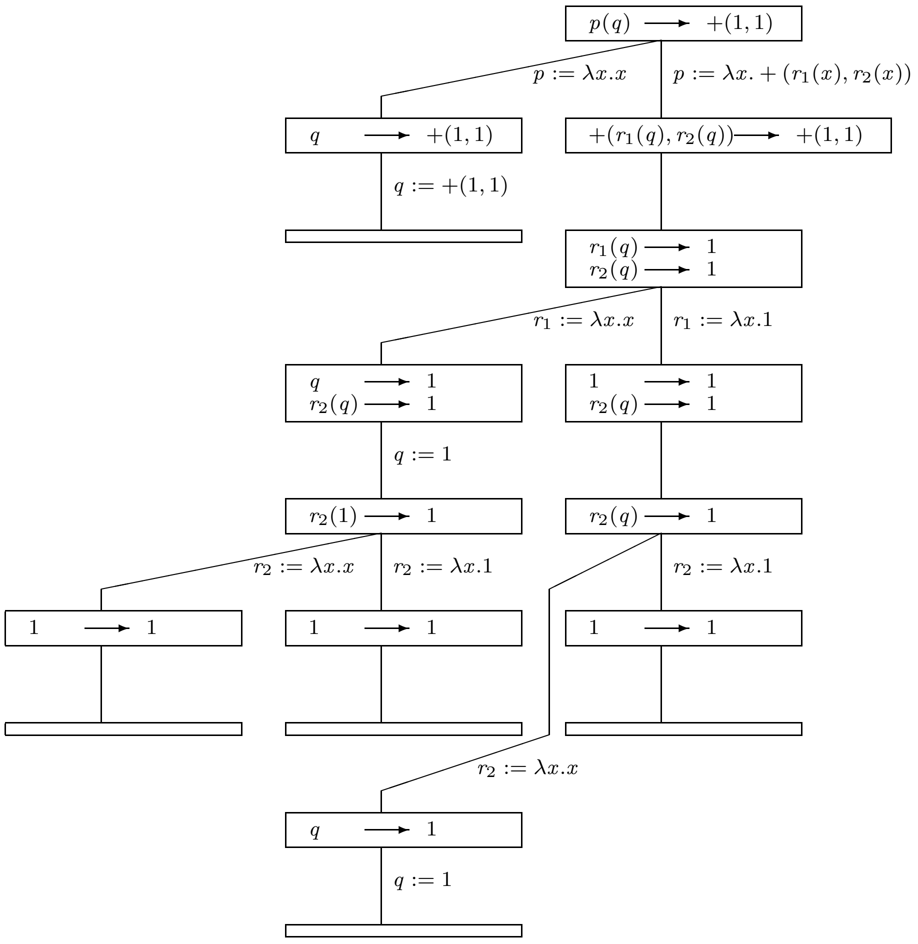

Figure 3.2: A Huet and Lang matching tree for $p(q) \rightarrow 1 + 1$

problem, we are left with the following results:

$$(p := \lambda x.x, q := 1 + 1)$$
$$(p := \lambda x.x + x, q := 1)$$
$$(p := \lambda x.x + 1, q := 1)$$
$$(p := \lambda x.1 + x, q := 1)$$
$$(p := \lambda x.1 + 1)$$

Notice that Huet and Lang's algorithm breaks down a matching problem in many more stages than the one-step algorithm. It also returns significantly less results – compare the above list with that given by the one-step algorithm for the same problem on page 72.

3.4.4 Discussion

From our perspective, Huet and Lang's algorithm has two major weaknesses. Firstly, as we have shown, it does not return the matches we require for transformations such as the fast reverse derivation. Secondly, it is limited to simply-typed λ-terms, although this can be extended to polymorphically-typed terms relatively straightforwardly by introducing type variables. This enhancement and extensions to the other calculi of Barendregt's λ-cube is described by Dowek in [30], which also removes the restriction that constants in the term must be third-order or lower.

Curien, Qian and Shi [23] describe and implement modifications to the algorithm which they claim speed it up significantly when the pattern contains many free variables or the term contains many bound variables, but has a worse performance when the term contains many constant symbols. Their changes "short-circuit" the creation of intermediate variables and in fact result in an algorithm that is somewhat similar to the one-step algorithm restricted to second-order variables, in that it also tries to form matches by abstracting subexpressions from the term being matched against.

We do not have any concrete performance comparisons between the one-step algorithm and Huet and Lang's algorithm; it would be difficult to do a direct comparison because of the larger set of results returned by the one-step algorithm. However, it should be possible to compare the average time taken per match returned, for example.

The key difference between the two algorithms that is likely to have a performance impact can be seen by comparing Figures 3.1 and 3.2, which show matching trees for the same problem. At each point in the matching tree where the pattern of the rule being considered is an application, the one-step algorithm immediately branches the tree in several different ways; if no subterms of the pattern are applications, then none of these branches will themselves branch further. In contrast, Huet and Lang's algorithm creates a limited number of branches (one for each "projection" and one for the "imitation"), and the new free variables introduced by this process ensure that more branching will occur later in the tree. In some cases, we might find that Huet and Lang's approach wins because entire branches can be cut off without fully exploring them; in others the one-step algorithm might save on the overhead of creating and matching new free variables. It seems likely that a performance comparison would yield similar results to those seen by Curien, Qian and Shi.

Chapter 4

Two-step matching

Although use of one-step matching allows us to apply many transformations, particularly those requiring cat-elimination, there are various examples of transformations for which the one-step algorithm is unable to find the necessary matches. One example of this is the minimum depth example from Section 1.2. To remedy this deficiency, we present the *two-step* algorithm, which returns a larger set of results than the one-step algorithm, but is only valid if certain extra restrictions on the pattern are satisfied. The set of results returned includes all third-order matches, but is guaranteed to be finite as a result of our restrictions. Used in conjunction with the one-step algorithm when these restrictions are not satisfied, it significantly extends the range of transformations that we are able to apply. We shall consider the minimum depth example in detail later on in this chapter as an example of a transformation that is enabled by this algorithm. This work has been published in [84].

4.1 Specification

For all rules $P \to T$, the pattern P is restricted as follows. For each subexpression $F\,E$ of P such that F is flexible, *i.e.* has a pattern variable or λ-abstraction as its head, E must satisfy the following conditions: Firstly

Pattern	*Valid?*	*Reason*
$p\,(\lambda x.x + x)$	Yes	Constant + and local variable x present in body
$p\,(\lambda x.x)$	No	No constant or external local variable in body
$p\,(\lambda x.0)$	No	Local variable x not in body
$p\,(\lambda x\ y.x)$	No	Local variable y not in body
$\lambda x.p\,(\lambda y.y\ x)$	Yes	External local x and local y present in body of $\lambda y.y\ x$
$p\,(\lambda x.x\ q)$	No	Pattern variable q present in $\lambda x.x\ q$

Table 4.1: Examples of applying the two-step matching restrictions to patterns

it must contain no pattern variables. Secondly, suppose $E = \lambda x_1 ... x_n.B$, where n is possibly 0, but B does not contain any outermost λs. Then each of $x_1 ... x_n$ must occur at least once in B, and B must contain at least one constant symbol, or alternatively a local variable that is bound in P outside $F\ E$. So for example, $p\,(\lambda x.x + x)$ is a valid pattern, but $p\,(\lambda x.x)$ is not because $\lambda x.x$ contains no constant or local variable other than x, and neither is $p\,(\lambda x.0)$ because the body of $\lambda x.0$ does not contain the local variable x. A fuller set of examples is given in Table 4.1.

These restrictions may seem somewhat arbitrary; the motivation for them is that given a rule whose pattern obeys them, only a finite number of substitutions will be returned by the algorithm for two-step matching that we shall present – we shall justify this claim later. We have found that they do not obstruct the application of the program transformations we are interested in; in the cases where they do not hold, the one-step matching algorithm can instead be applied successfully. Miller's higher-order patterns [65], which we discussed in Section 2.1, restrict the arguments to functions rather more drastically.

The two-step algorithm is specified by the *app* function *twice*:

$$\begin{aligned} \mathit{twice}\,(\lambda x.B)\,E &= \mathit{unmark}\,(\mathit{reduce}\,\mathit{markedonce}\,((x := \mathit{mark}\,E)B)) \\ \mathit{twice}\,E_1\,E_2 &= E_1\,E_2 \end{aligned}$$

As with *once*, a β-redex is reduced. However, this is then taken a stage further. Before the expression E is substituted into the body of B, all the *outer* λs are *marked* with the function *mark*, turning them into λ's:

$$\begin{aligned} \mathit{mark}\;c &= c \\ \mathit{mark}\;x &= x \\ \mathit{mark}\;p &= p \\ \mathit{mark}\,(\lambda x.E) &= \lambda' x.(\mathit{mark}\;E) \\ \mathit{mark}\,(E_1\,E_2) &= E_1\,E_2 \end{aligned}$$

The result of this is that *new* β-redexes that were introduced by the reduction of $(\lambda x.B)\;E$ have their λs marked. As a result, we can easily reduce precisely those redexes one more time. The function *markedonce* is just like *once* but only operates on redexes with marked λs:

$$\begin{aligned} \mathit{markedonce}\,(\lambda' x.B)\,E &= (x := E)B \\ \mathit{markedonce}\,E_1\,E_2 &= E_1\,E_2 \end{aligned}$$

Finally, some marks might have been left if a λ-abstraction was substituted into a position where it was not applied to an argument. A simple recursive traversal eliminates the marks:

$$\begin{aligned} \mathit{unmark}\;c &= c \\ \mathit{unmark}\;x &= x \\ \mathit{unmark}\;p &= p \\ \mathit{unmark}\,(\lambda x.E) &= \lambda x.(\mathit{unmark}\;E) \\ \mathit{unmark}\,(\lambda' x.E) &= \lambda x.(\mathit{unmark}\;E) \\ \mathit{unmark}\,(E_1\,E_2) &= (\mathit{unmark}\;E_1)\,(\mathit{unmark}\;E_2) \end{aligned}$$

We write $\mathit{twostep} = \mathit{reduce}\,\mathit{twice}$ and $\mathit{markedstep} = \mathit{reduce}\,\mathit{markedonce}$. Intuitively, *twostep* is a parallel reduction step in the same way as *step*; however the difference is that β-redexes are reduced twice, in that any redexes directly created by the reduction of the original redex are themselves reduced.

To illustrate, consider

$$\mathit{twostep}\,((\lambda x.x\,1)\,(\lambda y.y + y)) \quad = \quad 1 + 1$$

If we used *step* as defined in the previous chapter, we would have

$$\mathit{step}\,((\lambda x.x\,1)\,(\lambda y.y + y)) \quad = \quad (\lambda y.y + y)\,1$$

It should be noted that although very similar, $\mathit{twostep}\,E$ is not always the same as $\mathit{step}\,(\mathit{step}\,E)$, as the following (rather contrived) example shows:

Define the expression E by

$$E = (\lambda f.f\,1)\,(\lambda x\,g.x + g\,2)\,(\lambda y.y + 3)$$

Then we have that

$$\mathit{step}\,E = (\lambda x\,g.x + g\,2)\,1\,(\lambda y.y + 3)$$

and so it follows that

$$\mathit{step}\,(\mathit{step}\,E) = 1 + (\lambda y.y + 3)\,2$$

In contrast, applying *twostep* gives

$$\mathit{twostep}\,E = 1 + (2 + 3)$$

This example shows that there are some expressions where *twostep* reduces more β-redexes than applying *step* twice. However, there are also cases where the converse is true – consider the following:

$$E = (\lambda f.f\,(\lambda x.x + 1)\,2)\,(\lambda y.y)$$

We have that

$$\mathit{step}\,E = (\lambda y.y)\,(\lambda x.x + 1)\,2$$

and so applying *step* once more gives us

$$\mathit{step}\,(\mathit{step}\,E) = 2 + 1$$

In contrast,

$$\mathit{twostep}\,E = (\lambda x.x + 1)\,2$$

We previously showed that if using simple typing, then for rules with β-normal patterns, simple matching gives all first-order general matches and one-step matching all second-order general matches. In a similar vein, we now show that two-step matching produces all third-order general matches. Although third-order matching can in general produce an infinite set of matches, this is not the case for patterns that satisfy the restrictions we have outlined.

Before we give this proof, we note that two-step matching also produces some fourth-order matches. Consider for example the following rule:

$$p\,(\lambda x\; g.x + g\,2)\,(\lambda y.y + 3) \rightarrow 1 + (2 + 3)$$

The match $(p := \lambda f.f\,1)$ is a fourth-order general match for this rule, since $(\lambda x\; g.x + g\,2)$ is a third-order function. As we showed earlier, we have that

$$\mathit{twostep}\,((\lambda f.f\,1)\,(\lambda x\; g.x + g\,2)\,(\lambda y.y + 3)) = 1 + (2 + 3)$$

Thus, this match is also a two-step match.

Returning to our claim that two-step matching produces all third-order general matches, our proof proceeds by contradiction, along similar lines to that in Section 3.1. Suppose ϕ is a third-order general match for a rule $P \rightarrow T$, with P β-normal, and further suppose that ϕ does not satisfy the two-step matching specification $\mathit{twostep}\,(\phi P) \simeq T$. Then $\mathit{twostep}\,(\phi P)$ must contain a β-redex, and applying Lemma 4.2 below tells us that ϕP must contain a β-redex whose left-hand side is of order 4 or higher, which must have been introduced by ϕ, since P is β-normal. Thus ϕ is not a third-order match as we had supposed.

Before proving Lemma 4.2, we need to prove the following lemma about *markedstep*:

Lemma 4.1. *If markedstep T contains a β-redex $(\lambda x.B)\,E$, where $\lambda x.B$ is an expression of order n, then either T contains a β-redex $(\lambda' y.C)\,F$, where $\lambda' y.C$ is an expression of order $n{+}1$ or higher, or a β-redex $(\lambda x.B')\,E'$ where $\lambda x.B'$ is of order n or higher.*

Proof. By induction on the size of T. Suppose that the result is true for all expressions smaller than T. By Lemma 2.2, there exists $(T_0\ T_1) \trianglelefteq T$ such that:

$$(\lambda x.B)\,E \quad \trianglelefteq \quad \mathit{markedonce}\,(\mathit{markedstep}\ T_0)\,(\mathit{markedstep}\ T_1)$$

If *markedstep* T_0 is not a λ'-abstraction, then:

$$\mathit{markedonce}\,(\mathit{markedstep}\ T_0)\,(\mathit{markedstep}\ T_1)$$
$$=$$
$$(\mathit{markedstep}\ T_0)\,(\mathit{markedstep}\ T_1)$$

If *markedstep* $T_0 = \lambda x.B$ we apply part (1) of Lemma 3.2 to show that either T_0 contains the required β-redex or that $(T_0\ T_1)$ is the required β-redex. Otherwise, either $(\lambda x.B)\,E \trianglelefteq \mathit{step}\ T_0$ or $(\lambda x.B)\,E \trianglelefteq \mathit{step}\ T_1$, and we apply the induction hypothesis.

If *markedstep* $T_0 = \lambda' y.C$, then:

$$\mathit{markedonce}\,(\mathit{markedstep}\ T_0)\,(\mathit{markedstep}\ T_1)$$
$$=$$
$$(y := \mathit{markedstep}\ T_1)\ C$$

Either:

- $(\lambda x.B)\,E \trianglelefteq \mathit{markedstep}\ T_1$ (and y occurs in C)

  The result follows from the induction hypothesis.

- $(\lambda x.B')\,E' \trianglelefteq C$

  Then $(y := \mathit{markedstep}\ T_1)B' = B$ and $(y := \mathit{markedstep}\ T_1)E' = E$. Since substituting for an unbound local variable does not affect the order of an expression, the result again follows from the induction hypothesis.

- *markedstep* $T_1 = \lambda x.B$ and $y\, E' \trianglelefteq C$

  Then $(y := \mathit{markedstep}\ T_1)E' = E$. We have that:

  $$(\mathit{markedstep}\ T_0)\,(\mathit{markedstep}\ T_1) \;=\; (\lambda' y.C)\,(\lambda x.B)$$

  Since $\lambda x.B$ is of order n, $\lambda' y.C$ is of order $n+1$. By (1) of Lemma 3.2 either T_0 is a λ-abstraction of order $n+1$ whence $(T_0\ T_1)$ is the required β-redex, or T_0 contains the required β-redex.

□

Lemma 4.2. *If twostep T contains a β-redex $(\lambda x.B)\,E$, where $\lambda x.B$ is an expression of order n, then T contains a β-redex whose left-hand side is an expression of order $n+2$ or higher.*

Proof. By induction on the size of T. Assume that the result holds for all expressions smaller than T. By Lemma 2.2,

$$\exists (T_0\ T_1) \trianglelefteq T \;:\; (\lambda x.B)\,E \trianglelefteq \mathit{twice}\,(\mathit{twostep}\ T_0)\,(\mathit{twostep}\ T_1)$$

If *twostep* T_0 is not a λ-abstraction, then

$$\mathit{twice}\,(\mathit{twostep}\ T_0)\,(\mathit{twostep}\ T_1) = (\mathit{twostep}\ T_0)\,(\mathit{twostep}\ T_1)$$

Then either $(\lambda x.B)\,E \trianglelefteq \mathit{twostep}\ T_0$ or $(\lambda x.B)\,E \trianglelefteq \mathit{twostep}\ T_1$, and we apply the induction hypothesis.

Otherwise, *twostep* $T_0 = \lambda y.C$. Let $C' = (y := \mathit{mark}\,(\mathit{twostep}\ T_1))C$, then

$$\mathit{twice}\,(\mathit{twostep}\ T_0)\,(\mathit{twostep}\ T_1) \;=\; \mathit{unmark}\,(\mathit{markedstep} C')$$

By Lemma 4.1, there is a redex R such that $R \trianglelefteq C'$ and either $R = (\lambda x.B)\,E$ or $R = (\lambda' z.D)\ G$, for some z, D, G where $\lambda' z.D$ is of order $n+1$ or higher.

Either:

- $R \trianglelefteq \mathit{twostep}\ T_1$ (and y occurs in A). We apply the induction hypothesis.

- $R' \trianglelefteq C$, with $(y := \mathit{twostep}\ T_1)R' = R$. Since substituting for an unbound local variable does not affect the order of an expression, we can again use the induction hypothesis.

- $\mathit{mark}(\mathit{twostep}\ T_1)$ is the λ-abstraction part of R. In this case R is $(\lambda' z.D)\ G$ (since *mark* will not leave an outermost λ unmarked), and there exists G' such that $z\ G' \trianglelefteq C$ and $(y := \mathit{mark}\,(\mathit{twostep}\ T_1))\,G' = G$. Therefore, $\mathit{mark}\,(\mathit{twostep}\ T_1)$ is of order $n+1$ or higher, and so $\lambda y.C = \mathit{twostep}\ T_0$ is of order $n+2$ or higher. By (2) of Lemma 3.2 either T_0 is a λ-abstraction of order $n{+}2$ whence $(T_0\ T_1)$ is the required β-redex, or T_0 contains the required β-redex.

□

4.1.1 Example : *mindepth*

Applying promotion to the *mindepth* example from Section 1.2 leads to the following matching problem whilst verifying the second side-condition of the binary tree promotion law (see Section 6.1 for details on how this problem is obtained).

$$
\begin{array}{l}
\lambda t_1\ t_2 . f\ (\lambda d_1.\mathit{min}\ (\mathit{mindepth}\ t_1 + d_1))\ (\lambda d_2.\mathit{min}\ (\mathit{mindepth}\ t_2 + d_2)) \\
\rightarrow \\
\lambda t_1\ t_2\ d\ m.\ \mathbf{if}\ 1 + d\ \geq\ m \\
\qquad\qquad\quad \mathbf{then}\ m \\
\qquad\qquad\quad \mathbf{else}\ \ \mathit{min}\ (\mathit{mindepth}\ t_1 + (1+d)) \\
\qquad\qquad\qquad\qquad\qquad (\mathit{min}\ (\mathit{mindepth}\ t_2 + (1+d))\ m)
\end{array}
$$

The only substitution that solves this problem is the following:

$$
\begin{array}{l}
f\ :=\ \lambda g\ h\ d\ m.\ \mathbf{if}\ 1 + d\ \geq\ m \\
\qquad\qquad\qquad\quad \mathbf{then}\ m \\
\qquad\qquad\qquad\quad \mathbf{else}\ \ g\ (1+d)\ (h\ (1+d)\ m)
\end{array}
$$

If we apply this substitution to the pattern of the rule above and apply *step*, we find that we obtain the expression

$$\begin{array}{l} \lambda t_1\ t_2\ d\ m.\ \textbf{if}\ 1 + d\ \geq\ m \\ \qquad\qquad \textbf{then}\ m \\ \qquad\qquad \textbf{else}\ \ (\lambda d_1.\mathit{min}\ (\mathit{mindepth}\ t_1 + d_1))\ (1 + d) \\ \qquad\qquad\qquad\qquad (\lambda d_2.\mathit{min}\ (\mathit{mindepth}\ t_2 + d_2)\ (1 + d)\ m) \end{array}$$

In other words, the one-step algorithm is not able to solve this problem. In contrast, the extra reduction carried out by *markedstep* means that if we apply *twostep* instead we obtain the term of the rule, as required; thus the two-step algorithm is adequate for applying promotion to *mindepth*.

4.2 Algorithm

The two-step algorithm is implemented as follows:

$$\begin{array}{rcl} \mathit{appresolve\ twice}\ F\ E\ T & = & [\![\, [\![\, F \rightarrow \mathit{etaRed}(\lambda x.B)\,]\!] \mid \\ & & \quad x \text{ fresh}, B \leftarrow \mathit{abstracts}\ x\ E\ T\,]\!] \\ & & \text{if } F \text{ flexible} \\ \mathit{appresolve\ twice}\ F\ E\ (T_1\ T_2) & = & [\![\, [\![\, F \rightarrow T_1, E \rightarrow T_2\,]\!]\,]\!] \\ & & \text{if } F \text{ not flexible} \\ \mathit{appresolve\ twice}\ F\ E\ T & = & [\![\,]\!] \end{array}$$

As with *appresolve once*, this definition takes the term and tries to write it as an application, and then matches F and E against the function and argument parts of the application respectively. Either the term is already an application, or we can turn it into one by constructing a β-redex that reduces to it. If F is not flexible, it will not match against a λ-abstraction, so there is no point in trying to construct a β-redex; if the term is also an application we simply match up the functions and arguments respectively, and if not we do nothing.

If F is flexible, then E will not have any pattern variables, because we imposed that as a restriction, and thus any β-redex we construct to match against has E as the argument part. For the function part, we construct a new λ-abstraction, using *abstracts* to give possible values for the body of the abstraction. The *abstracts* function plays a somewhat similar role to that of *apps* in the one-step algorithm; however in this case we do not separate the case where the term T is already an application from other cases. As a result, if T is the application $T_0\,E$, then one body returned by *abstracts* will be the expression $T_0\,x$, and thus $\lambda x.B$ would be an η-redex. It is for this reason that top-level η-redexes are reduced by use of *etaRed*.

As before, we shall first give a specification for *abstracts* and later define it. The result of *abstracts* $x\ E\ T$ should contain all η-normal expressions B such that

$$\mathit{unmark}\,(\mathit{markedstep}\,((x := \mathit{mark}\ E)\ B)) \simeq T$$

In words, *abstracts* $x\ E\ T$ is specified to give exactly those expressions that will be reduced to T by *markedstep* when x is replaced by *mark* E. Since *markedstep* reduces marked β-redexes, and the only marks will be those on the outer λs of E, this is equivalent to saying that *abstracts* $x\ E\ T$ returns all the expressions that give T when each occurrence of x is replaced by E and a β-reduction pass is then performed over the new occurrences of E and all their arguments.

For example, if T is $1 + (0 + 0)$ and E is $\lambda y.y + y$, then a correct implementation of *abstracts* would give:

$$\mathit{abstracts}\ x\,(\lambda y.y + y)\,(1 + (0 + 0)) \quad = \quad [\![\, 1 + (0 + 0), 1 + x\,0 \,]\!]$$

Figure 4.1 shows a two-step matching tree for the rule $p\,(\lambda y.y + y) \rightarrow 1 + (0 + 0)$. In general, two-step matching trees exhibit much less branching than one-step matching trees (such as that in Figure 3.1) because of our restrictions on the pattern.

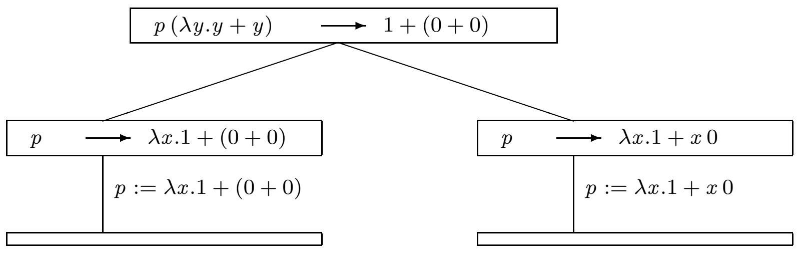
$p\,(\lambda y.y+y) \longrightarrow 1+(0+0)$
$p \longrightarrow \lambda x.1+(0+0)$
$p := \lambda x.1+(0+0)$
$p \longrightarrow \lambda x.1+x\,0$
$p := \lambda x.1+x\,0$

Figure 4.1: A two-step matching tree for $p\,(\lambda y.y + y) \to 1 + (0 + 0)$

4.2.1 Defining *abstracts*

As we remarked above, *abstracts* $x\ E\ T$ should contain all expressions that will give T when x applied to a set of arguments is replaced by the result of β-reducing E applied to the same arguments. For example, applying this procedure to the expression $x\,1$ where $E = \lambda y.y + y$ would give $1 + 1$.

Clearly, one candidate for such an expression is T itself. However, we can also replace any subexpression of T that can be obtained by the above procedure with an appropriate application of x to a set of arguments. For example, in the above situation any occurrence of $1+1$ in T could be replaced with $x\,1$. We call such a subexpression an *instance* of E in T; the procedure of replacing an instance with the variable x applied to an appropriate set of arguments is known as *abstracting* instances.

This procedure is somewhat complicated by the fact that instances may overlap; for example consider $T = (1 + 1) + (1 + 1)$ and $E = \lambda y.y + y$; then both the entire term T and the two occurrences of $(1 + 1)$ are instances of E in T. It is for this reason that our algorithm searches for instances by an iterative process, which we shall now describe.

The results of the functions $abstracts_n\ x\ E\ T$ are specified as being all the results of *abstracts* $x\ E\ T$ which have had precisely n instances of E abstracted from T. As we shall see shortly, $abstracts_n$ can be implemented by a recursive procedure, and thus the results of *abstracts* can be found by taking the union of all the results of the individual $abstracts_n$ functions. Because the recursive procedure for generating the results of $abstracts_n$ will inevitably result in duplicates, we choose to depart from the policy of using bags we decided on in Section 2.4 and give our algorithms in terms of sets. Thus, any practical implementation will have to be careful to explicitly remove duplicate elements from the results generated.

We define *abstracts* as follows:

$$\textit{abstracts}\, x\ E\ T \quad = \quad \cup\{\, \textit{abstracts}_n\, x\ E\ T \mid n = 0 \ldots \}$$

At first sight, this definition would appear to be non-terminating, since n could be any natural number. However, recall the conditions that our restrictions on the pattern impose on E; each formal parameter of E must appear at least once in E's body, and E must contain at least one constant symbol or local variable whose binding λ in the pattern occurs outside E. As a result, each instance of E in T will contain at least one more occurrence of this symbol than the expression involving x that it is replaced with, since any occurrence of this symbol in the arguments to x would also appear in the expression that resulted when these arguments were substituted into the body of E, together with the original occurrence in the body of E.

Thus, if m is the number of occurrences of this symbol in T, then for all $n > m$, the set $\textit{abstracts}_n\, x\ E\ T$ is empty. As a result, although the definition appears to be infinite, it is easy to implement in a terminating manner.

Defining $\textit{abstracts}_n$ itself is relatively straightforward. Abstracting 0 instances of E from T can only give T itself. To abstract $n + 1$ instances, we first abstract n instances and then look for ways in which one more can be abstracted. To facilitate the proof of correctness we shall give later, we use a different variable y when abstracting this new instance, and then rename it to x; a practical implementation could omit this step and just use x throughout.

$$\begin{array}{rcll}
\textit{abstracts}_0\, x\ E\ T & = & \{\, T \,\} & \\
\textit{abstracts}_{(n+1)}\, x\ E\ T & = & \{\, (y := x)C \mid & B \in \textit{abstracts}_n\, x\ E\ T, \\
 & & & C \in \textit{abstract}\, x\ y\ E\ B, \\
 & & & y \text{ fresh} \,\}
\end{array}$$

The function *abstract* is responsible for abstracting precisely one instance of E. For each subexpression S of T, it calls *instance* to establish whether S is an instance of E, and if so it replaces S with an appropriate expression, which is returned by instance in the event of success.

It is here that the complication raised by overlapping instances shows itself; we wish to keep track of the number of instances abstracted by counting the occurrences of x in the result, but consider the following call to $\mathit{abstracts}_2$:

$$\mathit{abstracts}_2\, x\, (\lambda z.z + z)\, ((1+1) + (1+1))$$

One of its members is the expression $x\,1 + x\,1$, and abstracting one further instance of $\lambda z.z+z$ from it gives us $x\,(x\,1)$, which also has two occurrences of x. Therefore, we make the stipulation that instances should be abstracted in an *outermost* order, and enforce this by stipulating that x should not occur in any subexpression S that is a candidate for abstraction. Thus, in the above example, the above sequence would be ruled out and we would have the following sequence of abstractions to obtain $x\,(x\,1)$:

$$(1+1) + (1+1) \rightarrow x\,(1+1) \rightarrow x\,(x\,1)$$

A second complication also raises its head here. Suppose that E is the function $\lambda w\; z.z+w$ and that T contains the subexpression $\lambda v.v+1$. Then $v+1$ is an instance of E which can be replaced by $y\,1\,v$, which will lead to a result containing the subexpression $\lambda v.y\,1\,v$. This is not η-normal, which violates our specification of *abstracts*. Therefore, after an instance is abstracted, we call *etanormalise* on the result.

$$\begin{array}{lcll} \mathit{abstract}\; x\; y\; E\; T & = & \{\; \mathit{etanormalise}\,(\mathit{replace}\; \mathit{loc}\; R\; T) & \\ & & \mid\; (S, \mathit{loc}) \in \mathit{subexps}\; T & \\ & & \quad x \not\trianglelefteq S & \\ & & \quad R \in \mathit{instance}\; y\; E\; S & \} \end{array}$$

Finally, it remains to define *instance*. Suppose that E is an expression of the form $\lambda x_1 \ldots x_m.B$, where B does not contain any outer λs. Then, the expression S is an instance of E precisely if there are some $E_1 \dots E_m$ such that:

$$S = (x_1 := E_1, \dots\ x_m := E_m)B$$

We can ascertain whether this is the case or not by using the simple matching algorithm from Section 2.6. Recall that our conditions on E mean that B cannot contain any pattern variables and must contain at least one occurrence of each of the x_is, thus if we treat $x_1 \dots x_m$ as pattern variables the simple matching algorithm will return a substitution ϕ whose domain is precisely these variables, such that each E_i from above is given by ϕx_i. We can then replace S by $y\, E_1 \dots E_m$.

$$\begin{array}{lcl} \mathit{instance}\ y\ E\ S & = & \{\ y\ (\phi\ x_1) \dots (\phi\ x_m) \\ & & \quad \mid\ \ (x_1, \dots, x_m) = \mathit{params}\ E \\ & & \quad\quad \phi \in \mathit{matches\ none}\ [\![\ \mathit{body}\ E \rightarrow S\ ]\!] \quad \} \end{array}$$

The functions *params* and *body* express E as $\lambda x_1 \dots x_m.B$ as above and return $(x_1, \dots, x_m)$ and B respectively, and are easy to define. Note that any local variables that occur in *body* E or T without their binding lambdas should be treated as *constants* during the application of simple matching; since the substitution ϕ is applied immediately, any occurrences of such local variables in the range of ϕ will not cause a variable capture problem.

We now give a simple example of applying *abstracts*. We take T to be $1 + ((0 + 0) + (0 + 0))$ and E to be $\lambda z.z + z$.

The definition of $\mathit{abstracts}_0$ gives us:

$$\mathit{abstracts}_0\ x\ E\ T = \{\ 1 + ((0 + 0) + (0 + 0))\ \}$$

In order to compute $\mathit{abstracts}_{(n+1)}\ x\ E\ T$, we choose a fresh y and evaluate $\mathit{abstract}\ x\ y\ E\ B$ for each B in $\mathit{abstracts}_0\ x\ E\ T$. The only value for B for $n = 0$ is T itself; the first step is to select all subexpressions S of T which do not contain x, and calculate $\mathit{instance}\ y\ E\ S$ for each.

To evaluate *instance* for each subexpression, we will need to split up E into the list of formal parameters (z) and the body $z + z$. With this done, we can see that *instance* $y\,E\,S$ will only give results if there are simple matches between $z + z$ and S, treating z as a pattern variable. Thus, the subexpressions that will produce results from *instance* are $0 + 0$ (twice) and $(0+0)+(0+0)$. In the first case, the result will be $\{\, y\,0 \,\}$ and in the second it will be $\{\, y\,(0+0) \,\}$.

So, using these results to calculate *abstract* $x\;y\;E\;T$, we have the set

$$\{\, 1 + y\,(0+0), 1 + (y\,0 + (0+0)), 1 + ((0+0) + y\,0) \,\}$$

Renaming the variable y to x gives us the value of $\mathit{abstracts}_1\,x\;E\;T$:

$$\{\, 1 + x\,(0+0), 1 + (x\,0 + (0+0)), 1 + ((0+0) + x\,0) \,\}$$

We now repeat the process for each element of this new set, obtaining the following set for $\mathit{abstracts}_2\,x\;E\;T$:

$$\{\, 1 + x\,(x\,0), 1 + (x\,0 + x\,0) \,\}$$

Notice that the second element of this set is obtained starting from both the second and third elements of $\mathit{abstracts}_1\,x\;E\;T$.

Attempting to iterate one more time, we find that $\mathit{abstracts}_3\,x\;E\;T$ is empty. Although the subexpression $(x\,0 + x\,0)$ of the second element of the above set would produce a result if passed to *instance*, it violates the condition that x should not occur in such subexpressions.

Thus, there is no need to calculate $\mathit{abstracts}_n\,x\;E\;T$ for $n \geq 4$. Combining the above results gives us:

$$\begin{array}{lcll} \mathit{abstracts}\,x\;E\;T & = & \{ & 1 + ((0+0) + (0+0)), \\ & & & 1 + x\,(0+0), \\ & & & 1 + (x\,0 + (0+0)), \\ & & & 1 + ((0+0) + x\,0), \\ & & & 1 + (x\,(x\,0)), \\ & & & 1 + (x\,0 + x\,0) \qquad\qquad \} \end{array}$$

4.3 Proof of correctness

We now move onto proving that the above algorithm satisfies the specification we have given. The details are rather technical, and readers who are only interested in using our algorithm may wish to skip to Section 4.4.

We refer to the conditions on *appresolve* from page 59. Once again, conditions {3} and {4}, which respectively restrict the set of variables in the newly generated rules and require that we make progress, are easily satisfied.

For the non-redundancy condition {2}, suppose that:

$$\mathit{appresolve}\ \mathit{twice}\ F\ E\ T = [\![\ \mathit{Ys}_1, \ldots , \mathit{Ys}_k\ ]\!]$$

If F is not flexible then $k = 1$ and the condition is trivially satisfied. Otherwise, suppose there exists ϕ such that $\phi \vdash_{\mathit{twice}} \mathit{Ys}_i$ and $\phi \vdash_{\mathit{twice}} \mathit{Ys}_j$. Then there exist $B_i, B_j \in \mathit{abstracts}\ x\ E\ T$ such that

$$\mathit{etaRed}\,(\lambda x.B_i) = \mathit{twostep}'\,(\phi\, F) = \mathit{etaRed}\,(\lambda x.B_j)$$

If B_i is of the form $C_i\, x$ with x not free in C_i, then $\mathit{etaRed}\,(\lambda x.B_i) = C_i$, and otherwise $\mathit{etaRed}\,(\lambda x.B_i) = \lambda x.B_i$. The same is true of B_j, so if they are either both of this form or both not of this form then $B_i = B_j$ and thus $i = j$ since $\mathit{abstracts}\ x\ E\ T$ is a set. Otherwise, assume without loss of generality that B_i is of this form and B_j is not. Then $C_i = \lambda x.B_j$, and so B_i contains a β-redex. From the specification of *abstracts*,

$$\mathit{unmark}\,(\mathit{markedstep}\,((x := \mathit{mark}\ E)B_i)) \simeq T$$

Since only β-redexes with marked λs are reduced by *markedstep*, and marked λs are only introduced by *mark*, T contains a β-redex, which violates the condition that T is normal.

We are now left with the soundness and completeness condition {1}. Unfortunately, this proof is rather involved, due to the complexity of the definition of *abstracts*. We have broken it down into what we hope are reasonably self-contained parts, each proving intuitively believable results.

We start by making use of *reduce*′ from Section 2.5.2.1 to simplify the proof obligations. Recall that *reduce*′ mirrored the definition of *reduce*, but removed η-redexes at any point where they might otherwise have been generated. Define $\mathit{markedstep}' = \mathit{reduce}'\ \mathit{markedonce}$ and $\mathit{twostep}' = \mathit{reduce}'\ \mathit{twice}$, and twice' by:

$$\begin{aligned}\mathit{twice}'\,(\lambda x.B)\,E &= \mathit{unmark}\,(\mathit{markedstep}'\,((x := \mathit{mark}\,E)B))\\ \mathit{twice}'\,E_1\,E_2 &= E_1\,E_2\end{aligned}$$

Then, since for all F, $\mathit{markedstep}'\,F = \mathit{etanormalise}\,(\mathit{markedstep}\,F)$,

$$\begin{gathered}\mathit{etanormalise}\,(\mathit{twice}\,(\mathit{reduce}'\,\mathit{twice}\,(\phi F))\,(\mathit{reduce}'\,\mathit{twice}\,(\phi E)))\\ =\\ \mathit{twice}'\,(\mathit{twostep}'\,(\phi F)\,(\phi E))\end{gathered}$$

Thus, our proof obligation is reduced to showing the following for all substitutions ϕ:

$$\begin{gathered}\mathit{twice}'\,(\mathit{twostep}'\,(\phi F))\,(\mathit{twostep}'\,(\phi E)) = T\\ \equiv\\ \exists\, Ys \in \mathit{appresolve}\ \mathit{twice}\ F\ E\ T\ :\ \phi \vdash_{\mathit{twice}} Ys\end{gathered}$$

Now,

$$\begin{aligned}&Ys \in \mathit{appresolve}\ \mathit{twice}\ F\ E\ T\\ \equiv\quad &\{\text{definition of } \mathit{appresolve}\ \mathit{twice}\}\\ &\left(F \text{ flexible} \wedge \begin{array}{c}\exists x \text{ fresh}, B \in \mathit{abstracts}\ x\ E\ T\ :\\ Ys = [\![\, F \to \mathit{etaRed}(\lambda x.B)\,]\!]\end{array}\right)\\ \vee\ &\left(\begin{array}{ll} & F \text{ not flexible}\\ \wedge & \exists T_0, T_1\ :\ T = T_0\ T_1\\ \wedge & Ys = [\![\, F \to T_0, E \to T_1\,]\!]\end{array}\right)\end{aligned}$$

We can now continue separately for the cases where F is flexible and when it is not. If F is not flexible, then:

$$\begin{aligned}&\mathit{twice}'\,(\mathit{twostep}'\,(\phi F))\,(\mathit{twostep}'\,(\phi E)) = T\\ \equiv\quad &\{\text{if } F \text{ is not flexible } \mathit{twostep}'\,(\phi F) \text{ is not a } \lambda\text{-abstraction}\}\\ &(\mathit{twostep}'\,(\phi F))\,(\mathit{twostep}'\,(\phi E)) = T\end{aligned}$$

$$\equiv \quad \{\text{definition of } =\}$$
$$\exists T_0, T_1 \,:\, T = T_0\ T_1 \wedge \mathit{twostep}'\,(\phi F) = T_0 \wedge \mathit{twostep}'\,(\phi E) = T_1$$
$$\equiv \quad \{\text{definition of } \vdash_{\mathit{twice}}\}$$
$$\exists T_0, T_1 \,:\, T = T_0\ T_1 \wedge \phi \vdash_{\mathit{twice}} [\![\, F \to T_0, E \to T_1 \,]\!]$$

For the case where F is flexible, we define the abbreviation ms' as follows:

$$\mathit{ms}'\,A \quad = \quad \mathit{unmark}\,(\mathit{markedstep}'\,A)$$

We also define $E' = \mathit{mark}\,E$. Recall that the specification of *abstracts* states that for all $B \in \mathit{abstracts}\,x\ E\ T$,

$$\mathit{unmark}\,(\mathit{markedstep}\,((x := \mathit{mark}\ E)B_i) \simeq T$$

Since T is η-normal, we can now write this as:

$$\mathit{ms}'\,((x := E')B) = T$$

We now argue as follows:

$$\mathit{twice}'\,(\mathit{twostep}'\,(\phi F))\,(\mathit{twostep}'\,(\phi E)) = T$$
$$\equiv \quad \{\text{let } F' = \mathit{twostep}'\,(\phi F)\text{, conditions on the pattern mean } \phi E = E,\ \mathit{twostep}'\,E = E\}$$
$$\mathit{twice}'\,F'\,E = T$$
$$\equiv \quad \{\text{definition of } \mathit{twice},\ E',\ \mathit{ms}'\}$$
$$\begin{pmatrix} \text{case } F' \text{ of} & & \\ (\lambda x.B) & \to & \mathit{ms}'\,((x := E')\,B) \\ \_ & \to & F'\,E = T \end{pmatrix}$$
$$\equiv \quad \{\text{semantics of case, } F'\ \eta\text{-normal, } \phi F \text{ closed} \Rightarrow F' \text{ closed}\}$$
$$\begin{pmatrix} & & \mathit{ms}'\,((x := E')\,B) = T \\ \exists B. & \wedge & F' = \lambda x.B \\ & \wedge & F' \text{ not an } \eta\text{-redex} \end{pmatrix}$$
$$\vee$$
$$\begin{pmatrix} & F'\,E = T \\ \wedge & F' \text{ not a } \lambda\text{-abstraction} \end{pmatrix}$$

$$
\begin{array}{ll}
\equiv & \{\text{definition of substitution, } \mathit{ms}',\ T \text{ closed and normal}\} \\
& \exists B. \begin{array}{c}\left(\begin{array}{ll} & \mathit{ms}'\,((x := E')\,B) = T \\ \wedge & F' = \lambda x.B \\ \wedge & F' \text{ not an } \eta\text{-redex}\end{array}\right) \\ \vee \\ \left(\begin{array}{ll} & \mathit{ms}'\,((x := E')\,B) = T \\ \wedge & B = F'\,x \\ \wedge & x \not\trianglelefteq F'\end{array}\right)\end{array} \\
\equiv & \{\text{definition of } \mathit{etaRed},\ F'\ \eta\text{-normal}\} \\
& \exists B.(\mathit{ms}'\,((x := E')\,B) = T \wedge F' = \mathit{etaRed}\,(\lambda x.B) \wedge B\ \eta\text{-normal}) \\
\equiv & \{\text{specification of } \mathit{abstracts}\} \\
& \exists B \in \mathit{abstracts}\,x\,E\,T\ :\ F' = \mathit{etaRed}\,(\lambda x.B) \\
\equiv & \{\text{definition of } F', \vdash_{\mathit{twice}}\} \\
& \exists B \in \mathit{abstracts}\,x\,E\,T\ :\ \phi \vdash_{\mathit{twice}} F \rightarrow \mathit{etaRed}\,(\lambda x.B)
\end{array}
$$

4.3.1 Correctness of *abstracts*

To show this, we prove by induction that $\mathit{abstracts}_n$ satisfies the following specification. The specification of *abstracts* then follows immediately, since *abstracts* is defined as the union of the $\mathit{abstracts}_n$ over all n. The notation $\#_x(B)$ gives the number of occurrences of x in B.

$$
B \in \mathit{abstracts}_n\,x\,E\,T \quad \equiv \quad \left(\begin{array}{l} \#_x(B) = n \\ \mathit{ms}'\,((x := E')\,B)) = T \\ B\ \eta\text{-normal}\end{array}\right)
$$

For the base case $n = 0$, then $B = T$ and this specification is trivially satisfied. For the step case, we suppose that it is valid for some n.

We start by making some definitions: denote the number of parameters that E takes by m. We will wish to make assertions about the first m arguments of occurrences of certain variables in an expression, and thus for a variable z and expression A define $\mathit{args}_m\,z\,A$ to be set containing the first m

arguments of each occurrence of z in A. We lift the "is a subexpression of" operator $\trianglelefteq$ to having sets as the right-hand argument in the obvious way, and thus $x \ntrianglelefteq \mathit{args}_m\ z\ A$ will denote the logical statement that could be written in English as "x does not occur in the first m arguments of any occurrence of z in A".

In the course of the calculation below, we shall need to make use of the fact that *abstract* abstracts one further instance of E, and whilst doing so preserves η-normalness. The proof of this claim is deferred to the next section.

We are now in a position to argue as follows:

$$\begin{array}{ll}
 & B \in \mathit{abstracts}_{(n+1)}\ x\ E\ T \\
\equiv & \quad \{\text{definition of } \mathit{abstracts}_{(n+1)}\} \\
 & \exists C, D, y \text{ fresh.} \begin{array}{ll} & C \in \mathit{abstracts}_n\ x\ E\ T \\ \wedge & D \in \mathit{abstract}\ x\ y\ E\ C \\ \wedge & B = (y := x)D \end{array} \\
\equiv & \quad \{\text{induction hypothesis}\} \\
 & \exists C, D, y \text{ fresh.} \begin{array}{ll} & \#_x(C) = n \\ \wedge & \mathit{ms}'\ ((x := E')\ C) = T \\ \wedge & C\ \eta\text{-normal} \\ \wedge & D \in \mathit{abstract}\ x\ y\ E\ C \\ \wedge & B = (y := x)D \end{array} \\
\equiv & \quad \{\text{claim, see Section 4.3.1.1}\} \\
 & \exists C, D, y \text{ fresh.} \begin{array}{ll} & \#_x(D) = n, \#_y(D) = 1 \\ \wedge & \mathit{ms}'\ ((x := E')\ C) = T \\ \wedge & \mathit{ms}'\ ((y := E')\ D) = C \\ \wedge & D\ \eta\text{-normal} \\ \wedge & x \ntrianglelefteq \mathit{args}_m\ y\ D \\ \wedge & B = (y := x)D \end{array}
\end{array}$$

$\equiv$ {value of C}

$$\begin{array}{rll} & & \#_x(D) = n, \#_y(D) = 1 \\ & \wedge & \mathit{ms}'\,((x := E')\,(\mathit{ms}'\,((y := E')\,D))) = T \\ \exists D, y \text{ fresh.} & \wedge & D\ \eta\text{-normal} \\ & \wedge & x \ntrianglelefteq \mathit{args}_m\ y\ D \\ & \wedge & B = (y := x)\,D \end{array}$$

$\equiv$ {the redexes reduced by each application of ms' are disjoint, so the applications can be merged}

$$\begin{array}{rll} & & \#_x(D) = n, \#_y(D) = 1 \\ & \wedge & \mathit{ms}'\,((x := E')((y := E')\,D)) = T \\ \exists D, y \text{ fresh.} & \wedge & D\ \eta\text{-normal} \\ & \wedge & x \ntrianglelefteq \mathit{args}_m\ y\ D \\ & \wedge & B = (y := x)\,D \end{array}$$

$\equiv$ {reordering substitutions}

$$\begin{array}{rll} & & \#_x(D) = n, \#_y(D) = 1 \\ & \wedge & \mathit{ms}'\,((x := E')((y := x)\,D)) = T \\ \exists D, y \text{ fresh.} & \wedge & D\ \eta\text{-normal} \\ & \wedge & x \ntrianglelefteq \mathit{args}_m\ y\ D \\ & \wedge & B = (y := x)\,D \end{array}$$

$\equiv$ {backwards, choose D by replacing an outermost occurrence of x in B with y}

$$\begin{array}{ll} & \#_x(B) = n + 1 \\ \wedge & \mathit{ms}'\,((x := E')\,B) = T \\ \wedge & B\ \eta\text{-normal} \end{array}$$

4.3.1.1 Claim about *abstract*

The following claim about *abstract* encodes the fact that it abstracts precisely one instance of E, and preserves η-normalness.

$$\left(\begin{array}{l} \#_x(D) = n, \#_y(D) = 1 \\ ms'\,((y := E')\,D) = C \\ x \not\trianglelefteq args_m\ y\ D \\ D\ \eta\text{-normal} \end{array}\right) \equiv \left(\begin{array}{l} \#_x(C) = n \\ D \in abstract\ x\ y\ E\ C \\ C\ \eta\text{-normal} \end{array}\right)$$

In order to understand this claim more intuitively, recall E is an expression that occurs as the argument in an application whose function part has a flexible head. Thus, E satisfies restrictions on what its body should and should not contain, as given in Section 4.1. The reader is also reminded that we use E' as a shorthand for *mark* E, that is E with all outer λs marked.

We first aim to eliminate the η-normalness conditions from our proof obligation. Recall the definition of *abstract*:

$$\begin{array}{rcl} abstract\ x\ y\ E\ T & = & \{\ etanormalise\ (replace\ loc\ R\ T) \\ & & \quad\quad\quad\quad \mid\ (S, loc) \in subexps\ T \\ & & \quad\quad\quad\quad\quad x \not\trianglelefteq S \\ & & \quad\quad\quad\quad\quad R \in instance\ y\ E\ S \quad \} \end{array}$$

Clearly, the condition that D should be η-normal is trivially satisfied. Similarly, the condition that C should be η-normal follows from the specification of *markedstep*$'$.

Now, specify the function *replace*$'$ by the following:

$$replace'\ loc\ S\ T = etanormalise\ (replace\ loc\ S\ T)$$

As with the definition of *reduce*$'$ from Section 2.5.2.1, we can give a direct definition of *replace*$'$ that strips off η-redexes as they are created, provided that S is η-normal.

$$\begin{array}{rcl}
\mathit{replace}'\,\langle\rangle\,S\,E &=& S \\
\mathit{replace}'\,(\mathit{Body}; \mathit{loc})\,S\,(\lambda x.E) &=& \mathit{etaRed}\,(\lambda x.(\mathit{replace}'\,\mathit{loc}\,S\,E)) \\
\mathit{replace}'\,(\mathit{Func}; \mathit{loc})\,S\,(E_1\,E_2) &=& (\mathit{replace}'\,\mathit{loc}\,S\,E_1)\,E_2 \\
\mathit{replace}'\,(\mathit{Arg}; \mathit{loc})\,S\,(E_1, E_2) &=& E_1\,(\mathit{replace}'\,\mathit{loc}\,S\,E_2)
\end{array}$$

Unfolding the definition of *abstract* gives us the following equivalence:

$$\begin{array}{ll}
& D \in \mathit{abstract}\,x\,y\,E\,C \\
\equiv & \quad \{\text{definition of } \mathit{abstract}\} \\
& \exists R, S, \mathit{loc}. \begin{array}{ll} & D = \mathit{etanormalise}\,(\mathit{replace}\,\mathit{loc}\,R\,C) \\ \wedge & (S, \mathit{loc}) \in \mathit{subexps}\,C \\ \wedge & \#_x(S) = 0 \\ \wedge & R \in \mathit{instance}\,y\,E\,S \end{array}
\end{array}$$

Therefore, since the definition of *instance* shows that R is η-normal, our proof obligation has been reduced to:

$$\left(\begin{array}{l} \#_x(D) = n \\ \#_y(D) = 1 \\ \mathit{ms}'\,((y := E')\,D) = C \\ x \ntrianglelefteq \mathit{args}_m\,y\,D \end{array}\right) \equiv \exists R, S, \mathit{loc}. \left(\begin{array}{l} \#_x(C) = n \\ D = \mathit{replace}'\,\mathit{loc}\,R\,C \\ (S, \mathit{loc}) \in \mathit{subexps}\,C \\ \#_x(S) = 0 \\ R \in \mathit{instance}\,y\,E\,S \end{array}\right)$$

We prove this by induction on the structure of D. The induction is over all D that contain precisely one occurrence of y; it is valid to restrict the values of D considered thus because both sides of the proof obligation imply that this is true. For the left-hand side, this implication is clearly trivial. For the right-hand side, recall that y is a fresh variable and thus does not occur in C. The definition of *instance* implies that R contains one occurrence of y, and so the definition of *replace*′ tells us that the same is true of D.

Our base case is slightly unorthodox, because of the nature of the results produced by *instance*. It considers the situation where $D = y\,A_1 \ldots A_r$, for $0 \leq r \leq m$.

There are two step cases. The first sets $D = \lambda x.D'$ and assumes that the result holds for D'. The second has $D = D_1\, D_2$ for values of D not already covered by the base case, and assumes that the result holds for whichever of D_1 or D_2 contains the occurrence of y. The proofs of these cases are relatively straightforward, and we omit them. We shall also omit the details of the proof of a lemma required for the base case, which simply follows from unfolding certain definitions.

$$
\begin{array}{lll}
 & & \#_x(D) = n, \#_y(D) = 1 \\
 & \wedge & \mathit{ms}'\,((y := E')\,D) = C \\
 & \wedge & x \not\trianglelefteq \mathit{args}_m\; y\; D \\
\equiv & & \{\text{value of } D\} \\
 & & n = 0 \\
 & \wedge & \mathit{ms}'\,(E'\,A_1 \ldots A_r) = C \\
 & \wedge & x \not\trianglelefteq A_1 \ldots A_r \\
\equiv & & \{\text{definition of } \mathit{ms}', E'\} \\
 & & n = 0 \\
 & \wedge & \lambda x_{r+1} \ldots x_m.((x_1 := A_1, \ldots, x_r := A_r)\,(\mathit{body}\; E)) = C \\
 & \wedge & (x_1, \ldots, x_m) = \mathit{params}\; E \\
 & \wedge & x \not\trianglelefteq A_1 \ldots A_r \\
\equiv & & \{C \text{ is a } \lambda\text{-abstraction with } r \text{ parameters}\} \\
 & & \quad n = 0 \\
 & & \quad \wedge \;\; C = \lambda x_{r+1} \ldots x_m.S \\
 & & \exists S. \wedge \;\; S = (x_1 := A_1, \ldots, x_r := A_r)\,(\mathit{body}\; E) \\
 & & \quad \wedge \;\; (x_1, \ldots, x_m) = \mathit{params}\; E \\
 & & \quad \wedge \;\; \#_x(S) = 0
\end{array}
$$

$\equiv$ {introduce ϕ with domain $\{ x_1, \ldots, x_r \}$, each of $x_1 \ldots x_r$ appears in *body* E because of restrictions on pattern}

$$\exists S, \phi. \begin{array}{ll} & n = 0 \\ \wedge & \forall\, 1 \leq i \leq r. \phi x_i = A_i \\ \wedge & \forall\, r+1 \leq i \leq m. \phi x_i = x_i \\ \wedge & C = \lambda x_{r+1} \ldots x_m . S \\ \wedge & \#_x(S) = 0 \\ \wedge & (x_1, \ldots, x_m) = \mathit{params}\ E \\ \wedge & \phi\,(\mathit{body}\ E) = S \\ \wedge & \forall \psi . \psi\,(\mathit{body}\ E) = S, \phi \leq \psi \end{array}$$

$\equiv$ {definition of $=$}

$$\exists S, \phi. \begin{array}{ll} & \#_x(C) = n \\ \wedge & y\, A_1 \ldots A_r\, x_{r+1} \ldots x_m = y\,(\phi\, x_1) \ldots (\phi\, x_m) \\ \wedge & C = \lambda x_{r+1} \ldots x_m . S \\ \wedge & \#_x(S) = 0 \\ \wedge & (x_1, \ldots, x_m) = \mathit{params}\ E \\ \wedge & \phi\,(\mathit{body}\ E) = S \\ \wedge & \forall \psi . \psi\,(\mathit{body}\ E) = S, \phi \leq \psi \end{array}$$

$\equiv$ {specification of *matches none* from Section 2.6}

$$\exists S, \phi. \begin{array}{ll} & \#_x(C) = n \\ \wedge & y\, A_1 \ldots A_r\, x_{r+1} \ldots x_m = y\,(\phi\, x_1) \ldots (\phi\, x_m) \\ \wedge & C = \lambda x_{r+1} \ldots x_m . S \\ \wedge & \#_x(S) = 0 \\ \wedge & (x_1, \ldots, x_n) = \mathit{params}\ E \\ \wedge & \phi \in \mathit{matches\ none}\,(\mathit{body}\ E)\ S \end{array}$$

$\equiv$ {Lemma 4.3}

$$\exists S, loc, \phi. \begin{array}{ll} & \#_x(C) = n \\ \wedge & y\,A_1\,\ldots\,A_r = \mathit{replace}'\,\mathit{loc}\,(y\,(\phi\,x_1)\,\ldots\,(\phi\,x_m))\,C \\ \wedge & (S, \mathit{loc}) \in \mathit{subexps}\,C \\ \wedge & \#_x(S) = 0 \\ \wedge & (x_1, \ldots, x_n) = \mathit{params}\,E \\ \wedge & \phi \in \mathit{matches}\,\mathit{none}\,(\mathit{body}\,E)\,S \end{array}$$

$\equiv$ {definition of D, introduce R}

$$\exists R, S, loc, \phi. \begin{array}{ll} & \#_x(C) = n \\ \wedge & D = \mathit{replace}'\,\mathit{loc}\,R\,C \\ \wedge & (S, \mathit{loc}) \in \mathit{subexps}\,C \\ \wedge & \#_x(S) = 0 \\ \wedge & R = y\,(\phi\,x_1)\,\ldots\,(\phi\,x_m) \\ \wedge & (x_1, \ldots, x_n) = \mathit{params}\,E \\ \wedge & \phi \in \mathit{matches}\,\mathit{none}\,(\mathit{body}\,E)\,S \end{array}$$

$\equiv$ {definition of *instance*}

$$\exists R, S, loc. \begin{array}{ll} & \#_x(C) = n \\ \wedge & D = \mathit{replace}'\,\mathit{loc}\,R\,C \\ \wedge & (S, \mathit{loc}) \in \mathit{subexps}\,C \\ \wedge & \#_x(S) = 0 \\ \wedge & R \in \mathit{instance}\,y\,E\,S \end{array}$$

Lemma 4.3.

$$\begin{array}{ll} & y\,A_1 \ldots A_r\,x_{r+1} \ldots x_m = y\,(\phi\,x_1) \ldots (\phi\,x_m) \\ \wedge & C = \lambda x_{r+1} \ldots x_m . S \end{array}$$

$$\equiv$$

$$\begin{array}{ll} & y\,A_1 \ldots A_r = \mathit{replace}'\,\mathit{loc}\,(y\,(\phi\,x_1)\,\ldots\,(\phi\,x_m))\,C \\ \wedge & (S, \mathit{loc}) \in \mathit{subexps}\,C \end{array}$$

Proof. By unfolding the specification of *replace′* and the definitions of *replace*, $\simeq$, *subexps*, and taking advantage of the η-normalness of $(y\,A_1 \ldots A_r)$ and C. □

4.4 Related work : Third-order matching

In this section we discuss an algorithm by Comon and Jurski [22], which generates a *representation* of the set of third-order matches for the simply typed λ-calculus. The first step is to transform a matching problem into multiple *interpolation equations*. Next, each interpolation equation is converted into a tree automaton which *accepts* λ-terms if and only if they are solutions of the corresponding equation. Finally, the results are combined to give a single automaton which represents the set of solutions for the problem.

4.4.1 Tree automata

The standard definition of tree automata [35, 89] is modified slightly for the purposes of this algorithm to allow all closed terms of a particular type to be represented by one state. In this context, a tree automaton consists of a set F of *labels* (which are used to construct λ-terms), including the special variables $\Box_{\tau_1}, \ldots, \Box_{\tau_n}$ which are intended to represent any term of type $\tau_1, \ldots, \tau_n$ respectively, a set Q of *states* including a subset Q_f of *final* states, and a set Δ of transition rules of the form $f(q_1, \ldots, q_m) \to q$ for $f \in F$ and $q, q_1, \ldots, q_m \in Q$.

The *forgetful relation* $\sqsubseteq$ is defined as the least reflexive relation on terms such that:
$\Box_\tau \sqsubseteq u$ for each term u of type τ.
If $u_1 \sqsubseteq v_1, \ldots, u_n \sqsubseteq v_n$ then $f(u_1, \ldots, u_n) \sqsubseteq f(v_1, \ldots, v_n)$ for each f of appropriate type.

Each transition rule Δ has a label with some (possibly none) states as parameters on the left, and a new state on the right. Given a λ-term expressed as a tree, a rule can be applied to a subterm to produce a new tree in which one of the leaves is a state. The idea is that if this can be done repeatedly to some term to produce a final state, then that term, or any term that can be made from it by replacing $\Box$s with terms of the appropriate type, is *accepted* by the automaton.

Formally, the set of transition rules induces a relation $\underset{\Delta}{\rightarrow}$ with reflexive transitive closure $\underset{\Delta}{\rightarrow}^*$, and a term t is accepted by an automaton if there exists a term u (which may contain some $\Box$ symbols) such that $u \sqsubseteq t$ and $u \underset{\Delta}{\rightarrow}^* q_f$ for some final state q_f.

For example, suppose we have an automaton with labels $\{ x, \lambda x, a, \Box_o \}$, states $\{ q_\Box, q_a, q_f \}$ where q_f is a final state, and the following transition rules:

$$\begin{array}{rcl} a & \rightarrow & q_a \\ \Box_o & \rightarrow & q_\Box \\ x(q_a, q_\Box) & \rightarrow & q_a \\ \lambda x.q_a & \rightarrow & q_f \end{array}$$

Then we have the following transition sequence:

$$\lambda x.x(a, \Box_o) \underset{\Delta}{\rightarrow} \lambda x.x(q_a, \Box_o) \underset{\Delta}{\rightarrow} \lambda x.x(q_a, q_\Box) \underset{\Delta}{\rightarrow} \lambda x.q_a \underset{\Delta}{\rightarrow} q_f$$

Thus $\lambda x.x(a, b)$ is accepted by the automaton for any b of type o.

Tree automata are closed under union and intersection; that is, given two automata we can construct a third automaton which accepts exactly those terms in the union or intersection of the sets of terms accepted by the original automata.

4.4.1.1 Generating the interpolation equations

An *interpolation equation* is an equation of the form $x(s_1, \ldots, s_n) = t$ where x is a free variable and $s_1, \ldots, s_n, t$ are closed and normal.

Comon and Jurski proposed the following procedure to generate a particular set of interpolation equations from the matching problem $s = t$.

Start with an empty set of interpolation equations. Let $x(s_1, \ldots, s_n)$ of type τ be an occurrence of a free variable together with its arguments in s, where $s_1, \ldots, s_n$ do not contain any free variables. Let r be either $\Box_\tau$ or a subterm of t with some (or all or none) of its own subterms replaced by variables from $s_1, \ldots, s_n$ which are bound higher up in s. In the special case that x is the last free variable in s, the only option allowed for r is t itself.

Finally, replace $x(s_1, \dots, s_n)$ with r in s, and if r is not $\Box_\tau$ then add the equation $x(s_1, \dots, s_n) = r$ to the set of interpolation equations, and repeat this procedure until we are left with the equation $t = t$ (which is guaranteed by the restriction on r for the last free variable in s).

This procedure gives a set of interpolation equations whose conjunction can be solved to give a set of solutions to the original matching problem. If we generate all such possible sets of equations, the union of the sets of solution will give a complete set of third order solutions.

4.4.2 Solving interpolation equations

The interpolation equation $x(s_1, \dots, s_n) = t$, where x is of type $\tau_1, \dots, \tau_n \to o$ is solved by any term accepted by the following automaton:

The labels are the constant symbols occuring in t, the fresh variables $x_1, \dots, x_n$ of types $\tau_1, \dots, \tau_n$, and the special symbols $\Box_\tau$ for each base type τ.

The states Q are q_u for all subterms u of t, $q_{\Box_\tau}$ for all base types τ, and the final state q_f.

The transition rules are:

- $g(q_{t_1}, \dots, q_{t_n}) \to q_{g(t_1,\dots,t_n)}$ for each $q_{g(t_1,\dots,t_n)}$ in Q – note that this includes the possibility of $n = 0$ and thus rules like $g \to q_g$ and $\Box_\tau \to q_{\Box_\tau}$

- $x_i(q_{t_1}, \dots, q_{t_n}) \to q_u$ where u is a subterm of t and $s_i(t_1, \dots, t_n)$ has u as its normal form. For each j, if t_j is of type τ and the expression $s_i(t_1, \dots, t_{j-1}, \Box_\tau, t_{j+1}, \dots, t_n)$ also has u as its normal form then t_j should be set to $\Box_\tau$ to ensure the rule covers the most general term possible

- $\lambda x_1, \dots, x_n.q_t \to q_f$

4.4.3 Combining results

It remains to combine the resulting automata by union or intersection in the manner determined when the interpolation equations were generated. Generating the union of two automata is easy; we simply take the union of the sets of labels, states, final states and transition rules respectively.

Dealing with intersection is a little bit more tricky. The set of labels is again obtained by set union; however, the set of states and final states is obtained by taking cartesian products.

We obtain the set of transition rules by combining compatible rules, (*i.e.* those with the same outermost symbol on the left-hand side), from each of the original sets. So for example $x(q_{11}, q_{12}) \rightarrow q_{13}$ and $x(q_{21}, q_{22}) \rightarrow q_{23}$ combine to give $x((q_{11}, q_{21}), (q_{12}, q_{22})) \rightarrow (q_{13}, q_{23})$, and $\lambda x.q_{11} \rightarrow q_{12}$ combines with $\lambda x.q_{21} \rightarrow q_{22}$ to give $\lambda x.(q_{11}, q_{21}) \rightarrow (q_{12}, q_{22})$. Note that s and $\Box_\tau$ are compatible if s is of type τ, so the rules $s \rightarrow q_1$ and $\Box_\tau \rightarrow q_2$ combine to give $s \rightarrow (q_1, q_2)$.

We also add the special rules $(q_{s_1}, q_{s_2}) \rightarrow (q_{\Box_{\tau_1}}, q_{s_2})$ and $(q_{s_1}, q_{s_2}) \rightarrow (q_{s_1}, q_{\Box_{\tau_2}})$ where s_1 and s_2 are of type τ_1 and τ_2 respectively (these are not strictly transition rules according to the definition given earlier, but their use is obvious).

4.4.4 Example

Take the problem $x(\lambda z_1.x(\lambda z_2.z_1)) = c(a)$. For simplicity, we assume that there is only one base type, o. By the procedure in Section 4.4.1.1, the problem reduces to these five sets of interpolation equations:

$$\begin{aligned}
x(\lambda z_1.\Box) &= c(a) \\
x(\lambda z_1.a) &= c(a) \quad \wedge \quad x(\lambda z_2.z_1) = a \\
x(\lambda z_1.c(a)) &= c(a) \quad \wedge \quad x(\lambda x_2.z_1) = c(a) \\
x(\lambda z_1.z_1) &= c(a) \quad \wedge \quad x(\lambda z_2.z_1) = z_1 \\
x(\lambda z_1.c(z_1)) &= c(a) \quad \wedge \quad x(\lambda z_2.z_1) = c(z_1)
\end{aligned}$$

Note that z_1 cannot appear in the solutions of the second equations of each set; this is guaranteed by the omission of transition rules with bound variables on the left-hand side.

We derive the solutions for the fourth set of equations – the procedure for the others is identical and combining the sets of solutions is a trivial matter of taking the union of the automata.

The equation $x(\lambda z_1.z_1) = c(a)$ is solved by the automaton with labels $\{x_1, c, a, \Box_o\}$, states $\{q_{c(a)}, q_a, q_{\Box_o}, q_f\}$, and the following transition rules:

$$\begin{aligned}
c(q_a) &\rightarrow q_{c(a)} \\
a &\rightarrow q_a \\
\Box_o &\rightarrow q_{\Box_o} \\
x_1(q_a) &\rightarrow q_a \\
x_1(q_{c(a)}) &\rightarrow q_{c(a)} \\
\lambda x_1.q_{c(a)} &\rightarrow q_f
\end{aligned}$$

The second equation $x(\lambda z_2.z_1) = z_1$ is solved by the automaton with labels $\{x_1, z_1, \Box_o\}$, states $\{q_{z_1}, q_{\Box_o}, q_f\}$, and the following transition rules:

$$\begin{aligned}
\Box_o &\rightarrow q_{\Box_o} \\
x_1(q_{\Box_o}) &\rightarrow q_{z_1} \\
\lambda x_1.q_{z_1} &\rightarrow q_f
\end{aligned}$$

The next step is to calculate the intersection automaton. The set of labels is $\{x_1, c, a, z_1, \Box_o\}$, and the set of states is $\{(q_{c(a)}, q_{z_1}), (q_{c(a)}, q_{\Box_o}), \ldots\}$. The set of transition rules is:

$$\begin{aligned}
\Box_o &\rightarrow (q_{\Box_o}, q_{\Box_o}) \\
a &\rightarrow (q_a, q_{\Box_o}) \\
c((q_a, q_{\Box_o})) &\rightarrow (q_{c(a)}, q_{\Box_o}) \\
x_1((q_a, q_{\Box_o})) &\rightarrow (q_a, q_{z_1}) \\
x_1((q_{c(a)}, q_{\Box_o})) &\rightarrow (q_{c(a)}, q_{z_1}) \\
\lambda x_1.(q_{c(a)}, q_{z_1}) &\rightarrow (q_f, q_f)
\end{aligned}$$

We also add these special transition rules (others could be added but they are useless in practice since the states on the left-hand side are never reached).

$$\begin{aligned}
(q_a, q_{z_1}) &\rightarrow (q_a, q_{\Box_o}) \\
(q_{c(a)}, q_{z_1}) &\rightarrow (q_{c(a)}, q_{\Box_o})
\end{aligned}$$

The only final state of this automaton is (q_f, q_f). By working backwards from this state, we can deduce that the solutions are precisely $\lambda x_1.x_1^n(c(x_1^m(a)))$ where $n > 0$. In general, this can be done by constructing a tree starting from the final state and conducting a breadth-first search for initial states.

4.4.5 Discussion

The potentially infinite set of results returned by Comon and Jurski's algorithm (or indeed, any algorithm capable of third-order matching) makes it at best inconvenient to use in a practical program transformation system. One solution would be to somehow select *representatives* from infinite sets of matches, but it would require a heuristic to determine which were likely to be useful, making it hard to specify the set of results returned.

An alternative would be to use Comon and Jurski's algorithm in conjunction with the restrictions on the pattern we developed for the two-step matching algorithm (given in Section 4.1). For the same reasons as for the two-step algorithm, this would guarantee that only a finite set of results would be returned.

However, the two-step algorithm has certain advantages which we believe make it more useful for program transformation. Firstly, and most importantly, its type-free nature makes it more widely applicable than Comon and Jurski's algorithm, which is restricted to simply-typed lambda terms.

Secondly, as we have shown in Section 4.1, the two-step algorithm is strictly more powerful than third-order matching; for simply-typed terms, it will return all third-order results and in some cases some results of higher order. However, this is not such an advantage; we have not yet found any program transformation problems that require this extra power. In fact, Comon and Jurski also give in the same paper a (rather more complicated) algorithm which carries out fourth-order matching, again generating a representation of the result set, so if fourth-order results really were useful it would make sense to further investigate using their algorithm.

Chapter 5

Practical implementation of matching

The algorithms described in this thesis have all been implemented as part of the program transformation system MAG [26]. In this chapter, we describe MAG and present the essentials of these implementations.

MAG is a term rewriting system that implements the procedure that we sketched in Section 1.2.1. Given an expression and a set of rewrite rules, it applies the rewrite rules to the expression repeatedly until no more apply. These rules can be conditional rules, in which case sub-calculations are carried out in order to satisfy the side conditions.

It should be noted that MAG is a simple prototype intended to be a proof-of-concept for the principle of active source, not an industrial strength transformation system. As such, it lacks many features that would be required for use in large scale software development. In particular, the language MAG transforms is a relatively small subset of full Haskell. It uses Hindley-Milner types [66] which do not support type classes, amongst other things. In addition, much of the syntactic sugar of Haskell is missing.

Similarly, we have not implemented the **transform** . . . **where** . . . **with** syntactic sugar suggested at the end of Section 1.2, and thus rewrite rules must be specified in a separate file rather than as annotations to the program being transformed. Every rewrite rule must be given in full even if the details could be deduced from the program – for example the promotion rule could be automatically generated from the relevant datatype, and rules that simply unfold definitions could come from duplicating the relevant program text.

No attempt is made at performing strictness analysis, and thus it is not possible to specify strictness side conditions to rewrite rules. As a result, the application of promotion by MAG is technically unsound, and it is incumbent on the user to verify that promotion has not introduced a potential for non-termination, or alternatively to guarantee that the transformed program will not be used on infinite data structures.

The confluence or otherwise of the set of rewrite rules is ignored. As a consequence, the results of a transformation may be sensitive to the order in which they are specified (and indeed there may not even be an order which produces the desired transformation). We took this decision because proving confluence in a higher-order conditional term rewriting system is rather difficult [3]. Indeed, even in a first-order unconditional system generating a confluent set of rules is prone to failure [46].

However, MAG does provide features to help programmers find an appropriate set of rewrite rules to apply a transformation. Firstly, the individual rewrite steps carried out during a calculation (including those from the side calculations required to apply conditional rules) are output, so that it is easy to see exactly which rules were applied or not. Secondly, there is an optional second form of output which shows the substitutions that were generated during matching for each rule that applied, and the side calculations from *failed* attempts at applying conditional rules. This last detail in particular is invaluable for determining exactly why a rule was not applied in a complex derivation.

MAG applies rules in the order specified in the theory file, to outermost subexpressions first. A consequence of this is that we can guarantee that a particular rule will not be applied to an expression unless all earlier rules have failed to apply to that expression, which provides the user with some rudimentary but easy to understand control over the rewriting process.

5.1 Example session with MAG

Input to MAG comes in three parts. Firstly, a *program* file, which is currently only used for typechecking and thus we omit details. Secondly, a *theory* file that contains a set of rewrite rules. Finally, the expression to be rewritten. For the fast reverse example, we use the following theory file:

```
{- reverse.eq -}

fastreverse: fastreverse xs ys = reverse xs ++ ys;

reverse0: reverse []     = [];
reverse1: reverse (x:xs) = reverse xs ++ [x];

cat0: []     ++ xs = xs;
cat1: (x:xs) ++ ys = x:(xs++ys);

catassoc: (xs ++ ys) ++ zs = xs ++ (ys ++ zs);

promotion: f (foldr plusl e xs) = foldr crossl e' xs,
            if {f e = e';
                \ x y -> f (plusl x y)
                   = \ x y -> crossl x (f y)}
```

As we described in Section 1.2.1, the specification of *fastreverse* and the definitions of *reverse* and ++ are included as rewrite rules, together with a law stating the associativity of ++ and the promotion rule. Each rule is implictly universally quantified over all the free variables appearing in it and is labelled with a name that is displayed in the derivation when the rule is applied.

Recall from Section 1.2 that the fast reverse derivation is acheived by applying promotion to *fastreverse* (*foldr* (:) [ ] *xs*) *ys*. In MAG, terms to be rewritten should not include free variables. Therefore, we explicitly bind the variables *xs* and *ys*, and instead ask MAG to rewrite the expression λ*xs ys*.*fastreverse* (*foldr* (:) [ ] *xs*) *ys*. Each intermediate step in the derivation is displayed by MAG, together with the initial and final expressions.

The following is the output from MAG when given the above theory file as input. Because of the potential for name conflicts, MAG gives fresh names to all the bound variables in expressions during processing of each rewrite step, so the names that are displayed bear no relation to the names that were input. As a minor concession to readability, a final renaming is applied before pretty-printing so that the set of names used there starts from the letter 'a', rather than whatever part of the internal name supply had been reached during internal processing. Expressions are also η-contracted before being displayed, hence our initial input is displayed as λa.*fastreverse* (*foldr* (:) [ ] *a*):

```
   (\ a -> fastreverse (foldr (:) [] a))
= { fastreverse }
   (\ a -> (++) (reverse (foldr (:) [] a)))
= { promotion


      (++) (reverse [])
   = { reverse0 }
      (++) []
   = { cat0 }
      (\ a -> a)


      (\ a b -> (++) (reverse (a : b )))
   = { reverse1 }
      (\ a b -> (++) (reverse b ++ (a : [])))
   = { catassoc }
      (\ a b c -> reverse b ++ ((a : []) ++ c))
```

```
  = { cat1 }
     (\ a b c -> reverse b ++ (a : ([] ++ c)))
  = { cat0 }
     (\ a b c -> reverse b ++ (a : c ))
 }
  foldr (\ c d e -> d (c : e )) (\ f -> f)
```

Notice that the result is also η-contracted. If we want to see the more intuitive directly recursive form, we will have to unfold the use of *foldr* manually, although it would not be particularly difficult to automate this procedure. Note however that it is better to leave it as a *foldr* from the point of view of enabling certain automatic optimisations by the compiler, as detailed in Section 1.3.2.

5.2 Implementation

As with the rest of MAG, the language used to implement our matching algorithms is Haskell. Haskell is a lazy functional language, which means that if users of our algorithms are only interested in the first result (for example), then unnecessary work is not done in computing the rest; this effect can be achieved in non-lazy languages but it is necessary to explicitly encode this "demand-driven" approach. Here we will give just a brief outline of the more complex or uncommon features that we shall use; for a complete description of the language and its syntax we refer the reader to [12, 76].

We shall start by presenting a program that closely follows the descriptions of our algorithms in earlier chapters, and then show what optimisations are necessary to achieve respectable performance. Unfortunately, since transforming full Haskell is beyond the scope of MAG, most of them must be applied by hand. We will however use MAG for one optimisation, giving the details in Section 6.5.

5.2.1 Preliminaries

We shall attempt to make our implementation mirror the more abstract description given in earlier chapters as closely as possible. For this, it will be useful to have a syntax for representing the bag and set comprehensions we used to describe the results of various functions. Conveniently, Haskell has support for *list comprehensions* [96] which provide exactly what we need; $[\![\, 2 * x \mid x \in X \,]\!]$ can be implemented as $[2 * x \mid x \leftarrow X]$. If we want to represent a set, we can use the standard Haskell function *nub* to remove all duplicates from a list after it is generated.

We shall take some liberties with Haskell's syntax for the purposes of readability. In particular, we shall sometimes use Greek identifiers for variable names, and we shall define some infix operators that are not legal Haskell names.

The following datatype is used to represent expressions:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Exp} & = & \mathit{LVar}\ \mathit{VarId}\ \mathit{Type} \mid \mathit{PVar}\ \mathit{VarId}\ \mathit{Type} \\ & \mid & \mathit{Con}\ \mathit{Constant}\ \mathit{Type} \\ & \mid & \mathit{Ap}\ \mathit{Exp}\ \mathit{Exp}\ \mathit{Type} \mid \mathit{Lam}\ \mathit{VarId}\ \mathit{Exp}\ \mathit{Type} \end{array}$$

This datatype defines an expression (*Exp*) as either a local variable, a pattern variable, a constant, an application or a lambda abstraction.

One important detail of this implementation is that, as with any practical program transformation system, our expressions have types, and these types might affect whether two expressions could be considered equal or not, or even whether an expression that we might construct was legal. In addition, the context in which pattern variables are used on the right-hand side of a rewrite rule might further restrict the allowable types for expressions which the matching procedure tried to assign to those pattern variables.

Thus, we need to ensure that any matches we generate are valid for the type system being used. However, in keeping with our earlier claim that our matching algorithms are independent of any particular type system, we shall not go into the details of the types we use, and shall instead view them as an abstract datatype with a few basic operations. If an implementation which worked on untyped expressions was required, it would be a simple matter to remove all the parts which deal with types.

Pattern and local variables are explicitly distinguished from each other by the use of different constructors, and individual variables of each type are denoted by an identifier of type *VarId*. We will take care to ensure that these identifiers are unique to avoid problems of variable capture; for this purpose we shall maintain a supply of fresh names from which new identifiers can be drawn when needed.

Since Haskell is a “pure” language without updatable variables, it is necessary to pass this supply from function to function, which is rather inconvenient. Therefore, we wrap this supply up in a *state monad* [101] which “stores” the list of names and allows functions to access it when needed without needing to explicitly pass it around. A value of type α that requires a name supply to be computed is instead given the type $\mathit{Name}\,\alpha$, and functions are provided to convert between values of type α and values of type $\mathit{Name}\,\alpha$. The notational penalty we pay for this is that programs have to be written in a somewhat imperative style.

Alternative approaches to the problem of variable naming would be to follow the approach of [2] and use an impure function to generate the fresh names, or to go a stage further and use a language such as FreshML [79] or some similar work by Miller [64] with built-in support for representing terms modulo α-conversion.

Unfortunately, the use of a state monad conflicts slightly with the use of list comprehensions as described above. To achieve an elegant notation, we make use of *monad transformers* [60] , which allow certain monads to be “stacked”. Since lists are also a monad, we can create a combined monad (which we call *NameL*) that encapsulates both the name supply and the notion of maintaining a “list of results”. Instead of using list comprehensions, we use *monad comprehensions* [100]; as a result, the program for $[\![\, 2 * x \mid x \in X \,]\!]$ is written:

$$\begin{aligned}&\textbf{do}\ \ x \leftarrow X\\ &\quad\ \ \ \mathit{return}\ 2 * x\end{aligned}$$

If we require a fresh variable name we make use of the function *freshVar*, which generates a single fresh identifier:

$$\mathit{freshVar} :: \mathit{NameL}\ \mathit{VarId}$$

For example, the following snippet produces a fresh local variable whose type is t:

$$\begin{aligned}&\textbf{do}\ \ x \leftarrow \mathit{freshVar}\\ &\quad\ \ \ \mathit{return}\ (\mathit{LVar}\ x\ t)\end{aligned}$$

Sometimes, we will have a normal list of type $[\alpha]$ that we wish to convert to a $\mathit{NameL}\,\alpha$. For this we use the function *liftSt*:

$$\mathit{liftSt} :: [\alpha] \rightarrow \mathit{NameL}\,\alpha$$

We shall not expose the underlying implementation of *Subst*, the datatype of substitutions. We manipulate them by the following operations:

$$\begin{aligned}&\mathit{idSubst} &&:: \mathit{Subst}\\ &(:=) &&:: \mathit{VarId} \rightarrow \mathit{Exp} \rightarrow \mathit{Subst}\\ &(\circ) &&:: \mathit{Subst} \rightarrow \mathit{Subst} \rightarrow \mathit{Subst}\\ &\mathit{apply} &&:: \mathit{Substitutable}\,\alpha \Rightarrow \mathit{Subst} \rightarrow \alpha \rightarrow \alpha\end{aligned}$$

The identity substitution is denoted *idSubst*. The expression $(p := e)$ returns the substitution that maps the pattern variable identified by p to the expression e and leaves all other variables unchanged. The $(\circ)$ function gives the composition of two substitutions.

Finally, for *apply*, the function which applies a substitution, we make use of a *type class.* Type classes provide Haskell with ad-hoc polymorphism; the reason we use them is that we would like to be able to apply substitutions to more than one type using the same function. To do this we make each of the types an instance of the class *Substitutable*, providing an appropriate definition of *apply* for that type at the same time. In particular, we shall want to apply substitutions to expressions, rules and lists of rules.

A rule is simply a pair of expressions; no attempt is made in the datatype definition to enforce the necessary restrictions on the expressions:

$$\textbf{newtype}\, \mathit{Rule} \;=\; \mathit{Rule}\,(\mathit{Exp}, \mathit{Exp})$$

We use the keyword **newtype** rather than the more usual **data** because GHC is able to optimise away the constructor in a single constructor datatype if we use this keyword. We cannot use a standard type synonym because we wish to make *Rule* an instance of the *Substitutable* class.

Two pieces of standard Haskell syntax that we shall make heavy use of are the "_" placeholder in the list of formal parameters of a definition to indicate that the definition does not require the value of that argument (and thus there is no point in giving it a name), and the $ operator which is just like the normal space operator for function application except that it has a low precedence and associates to the right, which often removes the needs for many levels of nested brackets in an expression. Using $ allows us to write (*addone* (*sum* (*map square xs*))) as (*addone* $ *sum* $ *map square xs*), for example.

5.2.2 Framework

In this section, we shall implement the framework described in Chapter 2. Our earlier description of the algorithms parametrised the *matches* and *resolve* functions by the *app* function. For a practical implementation, it makes more sense to parametrise by the specific *appresolve* function for that algorithm, since *app* is part of the specification rather than the implementation. To make this easier, we define the following type synonym:

$$\textbf{type}\, \mathit{AppResolver} \;=\; \mathit{Exp} \;\rightarrow\; \mathit{Exp} \;\rightarrow\; \mathit{Exp} \;\rightarrow\; \mathit{NameL}\,[\mathit{Rule}]$$

In other words, an *AppResolver* takes three expressions (the function and argument parts of the pattern and the term) and returns a selection of rule sets.

Recall that *NameL* encapsulates a list of results and that [*Rule*] represents a rule set.

The *matches* function takes an *AppResolver* and a rule set, and returns a selection of substitutions:

```
matches :: AppResolver → [Rule] → NameL Subst
```

Its implementation closely reflects the definition given in Section 2.4:

```
matches _               []         = return idSubst
matches appresolver (x : xs)  =
                  do  (σ, ys) ← expresolve appresolver x
                      φ ← matches appresolver (apply σ (xs ++ ys))
                      return (φ ∘ σ)
```

One slight modification is that, as we remarked earlier, our expressions do have types, and we need to verify that these match as well. For this reason, we define a function *expresolve* which matches the types and then calls the “real” *resolve* function. We mentioned earlier that expressions being rewritten are not allowed to contain free variables; the reason for this is that these would then appear in the term during matching which would violate the specification of the matching algorithms. It turns out that *expresolve* is also a convenient place to check this restriction.

As with the *resolve* function that will follow, *expresolve* takes in an *AppResolver* and a rule, and returns a selection of (substitution, rule set) pairs:

```
expresolve :: AppResolver → Rule → NameL (Subst, [Rule])
expresolve appresolver (Rule (e_1, e_2)) =
  if not (null (freevars e_2))
   then error ("free variables in term : " ++ showExp e_2)
   else  do  τ ← liftSt $ singletype (exptype e_1) (exptype e_2)
             (φ, rules) ← resolve appresolver (Rule (e_1, e_2))
             return (φ ∘ τ, apply τ rules)
```

The *exptype* function simply returns the type of an expression. We then use *singletype* to generate a substitution τ which makes the type of e_1 equal to the type of e_2; if this is not possible then *singletype* will return no substitutions and thus matching will fail at this point. Otherwise, *resolve* is called to do the main work of breaking down *rule* into a new set of rules together with a substitution ϕ. Finally, we apply τ to the set of rules returned and compose ϕ and τ to give the final result of *expresolve*.

As with *matches*, the implementation of *resolve* closely mirrors the description in Section 2.4.2. Its parameters are an *AppResolver*, which will be called if the pattern is a function application, and a rule:

$$resolve :: AppResolver \rightarrow Rule \rightarrow NameL\,(Subst,\ [Rule])$$

Two local variables match giving the identity substitution and no more rules, if and only if they are the same local variable:

```
resolve _ (Rule (LVar x _ , LVar y _ )) =
  liftSt $ if x == y then [(idSubst, [ ])]
                     else [ ]
```

Similarly, two constants must be the same to match:

```
resolve _ (Rule (Con a _ , Con b _ )) =
  liftSt $ if c == d then [(idSubst, [ ])]
                     else [ ]
```

A pattern variable will match with any term, as long as the term does not contain any unbound local variables:

```
resolve _ (Rule (PVar p _ , t))
  liftSt $ if null (unboundlocals e) then [(p := t, [ ])]
                                     else [ ]
```

Two λ-abstractions match if their bodies match, but we must rename one so that they are both abstractions over the same local variable:

```
resolve _ (Rule (Lam x e _, Lam y f _)) =
  liftSt [(idSubst, [Rule (e, renamebound y x f)])]
```

If a λ-abstraction is matched against a non-λ abstraction we η-expand the term and apply the rule above, as described in Section 2.4.2. The local variable used to do the η-expansion will be given the same type as the bound variable in the pattern; the function *apl* is used to construct a function application with an appropriate type:

```
resolve _ (Rule (Lam x e s, f)) =
  liftSt [(idSubst, [Rule (e, apl f (LVar x (argtype s)))])]
```

The case when the pattern is an application, where our algorithms differ from each other, is passed to the *appresolver* function. The substitution returned in this case is always the identity substitution, so we only require a selection of rule sets from *appresolver*:

```
resolve appresolver (Rule (Ap f a _, e)) =
  do rules ← appresolver f a e
     return (idSubst, rules)
```

Finally, if none of the above definitions applied then there are no matches:

```
resolve _ _ = liftSt []
```

5.2.3 Simple matching

The definition for *appresolver* for simple matching (Section 2.6) is straightforward. Either the term is also an application, in which case we return a single rule set which matches the function and argument parts respectively, or it does not and we return no rule sets:

```
simpleappresolve :: AppResolver
simpleappresolve f a (Ap f′ a′ _) = return [Rule (f, f′), Rule (a, a′)]
simpleappresolve _ _ _            = liftSt []
```

5.2.4 One-step matching

The definition of *onestepappresolve* follows the description in Section 3.2 closely. The operator $+\!\!+_m$ joins together two *NameL*s, just like $+\!\!+$ joins two lists (for those familiar with monads in Haskell, we have made *NameL* an instance of *MonadPlus*, and $+\!\!+_m$ is just the '*mplus*' operator). The function *lambda* is used to simplify the construction of λ-abstractions; the expression *lambda* (x, t) e constructs the λ-abstraction whose body is e and whose bound variable x has type t. The first part of the definition makes use of *simpleappresolve* to handle the case where the term is already a function application:

```
onestepappresolve :: AppResolver
onestepappresolve f a e =
  simpleappresolve f a e
  ++_m
  do (f', a') ← apps e
     return [Rule (f, f'), Rule (a, a')]
  ++_m
  do x ← freshVar
     return [Rule (f, lambda (x, exptype a) e)]
```

The function *apps* takes in an expression and returns a selection of pairs of expressions which give the function and argument parts of a constructed β-redex:

```
apps :: Exp → NameL (Exp, Exp)
```

Again, the definition mirrors Section 3.2.1. The function *subsplus* returns all non-empty subsets of its argument, thus eliminating the need for a separate check for emptiness of *loc*′:

```
apps term =
  do x ← freshVar
     (subexp, locs) ← liftSt $ collect $ filter okse $ subexps term
     locs′ ← liftSt $ subsplus locs
     let t = exptype subexp
     let body = replaces term locs′ (LVar x t)
     let func = lambda (x, t) body
     if normal func then return (func, subexp)
                    else liftSt []
  where okse (subexp, _) = unboundlocals subexp ⊆ oklocals
        oklocals = unboundlocals term
```

5.2.5 Two-step matching

Before invoking the two-step matching algorithm, it is necessary to check that the restrictions from Section 4.1 are satisfied. The function *twostepvalid* takes a pattern and returns a *Bool* indicating whether it satisfies them:

```
twostepvalid :: Exp → Bool
```

A λ-abstraction is a valid pattern if its body is:

```
twostepvalid (Lam x b _) = twostepvalid b
```

If a function application has a flexible function part, then its argument must satisfy certain conditions, which are checked in *validarg*. In any case, both function and argument parts must themselves both be valid patterns:

```
twostepvalid (Ap f a _) =    (not (flex f) ∨ validarg a)
                           ∧ twostepvalid f
                           ∧ twostepvalid a
```

Finally, anything else must be either a variable or a constant and will certainly be a valid pattern:

```
twostepvalid _ = True
```

The *validarg* function is defined as follows. The subsidiary function *getpb* splits up an expression $\lambda x_1 \ldots x_n.B$ into the parts $\{ x_1 \ldots x_n \}$ and B:

```
validarg exp =    null (freevars body)
                ∧ params ⊆ unboundlocals body
                ∧ (not (   null (constants body)
                         ∧ unboundlocals body ⊆ params))
  where (params, body) = getpb exp [ ]
        getpb (Lam x b _) xs = getpb b (x : xs)
        getpb e xs = (xs, e)
```

In other words, the body B can contain no free variables, but must contain at least one occurrence of each of the arguments $\{ x_1 \ldots x_n \}$. In addition, it must either contain at least one constant or at least one unbound local variable which is not one of $\{ x_1 \ldots x_n \}$.

Before commencing two-step matching, MAG checks the pattern using *twostepvalid* and if it is not valid falls back to the one-step algorithm.

Once more, we follow the description from Section 4.2. If the pattern is not flexible, we can again make use of *simpleappresolve*:

```
twostepappresolve :: AppResolver
twostepappresolve f e t =
  if flex f
   then do x ← freshVar
           b ← liftSt $ abstracts x e t
           return [Rule (f, etaRed (lambda (x, exptype e) b))]
   else  simpleappresolve f e t
```

The expression *abstracts x e t* returns all the possible ways in which instances of the function e can be abstracted from the term t using a local variable identified by x:

```
abstracts :: VarId → Exp → Exp → [Exp]
```

The obvious definition for *abstracts* would be the following:

$$\mathit{abstracts}\ x\ e\ t\ =\ \mathit{concat}\,[\mathit{abstractssub}\ n\ x\ e\ t \mid n\ \leftarrow\ [0\ldots]\,]$$

However, this will not terminate, since even though there will be some point after which the *abstractssub* functions all return empty lists (as we argued in Section 4.2.1), the computer cannot know this. Therefore, we modify this definition to the following, making use of the *takeWhile* function to stop enumerating *abstractssub* at the first empty list returned. Clearly, if there is no way to abstract n instances of e from t then there can be no way to abstract $n+1$ instances either.

$$\begin{aligned}\mathit{abstracts}\ x\ e\ t\ =\ & \mathit{concat}\ \$\ \mathit{takeWhile}\ (\mathit{not.null})\ \$\\ & \qquad [\mathit{abstractssub}\ n\ x\ e\ t \mid n\ \leftarrow\ [0\ldots]\,]\end{aligned}$$

The function *abstractssub* takes the same parameters as *abstracts*, as well as an integer to indicate how many instances should be abstracted:

$$\mathit{abstractssub}\ ::\ \mathit{Int}\ \rightarrow\ \mathit{VarId}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ [\mathit{Exp}]$$

Either we wish to extract 0 instances of e, in which case the only possible result is t, or we wish to extract $n+1$ instances, in which case we first abstract n instances, then use *abstract* to abstract an extra one, and finally use *nub* to remove any duplicates. Note that it was not necessary to use *nub* in the definition of *abstracts*, since there can be no overlap between results from calls to *abstractssub* for different values of n:

$$\begin{aligned}&\mathit{abstractssub}\ 0\ \ x\ e\ t && = [t]\\ &\mathit{abstractssub}\ (n+1)\ x\ e\ t && = \mathit{nub}\,[c \mid b\ \leftarrow \mathit{abstractssub}\ n\ x\ e\ t,\\ &&& \qquad\qquad c\ \leftarrow \mathit{abstract}\ x\ e\ b]\end{aligned}$$

Abstracting a single instance is done by *abstract*:

$$\mathit{abstract}\ ::\ \mathit{VarId}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ \mathit{Exp}\ \rightarrow\ [\mathit{Exp}]$$

The definition is straightforward; we find all subexpressions of t, and for each one check that it does not already contain x before replacing it with the result of calling *instanc* (if any). Note that *replace*$'$ is used to η-reduce while doing the replacement. There is no need to remove duplicates from the list generated, each subexpression of t considered will give rise to a different result:

```
abstract x e t = [replace' t loc r | (s, loc) ← subexps t,
                                     not (x ‘occursin‘ s),
                                     r ← instanc x e s]
```

The function *instanc* is so named because *instance* is a keyword in Haskell. We first use *params* and *body* to split up e into a list of arguments and a body; these functions also convert the arguments into pattern variables from local variables. As we remarked in Section 4.2.1, it is necessary to treat unbound local variables from *body* e and t as constants; therefore we *freeze* them before starting simple matching and *thaw* then in the resulting substitutions. The function *funcom* lifts a function on expressions to one on substitutions by applying it to every expression in the range of the substitution. Finally, the use of *evalSt* $([\,],[\,])$ provides a name supply to the simple matching algorithm; since *simpleappresolve* does not use any fresh names it is safe for this name supply to be empty:

```
instanc x e t =
  [foldl apl (LVar x (exptype e)) (apply φ xs)
                              | let xs = params e,
                                φ ← simplematch (body e) t]
  where simplematch p t =
              map (funcom (thaw cs)) $
                evalSt ([], []) $
                  matches simpleappresolve [Rule (p', t')]
          where bs = unboundlocals p
                cs = unboundlocals t
                p' = freeze bs p
                t' = freeze cs t
```

5.3 Efficiency

Whilst the above implementation of our algorithms is easy to follow given the more abstract descriptions from the previous chapters, using it directly leads to abysmal performance, making MAG completely unusable. It turns out that two optimisations in particular bring running times down to an acceptable level. In this section we discuss these two optimisations, various other possibilities that are only of marginal value at best, and finally give some rough-and-ready performance statistics to give some quantitative indication of the value of each optimisation.

5.3.1 Viability test

Often, it is possible to quickly establish that a particular rule cannot be solved. In particular, any constant in the pattern that does not appear in a subexpression with a flexible head cannot be removed by substitution followed by any amount of β-reduction; such a constant is known as a *rigid*. If some rigid in the pattern does not appear anywhere in the term, we know that there can be no matches. The *viable* predicate encodes this condition:

$$\mathit{viable}\ p\ t \ = \ \mathit{rigids}\ p \ \subseteq \ \mathit{consts}\ t$$

It is easy to define *rigids* and *consts*. Note that local variables bound outside p and t are also considered as constants for our purposes.

We can take this a step further. If *viable p t* is false, then p will not match against any subexpression of t either. Thus, by exporting the *viable* function to the term rewriting engine, we can prevent MAG from even trying to apply rules that have no hope of success.

We also make use of *viable* in the body of *matches*. If one of the rules returned by *resolve* is not viable, we can throw away the entire set it is contained in:

```
matches appresolver (x : xs) =
   do (σ, ys) ← expresolve appresolver x
      if all viableRule ys
       then do φ ← matches appresolver (apply σ (xs ++ ys))
               return (φ ∘ σ)
       else liftSt [ ]
viableRule (Rule (p, t)) = viable p t
```

Finally, it also turns out to be useful to check for viability early on in the definition of *apps*. The second element of the tuple it returns, which is always a subexpression of the *term* parameter, will be matched against the argument part of the pattern of the current rule. Thus, if we modify *onestepappresolve* slightly to pass this argument part to *apps*, we can filter the subexpressions as they are generated:

```
apps arg term =
   do x ← freshVar
      (subexp, locs) ← liftSt $ collect $ filter checkse $ subexps term
      locs′ ← liftSt $ subsplus locs
      let t = exptype subexp
      let body = replaces term locs′ (LVar x t)
      let func = lambda (x, t) body
      if normal func then return (func, subexp)
                     else liftSt [ ]
  where viablese (subexp, _ ) = viable arg subexp
        okse (subexp, _ ) = unboundlocals subexp ⊆ oklocals
        oklocals = unboundlocals term
        checkse se = viablese se ∧ okse se
```

Another possible check we could make is to verify that the pattern and term of a rule have compatible types. However, this turns out not to produce any performance gain in practice.

5.3.2 Checking flexibility

The definition of *twostepappresolve* includes a check for whether the function part of the pattern is flexible or not; if not, then we only need to consider the results from *simpleappresolve*. This check is an essential part of the two-step algorithm, whereas it is not needed for the one-step algorithm. However, we can include it anyway and improve performance significantly:

```
$onestepappresolve\ f\ a\ t\ =$
    $\mathbf{if}\ flex\ f\ \mathbf{then}\ simpleappresolve\ f\ a\ t$
                        $+\!\!+_m$
                        $\mathbf{do}\ \ (f', a') \leftarrow apps\ a\ t$
                              $return\ [Rule\ (f,\ f'),\ Rule\ (a,\ a')]$
                        $+\!\!+_m$
                        $\mathbf{do}\ \ x \leftarrow freshVar$
                              $return\ [Rule\ (f,\ lambda\ (x, exptype\ a)\ t)]$
                 $\mathbf{else}\ \ simpleappresolve\ f\ a\ t$
```

5.3.3 Other optimisations

The two optimisations described above are by far the most important for achieving acceptable performance. The following are some other changes that can be made, some of which have some effect on performance and some of which have none. We detail the performance results in the next section.

- Closely following the description of the algorithm for *abstracts* from Section 4.2.1 leads to an obvious inefficiency – *abstractssub* $n\ x\ e\ t$ will be computed once directly in the body of *abstracts*, and once for each *abstracstssub* $m\ x\ e\ t$ with $m > n$ that is computed. We use MAG to address this deficiency in Section 6.5.

- The function *apps* constructs a λ-abstraction $\lambda x.B$ and then throws it away if it is not normal. Since B is constructed by replacing subexpressions of T with x, and T is itself normal, the only possibility for $\lambda x.B$

not to be normal is if T is of the form $(T_0\ T_1)$ and T_1 is replaced by x. This is equivalent to the $locs'$ value being precisely the set of locations $\{\langle Arg \rangle\}$, so we can apply this less expensive test instead.

- The choice to merge the *Name* monad and the list monad using a monad transformer to get the *NameL* monad gives us an elegant program, but means that the name supply is "forked" for the computation of each element of the list, rather than being threaded between the computations. As a result, a clever compiler such as GHC cannot optimise the passing of the name supply by storing it in one place rather than repeatedly copying it.

  Instead, we can abandon the idea of using a monad transformer and make use of the subtly different datatype *Name* $[\alpha]$ to pass around lists of αs, where *Name* is the normal state monad encapsulating a name supply.

- In the definition of *matches*, we take the new rule set generated by *resolve* and add it to the beginning of the other rules we had. We have freedom in how we order the rules in a rule set, and therefore it might make sense to delay consideration of rules that could be considered "hard", in the hope that resolving other rules would make this unnecessary. A "hard" rule would be defined as one where the pattern was an application with a flexible head.

5.3.4 Performance tests

In order to provide an approximate indicator of the effectiveness of each of the above optimisations, we conducted some simple timing tests by running MAG on a set of about twenty examples, with the above optimisations progressively enabled. A more detailed performance analysis would require some substantially bigger examples and some more work on tuning the implementation appropriately. We have also not investigated the formal complexity of our algorithms, second-order matching is known to be NP-complete [8] and

Optimisation	*Running time*
None	>24h
Viability test	43m 44.96s
Checking flexibility	21.02s
Improving *abstracts*	18.51s
Improving *apps*	18.72s
Use of *Name* $[\alpha]$ datatype	18.60s
Considering “hard” rules last	18.79s

Table 5.1: MAG performance tests

thus one-step matching must be at least NP-hard since it returns at least all the second-order results. The restrictions we impose on two-step matching mean that we cannot infer anything from the known complexity of third-order matching (also NP-complete [22, 102]), but it seems likely that it is also at least NP-hard.

Table 5.1 shows the total running time taken by MAG on our set of examples, with the listed optimisations progressively enabled. These figures are almost meaningless for any purpose but comparison with each other, so we omit details of the environment in which they were produced. We find that applying the viability test brings running times down from being completely unacceptable on almost all examples to merely almost entirely unacceptable; adding the test for flexibility brings them down further to being quite reasonable on the size of examples we have tried. Of the remaining optimisations, only the improvements to *abstracts* had a significant impact; it is possible that the others would become more important for substantially larger examples.

Chapter 6

Examples

In this chapter, we shall give various examples of examples of derivations that we have successfully mechanised using MAG. The actual input to and output from MAG is rather long, so we just present the essential details here. The full text can be found in Appendix B.

6.1 Minimum depth

We introduced this example in Section 1.2 and in Section 4.1.1 gave it as an example which requires the use of the two-step algorithm. Here we present the details of applying the transformation in MAG.

First, recall the datatype of binary trees:

$$\textbf{data}\ \mathit{Tree}\ \alpha \ =\ \mathit{Leaf}\ \alpha \mid \mathit{Bin}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)$$

The fold on this datatype *foldbtree* is defined by

$$\begin{aligned} \mathit{foldbtree}\ b\ l\ (\mathit{Leaf}\ x) &= l\ x \\ \mathit{foldbtree}\ b\ l\ (\mathit{Bin}\ t_1\ t_2) &= b\ (\mathit{foldbtree}\ b\ l\ t_1)\ (\mathit{foldbtree}\ b\ l\ t_2) \end{aligned}$$

The corresponding promotion law states that

$$\begin{array}{rcl} f\ (\mathit{foldbtree}\ b\ l\ t) & = & \mathit{foldbtree}\ b'\ l'\ t \\ \text{if} & & \forall a. f\ (l\ a) = l'\ a \\ & & \forall x\ y. f\ (b\ x\ y) = b'\ (f\ x)\ (f\ y) \end{array}$$

Now, *mindepth* was defined by

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

Recall that the specification of *md* states:

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

We write this in a form that combines both the specification of the optimisation and the decision that it should be transformed using promotion on the argument t:

$$\mathit{md}\ t\ d\ m\ =\ \mathit{min}\ (\mathit{mindepth}\ (\mathit{foldbtree}\ \mathit{Bin}\ \mathit{Leaf}\ t)\ +\ d)\ m$$

It now remains to apply promotion to *md*, a derivation that almost mirrors that found in Section 1.2. However, there is one detail we must be careful of. One of the steps of that derivation was:

$$\begin{aligned} & \mathit{min}\ ((\mathit{min}\ (\mathit{mindepth}\ s)\ (\mathit{mindepth}\ t))\ +\ (1+d))\ m \\ = \quad & \{\text{the result of } \mathit{mindepth} \text{ is non-negative}\} \\ & \textbf{if}\ 1+d \geq m\ \textbf{then}\ m \\ & \qquad\qquad \textbf{else}\ \ \mathit{min}\ (\mathit{min}\ (\mathit{mindepth}\ s)\ (\mathit{mindepth}\ t) \\ & \qquad\qquad\qquad\qquad +\ (1+d)) \\ & \qquad\qquad\qquad m \end{aligned}$$

This step involves making use of the fact that in a particular case we can short-circuit the evaluation of both arguments of *min*. As a result, the original expression appears as a subexpression of the final expression, since if this case does not apply it must be evaluated as before. Therefore, any rewrite rule that carried out this step would lead to an infinite (and useless) rewriting chain. Therefore, we formulate a rule that encompasses both this step and the next step (which distributed + over the inner *min* in the expression):

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

Introducing this rule could be considered to be rather dangerous, since it is not true if a or b is negative, and in a long derivation might be applied in an unexpected place. In this instance we can deal with this problem by using a datatype for numbers that excludes negative values entirely, but in general this is not an adequate solution. One answer would be to annotate the rule with the relevant assumption. Using a theorem prover to discharge this assumption would be a possibility, but this is a rather heavyweight technique that might not succeed, and goes against our principle that the behaviour of MAG should be easy to predict and understand. A better option would be for MAG simply to remember that the assumption was made and present this to the user along with the final result (with the relevant pattern variables appropriately instantiated). It would then be incumbent on the user to verify that the rule had been used appropriately.

With these difficulties dealt with, the rest is straightforward. We give MAG the above rules, together with rules giving associativity of + and *min* and the definitions of +, *md* and *mindepth*, to get the following result:

$$\begin{array}{ll}
& \mathit{md} \\
= & \quad \{\text{promotion } (\ \ldots \ )\} \\
& \mathit{foldbtree}\ (\lambda f\ g\ d\ m.\, \textbf{if}\ 1+d \geq m \\
& \qquad\qquad\qquad\quad \textbf{then}\ m \\
& \qquad\qquad\qquad\quad \textbf{else}\ \ f\ (1+d)\ (g\ (1+d)\ m)\ ) \\
& \qquad\quad (\lambda x\ d\ m.\mathit{min}\ d\ m)
\end{array}$$

Unfolding (and η-expanding) this new definition for *md* gives us the more readable

$$\begin{array}{lll}
\mathit{md}\ (\mathit{Leaf}\ x)\ d\ m & = & \mathit{min}\ d\ m \\
\mathit{md}\ (\mathit{Bin}\ t_1\ t_2)\ d\ m & = & \textbf{if}\ 1+d \geq m \\
& & \textbf{then}\ m \\
& & \textbf{else}\ \ \mathit{md}\ t_1\ (1+d)\ (\mathit{md}\ t_2\ (1+d)\ m)
\end{array}$$

If we now combine this with the definition of *mindepth* in terms of *md*, we have the desired optimisation:

$$\mathit{mindepth}\ t \;=\; \mathit{md}\ t\ 0\ \infty$$

6.2 Alpha-beta pruning

A rather more sophisticated optimisation than *mindepth* is the problem of *alpha-beta pruning*. Consider a program designed to play a game such as chess. In order to choose a move from any given position, the program will generate a tree representing possible paths of play from that position. The position at the end of each path will have a score associated with it, representing the value of that position to the computer. The computer then searches through this tree trying to choose the move that will result in the best possible situation for it; the key point to note is that alternate moves are controlled by the opponent who will presumably be trying to choose moves that will result in the *worst* possible situation for the computer, and therefore the computer needs to allow for this when evaluating moves.

The procedure by which the tree is searched is known as *minimaxing*; the value of the top node of the tree is the maximum of the values of all the nodes at the next level down. These values are calculated by taking the *minima* of those one further down, which are calculated by taking maxima of those below them, and so on. The value of a leaf is simply the value of the position at that leaf.

As with *mindepth*, this algorithm is clear but inefficient; it will often be the case that we can ignore large sections of the search tree. For example, consider the tree shown in Figure 6.1. Scanning it from right-to-left, it is immediately apparent that the computer can make a score of 3 by making move A. If on the other hand the computer chooses move B, then the other player could choose move C and obtain a final result of 2 for the computer. Thus, it is apparent just by examining the leaves marked 2 and 3 that choosing move A will result in a score of 3, and assuming rational and intelligent opposition move B will result in a score of at most 2. Clearly it is not worth

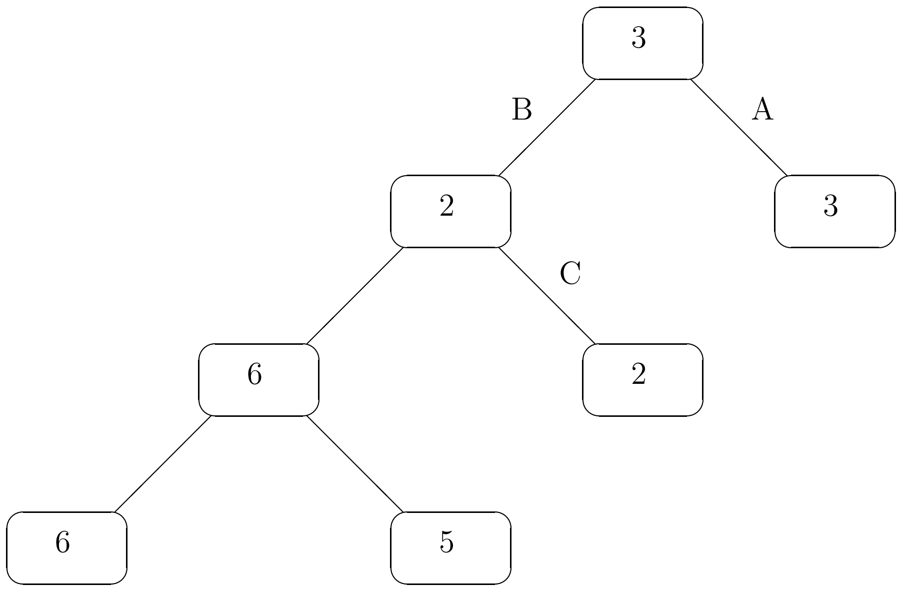

Figure 6.1: A game tree: labelled leaves indicate termination of the game with that score for the computer, and labelled nodes show the overall value of that node for the computer.

exploring move B further.

We can generalise this insight by maintaining two accumulating parameters representing the minimum and maximum possible values for the current tree being explored. This allows us to cut off the search of a subtree once we know that the subtree cannot affect the eventual result. This optimisation was first applied in a functional setting by Bird and Hughes [13].

An alternative approach to the minimax algorithm is to always maximise the *negation* of the values at the level beneath; this approach avoids the need to continually swap back and forth between minimising and maximising and thus simplifies the statement of the optimisation problem. MAG is capable of dealing with the problem in either form, but for the purposes of simplicity of presentation we take the latter approach here. We also require that the input tree must have leaves on alternate levels negated; this saves us from having to introduce an extra parameter to keep track of whose move it is.

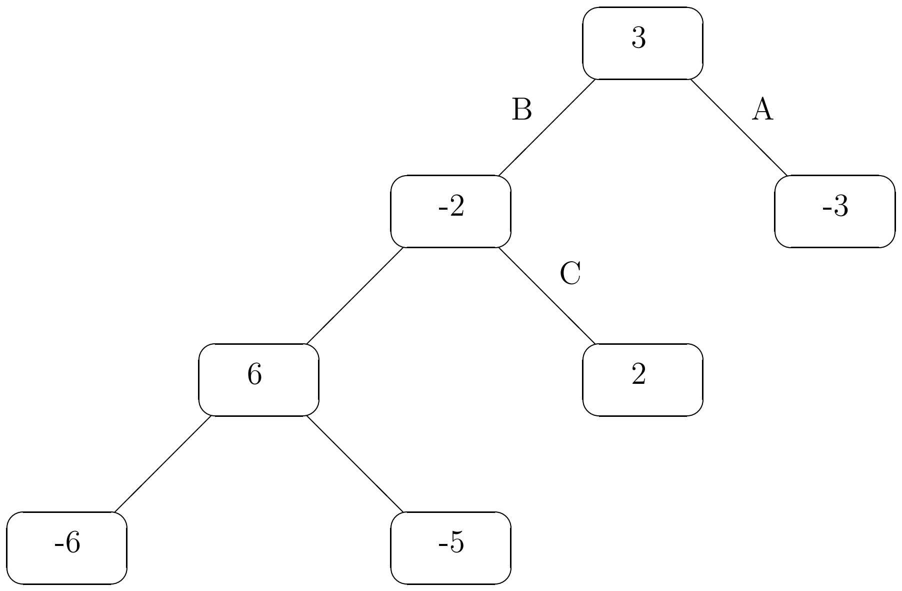

Figure 6.2: The game tree from Figure 6.1 with leaves and nodes for the opponent's turn negated

Figure 6.2 shows the tree from Figure 6.1 appropriately modified.

The datatype for rose trees, *RTree* is defined as follows:

$$\textbf{data}\ \mathit{RTree}\ \alpha\ =\ \mathit{RLeaf}\ \alpha \mid \mathit{RNode}\ [\mathit{RTree}\ \alpha]$$

This datatype differs from the ones we have considered so far because the recursion in the definition is actually an implicit expression of two *mutually* recursive datatypes. The following would be equivalent:

$$\begin{aligned}\textbf{data}\ \mathit{RTree}\ \alpha\ &=\ \mathit{RLeaf}\ \alpha \mid \mathit{RNode}\ (\mathit{RForest}\ \alpha)\\ \textbf{data}\ \mathit{RForest}\ \alpha\ &=\ \mathit{RNil} \mid \mathit{RCons}\ (\mathit{RTree}\ \alpha)\ (\mathit{RForest}\ \alpha)\end{aligned}$$

As a result, the fold function over *RTree*s is also defined by mutual recursion; as with other fold functions it replaces the constructors in its argument with functions.

$$\mathit{rtreefold}\ \mathit{leaf}\ \mathit{node}\ \mathit{cons}\ \mathit{nil}\ (\mathit{RLeaf}\ x)\ =\ \mathit{leaf}\ x$$

$$
\begin{aligned}
&\mathit{rtreefold\ leaf\ node\ cons\ nil}\ (\mathit{RNode\ ts}) \ = \\
&\quad \mathit{rforestfold\ leaf\ node\ cons\ nil\ ts} \\
&\mathit{rforestfold\ leaf\ node\ cons\ nil\ RNil} \ = \ \mathit{nil} \\
&\mathit{rforestfold\ leaf\ node\ cons\ nil}\ (\mathit{RCons\ t\ ts}) \ = \\
&\quad \mathit{cons}\ (\mathit{rtreefold\ leaf\ node\ cons\ nil\ t}) \\
&\qquad\quad (\mathit{rforestfold\ leaf\ node\ cons\ nil\ ts})
\end{aligned}
$$

This mutual recursion also complicates the definition of the promotion law somewhat; the side conditions generate a new function f' which does not appear on the right-hand side of the main law. This function is effectively the equivalent of f for the *rforestfold* part of the fold.

$$
\begin{aligned}
& && f\ (\mathit{rtreefold\ leaf\ node\ cons\ nil\ t}) \\
&= && \mathit{rtreefold\ leaf'\ node'\ cons'\ nil'\ t} \\
&\text{if} && \forall a. f\ (\mathit{leaf}\ a) = \mathit{leaf'}\ a \\
& && f'\mathit{nil} = \mathit{nil'} \\
& && \forall x\ y. f'\ (\mathit{cons}\ x\ y) = \mathit{cons'}\ (f\ x)\ (f'\ y) \\
& && \forall a. f\ (\mathit{node}\ a) = \mathit{node'}\ (f'\ a)
\end{aligned}
$$

The naïve evaluation function for a game tree, which we name *flipeval* because it evaluates the values of all its children and then flips their sign, is defined as follows:

$$
\begin{aligned}
&\mathit{flipeval}\ (\mathit{RLeaf}\ x) \ = \ x \\
&\mathit{flipeval}\ (\mathit{RNode\ ts}) \ = \ \mathit{listmax}\ (\mathit{map}\ (\mathit{negate} \circ \mathit{flipeval})\ \mathit{ts})
\end{aligned}
$$

The quantity that *flipeval* is evaluating is the value of the current position to whoever is next to play. Thus, if the current position is a leaf, the game has terminated with that score to them. Otherwise, the position is a node, each of whose children evaluate to the value of that choice *to the opponent*. Therefore, we evaluate them, and negate the result to give the value to the current player before taking the maximum.

To optimise this, we first introduce the function *bound*, which given a value x and lower and upper bounds a and b respectively, gives x if it is between the bounds, a if it is below them and b if it is above them:

$$\mathit{bound}\ x\ a\ b\ =\ \mathit{min}\ (\mathit{max}\ a\ x)\ b$$

We can now specify the function *fastflipeval*, which evaluates a tree t subject to bounds a and b. It can be used to redefine *flipeval* simply by instantiating the bounds to $-\infty$ and ∞ respectively.

$$\begin{aligned}
&\mathit{fastflipeval}\ t\ a\ b\ =\ \mathit{bound}\ (\mathit{flipeval}\ (\mathit{rtreefold}\ \mathit{RLeaf}\ \mathit{RNode}\ (:)\ [\,]\ t))\ a\ b\\
&\mathit{flipeval}\ t\ =\ \mathit{fastflipeval}\ t\ \ {-\infty}\ \infty
\end{aligned}$$

We now give the rewrite rules that will be required. The obvious rule for distributing *bound* over *max* is the following:

$$\mathit{bound}\ (\mathit{max}\ c\ d)\ a\ b\ =\ \mathit{bound}\ c\ (\mathit{bound}\ d\ a\ b)\ b$$

In other words, we first update the lower bound using the value of d, and then bound c using the new bounds.

However, consider what will happen if $d \geq b$. In this case, *bound* $d\ a\ b$ will evaluate to b, and so the entire expression will evaluate to b without any need to examine the value of c. It is this observation which encodes the insight we described earlier that allows parts of the search to be cut off.

Therefore, we give the following rewrite rule:

$$\begin{aligned}
&\mathit{bound}\ (\mathit{max}\ c\ d)\ a\ b\ =\ \textbf{if}\ a' == b\ \textbf{then}\ b\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{bound}\ c\ a'\ b\\
&\quad \textbf{where}\ \ a' = \mathit{bound}\ d\ a\ b
\end{aligned}$$

In addition to this rule, we will need to know how *bound* distributes over *negate* and how it interacts with $-\infty$:

$$\begin{aligned}
&\mathit{bound}\ (\mathit{negate}\ c)\ a\ b\ =\ \mathit{negate}\ (\mathit{bound}\ c\ (\mathit{negate}\ b)\ (\mathit{negate}\ a))\\
&\mathit{bound}\ \ {-\infty}\ a\ b\ =\ a
\end{aligned}$$

Applying these rules together with the appropriate definitions and unfolding gives us the following program. Note that the definition of *bound* is *not* included in the set of rewrite rules; since it does not pattern match on its argument, such a rewrite rule would be always applicable and would interfere with the more specialised rules above that are essential to this derivation.

```
$fastflipeval\ (Leaf\ x)\ a\ b\ =\ bound\ x\ a\ b$
$fastflipeval\ (Node\ ts)\ a\ b\ =\ fastflipeval'\ ts\ a\ b$
$fastflipeval'\ []\ a\ b\ =\ a$
$fastflipeval'\ (t : ts)\ a\ b\ =$
  if $a' == b$ then $b$
              else $negate\ (fastflipeval\ t\ (negate\ b)\ (negate\ a'))$
  where $a' = fastflipeval'\ ts\ a\ b$
```

6.3 Steep sequences

We now show how *tupling* optimisations can also be accomplished with promotion. A sequence (represented here as a list) is considered to be *steep* if each element is larger than the sum of all its successors. The program for this is written as follows:

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

For each element of the list, the sum of the rest of the list will be calculated and then thrown away, despite the fact that this value would be useful for the same calculation for any elements further to the left in the sequence. We can specify *faststeep* to keep track of both the steepness (or not) and the current sum:

$$\begin{aligned} faststeep\ xs &= (steep\ xs, sum\ xs) \\ steep\ xs &= fst\ (faststeep\ xs) \end{aligned}$$

We can apply promotion to this specification by replacing both instances of xs on the right-hand side of *faststeep* with $foldr\ (:)\ []\ xs$, but there is a difficulty. In unfolded form, the optimised program we are expecting would be:

$$\begin{array}{lcl} \mathit{faststeep}\,[\,] & = & (\mathit{True},\ 0) \\ \mathit{faststeep}\,(x : \mathit{xs}) & = & (x\ >\ \mathit{sm}\ \wedge\ \mathit{st},\ x + \mathit{sm}) \\ & & \qquad \textbf{where}\ (\mathit{st}, \mathit{sm})\ =\ \mathit{faststeep}\ \mathit{xs} \end{array}$$

In other words, one of the functions generated during matching will have to be a λ-abstraction of the form $\lambda\,(\mathit{st}, \mathit{sm})\ \ldots$, *i.e.* one which breaks up a tuple of values. Our matching algorithms are not capable of generating such functions, so MAG will fail to apply this optimisation.

To circumvent this problem, we *specialise* the promotion rule for a tupling optimisation. Define the functions *split* and *uncurry* as follows:

$$\begin{array}{lcl} \mathit{split}\,f\,f'\,x & = & (f\ x, f'\ x) \\ \mathit{uncurry}\,f\ (x, y) & = & f\ x\ y \end{array}$$

Now, recall the promotion rule for lists:

$$\begin{array}{rcl} f\,(\mathit{foldr}\ g\ e\ \mathit{xs}) & = & \mathit{foldr}\ h\ e'\ \mathit{xs} \\ \text{if} & & f\ e = e' \\ & & \forall x, y\ :\ f\,(g\,x\,y) = h\,x\,(f\ y) \end{array}$$

Replace the function f by the expression $\mathit{split}\,f\,f'$ and apply the definition of *split* on the right-hand side of the conditions. The first condition is unchanged, but the second becomes:

$$\forall x, y\ :\ (f\,(g\,x\,y), f'\,(g\,x\,y))\quad =\quad h\,x\,(f\ y, f'\ y)$$

Now, define h' by $h\,x = \mathit{uncurry}\,(h'\,x)$. The above condition now becomes

$$\forall x, y\ :\ (f\,(g\,x\,y), f'\,(g\,x\,y))\quad =\quad h'\,x\,(f\ y)\,(f'\ y)$$

Since it is h' that the matching algorithm has to find a value for, it no longer needs to construct a λ-abstraction that takes apart a tuple, and MAG will be able to apply the following modified promotion rule:

$$\begin{aligned} \mathit{split}\,f\,f'\,(\mathit{foldr}\;g\;e\;\mathit{xs}) &= \mathit{foldr}\,(\lambda x.\mathit{uncurry}\,(h'\,x))\;e'\;\mathit{xs} \\ &\text{if}\quad \mathit{split}\,f\,f'\;e = e' \\ &\quad\;\; \forall x, y\ :\ \mathit{split}\,f\,f'\,(g\;x\;y) = h'\;x\;(f\;y)\;(f'\;y) \end{aligned}$$

With this difficulty dealt with, we rewrite the specification of *faststeep* in terms of *split*, *steep* and *sum* and add the identity fold as a seed, as usual:

$$\mathit{faststeep}\;\mathit{xs} \;=\; \mathit{split}\;\mathit{steep}\;\mathit{sum}\;(\mathit{foldr}\;(:)\;[\,]\;\mathit{xs})$$

It just remains to give MAG the definitions of *steep*, *sum* and *split* to apply the optimisation. No extra rewrite rules are required.

6.4 The path sequence problem

The most complex derivation we have carried out using MAG is Bird's *path sequence problem* [9]. The problem is this: given a directed graph and a list of vertices *xs*, determine the length of the longest (not necessarily contiguous) subsequence of *xs* that is a connected path in the graph. The graph is represented as a predicate *arc* on pairs of vertices indicating whether they are connected or not.

The straightforward program for this algorithm is the following. The function *subs* simply returns all subsequences of its argument.

$$\mathit{llp}\;\mathit{xs} \;=\; \mathit{listmax}\;(\mathit{map}\;\mathit{length}\;(\mathit{filter}\;\mathit{path}\;(\mathit{subs}\;\mathit{xs})))$$

The *path* function is defined in the obvious way:

$$\begin{aligned} \mathit{path}\;[\,] &= \mathit{True} \\ \mathit{path}\;[x] &= \mathit{True} \\ \mathit{path}\;(x : y : \mathit{xs}) &= \mathit{arc}\;x\;y \;\wedge\; \mathit{path}\;(y : \mathit{xs}) \end{aligned}$$

Since the number of subsequences of *xs* is exponential in the length of *xs*, so is this algorithm. However, it seems likely that there will be a lot of duplicated work, since many of the subsequences will have large parts in common with each other.

To gain a hint as to how we might optimise it, we first do some calculation (using MAG) on *llp* applied to empty and non-empty lists. With the addition of a few reasonably straightforward rewrite rules to the definitions, we obtain the following:

$$\begin{aligned}
&llp\,[\,] &&= 0\\
&llp\,(x : xs) &&= max\ (listmax\ (map\ length\ (filter\ path\ (subs\ xs)))\\
& && \qquad (1 + listmax\ (map\ length\\
& && \qquad\qquad (filter\ (\lambda ys.path\ (x : ys))\ (subs\ xs))
\end{aligned}$$

The first argument to *max* in this new formula for $llp\,(x : xs)$ is just $llp\,xs$. The second argument suggests that it might make sense to introduce a new function, llp', defined as follows:

$$llp'\ x\ xs\ =\ listmax\ (map\ length\ (filter\ (\lambda ys.path\ (x : ys))\ (subs\ xs)))$$

This function returns the length of the longest path in *xs* that could be prefixed by *x* and still give a path.

Having done this, we can define *fastllp* using tupling, Notice that the second element of the tuple is a function

$$fastllp\ xs\ =\ (llp\ xs,\ \lambda x.llp'\ x\ xs)$$

Writing this in the form required to apply the specialised promotion rule for tupling we described in the previous section, we get:

$$split\ llp\ (\lambda ys\ x.llp'\ x\ ys)\ (foldr\ (:)\ [\,]\ xs)$$

Applying promotion gives the following (unfolded) result. Again, the extra rewrite rules required are relatively straightforward.

$$\begin{aligned}
&fastllp\,[\,]\ =\ (0,\ \lambda x.0)\\
&fastllp\,(y : ys)\ =\ (max\ l\ l',\ \lambda x.\ \mathbf{if}\ arc\ x\ y\ \mathbf{then}\ max\ (f\ x)\ l'\ )\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ f\ x
\end{aligned}$$

$$\begin{aligned} \textbf{where}\ (l, f) &= \mathit{fastllp}\ \mathit{ys} \\ l' &= 1 + f\ y \end{aligned}$$

This program is certainly an improvement on the original; the intermediate lists produced by *subs* have been removed, as has the need to explicitly scan each of these lists from one end to the other to check whether it is a path. However, the presence of the function in the second element of the tuple, coupled with the fact that this function makes two calls to the corresponding function on the remainder of the list, means that this is still an exponential program.

To do better than this, we define the following.

$$\mathit{llp}''\ \mathit{ts}\ \mathit{xs} = \mathit{max}\ (\mathit{llp}\ \mathit{xs})\ (\mathit{listmax}\ (\mathit{map}\ (\lambda(k, x).k + \mathit{llp}'\ x\ \mathit{xs})\ \mathit{ts}))$$

The idea is essentially that we keep track of a list of the current possible path prefixes. We do not need to remember all the elements of each prefix, simply the length and the last element.

We can then redefine *llp* by instantiating the list of prefixes to the empty list:

$$\mathit{llp}\ \mathit{xs} = \mathit{llp}''\ [\,]\ \mathit{xs}$$

To apply promotion to our definition of *llp*″, we first rewrite the λ-abstraction so that it does not have a tuple for a parameter. It is not hard to add support for such λ-abstractions if we do not expect the matching algorithms to *return* such results, but it would have complicated the description of the implementation in the previous chapters somewhat and we therefore chose to leave this out.

$$\begin{aligned} \mathit{llp}''\ \mathit{ts}\ \mathit{xs} = {} & \mathit{max}\ (\mathit{llp}\ \mathit{xs}) \\ & \quad (\mathit{listmax}\ (\mathit{map}\ (\lambda kx.(\mathit{fst}\ kx + \mathit{llp}'\ (\mathit{snd}\ kx)\ \mathit{xs})\ \mathit{ts})) \end{aligned}$$

Next, we define *fastllp*″ with the usual identity fold. Since we are expecting to obtain an expression for *fastllp*″ expressing it as a fold on the *xs* that manipulates the accumulating parameter *ts*, we write the definition *fastllp*″ with the parameter *xs* first:

$$\mathit{fastllp''}\ \mathit{xs}\ \mathit{ts} \;=\; \mathit{llp''}\ \mathit{ts}\ (\mathit{foldr}\ (:)\ [\,]\ \mathit{xs})$$

Most of the rewrite rules required for the derivation are quite straightforward, as before. However, there are two that are not. Firstly, at one point MAG needs to apply to commutativity of *max*. Clearly we cannot just use the obvious rule; instead we specialise it so that it will only apply forwards at the appropriate point.

The second is the following:

$$v + (1 + \mathit{llp'}\ y\ \mathit{ys}) = (\lambda kx.\mathit{fst}\ kx + \mathit{llp'}\ (\mathit{snd}\ kx)\ \mathit{ys})\ (1 + v, y)$$

This rule essentially states how a newly computed path prefix can be expressed as a tuple of the length and the final element (we derived the precise form of this and the specialised rule for the commutativity of *max* by observing where the derivation got stuck without the rules present).

Running this derivation through MAG and unfolding gives the following quadratic time program:

$$\begin{array}{ll}
\mathit{llp''}\ \mathit{ts}\ [\,] & = \mathit{max}\ 0\ (\mathit{listmax}\ (\mathit{map}\ \mathit{fst}\ \mathit{ts})) \\
\mathit{llp''}\ \mathit{ts}\ (x : \mathit{xs}) & = \mathit{llp''}\ ((v, x) : \mathit{ts})\ \mathit{xs} \\
\quad \textbf{where}\ \ v = 1 + \mathit{max}\ 0\ (\mathit{listmax}\ (\mathit{map}\ \mathit{fst} & \\
\qquad\qquad (\mathit{filter}\ (\lambda t.\mathit{arc}\ (\mathit{snd}\ t)\ x)\ \mathit{ts}))) &
\end{array}$$

6.5 Optimising *abstracts*

For our final example we move away from promotion calculations and carry out a derivation on *fixed points* instead. As we remarked in Section 5.3.3, the definition of the *abstracts* function that we gave in Section 5.2.5 has the disadvantage that it repeatedly recomputes certain values. Here we show how MAG can be used to eliminate this inefficiency.

Rewriting the definitions to use the *map* and *concat* functions instead of list comprehensions (which MAG does not support), we have the following:

$$\begin{array}{lcl}
\mathit{abstracts}\ x\ e\ t & = & \mathit{concat}\ \$\ \mathit{takeWhile}\ (\mathit{not}.\mathit{null})\ \$ \\
 & & \qquad \mathit{map}\ (\lambda n.\mathit{abstractssub}\ n\ x\ e\ t)\ (\mathit{from}\ 0)
\end{array}$$

$$\begin{aligned}
&\mathit{abstractssub}\ 0\ x\ e\ t &&= [t] \\
&\mathit{abstractssub}\ (n+1)\ x\ e\ t &&= \mathit{nub}\ (\mathit{concat}\ (\mathit{map}\ (\mathit{abstract}\ x\ e\ b) \\
& && \qquad\qquad\qquad (\mathit{abstractssub}\ n\ x\ e\ t)))
\end{aligned}$$

We have also written $[0\ldots]$ as the expression *from* 0, where the function *from* is defined in terms of a more general function *iterate*:

$$\begin{aligned}
\mathit{iterate}\, f\ x &= x : \mathit{iterate}\, f\ (f\ x) \\
\mathit{from}\ n &= \mathit{iterate}\ (+1)\ n
\end{aligned}$$

It would be nicer to be able to define *from* as a directly recursive function:

$$\mathit{from}\ n = n : \mathit{from}\ (n+1)$$

Unfortunately, it is impossible to then use MAG to automatically derive the version based on *iterate*. Any attempt to use this definition as a rewrite rule would lead to an infinite rewriting chain, and thus we cannot "seed" our derivation as we did with promotion.

The first step of the derivation is to use the following *map-iterate* law to fuse the *map* and the *from* in the definition of *abstracts*:

$$\begin{aligned}
\mathit{map}\ g\ (\mathit{iterate}\, f\ x) &= \mathit{iterate}\ h\ y \\
\text{if} &\quad g\ x = y \\
&\quad \forall a\ :\ g\ (f\ a) = h\ (g\ a)
\end{aligned}$$

Next, we define *iterate* in terms of a more basic function *fix*. Placing the rewrite rule that applies this definition after the *map-iterate* law in the theory file ensures that this definition will only be applied after that law has been used.

$$\begin{aligned}
\mathit{fix}\ \mathit{rec} &= \mathit{rec}\ (\mathit{fix}\ \mathit{rec}) \\
\mathit{iterate}\, f\ x &= \mathit{fix}\ (\lambda g\ y.y : g\ (f\ y))\ x
\end{aligned}$$

The *fixpoint fusion* law states that:

$$\begin{array}{rcl} \mathit{func}\,(\mathit{fix}\;\mathit{rec}\,x) & = & \mathit{fix}\;\mathit{rec}'\,x \\ & \text{if} & \mathit{func}\text{ strict} \\ & & \forall f\;:\;\mathit{func}\circ(\mathit{rec}\,f) = \mathit{rec}'\,(\mathit{func}\circ f) \end{array}$$

We give MAG the *map-iterate* law, the fixpoint fusion law and the following two laws about **if**:

$$\begin{array}{rcl} \mathbf{if}\;\mathit{not}\;b\;\mathbf{then}\;x\;\mathbf{else}\;y & = & \mathbf{if}\;b\;\mathbf{then}\;y\;\mathbf{else}\;x \\ f\;(\mathbf{if}\;b\;\mathbf{then}\;x\;\mathbf{else}\;y) & = & \mathbf{if}\;b\;\mathbf{then}\,f\;x\;\mathbf{else}\,f\;y \end{array}$$

All this leaves us with the following program for *abstracts*:

```
abstracts x e t =
  fix (λf ts. if null ts
              then [ ]
              else  ts ++ f (nub (concat (map (abstract x e) ts)))) [t]
```

Since MAG cannot check the strictness condition in the application of fixpoint fusion itself, it is necessary to verify it manually. This is rather more important for fixpoint derivations than for those involving promotion, since there is much more potential to introduce non-termination.

Unfolding the definition of *fix* using the auxiliary function *abstracts′* leaves us with:

```
abstracts′ :: VarId → Exp → [Exp] → [Exp]
abstracts′ x e ts =
  if null ts
   then [ ]
   else  ts ++ (abstracts′ x e (nub (concat (map (abstract x e) ts))))
abstracts x e t = abstracts′ x e [t]
```

Chapter 7

Discussion

This work on active source is part of a larger effort currently underway to develop an *Intentional Programming* (IP) system [83]. The goal is to produce a system in which *domain-specific* languages, that is languages specialised for a particular programming task (such as GUI design or database access), can be implemented easily. The idea is that individual language features, known as *intentions* because they should be designed to enable users to program with them in an intuitive fashion, are implemented in a highly modular fashion. Higher-level intentions would be implemented in terms of lower-level intentions as much as possible. Of course, much of this could equally well be implemented as a combinator library in a very high-level language with a rich type system such as Haskell. Where IP differs substantially from this line of research is that an intention writer should be able to describe domain-specific optimisation opportunities that might arise when their intentions are being used, as well as giving a basic description of how they are constructed. It is here that our work fits in; it is very likely that many of these optimisations will suffer from the problems we outlined in the introduction, namely that completely automatic application would be infeasible or impossible.

For active source to be a success, the annotations to the code should be *robust*; if at all possible, the intended transformation should still apply after a change is made to the original program. Experience with MAG suggests that they are relatively immune to changes to the original program that are not directly relevant to the transformation; for example it is straightforward to reuse the core of the fast reverse annotations to optimise a similar program that carries out a post-order traversal of a rose tree. However, they tend to be extremely sensitive to changes that do affect the transformation; in an example such as *mindepth* which relies on the associativity of + to correctly update the accumulating parameter containing the current depth, altering the order of the arguments to + without also altering the sense of the associativity rule is fatal to the derivation. In such situations, *interactivity* is vital to help the programmer quickly identify that there is a problem and to rectify it with the help of information from the transformation system about where a derivation failed.

Of course, the source-to-source rewrites that we have focused on in this thesis are just one of many kinds of transformation that could benefit from being defined in annotations to the source code. The transformations we have described are essentially *algorithmic* refinements. Sanabria-Piretti [80] has shown how *packages* of first-order rewrite rules known as *transforms* can be used to carry out *data* refinements, translating a program from using abstract datatypes to concrete implementations. A key feature of his work is the abstract specification of a datatype can expose a limited interface that is not tuned for any particular implementation, but during the transformation process particular patterns of access can be recognised and translated to an efficient implementation; thus for example if the interface to a set includes operations to find the minimum element and to delete a given element and the set is then implemented as an ordered list, a sequence of operations in the original program that asks for the minimum element and then deletes it can be refined to simply finding the head of a list and then replacing the list by its tail.

7.1 MAG

Promotion is a powerful technique for program optimisation. Many complex derivations, some of which we have presented here, start with a program that can be expressed as a fold over an inductive datatype and add some accumulating parameters to produce another program that is a fold over the same datatype. Indeed, we have not yet found an example of such a derivation which cannot be recast in terms of the appropriate promotion rule. In addition, the combination of the two matching algorithms we have presented here has always been enough to mechanise the use of promotion, given the derivations for the individual side conditions. Of course, it is this last proviso that is the sticking point with MAG. As we made clear in Chapter 5, its rewriting strategy is rather primitive, and the experience of the path sequence problem in particular suggests that this approach will not scale well. On the other hand, the simplicity of MAG is also something of an advantage from the point of view of interactivity; it is relatively straightforward to understand *why* a transformation is not working, even if dealing with this problem may prove cumbersome. The clear specification of our matching algorithms is an important part of providing this understanding.

For larger examples, one solution might be to make use of configurable *rewriting strategies*, which are essentially meta-programs supplied by the user which guide the use of rewrite rules. They have been extensively studied in, for example, [21, 56, 91, 92]. If the required strategies for any particular problem could be expressed concisely, they could be included as part of the transformation annotation. A related but simpler possibility would be to continue to employ brute-force rewriting, but allow the user to override the process at certain points by specifying that a certain rule should be given priority when the current term matches a particular pattern (which would presumably be a specialisation of the left-hand side of the rule specified).

It is hoped that MAG can be a useful teaching tool (indeed, it was originally written for this purpose). Undergraduate students of functional programming are often taught the basics of calculational program derivation but do not have enough time to experiment enough to gain experience with this. By eliminating the need to deal with the tedious details of such calculations, students might be able to learn more of the guiding principles.

Clearly, MAG requires significant work to make it really usable by others as a tool for program derivation. Much of this work is mostly cosmetic in nature – most importantly, the parser needs to be modified to handle input syntax similar to that suggested on page 17, and the generation of the promotion rule for any particular datatype should be mechanised. The final program resulting from a promotion or fixpoint derivation is often somewhat unreadable and could be automatically unfolded to present it to the user. However, it should probably be left expressed in terms of the appropriate higher-order function for passing to GHC, given the potential for the short-cut deforestation optimisations we discussed in Section 1.3.2.

More significantly, the treatment of side conditions should be improved to allow different types of side conditions to be handled in a modular fashion. We have already seen that the lack of strictness analysis is a weakness for promotion and fixpoint derivations, but other conditions may be necessary in order to use MAG to apply different derivation rules. Another useful feature would be the ability to transform full Haskell – recent development of independent type-checking and parsing modules for Haskell would hopefully make implementing this easier [53, 73].

7.2 Variations on our algorithms

The decision to develop and use the matching algorithms presented here, rather than making use of existing matching or unification algorithms, was motivated by the desire to have algorithms that were both clearly specified and capable of solving the matching problems generated by the kind of transformation we have described here. We feel that this was a worthwhile goal; although all the matching problems we have encountered can also be solved by Huet's higher-order unification procedure [44] (as implemented in

λProlog [67]), the user is offered no guarantees that this will be the case – indeed, Larry Paulson, one of the main authors of Isabelle, a widely-used theorem prover which makes use of Huet's procedure, has described it as "powerful but sometimes ill-behaved" [75] and suggested instead using Huet and Lang's matching algorithm.

In Section 6.3, we described a procedure of specialising a promotion rule to deal with tupling optimisations. The original version of MAG, as described in [26], was based on the one-step matching algorithm only and use was made of a similar procedure to solve problems such as minimum depth. The idea is not new; Pfenning and Elliott suggested it for dealing with variable capture problems [78], based on an similar technique in [74]. Although this solution does work, it is somewhat inconvenient – in particular, it is rather difficult to understand the specialised rule. It seems better from the point of view of user transparency to deal with the issue in the matching algorithm, which performs a function that is intuitively easy to grasp. It was for this reason that we developed the two-step algorithm, and hope in future work to deal with the tupling issue also by developing a matching algorithm that can carry out matching *modulo products*, that is where necessary generating functions that deconstruct tuples.

With regard to other developments of our algorithms, various avenues of investigation are open. The next stage beyond matching modulo products would be to carry out matching modulo *coproducts* and generate functions with **case** statements to choose between the different elements of a constructed datatype. However, our suspicion is that it would be difficult to restrict this problem sufficiently to obtain a specification which guarantees decidability and a finite set of results but was also useful for solving non-trivial matching problems.

The names of the one-step and two-step algorithms naturally lead to questions about three-step, or even n-step matching. There are two reasons why we do not believe this would be a fruitful line of inquiry. Firstly, we remarked earlier that we have not yet found an example of a derivation of the kind that we are interested which we could not express in terms of a promotion rule. It is also the case that once recast in this way, none of these derivations were prevented from being mechanised because the results of our matching algorithms were not "higher-order enough". Thus, from a practical point of view we see no need to extend them in this direction, although we have not investigated transformations of programs involving continuation-passing code or parser combinators, two areas in which functions of quite high order naturally arise.

Secondly, we feel that the extra complexity of the two-step matching algorithm compared to that of the one-step algorithm suggests that any possible algorithm for three-step matching would be very difficult to produce and prove correct. Additionally, the restrictions on patterns that would be required to guarantee a finite set of results would be much stronger than those of the two-step algorithm, which might lead to problems when actually trying to use such an algorithm.

Another extension would be to incorporate some knowledge of semantic laws of the expressions being matched. For example, *associative-commutative matching* would allow the matching algorithm to make use of such algebraic properties of functions in the terms being matched; thus for example the pattern $x + 1$ could match against the term $1 + 2$. Various systems, for example ELAN [14] and Maude [56] have implemented this.

7.3 Implementation of matching

The λ-calculus is a powerful and simple syntax for encoding programs, but real programming languages are rather more complicated. For languages such as Haskell, it is straightforward to perform a translation, but for imperative languages in particular this is likely to be more complex. Recent work on *generic*, or *polytypic* programming may offer a solution; languages such as PolyP [5, 51] and Generic Haskell [49] (currently unfinished) allow programs to be written which traverse data structures in a manner that is independent of the actual datatype involved. A simple (first-order) pattern matcher has already been implemented in this way [50].

Conceptually, the application of a set of rewrite rules to an expression is best described by a procedure of matching the left-hand side of each rule in turn against the expression. This is also the simplest way to implement it, but is not very efficient since the term is traversed once for each rule. Cai [16] describes a procedure for carrying out simple matching against a particular term for many patterns at once by traversing the term *bottom-up*; extending this to higher-order matching would introduce significant complications since the matching process is no longer a completely straightforward tree traversal. Since the one-step algorithm works by abstracting subexpressions from the term when a function application is present in the pattern, the difficulties might be surmountable by keeping a list of all subexpressions seen while traversing the term. The core of the two-step algorithm uses the bodies of functions appearing in the pattern on the right-hand side of applications with flexible heads as patterns for simple matching, and thus adding all such bodies to the set of patterns being considered might make Cai's algorithm applicable here too.

Our implementation uses the traditional trees to represent λ-terms. Trees have the advantage of being easy to manipulate, but they are not necessarily the most efficient structure to use for term rewriting. In particular, trees do not allow us to take advantage of common subexpressions in the term being rewritten. Instead, we could represent terms as *fully collapsed jungles*, a kind of directed acyclic graph in which all occurrences of any particular subexpression are represented by a single subgraph [42]. It seems that this approach should be particularly valuable for the one-step matching algorithm, which frequently needs to collect common subexpressions, although profiling data from our performance tests suggest that there would not be a significant gain for the small examples we have worked with so far.

7.4 Other applications

Finally, higher-order matching also has applications beyond the field of term rewriting. Matching is used to recognise occurrences of the left-hand side of a rewrite rule in an expression before replacing the occurrence with the right-hand side of the rewrite rule (appropriately instantiated). Allowing side-conditions enables some quite general transformations to be expressed in such a way, but rewrite rules are not always powerful enough. In some cases, using matching to recognise application sites before doing more complex analysis to actually carry out the transformation would make sense. For example, loop strengthening is a transformation that identifies situations in a loop body where the loop variable is multiplied by a constant factor on each iteration, and replaces this with a new variable that is appropriately initialised and incremented by that constant factor each time instead. Pattern matching would be appropriate for spotting this pattern, but the actual transformation requires some careful calculation and code reorganisation that might be best carried out by a more sophisticated processor than a term rewriting engine.

Bibliography

Numbers following each entry indicate the page(s) where references occur.

[1] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Algorithms and complexity*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, The Netherlands, 1990. 31

[2] L. Augustsson, M. Rittri, and D. Synek. Functional pearl: On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994. 126

[3] J. Avenhaus and C. Loría-Sáenz. Higher-order conditional rewriting and narrowing. In J. Jouannaud, editor, *First International Conference on Constraints in Computational Logics*, number 845 in Lecture Notes in Computer Science, pages 269–284. Springer-Verlag, 1994. 121

[4] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. 18

[5] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming – an introduction. In *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, 1998. 163

[6] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1984. 40, 64, 65, 65

[7] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford Science Publications, 1992. 32, 65

[8] L. D. Baxter. *The Complexity of Unification.* Ph.D. thesis, University of Waterloo, 1976. 140

[9] R. S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, 1984. Erratum, ibid. 7(3):490–492, 1985. 13, 152

[10] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987. 14

[11] R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989. 14

[12] R. S. Bird. *Introduction to Functional Programming in Haskell.* International Series in Computer Science. Prentice Hall, 1998. 124

[13] R. S. Bird and R. J. M. Hughes. An alpha–beta algorithm: an exercise in program transformation. *Information Processing Letters*, 24(1):53–57, 1987. 146

[14] P. Borovanský, C. Kirchner, H. Kirchner, P. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Available from URL: `http://www.elsevier.nl/gej-ng/31/29/23/29/23/37/tcs4004.ps`. 163

[15] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977. 13

[16] J. Cai, R. Paige, and R. E. Tarjan. More efficient bottom-up multipattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, 1992. 164

[17] W. Chin. Safe fusion of functional expressions. In *7th ACM Conf on Lisp and Functional Programming*, pages 11–20. ACM Press, 1992. 23

[18] W. Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, 1994. 23

[19] W. Chin and S. Khoo. Better consumers for deforestation. In D. Swierstra, editor, *Programming Languages: Implementations, Logics and Programs*, volume 982 of *Lecture Notes in Computer Science*, pages 223–240. Springer-Verlag, 1995. 23

[20] O. Chitil. Type-inference based short cut deforestation (nearly) without inlining. In C. Clack and P. Koopman, editors, *Eleventh International Workshop on Implementation of Functional Languages*, volume 1868 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, 2000. 24

[21] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996. 160

[22] H. Comon and Y. Jurski. Higher-order matching and tree automata. In M. Nielsen and W. Thomas, editors, *Proc. Conf. on Computer Science Logic*, volume 1414 of *Lecture Notes in Computer Science*, pages 157–176. Springer-Verlag, 1997. 33, 113, 141

[23] 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. 85

[24] N. G. de Bruijn. Lambda calculus notation with nameless dummies. a tool for automatic formula manipulation with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972. 36

[25] P. de Groote. Linear higher-order matching is NP-complete. In L. Bachmair, editor, *11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 127–140, 2000. 33

[26] O. de Moor and G. Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 116–149. Springer-Verlag, 1998. 120, 162

[27] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. In A. Middledorp, editor, *Proceedings of the 4th*

Fuji International Symposium on Functional and Logic Programming, volume 1722 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, 1999. Extended Abstract. Available from URL: `http://www.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`. 63

[28] O. de Moor and G. Sittampalam. Higher-order matching for program transformation. *Theoretical Computer Science*, 269:135–162, 2001. Available from URL: `http://www.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`. 63

[29] G. Dowek. L'indécidabilité du filtrage du troisième ordre dans les calculs avec types dépendants ou constructeurs de types. *Comptes rendus à l'Académie des Sciences I*, 312:951–956, 1991. Erratum, ibid. 318:873, 1994. 33

[30] G. Dowek. A second-order pattern matching algorithm for the cube of typed lambda calculi. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 151–160. Springer-Verlag, 1991. 33, 85

[31] G. Dowek. Third order matching is decidable. In M. Nielsen and W. Thomas, editors, *Logic in Computer Science*, pages 2–10. IEEE, 1992. 33

[32] G. Dowek. The undecidability of pattern matching in calculi where primitive recursive functions are representable. *Theoretical Computer Science*, 107(2):349–356, 1993. Note. 33

[33] G. Dowek, G. Huet, and B. Werner. On the definition of the eta-long normal form in type systems of the cube. In H. Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs, Nijmegen, The Netherlands*, 1993. Available from URL: `http://pauillac.inria.fr/~dowek/Publi/eta.ps.gz`. 35

[34] L. Fegaras, T. Sheard, and T. Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1994. Available from URL: `ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Sestoft/pepm94-proceedings/fegaras.ps.gz`. 24

[35] F. Gécseg and M. Steinby. Tree automata. Akadémiai Kiadó, Budapest, 1984. 113

[36] N. Ghani. $\beta\eta$-equality for coproducts. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902, pages 171–185, 1995. 35

[37] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, 1993. 23, 28, 29

[38] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987. 33

[39] W. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981. 32

[40] C. M. Hoffman. *Group-theoretic algorithms and graph isomorphism.* Number 136 in Lecture Notes in Computer Science. Springer-Verlag, 1982. 31

[41] C. M. Hoffman and M. J. O'Donnell. Pattern matching in trees. *Journal of the Association for Computing Machinery*, 29(1):68–95, 1982. 31

[42] B. Hoffmann and D. Plump. Implementing term rewriting by jungle evaluation. *R. A. I. R. O. Informatique Theorique et Applications/Theoretical Informatics and Applications*, 25, 1991. 164

[43] G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–267, 1973. 32

[44] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975. 161

[45] G. Huet. Résolution d'équations dans les langages d'ordre 1,2,...,ω. Thése doctorat d'état, Université Paris VII, Paris, France, 1976. 32

[46] G. Huet. A complete proof of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 23:11–21, 1981. 121

[47] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978. 33, 63, 79

[48] R. J. M. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989. 7

[49] J. Jeuring. Generic Haskell project proposal. Available from URL: `http://www.cs.ruu.nl/research/projects/generic-haskell/gh.ps`. 163

[50] J. Jeuring. Polytypic pattern matching. In *Functional Programming Languages and Computer Architecture*, pages 238–248. ACM Press, 1995. 164

[51] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, number 1129 in Lecture Notes in Computer Science, pages 68–114. Springer-Verlag, 1996. 163

[52] P. Johann and E. Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. *Annals of Mathematics and Artificial Intelligence*, 39:1–34, 2000. 24

[53] M. P. Jones. Typing Haskell in Haskell. Avaiable from URL: `http://www.cse.ogi.edu/~mpj/thih/`. 161

[54] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993. 25

[55] J. Ketonen. EKL - a mathematically oriented proof checker. In *Conference on Automated Deduction*, number 170 in Lecture Notes in Computer Science, pages 65–79. Springer-Verlag, 1984. 33

[56] C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J. Lassez, and V. Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, 1993. MIT Press. 160, 163

[57] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computation*, 6(1):323–350, 1977. 31

[58] B. Krieg-Brückner, J. Liu, H. Shi, and B. Wolff. Towards correct, efficient and reusable transformational developments. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 1995. 63

[59] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Functional Programming Languages and Computer Architecture*, pages 314–323. ACM Press, 1995. 24

[60] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1995. 126

[61] G. Malcolm. Homomorphisms and promotability. In J. van der Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, 1989. 14

[62] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990. 14

[63] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991. 14, 24

[64] D. Miller. An extension to ML to handle bound variables in data structures. In *Proceedings of the Logical Frameworks BRA Workshop*, 1990. Available from URL: `ftp://ftp.cis.upenn.edu/pub/papers/miller/mll.pdf`. 126

[65] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:479–536, 1991. 33, 88

[66] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 29, 120

[67] G. Nadathur. The metalanguage λProlog and its implementation. In H. Kuchen and K. Ueda, editors, *Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2001. 162

[68] L. Németh. *Catamorphism-Based Program Transformations for Non-Strict Functional Languages*. Ph.D. thesis, University of Glasgow, 2000. 24

[69] T. Nipkow. Higher-order unification, polymorphism, and subsorts. In S. Kaplan and M. Okada, editors, *Proc. 2nd International Workshop on Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 436–447. Springer-Verlag, 1990. 33

[70] T. Nipkow. Functional unification of higher-order patterns. In *8th IEEE Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993. 34

[71] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In R. S. Bird and L. Meertens, editors, *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997. 24

[72] V. Padovani. Filtrage d'ordre supérieure. Thése doctorat d'état, Université Paris VII, Paris, France, 1996. 33

[73] S. Panne, S. Marlow, and N. Winstanley. hsparser: The 100% pure Haskell parser. Available from URL: `http://www.pms.informatik.uni-muenchen.de/mitarbeiter/panne/haskell_libs/hsparser.html`. 161

[74] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986. 162

[75] L. C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 415–475. Oxford University Press, 1992. 162

[76] S. L. Peyton Jones and J. Hughes, editors. *The Haskell 98 Report*, 1999. Available from URL: `http://www.haskell.org/definition/`. 124

[77] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. Submitted to ICFP 2001. Available from URL: `http://www.research.microsoft.com/users/simonpj/papers/rules.ps.gz`. 30

[78] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988. 162

[79] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proceedings of Fifth International Conference on Mathematics of Program Construction (MPC2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, 2000. 126

[80] I. Sanabria-Piretti. *Data refinement by transformation.* D.Phil. thesis, Oxford University, 2001. In preparation. 159

[81] A. Schubert. Linear interpolation for the higher-order matching problem. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 441–452. Springer-Verlag, 1996. 33

[82] T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming and Computer Architecture*, pages 233–242. ACM Press, New York, 1993. 24

[83] C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1. Available from URL `http://www.research.microsoft.com/research/ip/`, 1996. 158

[84] G. Sittampalam and O. de Moor. Higher-order pattern matching for automatically applying fusion transformations. In O. Danvy and A. Filinski, editors, *Proceedings of 2nd Symposium on Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 198–217. Springer-Verlag, 2001. Available from URL: `http://www.comlab.ox.ac.uk/oucl/work/oege.demoor/pubs.htm`. 87

[85] M. Sørenson. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Trees in Algebra and Programming – CAAP '94*, number 787 in Lecture Notes in Computer Science, pages 335–351. Springer-Verlag, 1994. 23

[86] J. Springintveld. Third-order matching in the polymorphic lambda calculus. In G. Dowek, J. Heering, M. K., and B. Möller, editors, *Higher-Order Algebra, Logic and Term Rewriting*, volume 1074 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995. 33

[87] J. Springintveld. Third-order matching in the presence of type constructors. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 428–442. Springer-Verlag, 1995. 33

[88] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 306–316. ACM Press, 1995. 24

[89] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook on Theoretical Computer Science, Vol. A*, pages 133–191. Elsevier, 1990. 113

[90] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986. 25

[91] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, 1999. Springer-Verlag. 160

[92] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht. Available from URL: `http://www.cs.uu.nl/~visser/ftp/Vis2000.ps.gz`. 160

[93] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, number 2051 in Lecture Notes in Computer Science, pages 357–362. Springer-Verlag, 2001. 24

[94] P. Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time. In *ACM Symposium on Lisp and Functional Programming*, 1984. 23

[95] P. Wadler. Listlessness is better than laziness II: composing listless functions. In *Workshop on Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, 1985. Springer-Verlag. 23

[96] P. Wadler. List comprehensions. In S. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 7. Prentice-Hall International, 1987. 125

[97] P. Wadler. The concatenate vanishes. Technical report, University of Glasgow, 1989. Available from URL: `http://cm.bell-labs.com/who/wadler/papers/vanish/vanish.pdf`. 8

[98] P. Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989. 29

[99] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. 23

[100] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. (Special issue of selected papers from 6th Conference on Lisp and Functional Programming.). 126

[101] P. Wadler. The essence of functional programming. In *19th ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992. 126

[102] D. A. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1993. 32, 141

Index

Appendix A

Quick reference

A.1 Definitions and notation

- Expressions are built recursively from variables, constants, applications and λ-abstractions
- Variables are either pattern (free) or local (bound).
- a, b, c denote constants.
- p, q, r denote pattern variables.
- x, y, z denote local variables.
- Capital letters denote arbitrary expressions.
- Application is written with a space, so $F\ E$ denotes F applied to E.
- λ-abstraction is written: $\lambda x.B$
- $=$ is equality modulo α-conversion.
- $\simeq$ is equality modulo $\alpha\eta$-conversion.
- An expression is closed if it does not contain any pattern variables.
- A β-redex is an expression of the form $(\lambda x.B)\ E$, with β-contractum $(x := B)E$.
- A η-redex is an expression of the form $\lambda x.F\ x$ with x not occurring in F. Its η-contractum is F.

- An expression is β-normal if it contains no β-redexes, η-normal if it contains no η-redexes, and $\beta\eta$-normal, or just normal, if it contains neither.
- $\xrightarrow{\beta}$ is the relation denoting reduction of a single β-redex in an expression, with reflexive transitive closure $\xrightarrow{\beta}^*$.
- $S \trianglelefteq E$ states that S is a subexpression of E.
- $S \ntrianglelefteq E$ states that S is not a subexpression of E, or equivalently that S does not occur in E.
- A direction is either *Func*, *Arg* or *Body* to denote the function and argument parts of an application and the body of a λ-abstraction respectively.
- A location is a sequence of directions denoting a position in a term; $\langle\rangle$ denotes the root of the term, and $\frown$ is the operator which joins two locations one after another. *dir*; *loc* is shorthand for $\langle dir\rangle^\frown loc$.
- The substitution $(p := \lambda x.x,\, q := \lambda y.y)$ makes the indicated assigments to p and q and leaves all other variables unchanged.
- $(\phi \circ \psi)E = \phi\,(\psi E)$
- $\phi \leq \psi \equiv \exists\delta \,:\, \delta \circ \phi = \psi$
- $P \rightarrow T$ is a rule with pattern P and term T.
- *Xs* $\ll$ *Ys* indicates that the measure of *Xs* is strictly less than the measure of *Ys* in a lexicographic comparison.
- $\phi \vdash_{app} P \rightarrow T \equiv \mathit{reduce}\ \mathit{app}\ (\phi P) \simeq T$

A.2 Specification

A.2.1 Beta-reduction

$$\begin{array}{rcl}
\mathit{reduce\ app}\ c & = & c \\
\mathit{reduce\ app}\ x & = & x \\
\mathit{reduce\ app}\ p & = & p \\
\mathit{reduce\ app}\ (\lambda x.E) & = & \lambda x.(\mathit{reduce\ app}\ E) \\
\mathit{reduce\ app}\ (E_1\, E_2) & = & \mathit{app}\ (\mathit{reduce\ app}\ E_1)\ (\mathit{reduce\ app}\ E_2)
\end{array}$$

$$\begin{aligned}
\mathit{reduce}'\,\mathit{app}\,c &= c\\
\mathit{reduce}'\,\mathit{app}\,x &= x\\
\mathit{reduce}'\,\mathit{app}\,p &= p\\
\mathit{reduce}'\,\mathit{app}\,(\lambda x.E) &= \mathit{etared}\,(\lambda x.(\mathit{reduce}'\,\mathit{app}\,E))\\
\mathit{reduce}'\,\mathit{app}\,(E_1\,E_2) &= \mathit{etanormalise}\\
&\qquad(\mathit{app}\,(\mathit{reduce}'\,\mathit{app}\,E_1)\,(\mathit{reduce}'\,\mathit{app}\,E_2))
\end{aligned}$$

$$\begin{aligned}
\mathit{full}\,(\lambda x.B)\,E &= \mathit{reduce}\,\mathit{full}\,((x := E)B)\\
\mathit{full}\,F\,E &= F\,E\\
\mathit{none}\,F\,E &= F\,E
\end{aligned}$$

$$\mathit{step} = \mathit{reduce}\,\mathit{once}$$

$$\begin{aligned}
\mathit{once}\,(\lambda x.B)\,E &= (x := E)B\\
\mathit{once}\,F\,E &= F\,E
\end{aligned}$$

$$\begin{aligned}
\mathit{twostep} &= \mathit{reduce}\,\mathit{twice}\\
\mathit{markedstep} &= \mathit{reduce}\,\mathit{markedstep}
\end{aligned}$$

$$\begin{aligned}
\mathit{twice}\,(\lambda x.B)\,E &= \mathit{unmark}\,(\mathit{reduce}\,\mathit{markedonce}\,((x := \mathit{mark}\,E)B))\\
\mathit{twice}\,E_1\,E_2 &= E_1\,E_2
\end{aligned}$$

$$\begin{aligned}
\mathit{mark}\,c &= c\\
\mathit{mark}\,x &= x\\
\mathit{mark}\,p &= p\\
\mathit{mark}\,(\lambda x.E) &= \lambda' x.(\mathit{mark}\,E)\\
\mathit{mark}\,(E_1\,E_2) &= E_1\,E_2
\end{aligned}$$

$$\begin{aligned}
\mathit{markedonce}\,(\lambda' x.B)\,E &= (x := E)B\\
\mathit{markedonce}\,E_1\,E_2 &= E_1\,E_2
\end{aligned}$$

$$\begin{aligned}
\mathit{unmark}\,c &= c\\
\mathit{unmark}\,x &= x\\
\mathit{unmark}\,p &= p\\
\mathit{unmark}\,(\lambda x.E) &= \lambda x.(\mathit{unmark}\,E)\\
\mathit{unmark}\,(\lambda' x.E) &= \lambda x.(\mathit{unmark}\,E)\\
\mathit{unmark}\,(E_1\,E_2) &= (\mathit{unmark}\,E_1)\,(\mathit{unmark}\,E_2)
\end{aligned}$$

A.2.2 Matching

$\mathcal{M}$ is a match set with respect to *app* of *Xs* if:

- For all ϕ: $\phi \vdash_{app} Xs$ iff there exists $\psi \in \mathcal{M}$ such that $\psi \leq \phi$.
- For all $\phi_1, \phi_2 \in \mathcal{M}$: if $\phi_1 \leq \phi_2$, then $\phi_1 = \phi_2$.

A.2.3 *resolve*

For a particular *app* function, suppose that

$$\mathit{resolve}\ \mathit{app}\ X \quad = \quad [\![(\sigma_0, Ys_0), (\sigma_1, Ys_1), \ldots, (\sigma_k, Ys_k)]\!]$$

We require that

(1) For all substitutions ϕ:

$$(\phi \vdash_{app} X) \quad \equiv \quad \bigvee_i (\phi \vdash_{app} Ys_i \wedge \sigma_i \leq \phi)$$

(2) For all substitutions ϕ and indices i and j:

$$(\phi \vdash_{app} Ys_i) \wedge (\phi \vdash_{app} Ys_j) \quad \Rightarrow \quad i = j$$

(3) For each index i, σ_i is pertinent to X.

(4) The pattern variables in Ys_i are contained in the pattern variables of X.

(5) For each index i:

$$Ys_i \ll X$$

A.2.4 *appresolve*

Suppose that $\mathit{appresolve}\ \mathit{app}\ F\ E\ T = [\![\ Ys_1, ..., Ys_k\ ]\!]$. Then we require that:

{1} For all substitutions ϕ:

$$\mathit{etanormalise}\ (\mathit{app}\ (\mathit{reduce}'\ \mathit{app}\ (\phi F))\ (\mathit{reduce}'\ \mathit{app}\ (\phi E))) = T$$
$$\equiv$$
$$\bigvee_i \phi \vdash_{app} Ys_i$$

{2} For all substitutions ϕ:

$$(\phi \vdash_{app} Ys_i) \wedge (\phi \vdash_{app} Ys_j) \quad \Rightarrow \quad i = j\ .$$

{3} The pattern variables in Ys_i are contained in the pattern variables of $F\,E$.

{4} For each index i:

$$Ys_i \ll (F\,E \rightarrow T)\ .$$

A.3 Implementation

A.3.1 *matches*

$$\begin{array}{rcl}
\mathit{matches\ app} & :: & [\![\,\mathit{Rule}\,]\!] \rightarrow [\![\,\mathit{Subst}\,]\!] \\
\mathit{matches\ app}\,[\![\,]\!] & = & [\![\,\mathit{idSubst}\,]\!] \\
\mathit{matches\ app}\,([\![X]\!] + \mathit{Xs}) & = & [\![\ (\phi \circ \sigma) \mid\ (\sigma, \mathit{Ys}) \in \mathit{resolve\ app}\,X, \\
& & \qquad\qquad \phi \in \mathit{matches\ app}\,(\sigma\,(\mathit{Xs} + \mathit{Ys}))\ ]\!]
\end{array}$$

A.3.2 *resolve*

X	$\mathit{resolve\ app}\ X$
$x \rightarrow y$	$[\![\,(\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $x = y$ $[\![\,]\!]$, otherwise
$a \rightarrow b$	$[\![\,(\mathit{idSubst}, [\![\,]\!])\,]\!]$, if $a = b$ $[\![\,]\!]$, otherwise
$p \rightarrow T$	$[\![\,(p := T, [\![\,]\!])\,]\!]$, if T does not contain unbound local variables $[\![\,]\!]$, otherwise
$(\lambda x.P) \rightarrow (\lambda x.T)$	$[\![\,(\mathit{idSubst}, [\![\,P \rightarrow T\,]\!])\,]\!]$
$(\lambda x.P) \rightarrow T$	$[\![\,(\mathit{idSubst}, [\![\,P \rightarrow (T\,x)\,]\!])\,]\!]$
$(F\,E) \rightarrow T$	$[\![\,(\mathit{idSubst}, \mathit{Ys}) \mid \mathit{Ys} \in \mathit{appresolve\ app}\ F\ E\ T\,]\!]$
$P \rightarrow T$	$[\![\,]\!]$

A.3.3 *appresolve none*

$$\begin{array}{rcl}
\mathit{appresolve\ none}\ F\ E\ (T_0\ T_1) & = & [\![\,[\![\,F \rightarrow T_0, E \rightarrow T_1\,]\!]\,]\!] \\
\mathit{appresolve\ none}\ F\ E\ T & = & [\![\,]\!],\ \text{if}\ T \neq T_0\ T_1
\end{array}$$

A.3.4 *appresolve once*

$$\begin{array}{rcl}
\mathit{appresolve\ once}\ F\ E\ T & = & [\![\,[\![\,(F \to T_0), (E \to T_1)\,]\!] \mid (T_0\ T_1) = T\,]\!] \\
& + & [\![\,[\![\,(F \to T_0), (E \to T_1)\,]\!] \mid (T_0, T_1) \leftarrow \mathit{apps}\ T\,]\!] \\
& + & [\![\,[\![\,F \to (\lambda x.T)\,]\!] \mid\ x \text{ fresh}\,]\!]
\end{array}$$

$$\begin{array}{rcll}
\mathit{apps}\ T & = & [\![\,(\lambda x.B, S) \mid & (S, \mathit{locs}) \leftarrow \mathit{collect}\,(\mathit{subexps}\ T), \\
&&& \mathit{unboundlocals}\ S \subseteq \mathit{unboundlocals}\ T, \\
&&& \mathit{locs}' \subseteq \mathit{locs}, \\
&&& \mathit{locs}' \neq \{\,\}, \\
&&& B = \mathit{replaces}\ T\ \mathit{locs}'\ x, \\
&&& \lambda x.B \text{ normal}, \\
&&& x \text{ fresh} \qquad\qquad]\!]
\end{array}$$

A.3.5 *appresolve twice*

$$\begin{array}{rcl}
\mathit{appresolve\ twice}\ F\ E\ T & = & [\![\,[\![\,F \to \mathit{etaRed}(\lambda x.B)\,]\!] \mid \\
&& \quad x \text{ fresh}, B \leftarrow \mathit{abstracts}\ x\ E\ T\,]\!] \\
&& \text{if } F \text{ flexible} \\
\mathit{appresolve\ twice}\ F\ E\ (T_1\ T_2) & = & [\![\,[\![\,F \to T_1, E \to T_2\,]\!]\,]\!] \\
&& \text{if } F \text{ not flexible} \\
\mathit{appresolve\ twice}\ F\ E\ T & = & [\![\,]\!]
\end{array}$$

$$\mathit{abstracts}\ x\ E\ T \;=\; \cup\{\,\mathit{abstracts}_n\ x\ E\ T \mid n = 0 \ldots\}$$

$$\begin{array}{rcll}
\mathit{abstracts}_0\ x\ E\ T & = & \{\,T\,\} & \\
\mathit{abstracts}_{(n+1)}\ x\ E\ T & = & \{\,(y := x)C \mid & B \in \mathit{abstracts}_n\ x\ E\ T, \\
&&& C \in \mathit{abstract}\ x\ y\ E\ B, \\
&&& y \text{ fresh}\,\}
\end{array}$$

$$\begin{array}{rcll}
\mathit{abstract}\ x\ y\ E\ T & = & \{\,\mathit{etanormalise}\,(\mathit{replace}\ \mathit{loc}\ R\ T) & \\
&& \quad \mid & (S, \mathit{loc}) \in \mathit{subexps}\ T \\
&&& x \ntrianglelefteq S \\
&&& R \in \mathit{instance}\ y\ E\ S \qquad \}
\end{array}$$

$$\begin{array}{rcll}
\mathit{instance}\ y\ E\ S & = & \{\,y\,(\phi\,x_1) \ldots (\phi\,x_m) & \\
&& \quad \mid & (x_1, \ldots, x_m) = \mathit{params}\ E \\
&&& \phi \in \mathit{matches}\ \mathit{none}\ [\![\,\mathit{body}\ E \to S\,]\!] \quad \}
\end{array}$$

Appendix B

MAG derivations

B.1 Minimum depth

Theory file

```
{- mindepth.eq -}

md: md t d m = min (mindepth (foldbtree Bin Leaf t) + d) m;

plusunit: 0+a = a;
plusassoc: (a+b)+c = a+(b+c);

minassoc: min (min a b) c = min a (min b c);

cutmin: min (min mq mr + s) c
        =
        if s>=c
        then c
        else min (min (mq +s) (mr+s)) c;

mindepth0: mindepth (Leaf a) = 0;
mindepth1: mindepth (Bin x y) = min (mindepth x) (mindepth y) + 1;

treefusion: h (foldbtree plus f t) = foldbtree times g t,
            if { \b -> h (f b) = \b -> g b;
                 \x y -> h (plus x y) = \x y -> times (h x) (h y) }
```

Derivation

```
   md
= { md }
   (\ a b -> min (mindepth (foldbtree Bin Leaf a) + b))
= { treefusion

      (\ a b -> min (mindepth (Leaf a) + b))
   = { mindepth0 }
      (\ a b -> min (0 + b))
   = { plusunit }
      (\ a -> min)

      (\ a b c -> min (mindepth (Bin a b) + c))
   = { mindepth1 }
      (\ a b c -> min ((min (mindepth a) (mindepth b) + 1) + c))
   = { plusassoc }
      (\ a b c -> min (min (mindepth a) (mindepth b) + (1 + c)))
   = { cutmin }
      (\ a b c ->
       (\ d ->
        if e >= d
         then d
         else min (min (mindepth a + e) (mindepth b + e)) d

       )
       where e = 1 + c
      )
   = { minassoc }
      (\ a b c ->
       (\ d ->
        if e >= d  then d
                   else min (mindepth a + e)
                            (min (mindepth b + e) d)

       )
       where e = 1 + c
      )
  }
   foldbtree (\ d e f ->
               (\ g -> if a >= g  then g else d a (e a g))
               where a = 1 + f
              )
              (\ h -> min)
```

B.2 Alpha-beta pruning

Theory file

```
fastflipeval: fastflipeval t a b
            = bound (flipeval (idrtreefold t)) a b;

idrtreefold: idrtreefold = rtreefold RLeaf RNode (:) [];

{- bound: bound x a b = min (max a x) b; -}

flipeval0: flipeval (RLeaf x) = x;
flipeval1: flipeval (RNode ts)
         = listmax (map (negate.flipeval) ts);

listmax0: listmax [] = neginf;
listmax1: listmax (x:xs) = max x (listmax xs);

map0: map f [] = [];
map1: map f (x:xs) = f x : map f xs;

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

boundmax: bound (max c d) a b
        = let a'=bound c a b in
           if a'==b then b
           else bound d a' b;

negbound: bound (negate c) a b
        = negate (bound c (negate b) (negate a));

boundneginf: bound neginf a b = a;

promotion:
   f (rtreefold leaf node cons nil t)
    = rtreefold leaf' node' cons' nil' t,
  if {
        \x -> f (leaf x) = \x -> leaf' x;
        f' nil = nil';
        \x y -> f' (cons x y) = \x y -> cons' (f x) (f' y);
        \x -> f (node x) = \x -> node' (f' x)
```

```
     }
```

Derivation

```
   fastflipeval
= { fastflipeval }
   (\ a -> bound (flipeval (idrtreefold a)))
= { idrtreefold }
   (\ a -> bound (flipeval (rtreefold RLeaf RNode (:) [] a)))
= { promotion

      (\ a -> bound (flipeval (RLeaf a)))
   = { flipeval0 }
      bound

      bound (listmax (map (\ c -> negate (flipeval c)) []))
   = { map0 }
      bound (listmax [])
   = { listmax0 }
      bound neginf
   = { boundneginf }
      (\ a b -> a)

      (\ a b ->
       bound (listmax (map (\ e -> negate (flipeval e))
                           (a : b )))
      )
   = { map1 }
      (\ a b ->
       bound (listmax (negate (flipeval a)
                       :
                       map (\ e -> negate (flipeval e)) b))
      )
   = { listmax1 }
      (\ a b ->
       bound (max (negate (flipeval a))
                  (listmax (map (\ e -> negate (flipeval e)) b)))
      )
   = { boundmax }
      (\ a ->
       (\ b c d ->
        (if g == d
          then d
          else bound (listmax (map (\ e -> negate (flipeval e))
                                   b))
                     g
                     d
         )
        where g = bound f c d
       )
       where f = negate (flipeval a)
      )
   = { negbound }
      (\ a ->
       (\ b c d ->
        if bound (negate f) c d == d
         then d
         else bound (listmax (map (\ e -> negate (flipeval e))
                                  b))
                    (negate (bound f (negate d) (negate c)))
                    d

       )
       where f = flipeval a
      )
   = { negbound }
      (\ a ->
       (\ b c ->
        (\ d ->
         (if h == d
           then d
           else bound (listmax (map (\ e -> negate (flipeval e))
                                    b))
                      h
                      d
          )
         where h = negate (bound f (negate d) g)
        )
        where g = negate c
       )
       where f = flipeval a
      )

      (\ a -> bound (flipeval (RNode a)))
   = { flipeval1 }
      (\ a -> bound (listmax (map (negate . flipeval) a)))
   = { compose }
      (\ a ->
       bound (listmax (map (\ d -> negate (flipeval d)) a))
      )
  }
   rtreefold bound
             (\ g -> g)
             (\ j k l ->
              (\ m ->
               (if b == m  then m else k b m)
               where b = negate (j (negate m) a)
              )
              where a = negate l
             )
             (\ n o -> n)
```

B.3 Steep sequences

Theory file

```
{- steep.eq -}

faststeep: faststeep x = split steep sum (foldr (:) [] x);

steep0: steep [] = True;
steep1: steep (a:x) = a > sum x && steep x;

sum0: sum [] = 0;
sum1: sum (a:x) = a + sum x;

tupling: split f g (foldr step e x) =
                 foldr (\ a -> uncurry (h a)) c x,
            if {\ a x -> split f g (step a x)  =
                  \ a x -> h a (f x) (g x);
                split f g e = c};

split: split f g x = (f x, g x)
```

Derivation

```
   faststeep
= { faststeep }
   (\ a -> split steep sum (foldr (:) [] a))
= { tupling

      (\ a b -> split steep sum (a : b ))
   = { split }
      (\ a b ->
       ( steep c, sum c )
       where c = a : b
      )
   = { steep1 }
      (\ a b -> ( a > sum b && steep b, sum (a : b ) ))
   = { sum1 }
      (\ a b ->
       ( a > c  && steep b, a + c  )
       where c = sum b
      )

      split steep sum []
   = { split }
      ( steep [], sum [] )
   = { steep0 }
      ( True, sum [] )
   = { sum0 }
      ( True, 0 )
  }
   foldr (\ b -> uncurry (\ d e -> ( b > e  && d, b + e  )))
         ( True, 0 )
```

B.4 Path sequence problem

B.4.1 Initial calculations with *llp*

Theory file

```
{- llp0.eq -}

llp: llp xs = listmax (map length (filter path (subs xs)));
```

```
map0: map f [] = [];
map1: map f (x:xs) = f x : map f xs;

filter0: filter p [] = [];
filter1: filter p (x:xs) = if p x then x:filter p xs
                           else filter p xs;

subs0: subs []=[[]];
subs1: subs (x:xs) = subs xs ++ map (\ys -> x:ys) (subs xs);

path0: path [] = True;
path1: path [x] = True;
path2: path (x:y:xs) = arc x y && path (y:xs);

length0: length []=0;
length1: length (x:xs) = 1+length xs;

listmax0: max x (listmax []) = x;
listmax1: listmax (x:xs) = max x (listmax xs);

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

filtercat: filter p (xs++ys) = filter p xs ++ filter p ys;
iftrue: if True then x else y = x;
filtermap: filter p (map f xs) = map f (filter (p.f) xs);
mapcat: map f (xs++ys) = map f xs ++ map f ys;
listmaxcat: listmax (xs++ys) = max (listmax xs) (listmax ys);
mapmap: map f (map g xs) = map (f.g) xs;
listmaxplusone: listmax (map (\x -> 1 + f x) xs)
              = 1+listmax (map f xs)
```

Derivation for empty list

```
   llp []
= { llp }
   listmax (map length (filter path (subs [])))
= { subs0 }
   listmax (map length (filter path ([] : [])))
= { filter1 }
   listmax (map length (if path [] then [] : a else a))
   where a = filter path []
= { filter0 }
```

```
   listmax (map length
                (if path [] then [] : filter path [] else []))
= { filter0 }
   listmax (map length (if path [] then [] : [] else []))
= { path0 }
   listmax (map length (if True then [] : [] else []))
= { iftrue }
   listmax (map length ([] : []))
= { map1 }
   listmax (length [] : map length [])
= { map0 }
   listmax (length [] : [])
= { length0 }
   listmax (0 : [])
= { listmax1 }
   max 0 (listmax [])
= { listmax0 }
   0
```

Derivation for non-empty list

```
   (\ a b -> llp (a : b ))
= { llp }
   (\ a b -> listmax (map length (filter path (subs (a : b )))))
= { subs1 }
   (\ a b ->
    listmax (map length (filter path (c ++ map ((:) a) c)))
    where c = subs b
   )
= { filtercat }
   (\ a b ->
    listmax (map length
                 (filter path c ++ filter path (map ((:) a) c)))
    where c = subs b
   )
= { filtermap }
   (\ a b ->
    listmax (map length (filter path c
                         ++
                         map ((:) a) (filter (path . (:) a) c)))
    where c = subs b
   )
```

```
= { compose }
   (\ a b ->
    listmax (map length (filter path c
                         ++
                         map ((:) a)
                             (filter (\ f -> path (a : f )) c)))
    where c = subs b
   )
= { mapcat }
   (\ a b ->
    listmax (map length (filter path c)
             ++
             map length (map ((:) a)
                             (filter (\ g -> path (a : g )) c)))
    where c = subs b
   )
= { listmaxcat }
   (\ a b ->
    max (listmax (map length (filter path c)))
        (listmax (map length
                      (map ((:) a)
                           (filter (\ g -> path (a : g )) c))))
    where c = subs b
   )
= { mapmap }
   (\ a b ->
    max (listmax (map length (filter path c)))
        (listmax (map (length . (:) a)
                      (filter (\ h -> path (a : h )) c)))
    where c = subs b
   )
= { compose }
   (\ a b ->
    max (listmax (map length (filter path c)))
        (listmax (map (\ e -> length (a : e ))
                      (filter (\ f -> path (a : f )) c)))
    where c = subs b
   )
= { length1 }
   (\ a b ->
    max (listmax (map length (filter path c)))
        (listmax (map (\ e -> 1 + length e)
                      (filter (\ f -> path (a : f )) c)))
    where c = subs b
   )
= { listmaxplusone }
   (\ a b ->
    max (listmax (map length (filter path c)))
        (1 + listmax (map length
                           (filter (\ f -> path (a : f )) c)))
    where c = subs b
   )
```

B.4.2 Calculation of *fastllp*

Theory file

```
{- llp1.eq -}

fastllp: fastllp xs = split llp (\xs x -> llp' x xs)
                                   (foldr (:) [] xs);

llp: llp xs = listmax (map length (filter path (subs xs)));

llp': llp' x xs = listmax (map length (filter (\ys -> path (x:ys))
                      (subs xs)));

map0: map f [] = [];
map1: map f (x:xs) = f x : map f xs;

filter0: filter p [] = [];
filter1: filter p (x:xs) = if p x then x:filter p xs
                           else filter p xs;

subs0: subs []=[[]];
subs1: subs (x:xs) = subs xs ++ map (\ys -> x:ys) (subs xs);

path0: path [] = True;
path1: path [x] = True;
path2: path (x:y:xs) = arc x y && path (y:xs);

length0: length []=0;
length1: length (x:xs) = 1+length xs;

listmax0: max x (listmax []) = x;
```

```
listmax1: listmax (x:xs) = max x (listmax xs);

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

filtercat: filter p (xs++ys) = filter p xs ++ filter p ys;
iftrue: if True then x else y = x;
filtermap: filter p (map f xs) = map f (filter (p.f) xs);
mapcat: map f (xs++ys) = map f xs ++ map f ys;
listmaxcat: listmax (xs++ys) = max (listmax xs) (listmax ys);
filterand: filter (\x -> p x && q x) xs = filter p (filter q xs);
filterconst: filter (\x -> c) xs = if c then xs else [];
funcif: f (if c then x else y) = if c then f x else f y;
mapmap: map f (map g xs) = map (f.g) xs;
listmaxplusone: listmax (map (\x -> 1 + f x) xs)
              = 1+listmax (map f xs);

tupling: split f g (foldr step e x) =
                 foldr (\ a -> uncurry (h a)) c x,
            if {\ a x -> split f g (step a x)  =
                  \ a x -> h a (f x) (g x);
                split f g e = c};

split: split f g x = (f x, g x)
```

Derivation

```
   fastllp
= { fastllp }
   (\ a -> split llp (\ c d -> llp' d c) (foldr (:) [] a))
= { llp }
   (\ a ->
    split (\ b -> listmax (map length (filter path (subs b))))
          (\ e f -> llp' f e)
          (foldr (:) [] a)
   )
= { llp' }
   (\ a ->
    split (\ b -> listmax (map length (filter path (subs b))))
          (\ e f ->
           listmax (map length (filter (\ h -> path (f : h ))
                                        (subs e)))
          )
          (foldr (:) [] a)
   )
= { tupling

      (\ a b ->
       split (\ c -> listmax (map length (filter path (subs c))))
             (\ f g ->
              listmax (map length (filter (\ i -> path (g : i ))
                                          (subs f)))
             )
             (a : b )
      )
   = { split }
      (\ a b ->
       ( listmax (map length (filter path c))
       , (\ e ->
          listmax (map length (filter (\ g -> path (e : g )) c))
         )
       )
       where c = subs (a : b )
      )
   = { subs1 }
      (\ a b ->
       ( listmax (map length (filter path (subs (a : b ))))
       , (\ e ->
          listmax (map length (filter (\ g -> path (e : g ))
                                      (c ++ map ((:) a) c)))
         )
       )
       where c = subs b
      )
   = { subs1 }
      (\ a b ->
       ( listmax (map length (filter path d))
       , (\ f ->
          listmax (map length (filter (\ h -> path (f : h )) d))
         )
       )
       where {
             c = subs b;
             d = c ++ map ((:) a) c
             }
      )
```

```
= { filtercat }
   (\ a b ->
    ( listmax (map length (filter path (c ++ d )))
    , (\ f ->
       listmax (map length (filter (\ h -> path (f : h )) c
                            ++
                            filter (\ i -> path (f : i )) d))
      )
    )
    where {
          c = subs b;
          d = map ((:) a) c
          }
   )
= { filtercat }
   (\ a b ->
    ( listmax (map length (filter path c ++ filter path d))
    , (\ g ->
       listmax (map length (filter (\ i -> path (g : i )) c
                            ++
                            filter (\ j -> path (g : j )) d))
      )
    )
    where {
          c = subs b;
          d = map ((:) a) c
          }
   )
= { filtermap }
   (\ a b ->
    ( listmax (map length (filter path c
                           ++
                           filter path (map ((:) a) c)))
    , (\ g ->
       listmax (map length
                     (filter (\ i -> path (g : i )) c
                      ++
                      map ((:) a)
                          (filter ((\ l -> path (g : l ))
                                   .
                                   (:) a)
                                  c)))
      )
    )
    where c = subs b
   )
= { compose }
   (\ a b ->
    ( listmax (map length (filter path c
                           ++
                           filter path (map ((:) a) c)))
    , (\ g ->
       listmax (map length
                     (filter (\ i -> path (g : i )) c
                      ++
                      map ((:) a)
                          (filter (\ k -> path (g : (a : k )))
                                  c)))
      )
    )
    where c = subs b
   )
= { path2 }
   (\ a b ->
    ( listmax (map length (filter path c
                           ++
                           filter path (map ((:) a) c)))
    , (\ g ->
       listmax (map length
                     (filter (\ i -> path (g : i )) c
                      ++
                      map ((:) a)
                          (filter (\ k ->
                                    arc g a && path (a : k )
                                  )
                                  c)))
      )
    )
    where c = subs b
   )
= { filtermap }
   (\ a b ->
    ( listmax (map length (filter path c
                           ++
                           map ((:) a)
                               (filter (path . (:) a) c)))
```

```
       , (\ i ->
          listmax (map length
                       (filter (\ k -> path (i : k )) c
                        ++
                        map ((:) a)
                            (filter (\ m ->
                                     arc i a && path (a : m )
                                    )
                                    c)))
         )
       )
       where c = subs b
      )
   = { compose }
      (\ a b ->
       ( listmax (map length
                      (filter path c
                       ++
                       map ((:) a)
                           (filter (\ f -> path (a : f )) c)))
       , (\ g ->
          listmax (map length
                       (filter (\ i -> path (g : i )) c
                        ++
                        map ((:) a)
                            (filter (\ k ->
                                     arc g a && path (a : k )
                                    )
                                    c)))
         )
       )
       where c = subs b
      )
   = { mapcat }
      (\ a b ->
       ( listmax (map length
                      (filter path c
                       ++
                       map ((:) a)
                           (filter (\ f -> path (a : f )) c)))
       , (\ g ->
          listmax (map length (filter (\ i -> path (g : i )) c)
                   ++
                   map length
                       (map ((:) a)
                            (filter (\ l ->
                                     arc g a && path (a : l )
                                    )
                                    c)))
         )
       )
       where c = subs b
      )
   = { mapcat }
      (\ a b ->
       ( listmax (map length (filter path c)
                  ++
                  map length
                      (map ((:) a)
                           (filter (\ g -> path (a : g )) c)))
       , (\ h ->
          listmax (map length (filter (\ j -> path (h : j )) c)
                   ++
                   map length
                       (map ((:) a)
                            (filter (\ m ->
                                     arc h a && path (a : m )
                                    )
                                    c)))
         )
       )
       where c = subs b
      )
   = { listmaxcat }
      (\ a b ->
       ( listmax (map length (filter path c)
                  ++
                  map length
                      (map ((:) a)
                           (filter (\ g -> path (a : g )) c)))
       , (\ h ->
          max (listmax (map length
                            (filter (\ j -> path (h : j )) c)))
              (listmax (map length (map ((:) a)
                                        (filter (\ m ->
                                                 arc h a
```

```
                                              &&
                                              path (a : m )
                                             )
                                             c))))
      )
    )
    where c = subs b
   )
= { listmaxcat }
   (\ a b ->
    ( max (listmax (map length (filter path c)))
          (listmax (map length
                        (map ((:) a)
                             (filter (\ g -> path (a : g ))
                                     c))))
    , (\ h ->
       max (listmax (map length
                         (filter (\ j -> path (h : j )) c)))
           (listmax (map length (map ((:) a)
                                     (filter (\ m ->
                                              arc h a
                                              &&
                                              path (a : m )
                                             )
                                             c))))
      )
    )
    where c = subs b
   )
= { filterand }
   (\ a b ->
    ( max (listmax (map length (filter path c)))
          (listmax (map length (map ((:) a) d)))
    , (\ h ->
       max (listmax (map length
                         (filter (\ j -> path (h : j )) c)))
           (listmax (map length
                         (map ((:) a)
                              (filter (\ m -> arc h a) d))))
      )
    )
    where {
          c = subs b;
```

```
                d = filter (\ g -> path (a : g )) c
                }
         )
   = { filterconst }
         (\ a b ->
          ( max (listmax (map length (filter path c)))
                (listmax (map length (map ((:) a) d)))
          , (\ h ->
              max (listmax (map length
                                  (filter (\ j -> path (h : j )) c)))
                  (listmax (map length
                                  (map ((:) a)
                                       (if arc h a then d else []))))
            )
          )
          where {
                c = subs b;
                d = filter (\ g -> path (a : g )) c
                }
         )
   = { funcif }
         (\ a b ->
          ( max (listmax (map length (filter path c))) d
          , (\ h ->
              (if arc h a
                then max e d
                else max e (listmax (map length (map ((:) a) [])))
               )
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
          )
          where {
                c = subs b;
                d = listmax (map length
                                 (map ((:) a)
                                      (filter (\ g -> path (a : g ))
                                              c)))
                }
         )
   = { map0 }
         (\ a b ->
          ( max (listmax (map length (filter path c))) d
          , (\ h ->
              (if arc h a then max e d
                         else max e (listmax (map length []))
               )
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
          )
          where {
                c = subs b;
                d = listmax (map length
                                 (map ((:) a)
                                      (filter (\ g -> path (a : g ))
                                              c)))
                }
         )
   = { map0 }
         (\ a b ->
          ( max (listmax (map length (filter path c))) d
          , (\ h ->
              (if arc h a then max e d else max e (listmax []))
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
          )
          where {
                c = subs b;
                d = listmax (map length
                                 (map ((:) a)
                                      (filter (\ g -> path (a : g ))
                                              c)))
                }
         )
   = { listmax0 }
         (\ a b ->
          ( max (listmax (map length (filter path c))) d
          , (\ h ->
              (if arc h a then max e d else e)
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
          )
          where {
                c = subs b;
                d = listmax (map length
                                 (map ((:) a)
                                      (filter (\ g -> path (a : g ))
                                              c)))
                }
         )
   = { mapmap }
         (\ a b ->
          ( max (listmax (map length (filter path c)))
                (listmax (map length (map ((:) a) d)))
          , (\ h ->
              (if arc h a
                then max e (listmax (map (length . (:) a) d))
                else e
               )
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
          )
          where {
                c = subs b;
                d = filter (\ g -> path (a : g )) c
                }
         )
   = { compose }
         (\ a b ->
          ( max (listmax (map length (filter path c)))
                (listmax (map length (map ((:) a) d)))
          , (\ h ->
              (if arc h a
                then max e (listmax (map (\ k -> length (a : k )) d))
                else e
               )
              where e = listmax (map length
                                     (filter (\ j -> path (h : j ))
                                             c))
            )
```

```
       )
       where {
             c = subs b;
             d = filter (\ g -> path (a : g )) c
             }
      )
   = { length1 }
      (\ a b ->
       ( max (listmax (map length (filter path c)))
             (listmax (map length (map ((:) a) d)))
       , (\ h ->
          (if arc h a
            then max e (listmax (map (\ k -> 1 + length k) d))
            else e
           )
          where e = listmax (map length
                                    (filter (\ j -> path (h : j ))
                                            c))
         )
       )
       where {
             c = subs b;
             d = filter (\ g -> path (a : g )) c
             }
      )
   = { mapmap }
      (\ a b ->
       ( max (listmax (map length (filter path c)))
             (listmax (map (length . (:) a) d))
       , (\ i ->
          (if arc i a
            then max e (listmax (map (\ l -> 1 + length l) d))
            else e
           )
          where e = listmax (map length
                                    (filter (\ k -> path (i : k ))
                                            c))
         )
       )
       where {
             c = subs b;
             d = filter (\ h -> path (a : h )) c
             }
      )
   = { compose }
      (\ a b ->
       ( max (listmax (map length (filter path c)))
             (listmax (map (\ e -> length (a : e )) d))
       , (\ g ->
          (if arc g a
            then max h (listmax (map (\ j -> 1 + length j) d))
            else h
           )
          where h = listmax (map length
                                 (filter (\ i -> path (g : i ))
                                         c))
         )
       )
       where {
             c = subs b;
             d = filter (\ f -> path (a : f )) c
             }
      )
   = { length1 }
      (\ a b ->
       ( max (listmax (map length (filter path c))) d
       , (\ g ->
          (if arc g a then max h d else h)
          where h = listmax (map length
                                 (filter (\ i -> path (g : i ))
                                         c))
         )
       )
       where {
             c = subs b;
             d = listmax (map (\ e -> 1 + length e)
                              (filter (\ f -> path (a : f )) c))
             }
      )
   = { listmaxplusone }
      (\ a b ->
       ( max (listmax (map length (filter path c)))
             (listmax (map (\ e -> 1 + length e) d))
       , (\ g ->
          (if arc g a then max h (1 + listmax (map length d))
                      else h
```

```
        )
       where h = listmax (map length
                               (filter (\ i -> path (g : i ))
                                       c))
      )
    )
    where {
          c = subs b;
          d = filter (\ f -> path (a : f )) c
          }
   )
= { listmaxplusone }
   (\ a b ->
    ( max (listmax (map length (filter path c))) d
    , (\ g ->
       (if arc g a then max e d else e)
       where e = listmax (map length
                               (filter (\ i -> path (g : i ))
                                       c))
      )
    )
    where {
          c = subs b;
          d = 1 + listmax (map length
                                (filter (\ f -> path (a : f ))
                                        c))
          }
   )

   split (\ a -> listmax (map length (filter path (subs a))))
         (\ d e ->
          listmax (map length (filter (\ g -> path (e : g ))
                                      (subs d)))
         )
         []
= { split }
   ( listmax (map length (filter path a))
   , (\ c ->
      listmax (map length (filter (\ e -> path (c : e )) a))
     )
   )
   where a = subs []
= { subs0 }
```

```
      ( listmax (map length (filter path (subs [])))
      , (\ c ->
         listmax (map length (filter (\ e -> path (c : e ))
                                     ([] : [])))
        )
      )
   = { filter1 }
      ( listmax (map length (filter path (subs [])))
      , (\ c ->
         listmax (map length
                      (if path (c : []) then [] : a else a))
         where a = filter (\ e -> path (c : e )) []
        )
      )
   = { filter0 }
      ( listmax (map length (filter path (subs [])))
      , (\ c ->
         listmax (map length
                      (if path (c : [])
                        then [] : filter (\ e -> path (c : e ))
                                         []
                        else []
                       ))
        )
      )
   = { filter0 }
      ( listmax (map length (filter path (subs [])))
      , (\ c ->
         listmax (map length
                      (if path (c : []) then [] : [] else []))
        )
      )
   = { subs0 }
      ( listmax (map length (filter path a))
      , (\ c ->
         listmax (map length (if path (c : []) then a else []))
        )
      )
      where a = [] : []
   = { filter1 }
      ( listmax (map length (if path [] then [] : a else a))
      , (\ d ->
         listmax (map length
                      (if path (d : []) then [] : [] else []))
        )
      )
      where a = filter path []
   = { filter0 }
      ( listmax (map length (if path [] then [] : filter path []
                                       else []
                           ))
      , (\ c ->
         listmax (map length
                      (if path (c : []) then [] : [] else []))
        )
      )
   = { filter0 }
      ( listmax (map length (if path [] then a else []))
      , (\ b ->
         listmax (map length (if path (b : []) then a else []))
        )
      )
      where a = [] : []
   = { path0 }
      ( listmax (map length (if True then a else []))
      , (\ b ->
         listmax (map length (if path (b : []) then a else []))
        )
      )
      where a = [] : []
   = { path1 }
      ( a, (\ b -> a) )
      where a = listmax (map length
                           (if True then [] : [] else []))
   = { iftrue }
      ( listmax (map length (if True then a else []))
      , (\ b -> listmax (map length a))
      )
      where a = [] : []
   = { map1 }
      ( listmax (map length (if True then [] : [] else []))
      , (\ b -> listmax (length [] : map length []))
      )
   = { map0 }
      ( listmax (map length (if True then [] : [] else []))
      , (\ b -> listmax (length [] : []))
```

```
      )
   = { length0 }
      ( listmax (map length (if True then [] : [] else []))
      , (\ b -> listmax (0 : []))
      )
   = { listmax1 }
      ( listmax (map length (if True then [] : [] else []))
      , (\ b -> max 0 (listmax []))
      )
   = { listmax0 }
      ( listmax (map length (if True then [] : [] else []))
      , (\ b -> 0)
      )
   = { iftrue }
      ( listmax (map length ([] : [])), (\ b -> 0) )
   = { map1 }
      ( listmax (length [] : map length []), (\ b -> 0) )
   = { map0 }
      ( listmax (length [] : []), (\ a -> 0) )
   = { length0 }
      ( listmax (0 : []), (\ a -> 0) )
   = { listmax1 }
      ( max 0 (listmax []), (\ a -> 0) )
   = { listmax0 }
      ( 0, (\ a -> 0) )
  }
   foldr (\ b ->
          uncurry (\ d e ->
                   ( max d a
                   , (\ f ->
                      (if arc f b then max c a else c)
                      where c = e f
                     )
                   )
                   where a = 1 + e b
                  )
         )
         ( 0, (\ g -> 0) )
```

B.4.3 Calculation of llp''

Theory file

```
{- llp2.eq -}

fastllp'': fastllp'' xs ts = llp'' ts (foldr (:) [] xs);

llp'': llp'' ts xs
     = max (llp xs) (listmax
          (map (\kx -> fst kx + llp' (snd kx) xs) ts));

llp0: llp [] = 0;
llp1: llp (x:xs) = max (llp xs) (1+llp' x xs);

llp'0: llp' x [] = 0;
llp'1: llp' x (y:ys) = if arc x y
                        then max (llp' x ys) (1+ llp' y ys)
                        else llp' x ys;

{-
map0: map f [] = [];
map1: map f (x:xs) = f x : map f xs;
-}

listmax0: listmax [] = neginf;
listmax1: listmax (x:xs) = max x (listmax xs);

ifmax: if c then max a b else a
            = max a (if c then b else listmax []);
plusmax: n + max x y = max (n+x) (n+y);
listmaxmax: listmax (map (\x -> max (f x) (g x)) xs)
               = max (listmax (map f xs))
                     (listmax (map g xs));
maxassoc: max (max a b) c = max a (max b c);
maxplus: max n (x+n) = max 0 x + n;
plusneginf: n + neginf = neginf;
listmaxmapconst: listmax (map (\x -> a) xs) = a;
maxneginf: max x neginf = x;
pluszero: n + 0 = n;
maxlistmax: max (listmax (map f xs)) (f x)
                     = listmax (map f (x:xs));
maxmapif: listmax (map (\x -> f x (if g x then v else w)) xs)
        = max (listmax (map (\x -> f x v) (filter g xs)))
```

```
            (listmax (map (\x -> f x w)
               (filter (\x -> not (g x)) xs)));
maxmapplus: listmax (map (\x -> f x + y) xs)
               = listmax (map f xs) + y;
maxcom: max b (max c (listmax (map f xs)+b))
             = max c (max b (listmax (map f xs)+b));

plusllp: v+(1+llp' y ys)
          = (\kx -> fst kx + llp' (snd kx) ys) (1+v,y);

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

Derivation

```
   fastllp''
= { fastllp'' }
   (\ a b -> llp'' b (foldr (:) [] a))
= { llp'' }
   (\ a ->
    (\ b ->
     max (llp c) (listmax (map (\ e -> fst e + llp' (snd e) c) b))
    )
    where c = foldr (:) [] a
   )
= { promotion

      (\ a ->
       max (llp [])
           (listmax (map (\ b -> fst b + llp' (snd b) []) a))
      )
   = { llp0 }
      (\ a ->
       max 0 (listmax (map (\ b -> fst b + llp' (snd b) []) a))
      )
   = { llp'0 }
      (\ a -> max 0 (listmax (map (\ b -> fst b + 0) a)))
   = { pluszero }
```

```
      (\ a -> max 0 (listmax (map fst a)))

      (\ a b ->
       (\ c ->
        max (llp e)
            (listmax (map (\ d -> fst d + llp' (snd d) e) c))
       )
       where e = a : b
      )
   = { llp1 }
      (\ a b c ->
       max (max (llp b) (1 + llp' a b))
           (listmax (map (\ d -> fst d + llp' (snd d) (a : b ))
                         c))
      )
   = { llp'1 }
      (\ a b ->
       (\ c ->
        max (max (llp b) e)
            (listmax (map (\ d ->
                           fst d
                           +
                           (if arc f a then max g e else g)
                           where {
                                 f = snd d;
                                 g = llp' f b
                                 }
                          )
                          c))
       )
       where e = 1 + llp' a b
      )
   = { ifmax }
      (\ a b ->
       (\ c ->
        max (max (llp b) e)
            (listmax (map (\ d ->
                           fst d
                           +
                           max (llp' f b)
                               (if arc f a then e else listmax [])
                           where f = snd d
                          )
```

```
                       c))
       )
       where e = 1 + llp' a b
      )
   = { listmax0 }
      (\ a b ->
       (\ c ->
        max (max (llp b) e)
            (listmax (map (\ d ->
                            fst d
                            +
                            max (llp' f b)
                                (if arc f a then e else neginf)
                            where f = snd d
                           )
                           c))
       )
       where e = 1 + llp' a b
      )
   = { plusmax }
      (\ a b ->
       (\ c ->
        max (max (llp b) e)
            (listmax (map (\ d ->
                            max (g + llp' f b)
                                (g + (if arc f a then e
                                                  else neginf
                                      ))
                            where {
                                  f = snd d;
                                  g = fst d
                                  }
                           )
                           c))
       )
       where e = 1 + llp' a b
      )
   = { listmaxmax }
      (\ a b ->
       (\ c ->
        max (max (llp b) f)
            (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
                 (listmax (map (\ e ->
                                fst e + (if arc (snd e) a
                                          then f
                                          else neginf
                                         )
                                )
                                c)))
       )
       where f = 1 + llp' a b
      )
   = { maxassoc }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max f (max (listmax (map (\ d ->
                                       fst d + llp' (snd d) b
                                      )
                                      c))
                        (listmax (map (\ e ->
                                       fst e + (if arc (snd e) a
                                                 then f
                                                 else neginf
                                                  )
                                      )
                                      c))))
       )
       where f = 1 + llp' a b
      )
   = { maxmapif }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max i
                 (max (listmax (map (\ d ->
                                      fst d + llp' (snd d) b
                                     )
                                     c))
                       (max (listmax (map (\ e -> fst e + i)
                                         (filter (\ f ->
                                                  arc (snd f) a
                                                 )
                                                 c)))
                           (listmax (map (\ g -> fst g + neginf)
                                         (filter (\ h ->
                                                  not (arc (snd h)
                                                           a)
                                                 )
                                                 c))))))
       )
       where i = 1 + llp' a b
      )
   = { plusneginf }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max i
                 (max (listmax (map (\ d ->
                                      fst d + llp' (snd d) b
                                     )
                                     c))
                       (max (listmax (map (\ e -> fst e + i)
                                         (filter (\ f ->
                                                  arc (snd f) a
                                                 )
                                                 c)))
                           (listmax (map (\ g -> neginf)
                                         (filter (\ h ->
                                                  not (arc (snd h)
                                                           a)
                                                 )
                                                 c))))))
       )
       where i = 1 + llp' a b
      )
   = { listmaxmapconst }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max g (max (listmax (map (\ d ->
                                       fst d + llp' (snd d) b
                                      )
                                      c))
                        (max (listmax (map (\ e -> fst e + g)
                                           (filter (\ f ->
                                                    arc (snd f) a
                                                   )
                                                   c)))
```

```
                           neginf)))
       )
       where g = 1 + llp' a b
      )
   = { maxneginf }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max g
                 (max (listmax (map (\ d ->
                                      fst d + llp' (snd d) b
                                     )
                                     c))
                      (listmax (map (\ e -> fst e + g)
                                     (filter (\ f -> arc (snd f) a)
                                             c)))))
       )
       where g = 1 + llp' a b
      )
   = { maxmapplus }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max e (max (listmax (map (\ d ->
                                        fst d + llp' (snd d) b
                                       )
                                       c))
                        (listmax (map fst (filter (\ f ->
                                                    arc (snd f) a
                                                   )
                                                   c))
                         +
                         e)))
       )
       where e = 1 + llp' a b
      )
   = { maxcom }
      (\ a b ->
       (\ c ->
        max (llp b)
            (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
                 (max e (listmax (map fst (filter (\ f ->
                                                    arc (snd f) a
```

```
                                               )
                                               c))
                      +
                      e)))
    )
    where e = 1 + llp' a b
   )
= { maxplus }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (max 0 (listmax (map fst (filter (\ f ->
                                               arc (snd f) a
                                              )
                                              c)))
              +
              (1 + llp' a b)))
   )
= { plusllp }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (fst e + llp' (snd e) b))
    where e = ( 1 + max 0 (listmax (map fst
                                        (filter (\ f ->
                                                 arc (snd f) a
                                                )
                                                c)))
              , a
              )
   )
= { plusmax }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (fst ( 1 + max 0 e, a )
              +
              llp' (snd ( max (1 + 0) (1 + e), a )) b))
    where e = listmax (map fst (filter (\ f -> arc (snd f) a)
                                       c))
   )
= { plusmax }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (fst g + llp' (snd g) b))
    where g = ( max e (1 + listmax (map fst
                                        (filter (\ f ->
                                                 arc (snd f) a
                                                )
                                                c)))
              , a
              )
   )
   where e = 1 + 0
= { pluszero }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (fst ( max (1 + 0) e, a )
              +
              llp' (snd ( max 1 e, a )) b))
    where e = 1 + listmax (map fst
                               (filter (\ f -> arc (snd f) a)
                                       c))
   )
= { pluszero }
   (\ a b c ->
    max (llp b)
        (max (listmax (map (\ d -> fst d + llp' (snd d) b) c))
             (fst e + llp' (snd e) b))
    where e = ( max 1 (1 + listmax (map fst
                                        (filter (\ f ->
                                                 arc (snd f) a
                                                )
                                                c)))
              , a
              )
   )
= { maxlistmax }
   (\ a b c ->
    max (llp b)
        (listmax (map (\ d -> fst d + llp' (snd d) b)
                      (( max 1
                             (1
                              +
                               listmax (map fst
                                            (filter (\ f ->
                                                     arc (snd f)
                                                         a
                                                    )
                                                    c)))
                        , a
                        )
                        :
                        c)))
    )
}
 foldr (\ c d e ->
        d (( max 1 (1 + listmax (map fst (filter (\ g ->
                                                  arc (snd g) c
                                                 )
                                                 e)))
           , c
           )
           :
           e)
       )
       (\ h -> max 0 (listmax (map fst h)))
```

B.5 Optimising *abstracts*

Theory file

```
abstracts: abstracts x e t
         = concat (takeWhile (not.null)
               (map (\n -> abstractssub n x e t) (from 0)));

abstractssub0: abstractssub 0 x e t = [t];
abstractssub1: abstractssub (n+1) x e t
             = nub (concat (map (abstract x e)
                         (abstractssub n x e t)));


from: from m = iterate (\n->n+1) m;

concat0: concat [] = [];
concat1: concat (xs:xss) = xs ++ concat xss;
```

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

takeWhile0: takeWhile p [] = [];
takeWhile1: takeWhile p (x:xs)
          = if p x then x:takeWhile p xs else [];

ifnot: if not b then x else y = if b then y else x;
funcif: f (if b then x else y) = if b then f x else f y;

mapiterate: map g (iterate f x) = iterate h y,
   if { g x = y;
        \a -> g (f a) = \a -> h (g a) };

iterate: iterate f = fix (\g x -> x:g (f x));

fixpointfusion: func (fix rec x) = fix rec' x,
      if { \f -> func.(rec f) = \f -> rec' (func.f) }
```

Derivation

```
   abstracts
= { abstracts }
   (\ a b c ->
    concat (takeWhile (not . null)
                      (map (\ g -> abstractssub g a b c)
                           (from 0)))
   )
= { from }
   (\ a b c ->
    concat (takeWhile (not . null)
                      (map (\ g -> abstractssub g a b c)
                           (iterate (\ h -> h + 1) 0)))
   )
= { compose }
   (\ a b c ->
    concat (takeWhile (\ d -> not (null d))
                      (map (\ e -> abstractssub e a b c)
                           (iterate (\ f -> f + 1) 0)))
   )
= { mapiterate
```

```
      abstractssub 0 ef eg eh
   = { abstractssub0 }
      eh : []

      (\ a -> abstractssub (a + 1) ef eg eh)
   = { abstractssub1 }
      (\ a ->
       nub (concat (map (abstract ef eg)
                        (abstractssub a ef eg eh)))
      )
  }
   (\ a b c ->
    concat (takeWhile (\ d -> not (null d))
                      (iterate (\ e ->
                                 nub (concat (map (abstract a b)
                                                  e))
                                )
                                (c : [])))
   )
= { iterate }
   (\ a b c ->
    concat (takeWhile (\ d -> not (null d))
                      (fix (\ e f ->
                             f : e (nub (concat (map (abstract a
                                                               b)
                                                     f)))
                            )
                            (c : [])))
   )
= { fixpointfusion

      (\ a ->
       (\ c -> concat (takeWhile (\ d -> not (null d)) c))
       .
       (\ e -> e : a (nub (concat (map (abstract ef eg) e))))
      )
   = { compose }
      (\ a b ->
       concat (takeWhile (\ c -> not (null c))
                         (b : a (nub (concat (map (abstract ef
                                                            eg)
                                                  b)))))
```

```
     )
  = { takeWhile1 }
     (\ a b ->
      concat (if not (null b)
               then b
                    :
                    takeWhile (\ c -> not (null c))
                              (a (nub (concat (map (abstract ef
                                                             eg)
                                                   b))))
               else []
              )
     )
  = { ifnot }
     (\ a b ->
      concat (if null b
               then []
               else b
                    :
                    takeWhile (\ c -> not (null c))
                              (a (nub (concat (map (abstract ef
                                                             eg)
                                                   b))))
              )
     )
  = { funcif }
     (\ a b ->
      if null b
       then concat []
       else concat (b
                    :
                    takeWhile (\ c -> not (null c))
                              (a (nub (concat (map (abstract ef
                                                             eg)
                                                   b)))))

     )
  = { concat0 }
     (\ a b ->
      if null b
       then []
       else concat (b
                    :
```

```
                    takeWhile (\ c -> not (null c))
                              (a (nub (concat (map (abstract ef
                                                             eg)
                                                    b)))))

    )
  = { concat1 }
    (\ a b ->
     if null b
      then []
      else b
           ++
           concat (takeWhile (\ c -> not (null c))
                             (a (nub (concat (map (abstract ef
                                                             eg)
                                                   b)))))

    )

    (\ a ->
     aq ((\ d -> concat (takeWhile (\ e -> not (null e)) d))
         .
         a)
    )
  = { compose }
    (\ a ->
     aq (\ c -> concat (takeWhile (\ d -> not (null d)) (a c)))
    )
 }
  (\ a b c ->
   fix (\ d e ->
        if null e
         then []
         else e ++ d (nub (concat (map (abstract a b) e)))

       )
       (c : [])
  )
```