AN ABSTRACT OF THE DISSERTATION OF

Deling Ren for the degree of Doctor of Philosophy in Computer Science presented on August 28, 2006.

Title: Update Programming – A Safe Approach to Software Maintenance

Abstract approved: _____________________________________________________________

Martin Erwig

Software maintenance accounts for a large portion of the software development cost, particularly the process of updating programs either to adapt for requirement change or to enhance design or efficiency. Currently, program updates are generally performed manually by programmers using text editors. This is an unreliable method because syntax and type errors are easily introduced, not to mention logic and semantic errors. The problem with this method is viewing programs on a low level, namely, as streams of characters. Rather than the textual representation, we propose to view the programs as instances of abstract data types and to update programs through programming updates in update languages. In this dissertation, we first study a specific program update problem, monadification, which is the process of automatic introduction of monads into functional programs. Later we investigate a more general problem of program updates and present an update language. We design
a core calculus for the update language as well as its semantics and type system. Finally, we study the problem of generic traversals, which itself is a means of reducing the effort for software maintenance, but also serves as a basis for implementing program updates and other metaprogramming tasks.
Update Programming – A Safe Approach to Software Maintenance

by

Deling Ren

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 28, 2006
Commencement June 2007

APPROVED:

________________________________________
Major Professor, representing Computer Science

________________________________________
Director of the School of Electric Engineering and Computer Science

________________________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

________________________________________
Deling Ren, Author
ACKNOWLEDGEMENTS

I owe the completion of this dissertation to my academic advisor Martin Erwig, who taught me everything I know about programming languages. I also would like to thank Ralf Lämmel who inspired a great portion of the work described in this dissertation, including the initial idea of monadification as well as the foundation of Reclib.

I thank all contributors to Haskell, the official programming I have been using for the past five years. I also thank everyone who contributed to TeX, \LaTeX, and \BibTeX, with which this dissertation is typeset.

I particularly thank my alma mater, University of Science and Technology of China. I met some of the greatest people in the world there. I also received rigorous training in mathematics and theoretical computer science, which will benefit my whole life.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Safety of Program Updates</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Representation of Programs</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Examples</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>Organization</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Literature Review</td>
<td>14</td>
</tr>
<tr>
<td>2.1</td>
<td>Strategic Term Rewriting</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Stratego</td>
<td>18</td>
</tr>
<tr>
<td>2.1.2</td>
<td>ELAN</td>
<td>26</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Maude</td>
<td>28</td>
</tr>
<tr>
<td>2.1.4</td>
<td>Discussion</td>
<td>31</td>
</tr>
<tr>
<td>2.2</td>
<td>Language-Based Approaches</td>
<td>31</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Generic Haskell</td>
<td>33</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Template Haskell</td>
<td>36</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Scrap-Your-Boilerplate</td>
<td>45</td>
</tr>
<tr>
<td>2.3</td>
<td>Practical Systems</td>
<td>54</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Eclipse</td>
<td>55</td>
</tr>
<tr>
<td>2.3.2</td>
<td>HaRe</td>
<td>62</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Visual Haskell</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>Monadification</td>
<td>68</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>69</td>
</tr>
<tr>
<td>3.2</td>
<td>Examples</td>
<td>75</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Exception Handling</td>
<td>75</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Counting Operations</td>
<td>76</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Producing Output</td>
<td>78</td>
</tr>
<tr>
<td>3.3</td>
<td>The Essence of Monadification</td>
<td>79</td>
</tr>
<tr>
<td>3.4</td>
<td>Monadification Algorithm</td>
<td>88</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Characterizations of Subexpressions</td>
<td>91</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Locating Return Expressions</td>
<td>93</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Wrapping Return Expressions</td>
<td>95</td>
</tr>
<tr>
<td>3.5</td>
<td>Correctness of Monadification</td>
<td>100</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

3.6 Runnable Monads .................................................. 103
3.7 Adding Actions to Monads ........................................... 106
   3.7.1 Three Options for Adding Actions ....................... 107
   3.7.2 A Context Update Language ............................... 108
   3.7.3 Type Preservation ......................................... 113
3.8 Alternative Approaches ........................................... 113
   3.8.1 Type-Directed Approach .................................. 114
   3.8.2 The CPS Approach ....................................... 117
3.9 Summary and Future Work ....................................... 123

4  Update Language .................................................... 124
   4.1 Introduction .................................................... 125
   4.2 The Update Calculus ......................................... 125
      4.2.1 The Object Language .................................... 125
      4.2.2 Syntax .................................................... 128
      4.2.3 Semantics ................................................ 134
      4.2.4 Type System ............................................. 141
      4.2.5 Possible Extensions ................................... 158
   4.3 The Haskell Update Language ................................ 159
      4.3.1 Syntax .................................................... 160
      4.3.2 Translation into the Update Calculus ................ 162
   4.4 Summary ......................................................... 168

5  Generic Recursion and Traversals ................................. 169
   5.1 Introduction .................................................... 170
      5.1.1 Possible Extensions ................................. 172
      5.1.2 Contribution and Organization ....................... 177
   5.2 Design Space .................................................... 179
   5.3 Programming Interface ....................................... 183
      5.3.1 Building Generic Functions .......................... 184
      5.3.2 Traversal Engine ...................................... 187
      5.3.3 Syntactic Sugar ....................................... 189
      5.3.4 Crafting Traversals .................................. 191
      5.3.5 The Mother of All Traversals ...................... 195
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.6 Failure and Continuation</td>
<td>197</td>
</tr>
<tr>
<td>5.3.7 Fixpoint Traversals</td>
<td>201</td>
</tr>
<tr>
<td>5.4 Examples</td>
<td>204</td>
</tr>
<tr>
<td>5.5 Application in Monadification</td>
<td>207</td>
</tr>
<tr>
<td>5.6 Related Work</td>
<td>210</td>
</tr>
<tr>
<td>5.7 Summary and Future Work</td>
<td>212</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>214</td>
</tr>
<tr>
<td>Bibliography</td>
<td>216</td>
</tr>
<tr>
<td>Appendices</td>
<td>224</td>
</tr>
<tr>
<td>A Algebraic Data Types</td>
<td>225</td>
</tr>
<tr>
<td>B Type Classes</td>
<td>230</td>
</tr>
<tr>
<td>C Monads</td>
<td>234</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Safe Update Theorems</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>A Parse Tree</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>An Abstract Syntax Tree</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Context Free Grammar of $B$</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Rewrite Rules for Evaluating $B$ Expressions</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Stratego: A Typical Application</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Syntax Definition of $B$ in SDF Bool.def</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Regular Tree Grammar for $B$ Bool.rtg</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>Algebraic Signatures of the Abstract Syntax of $B$ BoolSig.str</td>
<td>24</td>
</tr>
<tr>
<td>2.7</td>
<td>Rules and Strategies for Evaluating $B$ Expressions BoolEval.str</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>An ELAN Program for Evaluating $B$: bool.eln</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>A Maude Program for Evaluating $B$: bool.maude</td>
<td>29</td>
</tr>
<tr>
<td>2.10</td>
<td>An Example Solution to the 8-Queen Problem</td>
<td>39</td>
</tr>
<tr>
<td>2.11</td>
<td>Typing States and Levels for Template Haskell</td>
<td>44</td>
</tr>
<tr>
<td>2.12</td>
<td>An Application of Function gmapT</td>
<td>49</td>
</tr>
<tr>
<td>2.13</td>
<td>The Eclipse IDE</td>
<td>56</td>
</tr>
<tr>
<td>2.14</td>
<td>Renaming a Java Element in Eclipse</td>
<td>58</td>
</tr>
<tr>
<td>2.15</td>
<td>Extracting Method in Eclipse</td>
<td>59</td>
</tr>
<tr>
<td>2.16</td>
<td>Extracting Method in Eclipse</td>
<td>60</td>
</tr>
<tr>
<td>2.17</td>
<td>Visual Haskell: Developing Environment</td>
<td>65</td>
</tr>
<tr>
<td>2.18</td>
<td>Visual Haskell: Tooltip for Type Information</td>
<td>66</td>
</tr>
<tr>
<td>2.19</td>
<td>Visual Haskell: Reporting a Scoping Error</td>
<td>66</td>
</tr>
<tr>
<td>2.20</td>
<td>Visual Haskell: Reporting a Type Error</td>
<td>66</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1</td>
<td>Syntax of the Object Language</td>
<td>83</td>
</tr>
<tr>
<td>3.2</td>
<td>Saturated Function Calls</td>
<td>91</td>
</tr>
<tr>
<td>3.3</td>
<td>Expressions Containing Recursive Calls</td>
<td>92</td>
</tr>
<tr>
<td>3.4</td>
<td>Syntax of Contexts</td>
<td>92</td>
</tr>
<tr>
<td>3.5</td>
<td>Definition of the $N$ Operator</td>
<td>94</td>
</tr>
<tr>
<td>3.6</td>
<td>Definition of the $W$ Operator</td>
<td>99</td>
</tr>
<tr>
<td>3.7</td>
<td>Syntax of the Context Update Language</td>
<td>109</td>
</tr>
<tr>
<td>3.8</td>
<td>Semantics of Context Updates</td>
<td>111</td>
</tr>
<tr>
<td>3.9</td>
<td>Type-Directed Monad Introduction</td>
<td>115</td>
</tr>
<tr>
<td>4.1</td>
<td>Abstract Syntax of Lambda Calculus</td>
<td>126</td>
</tr>
<tr>
<td>4.2</td>
<td>Types for Lambda Calculus</td>
<td>126</td>
</tr>
<tr>
<td>4.3</td>
<td>Type System for Lambda Calculus</td>
<td>127</td>
</tr>
<tr>
<td>4.4</td>
<td>Patterns</td>
<td>129</td>
</tr>
<tr>
<td>4.5</td>
<td>Syntax of Updates</td>
<td>132</td>
</tr>
<tr>
<td>4.6</td>
<td>Semantics of Updates</td>
<td>139</td>
</tr>
<tr>
<td>4.7</td>
<td>Type Changes</td>
<td>144</td>
</tr>
<tr>
<td>4.8</td>
<td>Applicative Instance</td>
<td>146</td>
</tr>
<tr>
<td>4.9</td>
<td>Type Change System</td>
<td>149</td>
</tr>
<tr>
<td>4.10</td>
<td>Syntax of the Haskell Update Language</td>
<td>162</td>
</tr>
<tr>
<td>4.11</td>
<td>Translation of HULA into the Core Calculus</td>
<td>163</td>
</tr>
<tr>
<td>5.1</td>
<td>Relation between Redlib and Related Approaches</td>
<td>178</td>
</tr>
<tr>
<td>5.2</td>
<td>Feature comparison of Redlib and Related Approaches</td>
<td>178</td>
</tr>
<tr>
<td>5.3</td>
<td>Types of 6 Kinds of Visits</td>
<td>180</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Traversals of 4 Kinds of Coverage</td>
<td>182</td>
</tr>
<tr>
<td>5.5</td>
<td>Vertical Directions</td>
<td>183</td>
</tr>
<tr>
<td>5.6</td>
<td>Horizontal Directions</td>
<td>183</td>
</tr>
<tr>
<td>5.7</td>
<td>Types of Traversals</td>
<td>188</td>
</tr>
<tr>
<td>5.8</td>
<td>Children of the Mother of Traversals</td>
<td>197</td>
</tr>
</tbody>
</table>
DEDICATION

To my parents and to my wife.
In memory of Xue, Deqi (1937-2006).
Chapter 1 – Introduction

Software maintenance is the process of enhancing and optimizing software, as well as fixing defects. It is an important phase in the life cycle of software development. Programs usually undergo many changes throughout their lifetime, or as Lehman puts it: “There is no such thing as a ‘finished’ computer program” [48]. Software maintenance accounts for a large portion of the overall software development expense. Different estimates have been made. Pressman estimates that over 60% of the effort in software development is in software maintenance. Some estimate it as high as 70% [78, 2], or even 80% [83]. Improvements to software maintenance will greatly reduce the cost of software development, and increase its reliability, security, and performance.

In this dissertation, we propose an approach to software maintenance called *update programming*. This approach provides a means to effective and safe software maintenance. In this approach, updates are expressed as *update programs*. Update programs are written in *update languages*. The inputs and outputs to update programs are original and resulting programs. One advantage of this approach is that the update programs can guarantee certain properties of the resulting program. Also, once the program updates are written, they can be performed many times on different programs. Performing updates with update programs also has potential advantages in version control. Suppose at a certain point we branch off from the
program to include features not to be included in the main version. In a traditional
editor approach, when the main program evolves, this branch has to fork again,
and all the changes must be performed on new version of the program. However,
if the branch can be described with an update \( u \) on the original program, as long
as \( u \) does not conflict with the evolution of the program, it can be applied to the
new version.

To elaborate the idea of update programming, we will first define the concept
of safe updates. A key factor to safe update programming is an effective represen-
tation of programs in the form of abstract data types. To motivate the research
described in this dissertation, we will give a brief overview of these two issues in
this introduction, and will provide several concrete examples of program updates.

1.1 Safety of Program Updates

Let us first describe a general scenario of program updates and introduce some
concepts. A program \( p \) to be updated is an element of a language \( P \), also called
object language. A property \( \pi \) on a language \( P \) is given by a boolean function
on \( P \), that is, properties are propositions about programs. The fact that \( p \) is
correct with respect to the property \( \pi \) is written as \( \pi(p) \). In that case we say that
\( p \) is \( \pi \)-correct or \( \pi \)-valid. Updates are specified in an update language \( U \). The
meaning of an update program \( u \) is a function \([u]\) on object programs, that is,
\([\_] : U \rightarrow P \rightarrow P \). The following definition introduces an essential concept of the
update-programming approach.
Definition 1 (Safety of Updates) An update $u$ is safe with respect to the language property $\pi$ (or, $u$ is $\pi$-safe, for short) if and only if $\forall p \in P. \pi(p) \implies \pi([u](p))$.

Since $U$ is a language, we can identify properties for $U$ (as we did for $P$). We use $\mu$ to denote a property on $U$. To design an update language we have to find characterizations of safe updates, that is, we want to find properties $\mu$ such that $\mu$-correctness implies $\pi$-safety. In other words, given $P$, $\pi$, and $U$, find $\mu$ such that $\mu(u) \implies u$ is $\pi$-safe, or, expanding the definition of $\pi$-safety, $\forall u. \mu(u) \implies (\forall p. \pi(p) \implies \pi([u](p)))$. This generic schema for theorems is illustrated in Figure 1.1.

![Figure 1.1: Safe Update Theorems.](image-url)

We can now investigate a variety of update languages for different object languages preserving different properties under varying criteria. We call any instance $(P, \pi, U, \mu)$ of this general setting an update scenario. The existence of safe update theorems serves as the key question for the significance of any update scenario.
because if \( \mu \)-correctness implies \( \pi \)-safety, we can work on updates completely independently from the object programs to which they will be eventually applied. This property is as important as static typing is for object programs—the partial correctness of an update (in the sense of not producing an object program violating \( \pi \)) can be checked without having to execute it (that is, apply it to an object program). The focus on type correctness as one important property that an update language should preserve is justified by empirical studies [70]. However, from a practical point of view, there are also other properties whose preservation is worthwhile. For example, the information about to what degree updates can preserve layout or comments of updated programs is an important criterion for offering these updates in a program editor because programmers would probably not like nor expect their programs to be completely reformatted.

In the context of this dissertation, we will study several scenarios with \( P = \) lambda calculus (possibly with extensions), \( \pi = \) type correctness.

### 1.2 Representation of Programs

Changes to programs are usually done by the programmers manually using text editors. This approach is often problematic. Syntactic errors can be easily introduced by inconsistent or erroneous changes, let alone type errors and logical errors. For example, a misspelled keyword name can make the program syntactically incorrect, causing compilation to fail. This unfortunate situation is caused by the fact that programs are treated at a low level, namely, streams of characters. This
view of programs totally ignores the structure of programs, but focuses on the textual representation. Several researchers have identified the situation of lacking tool support for software maintenance [78, 83, 74].

Programs can be alternatively viewed on a higher level, that is, abstract syntax. An abstract syntax tree is used to represent a program. The internal nodes of an abstract syntax tree are labeled by constructors, and the leaves are labeled by values. The subtrees of a node are the arguments to the constructor. An abstract syntax tree should not be confused with a parse tree. The former does not contain nodes for syntax elements that do not have effect on the semantics, for example, keywords. An abstract syntax tree reveals the semantic structure rather than the syntactic structure of the program. Consider an assignment $x := y + 3$, the parse tree of this statement is presented in Figure 1.2. The abstract syntax tree for the same statement is presented in Figure 1.3.

```
assignment
  
  x := expr
  
  y + 3
```

Figure 1.2: A Parse Tree

An abstract syntax tree can be constructed from a program by a parser. A textual representation can be restored from the abstract syntax tree by the process of pretty printing according to the syntax rules. As long as the pretty printing is
implemented correctly, the generated program is guaranteed to be syntactically correct. Because of this property, program updates can be conveniently expressed on the ASTs, such that the result program is guaranteed syntactically correct.

Updating programs on the textual representation level also renders problems other than syntactic incorrectness. Even with tool support, such as perl or awk, or syntax-directed editors, these approaches do not have knowledge of the type system and scoping rules for the object programs. For example, changing a function definition using an editor might fail to change all the references, changing a variable name might cause the capture of a free variable. Even worse, some of these problems might slip by the compiler and cause logic errors, unnoticed. In such cases, correcting defects in programs might introduce new errors.

Carefully designed, the operations on the ASTs can respect the typing rules and scoping rules of the object language. Therefore, the resulting programs are not only syntactically correct, but can also be type correct. In this dissertation,
we will define a update language for programming updates. A set of basic updates can be defined in the language. A complex update program can be constructed from basic updates.

1.3 Examples

To illustrate the idea of safe program updates, let us look at several examples. Our first example is a program in Haskell.

Suppose we want to extend a module for binary search trees by a size operation giving the number of nodes in a tree. Moreover, we want to support this operation in constant time. We extend the representation of the tree data type by an integer field for storing the information about the number of nodes contained in a tree. The desired program extension goes beyond the idea of refactoring (see Chapter 2), which is concerned with semantics-preserving restructuring of programs, and illustrates that update programming has applications that are more general than refactorings. The definition of the original tree data type and an insert function are as follows.¹

```haskell
data Tree a = Leaf | Node a Tree Tree

insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r) = if x < y then Node y (insert x l) r
else Node y l (insert x r)
```

The program update requires a new function definition size, a changed type for

¹An introduction to algebraic data types can be found in Appendix A.
the \texttt{Node} constructor (since a leaf always contains zero nodes, no change for this constructor is needed), and a corresponding change for all occurrences of \texttt{Node} in patterns and expressions. Adding the definition for the size function is straightforward and is not very exciting from the update programming point of view. The change of the \texttt{Node} constructor is more interesting since the change of its type in the \texttt{data} definition has to be accompanied by corresponding changes in all \texttt{Node} patterns and \texttt{Node} expressions. This series of changes can be systematically expressed by the following hypothetical update, given in \textit{HULA} syntax (see Chapter 4).

\begin{verbatim}
  con Node:{Int} t where
      (case Node {s} \rightarrow Node {suc s}
       | Leaf \rightarrow Node {1}) \mid Node {1}
\end{verbatim}

The update can be read as follows. The \texttt{con} update operation adds the type \texttt{Int} as a new first parameter to the definition of the \texttt{Node} constructor. The notation \( a \{ r \} b \) is an abbreviation for the rewrite rule \( a \cdot b \rightarrow a \cdot r \cdot b \). So \( \{ \text{Int} \} \ t \) means extend the type \( t \) on the left by \texttt{Int}. The keyword \texttt{where} introduces the updates that apply to the scope of the \texttt{Node} constructor. Here, a \texttt{case} update specifies how to change all pattern matching rules that use the \texttt{Node} constructor: \texttt{Node} patterns are extended by a new object variable \( s \), and to each application of the \texttt{Node} constructor in the return expression of that rule, the expression \texttt{suc s} is added as a new first argument (\texttt{suc} denotes the successor function on integers, which is predefined in Haskell). The \texttt{Leaf} pattern is left unchanged, and all occurrences of the \texttt{Node} constructor within its return expression are extended by \texttt{1}. As an alternative to the \texttt{case} update, the rule \texttt{Node \{1\}} extends all other \texttt{Node} expressions by \texttt{1}. 
The application of the update to the original program yields the following object program.

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

insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf = Node 1 x Leaf
insert x (Node s y l r) =
  if x<y then Node (succ s) y (insert x l) r
  else Node (succ s) y l (insert x r)
```

It is striking that with the shown definition the `case` update is applied to all case expressions in the whole program. In this example, this works well since we have only one function definition in the program. In general, however, we want to be able to restrict `case` updates to specific functions or specify different `case` updates for different functions. This behavior can be achieved by using a further update operation that performs updates on function definitions:

```haskell
con Node::{Int} t where
  fun insert x y :
    (case Node {s} -> Node {succ s}
     | Leaf -> Node {1}) | Node {1}
```

This update applies the `case` update only to the definition of the function `insert`.

Uses of the function `insert` need not be updated, which is indicated by the absence of the keyword `where` and a following update. We can add further `fun` updates for other functions in the program to be updated each with its own `case` update. Note that the variables `x` and `y` of the update language are metavariables with respect to Haskell that match any object (that is, Haskell) variable.
The idea of update programming is, of course, not limited to Haskell. To illustrate how the described concepts can be employed in other programming languages, we describe a Java version of the presented tree example. In Java, the abstract data type can be defined as a class Tree, with a constructor Tree and an insert method. We omit the implementation details of insert, because they are not needed to illustrate the idea of update programming.

```java
public class Tree
{
    Object content;
    Tree left, right;

    public Tree()
    {
        content = null;
        left = null;
        right = null;
    }
    public void insert(Tree t)
    {
        ...
        return;
    }
}
```

To extend the class with a tag for size, we add a field size and make corresponding changes to affected member methods. In this case, the class constructor Tree needs to be changed so that size is initialized when the tree is constructed; the insert method needs to be changed so that size is increased whenever a node is added. A corresponding update program could be defined as follows:
class Tree:
{
    int size;
    Object content;
    Tree left, right;

public Tree ()
{
    size = 0;
    content = null;
    left   = null;
    right  = null;
}
public void insert (Tree t)
{
    size ++;
    ... 
    return;
}
}

In all three cases the metavariable \( d \) matches the definition of the class, method, and constructor, respectively. The expressed updates are simply inserting a variable declaration (in the case of the class update) and a variable assignment (in the case of the method and constructor update) in front of the existing definitions. Applying the update to the original program will yield the following updated Java program, with added code marked.

public class Tree
{
    int size;
    Object content;
    Tree left, right;

public Tree ()
{
    size = 0;
    content = null;
    left   = null;
    right  = null;
}
public void insert (Tree t)
{
    size ++;
    ... 
    return;
}
}

To give another example, consider the task of generalizing a function definition,
which works by identifying expressions in a function definition that should be made variable. This update can be identified by two parameters, the name of the function to be generalized \( f \) and the expression \( e \) to be generalized within \( f \)'s definition. The generalization works essentially in three steps. First, a new parameter \( x \) is added to the definition of \( f \), as the \( i \)th parameter, which is expressed by placing \( \{ i | x \} \) after the function name \( f \). Second, the expression \( e \) to be abstracted is replaced everywhere in \( f \)'s definition by the new variable \( x \). The replacement is expressed using the substitution notation \( \{ e \leftarrow x \} \) (read: “replace \( e \) by \( x \)”). Two special cases of this substitution notation are \( \{ e \leftarrow \} \) (“delete \( e \)”) and \( \{ x \} \) (“insert \( x \)”), which we have already seen in the size update. Finally, all applications of the function are to be extended by adding a new argument, which we choose to be the abstracted expression so that the meaning of the original program is preserved.

\[
\text{genFun} \ (e, f) = \text{fun} : \{ e \leftarrow x \} \ \text{where} \ f \ \{ e \}
\]

1.4 Organization

In this dissertation, we study the problem of safe program updates and related tools, in particular, generic traversals. We begin with a literature review in Chapter 2. By investigating three systems we study a common practice in metaprogramming, term rewriting, that can be used to implement program updates. Then we turn to an alternative to program updates, generic programming, whose purpose is to make programs generic and avoid updates as much as possible. We also study two practical tools for program updating and syntax-directed editing. The purpose
of this rather long chapter is not only to discuss related work, but also to provide a detailed introduction to the necessary background for later parts of the dissertation (for instance, Section 3.7.2). In Chapter 3, we study a concrete problem in program updates, monadification. It begins with an introduction to monads and monadic style of programming. We then observe some examples and propose the idea of automatic monad introduction. An algorithm is presented and compared with other known approaches. In Chapter 4, we present an update calculus for a simple object language, which is an extension of lambda calculus. We also define an update language, HULA, for Haskell. We then present difficulties and problems that prevent us from representing complex updates such as monadification using the update calculus. Chapter 5 is independent of other chapters to a certain extent. It presents a generic traversal library in Haskell that particularly can facilitate the implementation of program updates, but has contributions in areas beyond program updating. This library contains a collection of generic traversal strategies and releases the burden of writing “boilerplate codes” from programmers. Summaries and conclusions are presented in Chapter 6.
Chapter 2 – Literature Review

Significant efforts have been made in the area of automatic program updates. In this chapter, we will conduct an overview in the area of program transformation systems and tools. In Section 2.1, we will study term rewriting through three term rewriting systems (Stratego, ELAN, and Maude) by defining a simple boolean language and the evaluation problem of the language. We present and compare solutions to the problem in each of the systems. In Section 2.2, we will take a different perspective: try to minimize the need for program transformation by making programs generic. Three approaches are studied: Generic Haskell, Template Haskell, and Scrap-Your-Boilerplate. Examples are given in each of these approaches to reveal the distinctive features. Finally, we will investigate three practical programming tools in Section 2.3: Eclipse, Visual Haskell, and HaRe. The designs of these systems focus on refactoring, a specific kind of program transformation.

2.1 Strategic Term Rewriting

Term rewriting [79] is a formalism for reducing a term to normal form with respect to a set of rewrite rules. Term rewriting is a cornerstone of theorem proving systems [65]. It also has broad applications in metaprogramming, such as, optimization [89].
We will now illustrate the basic concepts of a rewrite system through an example of evaluating boolean expressions by repeatedly simplifying its subterms.

The Object Language $B$

In the rest of this section, we will walk through a simple example to explain the workflow of creating a typical application in several term rewriting systems. We consider a trivial boolean expression language $B$ with only constants, negations, conjunctions and disjunctions. We define the grammar of $B$ in Figure 2.1.

$$b ::= \text{True} \mid \text{False} \mid !b \mid b \& b \mid b \mid b \mid (b)$$

Figure 2.1: Context Free Grammar of $B$

Constants can be either True or False. Negation is a right-associative unary operator $!$ and has the highest priority. Conjunction $\&$ and disjunction $|$ are left-associative binary operators in that order of priority. Some examples of expressions in $B$ are shown below.

True

True & False

True & (False | !!True | False & True)

We consider the problem of evaluating expressions in this language in the usual boolean algebra. This problem can be viewed as a term rewriting problem. Constants are irreducible. An expression is a redex if it matches one the following two
patterns.

- It is a negation of a constant.
- It is a conjunction or disjunction with one subexpression being a constant.

In each step of reduction, a subexpression that is a redex is reduced by one of the rewrite rules defined in Figure 2.2.

<table>
<thead>
<tr>
<th>NOT1:</th>
<th>!True ⇒ False</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT2:</td>
<td>!False ⇒ True</td>
</tr>
<tr>
<td>AND1:</td>
<td>True &amp; b ⇒ b</td>
</tr>
<tr>
<td>AND2:</td>
<td>False &amp; b ⇒ False</td>
</tr>
<tr>
<td>AND3:</td>
<td>b &amp; True ⇒ b</td>
</tr>
<tr>
<td>AND4:</td>
<td>b &amp; False ⇒ False</td>
</tr>
<tr>
<td>OR1:</td>
<td>False</td>
</tr>
<tr>
<td>OR2:</td>
<td>True</td>
</tr>
<tr>
<td>OR3:</td>
<td>b</td>
</tr>
<tr>
<td>OR4:</td>
<td>b</td>
</tr>
</tbody>
</table>

Figure 2.2: Rewrite Rules for Evaluating $B$ Expressions

It is easy to see that if an expression contains no redex as a subexpression, it must be a constant. It can also be proved that the reduction is confluent, that is, when an expression contains more than one redex, the order in which they are reduced does not affect the final result.

The following shows a sequence of rewritings. We underline the term which is
to be rewritten in each step.

\[
\begin{align*}
\text{True} & \& (\neg \text{False} \mid \neg \neg \text{True} \mid \neg \text{False} \& \text{True}) & \rightarrow & \text{AND1} \\
\neg \text{False} & \mid \text{neg} \neg \text{True} & \rightarrow & \text{AND2} \\
\neg \text{False} & \mid \neg \text{True} & \rightarrow & \text{OR3} \\
\neg \text{False} & \mid \text{True} & \rightarrow & \text{NOT1} \\
\neg \text{False} & \mid \neg \text{False} & \rightarrow & \text{NOT2} \\
\neg \text{False} & \mid \text{True} & \rightarrow & \text{OR4} \\
\rightarrow & \text{True}
\end{align*}
\]

The basic component of a rewrite system is a set of rules. A rule consists of a left-hand side and a right-hand side, both of which are terms. Here are two rules for evaluating negation of constants.

\[
\begin{align*}
\neg \text{False} & \Rightarrow \text{True} \\
\neg \text{True} & \Rightarrow \text{False}
\end{align*}
\]

Such rewrite rules are applied to terms that match the left-hand side and replace them with the right-hand side. A rewrite rule can also contain variables. Consider the following rule.

\[
\text{True} \& b \Rightarrow b
\]

The LHS of the rule is matched against a term. The variables can be matched against any subterm, and a binding is generated. The binding is then applied to the RHS to generate the new term. For example, if we apply the above rewrite
rule to a term, say, $\text{True} \mid (\text{False} \mid \text{True})$, the term is matched against the LHS of the rule, and a binding \( b \mapsto (\text{False} \mid \text{True}) \) is generated. Then the RHS is instantiated by replacing $b$ with $(\text{False} \mid \text{True})$.

Another key component of a rewrite system is strategies [65, 40]. Strategies control the order in which the rules are applied. The rules are sometimes not confluent and terminating. Typical examples of strategies include top-down and innermost. We will delve more into this topic in Chapter 5.

2.1.1 Stratego

Stratego/XT [77, 86, 11] is a framework for the development of transformation systems. It consists of the language Stratego [88, 85] and a collection of tools. The underlying formalism of Stratego is System S [88]. It provides a rich set of traversal strategies. However, it has two main disadvantages. First, it treats the input as untyped plain terms. Therefore, when dealing with abstract syntax trees of programs, rewrite rules may produce syntactically invalid terms. Second, it is a standalone tool rather than a library. Programs that want to make use of it must generate inputs for a Stratego program and take the output in a special ATerm format.

In Stratego, the specification of a program transformation consist of a signature, a set of rewrite rules, and a strategy for applying the rules. The program transformation accepts abstract syntax trees, which are represented by first-order terms, called ATerms [80, 15]. The algebraic signature describes the constructors
of these terms. The concrete syntax definition of the object language is given in SDF [37, 90, 82]. A rewrite rule is a basic transformation step on an abstract syntax tree. A strategy combines a set of named rules into a complete transformation. It does so by using control and traversal combinators.

Figure 2.3 shows the architecture of a typical transformation system in Stratego. The oval nodes are programs from the tool set. Boxes represent data. Double framed boxes are data from the user input. The output of sdf2rtg directly pipelined to the input of rtg2sig, therefore a box is omitted. An evaluator is the product of strc but as well as a program itself.

Syntax Definition

Stratego uses SDF for syntax definition. In SDF, the syntax definition of a language is usually divided into modules. There is a main module that imports other modules. A tool pack-sdf is used to collect all modules imported by the main module and create a definition file. In our example, language B is simple enough to be defined in one main module and an auxiliary module just for layout. Therefore, we omit the step of defining modules and directly go to the definition file. The result is the definition shown in Figure 2.4.

The main module Bool defines the concrete syntax of B. In contrast with Lex+Yacc-like or BNF formalism, in SDF the productions are backwards. This is closer to the actual process of a bottom-up parsing. Other than this slight difference, SDF has some advantages over Lex+Yacc:
• SDF provides a means of defining lexical and context-free syntax.

• SDF allows defining associativity and priority between productions.

• Most importantly, the SDF tools allow automatic construction of abstract syntax trees.

The keyword sorts declares a collection of algebraic sorts defined in the module. In this example, we only export one sort Bool. What follows are the definitions of the concrete syntax as well as the abstract syntax. Every line describes a production.
definition

module Bool

imports Layout
exports
sorts Bool

context-free start-symbols Bool
context-free syntax
  "True" -> Bool {cons("True")}
  "False" -> Bool {cons("False")}
  Bool "&" Bool -> Bool {left,cons("And")}
  Bool "|" Bool -> Bool {left,cons("Or")}
  "!" Bool -> Bool {cons("Not")}
  "(" Bool ")" -> Bool {bracket}

context-free priorities
  "!" Bool -> Bool >
  Bool "&" Bool -> Bool >
  Bool "|" Bool -> Bool

module Layout

exports
lexical syntax
  [\t
] -> LAYOUT

context-free restrictions
  LAYOUT? -/\ [\t
]

Figure 2.4: Syntax Definition of $B$ in SDF Bool.def
Enclosed inside the curly brackets are the descriptions of the constructors of the abstract syntax as well as the associativity of the operator. Arguments to the constructors are the non-terminals on the left hand side of the arrow. Consider, for example, the following production rule.

\[
\text{Bool } \&\text{ Bool } \rightarrow \text{Bool} \{\text{left, cons("And")}\}
\]

It includes the lexical definition of the conjunction operator \&, its left associativity, and the constructor \texttt{And} for the abstract syntax, which takes two \texttt{Bool} values as arguments. What follows is the priority definition of the operators.

The module \texttt{Layout} introduces spaces, tabs, and newlines for the language and eliminates the ambiguity in parsing layouts.

**Parsing**

We can generate a parse table \texttt{Bool.tbl} for \texttt{B} from the syntax definition using command \texttt{sdf2table -m Bool -i Bool.def -o Bool.tbl}.

The \texttt{sglri} tool takes a parse table generated by \texttt{sdf2table} and parses a text file. The result is an abstract syntax tree in ATerm notation. Assume we have a file \texttt{Test.b} containing the following expression:

\[
\text{True} \& ((\text{False} | !\text{True}) | \text{False} \& \text{True})
\]

When we run the parser driven by the parse table just generated on \texttt{Test.b}, we get an annotated term as follows.
And(True, Or(Or(Not(False), Not(Not(True))), And(False, True)))

Signature of the Abstract Syntax

Our next task is to create the evaluator with Stratego/XT. In this example, we need two major parts for the Stratego program: a set of signatures of the abstract syntax, which can be automatically generated, and a set of rules and strategies for the actual transformation. The algebraic signatures describe the constructors. They can be derived from the regular tree grammar (RTG) using the rtg2sig tool. The regular tree grammar is in turn automatically generated from the syntax definition by the sdf2rtg tool. The regular tree grammar is shown in Figure 2.5, and the signatures in Figure 2.6 are generated from the regular tree grammar in Figure 2.5.

```
regular tree grammar
start Bool
productions
  Bool -> Not(Boolean)
  Bool -> Or(Boolean, Boolean)
  Bool -> And(Boolean, Boolean)
  Bool -> False()
  Bool -> True()
```

Figure 2.5: Regular Tree Grammar for $B\text{Bool}.rtg$
module BoolSig

signature
  constructors
    Not : Bool -> Bool
    Or : Bool * Bool -> Bool
    And : Bool * Bool -> Bool
    False : Bool
    True : Bool

Figure 2.6: Algebraic Signatures of the Abstract Syntax of $B$BoolSig.str

Rules and Strategies

The rules in Figure 2.2 can be directly written as a Stratego program as shown in Figure 2.7. A rule in Stratego needs a name to be referred to for constructing other rules and strategies. Different rules may have the same name.

The rules we have defined consist of basic steps in a transformation. We need to define a strategy to determine where and in what order to apply these rules. In this particular example, we want to apply any of the rules everywhere until none of the rules is applicable anymore. Since we do not have a preference over the reductions rules (the reduction is confluent), we give all the rules the same name. We also do not care the order in which redex are reduced. We pick the outermost strategy from the library which applies its argument to the outermost, leftmost subexpression repeatedly until it is not applicable anymore. It is often more efficient than an innermost strategy since it generally needs fewer reduction steps.
module BoolEval
imports libstrategolib
imports BoolSig

rules
  R: Not(False) --> True
  R: Not(True) --> False
  R: And(False,b) --> False
  R: And(b,Not(b)) --> False
  R: And(True,b) --> b
  R: And(b,True) --> b
  R: Or(True,b) --> True
  R: Or(b,True) --> True
  R: Or(False,b) --> b
  R: Or(b,False) --> b

strategies
  main = io-wrap(eval)
  eval = outermost(R)

Figure 2.7: Rules and Strategies for Evaluating B Expressions BoolEval.str

A Stratego program is compiled using the strc command strc -i BoolEval.str
-la stratego-lib. The -la stratego-lib argument tells the compiler to link
the Stratego library since we used the outermost strategy. The result is an ex-
ecutable file BoolEval. Given an abstract syntax tree as the input, the command
sglri -p Bool.tbl -i Test.b | ./BoolEval executes the main strategy and
produces the output True.
What is Not Covered

We have shown a typical work flow of a transformation system in Stratego. In this trivial example, we have not shown the powerful expressiveness of strategies defined in the library which are a strong point of Stratego. The strategies are important to a program transformation system. We will defer a more detailed account of this aspect until Chapter 5, where we define a library of generic traversal strategies.

2.1.2 ELAN

The next rewriting system we will examine is ELAN [8, 7, 10, 9]. ELAN is similar to Stratego in the sense that it also provides an environment for specifying a term rewriting system based on rules controlled by strategies. The strong point of ELAN is its handling of associative-commutative matching (AC-matching) [18], and strategies to control the non-determinism induced by non-confluent rewrite systems. However, since ELAN lacks a rich set of recursive generic traversal strategies, it does not scale well.

In the following section, we solve the same problem of $B$ expressions using ELAN’s rewriting strategies. The result is an ELAN program bool.eln defined in Figure 2.8.

Similar to Stratego, an ELAN program also begins with a concrete syntax definition and a signature of the abstract syntax. In ELAN, they are combined in one definition. The following declarations define True and False as values of sort bool.
module bool

sort bool;
end

operators global
  True : bool;
  False : bool;
  ! @ : (bool) bool pri 3;
  @ & @ : (bool bool) bool assocLeft pri 2;
  @ | @ : (bool bool) bool assocLeft pri 1;
  ( @ ) : (bool) bool;
end

rules for bool
  b: bool;
  global
  [ ] !False => True end
  [ ] !True => False end
  [ ] False & b => False end
  [ ] b & False => False end
  [ ] True & b => b end
  [ ] b & True => b end
  [ ] True | b => True end
  [ ] b | True => True end
  [ ] False | b => b end
  [ ] b | False => b end
end
end

Figure 2.8: An ELAN Program for Evaluating $B$: bool.eln
\begin{verbatim}
True : bool;
False : bool;
\end{verbatim}

Below we show the definition of an infix conjunction operator \& that takes two bool terms and constructs one bool term.

\begin{verbatim}
@ & @ : (bool bool) bool assocLeft pri 2;
\end{verbatim}

The definition also specifies that the operator is left associative and has a priority of 2. The bigger the number is, the higher priority the operator has. The @ symbol is a placeholder for terms.

Following the signature are the rules, with a one-to-one correspondence to those defined in Figure 2.2. However, we do not need to specify a recursive strategy here because ELAN implicitly uses an outermost recursive strategy. The following is a session with ELAN.

\begin{verbatim}
[] start with term :
    True&!False|!!True|(False&True) end
[] result term:
    True
\end{verbatim}

2.1.3 Maude

The final rewriting system we will examine is Maude [55] developed at the Department of Computer Science, University of Illinois at Urbana-Champaign. The formal background of Maude is rewriting logic [57]. N. Marti-Oliet and J. Meseguer give a comprehensive survey and overview on rewriting logic in [58]. A good entry point is also [59].
We use Maude as a metalanguage [14] to solve the problem of evaluating $B$ by rewriting. Figure 2.9 shows the complete program.

```
mod bool is
  sort bool .
  op True : -> bool .
  op False : -> bool .
  op _&_ : bool bool -> bool [prec 20 gather (E e)] .
  op _|_ : bool bool -> bool [prec 30 gather (E e)] .

  var b : bool .

  rl [R] : ! True => False .
  rl [R] : ! False => True .

  rl [R] : True & b => b .
  rl [R] : False & b => False .
  rl [R] : b & True => b .
  rl [R] : b & False => False .

  rl [R] : False | b => b .
  rl [R] : True | b => True .
  rl [R] : b | False => b .
  rl [R] : b | True => True .
endm
```

Figure 2.9: A Maude Program for Evaluating $B$: bool.maude

Similar to Stratego and ELAN, the program begins by the definition of the signature. The definition is very similar to those in the Stratego and ELAN programs. What is worth mentioning is the _ symbol which serves the purpose of defining the fixity of the operator (similar to @ in ELAN). The prec keyword defines the pri-
ority of the operator. In contrast to ELAN, a smaller number represents a higher priority. The associativity is specified in Maude by the means of gathering pattern. A gathering pattern is given as a sequence of the following possible pattern symbols. Each symbol is a place-holder for an argument to the constructor.

- **E**: The argument must have a precedence value equal to or lower than the precedence value of the operator.

- **e**: The argument must have a precedence value strictly lower that the precedence value of the operator.

- **&**: The operator allows any precedence value for the corresponding argument.

The following code defines an infix operator & with priority 20. In the gathering pattern, the first pattern allows operators of the same precedence. Therefore, & is a left-associative operator.

    op _&_ : bool bool -> bool [prec 20 gather (E e)] .

Following the signature are the rewrite rules. Each rule begins with the keyword rl and a rule name. Similar to the Stratego implementation, we choose to have the same name for all rules.

And that is all we need to do to define the evaluator. We can load the module into the Maude interpreter and issue commands. In our case, we want to use the rewrite command to evaluate a boolean term like the following session.

    Maude> rewrite True & ! False | !! True | (False & True) .
    result bool: True
The **rewrite** command applies the rules using a default strategy, which is an outermost strategy. Different strategies can be defined by rewrite rules at the meta level.

2.1.4 Discussion

We have studied several term rewriting systems: Stratego, ELAN, and Maude. Each has some strong and weak points. Stratego has a rich set of traversal strategies. The characteristic features of ELAN are AC-matching and non-determinism handling. Maude provides a very convenient and intuitive syntax for defining a rewrite logic. ELAN and Maude are less expressive with regard to traversal strategies. They are all candidates for implementing a rewrite system. However, for implementing program updates, such as the size update we presented in Chapter 1, simple rewriting is not enough. Knowledge of the static semantics and the type system of the object language are needed. Inherently, these can not be handled by a rewrite system.

2.2 Language-Based Approaches

As an alternative to updating programs, generic programming, also known as polymorphic programming [31] (in the rest of the dissertation, we use these two terms interchangeably), tries to solve the problem on the language level.

A common practice in functional programming is defining polymorphic func-
tions. A polymorphic function is parameterized over one or more types thus ab-
stracting from the specifics of these types [61, 69]. A polytypic function takes this
idea one step further and is defined by induction on the structure of algebraic data
types [36, 64].

Polytypism differs from both parametric polymorphism and ad hoc polymor-
phism. A typical parametric polymorphic function is the map function that applies
a function to all elements of a list. Here is the definition of map in Haskell.

\[
\text{map} :: (a -> b) -> [a] -> [b]
\]
\[
\text{map} (\text{[]}) = [\text{[]}]
\]
\[
\text{map} f (x:xs) = f x : \text{map} f xs
\]

It works for a family of types, namely, lists whose elements are of any type.

In contrast to a parametric polymorphic function which has one definition, an
ad hoc polymorphic function has many instances, one for each type. An example
is the primitive operator + with this type:

\[
(+) :: \text{Num} \ a \Rightarrow a \rightarrow a \rightarrow a
\]

The type class \text{Num} has many instances, such as \text{Int}, \text{Float}, and \text{Rational}. The +
operator for each of these types has to be implemented independently. It happens
that they all share the same notation.

Coming back to polymorphism, the map function works only for lists. One might
need such a function for each algebraic data type parameterized by one type, such
as a tree. In Haskell, this advanced form of genericity can be expressed using type
classes. The map example leads to the definition of a type class \text{Functor} in Haskell,
which defines a member function `fmap` that works with data types parameterized by one type.

```haskell
  class Functor f where
      fmap :: (a -> b) -> f a -> a b
```

One instance of `fmap` needs to be defined for each data type we want to map. This approach uses ad hoc polymorphism to achieve genericity over different algebraic data types. For data types with two argument types, we would need another type class bifunctor, a two-argument functor [68, 4]. However, with polytypic programming, we can define a `gmap` function that works with any data type. We will examine such a function a little later.

Another approach we will investigate is multi-stage metaprogramming. In this approach, the programmer not only writes programs to solve the problem, but also instructs the compiler to automatically generate code to solve the problem. The compiler runs part of the user’s program (metaprogram) and generates programs. It then compiles the generated program as if they were written by the user in the first place.

In the rest of this section, we will explore two examples of generic programming and one example of metaprogramming.

### 2.2.1 Generic Haskell

Generic Haskell [13] is an extension to Haskell that provides a means of defining generic functions. Generic functions are defined over the structure of types. The
usual way of defining an algebraic data type in Haskell is using a data declaration as in the following example of a tree data type.

\[
data \text{Tree } a = \text{Leaf} \mid \text{Node } a \text{ (Tree } a) \text{ (Tree } a)
\]

This defines a data type Tree \(a\) with type parameter \(a\). The right hand side is an alternation of two constructors for two cases of a binary tree. The first constructor takes no argument while the second one takes three, two of which are Tree \(a\) itself. A Haskell data type can be viewed as a sum of products. But the arity of the sum or product is unbounded. Hinze adopts a simpler model, dealing with only binary sums and products [31]. Each data type is represented in this form by replacing alternatives with the binary sum operator :+: and connecting all fields in a constructor by the binary product operator :*:. A data type Unit is used for the cases where there are no fields. These operators are sufficient to capture the algebraic structure of data types. However, we still need a way to record the names of the constructors and fields. Below is the data type definition for representing Haskell data types:

\[
data a :+: b = \text{Inl } a \mid \text{Inr } b \\
data a :*: b = a :*: b \\
data \text{Unit} = \text{Unit}
\]

For example, the type Tree has the following structure type.\(^1\)

\[
type \text{Tree } a = \text{Unit} :+: (a :*: \text{Tree } a :*: \text{Tree } a)
\]

\(^1\)We intentionally omitted the definitions of Con and Lab to simplify the presentation. Interested readers can find more information in [13].
Generic functions can be defined by giving cases for the structure type constructors and primitive types. An example of the generic map function is given in [13] as follows.

\[
\begin{align*}
gmap \{ | \text{Unit} | \} &= \text{id} \\
gmap \{ | \text{Sum} \ a \ b \ | \} (\text{Inl} \ x) &= \text{Inl} \ (\text{gmap} \ { | \ a \ | } \ x) \\
gmap \{ | \text{Sum} \ a \ b \ | \} (\text{Inr} \ y) &= \text{Inr} \ (\text{gmap} \ { | \ b \ | } \ y) \\
gmap \{ | \text{Prod} \ a \ b \ | \} (x :*: y) &= \\
& (\text{gmap} \ { | \ a \ | } \ x) :*: (\text{gmap} \ { | \ b \ | } \ y)
\end{align*}
\]

The \textit{gmap} function is a generic version of the \textit{map} function that works for any type. The above definition deals with the structure types. To be applicable to values that contain primitive types it needs to be defined for primitive types as well. This task is rather trivial. For example:

\[
gmap \{ | \text{Int} | \} = \text{id}
\]

defines the \textit{gmap} function for \text{Int}. Other primitive types are similar. Now, we can apply \textit{gmap} \{ | \text{Tree} | \} \text{succ} to a \text{Tree} value.

There is indeed an instance of \textit{gmap} for every specific type. The type of such an instance is indexed by the kind\(^2\). For a type \text{t} of kind *, it is obvious that the instance of \textit{gmap} has type \text{t} -> \text{t}, because a value of \text{t} contains no values to be mapped. However, if \text{t} is of kind * -> * (for example, \text{Tree} or \text{list}), the instance of \textit{gmap} takes a function of type \text{a} -> \text{b} and maps \text{a} \text{t} \text{a} to \text{a} \text{t} \text{b}. Similarly, if \text{t} is of kind * -> * -> * (for example, \text{Either}), it takes two functions whose types are \text{a} -> \text{c} and \text{b} -> \text{d}, and maps \text{t} \text{a} \text{b} to \text{t} \text{c} \text{d}. The complete type information for \textit{gmap} is given by the following recurrence:

\(^2\text{Kind} \) basically describes the arity of a type. A non-functional type has kind *, a type constructor with one parameter has kind * -> *, So does the type of a single-parameter function.
type GMap {I k v} s t = s -> t

GMap {I k v} a b -> GMap {I v} (s a) (t b)
gmap {I t::k v} :: GMap {I k v} t t

Here are two examples of the types of the instances of gmap:

gmap {I Int v} :: Int -> Int
gmap {I Tree v} :: (a -> b) -> Tree a -> Tree b
gmap {I Either v} :: (a -> c) -> (b -> d) ->
    Either a b -> Either c d

For types of kind * -> *, gmap basically implements a functor. For types of kind
* -> * -> *, gmap implements a bifunctor. Without generic programming, we
would have to define a type class for each kind, and instantiate for each type.

To define generic functions in Generic Haskell, the programmer needs to adapt
to the view of algebraic data types as sums of products, and deal with the structure
type. These generic functions are usually sophisticated and hard to comprehend.
They scale well up to different data types. But still, they do not solve all the
problems in program updates. Logic and semantic updates of programs cannot be
avoided by using Generic Haskell at all.

2.2.2 Template Haskell

Overview

Template Haskell [76, 54] is an extension to Haskell that supports compile-time
meta-programming. It enables Haskell programmers to generate codes rather than
write them. Template Haskell is analogous to templates of C++ and macros of various languages. However, Template Haskell is implemented in a static type system, that is, all the code generation is done at compile time and generated code is type-checked before run time. Therefore, the philosophy “type-correct programs do not go wrong” applies to Template Haskell programs. Moreover, programmers have full access to the abstract syntax of Haskell, which is defined as ordinary algebraic data types. Therefore, the generated programs are guaranteed to be syntactically correct. Currently, the only compiler supporting Template Haskell is GHC 6.x [26].

The most basic concept in Template Haskell is splicing, denoted by the symbol $. (Note that this is a special syntax form, not to be confused with the binary function application operator $). The splicing symbol is followed immediately by an expression without spaces. The $ says “evaluate at compile time”. The expression following it is evaluated by the compiler and a fragment of program is generated in place of the original expression. Then the compilation of the original program proceeds. In the example we shall see, the expression $(\text{mkLst} 4)$ is a splice. When the compiler sees this expression, it evaluates the expression \text{mkLst} 4. The result of this function call is a Haskell expression. This expression is then used in the place of the original expression $(\text{mkLst} 4)$. The behavior of splicing can be compared to the eval command in some shell languages such as bash and tcsh. The difference is that Template Haskell is typed and the evaluation happens at compile time.

The dual of splicing is templates, which are the interface to representing Haskell
programs. The syntax is placing Haskell programs in *quasi-quote* brackets \([| \ldots |]\). The expression \([| \backslash x \rightarrow x |]\) is of type \(\text{ExpQ}\) and evaluates to a value representing the identity function (rather than the function itself).

However, the quasi-quotations are not enough for constructing arbitrary programs. For instance, if a variable name, or the length of a list, is to be determined by a variable, the quasi-quotations cannot be used. In such a case, we have to construct the program explicitly by calling the smart constructors provided for the algebraic data types for Haskell abstract syntax. Template Haskell provides a family of such constructors, such as \texttt{1am}, which constructs a lambda abstraction. The two styles can be mixed freely, as we shall see in the following example.

An Example

We will exemplify the concepts of Template Haskell through an example. The problem is the classical 8-queen problem, which is defined as placing 8 queens on a \(8 \times 8\) chess board such that no queen threatens any other. In other words, there is exactly one queen in every row and every column, and there is at most one queen in each diagonal line. We represent this problem by a list of 8 integers. Each integer corresponds to one row, and the value is the column position where the queen is placed. For example, list \([1, 5, 8, 6, 3, 7, 2, 4]\) represents the solution shown in Figure 2.10.

We consider a more general \(n\)-queen problem where the board size is \(n \times n\) and we have \(n\) queens to place. A typical approach to this problem is backtracking,
with the help of a stack. However, a brute-force enumeration is easier to both implement and understand if \( n \) is fixed. An imperative programmer probably would write a nested loop of depth \( n \). In Haskell, we can use a list comprehension to solve the problem. The following is the solution for the case \( n = 4 \).

\[
\begin{array}{l}
[ [x_1, x_2, x_3, x_4] \mid \\
x_1 \leftarrow [1..4], \\
x_2 \leftarrow [1..4], \text{and} [\text{nt } 1 \ x_1 \ x_2], \\
x_3 \leftarrow [1..4], \text{and} [\text{nt } 1 \ x_3 \ x_2, \text{nt } 2 \ x_3 \ x_1], \\
x_4 \leftarrow [1..4], \text{and} [\text{nt } 1 \ x_4 \ x_3, \text{nt } 2 \ x_4 \ x_2, \text{nt } 3 \ x_4 \ x_1]
\end{array}
\]

where \( \text{nt} \) is a function for determining if two queens do not threaten each other. It takes the distance between two rows and the column positions of the queens and returns \( \text{True} \) if they threaten each other and \( \text{False} \) otherwise.

\[
\begin{align*}
\text{nt} & : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
\text{nt } n \ x_1 \ x_2 \ | \ x_1 == x_2 & = \text{False} \\
| \ abs (x_1-x_2) == n & = \text{False} \\
| \ otherwise & = \text{True}
\end{align*}
\]
The expression is a list of lists. Each inner list represents a configuration. However, the size of the definition grows quadratically with \( n \), and soon becomes unmanageable and error-prone. For instance, when \( n = 8 \), the definition is as follows:

\[
[ \ [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8] \mid \\
x_1 \gets [1..8], \\
x_2 \gets [1..8], \text{ and } [nt 1 x_1 x_2], \\
x_3 \gets [1..8], \text{ and } [nt 1 x_3 x_2, nt 2 x_3 x_1], \\
x_4 \gets [1..8], \text{ and } [nt 1 x_4 x_3, nt 2 x_4 x_2, nt 3 x_4 x_1], \\
x_5 \gets [1..8], \text{ and } [nt 1 x_5 x_4, nt 2 x_5 x_3, nt 3 x_5 x_2, nt 4 x_5 x_1], \\
x_6 \gets [1..8], \text{ and } [nt 1 x_6 x_5, nt 2 x_6 x_4, nt 3 x_6 x_3, nt 4 x_6 x_2, nt 5 x_6 x_1], \\
x_7 \gets [1..8], \text{ and } [nt 1 x_7 x_6, nt 2 x_7 x_5, nt 3 x_7 x_4, nt 4 x_7 x_3, nt 5 x_7 x_2, nt 6 x_7 x_1], \\
x_8 \gets [1..8], \text{ and } [nt 1 x_8 x_7, nt 2 x_8 x_6, nt 3 x_8 x_5, nt 4 x_8 x_4, nt 5 x_8 x_3, nt 6 x_8 x_2, nt 7 x_8 x_1] 
\]

However, such an expression can be dynamically generated with the help of Template Haskell. Let us try to define a function \texttt{mkLst} of the following type:

\[
\texttt{mkLst :: Int -> ExpQ}
\]

This is a metafunction which takes the number of queens \( n \), and generates an expression which is the solution to the \( n \)-queen problem. The generated expression contains \( n \) local variables whose scope is within the expression. Since we do not need to worry about capturing free variables, we can simply use \( x_i \). We first define a list of names for these variables:

\[
xs = [\texttt{mkName ("x"++show i) | i <- [1..n]}]
\]
The `mkName` function converts a string into a `Name` data type used in the abstract syntax.

For each `x_i`, there are two qualifiers: one generator `x_i <- [1..8]` and one guard `and [nt 1 x_i x_{(i-1)}, ..., nt (i-1) x_i x_{i}]`. The result expression `[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]` follows the qualifiers (remember that a list comprehension is simply syntactic sugar for a `do` expression, so it is in fact the last part in the `do` expression). Therefore, all the qualifiers can be generated from the generators and guards:

```
quals = concat [[ gen i, guard i ] | i <- [1..n]]
```

The generator for `x_i` is

```
x_i <- [1..n]
```

which is a `Stmt` with binding. Smart constructor `bindS` is used to generate such a statement. Given a name, the smart constructor `varP`, generates a pattern variable. Note that `[1..n]` is quasi-quoted because it is a meta-code, which should appear literally in the generated program, but not to be evaluated at stage 1.

```
gen i = bindS (varP (xs!!(i-1))) [] [1..n] []
```

The guard, however, is a non-binding statement. It is essentially an application of `and` to a list of conditions. Each condition tests if the `i`th queen threatens a previously placed queen. The smart constructor is similar to `varP` but it constructs a variable to be used in an expression rather than a pattern. Smart constructor `listE` takes a list of expressions and construct a list expression. Note that the
argument to and needs to be spliced by \([1\ldots i]\) because it is quasi-quoted but need to be evaluated at stage 1.

\[
guard \ i = \text{noBindS} \\
\quad [! \\
\quad \quad (\text{listE} \ [[! \ nt \ k \ $\text{varE} \ (xs!!(i-1))\\n\quad \quad \quad $\text{varE} \ (xs!!(i-k-1)) \] \\
\quad \quad \quad \mid \ k < \ [1..i-1]]) \\
\quad ]
\]

The result \([x_1, \ldots, x_n]\) is a statement without binding, whose body is a list expression:

\[
\text{result} = \text{noBindS} \ [! \ ($\text{listE} \ (\text{map} \ \text{varE} \ \text{xs})) \ ]]
\]

Now we have all the parts and are ready to assemble the expression, which can be constructed by smart constructor \text{compE}.

\[
\text{mkLst} \ n = \text{compE} \ (\text{quals} \ ++ \ [\text{result}])
\]

To get the result, we need to run \text{mkLst} to generate the expression and then evaluate the expression by splicing it:

\[
*\text{Queens}> \ $\text{mkLst} \ 4 \[[2,4,1,3], [3,1,4,2]]
\]

One can also run GHC or GHCi with \texttt{-ddump-splices} option to trace the code generation. We can verify that the code generated is exactly identical to what we manually wrote.

In summary, the \(n\)-queen problem can be defined in Template Haskell as follows:
mkLst :: Int -> ExpQ
mkLst n = compE (quals ++ [result])
  where xs :: [Name]
    xs = [mkName ("x"++show i) | i <- [1..n]]
    quals = concat [[ gen i, guard i ] | i <- [1..n]]
    gen i = bindS (varP (xs!!(i-1))) [i [1..n] []]
    guard i = noBindS
      |
    and $(listE [[] nt k $(varE (xs!!(i-1)))
      $(varE (xs!!(i-k-1))) |] 
    | k <- [1..i-1]])
    []
result = noBindS [i $(listE (map varE xs))][]

Another important aspect of Template Haskell we have not mentioned is reification. Reification is a way of allowing programmers to access the internal data structure of the compiler. For example, if we have a data type declaration:

    data T a = Tip a | For (T a) (T a)

The programmer can obtain the representation of the declaration by using `reifyDecl` function as if it is constructed by the programmer:

    repT :: Decl
    repT = reifyDecl T

One application of reification is to generate boilerplate codes for the data type.

Different States

Template Haskell is a rather complicated system, where the compilation and type checking are interwoven with execution. Several states and levels are involved in
the process of compiling and type checking. Figure 2.11 is borrowed from [76] and shows these states and levels and the transitions.

The most basic state is *compiling* (C), which is the state of normal compilation. The program would be compiled in this state if no metaoperations were involved. A quasi-quotiation drives the compiler into *bracket* (B) state. In this state, programs are being compiled and type checked, but the result is not used by the compiler itself. Rather, the representation of the code is returned. The compiler enters the state *splicing* (S) when it sees the $ symbol and begins evaluating the expression. The evaluated result takes place of the original expression, the compiler restores its original state and proceeds with the generated code as if it was there in the first place. In addition to different states, the compiler needs to count a level to distinguish a top-level splicing and a splicing inside a quasi-quotiation.

![Diagram](image_url)

Figure 2.11: Typing States and Levels for Template Haskell

Template Haskell is an extension to Haskell and provides the programmer an
option to generate Haskell codes by writing such generators. The generation is done at compile time. The compiler can be at different states and levels. Change of states and level are done by bracketing and splicing. This is generally very hard for the programmer to follow. Generally, if the metalanguage and the object language are the same, confusion is very likely to occur. Template Haskell also only provides a means of generating programs, not updating programs. To this end, it does not solve all the problems in software maintenance either.

2.2.3 Scrap-Your-Boilerplate

Lämmel and Peyton Jones proposed a design pattern for traversing functional data structures called the scrap-your-boilerplate (SYB) approach [42]. This approach is targeted at releasing the burden of writing boilerplate codes from programmers. Functional programmers frequently need to traverse data structures to apply functions to certain parts in a term. Boilerplate code refers to the code that deals with walking through the structure without doing the real work. These codes usually account for a big part of the program.

The examples given in [42] are based on a collection of data types that represent a simplified structure of a company. We repeat the definitions here for reference:³

³Derived instances for type classes Typeable and Data are declared but omitted here for clarity.
data Company = C [Dept]
data Dept = D Name Manager [Unt]
data Unt = PU Employee | DU Dept
data Employee = E Person Salary
data Person = P Name Address
data Salary = S Float
type Manager = Employee
type Name = String
type Address = String

Here is an example definition of a company according to the above data types:

genCom :: Company
genCom = C [D "Research" joe [PU mike, PU kate],
               D "Strategy" mary []]

joe, mike, kate, mary :: Employee
joe   = E (P "Joe" "Oregon") (S 8000)
mike  = E (P "Mike" "Boston") (S 1000)
kate  = E (P "Kate" "San Diego") (S 2000)
mary  = E (P "Mary" "Washington") (S 100000)

A simple transformation task is to define an increase function that increases everybody’s salary by a certain percentage. Normally, in Haskell we would have to define one increase function for each individual data type. The only purpose these functions serve is to traverse the data types and move the incS function to the Salary type where it actually increases the salary:

incC :: Float -> Company -> Company
incC k (C ds) = C (map (incD k) ds)

incD :: Float -> Dept -> Dept
incD k (D n m us) = D n (incE k m) (map (incU k) us)
... 

incS :: Float -> Salary -> Salary 
incS k (S s) = S (s * (1+k))

The `incS` function is the only interesting bit. All other code is “boilerplate code”. As the sizes of the data types grows, the boilerplate code becomes extremely clumsy and hard to maintain. It also does not scale up well. Changes to the data type definitions will entail many changes in the boilerplate codes. Also, such codes need to be rewritten for every function that needs to walk through the data structure. The SYB approach tries to eliminate these boilerplate codes.

Type Extension

A key component to the SYB approach is a technique called *type extension*. A function, such as `incS`, is applicable only to a single type (in this case, `Salary`). A type extension applied to this function makes it applicable to any type. The centerpiece is a type class `Typeable` and a related function `cast`. The `cast` function accepts a value of any type, say, `a`, and returns a value of any other type, say, `b`. This function has the following type signature:

```haskell
cast :: (Typeable b, Typeable a) => a -> Maybe b
```

It is totally driven by the types. When `a` and `b` are identical, `cast` simply encapsulates the input with a `Just` constructor. Otherwise, it returns a `Nothing`. 
For example, the following is the result of applying `cast` to the same argument but with different result type:

```
Prelude> (cast 'a') :: Maybe Int
Nothing
Prelude> (cast 'a') :: Maybe Char
Just 'a'
```

Since it is driven by the types, it is necessary to specify the result type of `cast` explicitly. It is not possible for the compiler to infer the type. With the help of `cast`, a `mkT` function can be defined that extends the type of a function. The extended function works exactly the same as the original function if type of the given argument matches the parameter type. Otherwise, it behaves like the identity function that returns whatever the input is. The `mkT` function is defined as follows [42].

```hs
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
mkT f = case cast f of
    Just g -> g
    Nothing -> id
```

Here are some examples of application of `mkT`:

```
Prelude> (mkT (incS 0.1)) (S 100)
110
Prelude> (mkT (incS 0.1)) 'a'
'a'
```

A function constructed by `mkT` is called a *generic function* since it can be applied to any data type.
One-Layer Traversal

The next important element is a type class Data⁴, which extends the type class Typeable, and a higher-order function gmapT that applies a function to all immediate subterms of a value. The function gmapT has the following type:

\[ \text{gmapT :: (forall b. Data b => b ->) -> a -> a} \]

This type is rank-2 polymorphic because the first argument type has a locally universally quantified type variable b. This means that gmapT only accepts generic functions as its first argument. When a generic function \( f \) and an arbitrary term \( C \ t_1 \ t_2 \ldots t_n \), where \( C \) is the constructor and \( t_1 \ldots t_n \) are subterms, is passed to gmapT, the function is applied to all the subterms, the result is obtained by applying the original constructor to the changed terms: \( C \ (f \ t_1) \ (f \ t_2) \ldots (f \ t_n) \). Since \( f \) is a generic function, it can be applied to terms of different types. The application of gmapT on a term is visually explained in Figure 2.12. We do not show the definition of gmapT because it requires knowledge of the function gfold1, which will be introduced in Chapter 5.

\[ \text{gmapT } f \quad C \ t_1 t_2 \ldots t_n \quad \Rightarrow \quad f \ t_1 f \ t_2 \ldots f \ t_n \]

Figure 2.12: An Application of Function gmapT.

---

⁴Originally it was called Term in [42]. It was later then renamed to Data in [43, 44] as well as in Haskell Hierarchy Libraries [27].
Recursive Traversal

The ultimate goal is to carry functions like $incS$ to all places in the term where applicable. This is done by using one-layer traversals as basic building blocks. For example, a function $everywhere$ that applies a function to every subterm of a value is defined mutually recursively with $gmapT$:

$$
everywhere :: Data a => (forall b. Data b => b -> b) -> a -> a$$

$$everywhere f x = f (gmapT (everywhere f) x)$$

The function $everywhere$ $f$ is recursively applied to all the direct subterms of $x$, then $f$ is applied to the top-level term. Thus $f$ is applied to all the subterms of $x$ in a bottom-up order. A top-down counterpart $everywhere'$ can also be easily defined.

The original salary increasing function can be defined with $everywhere$ and $mkT$:

$$increase :: Float -> Company -> Company$$

$$increase k = everywhere (mkT (incS k))$$

Applying $increase$ to the example company data produces the following result.

*Examples*> increase genCom

C [D "Research" (E (P "Joe" "Oregon") (S 8800.0))
   [PU (E (P "Mike" "Boston") (S 1100.0)),
    PU (E (P "Kate" "San Diego") (S 2200.0))],
   D "Strategy" (E (P "Mary" "Washington") (S 110000.0)) []
]

In summary, to utilize the SYB approach, programmers only need to provide the function that does the “real” work, then extend its type by applying $mkT$ function
to it. The result is a generic function that behaves like the original function if its argument type matches. This function is then passed to a recursive traversal. All the boilerplate codes are defined in the function `gmapT` once and for all.

Queries

The examples so far implement *transformations* in the sense that the result has the same type as the input. There is another kind of functions which are called *queries.*\(^5\) Generally, a query is a function that takes a value and returns a value of another type. A typical example of query is collecting all salaries from a company data and sum them up to create a bill. The programmer writes a specific query applicable to a certain type. Type extension is applied to this query, and a generic query is constructed, which is then applied to every subterm of a value.

However, unlike `mkT`, which defaults to the `id` function in case of a type mismatch, the type extension function `mkQ` for queries has nothing to fall back on. Therefore, the programmer needs to provide a default query value to `mkQ` and `mkQ` would have the following type.

\[
\text{mkQ} :: \text{Typeable a, Typeable b} \Rightarrow r \rightarrow (b \rightarrow r) \rightarrow (a \rightarrow r)
\]

Here \(r\) is the type of the query result. When the generic query is applied to a value whose type does not match the argument type of the original query, it returns the default value.

\(^5\)In Chapter 5, a more generally term *accumulation* is used. For now, we will just use “query” as coined by Lämmel and Peyton Jones.
\text{mkQ} \ r \ q \ a = \text{case cast a of}
\begin{align*}
\text{Just } b &\rightarrow q \ b \\
\text{Nothing} &\rightarrow r
\end{align*}

For a recursive query, the user also needs to provide a combinator to glue all
the query result together to form the final result. Therefore, a recursive query
\text{everything} that applies a query to all subterms of a value is defined as follows.

\text{everything} :: \text{Data} \ a \rightarrow
\begin{align*}
(r \rightarrow r \ r) &\rightarrow \\
(\forall b. \ \text{Data} \ b \rightarrow b \rightarrow r) &\rightarrow \\
a &\rightarrow r
\end{align*}
\text{everything} \ k \ f \ x = \text{foldl} \ k \ (f \ x) \ (\text{gmapQ} \ (\text{everything} \ k \ f) \ x)

The query \( f \) is applied to the term \( x \). Recursively, \text{everything} \ k \ f \ is passed to
\text{gmapQ} to be applied to direct subterms of \( x \). The combinator \( k \) is passed to the
left fold function to glue the query result from the subterms and the value itself.

We might be interested in querying more than one data type. In such cases, we
need a \textit{compound type extension} for queries. Given a generic query, we can extend
it with a type-specific query using the \texttt{extQ} function with the following type:

\text{extQ} :: (\text{Typeable} \ a, \text{Typeable} \ b) \rightarrow
\begin{align*}
(a \rightarrow r) &\rightarrow (b \rightarrow r) \rightarrow (a \rightarrow r)
\end{align*}

When the query \( f \) \texttt{`extQ`} \( g \) is applied to a value, it checks the type of the value,
if it matches the argument type of \( g \), then \( g \) is applied, otherwise \( f \) is applied.
Thus we can extend a query by any number of specific queries. Programmers can
write a specific query for each interesting data type and use \texttt{extQ} to combine them.
Similarly, an \texttt{extT} function is defined for extending transformations.
Scrap More Boilerplate

In [43] Lämmel and Peyton Jones extend the original approach by full support of type reflection, providing programmers full access to the data type and constructor information. This information finds its application in functions for computing generic size, generic show, generic serialization/deserialization, etc. We give an overview of some of these applications here without giving the implementation details.

The generic size function counts the number of constructors in a value of arbitrary type. It can be defined as a simple application of gmapQ:

\[
gsize :: \text{Data } a \Rightarrow a \rightarrow \text{Int} \\
gsize t = 1 + \text{sum} \ (\text{gmapQ gsize } t)
\]

It recursively computes the sizes of the direct subterms, sum them and add one to it to account for the top-level constructor.

This leads to a more sophisticated application, generic show. The goal is to take a value and render it as a text representation. To do this, we need to obtain the information of the constructor of the value. This calls for a new member of the type class Data, namely a toConstr function that returns a representation of the constructor of the value. Furthermore, an observer showConstr which has the following type is needed to get the text representation of the constructor.

\[
\text{showConstr} :: \text{Constr} \rightarrow \text{String}
\]

To render the constructor in accordance to its fixity and precedence (to avoid extra parentheses), we also need the following observer:
constrFixity :: Constr -> Fixity

The dual of generic show is generic read, which converts string representations to values. The procedure is the inverse of rendering. A constructor is parsed from its string representation and so are the subterms. We need a function fromConstr that complements the toConstr function.

String representation is human readable, but not efficient. A binary serialization is an encoding of the value in bits. Similar to the generic show, we also get the information about the constructor. To determine how many bits to assign for a constructor, we need to know how many constructors are there in total for the data type. This is given by the function maxConstrIndex. Deserialization is the opposite process.

The SYB approach provides an elegant solution to generic traversal problems. It scales up well to problems involving multiple complex data types. Similar to Generic Haskell, it is intended for defining generic functions. But it is more functionality specific. The SYB approach does not try to solve logic and semantic program updates either. What is more, while it provides a platform for defining generic traversals, it does not provide a library of such traversals. In Chapter 5, we will explore this problem in more detail and propose a solution.

2.3 Practical Systems

In this section, we are going to investigate several practical programming tools, in particular, Eclipse and HaRe, which support refactoring, and Visual Haskell,
which is a Visual Studio plug-in for a Haskell programming environment.

2.3.1 Eclipse

Eclipse [17] is an open source framework written in Java for creating an IDE. This extensible framework is enabled by its plug-in architecture. Figure 2.13 shows the user interface of Eclipse with a Java program opened. The Java part of Eclipse, JDT is able to perform several types of automatic refactorings on Java programs. In this section, we present several refactorings in Eclipse. The examples and screenshots are based on Eclipse 3.1, the latest version as of now (June, 2006).

A refactoring is a program transformation that preserves the behavior of the program. The purpose of refactoring is usually improving the design and restructuring the existing code. Various refactorings have been proposed [25]. Refactoring by hand suffers the problems we discussed in Chapter 1.

In the main menu of Eclipse, there is a “Refactor” drop-down menu. The sub-menus are divided into three sections, corresponding to three kinds of refactorings. The first kind changes the physical structure and layout of the code. The second kind changes the structure on the class level. The third one changes the code structure within a class. These refactorings are listed as follows.

- Physical Structure
  - Rename
  - Move
Figure 2.13: The Eclipse IDE

- Change Method Signature
- Convert Anonymous Class to Nested
- Move Member Type to New File

- Class Level Structure
  - Push Down
  - Pull Up
  - Extract Interface
– Generalize Type
– User Supertype Where Possible
– Infer Generic Type Arguments

• Structure inside a Class
  – Inline
  – Extract Method
  – Extract Local Variable
  – Extract Constant
  – Introduce Parameter
  – Introduce Factory
  – Convert Local Variable to Field
  – Encapsulate Field

Let us examine a simple refactoring “Rename”. As the name indicates, the “Rename” refactoring renames a Java element. It changes all the references as well. To rename a Java element, select the element in either the Package Explorer view, the outline view, or the Java source file, then select “Rename” from the “Refactor” menu. A dialog will appear asking you the new name and whether to update the references to the element. You can also preview before the actual changes by clicking on the “Preview” button. In the example shown in Figure 2.14, a function f is renamed to foo, and the dialog previews the changes to the definition as well as the references.
Let us look at a slightly more involved refactoring “Extract Method”. If some code appears more than once in a program and implements some logically independent functionality of its own, it is a good indication that this code should be generalized and made a separate method. The refactoring “Extract Method” does just that. In the source file, the user selects a chunk of code and invokes the refactoring. In the dialog shown in Figure 2.15, the user gives the new name, access modifier, and parameters. The variables used but not defined in that chunk of code are detected by Eclipse and used as default parameters for the method.

If the user clicks on the “Preview” button, a preview window will show up
highlighting the changes to the original program, as shown in Figure 2.16.

These refactorings in Eclipse can be very useful for Java programmers. However, they can sometimes introduce type and scoping errors. Consider the following imaginary code snippet.

```java
int i;
while (true)
{
    boolean i;
    i = true;
}

i = 1;
```
Select the last occurrence of the variable i and invoke “Rename” refactoring to change it to j. Eclipse will produce the following code.

```java
int j;
while (true)
{
    boolean i;
    j = true;
}

j = 1;
```

The i inside the while loop is incorrectly renamed by the refactoring producing a type error. It failed to realize that the inner i is locally defined and overrides the
outer i.

Another example where refactoring fails is with extracting methods. Consider the following function definition.

```java
private int f(boolean b)
{
    int n = 0;
    int i = 0;
    // from
    if (b)
        i = 1;
    if (b)
        n = n + i;
    // to
    return n;
}
```

Select the code between “from” and “to” and apply “Extract Method”, the following code is the result.

```java
private int f(boolean b) {
    int n = 0;
    int i = 0;
    n = newMethod(b, n);
    return n;
}

private int newMethod(boolean b, int n) {
    int i;
    if (b)
        i = 1;
    if (b)
        n = n + i;
    return n;
}
```
Although the above code can pass the compiler, it contains a serious problem. The variable `i` should be passed as a parameter to `new Method` rather than being a local variable.

To summarize, Eclipse provides a collection of practical refactorings for Java programmers. While they ensure syntax and type correctness on programs with simple structures, they fail on more involved programs.

2.3.2 HaRe

The Haskell Refactorer [49, 50, 51] is work in progress at the University of Kent. It is a refactoring tool integrated with text editors (Emacs and Vim) that supports the full Haskell 98 language. Similar to Eclipse, HaRe can perform a set of refactorings. The catalog on the project homepage [49] gives a list of examples. However, not all of them are currently implemented. The following are some of the implemented refactorings.

- Names/Scopes

  - Renaming
  - Lifting to top level
  - Lifting one level
  - Demoting

- Definitions
- Introducing new definition
- Unfolding definition
- Generalizing definition
- ...

Again, we examine a simple refactoring “Rename”. The rename refactoring changes the name of a given Haskell identifier. It changes all the places where the identifier is referenced. One places the cursor on the identifier to be renamed and activates the refactoring from the menu of Emacs or Vim. The changed program will be reloaded. The following shows a comparison of an original program and the changed program.

\[
\begin{align*}
f 0 &= 1 & \text{foo } 0 &= 1 \\
f n &= n \times f \ (n-1) & \text{foo } n &= n \times \text{foo } (n-1) \\
h \ y &= f \ y & h \ y &= \text{foo } y
\end{align*}
\]

The reference to \( f \) in \( h \) is renamed accordingly. However, if \( f \) is overridden by a local definition, HaRe is smart enough not to capture it, as shown in the following example.

\[
\begin{align*}
f 0 &= 1 & \text{foo } 0 &= 1 \\
f n &= n \times f \ (n-1) & \text{foo } n &= n \times \text{foo } (n-1) \\
h \ y &= f \ y & h \ y &= f \ y \\
& \quad \text{where } f = \text{id} & \quad \text{where } f = \text{id}
\end{align*}
\]

Another refactoring, “Generalizing”, extracts a subexpression and add a formal parameter to a function.
\[ f \, v = v + 1 \quad f \, w \, v = v + w \]

\[ e = f \, 3 \quad e = f \, 1 \, 3 \]

HaRe exposes an API to its infrastructure so that new refactorings or general program transformations can be implemented. This API contains functions for accessing abstract syntax for Haskell 98. This provides a candidate platform for implementing source-to-source Haskell transformations.

HaRe was started after we started the work in this dissertation and is not stable as a product yet. That is one reason why it is not used as a platform for implementation, although it is a good candidate.

2.3.3 Visual Haskell

Visual Haskell [3] is a Haskell programming environment based on Microsoft's extensible Visual Studio environment. It integrates with the Visual Studio editor to provide interactive features to Haskell program development, such as syntax highlighting, syntax and static semantic checking, word completion, etc. Because the development environment is extensible, it is a good candidate platform for implementing the Haskell Update Language we shall propose in Chapter 4.

Figure 2.17 shows the Visual Studio environment with a Haskell project loaded. The Haskell keywords, such as `module`, `where`, and `do`, are colored blue. Other lexical units are also colored according to their lexical category. Coloring happens on-the-fly as you type. Unlike the Haskell mode for Emacs, which does syntax highlighting using regular expressions, Visual Haskell has access to the underlying
compiler GHC and does a more accurate job.

The editor has access to the type checking results and can provide type information for identifiers. Hovering the mouse over an identifier causes popping up a tooltip with the type information, as shown in Figure 2.18.

The editor also checks the source for syntax errors as well as semantic errors, such as type error and scoping error. For instance, when we intentionally mistype the function putStrLn, the editor produces an error message `Not in Scope: 'putStrLn'`, as shown in Figure 2.19. Figure 2.20 shows another type error, which is
checked on the fly. In the function definition of \texttt{foo}, a parameter is missing causing the type error.

Since the editor has the source code constantly typechecked, it always provides the user with the up-to-date information. Word completion is another important feature in the Visual Studio environment, Visual C++, Visual Basic, as well as...
Visual Haskell. The word completion feature incrementally searches for a name in scope as the user types. When it is activated, it is very likely that the program is not complete and therefore the compiler will report errors. Therefore, it is a fairly difficult task to include all the locally bound names. Visual Haskell rather includes only the top-level names in the word completion.

Currently, Visual Haskell only provides a visual developing environment for Haskell programmers, with limited source code level editing aid. It aims to add refactoring functionality in the feature [3].
Chapter 3 – Monadification

Monads can be used to structure and modularize functional programs. The monadic form of a functional program can be exploited for quite different purposes, such as compilation or integration of imperative features. Despite the usefulness of monads, many functional programs are not given in monadic form because writing monadic code is not as convenient as writing other functional code. It would therefore be helpful to have tools for converting non-monadic programs into monadic form, a process that we call *monadification*. In the context of this dissertation, monadification represents an important class of program updates.

Monadification-like algorithms for the purpose of compilation have been known for some time now [24, 28, 29, 30]. The idea of these algorithms is to transform *all* functions in a program into monadic form. While these algorithms are quite simple and work well for their purpose, they are not well suited for the introduction of monads at selected places in a program. On the other hand, Lämmel has identified the selective introduction of monads into functional programs as a useful source code transformation [39]. He gives a specification of monad introduction for lambda calculus through a structured operational semantics.

In this chapter we present the first detailed treatment of an algorithm for the selective introduction of monads into functional programs. We identify correctness
criteria for monadification and investigate the correctness of the presented algorithm. We also demonstrate the principal limitation of all known monadification algorithms.

In the rest of this chapter, we will first examine the problem of monadification, how it is done by hand, and why it is important (Section 3.1). In Section 3.2 we will look at an example to illustrate how a program can be monadified in several different ways, and we will observe a common procedure in these different ways. In Section 3.3, we formalize the definition of monadification, define an object language and examine some programs in the object language that can be monadified. These programs reveal different aspects of the problem. A well-defined algorithm is then presented in Section 3.4. An analysis of the correctness of the algorithm is performed in Section 3.5. The problem of monad infestation (referring to the problem that when a monad is introduced in part of a program, a large part of the program will be affected) is addressed in Section 3.6, and a solution is proposed. In Section 3.7, several different possible solutions for adding monadic actions are proposed. Two common alternative approaches to monadification are presented and compared in Section 3.8. Finally, conclusions are presented in Section 3.9.

3.1 Introduction

Monads provide a standardized way to integrate a variety of language features into functional languages, such as I/O interaction, state-based computation, or exception handling [66].
The notion of monad originates in category theory [56]. Eugenio Moggi [62, 63] used monads to structure semantics definitions, which paved the way for using monads in functional languages [92]. An excellent survey is given by Phil Wadler in [93]. A more detailed introduction to monads is presented in Appendix C of this dissertation.

As an example for monadification, we consider the task of adding exception handling code to a function definition. Consider the following simple expression data type and a corresponding evaluation function, which we have borrowed from Richard Bird’s book [6, Chapter 10].

```haskell
data Expr = Con Int
  | Plus Expr Expr
  | Div Expr Expr

eval :: Expr -> Int
eval (Con x)  = x
eval (Plus x y) = eval x + eval y
eval (Div x y) = eval x 'div' eval y
```

One limitation of the shown definition of `eval` is that it does not handle exceptions. For example, when `eval` is applied to the argument `Div (Con 1) (Con 0)`, a runtime error will occur. In order to capture such exceptions, `Int` values can be wrapped by the `Maybe` monad, which is explained in more detail in Appendix C. In Haskell, the `do` notation is provided as a convenient syntax for monadic programming. Expressions using `do` are translated into calls to the monadic functions `return` and `>>=`. The translation is also explained in Appendix C.

We want to use the `Maybe` type in the `eval` function in the following way.
Whenever a computation can be performed successfully, the corresponding result value is injected into the `Maybe` type by wrapping a call to `return` around it. On the other hand, any erroneous computation should result in the `Nothing` constructor.

This strategy has an important implication on the definition of `eval`. First of all, the result type of `eval` changes from `Int` to `Maybe Int`. As a consequence of this, the results of recursive calls to the function `eval` cannot be directly used anymore as arguments of integer operations, such as `+` or `div`. Instead, we have to extract the integer values from the `Maybe` type (if possible) or propagate the `Nothing` constructor through the computation. Doing this “by hand”, that is, by explicit pattern matching of all `Maybe` subexpressions in `eval` with `case` expressions can become extremely tedious for larger programs. At this point the fact that `Maybe` is defined as an instance of the `Monad` class comes into play: the monad performs the unwrapping of values and propagation of `Nothing` automatically through the function `>>=`. However, this function has to be placed in `eval` at the proper places to make the monadic version of `eval` work. The (changed) types of the involved objects more or less dictate how this has to be done. In short, all recursively computed values have to be bound to variables that can then be used as arguments of integer operations—this binding process is the inverse operation to the wrapping performed by `return`.

The monadified version of `eval` is given below using the do notation [6]. In this chapter we use the naming convention to append an `M` to names of monadified functions.
evalM :: Expr -> Maybe Int
evalM (Con x) = return x
evalM (Plus x y) = do i <- evalM x
                     j <- evalM y
                     return (i+j)
evalM (Div x y) = do i <- evalM x
                     j <- evalM y
                     if j==0 then Nothing
                     else return (i `div` j)

The process of eval’s monadification consists of two parts: First, the Maybe monad is employed to hide the error status. Second, the adaptation in the last two lines in the above code catches the exception and correctly sets the error status. The first change should preserve the type correctness and the semantics of eval. The second change, the introduction of actions, changes the semantics, but does not change the types.

The advantage of evalM over eval is its proper handling of divide-by-zero errors, which do not cause runtime errors anymore.

However, not only are monads difficult for beginners, they are awkward for experts, too—for example, they force the programmer to specify evaluation order. Therefore, even experienced Haskell programmers often begin the development of a functional program by writing non-monadic functions and later changing it to monadic form. In other situations, monads are added to functions only temporarily, for example, for debugging purposes or to implement other tracing functionality. These monads are often to be removed later from the program. In any case, turning one or more functions into monadic computations is a frequently occurring task for functional programmers.
A tool for the automatic monadification of functional programs gives programmers the freedom to select monads on demand; they are not required to adopt a monadic (and thus more imperative) style from the start. This freedom means an important aid for the development process of functional programs, because a programmer does not have to worry about extensions of his or her program that might require a monadic structure for some of the functions. Monadification was also proposed in [91], but there it is assumed that programmers resort to the traditional approach for the adaptation by using a text editor. In [52] interpreters are written in a particular monadic style that facilitates the extensibility by new features. A drawback of this approach is that the program has to be written in a monadic style right from the beginning.

Automatic monadification has several advantages over manual monadification:

1. **Reliability.** Since an automatic monadification operator works on the abstract syntax level, no syntax errors can be introduced. Moreover, if the monadification operator is well designed and implemented, type correctness of the resulting program can be also guaranteed. The proper design of the monadification operator is the main contribution of this chapter.

2. **Reusability.** We may need to monadify different functions with the same monad. For example, various functions that need to manipulate integer values may all raise divide-by-zero exceptions. We can use the same monadification program for adding exception handling repeatedly.

3. **Versatility.** A function can be monadified with different monads, producing
different functions for different purposes.

4. *Efficiency of Transformations.* Using a tool to perform repeated monadification tasks is also much faster than performing all the required changes with a text editor.

From a more general point of view, it has been observed that monads can support aspect-oriented programming [60]. Aspect-oriented programming is concerned with adding functionality to a program that is orthogonal to the program’s functional decomposition [1, 19]. One example is exception handling that cannot be localized in one module, but spreads through the whole program. A program that is given in monadic form is prepared for the addition of aspects because the aspects can be realized through the definition of the monad operations; the monadic program structure ensures that the aspects are executed at the appropriate places in the program (these are called “join points”). One criticism of this approach to aspect-oriented programming has been that a programmer has to provide the join points explicitly and in advance, by providing the monadic program structure, which compromises much of the benefits of aspect-oriented programming [38]. With the proposed monadification this limitation of the monadic approach can be eliminated. To some degree, monadification can detect join points automatically.

Examining the `eval` and various other examples of monadification (see, for example, [6, 92]), we can observe that monadification is mostly a mechanical process that can be described by a systematic change of the source program. Such a transformation can be captured by the definition of a monadification operator,
which can be implemented, for example, with a metalanguage or as a stand-alone tool. This monadification operator should preserve syntax and type correctness of the transformed program. Moreover, monadification should change the program as little as possible and as much as needed, that is, the monadified program should behave similarly to the original program, only those parts that should be changed/improved by the introduction of the monad should expose a different behavior. For example, the monadification of eval did not monadify the + or div operation, because the goal was to adapt the behavior of eval and not that of other, predefined operations.

It is exactly this latter condition that makes the monadification a non-trivial program transformation.

3.2 Examples

By applying different monads and inserting different monadic actions, the evaluator can be monadified in different ways, exhibiting various behaviors fitted for different applications. In this section, we will demonstrate some of these monadifications taken from [6, Chapter 10].

3.2.1 Exception Handling

Earlier in this chapter, we have seen how the evaluator is monadified by a Maybe monad to realize a simple form of exception handling. The Maybe monad simply
regards all errors as the same, without any information about the error. This is
sometimes not enough. The following Exc monad associates an error message with
the error state, which can be shown to the user.

    data Exc a = Raise String | Return a

The monad instance for Exc is similarly to Maybe, where an exception is indicated
by Raise and propagated through the whole computation.

    instance Monad Exc where
        return x               = Return x
        (Raise x) >>= _        = Raise x
        (Return a) >>= f      = f a

With the Exc monad, we can now represent the divide-by-zero error by associating
a string with the Raise constructor.

    evalM :: Expr -> Exc Int
    evalM (Con x)      = return x
    evalM (Plus x y)   = do i <- evalM x
                            j <- evalM y
                            return (i+j)
    evalM (Div x y)    = do i <- evalM x
                            j <- evalM y
                            if j==0 then Raise "divide by 0"
                              else return (i `div` j)

3.2.2 Counting Operations

We will consider a state transformer monad that takes an initial state and returns
a value paired with the final state. A simple example of a state transformer is an
integer counter, whose state is represented by an integer. The definition of a state transformer is as follows.

```haskell
data ST s a = Trans (s -> (a,s))
instance Monad (ST s) where
  return x = Trans (\s -> (x,s))
  (Trans c) >>= f = Trans (\s -> let (x,s') = c s
                                Trans d = f x
                                in d s')
```

The expression `return x` produces a state transformer that returns `x` as the value and leaves the state unchanged. The expression `(Trans c) >>= f` applies the first state transformer to the initial state `s`, yielding an intermediate value/state pair `(x, s')`; then it applies the state transformer `f x` to `s'`. We need an operation that increases the counter by 1 and returns no value.

```haskell
tick :: ST Int ()
tick = Trans (\c -> ((),c+1))
```

To count the number of operations performed during the evaluation of an expression, we take the original evaluator, monadify it with `ST`, and insert the function `tick` just before the application of each arithmetic function to implement the counting.
evalM :: Expr -> ST Int Int
evalM (Con x) = return x
evalM (Plus x y) = do i <- evalM x
                    j <- evalM y
                    tick
                    return (i+j)
 evalM (Div x y) = do i <- evalM x
                    j <- evalM y
                    tick
                    return (i 'div' j)

3.2.3 Producing Output

In the examples we have seen so far, the monadification was done in a similar way: take the original program, which is not monadic, wrap the return value with a call to return, and potentially add some local “action” before return. The following example deviates from this schema in that it requires access to variables that are not defined near to the location of the action. First, we define an Out monad that couples an output string with the result value. The string is to be printed at the end; we have to thread all the output strings through the whole computation.

    data Out a = Out (String,a)
    instance Monad Out where
        return x = Out ("",x)
        Out (s,x) >>= f = Out (s',x') where Out (s1,x') = f x
                                    s' = s++s1

The call return x couples the value x together with an empty string. The call Out (s,x) >>= f applies f to x, returns its result value, and appends the output to s.
A basic operation on Out is to add a string to the output and return no value.

```haskell
out :: String -> Out ()
out x = Out (x,())
```

To add an execution trace to the evaluator, the original evaluator is modified with Out. The code for tracers is also added.

```haskell
eval :: Expr -> Out Int
eval (Con x) = do out (show (Con x) ++ "=" ++ show x)
    return x
eval (Plus x y) = do  i <- eval x
        j <- eval y
        out (show (Plus x y) ++ "=" ++ show (i+j))
        return (i+j)
eval (Div x y) = do  i <- eval x
        j <- eval y
        out (show (Div x y) ++ "=" ++ show (i ‘div‘ j))
        return (i ‘div‘ j)
```

The interesting aspect of this example is that the changes required for the original program are not local in the sense that they cannot be achieved by just adding a context-independent expression before return. Rather, the inserted expressions need to refer to the parameter of the function, which makes an automatic transformation more challenging.

### 3.3 The Essence of Monadification

Monadification is a source-level program transformation. The correctness of monadification can be characterized by the two notions of soundness and completeness.
Ideally, the behavior of a monadified function is identical to that of the original function, except that the return value is wrapped by a monad; in cases when the original function does not terminate with a result, the monadified function should not terminate either. These requirements can be formally expressed by referring to a semantics definition of the language to be transformed. To this end we assume an operational semantics that defines the reduction of expressions as a binary relation $\rightarrow$, whose reflexive, transitive closure is denoted by $\Rightarrow$. Such a semantics is given, for example, in [66] where the semantics of monadic operations is a particular focus.

The correctness requirements can be formalized as completeness and soundness properties as follows.

\[ \hat{f} \text{ is called a complete monadification of } f \text{ if} \]

\[ \text{return } (f\ x_1\ldots x_k) \Rightarrow y \implies \hat{f} x_1\ldots x_k \Rightarrow y \]

\[ \hat{f} \text{ is called a sound monadification of } f \text{ if} \]

\[ \hat{f} x_1\ldots x_k \Rightarrow y \implies \text{return } (f\ x_1\ldots x_k) \Rightarrow y \]

Soundness is a very strong criterion, which actually might not be always desirable in practice, in particular, when monadic actions are used. For example, the monadified function \texttt{evalM} from Section 3.1 is \textit{not} a sound monadification of \texttt{eval} because it does return a value for $y = 0$ while \texttt{eval} does not.

Before we develop our monadification algorithm, we discuss a general limitation
of monadification. If we define a mapping $\mathcal{M}$ that maps each function $f$ (of $n$
parameters) and a $k \leq n$ to its sound monadification $\hat{f}$ for $k$, we can observe that
$\mathcal{M}$ is a partial function.

**Theorem 1** There are functions for which it is impossible to find a sound monadification for arbitrary arguments.

Consider, for example, the task of monadifying the following function for one parameter.

$$f :: \text{Int} \to \text{Int} \to \text{Int}$$

$$f \; x \; y = \text{if} \; x==0 \; \text{then} \; y \; \text{else} \; f \; y \; (x-1)$$

What we need is a function definition of the following form:

$$fM :: \text{Int} \to m (\text{Int} \to \text{Int})$$

$$fM \; x = e1$$

We have to replace $e1$ by an expression of type $\text{Int} \to \text{Int}$. Of course, we cannot
use an arbitrary expression of that type because we want the resulting function
definition to behave like the original function except for the monad. Therefore,
the defining expression of the function must be somehow retained. The expression

$$\text{return} (\\lambda \; y \to \text{if} \; x==0 \; \text{then} \; y \; \text{else} \; f \; y \; (x-1))$$

works well except that it ignores the monad in the recursive call to $f$. We cannot
simply replace $f$ by $fM$ because $fM \; y$ has a monadic type (and not a function type)
and can therefore not be applied to $(x-1)$. The only way to make the recursion
work is to bind $fM \; y$ to a variable, say $g$, to obtain access to the integer function
of the monadic value. Therefore, we have to replace \( f \ y \ (x - 1) \) by an expression of the form \( \text{do} \ {g \leftarrow f \ M \ y; \ e2} \). We encounter this problem: whatever we try to substitute for \( e2 \), we obtain a monadic value for the whole \( \text{do} \) expression, which can never match the type \( \text{Int} \) of \( y \).

This example shows that monadification is necessarily a partial operation that does not work for all input and that any monadification algorithm will fail for cases like the one shown.

To prepare for the definition of a monadification operator, we will identify the rules that govern correct monadification by considering a number of small examples. The goal of monadification is to transform a given function \( f \) of type \( t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k \rightarrow t \) into a function \( \hat{f} \) of type \( t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k \rightarrow m \ t \), where \( m \) is a monad type constructor. Note that \( t \) is an arbitrary type and can be, in particular, a function type, such as \( t_{k+1} \rightarrow t_{k+2} \rightarrow \ldots \rightarrow t_n \). In other words, the notion of “return value” is relative in the presence of higher-order functions, that is, a multi-parameter function can be considered to have more than one “return type”. For instance, a function of type \( t_1 \rightarrow t_2 \rightarrow t_3 \) can be considered to return values of type \( t_3 \) or \( t_2 \rightarrow t_3 \).

Therefore, monadification can be performed on different return types, which means that the specification of the monadification of a function requires in addition to the function’s name the number \( k \) of parameters that are not part of the monadified result type.

We consider as an object language lambda calculus extended by \texttt{case} expressions and \texttt{let} expressions. The syntax is defined in Figure 3.1.
\[
\begin{align*}
p &::= c \mid v \mid pp \\
e &::= c \mid v \mid \nu \rightarrow e \mid e \mid \text{let } v = e \text{ in } e \mid \text{case } e \text{ of } \{ p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n \}
\end{align*}
\]

Figure 3.1: Syntax of the Object Language

For syntactic convenience we make use of the do notation, which is described in Appendix C.

Next we will examine several examples to better understand how to monadify functions in different situations.

The first example demonstrates the notion of a return expression, which is an expression that is subject to being wrapped by the monad operation return.

\[
f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
f = \lambda x \rightarrow \lambda y \rightarrow x+y
\]

If we consider \( f \) as a two-parameter function, after stripping off two lambda abstractions, \( x+y \) is the expression that defines the result. The most direct way to monadify the function is to wrap a call to return around the return expression.

\[
fM :: \text{Monad } m \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow m \text{ Int}
fM = \lambda x \rightarrow \lambda y \rightarrow \text{return } (x+y)
\]

The body of \( f \) could be of any syntactic form. It might be the case that the lambda abstractions are embedded in other syntactic structures, such as case expressions or applications. Here is such an example where lambda abstractions are embedded in a case expression.

\[
f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
f = \lambda x \rightarrow \text{case } x \text{ of }
\quad 0 \rightarrow \lambda y \rightarrow y+1
\quad n \rightarrow \lambda z \rightarrow z-1
\]
The definition of \( f \) contains two return expressions: \( y+1 \) and \( z-1 \). To monadify this function, \texttt{return} should be applied to both of them.

\[
fM :: \text{Monad } m \Rightarrow \text{Int} \to \text{Int} \to m \text{ Int}
fM = \lambda x \to \text{case } x \text{ of}
0 \to \lambda y \to \text{return } (y+1)
n \to \lambda z \to \text{return } (z-1)
\]

Moreover, a function can be defined in terms of other functions, or be the result of an application. In these cases, the number of parameters to the function does not match the number of lambda abstractions in the function definition. For example:

\[
f :: \text{Int} \to \text{Int}
f = (\lambda x \to \lambda y \to x+y) \; 0
\]

The syntactic structure of this one-parameter function is an application instead of a lambda abstraction. In this form, there is no return expression to apply \texttt{return} to. However, the above definition is \( \eta \)-equivalent to the following definition.

\[
f' = \lambda z \to (\lambda x \to \lambda y \to x+y) \; 0 \; z
\]

After the return expression has been exposed, it can now be monadified in the usual way.

\[
fM :: \text{Monad } m \Rightarrow \text{Int} \to m \text{ Int}
fM = \lambda z \to \text{return } ((\lambda x \to \lambda y \to x+y) \; 0 \; z)
\]

Alternatively, we can move monadification into the function of an application while increasing the number of lambda abstractions to be crossed by 1. This approach leads to simpler code. For the above example we obtain the following definition.
\[
fM :: Monad m \Rightarrow Int \to m Int
fM = (\x \to \y \to return (x+y)) \ 0
\]

Monadification is more complicated in the case of recursive function definitions, because the corresponding recursive calls change their types. Not properly handled, these subexpressions would introduce type errors. Let us consider a simple example.\(^2\)

\[
f :: Int \to Int
f = \n \to n*f (n-1)
\]

If we simply wrap a `return` around the return expression `n*f (n-1)`, the result `return (n*f (n-1))` is not type correct since the type of `f (n-1)` is `m Int` and not `Int`, which is required for the application of `*`. The solution is to bind the expression `f (n-1)` to a variable, say `x`, and use `x` in place of `f (n-1)`.

\[
fM :: Monad m \Rightarrow Int \to m Int
fM = \n \to do \{ x \leftarrow fM (n-1); return (n*x) \}
\]

Still, this is not a complete solution. An expression being bound and lifted out may contain local variables, which will become free variables after the lifting. This problem case can be exemplified by the following function, in which a local variable `n` is introduced by the second alternative of the `case` expression.

\[
f :: Int \to Int
f = \x \to case x of
  0 \to 1
  n \to n*(f (n-1))
\]

---

\(^2\)This function, as some other examples that appear in the rest of the chapter, does not terminate. However, this aspect is not really relevant because the definition could be easily changed into a terminating one by adding a `case` expression. To reveal the essential structure, we use the simpler non-terminating forms instead.
Since the scope of n is limited to the second body of the `case` expression, we should be careful not to lift f (n-1) outside that scope. In this case, the solution is to move the monadification into all branches of the `case` expression. Another reason for not lifting the expression f (n-1) is that in the original program, it is evaluated only when the second alternative of the `case` is matched. Lifting might cause this expression to be evaluated more than necessary, which increases the strictness of the program.

```haskell
fM :: Monad m => Int -> m Int
fM = \x -> case x of
  0 -> return 1
  n -> do {y <- fM (n-1); return (n*y)}
```

Since all bodies of a `case` expression have the same type as the type of the whole expression, operations on the `case` expression can be simply moved down to the bodies.

Scoping problems can also be introduced by lambda abstractions because in a lambda abstraction the type of the body differs from that of the whole expression by an “arrow”. Consider the following function.

```haskell
f :: Int -> Int
f = \n -> (\x -> n*(f x)) (n-1)
```

In this example, the return expression is (\x -> n*(f x)) (n-1), the recursive call f x needs to be lifted and bound. But the scope of x is within the lambda abstraction. Here, we can monadify the anonymous function (\x -> n*(f x)) and change its type from Int -> Int to Int -> m Int.
\[
\begin{align*}
\text{fM} & : \text{Monad m} \Rightarrow \text{Int} \rightarrow \text{m Int} \\
\text{fM} & = \backslash n \rightarrow (\backslash x \rightarrow \{ y \leftarrow \text{fM} x; \text{return} (n*y) \}) (n-1)
\end{align*}
\]

In summary, three kinds of operations are involved in monadifying a function definition:

- **Navigating.** Locate the return expressions in the function definition. The basic approach is to move down \( k \) lambda abstractions. Navigating might be taken down into case expressions. Whenever we cannot find enough lambda abstractions, we use \( \eta \)-expansion to create additional abstractions.

- **Binding.** After locating return expressions, we identify recursive calls and bind them to (fresh) variables. Then we replace the recursive calls with these variables.

- **Wrapping.** After having removed recursive calls from return expressions, we apply return.

It is worth mentioning that if two or more recursive calls exist in a return expression and are to be bound to variables, we can choose different orders of binding. For a purely functional program, this is not a concern because the order does not affect the semantics, although it might have an impact on the efficiency in the case of parallel execution. But the effects in the underlying monad often depend on the order. If the user wishes to add extra actions to the program, the order in which variables are bound does matter. The implementation of the algorithm can either choose a predefined scheme or be parameterized to leave the choice up to
the user. In any case, the user should take into consideration the relevant order.
Two standard strategies for ordering bindings resulting from recursive calls are
preorder and postorder, that is, arrange the bindings according to order in which
the corresponding recursive calls have been encountered in a preorder or postorder
traversal of the expression to be transformed. Since we are monadifying a group
of functions with one and the same monad, it does not seem to be required to offer
a different choice of order for different functions because the appropriate ordering
is probably determined by the monad and not by the functions using the monad.
We believe that such a choice should be a global parameter and not a parameter
that affects individual monadification steps. Therefore, we will not parameterize
the monadification algorithm presented in the next section by this aspect.

3.4 Monadification Algorithm

The general scenario is to monadify not just one, but a set of functions. These
functions are monadified simultaneously so that all their definitions are navigated
and calls to any of these functions will be bound. The function definitions in a
program can be grouped into three categories:

- functions to be monadified

- functions whose definitions contain calls to monadified functions, but which
  should have been originally not monadified

- functions that are not affected by monadification
The first set of functions is selected by the user. This selection then determines the remaining two function sets. A problem with functions in the second group is that monadification destroys type correctness because calls to monadified functions return values of type m.t in contexts where values of type t are expected. There are two ways to deal with this problem. First, functions could be moved into the first set until the second set becomes empty. Second, values can be extracted from monads at all call sites. This approach is discussed further in Section 3.6. In this section we focus on the monadification of set of functions and assume for simplicity that the second group of functions is empty. Therefore, we view a program $P$ as a collection of $n$ function definitions that are to be monadified plus a set of definitions $P'$ that are not affected by the monadification.

$$P = \{f_1 = e_1, \ldots, f_n = e_n\} \cup P'$$

Our goal is to define an operator $\mathcal{M}$ that is applied to each function definition and yields the corresponding monadified version. To uniquely identify the result type to be monadified, $\mathcal{M}$ needs for each function the number of its parameters, which has to be ultimately provided by the user. The function definitions together with the parameter information is called the monadification context.

$$F = \{(f_1 = e_1, k_1), \ldots, (f_n = e_n, k_n)\}$$

We refer to the components of the $i$th context element by $f_i^F$, $e_i^F$, and $k_i^F$, respectively.
\( \mathcal{M} \) is applied to \( P \) and yields the following monadified program \( \hat{P} \).

\[
\hat{P} = \mathcal{M}(P) = \{ \hat{f}_1 = \mathcal{M}_F(e_1), \ldots, \hat{f}_n = \mathcal{M}_F(e_n) \} \cup P'
\]

In this section, we are only concerned about the refactoring aspect of monadification, that is, we ignore the insertion of monadic actions. We will address this issue later in Section 3.7. We make the following assumption for the function definitions of \( f_1, \ldots, f_n \): any call \( f_i^F \) is always applied to at least \( k_i^F \) arguments where \( k_i^F \) is the number of parameters for \( f_i^F \), that is, there is no partial application of \( f_i^F \) that leaves calls to \( f_i^F \) “undersaturated” with arguments. This condition can be checked through the predicate \( \forall i, j.F(e_j^F, 0, f_i^F, k_i^F) \) where the judgment \( F(e, i, f, j) \) denotes the fact that in a context where \( e \) is applied to \( i \) arguments, all references to \( f \) are applied to at least \( j \) arguments. Therefore, \( \forall i, j.F(e_j^F, 0, f_i^F, k_i^F) \) requires that in any function definition to be monadified, any function to be monadified is applied to at least \( k \) arguments. \( F \) is defined in Figure 3.2. The parameter \( i \) is used to count the number of arguments that have already been provided by the context of the expression; it is needed to allow the checking of partial applications in the rule APP.

Note that requiring saturated function calls is not really a limitation of the algorithm because we can always supply additional arguments for undersaturated calls through \( \eta \)-expansion before applying the monadification.
3.4.1 Characterizations of Subexpressions

The definition of the monadification operator is steered by properties of expressions. First, we need a predicate that tells whether or not $e$ contains a call to $f$ with $k$ arguments. This property is captured in the definition of the predicate $\mathcal{R}(f, k, e)$, which is inductively defined in Figure 3.3.

In addition, we also need the information whether or not $e$ contains a recursive call to $f$ as a strict subexpression, that is, $e$ contains a recursive call to $f$, but $e$ itself is not a call to $f$. We write $\mathcal{S}(f, k, e)$ if $e$ has this property. For the definition of $\mathcal{S}$ we employ the notion of contexts. A context is essentially an expression with a hole, written as $\langle \cdot \rangle$. We can apply a context $C$ to an expression $e$, written as $C(e)$, which denotes the expression obtained by filling the hole of $C$ with $e$.

The syntax of contexts is given in Figure 3.4.
\[ \begin{align*}
\text{CALL} & \quad \mathcal{R}(f, k, f e_1 e_2 \ldots e_k) \\
\text{ABS} & \quad \frac{\mathcal{R}(f, k; d)}{\mathcal{R}(f, k; v \rightarrow d)} \\
\text{APP} & \quad \frac{\mathcal{R}(f, k; e_1)}{\mathcal{R}(f, k; e_1 e_2)} \\& \quad \frac{\mathcal{R}(f, k; e_2)}{\mathcal{R}(f, k; e_1 e_2)} \\
\text{LET} & \quad \frac{\mathcal{R}(f, k; e_1)}{\mathcal{R}(f, k; \text{let } x = e_1 \text{ in } e_2)} \\& \quad \frac{\mathcal{R}(f, k; e_2)}{\mathcal{R}(f, k; \text{let } x = e_1 \text{ in } e_2)} \\
\text{CASE} & \quad \frac{\mathcal{R}(f, k; e')}{\mathcal{R}(f, k; \text{case } e' \text{ of } \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\})} \\
& \quad \mathcal{R}(f, k; e_j) \text{ for some } j \in \{1, \ldots, n\} \quad f \notin \text{FV}(p_i) \\
& \quad \mathcal{R}(f, k; \text{case } e' \text{ of } \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\})
\end{align*} \]

Figure 3.3: Expressions Containing Recursive Calls

\[
C ::= \langle \cdot \rangle \mid \nu \rightarrow C \mid C e \mid e C \mid \text{let } \nu = C \text{ in } e \mid \text{let } \nu = e \text{ in } C \mid \\
\text{case } C \text{ of } \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\} \mid \\
\text{case } e \text{ of } \{p_1 \rightarrow e_1; \ldots; p_i \rightarrow C; \ldots; p_n \rightarrow e_n\}
\]

Figure 3.4: Syntax of Contexts

The definitions for free and bound variables extend in a natural way from expressions to contexts.

Now \( \mathcal{S} \) can be defined in terms of \( \mathcal{R} \) as follows.

\[
\mathcal{S}(f, k; e) = \exists e', C \neq \langle \cdot \rangle : e = C(e') \land \mathcal{S}(f, k; e')
\]

Another relationship between an expression \( e \) and its subexpressions \( e' \) is whether it is safe to lift \( e' \) to the outside of \( e \) and bind it to a variable. As we have elaborated in the examples, if \( e' \) contains a variable that is local to \( e \), say \( x \), we shall not lift \( e' \).
because otherwise $x$ would become unbound. Moreover, in the case that $e'$ resides in the body of a case expression, lifting $e'$ might change the termination behavior of the program, that is, lifting might make the program less lazy. To avoid this problem, we shall not lift such $e'$ either. If $e'$ can be safely lifted outside $e$, we say $e'$ is a liftable subexpression of $e$ and write $\mathcal{L}(e', e)$. For the definition of the liftability predicate $\mathcal{L}$ we again employ the notion of contexts.

$$\mathcal{L}(e', e) = (e = C(e') \land BV(C) \cap FV(e') = \emptyset \land C \neq C'(\text{case } e \text{ of } \{ \ldots ; p_i -> C^n ; \ldots \} ))$$

The last condition, which restricts the possible case contexts, guarantees that lifting does not increase the strictness of functions, at the expense of not being able to lift from RHSs of case rules. If increased strictness is not considered a problem, one could drop the last condition and thus extend the number of possible liftings, which will generally result in less complex monadified code.

3.4.2 Locating Return Expressions

We define a navigation operator $\mathcal{N}$ that moves monadification across lambda abstractions and into applications and eventually passes the found result expressions to the wrapping operator $\mathcal{W}$. $\mathcal{N}$ is also parameterized by a context $F$. More precisely, $\mathcal{N}_F(n, e)$ tries to “strip off” $n$ lambda abstractions from $e$ and then passes the result to $\mathcal{W}$. $\mathcal{N}$ can be defined inductively as follows.

For the base case, when $n = 0$, $e$ can be directly wrapped. Otherwise, the
syntactic structure of $e$ is scrutinized. When a lambda abstraction is lacking, $\eta$-expansion can be employed to generate the required abstraction. In the case of an application we can move $\mathcal{N}$ into the function. The definition of $\mathcal{N}$ is shown in Figure 3.5. In all expressions, $z$ has to be a fresh variable with respect to $e$, that is, $z$ has to be chosen such that $z \notin FV(e)$. Moreover, we assume $n > 0$. The variables $f_j$ and $k_j$ range over all $f$ and $k$ elements in the monadification context $F$. The metavariable $e$ in the last line works as a catch-all case and matches constants, variables, applications, and let expressions.

\[
\begin{align*}
\mathcal{N}_F(0, e) &= \mathcal{W}_F(e) \\
\mathcal{N}_F(n, \lambda v \to d) &= \lambda v \to \mathcal{N}_F(n-1, d) \\
\mathcal{N}_F(n, \text{case } e' \text{ of } \{p_i \to e_i\}) &= \\
&\begin{cases} \\
\mathcal{N}_F(n, \lambda z \to \text{case } e' \text{ of } \{p_i \to e_i \} z) & \text{if } R(f_j, k_j, e') \\
\text{case } e' \text{ of } \{p_i \to \mathcal{N}_F(n, e_i)\} & \text{otherwise} \\
\end{cases} \\
\mathcal{N}_F(n, e e') &= \mathcal{N}_F(n+1, e) e' \\
\mathcal{N}_F(n, e) &= \mathcal{N}_F(n, \lambda z \to e z)
\end{align*}
\]

Figure 3.5: Definition of the $\mathcal{N}$ Operator

It is worth mentioning that we use a slightly specialized version of $\eta$-expansion in the transformation of case expressions. Because of the following equality:

\[(\text{case } e' \text{ of } \{p_i \to e_i\}) \ e = \text{case } e' \text{ of } \{p_i \to e_i \ e\}\]

we can customize the $\eta$-expansion for case expressions as follows.

\[
\text{case } e' \text{ of } \{p_i \to e_i\} = \lambda z \to \text{case } e' \text{ of } \{p_i \to e_i \ z\}
\]
The reason for using this relationship instead of the general law is related to the
definition of the wrapping operator $\mathcal{W}$ that will be discussed below. Using the
simple form of $\eta$-expansion would force us to pass an application

$$(\text{case } e' \text{ of } \{ p_i \rightarrow e_i \}) \; z$$

to $\mathcal{W}$. If no topmost call to some $f_j$ in this expression is liftable, we have to resort
to applying $\mathcal{N}$ again, with the case expression as the parameter, which leads to
an infinite loop. By using the above transformation, we are able to avoid this
non-terminating situation.

### 3.4.3 Wrapping Return Expressions

Having exposed a return expression, we need to change its type from $t$ to $m \; t$. This is done by the operator $\mathcal{W}$, which takes a context $F$ and the expression to be
wrapped.

First, if there are no calls to any $f_i^F$ inside $e$, $e$ will be wrapped by a return
unless $e$ is a direct call to some $f_i^F$ because in this case its type is already monadic.
The condition is formally captured by the predicate $\forall i. \neg \mathcal{S}(f_i^F, k_i^F, e)$, and we get
in this case:

$$\mathcal{W}_F(e) = \begin{cases} 
\hat{f}_j \; e_1 \ldots \; e_{k_j} & \text{if } \exists j. e = f_j^F \; e_1 \ldots \; e_{k_j} \\
\text{return } e & \text{otherwise}
\end{cases}$$
Otherwise, that is, if a topmost call to any \( f_i^F \) inside \( e \) is liftable, the corresponding subexpression is lifted and bound to a fresh variable. The condition for this case is expressed formally by using contexts. The condition that \( e \) contains a liftable call to some \( f_i^F \) is expressed by the following formula.

\[
e = C(f_i^F \ e_1 \ e_2 \ \ldots \ \ e_{k_i^F}) \land \mathcal{L}(f_i^F \ e_1 \ e_2 \ \ldots \ \ e_{k_i^F}, e)
\]

To additionally ensure that \( C \) locates a non-nested call to some \( f_i^F \), that is, a call that is not nested inside a call to some other \( f_j^F \), we also require the following condition.

\[
\exists C' \neq \langle \cdot \rangle, j, e'_1 \ e'_2 \ \ldots \ \ e'_{k_j^F} : C'(z) = C'(f_j^F \ e'_1 \ e'_2 \ \ldots \ \ e'_{k_j^F})
\]

(for a fresh variable \( z \)). Note that for \( C' = \langle \cdot \rangle \), we allow the extraction of a nested call because this case covers the situation when \( e \) is a call to some \( f_j \) and contains a call to some \( f_i \) as a subexpression. In this case we have to lift and bind any such \( f_i \). The recursive application of \( W \) eventually replaces \( f_j \) with \( \hat{f}_j \). We combine these two conditions in the predicate \( \mathcal{C}_\mathcal{L} \), which is defined as follows.

\[
\mathcal{C}_\mathcal{L}(e, C, f_i^F \ e_1 \ \ldots \ \ e_{k_i^F}) =
\]

\[
(e = C(f_i^F \ e_1 \ \ldots \ \ e_{k_i^F}) \land \mathcal{L}(f_i^F \ e_1 \ \ldots \ \ e_{k_i^F}, e) \land
\exists C' \neq \langle \cdot \rangle, j, e'_1 \ \ldots \ \ e'_{k_j^F} : C'(z) = C'(f_j^F \ e'_1 \ \ldots \ \ e'_{k_j^F}))
\]
Now if $C_L(e, C, f_i^F e_1 \ldots e_k^F)$ holds, we obtain the following definition for $\mathcal{W}$.

$$\mathcal{W}_F(e) = \text{do} \{ z \leftarrow \mathcal{W}_F(f_i^F e_1 e_2 \ldots e_k^F); \mathcal{W}_F(C(z)) \}$$

where $z \notin \text{VARS}(e)$

Why do we require that $z \notin \text{VARS}(e)$? Because $z$ must not conflict with any variable in $e$, not only the free variables. This is because $z$ replaces a subexpression of $e$ and must not be captured by a binder in $e$.

Finally, if no topmost recursive call to any $f_i^F$ in $e$ is liftable, we have to scrutinize the syntactic structure of $e$ (since $e$ contains calls to some $f_i^F$, $e$ cannot be a variable or constant).

Case $e = \text{case } e' \text{ of } \{p_i \rightarrow e_i\}$. In this case, we have to wrap and bind $e'$ and move the operation down to the bodies $e_i$:

$$\mathcal{W}_F(e) = \text{do} \{ z \leftarrow \mathcal{W}_F(e'); \text{ case } z \text{ of } \{p_i \rightarrow \mathcal{W}_F(e_i)\}\}$$

Case $e = \lambda v \rightarrow e'$. This is a case for which monadification fails as we have discussed in Section 3.3. A possible remedy is discussed in Section 3.6.

Case $e = e_0 e_1 \ldots e_m$ where $e_0$ is not an application, that is, $e_0 \neq e' e''$ for any $e', e''$. A general solution to this case is to apply $\mathcal{W}$ to $e_1 \ldots e_m$, which makes their types monadic, bind them to fresh variables with respect to $e$, say $z_1, \ldots, z_m$, and also apply $\mathcal{N}$ to $e_0$. This requires the recursive application $\mathcal{N}_F(m, e_0)$ because $e_0$ is regarded as a function of $m$ parameters and has to change its return type to a monadic type. Only when $e_0$ is a recursive let expression, that is, if $e_0 =$
let \( v = C(v) \) in \( e' \), and contains a non-liftable call to an \( f^F_i \), \( \mathcal{N}_F(m, e_0) \) will apply \( \eta \)-expansions to \( e_0 \) and eventually pass it, applied to \( m \) arguments, down to \( \mathcal{W} \) again, which would cause an infinite loop. So monadification stops with an error in this case. We capture this latter condition in the predicate \( \mathcal{C}_\mathcal{L} \), which is defined as follows.

\[
\mathcal{C}_\mathcal{L}(e_0, C, f^F_i \ e_1 \ldots \ e_{k_F}) = (e_0 = C(f^F_i \ e_1 \ldots \ e_{k_F}) \land \neg \mathcal{L}(f^F_i \ e_1 \ldots \ e_{k_F}, e))
\]

Hence, if \( e_0 = \text{let } v = C(v) \text{ in } e' \implies \neg \mathcal{C}_\mathcal{L}(e_0, C, f^F_i \ e_1 \ldots \ e_{k_F}) \), we get the following definition for \( \mathcal{W} \).

\[
\mathcal{W}_F(e) = \text{do } \{ z_1 \leftarrow \mathcal{W}_F(e_1); \ldots; z_m \leftarrow \mathcal{W}_F(e_m); \mathcal{N}_F(m, e_0) \ z_1 \ldots \ z_m \}
\]

This solution might introduce unnecessary bindings in some of the \( e_i \) (1 \( \leq i \) \( \leq m \)) (but not all). We can eliminate these by optimizing the resulting expression through the left unit monad law [6].

Case \( e = \text{let } x = e_1 \text{ in } e_2 \). If \( \forall i. \neg \mathcal{R}(f^F_i, k, e_1) \) holds, which means there are no calls to any \( f^F_i \) in \( e_1 \), we can wrap \( e \) by only wrapping \( e_2 \):

\[
\mathcal{W}_F(e) = \text{let } x = e_1 \text{ in } \mathcal{W}_F(e_2)
\]

In case \( x \) is not recursively defined, that is, \( x \notin \text{FV}(e_1) \), \( e \) is treated like a \( \beta \)-redex.

\[
\mathcal{W}_F(e) = \text{do } \{ x \leftarrow \mathcal{W}_F(e_1); \mathcal{W}_F(e_2) \}
\]
But if not only $x$ is recursively defined, but also its definition contains non-liftable calls to some $f_i^F$, we are unable to apply $W$ to it. This is basically the same situation as for lambda abstraction shown above.

Finally, we have to define the monadification operator $M$, which can be given directly in terms of $N$:

$$M_F(e_i) = N_F(k_i, e_i)$$

The definition of $W$ is summarized in Figure 3.6.

1. If $\forall i \neg S(f_i^F, k_i^F, e)$:
   
   $$W_F(e) = \begin{cases} 
   \hat{f}_j e_1 \ldots e_{k_j} & \text{if } \exists j, e = f_j^F e_1 \ldots e_{k_j} \\
   \text{return } e & \text{otherwise}
   \end{cases}$$

2. Otherwise, if $C_L(e, C, f_i^F e_1 \ldots e_{k_i^F})$:
   
   $$W_F(e) = \text{do } \{ z \leftarrow W_F(f_j^F e_1 \ldots e_{k_j}^F); \ W_F(C(z)) \}$$
   where $z \notin \text{VARS}(e)$

3. Otherwise, if

   (a) $e = \text{case } e' \text{ of } \{ p_i \rightarrow e_i \}$:
   
   $$W_F(e) = \text{do } \{ z \leftarrow W_F(e'); \ \text{case } z \text{ of } \{ p_i \rightarrow W_F(e_i) \} \}$$

   (b) $e = e_0 e_1 \ldots e_m$ where $e_0 \neq e', e''$ for any $e', e''$ and $e_0 = \text{let } v = C(v) \text{ in } e' \implies \neg C_L(e_0, C, f_i^F e_1 \ldots e_{k_i^F})$:
   
   $$W_F(e) = \text{do } \{ z_1 \leftarrow W_F(e_1); \ldots; z_m \leftarrow W_F(e_m); \ N_F(m, e_0) z_1 \ldots z_m \}$$

   (c) $e = \text{let } x = e_1 \text{ in } e_2$:
   
   $$W_F(e) = \begin{cases} 
   \text{let } x = e_1 \text{ in } W_F(e_2) & \text{if } \forall i \neg R(f, k, e_1) \\
   \text{do } \{ x \leftarrow W_F(e_1); \ W_F(e_2) \} & \text{if } x \notin \text{FV}(e_1)
   \end{cases}$$

4. Otherwise, $W$ fails.

Figure 3.6: Definition of the $W$ Operator
3.5 Correctness of Monadification

In Section 3.3 we have identified two correctness criteria for the monadification of functions. Before we evaluate the monadification algorithm \( \mathcal{M} \) presented in Section 3.4 according to these criteria, we recall the restrictions of the algorithm. There are two cases that \( \mathcal{M} \) cannot deal with:

- A result expression is a lambda expression, which contains a recursive call involving local variables. (In practice, this should not be a common case because the return type is higher order.)

- A result expression is an application whose first part is a recursive \texttt{let} expression that contains a non-liftable recursive call.

We have already discussed the problem that binding recursive calls to variables might change the termination behavior of the transformed function. For \texttt{case} expressions we were able to avoid this problem by classifying the calls as not liftable (see the definition of \( \mathcal{L} \) in Section 3.4.1) and eventually moving the monadification down into \texttt{case} expressions. However, the problem is generally always present in situations in which expressions are lifted from non-strict functions because in a lazy evaluation setting, these recursive calls might not be evaluated in the original function, but they might be evaluated after having been lifted. This generally causes an “increased strictness” of monadified functions. In these cases, monadification is not complete. The following functions illustrate this case:

\[
\begin{align*}
g &= \_ \to 0 \\
f &= x \to g(f x)
\end{align*}
\]
Since $g$ does not rely on the input to produce a return value, $f$ always terminates and returns 0. But if we apply our monadification algorithm to $f$, we get:

$$f^M = \text{do } \{ y \leftarrow f^M x; \text{return } (g \ y) \}$$

The actual evaluation depends on the strictness of the implementation of the $>>=$ operation for the monad that is being used. For example, the implementation of $>>=$ for the `Maybe` monad inspects the pattern of the argument and therefore forces the evaluation of the expression that is to be bound (which is probably the case for most monads). Therefore, the argument $f^M$ is evaluated in this example, and inevitably $f^M$ would not terminate. To avoid this problem, we could avoid the binding of the result of the function application and substitute $y$ in the return expression by $f^M \ x$.

$$f^{M'} \ x = \text{do } \text{return } (g \ (f^{M'} \ x))$$

This solution works well when $g$ has a polymorphic type. However, if $g$’s type is constrained by a type signature to, say `Int -> Int`, the shown transformation will cause a type error because $g$ is applied to an argument of type $m \ \text{Int}$.

Now we can give the main result about the correctness of our monadification algorithm. The first result is that monadification is sound.

**Theorem 2** Given $f = e$, $\hat{f} = M_F(e)$ is a sound monadification of $f$.

To prove the soundness, we can consider two cases. If $f$ is not recursively defined, the soundness can be shown by a structural induction on the function definition. Otherwise, we can perform an induction on the recursive evaluation of $f$. The base
case is when the recursion of \( f \) ends, that is, when no recursive evaluation takes place. In the case of recursion, the inductive hypothesis and the left unit monad law can be applied to conclude soundness.

Although monadification is not complete, due to increased strictness, monadification is a complete transformation under eager evaluation.

**Theorem 3** Given \( f = e \), \( \hat{f} = \mathcal{M}_F(e) \) is a complete monadification of \( f \) under eager evaluation.

The completeness follows from soundness whenever \( \hat{f} \) is not less defined than \( f \). Under eager evaluation this condition is satisfied.

Moreover, we can show that the proposed algorithm terminates on all inputs. \( \mathcal{M} \) is defined in terms of \( \mathcal{N} \), which eventually passes expressions to \( \mathcal{W} \). Whenever a recursive definition occurs, \( \mathcal{W} \) is applied recursively to smaller subexpression so that the termination follows by a structural induction on the expression. There is one exception, namely when \( \mathcal{W} \) is applied to an application \( e_0 \ldots e_m \) and \( e_0 \) is a recursive \texttt{let} expression containing a recursive call that is not liftable. In that case, \( e_0 \) is passed to \( \mathcal{N} \) that might expand it and pass it back to \( \mathcal{W} \). Since our algorithm identifies this case, \( \mathcal{W} \) is guaranteed to terminate.

Finally, our monadification algorithm preserves the well typing of the monadified functions. Of course, external calls (from definitions in \( P' \)) to monadified functions will no longer be type correct. However, if the set of functions to be monadified is closed with respect to mutual calls, the well typing of the whole program is ensured. Note that monadification is \textit{not} type preserving since the
types of some functions are changed to monadic type. However, the programs that are produced by monadification are type correct if the input programs are type correct. This property follows from the fact that our algorithm will eventually change all the uses of a symbol \( f \) whose definition changes from a type \( t \) to \( m \). In particular, the value of \( f \) will be bound (in the sense of the monadic bind operation) to a variable, which has type \( t \). Since this variable is substituted for the used occurrence of \( f \), the well typing of the context is re-established. In those cases when a symbol cannot be changed, our algorithm stops with an error message and does not produce a possibly ill-typed program.

3.6Runnable Monads

In Section 3.4, we have seen a situation where \( \mathcal{W} \) cannot be applied to a lambda abstraction. This was due to the need to lift a subexpression out of a context that would also lift variables out of their scope.

Another problem in applying monadification to a function in a module containing other functions is that monadification is only locally type correct, that is, although it guarantees the type correctness of the monadified function, it does not guarantee that callers of the monadified function deal with the new monadic type correctly. Global type correctness can be recovered by a static analysis that identifies all calls of a monadified function and monadifies the calling functions accordingly. However, this might lead to a proliferation of monadic types all over the program. We call this the problem of *monad infestation.*
The simple concept of runnable monads provides a (partial) solution to both of these problems. A runnable monad is a monad that provides a run operation, which can be considered the dual operation to return and which has been already defined for some monads in Haskell. We can define runnable monads as a subclass of Monad as follows.

```haskell
class Monad m => MonadRun m where
  run :: m a -> a
```

Basically, the purpose of run is to extract the value from a monad.

To give an example, we show how to define the Maybe type constructor as an instance of MonadRun.

```haskell
instance MonadRun Maybe where
  run (Just x) = x
```

As another example we define a MonadRun instance for state transformers. The idea of getting the value out of the monad is to apply the state transformer to an initial state and extract the value from a value/state pair. To do this, we need to know what the initial state is. To this end, we can define a type class Initializable of initializable values. Any data type that is intended to be used as a state for a state transformer can be made an instance of Initializable by providing an initial value. For instance, an initial value for integers could be 0.

```haskell
class Initializable s where
  initValue :: s
  initValue = undefined

instance Initializable Int where
  initValue = 0
```
Now we can capture the requirement on the state of a state transformer monad to have an initial value by a corresponding class constraint in the instance definition for `MonadRun`.

\[
\text{instance \ Initializeable \ s \Rightarrow \ MonadRun \ (ST \ s) \ where}\\
\text{run \ (\text{Trans} \ f) = let \ (x,_) = f \ \text{initValue} \ \text{in} \ x}
\]

The above definition assumes the following definition for the state transformer data type.

\[
\text{data \ ST \ s \ a = Trans \ (s \to \ (a,s))}
\]

In addition to the three basic monad laws [6], a runnable monad should also satisfy the following inversion law [35].

\[
\text{run} \ (\text{return} \ x) = x \quad \quad \quad \quad \textbf{Monad Inversion}
\]

This law ensures that values injected into a monad can be recovered by `run`. It is easy to check that this law holds for the `Maybe` and the `ST` monads.

The value of the `run` operation lies in the fact that we can use it to “unwrap” a monadic expression at any place, meaning that we can extract the value from a monad in-place without lifting and binding. Therefore, in the process of wrapping, whenever a recursive call to `f` is encountered that cannot be lifted, we can apply `run` to get a proper non-monadic value instead. With this approach, the function `f` from Section 3.4.3 can be monadified as follows.

\[
f^M = \lambda x \to \text{return} \ (\lambda y \to \text{run} \ (f^M \ y) \ x)
\]

This method is sound and complete (at least for monads that satisfy the monad inversion law). However, by escaping monads, the essence of monads can be lost
to some degree at those places where \texttt{run} is used. Moreover, monadic actions cannot be inserted at these points. On the other hand, \texttt{run} provides a way to make monadification work in some cases.

Another use for \texttt{run} is to limit the effect that the monadification of a function has on the rest of a module. By wrapping \texttt{run} around some or all calls to the monadified function, we have precise control over what other functions have to be monadified. In this way, we can effectively bound monad infestation.

3.7 Adding Actions to Monads

So far, we have developed an algorithm for converting a group of functions into monadic form. In most cases the goal of this transformation is to add further code to these functions. This code is sometimes also referred to as \textit{monadic actions}. In the introductory example from Section 3.1, the conditional expression for handling divide-by-zero exceptions is such a monadic action. Other examples are the calls to \texttt{tick} and \texttt{out} in the monads presented in Section 3.2.

After discussing several approaches to integrating actions into monads in Section 3.7.1, we define the syntax and semantics of a language that can be used to express such updates in Section 3.7.2. In Section 3.7.3, we discuss the type safety of the approach.
3.7.1 Three Options for Adding Actions

A simple, but inflexible, approach is to (always) insert one particular action before return. Such an action can be passed down to \( \mathcal{W} \) from \( \mathcal{M} \) and \( \mathcal{N} \) as a parameter. We could define a function \( \mathcal{W}_a \) in almost the same way as \( \mathcal{W} \) except changing return \( e \) everywhere to \( a \Rightarrow \text{return} \ e \). However, this solution is rather limited because the context of the return expression is totally ignored, which means that the same action is inserted before every return. No useful adaptation can be achieved in this way.

A more general approach can be obtained by using some form of symbolic rewriting to describe the insertion of actions. As explained in Section 2.1, the idea of symbolic rewriting is to match a pattern against an expression to obtain a variable binding, then substitute the expression with another pattern, with the variables substituted according to the variable binding. The following rewrite rule describes the adaptation that is needed for the divide-by-zero exception handling in the example from Section 3.1.

\[
\text{return } (x \ '\text{div}' \ y) \Rightarrow \text{if } y == 0 \text{ then Nothing else return } (x \ '\text{div}' \ y)
\]

The pattern return \( (x \ '\text{div}' \ y) \) is matched against the expression return \( (i \ '\text{div}' \ j) \), which causes \( x \) to be bound to \( i \) and \( y \) to be bound to \( j \). The expression is then substituted by the conditional expression on the right-hand side, with \( x \) and \( y \) substituted by \( i \) and \( j \), respectively.

The purely syntactic rewriting approach is limited by the fact that only literal
occurrences of rewrite patterns can be identified. For example, if the program contains the expression `div i j`, the above rule would not match. We can imagine using rewriting modulo an equational theory to extend the applicability of matching, but the general problem remains.

Even though rewrite rules can incorporate context information through the use of metavariables in patterns, they cannot refer to parts of the context that is not being rewritten. However, this feature is sometimes very useful.

An example to illustrate this idea is the extension of the `eval` function by an output trace as described in Section 3.2.3.

The challenge for rewriting arises from the fact that the inserted expressions need to refer to the parameters of the function. To describe such a transformation we need a rewriting system that is capable of expressing rewrite rules with variable context dependencies. We can imagine the following rewrite rule for the task.

```haskell
case e' of p -> (return e) => out (show p ++ "=" ++ show e); return e)
```

The intended meaning is to match an enclosing context of a `case` expression, bind the metavariable `p`, and then apply the shown rewrite rule.

### 3.7.2 A Context Update Language

The last approach leads to an update language that can express context-dependent rewrites. A context update is a rewrite rule that has been placed into a context. Contexts were defined in Figure 3.4. We can apply a context to a rewrite rule
to obtain a context update. A rewrite rule is either a simple rule $p_1 \Rightarrow p_2$ or the composition of two rules $r_1; r_2$. Rewrite rules can contain metavariables that match any expression in the object program. Therefore, we extend the syntax of the object language defined in Figure 3.1 to include metavariables for patterns ($p$) and expressions ($e$) that are used to describe contexts. The definition of contexts from Figure 3.4 now refers to this extended form of expressions, which causes contexts to possibly contain metavariables.

\[
\begin{align*}
  u &::= C\langle r \rangle \\
  r &::= p \Rightarrow p \mid r; r \\
  p &::= c \mid v \mid m \mid p \mid p \mid \text{case } e \text{ of } \{ q_1 \rightarrow e_1; \ldots ; q_n \rightarrow e_n \} \\
  q &::= c \mid q \mid q \\
  e &::= c \mid v \mid m \mid e \mid e \mid \backslash v \rightarrow e \mid \text{let } v=e \text{ in } e \\
  \text{case } e \text{ of } \{ p_1 \rightarrow e_1; \ldots ; p_n \rightarrow e_n \}
\end{align*}
\]

Figure 3.7: Syntax of the Context Update Language

We require that contexts and patterns are linear in the sense that a metavariable does not appear more than once. However, within a context rewrite rule, a metavariable that occurs in the context might be reused in the patterns of rewrite rules where it is just substituted by the binding obtained from matching the context. We restrict patterns of rewrite rules to not include expressions that create bindings to prevent the illegal creation of unbound variables. The restricted form of case expressions

\[
\text{case } e \text{ of } \{ q_1 \rightarrow e_1; \ldots ; q_n \rightarrow e_n \}
\]

disallows the use of variables in the left-hand sides of case rules ($q$) for the same reason. On the other hand, having case expressions allows us to express updates
dealing with conditionals, such as the one for handling divide-by-zero errors.

When an update is applied to an expression, its context is matched against the expression and metavariables that occur in the context are bound to parts of the expression. After the context is matched, the rewrite rule in the hole is applied to the expression that matches the hole of the context. It will be applied recursively to all the subexpressions. All simple rules in a composition of rewrite rules are tried on the expression until one of them applies. A simple rule is applied as follows. The rule is instantiated by substituting those metavariables for which a binding is already provided by the matched context. Then the left-hand side of the rule is matched against the expression to obtain bindings for the remaining uninstantiated metavariables in the rule. These bindings are then used to replace the metavariables in the right-hand side to build the resulting expression. The semantics of the update language is defined in Figure 3.8 by rules that define the judgment $[[u]]_{\sigma} e = e'$ to express the fact that the update $u$ changes the expression $e$ to $e'$ under that mapping $\sigma$ from metavariables to expressions. This context update language is general enough to express all the approaches discussed in Section 3.7.1.

The first three inference rules in the first line define the core rewriting semantics. The first rule in the second line defines how a composition of rewrite rules in a context is applied recursively. The next group of rules from line two to four define the recursive application of rules in the context. We write $\mu r$ for a rule $r$ that has to be applied recursively. The rules implement a “stop-top-down” strategy (see, for example, [47] and Chapter 5) that does apply the rewrite rule recursively at possibly different places in the expression, but does not move recur-
Figure 3.8: Semantics of Context Updates
sively into expressions that have been rewritten. This is exactly the behavior we need for adding monadic actions because we generally have to replace more than one return expression that occurs, for example, in different branches of a case expression, but we do not want to replace recursive occurrences of return within other return expressions that are rewritten. The remaining rules starting in line five are congruence rules that define the matching of the context. In the definition of the semantics we need a judgment of the form $r \mathrel{\bowtie} e$ that tells whether or not the update $r$ applies to $e$. A rule $r$ is applicable to $e$ if we can apply the first inference rule in Figure 3.8 to perform the update. We also use a judgment $p \mathrel{\sigma} e$ that infers under which binding of metavariables to expressions the pattern $p$ matches the expression $e$, that is, $\sigma(p) = e$.

Note that the nonterminals $e$ from Figure 3.7 that are used in contexts might contain metavariables that are bound during the process of context matching described by the congruence rules. In contrast, the nonterminals $e$ from Figure 3.1 just represent expressions to which context rewrite rules are applied. These two different versions of nonterminals can be easily distinguished syntactically in the inference rules as follows. The nonterminals $e$ used for representing contexts always occur (in judgments) inside of the semantics brackets $[ ]$, whereas the “ordinary” expression nonterminals $e$ always occur outside of these brackets. This distinction has to be kept in mind when reading conditions, such as $e_0 \mathrel{\sigma} e_1$. 
3.7.3 Type Preservation

An important property of updates is type preservation: when updates are applied to well-typed expressions, they should ensure that any resulting expressions is of the same type as the original expression. A context update is type preserving if it satisfies the following two conditions.

1. For any update $C(e_1 \Rightarrow e'_1; \ldots; e_n \Rightarrow e'_n)$:

$$\bigcup_{1 \leq i \leq n} FV(e_i) \subseteq FV(C)$$

This condition ensures that no unbound variables are introduced by the update.

2. For any rule $p_1 \Rightarrow p_2$, the type of $p_2$ is the same as $p_1$. This condition ensures that we replace expressions with expressions of the same type. From the substitution lemma [95] it follows that the result of applying the update to the whole expression is type correct and has the same type.

3.8 Alternative Approaches

Other approaches to monadification also exist. In particular, we study two typical ones in this dissertation, namely, Lämmel’s type-directed approach and the CPS approach.
3.8.1 Type-Directed Approach

Lämmel has employed program-transformation techniques to reduce the need for anticipation in developing reusable software [39]. One example he considered is the transformation of programs into monadic form. In his approach he has employed a program-transformation technique called *sequencing* [24] to flatten an expression into `let` expressions. The applications are flattened by means of `let` expressions. The notation `let x : τ = t₁ in t₂` is represented as the term `(λx : τ. t₂) t₁`. The `λ` has the same semantics as `λ`. The difference in notation is to distinguish it as an administrative `λ`. When the result needs to be simplified and converted, they can be used to unflatten the expression. Take our evaluator example. Consider the following equation of the `eval` function.

```
    eval (Plus x y) = eval x + eval y
```

The result `eval x + eval y` can be sequenced to a flattened form as follows.

```
(λx₁ : Int. (λx₂ : Int. x₁ + x₂)) (eval x y)
```

This intermediate result is then transformed into a monadic computation by two operators defined in Figure 3.9.

The main operator `⇒_{ms}^{Γ' : τ; Γ}` is directed by the expected type `τ` of the resulting term. It is also driven by two type assignments `Γ` and `Γ'`. `Γ` contains type assignments regarding the input term, and `Γ'` is for the output term. The idea is that `Γ'` contains the same type assignments as `Γ` except that some types are inserted with
applications of monads $m$. Therefore, it is determined by the expected type of functions being monadified. The KEEP and LIFT rules convert a normal value into a monadic value (Lämmel refers to it as a computation) or keep it depending on the type. Rules VAR, APP, LAM, and LAM' deal with different language constructs and traverse into the terms. All the hard work is done in BIND that converts a let binding into a monadic binding.

To illustrate the approach, let us consider the above term

$$\lambda x_1 : \text{Int}. (\lambda x_2 : \text{Int}. x_1 + x_2) \quad (\text{eval} \ x \ y)$$

which is converted into

```do {x1 <- eval x; x2 <- eval y; return (x1+x2)}```

Since we try to monadify the `eval` function whose original type is `Expr -> Int`, the control parameter $\tau$ is $\text{Expr} \rightarrow m \text{Int}$. The input and output type assignments
are:
\[
\Gamma = \{ \text{eval : Expr} \rightarrow \text{Int} \}
\]
\[
\Gamma' = \{ \text{eval : Expr} \rightarrow m \text{ Int} \}
\]

To apply the BIND rule, we will need to satisfy the following premises:

(1) eval \(x\) \(\Rightarrow_{ms}^{m} \text{Int} : \Gamma : \Gamma'\) eval \(x\)

(2) \((\lambda x_2 : \text{Int}. x_1 + x_2)\) (eval \(x\)) \(\Rightarrow_{ms}^{m} \text{Int} : \Gamma_1 : \Gamma'_1\) do \(\{ x_2 \leftarrow \text{eval } y; \text{return } (x_1 + x_2) \}\)

where \(\Gamma_1\) and \(\Gamma'_1\) are \(\Gamma, x_1 : \text{Int}\) and \(\Gamma', x_1 : \text{Int}\), respectively. (1) can be obtained by applying APP rule. (2) can be recursively obtained by applying the BIND rule with the following premises.

(3) eval \(y\) \(\Rightarrow_{ms}^{m} \text{Int} : \Gamma : \Gamma'\) eval \(y\)

(4) \(x_1 + x_2 \Rightarrow_{ms}^{m} \text{Int} : \Gamma_2 : \Gamma'_2\) return \((x_1 + x_2)\)

where \(\Gamma_2\) and \(\Gamma'_2\) are \(\Gamma_1, x_2 : \text{Int}\) and \(\Gamma'_1, x_2 : \text{Int}\), respectively. (3) can be obtained very similarly to (1). (4) is lifted by the rule LIFT.

Lämmel’s approach is type-directed in contrast to our syntax-directed transformation method, which does not depend on the presence of typing information. Moreover, his transformation is given by inference rules in natural semantics style, whereas we describe a transformation algorithm.
3.8.2 The CPS Approach

Another approach to monadification has been known for a long time in the domain of compiler transformations [24, 28]. The idea of these algorithms is to transform all functions and intermediate results in a program into monadic form. Hatcliff and Danvy give two transformations, a call-by-value and a call-by-name version. Below we present their call-by-value transformation\(^3\), adjusted to our abstract syntax from Figure 3.1.

\[
\begin{align*}
M(c) & = \text{return } c \\
M(v) & = \text{return } v \\
M(\lambda v \to e) & = \text{return } (\lambda v \to M(e)) \\
M(e \ e') & = \text{do } \{ f \leftarrow M(e); \ x \leftarrow M(e'); \ f \ x \} \\
M(\text{let } v=e \text{ in } e') & = \text{let } v=M(e) \text{ in } M(e')
\end{align*}
\]

The call-by-name transformation is very similar. The only two differences are: (i) only recursively defined variables (by let) are wrapped by return, in contrast to all variables in the call-by-value transformation, and (ii) the monadification of a function argument is not bound to a variable, but directly monadified in place:

\[
\begin{align*}
M(e \ e') & = \text{do } \{ f \leftarrow M(e); \ f \ M(e') \}
\end{align*}
\]

While these algorithms are straightforward and simple, the translation of only a subset of functions into monadic form is complicated by the need to delimit

---

\(^3\)The algorithms given in [28] do not deal with data types or case expressions
the effect of monadic code to only the required places. One could imagine using
one of these simple monadification algorithms to monadify all function calls in a
program and then trying to get rid of the unwanted monadifications by selecting
the Id monad for all but the required places and simplifying the resulting code by
applications of the monad laws. This approach is possible since the monadification
introduces monad expressions that are principally independent of one another and
can therefore be of different monad types. However, this approach does not work
in general. We illustrate the limitations of the simple algorithms by two examples.
First, consider the following expression of type (a -> b) -> a -> b.

\f -> \x -> f x

The call-by-value translation yields the following monadified form.\footnote{The expression can be simplified to return (\f->return (\x->do {f x})).}

\nreturn (\f -> return (\x -> do g <- return f
    y <- return x
    g y))

This expression has the following type.

\n(Monad m1, Monad m2, Monad m3) =>
m1 ((a -> m2 b) -> m3 (a -> m2 b))

If the goal is to monadify the original expression for \(k = 2\) parameters, the resulting
expression must have the following type.

\n(a -> b) -> a -> m b

However, there is no choice for \(m1\), \(m2\), and \(m3\) to instantiate the above monadic
type to this type. The call-by-name translation produces the following result.
return (\f->return (\x->do {g <- f; g x}))

The type of this expression shown below suffers from the same limitations as the type of the call-by-value translation.

(Monad m1, Monad m2, Monad m3) =>
  m1 (m2 (a -> m2 b) -> m3 (a -> m2 b))

On the other hand, there is a simple monadification of the original expression that has the required type, namely the following expression.

\f -> \x -> return (f x)

This expression is exactly what our algorithm produces. The approach of these simple algorithms – to wrap all variables and constants into a monad – requires the ability to choose monads independently of one another (Id vs. m) to be able to selectively monadify a function at a particular result type. However, in cases when the result type of a higher-order argument (such as f in the example) contributes to the overall result type of the function to be monadified, the monadification of that argument forces its result type to be wrapped by the same monad as the overall result type (here m2). This fact limits the applicability of these algorithms with respect to selective monadification essentially to first-order functions. Another limitation is revealed by recursive definitions, which will be illustrated next.

As we have already seen from the call-by-value and call-by-name translations, the main source of variation in the simple algorithm is the translation for application. One might think of still another version (not described in [28]) that can
be obtained from the call-by-value transformation by wrapping the result of the application in an additional return.

$$\mathcal{M}(e e') = \text{do } \{f \leftarrow \mathcal{M}(e); \ x \leftarrow \mathcal{M}(e'); \ \text{return } (f \ x)\}$$

This variation ensures the independence of result types that was a problem with the other two translations. The application of this algorithm yields the following result that differs from the call-by-value result only by the additional return around g y.

$$\text{return } (\lambda x \to \text{return } (\lambda y \to \text{do } \{g \leftarrow \text{return } f; \ y \leftarrow \text{return } x; \ \text{return } (g \ y)\}))$$

The type of this expression is general enough to allow the instantiation to the desired result type.

$$(\text{Monad } m_1, \text{Monad } m_2, \text{Monad } m_3) \Rightarrow m_1 ((a -> b) -> m_2 (b -> m_3 b))$$

Alas, the transformation is unsound in the sense that it can produce ill-typed expressions. Consider again the example from Section 3.3, which can be monadified on its result type by our algorithm as follows.

\[
f :: \text{Monad } m \Rightarrow \text{Int} -> \text{Int} -> m \text{ Int} \\
f \ x \ y = \text{if } x==0 \text{ then return } y \text{ else } f \ y \ (x-1)
\]

The third algorithm produces the following definition (we have simplified the result according to the monad laws).

$$f \ x = \text{return } (\lambda y -> \text{if } x==0 \text{ then return } y \text{ else return } (f \ y \ (x-1)))$$
However, this definition contains a type error. The call-by-value and call-by-name transformations given by Hatcliff and Danvy work well for this example and both yield the following translation.

\[
f \ x = \text{return} \ (\ \lambda y. \text{if } x == 0 \ \text{then } y \ \text{else } \{ g \leftarrow f \ y ; \ g \ (x-1) \})
\]

The definition is type correct, but again due to the too many involved monads the type is too constrained to be instantiated as required.

\[
\text{Monad } m \Rightarrow \text{Int} \rightarrow m \ (\text{Int} \rightarrow m \ \text{Int})
\]

The shown examples demonstrate the limitations of the simple monadification algorithms with regard to the monadification of functions on selected parameters and that a tailor-made algorithm is required instead.

Moreover, since this approach monadifies \textit{all} the values, including those defined in other modules, the algorithm might fail for built-in functions or other functions whose source code is not available. Consider our primary example \texttt{eval} function. Since this algorithm does not handle case expressions (pattern matching is essentially case expressions), we consider only one equation.

\[
\text{eval} \ (\text{Plus} \ x \ y) = \text{eval} \ x + \text{eval} \ y
\]

This is equivalent to the following.

\[
\text{eval} = \ (\text{Plus} \ x \ y) \rightarrow (\ (+) \ (\text{eval} \ x)) \ (\text{eval} \ y)
\]

We can apply the call-by-value transformation to the definition as follows.
\[
\text{eval} = \mathcal{M}(\text{\textbackslash (Plus x y)} \rightarrow ((+) (\text{eval x})) \text{ (eval y)}) \\
= \text{do } f \leftarrow \mathcal{M}(\text{eval ((+) x)}) \\
\text{ z } \leftarrow \mathcal{M}(\text{eval y}) \\
\text{ f z} \\
= \text{do } f \leftarrow \text{do } f1 \leftarrow \mathcal{M}(+) \\
\text{ z1 } \leftarrow \mathcal{M}(\text{eval x}) \\
\text{ f1 z1} \\
\text{ f z} \\
= \text{do } f \leftarrow \text{do } f1 \leftarrow \text{return (+)} \\
\text{ z1 } \leftarrow \text{do } f2 \leftarrow \text{return eval} \\
\text{ z2 } \leftarrow \text{return x} \\
\text{ f2 z2} \\
\text{ f1 z1} \\
\text{ f z} \\
\text{ z } \leftarrow \text{do } f3 \leftarrow \text{return eval} \\
\text{ z3 } \leftarrow \text{return y} \\
\text{ f3 z3} \\
\text{ f z} 
\]

The resulting definition reveals the problem: the operator (+) needs to be monadified to type \text{Int} \rightarrow \text{m (Int -> m Int)}. And this cannot be done for built-in functions, for which the source code is unavailable.

Other work on local monad transformations can be found in the context of CPS transformations. For example, the approach taken in [73] translates only those functions that have the same continuation. This and all the other work on CPS transformations and monadic normal forms is targeted at compiling functional languages. In contrast, our work is concerned with source-code transformations of programs that are still to be used by programmers.
3.9 Summary and Future Work

We have shown how function definitions can be automatically converted into a monadic form by a process called monadification. The developed transformation is safe since it preserves syntax and type correctness of the transformed program. Moreover, automatic monadification can preserve the semantics of the original program to a large degree. One could even argue that automatic monadification preserves the semantics as much as possible.

Monadification is an example of a generic program transformation that can be effectively used as a very general functional refactoring [25]. In many cases, such refactorings are only preparatory steps toward adding further functionality to programs. In the monadic setting this means to add monadic actions. We have addressed this task by employing a simple but effective context-dependent rewriting system, which preserves the types of the monadified program.

We have an initial prototype implementation of a monadification tool written in Haskell, which can be run as a stand-alone tool. Although quite a few desirable features are still missing, the current prototype can reproduce all the examples discussed in this chapter. Future versions might contain an analysis phase that can detect closeness of a set of function with respect to calls of monadified functions, the possibility to specify the addition of actions, and an intelligent “pretty-reprinting”, that is, a pretty-printing of changed code parts that retains as much of the original program layout as possible. Ultimately, we plan to integrate our monadification tool into the Haskell refactoring tool “HaRe” [50]. Monadification could then be a refactoring that is offered through the menus of ordinary text editors.
Chapter 4 – Update Language\textsuperscript{1}

In Chapter 3, we have investigated a specific program transformation problem, monadification. In this chapter, we study the problem of program updates on a more general level. We describe the design and formalism of a rule-based update language for Haskell, called \textit{HULA}. This language offers a mechanism of expressing updates to Haskell programs in a systematic and reliable way. HULA provides a means of updating every construct of the object language, a subset of Haskell. Since a Haskell program can be translated into a lambda calculus expression, we can similarly focus on the updates on the essential constructs of the object language. As a result, HULA can be consolidated to a core calculus. In this chapter, we will first introduce and define the core update calculus in Section 4.2, as well as its semantics and a type system. We will also define type and structure requirements to ensure the safety of the updates, which means that they preserve the type-correctness of the transformed programs. Then in Section 4.3, we will define the syntax of HULA and a set of rules for translating it into core calculus.

\textsuperscript{1}Contents of this chapter are largely based on [22, 21].
4.1 Introduction

As we put in Chapter 1, updating programs through update programs on abstract syntax trees has the advantage that the resulting programs are guaranteed syntax correctness. This feature is given for free by the higher level view of programs. However, the type correctness is not. To guarantee the type correctness of the resulting programs to some extent, we need to carefully design the update languages and define $\mu$ (see Section 1.1) with $\pi =$ type correctness. To program updates on Haskell, a sophisticated update language is needed to handle all the language features. Many of these features, however, are introduced for the syntactic convenience of programmers, they do not add any functional features to the language. Any Haskell program can be translated into a core subset of Haskell, close to a bare lambda calculus. The translation procedure is defined in [67]. We will first introduce an update calculus for the core (but complete with regard to expressive power) subset of all functional language. A particular extension for Haskell, HULA will be presented as well. Principally, the core calculus can be used as a foundation for update languages of any functional language.

4.2 The Update Calculus

4.2.1 The Object Language

We consider lambda calculus together with a standard Hindley/Milner type system as the working object language. The syntax of lambda-calculus expressions is
shown in Figure 4.1. In addition to expressions $e$, we use $v$ to range over variables. For simplicity, we omit constants here.

$$
e ::= v \mid e \mid \lambda v.e \mid \text{let } v=e \text{ in } e$$

Figure 4.1: Abstract Syntax of Lambda Calculus

Types are built from type variables (denoted by $a$) and function types (see Figure 4.2). Type schemas are used to enable polymorphic typing for let-bound variables. We abbreviate a list of type variables $a_1 \ldots a_n$ by $\bar{a}$.

$$
t ::= a \mid t \rightarrow t \mid \forall \bar{a}.t$$

Figure 4.2: Types for Lambda Calculus

$FV$ gives the set of free variables of an expression, a type, or a type environment. Likewise, $BV$ computes bound variables. We denote by $[v \mapsto w]e$ the capture-free substitution of the variable $v$ by the variable $w$ in the expression $e$.

The type system defines judgments of the form $\Gamma \vdash e : t$ where $\Gamma$ is a type assumption, that is, a list of pairs given by $(v)$ and type schemas $(s)$. The inference rules shown in Figure 4.3 are standard except for the rule META, which defines the typing of metavariables and which is explained below in Section 4.2.2.

Since the theory of program updates is independent of the particular dynamic semantics of the object language (call-by-value, call-by-need, ...), we do not have to consider a dynamic semantics in the context of this dissertation.

The main idea to achieve a manageable update mechanism is to perform somehow “coordinated” updates of the definition and all corresponding uses of a symbol
\begin{align*}
\text{VAR}_v & : \quad \Gamma(v) = \forall \alpha.t' \quad [a_i \mapsto t_i][t'] = t \\
\Gamma \vdash v : t \\
\text{APP}_v & : \quad \Gamma \vdash e : t' \rightarrow t \\
\Gamma \vdash e' : t' \\
\Gamma \vdash e \ e' : t \\
\text{LET}_v & : \quad \{\bar{a}\} = FV(t') - FV(\Gamma) \\
\Gamma, v : t' \vdash e' : t' \\
\Gamma, v : \forall a.t' \vdash e : t \\
\Gamma \vdash \textbf{let } v=e' \textbf{ in } e : t \\
\text{META}_v & : \quad \Gamma(m) = t \\
\Gamma \vdash m : t \\
\end{align*}

Figure 4.3: Type System for Lambda Calculus

in a program. We therefore consider the available forms of symbol definitions in more detail. In general, a definition has the following form.

\texttt{let } v=d \texttt{ in } e

Here \( v \) is the symbol (variable) being defined, \( d \) is the defining expression, and \( e \) is the scope of the definition, that is, \( e \) is an expression in which \( v \) will be used with the definition \( d \) (unless hidden by another nested definition for \( v \)). We call \( v \) the \textit{symbol}, \( d \) the \textit{defining expression}, and \( e \) the \textit{scope} of the definition. If no confusion can arise, we sometimes refer to \( d \) also as the \textit{definition} (of \( v \)). \( \beta \)-redexes also fit the shape of a definition since a (non-recursive) \texttt{let } v=d \texttt{ in } e is just an abbreviation for \( (\lambda v->e) \ d \). However, the treatment of \texttt{let} differs from functions in the type system (allowing polymorphism) and also in the update language since it allows recursive definitions.
Several extensions of lambda calculus that make it a more realistic model for a language like Haskell also fit the general pattern of a definition, for example, data type/constructor definitions and pattern matching rules. We have demonstrated this idea by examples in Section 1.1 of Chapter 1, and we will explain the relationship in more detail based on the presented update calculus in Section 4.2.2.

4.2.2 Syntax

The update calculus basically consists of rewrite rules and a scope-aware update operation that is able to perform updates of the definition and uses of a symbol. In addition, we need operations for composing alternative updates and for recursive application of updates.

Rules

A rewrite rule has the following form.

\[ l \leadsto r \]

Here \( l \) and \( r \) are patterns, which are basically expressions that might contain metavariables \( m \). Metavariables are different from object variables and can represent arbitrary expressions. The syntax of patterns is defined in Figure 4.4.

If we remove metavariables, patterns reduce to expressions that do not introduce bindings. Binding constructs will be updated by a special form, scope update,
that takes care of the peculiarities of free/bound/fresh variables and their types that can occur with updates. Therefore, rewrite rules are restricted to patterns. The typing rule for metavariables in Figure 4.3 is similar to the rule \( \text{VAR}_t \), but we only allow the binding of types (and not type schemas).

An update can be performed on an expression \( e \) by applying a rule \( l \leadsto r \) to \( e \), which means to match \( l \) against \( e \), which, if successful, results in a binding \( \sigma \) (called substitution) for the metavariables in \( l \). Formally, a substitution is a mapping from variables to expressions. The fact that a pattern like \( l \) matches an expression \( e \) (under the substitution \( \sigma \)) is also written as: \( l \triangleright e \) (\( l \triangleright e \)). We assume that \( l \) is linear, that is, \( l \) does not contain any metavariable twice. The result of the update operation is \( \sigma(r) \), that is, \( r \) with all metavariables being substituted according to \( \sigma \). If \( l \) does not match \( e \), the update described by the rule is not performed, and \( e \) remains unchanged.

We use the matching definitions and notations also for types. If a type \( t \) matches another type \( t' \) (that is, \( t \triangleright t' \)), then we also say that \( t' \) is an instance of \( t \).

**Update Combinators**

More complex updates can be built from rules by alternation and recursion. For example, the alternation of two updates \( u_1 \) and \( u_2 \), written as \( u_1 \mid u_2 \), first tries to
perform the update \( u_1 \). If \( u_1 \) can be applied, the resulting expression is also the result of \( u_1 | u_2 \). Only if \( u_1 \) does not apply, the update \( u_2 \) is tried. The \textit{composition} of two updates \( u_1 \) and \( u_2 \), written as \( u_1 ; u_2 \), applies the two updates in that order, even if \( u_1 \) can be applied. \textit{Recursion} is needed to move updates arbitrarily deep into expressions. For example, since a rule is always tried at the root of an expression, an update like \( 1 \sim 2 \) has no effect when applied to the expression \( 1+(1+1) \). We therefore introduce a recursion operator \( \downarrow \) that causes its argument update to be applied (in a top-down manner) to all subexpressions. For example, the update \( \downarrow (1+1 \sim 1) \) applied to \( 1+(1+1) \) results in the expression \( 1+1 \). We use the recursion operator only implicitly in scope updates and do not offer it to the user.

The update operations described thus far do not take into account the scope of identifiers; they are rather like global search-and-replace rules. In contrast to global updates, scope updates always operate only on the uses of a symbol introduced by a particular definition.

In a \textit{scope update}, each element of a definition \texttt{let v=d in e}, that is, \( v, d, \) or \( e \), can be changed. Therefore, we need an update for each part. The update of the variable can just be a simple renaming, but the update of the definition and of the scope can be given by arbitrarily complex updates. We use the syntax \( \{v\sim v':u_d\} u_u \) for an update that renames \( v \) to \( v' \), changes \( v' \)'s definition by \( u_d \), and all of its uses by \( u_u \). We call \( v\sim v' \) the \textit{name update}, \( u_d \) the \textit{definition update}, and \( u_u \) the \textit{use update}. Note that \( u_u \) is always applied recursively, whereas \( u_d \) is only applied to the root of the definition. However, to account for recursive \texttt{let} definitions we apply \( u_u \) also recursively to the result obtained by the update \( u_d \).
We use \( x \) to range over variables \( (v) \) and metavariables \( (m) \), which means that we can use a scope update to update specific bindings (by using an object variable) or to apply to arbitrary bindings (by using a metavariable). Either one of the variables (but not both) can be missing from the name update. These special cases describe the creation \( (\leadsto v') \) or removal \( (v\leadsto) \) of a binding. In both cases, the definition update is replaced by an expression, which is optional for the binding creation but required in the case of binding removal because it is needed to replace all occurrences of the removed variable. Note that \( e' \) must not contain the variable that is to be removed. In the case of binding creation, for example, when applying \( \{\leadsto v = e'\} u_a \) to \( e \), \( e' \) is optional and is used, if present, to create an expression

\[
\text{let } v = e_1 \text{ in } e_2 \quad \text{where } e_1 \text{ is the result of applying } u_a \text{ to } e' \text{ and } e_2 \text{ is the result of applying } u_u \text{ to } e.
\]

Otherwise, that is, if \( e' \) is missing, the result is \( \backslash v \rightarrow e_2 \).

At first sight it seems that we also need a combinator to generate fresh variables in order to rename variables and to create new definitions. However, we know exactly all the places of an update where fresh variables are required, namely only in a renaming update or a binding creation, so that we can integrate the generation of fresh variables into the semantics for updates.

The syntax of updates is summarized in Figure 4.5.

We use an abbreviated notation for scope updates that do not change names, that is, we write \( \{v: u_d\} u_a \) instead of \( \{v\leadsto v: u_d\} u_a \). The updates of either the defining expression or the scope can be empty, which means that there is no update for that part. The updates are then simply written as \( \{v\leadsto v': u_d\} \) and \( \{v\leadsto v': u\} \), respectively, and are equivalent to updates \( \{v\leadsto v': u_d\} u \) and \( \{v\leadsto v': u\} u_a \), respec-
\[
\begin{array}{ll}
  u ::= & t \quad \text{Identity} \\
         | & p \rhd p \quad \text{Rule} \\
         | & \{x \rhd x; u\} u \quad \text{Change Scope} \\
         | & \{\lhd v[\doteq e]\} u \quad \text{Insert Scope} \\
         | & \{x \rhd e\} u \quad \text{Delete Scope} \\
         | & u \cdot u \quad \text{Composition} \\
         | & u \mid u \quad \text{Alternative} \\
         | & \downarrow u \quad \text{Recursion}
\end{array}
\]

Figure 4.5: Syntax of Updates

tively.

Let us consider some examples. We already have seen examples for rules. A simple example of change scope is an update for consistently renaming variables.

\[
\{v \rhd w\} v \rhd w
\]

This update applies to a lambda- or let-bound variable \( v \) and renames it and all of its occurrences that are bound by that definition to \( w \). The definition of \( v \) is usually not changed by this update. However, if \( v \) has a recursive definition, references to \( v \) in the definition will be changed to \( w \), too, because the use update is also applied to the definition of a symbol.

Recall the function generalization update from Section 1.1. A generalization of a function \( f \) can be expressed by the following update \( u \).

\[
\{f; \{\lhd w\} 1 \lhd w\} f \lhd f \ 1
\]

\( u \) is a change-scope update for \( f \), which does not rename \( f \), but whose definition update (\( \{\lhd w\} 1 \lhd w \)) introduces a new variable \( w \) and replaces all occurrences of a particular constant expression (here \( 1 \)) by \( w \) in the definition of \( f \). The use update of \( u \), that is, \( f \lhd f \ 1 \), ensures that all uses of \( f \) are extended by supplying a
new argument for the newly introduced parameter. Here we use the same expression that was generalized in \( f \)'s definition, which preserves the semantics of the program. Although we can express a particular function generalization with the update calculus, the definition of a reusable update function like \texttt{genFun} requires the extension of the update calculus by abstraction, application, and variables, which is not difficult and which we have omitted here for simplicity.

To express the size update example in the update calculus we have to extend the object language by constructors and \texttt{case} expressions and the update calculus by corresponding constructs, which is rather straightforward. An interesting aspect is that each alternative of a \texttt{case} expression is a separate binding construct that introduces bindings for variables in the pattern. The scope of the variables is the corresponding right-hand side of the \texttt{case} alternative. Since these variables do not have their own definitions, we can represent each \texttt{case} alternative by a lambda abstraction—just for the sake of performing an update. A \texttt{case} update can then be translated into an alternative of change-scope updates. For example, the translation of the size update yields the following update.

\[
\begin{array}{c}
\{\text{Node} : t \to \text{Int} \to t\} \\
\quad (\{\text{Node}\}((\text{s})\text{Node} \to \text{Node} (\text{succ} \ s))) | \\
\quad \{\text{Leaf}\}\text{Node} \to \text{Node} 1) | \\
\quad \text{Node} \to \text{Node} 1
\end{array}
\]

The outermost change-scope update expresses that the definition of the \texttt{Node} constructor, which is a type bound to the metavariable \( m \), is extended by \texttt{Int}. The use update is an alternative whose second part expresses to extend all \texttt{Node} expressions by \( 1 \) to accommodate the type change of the constructor. The first
alternative is itself an alternative of two change-scope updates. (Since the $|$ operation is associative, the brackets are strictly not needed.) The first alternative applies to definitions of Node which (by way of translation) can only be found in lambda abstractions representing case alternatives. The new-scope update will add another lambda-binding for s, and the use update extends all Node expressions by the expression succ s. The other alternative applies to lambda abstractions representing Leaf patterns.

This last example demonstrates that the presented update calculus is not restricted to deal just with lambda abstractions or let bindings, but rather can serve as a general model for expressing changes to binding constructs of all kinds.

4.2.3 Semantics

In the definition of the semantics for alternative updates and recursion we need to know whether an update $u$ is applicable to an expression $e$, which is the case if the semantics can be used to derive a result expression, that is, $\exists e' : [u](e) = e'$. Otherwise, when the semantics gets stuck, we say that $u$ is not applicable and write $[u](e) = \bot$. As an abbreviation for the semantics rules we use the update operation “try $u$” that tries to apply the update $u$ to an expression $e$ and returns the possibly changed expression if $u$ is applicable to $e$. However, if $u$ is not applicable to $e$, try $u$ yields $e$.

Generally, an update $u$ is applied to an expression $e$ recursively matching the structure of $u$ and $e$. In the procedure of application, there are three issues we
need to consider.

First, at the point where a rule is applied to an expression, the rule is applicable only if all the free (object) variables on the left-hand side of the rule are in scope of a enclosing change or delete update. Consider, for example, the rule \( f \leadsto f_1 \), which is the use update of the function generalization. Here, \( f \) is a free variable of the rule and is contained in the scope of the current update because the scope update extends the scope by \( f \). We denote this set of variables as \( \rho_s \).

We also have to consider the set of variables that are bound by the expression being updated but that are not in scope of the update, because in the semantics definition we have to ensure that newly introduced bound variables are fresh, i.e., they must not yet be bound in the updated expression. We denote this set of variables as \( \rho_B \). As an example, consider the situation when we apply the function generalization to a function definition for \( f \) that is local to a function definition for \( w \). In this case, it might happen that \( f \)’s definition contains a reference to \( w \). Now if we extended \( f \)’s definition by a new parameter \( w \), all those references would be illegally captured by that parameter and would not refer to the enclosing \( w \) anymore. To prevent this name capture, we have to ensure that we use \( w \) only if it is not bound in the current program to be updated; otherwise, we have to rename \( w \) appropriately. We also require that all the free (object) variables of the right-hand side of a rule are contained in the set of bound variables to prevent the creation of unbound variables.

Moreover, consider the case when a change or delete update is applied. If the bound variable is a metavariable, we need to match the metavariable against the
object variable in the expression and replace all the occurrences of that metavariable in the update by the object variable. If the bound variable is an object variable, it has to be the same variable in the expression.

We define the semantics in two steps. In the first step, we instantiate a general update based on the expression being applied, checking bound variables, replacing metavariables, and renaming fresh variables. In step two, this update is applied to the expression according to the semantics defined in Figure 4.6.

Instantiation takes place in a context of the two aforementioned sets of variables represented by a two-set partition \( \rho = (\rho_S, \rho_B) \) where \( \rho_S \) contains the variables that are in scope of the update and \( \rho_B \) contains the variables that are bound in the updated expression but that are not in scope of the update. We use two operations for moving variables between \( \rho_S \) and \( \rho_B \):

\[
(\rho_S, \rho_B)(v) := (\rho_S \cup \{v\}, \rho_B - \{v\})
\]

\[
(\rho_S, \rho_B)\{v\} := (\rho_S - \{v\}, \rho_B \cup \{v\})
\]

We use the notation \( v^\epsilon \rho w \) to express the fact that \( w \) is a variable that is fresh with respect to the expression \( e \) and the environment \( \rho_B \). This is a variable that neither is bound in \( e \) nor occurs in the current context \( (\rho_B) \). If \( v \) has this property, \( w = v \), otherwise an appropriate name will be constructed (for example, by repeatedly appending a prime symbol until a fresh symbol is found). It is not a problem when \( v \) occurs in \( \rho_S \), because in that case the \( v \) in \( \rho_S \) will be either renamed or deleted. We use the abbreviation \( v(x \sim x')^\epsilon \rho w \) that expresses the condition that \( x \)
matches the bound variable $v$ and the fresh variable generated from $x'$ is $w$. This predicate formalizes two steps of a scope update: (1) the initial matching of the variable $v$ to which the scope update is applied, and (2) providing a fresh variable, which might be $w$, to rename $w$. The predicate is defined as follows.

$$v(x \leadsto x')^e w \iff x \succ v \land \sigma(x')^e_w$$

The predicate covers two cases: First, if $x$ is a metavariable, say $m$, we require that if $x'$ is a metavariable, then $x' = x = m$. In this situation we obtain $v(m \leadsto m)^e_w$, which is always satisfied. Second, $x$ is the object variable $v$ and $x' = v'$ (where $v'$ might be $v$) or $x = m$ and $x' = v'$. Then $v(v \leadsto v')^e_w$ is satisfied by $v')^e_w = w$.

We also use the notations $[v(x)]u$ for the update $u$ with all free left occurrences of $x$ substituted by $v$, $[w]x']u$ for the update $u$ with all free right occurrences of $x'$ substituted by $w$, and $[v(x, w) x']u$ for $[w]x']([v(x)]u)$.

The judgment $u \xrightarrow{e} u'$ denotes that update $u$ is instantiated to $u'$ with respect to the expression $e$ under the context $\rho$. For a rule update, it is necessary to ensure the free variables are bound:

$$\text{FV}(l) \subseteq \rho_S \quad \text{FV}(r) \subseteq \rho_B$$

$$l \leadsto r \xrightarrow{e} l \leadsto r$$

In the case of change updates, metavariables, if any, will be matched and re-
placed. Newly introduced variables will be renamed if necessary:

\[
\frac{v(x \rightsquigarrow x')\rho^d_w}{\{x \rightsquigarrow x': u_d\}_u} \quad \frac{\{v(x, w')x\}_u\rho^d_w u'_d}{\{v(x, w')x\}_u e \rho^d_w u'_d}
\]

Note that the use of \((e\ d)\) in the above rule is to ensure the freshness of \(w\) with respect to both \(e\) and \(d\), because a \texttt{let} expression can bind a recursively defined function. Two more rules are needed for instantiating a change update, which are for beta redex and lambda abstractions. They only differ in that \(d\) is not needed in the first two judgments. Instantiation for insert and delete updates are also very similar and omitted. After instantiation, the update is applied based on the rules defined in Figure 4.6.

The semantics definition of top-down recursion uses an auxiliary operation \(\mu\) that applies updates recursively to all subexpressions.

We demonstrate the semantics with a small example. Let \(u = \{f: u_d\} u_u\) where \(u_d = \{\rightsquigarrow w\} 1 \rightsquigarrow w\) and \(u_u = f \rightsquigarrow f\ 1\) and let \(e_0 = \texttt{let}\ d\ \texttt{in}\ e\) where \(d = \\lambda v \rightarrow v + 1\) and \(e = f\ 3\). We first show that the result of instantiating \(u\) with regard to \(e_0\) and an empty context \((\varnothing, \varnothing)\) is \(u\) itself.

\[
u \quad \frac{e_0}{\{u\}_{(\varnothing, \varnothing})}
\]

This relationship can be inferred from the following facts.

1. \(f(1 \rightsquigarrow f)\{e\ d\}_{(\varnothing, \varnothing)} f\)
Figure 4.6: Semantics of Updates
2. \[ f(f, f)f(\{\sim w\}1 \sim w) = \{\sim w\}1 \sim w \text{ and } \{\sim w\}1 \sim w \xrightarrow{e} \{\sim w\}1 \sim w \]

3. \[ f(f, f)f(\sim f \ 1) = f \sim f \ 1 \text{ and } f \sim f \ 1 \xrightarrow{d} f \sim f \ 1 \]

We then derive the result of applying \( u \) to \( e_0 \).

\[ \llbracket u \rrbracket(e_0) = \text{let } f = w \rightarrow \langle v, v+w \rangle \text{ in } f \ 1 \ 3 \]

This expression can be obtained by applying \( \{\}_{1}^{chgs} \) and using the following premises.

4. \[ \llbracket u \rrbracket(\llbracket u \rrbracket(\langle v, v+1 \rangle)) = \langle w, w \rightarrow v \rangle \]

5. \[ \llbracket u \rrbracket(1 \ 3) = f \ 1 \ 3 \]

(4) can be further obtained by showing:

6. \[ \llbracket u \rrbracket(\langle v, v+1 \rangle) = \langle w, w \rightarrow v \rangle \]

7. \[ \llbracket u \rrbracket(\langle w, w \rightarrow v \rangle) = \langle w, w \rightarrow v \rangle \]

(6) is obtained by applying \( \{\}_{1}^{ins} \) with the following premises.

8. \[ \llbracket 1 \sim w \rrbracket(\langle v, v+1 \rangle) = \langle v \rightarrow v \rangle \]

(8) can further be obtained by applying \( \llbracket 1 \rrbracket \) to the following premises:

9. \[ \llbracket \text{try } (1 \sim w) \rrbracket(\langle v, v+1 \rangle) = \langle v \rightarrow v+1 \rangle \]

10. \[ \llbracket \mu(1 \sim w) \rrbracket(\langle v, v+1 \rangle) = \langle v \rightarrow v \rangle \]
(9) is a simple application of TRY[1] and \( \sim[1] \). (10) can be obtained by repeatedly applying the congruence rules of \( \mu[1] \) and \( \downarrow[1] \).

The goal of the update calculus is to provide a means for updating programs without introducing type errors. However, there is no practical way to analyze the semantics and logic of the object program. Logic errors might be introduced in the updated program, and it is the programmer’s responsibility to manually change the updated program to eliminate such logic errors. For example, in the update of extending the tree node with a size field, as demonstrated in Section 1.1, \texttt{succ} is introduced as a new first argument for the constructor \texttt{Node}, because this update is intended to be applied to the \texttt{insert} function that increases the size of the node by 1. However, imagine an \texttt{insert} function that does not insert duplicate elements. In this case, the size of the node does not always increase, so that some \texttt{Node} expressions might receive incorrect size values after the update.

4.2.4 Type System

The goal of the type system for the update calculus is to find all possible type changes that an update can cause to an arbitrary object program. We show that if these type changes “cover” each other appropriately, then the generated object program is guaranteed to be type correct.
Type Changes

Since updates denote changes of expressions that may involve a change of their
types, the types of updates are described by type changes. A type change (δ) is
essentially given by a pair of types (t~t), but it can also be an alternative of type
changes (δ | δ).

For example, the type change of the update 1 ~ True is Int ~ Bool, while the
type change of 1 ~ True | odd ~ 2 is Int ~ True | Int->Bool ~ Int.

Recursively applied updates might cause type changes in subexpressions that
affect the type of the whole expression. Possible dependencies of an expression’s
type on that of its subexpressions are expressed using the two concepts of type
hooks and context types. For example, the fact that the type of odd 1 depends
on the type of 1 is expressed by the hook Int ~ Bool, the dependency on odd is
Int->Bool ~ Bool. The dependency on the whole expression is by definition empty
(ε), and a dependency on any expression that is not a subexpression is represented
by a “constant hook” ~ Bool.

The application of a type hook C to a type t yields a context type denoted
by C\{t\} that exposes t as a possible type in a type derivation. The meaning of a
context type is given by the following equations.

\[ \epsilon(t) = t \]
\[ ~t_2(t) = t_2 \]
\[ t_1 ~ t_2(t) = \begin{cases} 
        t_2 & \text{if } t \succ t_1 \\
        \text{error} & \text{otherwise}
\end{cases} \]
The rationale behind context types is to capture changes of types that possibly happen only in subexpressions and do not show up as a top-level type change. Context types are employed to describe the type changes for use updates in scope updates. For example, the type change of the update \( u' = 1 \leadsto w \) is \( \text{Int} \leadsto a \). However, when \( u' \) is used as a use update of a scope update \( u = \{ \leadsto w \} 1 \leadsto w \), it is performed recursively, so that the type change is described using a context type \( C(\text{Int}) \leadsto C(a) \).

To describe the type change for \( u \), the type for the newly introduced abstraction has to be taken into account. Here we observe that the type of \( w \) cannot be \( a \) in general, because \( w \) might be, through the recursive application of the rule, placed into an expression context that constrains \( w \)'s type. For example, if we apply \( u \) to \( \text{odd} \ 1 \), we obtain \( \\backslash w \rightarrow \text{odd} \ w \) where \( w \)'s type has to be \( \text{Int} \). In general, the type of a variable is constrained to the type of the subexpression that it replaces. We can use a type hook that describes a dependency on a type of a subexpression \( e \) to express a constraint on a type variable that might replace \( e \). Such a constrained type is written as \( a_{|C} \). Its meaning is to constrain a type variable \( a \) by the type of a subexpression (represented by the left part of a type hook):

\[
\begin{align*}
    a_{|t_1 \mapsto t_2} &= t_1 \\
    t_{|t_1 \mapsto t_2} &= t \\
    t_{|e} &= t
\end{align*}
\]

The type change for \( u \) is therefore given by \( C(\text{Int}) \leadsto a_{|C} \rightarrow C(a_{|C}) \).

To see how type hooks, context types, and constrained types work, consider
the application of \( u \) to \( 1 \), which yields \( \text{w}\rightarrow\text{w} \). The corresponding type change \( \text{Int}\rightarrow\text{a}\rightarrow\text{a} \) is obtained using the type hook \( \epsilon \). However, applied to odd \( 1 \), \( u \) yields \( \text{w}\rightarrow\text{odd}\text{ w} \) with the type change \( \text{Bool}\rightarrow\text{Int}\rightarrow\text{Bool} \), which is obtained from the type hook \( \text{Int}\rightarrow\text{Bool} \). As another example consider the renaming update \( u = \{ \text{v}\rightarrow\text{w} \}\text{v}\rightarrow\text{w} \). For the update we obtain a type change \( C(\text{a}|C)\rightarrow C(\text{b}|C) \), which is the same as \( C(\text{a}|C)\rightarrow C(\text{a}|C) \) because the scopes of the two type variables are distinct. The type hook \( C \) results for the same reason as in the previous example. Applying \( u \) to the expression \( \text{w}\rightarrow\text{1} \) yields \( \text{w}\rightarrow\text{1} \) with a type change \( \text{a}\rightarrow\text{Int}\rightarrow\text{a}\rightarrow\text{Int} \), which is obtained by using the type hook \( \rightarrow\text{a}\rightarrow\text{Int} \). Similarly, \( \text{w}\rightarrow\text{v} \) is mapped by \( u \) into \( \text{w}\rightarrow\text{w} \) with a type change \( \text{a}\rightarrow\text{a}\rightarrow\text{a} \rightarrow\text{a} \), which is an instance of \( u \)'s type change for the context \( \epsilon \). Finally, \( u \) changes \( \text{w}\rightarrow\text{odd}\text{ v} \) to \( \text{w}\rightarrow\text{odd}\text{ w} \) with a type change \( \text{Int}\rightarrow\text{Bool}\rightarrow\text{Int}\rightarrow\text{Bool} \). This type change is \( u \)'s type change specialized for the type hook \( \text{Int}\rightarrow\text{Int}\rightarrow\text{Bool} \).

The syntax of hooks, contexts, and type changes is defined in Figure 4.7. To summarize, a type change can be given by a pair of simple types \( t\rightarrow t \) or by a pair of the more general context or constrained types (“extended types”), that is, \( \tau\rightarrow\tau \). A type change can also be given by an alternative of two type changes \( \delta\mid\delta \).

To explain the meaning of nested contexts, we define the composition of type

\[
\begin{align*}
\delta & ::= \tau\rightarrow\tau \mid \delta\mid\delta \quad \text{Type change} \\
\tau & ::= t \mid \tau\rightarrow\tau \mid C(\tau) \mid \tau|C \quad \text{Extended type} \\
C & ::= \epsilon \mid \triangleright t \mid t\triangleright t \quad \text{Type hook}
\end{align*}
\]

Figure 4.7: Type Changes
hooks as follows.

\[
\begin{align*}
\epsilon \cdot C & := C \\
C \cdot \epsilon & := C \\
\left( t_1 \leftarrow t_2 \right) \cdot \left( t'_1 \leftarrow t'_2 \right) & := \\
& \begin{cases}
  t'_1 \leftarrow t_2 & \text{if } t'_2 \supseteq t_1 \lor t_1 \text{ is empty} \\
  \text{error} & \text{otherwise}
\end{cases}
\end{align*}
\]

It is easy to verify that with this definition we obtain the following two equalities.

\[
\begin{align*}
C \cdot C''(t) & = C'\langle C''(t) \rangle \\
t |_{C \cdot C''} & = (t|_C)_{|_{C''}}
\end{align*}
\]

The effect is that the outermost hooks affect context types, whereas the innermost hooks are relevant for constrained types. Finally, since the inference rules generate, in general, context constraints for arbitrary type changes, we have to explain how contexts extend to type changes and alternative type changes.

\[
\begin{align*}
C\langle \tau \rightsquigarrow \tau' \rangle & := C\langle \tau \rangle \rightsquigarrow C\langle \tau' \rangle \\
C\langle \delta \mid \delta' \rangle & := C\langle \delta \rangle \mid C\langle \delta' \rangle
\end{align*}
\]

The definitions of free variables and generic instances extend naturally to type hooks, context types, and constrained types.

Types and type changes can be \textit{applicative instances} of one another. This relationship says that a type \(t\) is an applicative instance of a function type \(t' \rightarrow t\), written as \(t \triangleright t' \rightarrow t\). The rationale for this definition is that two updates \(u\) and
of different type changes \( t_1 \sim t_2 \) and \( t'_1 \sim t'_2 \), respectively, can be considered well typed in an alternative \( u | u' \) if one type change is an applicative instance of the other, that is, if \( t_1 \sim t_2 \succeq t'_1 \sim t'_2 \) or \( t'_1 \sim t'_2 \succeq t_1 \sim t_2 \), because in that case one update is just more specific than the other. Consider, for example, the following update.

\[
\{f : \text{succ} \sim \text{plus}\} f \ m \sim f \ m \ 1 | f \sim f \ 1
\]

The first rule of the alternative \( f \ m \sim f \ m \ 1 \) has the type change \( \text{Int} \sim \text{Int} \) whereas the second rule \( f \sim f \ 1 \) has the type change \( \text{Int} \rightarrow \text{Int} \sim \text{Int} \rightarrow \text{Int} \). Still both updates are compatible in the sense that the first rule applies to more specific occurrences of \( f \) than the second rule. This fact is reflected in the type change \( \text{Int} \sim \text{Int} \) being an applicative instance of \( \text{Int} \rightarrow \text{Int} \sim \text{Int} \rightarrow \text{Int} \). The relationship is defined in Figure 4.8.

![Figure 4.8: Applicative Instance](image)

**Type-Change Inference**

The type changes that are caused by updates are described by judgments of the form \( \Delta \triangleright u :: \delta \) where \( \Delta \) is a set of type-change assumptions, which can take one
of three forms:

(1) \( x \leadsto x' :: \tau \leadsto \tau' \) expresses that \( x \) of type \( \tau \) is changed to \( x' \) of type \( \tau' \). The following constraint applies: if \( x' \) is a metavariable, then \( x' = x \) and \( \tau' = \tau \).

(2) \( v : \tau \) expresses that \( v \) is a newly introduced (object) variable of type \( \tau \).

(3) \( x : \ell \tau \) expresses that \( x \) is an (object or meta) variable of type \( \tau \) that is only bound in the expression to be changed.

Type-change assumptions can be extended by assumptions using the “comma” notation as in the type system.

The type-change system builds on the type system for the object language. In the typing rule for rules we make use of projection operations that project on the left and right part of a type-change assumption. In computing these projections we have to map extended types \( \tau \) to simple types \( t \), which is performed by dropping potential hook variables and constraints as follows.

\[
\begin{align*}
| t | & = t \\
| C(\tau) | & = | \tau | \\
| \tau_C | & = | \tau | \\
| \tau_1 \rightarrow \tau_2 | & = | \tau_1 | \rightarrow | \tau_2 |
\end{align*}
\]

Now the projections are defined as follows:

\[
\begin{align*}
\Delta_\ell & := \{ x : | \tau | \mid x \leadsto x' :: \tau \leadsto \tau' \in \Delta \} \cup \{ x : | \tau | \mid x : \ell \tau \in \Delta \} \\
\Delta_\tau & := \{ x' : | \tau' | \mid x \leadsto x' :: \tau \leadsto \tau' \in \Delta \} \cup \{ x' : | \tau' | \mid x' : \ell \tau' \in \Delta \}
\end{align*}
\]
The type-change rules are defined in Figure 4.9. The rules for creating or deleting a binding have to insert a function argument type on either the right or the left part of a type change. This type insertion works across alternative type changes; we use the notation $\tau \vdash^\tau \delta$ ($\tau \vdash \delta$) to extend the argument (result) type of a type change to a function type. The definition is as follows.

$$\begin{align*}
\tau \vdash^\tau (\tau_l \leadsto \tau_r) & := (\tau \rightarrow \tau_l) \rightarrow \tau_r \\
\tau \vdash^\tau (\tau_l \leadsto \tau_r) & := \tau_l \rightarrow (\tau \rightarrow \tau_r) \\
\tau \vdash^\tau (\delta \mid \delta') & := (\tau \vdash^\tau \delta) \mid (\tau \vdash^\tau \delta') \\
\tau \vdash^\tau (\delta \mid \delta') & := (\tau \vdash \delta) \mid (\tau \vdash \delta')
\end{align*}$$

The inference rule $\leadsto$ connects the type system of the underlying object language (lambda calculus) with the type-change system. This rule is rather simple since rule updates cannot contain binding constructs, which means that all variables used in either $e$ or $e'$ have to be brought into $\Delta$ by scope updates.

We demonstrate the type-change rules by a small example. Again, let $u = \{ f : \leadsto w \} 1 \leadsto w \} f \leadsto f \; 1$. We will now show that

$$\emptyset \triangleright u :: C(\text{Int} \leadsto \text{Int})$$

This type change can be inferred from $\{ \} \vdash^\tau$ and the following premises:

1. $f \leadsto f :: C(\text{Int} \leadsto \text{Int}) \rightarrow C(\text{Int}) \triangleright \{ \leadsto w \} 1 \leadsto w :: C(\text{Int}) \leadsto \text{Int} \rightarrow C(\text{Int})$

2. $f \leadsto f :: C(\text{Int} \leadsto \text{Int}) \rightarrow C(\text{Int}) \triangleright f \leadsto f \; 1 :: \text{Int} \leadsto \text{Int}$
\[
\begin{align*}
\Delta \vdash p : \tau & \quad \Delta \vdash p' : \tau' \\
\Delta \triangleright p \leadsto p' & \implies \tau \leadsto \tau'
\end{align*}
\]

\[
\begin{align*}
\Delta \triangleright u :: \delta & \quad \Delta \triangleright u' :: \delta' & \quad \delta \lessdot \delta'' & \quad \delta' \lessdot \delta'' \\
\Delta \triangleright u | u' :: \delta' & \quad \Delta \triangleright u | u' :: \delta'' \\
\Delta \triangleright u :: \delta & \quad \Delta \triangleright u' :: \delta' \\
\Delta \triangleright u :: \delta & \quad \Delta \triangleright u' :: \delta'
\end{align*}
\]

\[
\begin{align*}
\Delta \triangleright u_1 :: \tau \leadsto \tau & \quad \Delta \triangleright u_2 :: \tau \leadsto \tau' \\
\Delta \triangleright u_1 ; u_2 :: \tau \leadsto \tau' \\
\{;}_{\triangleright}^{\text{chg}} \Delta, x \leadsto x' :: \tau \leadsto \tau' \triangleright u_d :: \tau \leadsto \tau' & \quad \Delta, x \leadsto x' :: \tau \leadsto \tau' \triangleright u_u :: \delta \\
\Delta \triangleright \{x \leadsto x': u_d\} u_u :: C(\delta) \\
\{;}_{\triangleright}^{\text{ins}} \Delta, w : \tau \vdash e : \tau & \quad \Delta, w : \tau \triangleright u :: \delta \\
\Delta \triangleright \{\leadsto w = e\} u :: C(\delta) \\
\{;}_{\triangleright}^{\text{del}} \Delta, x : \tau \triangleright u :: \delta & \quad \Delta, \vdash e : \tau \\
\Delta \triangleright \{x \leadsto e\} u :: \tau \triangleright C(\delta)
\end{align*}
\]

Figure 4.9: Type Change System

(2) results from an application of $\leadsto_{\triangleright}$, and (1) can be obtained by applying $\{;}_{\triangleright}^{\text{ins}}$ and the following premise:

(3) $w : \text{Int} \triangleright 1 \leadsto w :: \text{Int} \leadsto \text{Int}$

which is, again, simply an application of $\leadsto_{\triangleright}$.

One major limitation to the update calculus is the composition. It is not practical to foresee the object program when we analyze the updates statically. Therefore we do not know how the first update will change the program. Hence, in order to guarantee the safety of the update, we require that the first update in
a composed update does not have a type change.

Soundness of the Update Type System

In this section we define a class of well-structured updates that will preserve the well-typing of transformed object-language expressions. An update that, when applied to a well-typed expression, yields again a well-typed expression is called safe (see Chapter 1). In other words, we will show that typeable well-structured updates are safe. The structure condition captures the following two requirements:

(A) An update of the definition of a symbol that causes a change of its type or its name is accompanied by an update for all the uses of that symbol (with a matching type change).

(B) No use update can introduce a non-generalizing type change, that is, for each use update that has a type change $\tau \leadsto \tau' | \delta$ we require that $|\tau|$ is a generic instance of $|\tau'|$ or that one extended type, $\tau$ or $\tau'$, is an applicative instance of the other.

Condition (A) prevents ill-typed applications of changed symbols as well as unbound variables whereas (B) prevents type changes from breaking the well typing of their contexts. An intuitive explanation of why these conditions imply safety for well-typed updates can be obtained by looking at all the possible ways in which an update can break the type correctness of an expression and how these possibilities are prevented by the type system or the well-structuring constraints. We can find
out about possible type errors by looking at the type system for lambda calculus (see Figure 4.3): Essentially, type inference fails either in the rule $\text{VAR}_{\lambda}$ if the type for a variable cannot be found in the type environment or in the rule $\text{APP}_{\lambda}$ if the parameter type of a function does not agree with the type of the argument to which it is applied. On the other hand, the rules $\text{ABS}_{\lambda}$ and $\text{LET}_{\lambda}$ eventually refer to $\text{VAR}_{\lambda}$- and $\text{APP}_{\lambda}$- to ensure typing constraints that might fail. Let us now consider how updates can possibly introduce these errors.

Unbound variables. The free-variable problem can be introduced into an expression by an update that renames a bound variable without accordingly renaming all references to that variables; unrenamed variables might become free, thus leading to an error in the $\text{VAR}_{\lambda}$ rule, or they might be bound by an enclosing $\lambda$ or $\text{let}$ and thus might change the type that is obtained by the $\text{VAR}_{\lambda}$ rule, thus breaking eventually the $\text{APP}_{\lambda}$ rule. These kinds of changes are prevented by condition (A) and the type-change rules. Free variables could also be introduced by rules, such as $1 \rightarrow x$ (where $x$ is not bound). However, these kinds of updates are prevented by the type-change system since we cannot derive a type change for $1 \rightarrow x$ when we have no assumption about $x$ in the type-change environment.

Incorrect application. An application can become ill typed if the type of the function or the argument changes without a corresponding change of the other part of the application. Types can be changed by rules that replace objects of one type by objects of another type as in $1 \rightarrow \text{True}$. Such a change is only problematic if it is applied to a sub-expression; otherwise, a rule cannot change only one part of an application. However, since application to sub-expressions can only happen
through the recursion in use updates, such an update is not possible since it violates the condition (B) (in the example: Int is not an instance of Bool). Types can also be changed by change-scope updates. By just changing the type of a variable, say $v$ from $\tau$ to $\tau'$, we can break the type correctness in two different ways: applications of $v$ as well as contexts of $v$ (that is, applications of other expressions to $v$) can become ill typed. Both cases are prevented by conditions (A) and (B) together with the type-change system, because (i) having an update $u = v \rightsquigarrow e$ ensures that all occurrences of $v$ are changed (not necessarily by this rule, but no occurrence of $v$ is left unchanged), and (ii) the type change $\tau \rightsquigarrow \tau'$ derived by the type system is required to be "generalizing" (that is, $\tau' \gg \tau$, see below), which ensures that $e$ fits all contexts type wise.

Let us now express the well-structuring constraint more formally. We first identify some properties of change-scope updates. Let $u = \{x \rightsquigarrow x': u_d\} u_u$ and let $x \rightsquigarrow x' :: \tau_C \rightsquigarrow \tau'_C$ be the assumption that has been used in rule $\{\cdot\}_{\text{chg}}$ to derive its type change, say $C(\tau_1 \rightsquigarrow \tau_2 | \delta)$.

1. $u$ is self-contained iff $x \neq x' \lor \tau \neq \tau' \implies \exists u', u'', p$ such that $u_u = u' | x \rightsquigarrow p | u''$.
2. $u$ is smooth iff $\tau' \gg \tau$ or $\tau \gg \tau'$ or $\tau' \gg \tau$.
3. $u$ is (at most) generalizing iff $\tau_2 \gg \tau_1$.

An update $u$ is well structured iff it is well typed and all of its contained change-scope updates are self-contained, smooth, and generalizing.
When we consider the application of a well-structured update \( u \) to a well-typed expression \( e \), the following two cases can occur: (1) \( u \) does not apply to \( e \). In this case \( e \) is not changed by \( u \) and remains well typed. (2) \( u \) applies to \( e \) and changes it into \( e' \). In this case we have to show that from the result type of \( u \) we can infer the type of \( e' \). We express this result in the following theorem.

For the theorem, we need the notion of type-conform applicability. A change-scope update \( u = \{x \mapsto x': u_d\} u_u \) or a delete-scope update \( u = \{x \mapsto e_0\} u_u \) is type-conform applicable to an expression \( e = (\lambda v \rightarrow e') \) or \( e = \text{let } v = d \text{ in } e' \) if

(a) \( u \) is applicable to \( e \)
(b) \( \downarrow u_u \) is type-conform applicable to \( e' \),
(c) \( u_d \) is type-conform applicable to \( d \), and
(d) \( \Gamma, v : t \triangleright e' :: t' \land \Gamma \vdash d : t \implies \tau \mid_{C} = t \) where \( \tau \mid_{C} \) refers to the type of \( x \) used when the type change of \( u \) is inferred, that is, it is the same \( \tau \mid_{C} \) as in the rules \( \{\vdash \text{chg} \} \) and \( \{\vdash \text{del} \} \) from Figure 4.9.

Other updates \( u \) are type-conform applicable to \( e \) if they are applicable to \( e \).

**Theorem 4 (Soundness)** If \( u \) is well structured and type-conform applicable to \( e \), then:

\[
\Delta \triangleright u :: C^\tau_1 \mapsto \tau_2 | \delta \land \Delta_t \vdash e : t_1 \implies \Delta_r \vdash [u](e) : t_2
\]

where \( t_2 = C^\tau_2 \) for some context \( C^\cdot \).
Proof. The proof is by induction over the structure of \( e \) and \( u \). First, consider the case \( e = (\lambda v \to e_0) \) \( d \) and \( u = \{ x \leadsto x': u_d \} u_u \). We can assume

\[
(1) \quad \Delta \triangleright u :: C\langle \tau_1 \leadsto \tau_2 | \delta \rangle \text{ and }
\]

\[
(2) \quad \Delta \vdash e : C\langle \tau_1 \rangle.
\]

From (2) and rule APP\(_t\) from the object-language type system it follows that

\[
\text{(T1)} \quad \Delta \vdash d : t \text{ and }
\]

\[
\text{(T2)} \quad \Delta \vdash \lambda v \to e_0 : t \to C\langle \tau_1 \rangle
\]

for some type \( t \). Next, the first rule \( \{ : \}_I^{chg} \) from the semantics of updates tells that

\[
[u]_{\rho}(e) = (\lambda v \to e') \ d' \text{ where }
\]

\[
(M) \quad v \langle x \leadsto x' \rangle_{\rho}^{e} w
\]

\[
(U1) \quad [[v \langle x, w \rangle x'] u_d]_{\rho}(d) = d', \text{ and }
\]

\[
(U2) \quad [[v \langle x, w \rangle x'] u_u]_{\rho(w)}(e_0) = e'
\]

From (1) and the type-change rule \( \{ : \}_I^{chg} \) we know that

\[
(\text{C1}) \quad \Delta, x \leadsto x' :: \tau_C \leadsto \tau'_C, \triangleright u_d :: \tau_C \leadsto \tau'_C, \text{ and }
\]

\[
(\text{C2}) \quad \Delta, x \leadsto x' :: \tau_C \leadsto \tau'_C, \triangleright u_u :: \tau_1 \leadsto \tau_2 | \delta.
\]

where we can substitute \( v \) for \( x \) and \( w \) for \( x' \) due to (M). Since \( u \) is type-conform applicable to \( e \), we know that \( \tau_C = t \). Therefore, we can conclude from (C1), (T1), (U1), and the induction hypothesis

\[
\Delta, x : \tau'_C \vdash d' : \tau'_C
\]

Since \( w \notin d' \), we also have:
\[(*) \quad \Delta_r \vdash d' : \tau'_{|C}\]

Now let us consider \(u_u\). First of all, from (M) and (C2) we can conclude

\[(C3) \quad \Delta, v \leadsto w :: \tau_{|C} \leadsto \tau'_{|C} \triangleright u_u :: C(\langle \tau_1 \leadsto \tau_2 | \delta)\]

From (T2) and rule \(\text{ABS}_{\triangleright}\) we know that

\[\Delta_{\bar{\epsilon}}, v : t \vdash e_0 : C(\langle \tau_1)\]

Since \(u_u\) is well structured and type-conform applicable to \(e_0\), we can apply the theorem inductively and obtain

\[\Delta_r, w : \tau'_{|C} \vdash e' : C(\langle \tau_2)\]

which is the same as

\[\Delta_r \vdash \lambda w \rightarrow e' : \tau'_{|C} \rightarrow C(\langle \tau_2)\]

Together with (*) it follows from rule \(\text{APP}_{\triangleright}\) that

\[\Delta_r \vdash (\lambda w \rightarrow e') d' : C(\langle \tau_2)\]

which proves this case.

Next, consider the case \(e = \text{let } v = d \text{ in } e_0\) and \(u = \{v \leadsto w : u_d\} u_u\), which is very similar to the previous case, but slightly more involved due the possible recursion. We can assume
(1) \( \Delta \triangleright u :: C(\tau_1 \rightsquigarrow \tau_2 | \delta) \) and

(2) \( \Delta \vdash e :: t_1. \)

From (2) and rule LET\(_{\ell}\) of the object-language type system it follows that

(T1) \( \Delta \vdash v :: t :: d : t \) and

(T2) \( \Delta \vdash v :: \forall \bar{a} . e_0 :: t_1 \)

for some type \( t \). Next, the last \( \{ ; \}_{\text{chg}}^\dagger \) rule from the semantics of updates tells that

\[
[u](e) = (\lambda v \to e') d'
\]

where

(U1) \( \llbracket u_u \rrbracket ([u_d](d)) = d' \), and

(U2) \( \llbracket u_u \rrbracket (e_0) = e' \)

Effectively, \( \llbracket u_u \rrbracket ([u_d](d)) \) is equivalent to \( \llbracket u_u ; u_d \rrbracket (d) \) according to the semantics definition. It can be easily verified against the definitions that \( \llbracket u_u ; u_d \rrbracket (d) \) is well structured since \( u \) is well structured and it is type-conform applicable to \( d \).

From (1) and the type-change rule \( \{ ; \}_{\dagger}^\text{chg} \) we know that

(C1) \( \Delta , v \rightsquigarrow w :: \tau_C \rightsquigarrow \tau_C' \triangleright u_d :: \tau_C ' \rightsquigarrow \tau_C' \), and

(C2) \( \Delta , v \rightsquigarrow w :: \tau_C \rightsquigarrow \tau_C' \triangleright u_u :: \tau_1 \rightsquigarrow \tau_2 | \delta \).

Since \( u \) is type-conform applicable to \( e \), we know that \( \tau_C = t \). Therefore, we can conclude from (C1), (T1), (U1), and the induction hypothesis

\[
(*) \quad \Delta , w :: \tau_C' \vdash d' :: \tau_C'
\]

Now let us consider \( u_u \). First, from (M), (C2) and type-change \( \downarrow_{\tau} \), we can conclude

(C3) \( \Delta , v \rightsquigarrow w :: \tau_C \rightsquigarrow \tau_C' \triangleright u_u :: C(\tau_1 \rightsquigarrow \tau_2 | \delta) \)
From (T2) and rule $\text{LET}_r$, we know that

$$\Delta_{\ell}, v : t \vdash e_0 : t_1$$

Since $\downarrow u$ is well-structured and type-conform applicable to $e_0$, we can apply the theorem inductively and obtain

$$\Delta_{\ell}, w : \tau'_C \vdash e' : t_2$$

by rule $\text{LET}_r$, together with (*), we can conclude that

$$\Delta_{\ell} \vdash \text{let } w = d' \text{ in } e' : t_2$$

which proves this case. Other cases can be proved similarly. \qed

Theorem 4 expresses that the derivation of a type change that includes an alternative $\tau \leadsto \tau'$ ensures for any expression $e$ of type $\tau$ that $u$ transforms $e$ into an expression of type $\tau'$. We have to use $\tau$ in the theorem because the type change for $u$ is generally given by context types. For a concrete expression $e$, the type inference will fix any type hooks, which allows $\tau$ to be simplified to a type $t$.

Let us consider the safety of some of the presented example updates. The function generalization update from Section 1.3 is safe, which can be checked by applying the definitions of "well structured" and the rules of the type-change system. The first size update (Section 1.3) is also safe, although to prove it we need the extension of lambda calculus by constructors and case expressions. In con-
trast, the second size update is not safe since the case update will be applied only to the definition of insert (and not to other functions). However, it is safe for programs that only use Node expressions in a function insert. We will discuss this aspect of conditional update safety briefly in the next section when we talk about extensions of the system.

4.2.5 Possible Extensions

Regarding the presented calculus, we believe that the following aspects are most promising with respect to extending its expressiveness and usefulness.

*Named type changes.* Currently, the type-change system only reports a possible change for the result expression. For larger updates it would be interesting to obtain, for example, renamings and type changes for all (or at least all non-local) definitions. This extension seems to be orthogonal to the current system and not difficult to realize.

*Conditional update safety.* The well-structuring conditions that are required to guarantee safety of updates are rather strict so that some useful program updates would not be classified as safe. However, in many situations, "complete" safety is not mandatory. Instead, a form of conditional safety is sufficient. For example, the second size update and the lambda calculus update could be considered to be conditionally safe in the sense that type safety is preserved for those object programs that satisfy some constraints (such as referring to non-globally updated objects only in restricted places). This property is not as strong as unconditional
safety, but it is more widely applicable and is still much better than having no information at all.

Allowing non-generalizing type changes. Currently, well structured updates can change symbols only to more general types. This restriction is required to ensure type correctness because other type changes can break the well typing of contexts, which could only be captured by requiring an update that applies to all possible contexts and basically means to insert expressions that revert the type change. However, with the concept of conditional update safety we might be able to relax the covering criterion.

4.3 The Haskell Update Language

The update calculus captures the core functionality of an update language for functional languages, omitting the non-essential language features, such as data types and case expressions. In this section, we define the syntax and semantics of an update language HULA, which is functionally equivalent to the update calculus, but has built-in constructs to deal with language features of Haskell. Recall the size update from Section 1.3:

\[
\text{con } \text{Node} : \{ \text{Int} \} \ t \text{ where} \\
\begin{align*}
\text{(case } \text{Node } \{ s \} &\to \text{Node } \{ \text{succ } s \} \\
| \text{Leaf} &\to \text{Node } \{ 1 \} | \text{Node } \{ 1 \}
\end{align*}
\]

The \text{con} update operation adds the type \text{Int} as a new first parameter to the definition of the \text{Node} constructor. The notation \( a \{ r \} b \) is an abbreviation for the rewrite rule \( a \ b \sim a \ r \ b \) to avoid unnecessary repetition of common subexpressions.
The keyword \texttt{where} introduces the updates that apply to the scope of the \texttt{Node} constructor. A \texttt{case} update specifies how to change all pattern matching rules that use the \texttt{Node} constructor.

In the following, we will define the syntax of HULA precisely and define the semantics by giving a translation into the update calculus.

\subsection{Syntax}

The update language HULA builds on the core calculus. In particular, rules and alternation are reused. For syntactic convenience we introduce specialized notation for scope updates and use an economic notation for rules. The general syntactic schema of all scope update constructs is as follows.

\begin{verbatim}
cat bind : def where use
\end{verbatim}

\texttt{cat} marks the syntactic construct to be updated (for example, \texttt{con} for a constructor or \texttt{fun} for a function definition), \texttt{bind} denotes the symbol whose definition (\texttt{def}) and \texttt{use} is being updated and a possible renaming. In general, symbols can introduce bindings for local variables. Therefore, the \texttt{bind} part also allows the update of these bindings by renaming symbols or constructors or introducing new or deleting existing symbols. The "::" separates the updates for the bindings from the update of the definition for the updated symbol. Finally, the keyword \texttt{where} introduces the update of uses for the updated symbol. This update can be empty.
We adopt the syntactic convention that the use part extends as far as possible. The binding update is given by a rule, whereas the definition and use part of an update can be given by an arbitrary update expression. In practice, bind will be just a name (seldom a renaming) and sometimes followed by an insertion of locally defined variables. Moreover, in most cases, def and use are given by alternatives of rules or other scope updates. A special syntax is used for case updates since these require a list of binding and use updates (without definition updates).

The prevailing part of most updates consists of rules. To make update programs well readable we have therefore thought about what would be the most convenient rule notation. Traditionally, rules are written like $l \rightsquigarrow r$ (see Section 4.2.2). In many cases rules are used to provide context for adding, deleting or replacing a syntactic object, which means that quite frequently parts of $l$ are repeated in $r$.

We have therefore chosen the notation $a\{ l \rightsquigarrow r \} b$ as an abbreviation for $alb \rightsquigarrow arb$. Thus $\{ x \rightsquigarrow y \}$ means replace $x$ by $y$. This interpretation was chosen because we obtain as special cases $\{ x \rightsquigarrow \}$ meaning “delete $x$ from the context” and $\{ y \}$ meaning “insert $y$ into the context”, while $\{ x \rightsquigarrow y \}$ still reads like a rule. We say that such a rule is completely factored if $x$ and $y$ do not have a common prefix or suffix.

The syntax of the Haskell update language HULA is defined in Figure 4.10 and is built on top of the syntax for the manipulated object language Haskell. Here $hvar$ is a syntax category for Haskell variables, $mvar$ represents metavariables in HULA, and $hcons$ represents Haskell data constructors (see Appendix A).
\[
\text{upd} ::= \text{rule} \\
\hspace{1em} \text{cat bind \ upd} \ [\text{where \ upd}] \\
\hspace{2em} \text{case \ mrule} \ \{ \text{mrule} \} \\
\hspace{1em} \text{upd} \ | \ \text{upd} \\
\hspace{1em} (\text{upd}) \\
\text{mrule} ::= \text{bind} \rightarrow \text{upd} \\
\text{rule} ::= \text{chng} \ \{ \text{chng} \} \\
\text{chng} ::= \{ \text{pat} \leadsto \text{pat} \} \\
\hspace{1em} \{ \text{pat}[-\text{pat}] \} \\
\hspace{1em} \text{pat} \\
\text{bind} ::= \text{ren} \ \{ \text{ren} \} \\
\text{ren} ::= \{ \text{sym} \leadsto \text{sym} \} \\
\hspace{1em} \{ \text{sym} \} \\
\hspace{1em} \text{sym} \\
\text{pat} ::= \text{exp} \ | \ \text{type} \\
\text{sym} ::= \text{mvar} \ | \ \text{hvar} \ | \ \text{hcons} \\
\text{cat} ::= \text{data} \ | \ \text{con} \ | \ \text{fun}
\]

Factored rule  
Scope updates  
Case update  
Alternative  
Grouping  
Match-rule update  
Update rule  
Replacement  
Insertion  
No change  
Binding update  
Renaming  
New symbol  
Keep symbol  
Haskell objects  
Variables, constructors  
Scope categories

Figure 4.10: Syntax of the Haskell Update Language

4.3.2 Translation into the Update Calculus

The translation of HULA into the update calculus is defined by the function \( \mathcal{T} \), which uses four auxiliary functions \( \mathcal{T}_{D/D} \) and \( \mathcal{T}_{U/U} \) that deal with the translation of scope updates. The definitions for all these functions are shown in Figure 4.11.

The translation of rules and alternatives is rather obvious. The translation of scope updates (and case updates) is complicated by the fact that our rule notation allows the notation of a sequence of nested rules in a linear form (since a rule is given by a sequence of \textit{chng’s}). Each such sequence of elementary rules has to be translated into a nested scope update of the core calculus.

We can distinguish two kinds of elementary rules: (i) insertion rules and (ii)
\[
\begin{align*}
\mathcal{D}(p[l \rightsquigarrow r]) & = plp' \rightsquigarrow prp' \\
\mathcal{T}(\text{cat } r: u_d \text{ where } u_u) & = \mathcal{T}_D(r, \mathcal{T}(u_d), \mathcal{T}(u_u)) \\
\mathcal{T}(\text{cat } r: u_d) & = \mathcal{T}_D(r, \mathcal{T}(u_d), u) \\
\mathcal{T}(\text{case } r_1 \rightarrow u_1 | \ldots | r_n \rightarrow u_n) & = \mathcal{T}_U(r_1, \mathcal{T}(u_1)) | \ldots | \mathcal{T}_U(r_n, \mathcal{T}(u_n)) \\
\mathcal{T}(u_1 | u_2) & = \mathcal{T}(u_1) | \mathcal{T}(u_2) \\
\mathcal{T}((u)) & = \mathcal{T}(u)
\end{align*}
\]

\[\mathcal{D}(\{x \rightsquigarrow x'\}) r^* \{i_1\} r_1^* \ldots r_{n-1}^* \{i_n\} r_n^*; u_d, u_u) = \{x \rightsquigarrow x'; \mathcal{D}(r^*, \{\rightsquigarrow i_1\} \mathcal{D}(r_1^*, \ldots, \{\rightsquigarrow i_n\} \mathcal{D}(r_n^*) u_d)\}\]  
\[\mathcal{T}_U(\{x \rightsquigarrow x'\}) r^* \{i_1\} r_1^* \ldots r_{n-1}^* \{i_n\} r_n^*; u_u) = \{x \rightsquigarrow x'; u; \mathcal{U}(r^*, \{\rightsquigarrow i_1\} \mathcal{U}(r_1^*, \ldots, \{\rightsquigarrow i_n\} \mathcal{U}(r_n^*) u_d)\}\]

\[\mathcal{D}(\{x_1 \rightsquigarrow x'_1\} \ldots \{x_n \rightsquigarrow x'_n\}, u_d) = \{x_1 \rightsquigarrow x'_1; i\} \ldots \{x_n \rightsquigarrow x'_n; i\} u_d \]
\[\mathcal{U}(\{x_1 \rightsquigarrow x'_1\} \ldots \{x_n \rightsquigarrow x'_n\}, u_u) = \{x_1 \rightsquigarrow x'_1; \ldots \{x_n \rightsquigarrow x'_n; u_u\} i\ldots i\}
\]

Figure 4.11: Translation of HULA into the Core Calculus
The shown translation assumes that rules are completely factored and that the proper distinction between object and meta variables has already been made. We also assume that no-change rules have been expanded into corresponding identity replacements and that keep-symbol rules have been expanded into identity renamings. To understand how these functions work, it is best to look at some examples. The following update adds a parameter to the function definition for \( f \) and an argument to all calls to \( f \):

\[
\textbf{fun} f \{x\} : e \textbf{where} f \{3\}
\]

First, we expand the keep-symbol binding update \( f \) into the renaming \( \{f \leadsto f\} \) and the no-change rule \( e \) into the identity replacement \( \{e \leadsto e\} \). Note that the \( f \) in the use update is not expanded because it is parsed as a part of the factored rule \( f \{3\} \) that will be translated by \( \mathcal{T} \). Now the translation into the core calculus proceeds as follows:

\[
\begin{align*}
\mathcal{T}(\textbf{fun} f\{x\} : e \textbf{where} f \{3\}) &= \mathcal{T}(\textbf{fun} \{f \leadsto f\}\{x\} : \{e \leadsto e\} \textbf{where} f \{3\}) \\
&= \mathcal{T}_D(\{f \leadsto f\}\{x\}, \mathcal{T}(\{e \leadsto e\}, f \{3\}))
\end{align*}
\]

At this point we can apply the first translation rule twice to obtain

\[
\mathcal{T}(\{e \leadsto e\}) = e \leadsto e =: u_d
\]

and

\[
\mathcal{T}(f \{3\}) = f \leadsto f \{3\} =: u_u
\]
We continue by applying the definition for $T_D$ where $x$ and $x'$ both match $f$, $r^*$ is empty ($\epsilon$), $i_1$ matches $\{x\}$, and $r_n^*$ is also empty (we have $n = 1$). With $D(\epsilon, u) = u$ we obtain:

$$T_D(\{f \leadsto f\} \{x\}, u_d, u_u)$$

$$= \{f \leadsto f : D(\epsilon, \{x\} D(\epsilon, u_d))\} u_u$$

$$= \{f \leadsto f : \{x\} u_d\} u_u$$

$$= \{f \leadsto f : \{x\} e \leadsto e\} f \leadsto f \{3\}$$

$$= \{f : \{x\} e\} f \leadsto f \{3\}$$

This example demonstrates how nested binding updates used in scope updates are translated into nested definition updates. The translation is what we expect because the intention was to extend $f$’s definition by a new parameter, and this is exactly what the resulting core-calculus expression achieves. To understand the need for the functions $T_U$ and $U$, consider the translation of the following case update.

**case Node \{s\} \rightarrow Node \{1\}**

Again, we first expand the binding update so that we can apply the translation.

$$T(\text{case Node } \{s\} \rightarrow \text{Node } \{1\})$$

$$= T(\text{case } \{\text{Node } \leadsto \text{Node}\} \{s\} \rightarrow \text{Node } \{1\})$$

$$= T_U(\{\text{Node } \leadsto \text{Node}\} \{s\}, T(\text{Node } \{1\}))$$

The first rule for $T$ gives $T(\text{Node } \{1\}) = \text{Node } \leadsto \text{Node } 1$, which we abbreviate by $u$. Next we apply the definition for $T_U$. Here $x$ and $x'$ match $\text{Node}$, $r^*$ is empty, $i_1$ matches $\{s\}$, and $r_n^*$ is also empty (again, $n = 1$). With $U(\epsilon, u) = u$ we can
continue:
\[
T_{\mu}(\{\text{Node} \rightsquigarrow \text{Node}\} \{s\}, u)
\]
\[
= \{\text{Node} \rightsquigarrow \text{Node}: \iota\} U(\varepsilon, \{\rightsquigarrow s\} U(\varepsilon, u))
\]
\[
= \{\text{Node} \rightsquigarrow \text{Node}: \iota\}(\{\rightsquigarrow s\} u)
\]
\[
= \{\text{Node} \rightsquigarrow \text{Node}: \iota\}(\{\rightsquigarrow s\} \text{Node} \rightsquigarrow \text{Node} 1)
\]
\[
= \{\text{Node}\}(\{\rightsquigarrow s\} \text{Node} \rightsquigarrow \text{Node} 1)
\]

This example demonstrates that nested binding updates used in case updates are translated into nested use updates, which makes sense since the symbols introduced in match rules have no definition. Instead the use of the introduced symbols in the right-hand side of the match has to be updated. This is accomplished by the resulting core-calculus expression. As a slightly larger example we consider how the size update is translated by \(T\) into a core calculus expression. As we did in the previous examples, we first expand the binding update:

\[
\text{con} \{\text{Node} \rightsquigarrow \text{Node}\}: \{\text{Int}\} t \text{ where }
\]
\[
\begin{array}{ll}
\text{case} \{\text{Node} \rightsquigarrow \text{Node}\} \{s\} \rightarrow \text{Node} \{\text{succ } s\} & \\
| \{\text{Leaf} \rightsquigarrow \text{Leaf}\} & \rightarrow \text{Node} \{1\} | \\
\text{Node} \{1\} &
\end{array}
\]

We abbreviate the case update by \(u_c\) and the alternative update for \text{Node} extensions by \(u_a\), that is, we consider the translation of the update

\[
\text{con} \{\text{Node} \rightsquigarrow \text{Node}\}: \{\text{Int}\} t \text{ where } u_c \mid u_a
\]
We obtain:

\[
T(\text{con} \{\text{Node} \leadsto \text{Node}\} : \{\text{Int}\} \ t \ \text{where} \ u_c | u_a)
\]

\[
= T_D(\{\text{Node} \leadsto \text{Node}\}, T(\{\text{Int}\} t), T(u_c | u_a))
\]

\[
= T_D(\{\text{Node} \leadsto \text{Node}\}, t \leadsto \text{Int} t, T(u_c | u_a))
\]

The rule for case updates yields for \(T(u_c)\):

\[
T_u(\{\text{Node} \leadsto \text{Node}\} \{s\}, T(\text{Node} \{\text{succ} \ s\})) \mid T_u(\{\text{Leaf} \leadsto \text{Leaf}\}, T(\text{Node} \{1\}))
\]

Similar to the previous example, this expression can be further translated to the following expression.

\[
\{\text{Node}\}(\{\leadsto \text{s}\} \text{Node} \leadsto \text{Node} \ 1) \mid \{\text{Leaf}\} \text{Node} \leadsto \text{Node} \ 1
\]

Therefore, we obtain for \(T(u_c | u_a)\) the following core calculus expression.

\[
(\{\text{Node}\}(\{\leadsto \text{s}\} \text{Node} \leadsto \text{Node} \ 1) \mid \{\text{Leaf}\} \text{Node} \leadsto \text{Node} \ 1) \mid
\]

\[
\text{Node} \leadsto \text{Node} \ 1
\]

We can now complete the translation of the complete \text{con} update as follows.

\[
T(\text{con} \{\text{Node} \leadsto \text{Node}\} : \{\text{Int}\} \ t \ \text{where} \ u_c | u_a)
\]

\[
= T_D(\{\text{Node} \leadsto \text{Node}\}, t \leadsto \text{Int} t, T(u_c | u_a))
\]

\[
= \{\text{Node} \leadsto \text{Node}: t \leadsto \text{Int} \rightarrow t\}
\]

\[
(\{\text{Node}\}(\{\leadsto \text{s}\} \text{Node} \leadsto \text{Node} \ 1) \mid \{\text{Leaf}\} \text{Node} \leadsto \text{Node} \ 1) \mid
\]

\[
\text{Node} \leadsto \text{Node} \ 1
\]
By abbreviating the trivial rule $\text{Node} \leadsto \text{Node}$, we finally obtain:

\[
\{\text{Node} : t \leadsto \text{Int} \rightarrow t\}
\]

\[
(\{\text{Node}\}(\{\sim \text{s}\} \text{Node} \leadsto \text{Node } 1)) \mid (\{\text{Leaf}\} \text{Node} \leadsto \text{Node } 1)
\]

$\text{Node} \leadsto \text{Node } 1$

This expression represents a scope update that binds $\text{Node}$ and updates its type definition from $t$ to $\text{Node} \rightarrow t$. In the uses of $\text{Node}$ perform either of the following changes: (i) in the patterns bind $\text{Node}$ or $\text{Leaf}$. In the case of $\text{Node}$, change the pattern by inserting a pattern variable $s$. In both cases, update the bodies as well; (ii) in applications replace $\text{Node}$ with $\text{Node } 1$. This captures the semantics of the original update in the update calculus.

4.4 Summary

In this chapter, we have introduced an update calculus for lambda calculus. Its type-change system and semantics were studied. We also extended the core calculus to a more complete update language for Haskell. Update programs that satisfy certain criteria can perform updates on programs and guarantee that the resulting programs are type correct.
Chapter 5 – Generic Recursion and Traversals

Functional programmers who deal with complex data types often need to apply functions to specific nodes deeply nested inside of terms. Typically, implementations for those applications require so-called boilerplate code, which recursively visits the nodes and carries the functions to the places where they need to be applied. As explained in Section 2.2.3, the scrap-your-boilerplate approach proposed by Lämmel and Peyton Jones tries to solve this problem by defining a general traversal design pattern that performs the traversal automatically so that the programmers can focus on the code that performs the actual transformation.

In practice we often encounter applications that require variations of the recursion schema and call for more sophisticated generic traversals. Defining such traversals from scratch requires a profound understanding of the underlying mechanism and is everything but trivial.

In this chapter we analyze the problem domain of recursive traversal strategies, by integrating and extending previous approaches. We then extend the scrap-your-boilerplate approach by rich traversal strategies and by a combination of transformations and accumulations, which leads to a comprehensive recursive traversal library in a statically typed framework.

\footnote{Contents of this chapter are mainly based on [72].}
We define a two-layer library targeted at general programmers and programmers with knowledge in traversal strategies. The high-level interface defines a universal combinator that can be customized to different one-pass traversal strategies with different coverage and different traversal order. The lower-layer interface provides a set of primitives that can be used for defining more sophisticated traversal strategies such as fixpoint traversals. The interface is simple and succinct. Like the original scrap-your-boilerplate approach, it makes use of rank-2 polymorphism and functional dependencies, implemented in GHC [71].

5.1 Introduction

The scrap-your-boilerplate (SYB) approach relieves a big burden from Haskell programmers who need to traverse complex data structures frequently. They can now focus on the code that does the real job instead of the traversal itself. The boilerplate code to traverse arbitrary data structures can be automatically derived. In the following, we illustrate how to implement some traversals in our library through several examples. We begin with defining the $\text{increase}$ function using our interface:

\[
\text{increase} :: \text{Float} \to \text{Company} \to \text{Maybe Company}
\]

\[
\text{increase } k = \text{traverse } \text{Trans NoCtx Full FromBottom FromLeft}
\]

\[
\text{(always } (\text{incS } k))
\]

Compared to the original version, this $\text{increase}$ function is defined using more parameters which specify the traversal. In this particular case, the parameters
define the traversal to be a transformer that modifies nodes, independently of contextual information. It is a full traversal (all nodes in the tree will be visited), and the order of visiting the nodes is from bottom to top, from left to right. Another noticeable difference is the return type, which is a Maybe accounting for possible failures. We allow a transformation on a node to fail. A failed transformation will leave the node unchanged. Such mechanism can be used to construct contingent transformations. We will discuss more about failures in Section 5.2. The “interesting case” that deals with Salary data is still the incS function, which we can reuse without changes. However, instead of extending its type to make it a generic function, we take a slightly different approach. We define a few combinators to combine specific functions and pass a list of them to the traversal combinator. In this case, the combinator always takes the specific function incS k. This specific function is unconditionally applied and works on any term of type Salary.

In applications like this one, not all the parameters are interesting. The users usually do not care, or even do not know, about the context and the left-to-right traversal direction. All they need is a transformation. We have identified default values for the different dimensions along which a traversal can be customized and have introduced functions for all possible combinations of parameters following a strict naming scheme that will be explained in detail in Section 5.3.3. Employing the traversal that represents the shown traversal parameters, the presented example can be defined much more succinctly as follows.

```haskell
increase :: Float -> Company -> Company
increase k = transformB (always (incS k))
```
The B indicates “bottom-up”, which was chosen in the original SYB approach. The top-down version \texttt{transform} works just as well.

In the following we continue to use the expanded versions of the traversals to make the parameters and options explicit.

Our next example is an accumulation instead of a transformation. An accumulation can serve as a query defined in [42] but is more general. The following function computes the salary bill for a company by traversing the company data structure and accumulates all salaries.

\begin{verbatim}
 bill :: Company -> Maybe Float
 bill = traverse Accum NoCtx Full FromTop FromLeft
     (always col) 0
 where col a (Salary s) = a + s
\end{verbatim}

The local function \texttt{col} takes an accumulator (a), which is the sum collected so far, and a \texttt{Salary} and adds the salary to the sum. In the end of the traversal, the accumulator is the sum of all salaries.

5.1.1 Possible Extensions

The original SYB approach provides a type-based approach to the traversal of complex data. However, it only provides a framework rather than a complete problem-domain-oriented solution. One obvious feature that is missing is strategies (see Section 2.1). We will now explore some possible applications and potential solutions based on the SYB framework.
Accumulation and Transformation

Suppose we not only want to increase everyone’s salary, but also need the total amount being increased. We keep traversing the company data structure, increasing everyone’s salary and modifying the total amount at the same time. Again, we need to resort to a combinator that is similar to \texttt{everywhere}, but can maintain a state for the total. Such a function can be defined as follows. Upon a successful return, the result consists a total amount and a modified company value.

\begin{verbatim}
incBill :: Float \rightarrow Company \rightarrow Might (Float,Company)
incBill k = traverse AccTrans NoCtx Full FromTop FromLeft
            (always (colS k)) 0

colS :: Float \rightarrow Float \rightarrow Salary \rightarrow (Float, Salary)
colS k a (S s) = (a+k*s, S*(1+k))
\end{verbatim}

A similar application in program transformation occurs when we need to generate new variables that do not conflict with any existing variables in the original program. We need to keep track of variables that have been already generated to keep the variable names unique. In general, a transformation might need to access information accumulated from the nodes visited so far in the traversal. Accumulations find a broad range of applications in language processing area. Examples include counting certain nodes, collecting variables, collecting other constructs, etc. We will describe one such application is some detail in Section 5.5.
Partial Traversals

In some applications, not all the nodes in a term have to be visited. Consider a local transformation where we only want to apply the transformation to a certain part of the term. One such application is increasing salaries in a certain department rather than the whole company. This problem is addressed in Section 6.2 in [42] with a function \texttt{incrOne} defined using the \texttt{gmapT} function, which is rather complicated to come up with for ordinary programmers. Since a similar pattern can be observed in many applications, it would be beneficial to provide a general solution once and for all. An elegant way to realize such a transformation is to employ a so-called \textit{stop-traversal} [41]. A stop-traversal tries to apply a visit to all nodes. If the visit succeeds on a node, the traversal continues without descending into that node. In this example, another traversal is passed as a visit argument to the outer traversal. The nested traversal is the \texttt{increase} function. It is applied to nodes that are departments with a matching name. The \texttt{mwhenever} function is used to construct a conditional visit and will be explained in Section 5.3.1.

\[
\text{incOne :: Float -> Name -> Company -> Company}
\]
\[
\text{incOne k d = traverse Trans NoCtx Stop FromTop FromLeft}
\]
\[
\quad \text{(increase k 'mwhenever' isDpt d)}
\]

\[
\text{isDpt :: Name -> Dept -> Bool}
\]
\[
\text{isDpt d (D n _ _) = n==d}
\]

Another example of a stop-traversal was given in Section 3.7.1 where we introduced the context update language for adding actions to monadified programs.
We can also consider \textit{once-traversals} [89] where we only want to apply a transformation a single time. These are also a special case of partial traversals. For instance, we can increase the first salary we encounter when traversing the company data.

\[
\text{incFst} :: \text{Float} \rightarrow \text{Company} \rightarrow \text{Maybe Company} \\
\text{incFst} \ k = \text{traverse \ Trans NoCtx Once FromTop FromLeft} \\
\quad \text{(always (incS \ k))}
\]

\textbf{Traversal with Contexts}

Transformations that depend on non-local data are also difficult to express in the original SYB approach. Let us consider a more complicated application of increasing salaries. Say we want to adjust the increase rate according to the department. A context, which is the increase rate, is carried through the traversal. It is initialized to a default rate and is updated whenever the traversal is descended into a node so that all salaries inside that node will get increased by the new rate (unless the rate gets changed again before that salary is reached).

\[
\text{incDpt} :: \text{Float} \rightarrow \text{Company} \rightarrow \text{Maybe Company} \\
\text{incDpt} = \text{traverse \ Trans Ctx Full FromTop FromLeft} \\
\quad \text{(mk (\ \text{c \ d} \rightarrow \text{lookupRate \ d})} \\
\quad \text{(always \ incS)}
\]

Compared to the previous examples, this contextual traversal takes as an additional argument a context updater (\text{\textbackslash c \ d} \rightarrow \text{lookupRate \ d}), where the function \text{lookupRate} determines the increase rate for the department. Similar to visits, a context updater will be applied to terms of any type. Therefore, it needs to be
generic as well. The \texttt{mk} function is used to wrap a specific context updater and make it generic. It will be explained in Section 5.3.1 along with combinators for visits.

The careful reader might have noticed that the visit in this example, expressed by \texttt{always incS}, has a different type than before. Here, \texttt{incS} is used as a \textit{contextual visit}, which takes an extra parameter, the context. The types of all 6 visits are listed in Figure 5.3. The \texttt{always} function is overloaded in order to provide a uniform interface to the programmer.

We can also consider an application in language processing. Suppose we want to implement a beta reduction for lambda calculus. A beta redex is a lambda abstraction applied to an argument. The body of the lambda abstraction is traversed so that all the free occurrences of the bound variable are replaced by the argument. However, we have to be careful not to replace locally bound variables with the same name. When we descend into the term, we need to keep track of a collection of bound variables. The transformation needs to check against these variables. A beta reduction carries a list of bound variables as the context and it gets extended at lambda abstractions.

From these two problems, we can generalize a pattern of contextual traversal. An initial context is passed to the traversal and it gets updated by an update function when descending into subterms.
5.1.2 Contribution and Organization

The shown applications can be generally implemented by employing the generic fold operator `gfoldl` defined in [42]. However, this is not at all a trivial task. Our goal is to generalize the design pattern and extend it to support these applications. The approach we take is to combine elements from SYB, Strafusni, and Stratego and to create a fully typed generic traversal library consisting of categorized recursive traversal strategies and implement the library with strategy combinators.

L{"{a}}mmel and Visser present a combinator library for generic traversals and a set of traversal schemes as part of Strafusni [45, 46, 41]. However, it relies on the DrIFT tool to generate the instances of type class `Term`. Moreover, it is not a fully statically typed approach. One uses an abstract datatype for generic functions to separate typed and untyped code. In [41], L{"{a}}mmel presents a hierarchy of traversals and defines a `traverse` function that can be highly parameterized. We make use of this “mother of traversals” to derive all traversals.

Stratego [89, 88, 11] defines an abundant set of traversal strategies. Our main motivation comes from the need to apply these traversal strategies in our monadification tool (Chapter 3). However, we want to use them in the context of Haskell. We also want static type safety, which is not found in Strafusni and Stratego. We are also motivated by the need for a concise program interface without using complex data types, such as monads. Therefore, we propose the approach of defining a generic traversal library with a simple and general programming interface and a rich set of traversal strategies. In this experimental implementation, we focus on
the concepts rather than having a complete set of traversal strategies. However, with the genericity of the approach, new traversal strategies can be defined easily. The relationship between the library and Stratego, Stranfinski, and SYB is sketched in Figure 5.1. The source code of the library can be obtained online [71]. Features of different approaches are compared in Figure 5.2.

Figure 5.1: Relation between Reclib and Related Approaches

<table>
<thead>
<tr>
<th></th>
<th>Reclib</th>
<th>Stratego</th>
<th>Stranfinski</th>
<th>SYB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Typed</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Integrated in Haskell</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Strategies</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Concrete Syntax</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 5.2: Feature comparison of Reclib and Related Approaches

In the rest of this chapter, we categorize in Section 5.2 the problem domain of
traversals by extracting five parameters that are, to a large extent, orthogonal to each other. In Section 5.3, we describe a high-level programming interface. This interface provides a means to parameterize traversal strategies that cover all possible combinations of those five parameters. In the core of the interface, we define one generic traversal strategy that is the “mother” of all one-pass traversal strategies we explored. An intermediate layer of programming interface is also defined for users who require more than one-layer traversals. This interface is concise and clean. Two fixpoint strategies, innermost and outermost, are studied and implemented using the interface as examples for extendibility. Sections 5.3.4, 5.3.5, and 5.3.6 elaborate implementation details and can be skipped without jeopardizing understanding of the programming interface. In Section 5.4 we illustrate more examples that make use of the library in greater detail. In Section 5.5, we present a practical application in the implementation of the monadification algorithm. We discuss and compare related work in Section 5.6. In Section 5.7 we present conclusions and directions for future work.

5.2 Design Space

In a typical traversal, all or part of the nodes are visited in a particular order. We use the term visit to refer to one access to a particular node. During a visit, information is retrieved from the node, and/or the node is modified. The information and the modification might depend on the information retrieved from the nodes already visited in the traversal and/or the path from the node to the root.
A visit that retrieves information does so by taking already accumulated information and returning the new accumulator, which is threaded through all the visits in the traversal. We distinguish three kinds of visits. A transformer modifies a node without retrieving information, an accumulator retrieves information without modifying the node, and an accumulating transformer does both simultaneously. We borrow this categorization from [81]. Every visit may either succeed or fail. Therefore, the result of a visit is wrapped in a Maybe data type.

In the example of increasing salaries for people in a certain department, the traversal combinator needs to carry some information related to the path from the current node to the root. We call this a context. The combinator updates the context by applying a user-provided context function. It then passes the updated context to all the children of the node.

Therefore, there are all together six kinds of visits whose types are listed in Figure 5.3. As a convention, we use c to denote a context type, a for an accumulator type and t for a term type.

<table>
<thead>
<tr>
<th></th>
<th>Contextual</th>
<th>Non-contextual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformation</td>
<td>c -&gt; t -&gt; Maybe t</td>
<td>t -&gt; Maybe t</td>
</tr>
<tr>
<td>Accumulation</td>
<td>c -&gt; a -&gt; t -&gt; Maybe a</td>
<td>a -&gt; t -&gt; Maybe a</td>
</tr>
<tr>
<td>Acc. Transformation</td>
<td>c -&gt; a -&gt; t -&gt; Maybe (a,t)</td>
<td>a -&gt; t -&gt; Maybe (a,t)</td>
</tr>
</tbody>
</table>

Figure 5.3: Types of 6 Kinds of Visits

A traversal can be categorized regarding the number of times every node is visited. A one-pass traversal traverses a tree in one pass and visits each node at most once. A typical example is a depth-first search. A fixpoint traversal
[11] applies a visit to a tree using a certain strategy repeatedly until it is not applicable anymore. Innermost and outermost traversals fall into this category. We implement both kinds of traversals in our library, but we focus more on one-pass traversals.

In a one-pass traversal, it is not always desirable to visit all the nodes in a term. A typical scenario of a partial traversal is when we abort the traversal after a single successful visit. This kind of coverage is called once, as opposed to full, where all nodes are visited sequentially unless stopped by a failed visit. Another common situation is a so-called stop-traversal that tries to apply a visit to the root node of a tree. If it fails, it then tries to recursively apply it to all children. Otherwise, it stops. Effectively, a stop-traversal visits nodes on a frontier of a tree. A typical application for such traversals is optimization. We can significantly decrease the number of nodes visited by focusing on interesting nodes. Symmetrically, a spine-traversal visits a chain of nodes from the root to a leaf. A spine-traversal fails if no spine exists such that the visit succeeds on every node on the spine. Figure 5.4 illustrates these four different kinds of coverage. In the figure, the dashed line connects all nodes that are successfully visited, but does not include those tried but failed.

Furthermore, there are two kinds of directions that affect the order in which the nodes are visited: the vertical direction and the horizontal direction. A vertical direction can be either top-down or bottom-up. A horizontal direction is either left-to-right or right-to-left. In a top-down traversal, a root is visited before its descendants. In a bottom-up traversal, the children are visited before their parent.
Top-down and bottom-up traversals are also often referred to as preorder and postorder traversals, respectively. The order in which the siblings of a common parent are visited is determined by the horizontal direction, which can be either from the left or from the right. The directions usually matter for the accumulating traversals or once-traversals. Figures 5.5 and 5.6 illustrate vertical and horizontal directions, respectively.

In summary, five parameters determine a (one-pass) traversal: kind of the visit, context, coverage, and two directions. These parameters are mostly orthogonal to each other. We can obtain a rich set of traversals by customizing all these parameters.
5.3 Programming Interface

Our goal is to provide an easy-to-use and effective programming interface to users who wish to program generic traversals. In this section, we will describe the generic traversal combinators and some necessary helper functions. The interface is divided into two layers. A higher-level interface is provided to users who do not have profound knowledge in generic programming and term traversals. They
can easily program their own traversals using provided combinators and compose necessary arguments using the auxiliary functions. The library is flexible and extensible in the sense that an intermediate layer is exposed to users who wish to write traversal strategies that are not found in the library to meet their own needs. As an example, the implementation of fixpoint traversal combinators \texttt{innermost} and \texttt{outermost} will be presented. Other example traversals include \texttt{downup} and \texttt{updown} strategies [87, 11]. They, too, can be implemented with the intermediate layer.

5.3.1 Building Generic Functions

The visits as well as context updaters are generic in the sense that they are applicable to values of any type. The \texttt{mkT} function described in [42] creates a generic function of type \texttt{a -> a} out of a specific function, and \texttt{extT} extends a generic function with a specific function. However, this approach of using two combinators does not work for our purpose for two reasons. First, our visits return \texttt{Maybe} values. Second, we cannot expect one set of combinators to work for all kinds of visits, because they generally have different types, as we have seen in Figure 5.3. Having a separate set of such \texttt{mkT} and \texttt{extT} combinators for each kind of visit is very cumbersome. Therefore we decided to provide a universal mechanism for composing visits and hide the differences and details. The decision resulted in the design that the generic traversal function \texttt{traverse} (which will be explained in Section 5.3.2) takes a list of specific visits (and possibly context updaters) rather
than a generic one. This also relieves the users of the burden of applying the extending combinator. We need to encapsulate specific context updaters and visits with rank-2 polymorphic data types so that they can be put into lists. An example of such a data type for a contextual accumulating transformer is the data type \texttt{GenCAT}, defined as follows.

\begin{verbatim}
data GenCAT c a = forall t. Typeable t =>
   GenCAT (c -> a -> t -> Maybe (a,t))
\end{verbatim}

Specific visit functions that work on different types of nodes (but on the same context and accumulator types) can be wrapped with the data constructor \texttt{GenCAT} and put in a list which is passed to the \texttt{traverse} function. For each kind of visit listed in Figure 5.7, a separate data type is required. The context updater works in a similar way.

\begin{verbatim}
data GenU c a = forall t. Typeable t =>
   GenU (c -> t -> c)
\end{verbatim}

To hide the differences between these data constructors, one overloaded function \texttt{mke} is provided. It serves a similar purpose as \texttt{mkT} function except that it works for all visits and context updaters. A specific visit or context updater is passed to the \texttt{mke} function, and a generic one is constructed. For contextual accumulating transformers, its type is the following.

\begin{verbatim}
mke :: (c -> a -> t -> Maybe (a,t)) -> GenCAT c a
\end{verbatim}

Since in many cases a generic function is built from just one specific function, a function \texttt{mk} is defined to further hide the list structure.

\begin{verbatim}
mk x = [mke x]
\end{verbatim}
In fact, even if two or more specific functions are used to compose a generic one, the \texttt{mk} function can be used, and the results can be concatenated using ++ operator. Therefore clients usually do not need the \texttt{mke} function.

In addition to the \texttt{mk} combinator, we provide two sets of combinators for composing visits. To selectively apply one of two visits depending on the node, a combinator \texttt{mcond} is provided, which takes one predicate and two visits. It implements a conditional. In cases where the else part is missing (indicating a failed visit), the combinator \texttt{mwhenever} can be used. To apply a visit unconditionally, the combinator \texttt{malways} is used. A visit returns a \texttt{Maybe} value to indicate a success or failure. For visits that do not fail, it is an extra burden to handle the \texttt{Maybe} data type. We define three symmetric combinators \texttt{cond}, \texttt{whenever}, and \texttt{always} that take visits that do not return \texttt{Maybe} values. In the salary-increasing example, the visit can be composed using \texttt{always}: \texttt{always (incS k)}. A visit that increases every salary inside a node if the node is a certain department can be composed with the \texttt{increase} function and a predicate, that takes the department name as a parameter \texttt{d}.

\begin{verbatim}
  increase k `mwhenever` \(\text{D n }\_\_) \rightarrow \text{n==d}
\end{verbatim}

Such a visit can be used to compose a stop-traversal. It is recursively tried on every node in a term but has no effect on the node unless it is a department whose name matches \texttt{d}, in which case the \texttt{increase} function is applied recursively to the subtrees of that node.
5.3.2 Traversal Engine

The main component of the interface is a heavily overloaded function `traverse` that can be customized by all the five parameters we mentioned. And since it is an overloaded polymorphic function, its type varies. It is defined as a member function of type class `Traversal`:

```
class Traversal { ...
  traverse :: u -> v -> Coverage -> VD -> HD -> x ...
}
```

What is common to all instances are the first five parameters that identify a traversal. Type variable `u` represents the kind of visit, and `v` is either `Ctx` or `NoCtx` representing the presence or absence of the context. As explained in Section 5.2, type variables `a`, `c`, and `t` represent the types of the accumulator, context and term, respectively. The functional dependency helps the type system determine the instance of `traverse` when it is applied but the result type is not explicitly specified. The reader might wonder whether `c`, `a`, and `t` are really needed since they do not appear in the type of `traverse`. The answer is yes, they are indeed required, because `x`, the type of the traversal, depends on them.

Presented below are the data type definitions for the parameters of `traverse`.

```
data Trans = Trans
data Accum = Accum
data AccTrans = AccTrans

data Ctx = Ctx
data NoCtx = NoCtx

data Coverage = Full | Spine | Once | Stop
```
<table>
<thead>
<tr>
<th>Trans</th>
<th>v = Ctx</th>
<th>v = NoCtx</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccTrans</td>
<td>[GenU c a] -&gt; [GenCAT c a] -&gt; c -&gt; a -&gt; t -&gt; Maybe (a,t)</td>
<td>[GenAT c a] -&gt; a -&gt; t -&gt; Maybe (a,t)</td>
</tr>
</tbody>
</table>

Figure 5.7: Types of Traversals

```
data VD = FromTop | FromBottom
data HD = FromLeft | FromRight
```

Kind of visit and context presence are defined using one data type for each kind as opposed to the other three parameters in which each kind is represented by just one data constructor. This is simply a means for the compiler to choose the correct instance of traverse function. The rest of the parameters and the result type are all combined in x, which is the traversal type, determined by u, v, and the types of the accumulator, the context, and the term. For example, an instance of contextual accumulating transformers takes the following form.

```
instance Data t => Traversal AccTrans Ctx c a t
  ([GenU c a] -> [GenCAT c a]
   -> c -> a -> t -> Maybe (a,t))
```

where ...

A complete list of correspondence between x, u, and v is listed in Figure 5.7.

A list of context updaters ([GenU c a]) has to be provided for contextual traversals. A list of visit functions is required for all traversals. The type of the visit depends on the kind of the visit and presence of context. The most general
visit, a contextual accumulating transformer, has the following type, defined as a
type synonym (GCAT is not an abstract data type and should not be confused with
GenCAT previously mentioned).

\[
\text{type GCAT } c \ a = \text{forall } t \cdot \text{Data } t \Rightarrow c \rightarrow a \rightarrow t \rightarrow \text{Maybe } (a, t)
\]

where \(c\) is the type of the context, \(a\) is the type of the accumulator, and \(t\) is a uni-
versally quantified type variable, which means that a visit is a rank-2 polymorphic
function that should be applicable to values of any type. We provide auxiliary
combinators for composing such generic functions out of specific functions as we
have seen in Section 5.3.1. The result type of this visit, \(\text{Maybe } (a,t)\), captures
the nature of an accumulating transformer. Upon success, a new accumulator and
a modified node are returned. The visit returns \text{Nothing} to signal a failure. The
action to be taken upon a failed visit depends on the traversal: Full-traversal or
spine-traversal fail immediately, whereas once- and stop-traversals continue. How-
ever, while a once-traversal continues with the subterms only until a successful
visit, a stop-traversal continues even after a successful visit, it only stops descend-
ing into subterms. The types of other kinds of traversal can be deduced naturally.
For non-contextual visits, the \(c\) is omitted, transformers will not have the \(a\), and
an accumulator returns a value of type \(\text{Maybe } a\) instead.

5.3.3 Syntactic Sugar

The \text{traverse} function is the ultimate interface for the programmers. However,
programmers are not always interested in all the traversal parameters. In the
example of increasing everyone’s salary, the traversal order has no effect on the result. For cases like this, we define instances of the \texttt{traverse} function using default values. We introduce 96 functions, each of which is a partial application of \texttt{traverse} function to a combination of the traversal parameters. The functions follow a naming convention. The name consists of a \texttt{verb} and an optional prefix and three optional suffixes. The verb is either \texttt{transform}, \texttt{accumulate}, or \texttt{acctrans}. The prefix specifies the coverage, which defaults to full, when omitted. The first suffix is the presence of the context. A letter \texttt{C} follows the verb to obtain a contextual traversal, an absence indicates a non-contextual traversal. What follows is the vertical direction. A letter \texttt{B} indicates a bottom-up traversal. When it is omitted, a top-down traversal is obtained. Finally, a \texttt{'} symbol can be appended to the end to obtain a right-to-left traversal instead of the default left-to-right version.

According to these naming rules, a contextual, bottom-up, right-to-left accumulation corresponds to function \texttt{accumulateCB} of the following type.

\begin{verbatim}
Data t => [GenU c a] -> [GenCA c a] ->
c -> a -> t -> Maybe a
\end{verbatim}

With the conventions, the functions defined in Section 5.1.1 can be given in a more succinct way as follows.

\begin{verbatim}
increase k = transformB (always (incS k))
bill = accumulate (always col) 0
incBill k = acctrans (always (colS k)) 0
incOne k d = stopTransform (increase k ‘mwhenever‘ isDpt d)
incPst k = onceTransform (always (incS k))
incDpt k d = transformC (mk (\c d -> lookupRate d))
                 (always incS))
\end{verbatim}
5.3.4 Crafting Traversals

The combinator\textquotesingle s we have presented above provide enough flexibility for defining commonly used one-pass traversals. But more complicated traversals, such as a fixpoint traversal \textit{innermost} that might visit some nodes more than once, cannot be expressed. To help users who have knowledge in traversal strategies and need to define special traversals, our library also exposes an intermediate layer. In the rest of this section we explain how the recursive traversal strategy\textquotesingle s are defined using the intermediate layer.

A basic component of every traversal strategy is a one-layer strategy. Such a strategy does not apply a visit recursively. Instead, it applies another strategy to the immediate subterms. We define four such combinator\textquotesingle s. Strategy \texttt{all}_\texttt{r} applies a strategy to all the immediate subterms of a node in a left-to-right order. Strategy \texttt{one}_\texttt{r} tries a strategies on all subterms of a term and stops after a successful application. The other two, \texttt{all}_\texttt{r} and \texttt{one}_\texttt{r}, are their right-to-left counterparts. Recursive traversals can then be built on top of these one-layer strategies. For instance, a top-down full-traversal can be conceptually defined as follows.\footnote{The definition is taken from that of the \texttt{topdown} strategy in \cite{90}, but renamed here for the naming consistence.}

\[
\text{fulltd}(v) = v; \text{all(fulltd}(v))
\]

where \( v \) is the visit to be applied. The sequential composition operator \( ; \) \cite{88} takes two strategies and applies them sequentially. Failure of either one will cause
the failure of the whole strategy. Instantiating \texttt{all} \cite{88} in the above definition with \texttt{all\_l} and \texttt{all\_r} will result in left-to-right and right-to-left versions of top-down full-traversals. A one-layer strategy does not need to take into consideration the context because all immediate subterms will have the same context. It is the job of the recursive traversal strategies to update the context and pass it to one-layer traversals. We define a type synonym for a one-layer traversal without a context.

\begin{verbatim}
type GAT a = forall t. Data t => a -> t -> Maybe (a,t)
\end{verbatim}

This type represents a generic function that takes an accumulator and a term of any type and returns a new accumulator and term upon success. All the one-layer combinators take a strategy of this type and return a strategy of the same type. They are defined with the help of the \texttt{gfoldl} function \cite{27, 42} which works more or less in the same way as list folding.

\begin{equation*}
gfoldl \circ f \ (C \ t_1 \ t_2 \ldots \ t_n) = f(C) \circ t_1 \circ t_2 \cdots \circ t_n
\end{equation*}

The unary operator \( f \) is applied to the constructor \( C \), then the result is passed to the binary operator \( \circ \) with the first subterm, obtaining a result which is again passed to the binary operator along with the second subterm, and so on. Thus \texttt{all\_l} is defined as follows, and will be explained below.

\begin{verbatim}
newtype Xall\_l a t = Xall\_l \{unXall\_l :: Maybe (a,t)\}

all\_l :: GAT a -> GAT a
all\_l s a t = unXall\_l (gfoldl k z t)
  where z d = Xall\_l (return (a, d))
    k (Xall\_l x) t = Xall\_l (do (a,d) <- x
                                (a’,t’) <- s a t
                                return (a’, d t’))
\end{verbatim}
If this looks awfully complicated, it is the auxiliary data type \texttt{all\_l} that is to be blamed. Its sole purpose is to make the type system happy. Otherwise, the definition of \texttt{all\_l} could be simplified as follows.

\begin{verbatim}
all_l s a0 = gfoldl k z
  where z d = return (a0,d)
    k x t = do (a,d) <- x
            (a',t') <- s a t
           return (a',d t')
\end{verbatim}

Passed along the fold are an accumulator and a partially applied term, encapsulated in \texttt{Maybe}. A \texttt{Nothing} value indicates a failure in the previous computation and thus should be propagated (this is hidden by using the monad instance of \texttt{Maybe}). Otherwise, the value is passed to the binary operator \texttt{k} whose second parameter is the current subterm. The operator \texttt{k} applies the visit to the current subterm resulting in a new accumulator and a new term. The partially applied constructor is applied to the changed term and is returned along with the new accumulator. The initial value for the fold is obtained from the unary operator \texttt{z} which, when applied to the data constructor, returns the initial accumulator and the constructor.

Having understood the logic, we can then examine the type of \texttt{gfoldl}, which is the reason why the above simplified code does not type-check.

\begin{verbatim}
gfoldl :: (forall a t. Data t => c (t -> a) -> t -> c a)
  -> (forall g. g -> c g)
  -> b -> c b
\end{verbatim}

Understanding the above type signature is difficult. The first line is the type for the binary operator; the second line is the unary operator. It is not surprising to see that both operators have polymorphic types because they are applied to all
direct subterms that do not necessarily have the same type. The term to fold is of type \( b \) and the result is of type \( c \ b \). The same type constructor is used for the unary and binary operators. In the case of \( \text{all}_1 \), the pair whose type is \text{Maybe} \((a, b)\) does not match the form \( c \ b \). This is why the auxiliary data type is needed, that is, the type constructor \text{Xall}_1 \ a \ plays the role of \( c \) here.

Defining a right-to-left traversal is more tricky, because no \text{gfoldr} \ is available. We need to do a left fold and incrementally generate a function along the fold.\(^3\) The function, when applied to an accumulator, applies the traversal to the current term and the accumulator, and then passes the result to the function generated from the previous term.

\[
\text{newtype Xall}_r \ a \ t = \text{Xall}_r \ \{\text{unXall}_r :: a \to \text{Maybe} \ (a, t)\}\]

\[
\text{all}_r :: \text{GAT} a \to \text{GAT} a
\]

\[
\text{all}_r \ s \ a \ t = \text{unXall}_r \ (\text{gfoldl} \ k \ z \ t) \ a
\]

where \( z \ d = \text{Xall}_r \ (\lambda a \to \text{return} \ (a, d)) \)

\[
k \ (\text{Xall}_r \ g) \ t =
\]

\[
\text{Xall}_r \ (\lambda a \to \text{do} \ (a', t') \leftarrow s \ a \ t
\]

\[
(a'', d) \leftarrow g \ a'
\]

\[
\text{return} \ (a'', d \ t'))
\]

The other two one-layer strategies \text{one}_1 \ and \text{one}_r \ are slightly more involved, but can be defined similarly.

Now, to define the recursive traversal \text{fulltd}, we still need a sequential composition combinator, which can be defined as follows.

\[
\text{compose} :: \text{GAT} a \to \text{GAT} a \to \text{GAT} a
\]

\[
\text{compose} \ s1 \ s2 \ a \ t = \text{do} \ (a', t') \leftarrow s1 \ a \ t
\]

\[
s2 \ a' \ t'
\]

---

\(^3\)This approach is called second-order fold [75, 94].
With all₁ and compose, we are ready to define the top-down full-traversal strategy.

5.3.5 The Mother of All Traversals

Before we present the definition of the top-down full-traversal, let us first examine all the coverages we mentioned, namely, full, spine, stop, and once. If the horizontal direction is ignored, all the four variations can be summarized as follows.⁴

\[
\begin{align*}
\text{fulltd}(v) &= v; \text{all} (\text{fulltd}(v)) \\
\text{spinetd}(v) &= v; \text{one} (\text{spinetd}(v)) \\
\text{stoptd}(v) &= v + \text{all} (\text{stoptd}(v)) \\
\text{oncetd}(v) &= v + \text{one} (\text{oncetd}(v))
\end{align*}
\]

The choice combinator + takes two strategies, and tries the first one. Only if it fails, the second one is applied. Since the visits return Maybe values, the choice combinator can be defined in Haskell as follows.

\[
\text{choice} :: \text{GAT}\ a \to \text{GAT}\ a \to \text{GAT}\ a \\
\text{choice}\ s1\ s2\ a\ t = s1\ a\ t\ \text{mplus}\ s2\ a\ t
\]

We can observe a strong similarity among all these traversal strategies: they all have the same form, the only differences being all/one and the ;/+ combinators.

⁴stoptd is also called alltd in [86].
Examining the bottom-up versions reveals the same similarity:

\[
\begin{align*}
\text{fullbu}(v) &= \text{all}(\text{fullbu}(v)); v \\
\text{spinebu}(v) &= \text{one}(\text{spinebu}(v)); v \\
\text{stopbu}(v) &= \text{all}(\text{stopbu}(v)) + v \\
\text{oncebu}(v) &= \text{one}(\text{oncebu}(v)) + v
\end{align*}
\]

In fact, we can observe that these bottom-up strategies are just the flip side of the top-down strategies. Take this literally, replacing ; and + with their flipped versions in the definitions of the top-down strategies, we obtain exactly the bottom-up counterparts. Thus, we can generalize the pattern and define a “mother of all traversals” [41] that can generate all these traversal strategies given appropriate parameters.

\[
\text{mother}(s) = s \cdot f(\text{mother}(s))
\]

The combinator \( f \) is a one-layer strategy, which can be either \text{one}_l, \text{one}_r, \text{all}_l, \) or \text{all}_r. The combinator \( \cdot \) is taken from \text{compose}, \text{choice}, \text{compose}' and \text{choice}' where \text{compose}' and \text{choice}' are the flipped versions, with the two parameters swapped.\(^5\)

\[
\begin{align*}
\text{compose}' s1 s2 &= \text{compose} s2 s1 \\
\text{choice}' s1 s2 &= \text{choice} s2 s1
\end{align*}
\]

Each combination of parameters uniquely determines the behavior of the traversal. Figure 5.8 lists all possible combinations. With the mother of all traversals, travers-

\(^5\)We would have defined them using the \text{flip} function, but the type system prevented us from doing so, due to the rank-2 polymorphism.
sals of different coverage, vertical, and horizontal directions are just a matter of partial applications of fixed parameters. The actual definition of \texttt{mother} in Haskell takes into consideration the context.

\begin{verbatim}
mother :: (GAT a -> GAT a -> GAT a) ->
    (GAT a -> GAT a) ->
    GCU c ->
    GCAT c a ->
    GCAT c a

mother g f u s c a t = (s c
    'g' f (mother g f u s (u c t))
) a t
\end{verbatim}

The context \texttt{c} is updated by the context updater \texttt{u} and passed to one-layer strategy combinator \texttt{f}. The \texttt{mother} function is used to define instances of \texttt{traverse} by fixing the parameters \texttt{g} and \texttt{f} as shown in the next subsection.

5.3.6 Failure and Continuation

One issue worth mentioning is that a visit either fails or succeeds on a node. Continuation depends on the recursive traversal strategy. In the case of generic
traversals, since the generic visits are converted from specific visits, there is in fact a third case. That is, none of the visits is applicable to the node. Handling such cases requires discretion from the designers. In Redex it is handled differently depending on the coverage of the traversal. In a full or spine-traversal, such cases are regarded as successful visits that do not change the term nor the accumulator. The rationale behind this is that users write specific visits and apply them everywhere they are applicable. If they want to stop a traversal, they should explicitly signal a failure. Under this assumption, users are able to perform traversals even if they do not have complete knowledge of the whole tree. Therefore, in a full or spine-traversal, the traversal never fails unless a visit fails.

However, in a stop or once-traversal, a non-applicable visit is regarded as a failure. This is because in these two kinds of traversals, the traversal continues after failed visits. In a once or stop-traversal, the traversal succeeds only when there is a successful visit. Similarly, if a user does not have complete knowledge of the whole term, she is still able to handle those she is interested in and ignore others.

As we have seen in Section 5.3.5, we need to pass generic visit functions to the core combinator. However, the traverse function takes a list of specific functions. The gap is filled by type extension. Similar to the mkT and mkQ functions from [42], a generic function is used as the unit value for a fold operation over the list. The binary operator for the fold is the type extension function ext0 defined in the Data.Generics.Aliases module of the Haskell Hierarchical Libraries [27]. The unit value is chosen based on the policy we just described. For full or spine-
traversals, it is a function that always succeeds.

    vsucc :: GAT a
    vsucc a t = return (a, t)

For stop- or once-traversals, it is a function that always fails.

    vfail :: GAT a
    vfail _ _ = mzero

One of the two above combinators is chosen based on the coverage and used as a unit for the fold on the list of specific visits. In cases when the context updaters are present, they are also folded, with the unit being the constant function. The parameters g and f of the mother function presented above are chosen based on the coverage and traversal directions by looking up Figure 5.8. For instance, the instance of the traverse function for contextual accumulating transformations is given as follows.

    instance Data t => Traversal AccTrans Ctx c a t
        ([GenU c a] -> [GenCAT c a]
        -> c -> a -> t -> Maybe (a, t))
        where traverse _ _ cov vd hd us vs =
            travt cov vd hd (foldC us)
            (foldV (catchv cov) vs)

The function travt looks up the table and partially applies mother to appropriate parameters.

    travt :: Coverage -> VD -> HD -> GCU c
            -> GCAT c a -> GCAT c a
    travt cov v h = mother (g cov v) (f cov h)
        where g :: Coverage -> VD -> GAT a -> GAT a -> GAT a
g Full  FromTop  = compose
g Spine FromTop  = compose
g Once FromTop  = choice
g Stop FromTop  = choice
g Full FromBottom = compose'
g Spine FromBottom = compose'
g Once FromBottom = choice'
g Stop FromBottom = choice'
f :: Coverage -> HD -> GAT a -> GAT a
f Full FromLeft = all_l
f Stop FromLeft = all_l
f Spine FromLeft = one_l
f Once FromLeft = one_l
f Full FromRight = all_r
f Stop FromRight = all_r
f Spine FromRight = one_r
f Once FromRight = one_r

The function foldC folds the specific context updaters. It begins with the unit (the const function), extends with the specific functions in the list. The function foldV does the same for the visits. However, for the visits, the unit will be determined by the coverage as we have just explained. This is realized by the function catchv, which determines the unit value for foldV as follows.

    catchv :: Coverage -> GAT a
    catchv Full = vsucc
    catchv Spine = vsucc
    catchv Once = vfail
    catchv Stop = vfail

Other instances of traverse are defined similarly. In cases where the context is not present, a default value for the context is needed. We use undefined because
we need a value of type a and it will never be accessed in a lazy evaluation setting.
Transformations and accumulations are converted to accumulating transformations by providing a default implementation for the missing part and passed to the 
\texttt{mother} function and the result is converted back. We omit the tedious details here for simplicity.

5.3.7 Fixpoint Traversals

So far all the traversal strategies are one-pass strategies, which means that they apply a visit at most once to one node. Consider the case of beta reduction of lambda terms with applicative order. One step of reduction on a redex might result in a new redex inside the original one. A bottom-up traversal does not always result in a beta normal form. In such cases, an innermost traversal is needed. Such traversal strategies that apply visits to a term repeatedly until they are not applicable anymore are called fixpoint traversals. An innermost traversal applies a visit to an innermost subterm and obtains a new term. It repeats this process until no such subterm exists that the visit can be successfully applied. The \texttt{innermost} strategy is defined as follows \cite{89}.

\[
\text{innermost}(s) = \text{repeat(oncebu}(s))
\]

Here the \texttt{repeat} combinator applies a strategy to a term until it fails.

The library enables the definition in a typed framework. This combinator,
along with several other primitive combinators are part of the library targeted for
advanced users. So far, we have defined these combinators: a \texttt{succ} is a strategy
that always succeeds without changing the term or the accumulator. This is the
\texttt{vsucc} function we just defined. Note that it is also merely a curried version of the
\texttt{return} function of the \texttt{Maybe} monad. Not very surprisingly, the strategy \texttt{fail} that
always fails is the \texttt{vfail} function we defined in Section 5.3.6. The \texttt{try} strategy \cite{89}
takes another strategy and tries to apply it. If it fails, the \texttt{succ} strategy is used:

\begin{verbatim}
try ::= GAT a -> GAT a
try s = s `choice` vsucc
\end{verbatim}

Now, the \texttt{repeat} combinator \cite{89} is defined in terms of \texttt{try} recursively.\footnote{To avoid name clash with Prelude.\texttt{repeat}, it is named \texttt{rep}.}

\begin{verbatim}
rep ::= GAT a -> GAT a
rep s = try (s `compose` rep s)
\end{verbatim}

Note that passing an identity transformation (one that always succeeds and returns
the original term as the modified term) to \texttt{repeat} will cause an infinite loop. Notice
that an \texttt{outermost} strategy is symmetric to \texttt{innermost} \cite{89}:

\begin{verbatim}
outermost(s) = repeat(oncetd(s))
\end{verbatim}

Therefore, they both can be defined as instances of a more general \texttt{xmost} combi-
nator with the help of \texttt{mother}.
xm :: (GAT a -> GAT a -> GAT a) ->
    (GAT a -> GAT a) ->
    GCU c ->
    GCAT c a ->
    GCAT c a
xm g f u s c = rep (mother g f u s c)

By choosing g from choice and choice' and f from one_l and one_r, innermost
and outermost traversal strategies in both directions can be defined.

The aforementioned beta reduction application can be defined with innermost
or outermost traversals depending on the reduction strategy. The following two
Haskell functions implement applicative and normal-order beta reductions, respec-
tively. The data type Lam models lambda calculus and is shown as an example in
Appendix A.

appEval :: Lam -> Lam
appEval = innermost Trans NoCtx FromLeft
    (reduce ‘whenever‘ isRedex)

normEval :: Lam -> Lam
normEval = outermost Trans NoCtx FromLeft
    (reduce ‘whenever‘ isRedex)

isRedex :: Lam -> Bool
isRedex (App (Abs _ _) _) = True
isRedex _ = False

A visit reduces the term if it is a redex and fails otherwise. The innermost or
outermost traversal strategy applies such a visit repeatedly to some subterm until
it contains no redex anymore. A one-step reduction is performed by a full traversal
searching for occurrences of the bound variable. A list of locally bound variables
is passed as a context so that they are not substituted. The reduce function will be presented in Section 5.4.

5.4 Examples

In this section, we explore a few more sophisticated traversals and demonstrate how to implement them with our library. Suppose we again want to increase salaries in a company, but we only have a limited budget. We keep traversing the company data structure, increasing everyone's salary until the budget is all spent. The incS function then needs to know the total amount increased for the already visited people. This problem can be implemented by using an accumulating transformation. The remaining budget is passed along the traversal. Whenever we increase a salary, the increment has to be taken from the budget. The salary should not change if the budget is exhausted. The visit works on Salary values as did incS. The difference is that it returns a new budget paired with the changed salary.

\[
\text{incBud} :: \text{Data } t \Rightarrow \\
\text{Float} \rightarrow \text{Float} \rightarrow t \rightarrow \text{Maybe} (\text{Float},t) \\
\text{incBud} \text{ bud } k = \text{acctrans} (\text{always} (\text{incSbud } k)) \text{ bud}
\]

\[
\text{incSbud} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Salary} \rightarrow (\text{Float},\text{Salary}) \\
\text{incSbud } k \text{ c } (S \ s) = (c-i,S (s+i)) \\
\text{where } i = \text{min} (s*k) \ c
\]

In this application, if the budget is exhausted, those who are visited later in the traversal (in this case, those at the right and the bottom) are left without an
increase, which is not a fair strategy. A more sophisticated approach is to examine the salaries of all employees and the budget and then decide what to do with each individual salary. We can imagine different strategies. A social sensitive increase would start increasing the lowest salaries first. In a greedy approach, we would start with the highest salaries. Any such scheme can be passed as a parameter to a smart increase function. The scheme is a function that takes a list of all salaries and returns a list of new salaries. The company data structure is traversed and the salaries are collected in a list passed to the scheme. The salaries are replaced with the ones in the new list. It appears that two passes are needed to accomplish the whole task. However, thanks to lazy evaluation, we can implement it with just one pass using a trick devised by Bird in 1984 [5, 16]. The visit, which is an accumulating transformer, works on Salary values. The old salary is appended to the salary list. A new salary is taken out of the new list and replaces the old salary. The new list is obtained by applying the scheme to the old salary list, which is just the first component of the result of the smart increase function. Since the visit never fails and the traversal is a full-traversal, we can safely assume that the return value is never Nothing.

\[
\text{incSmt :: Data t =>} \\
\quad ([\text{Float}] \to [\text{Float}]) \to t \to ([\text{Float}], t) \\
\text{incSmt scheme } t = \text{fromJust (acctrans (always v) [] t)} \\
\quad \text{where } v \text{ a (S s) = (a++[s],S (new!!length a))} \\
\quad \text{new = scheme (fst (incSmt scheme t))}
\]

The above smart increase function provides endless possibilities. As an example, we show the greedy scheme as follows.
greedy :: Float -> Float -> [Float] -> [Float]
greedy bud k ys = ys3
  where (ys1,xs) = ixSort ys [1..]
      (_,ys2) = foldr f (bud,[]) ys1
      (_,ys3) = ixSort xs ys2
      f s (b,ys) = let i = min (s*k) b
                   in (b-i,(s+i):ys)

ixSort :: Ord a => [a] -> [a] -> ([a],[b])
ixSort xs ys =
  unzip $ sortBy ((\(x,_) (y,_)->compare x y) $ zip xs ys

The list of all salaries is zipped with an index list [1..] and is sorted by the salaries. We then perform a right fold, which increases salaries sequentially from the right, to obtain a new salary list zipped with the indices. The result is then sorted again by the indices to recover the original order and unzipped. The social sensitive scheme can be similarly defined using a left fold instead.

Now, let us consider the problem of beta reduction we brought up in Section 5.1.1. Our task is to implement a one-step beta reduction on a redex. This problem can be solved with a contextual transformation.

reduce :: Lam -> Lam
reduce (App (Abs v e) d) = fromJust (    
transformCB (mk upd) (always $ subst v d) [] e)
reduce e = e

upd :: [Name] -> Lam -> [Name]
upd bv (Abs v _) = v: bv
upd bv _ = bv

subst :: Name -> Lam -> [Name] -> Lam -> Lam
subst v d bv e@(Var (V v')) | v'==v && notElem v bv = d
subst _ _ _ e = e
The \texttt{reduce} function performs a bottom-up recursive transformation on the body of a beta redex. This context-sensitive transformation substitutes all the free occurrences of the formal parameter with the actual parameter. The context is a list of bound variables. It is updated by the \texttt{upd} function. The \texttt{subst} function takes the formal parameter, the actual parameter, a list of bound variable, and a term. If the term matches the formal parameter and is not bound, it is substituted by the actual parameter and otherwise unchanged.

5.5 Application in Monadification

The library we have described in this chapter has been successfully applied in the implementation of the monadification algorithm developed in Chapter 3. The full Haskell abstract syntax consists of about 10 data types and at least 30-40 constructors in total. Repeatedly implementing recursions over such structures is tedious and non-modular. In the implementation, we needed numerous such recursions, such as collecting free variables or bound variables and locating type signatures and definitions. The generic traversals greatly reduced the amount of code. A particular example is lifting a subexpression, $\mathcal{L}(e', e)$, described in Section 3.4.1. The lifting is done in one pass. The lifted subexpressions leave holes in the original expression, and these holes should be plugged with a newly generated variable. At the same time, the lifted expression should be returned such that a new monadic binding can be generated. As we described in Section 3.4.1, the liftability relies on bound variables at the location of the subexpression. The type
of this function is defined as follows.

\[
\text{liftE} :: \text{[HsName]} \rightarrow \text{MonCtx} \rightarrow \text{HsExp} \rightarrow \text{HsExp} \rightarrow \text{Maybe (HsExp,HsExp)}
\]

The arguments are: bound variables, the monadification context (see Section 3.4, do not confuse the monadification context with the context of generic traversals), the new subexpression (the fresh variable), and the original expression. The result is an optional pair of the changed expression and the original subexpression.

We can model the function as a once-traversal with a context being the bound variables and an accumulating transformation that does the replacing. The original subexpression replaced is returned as the result accumulator. We present here the full code to illustrate how the use of a traversal function can implement a highly complicated program transformation on full Haskell in only 40 lines.

\[
\text{liftE bv c@(ls,m) ne e} =
\text{onceAcctransC}
\quad (\text{mk upe} ++ \text{mk upd} ++ \text{mk upm})
\quad (\text{malways $ qte ne})
\quad (\text{bv,ls})
\quad \text{undefined}
\quad e
\]
\text{where}
\[
\text{qte ne (bv,ls) _ e} =
\quad \text{if any ('isKApp' e) ls' &&}
\quad \text{all ('notElem' bv) (freeVarsE e)}
\quad \text{then Just (ne,e)}
\quad \text{else Nothing}
\quad \text{where ls' = concatMap}
\quad \quad (\text{(\text{vs,n}) ->})
\quad \quad \quad \text{if length vs==1}
\quad \quad \quad \text{then [(head vs,n)]}
\]
else []
  ls
  upe (bv,ls) (HsLambda l ps e) = (bv++vs, fst $ maskCtx vs c)
  where vs = concatMap varsP ps
  upe (bv,ls) (HsLet ds e) = (bv++vs, fst $ maskCtx vs c)
  where vs = concatMap decVarsD ds
  upe c _ = c

upd c d = upd2 (upd1 c d) d
upd1 (bv,ls) d = (bv, concatMap k ls)
  where k ([x],n) = [[[x],n]]
        k z@(x:xs,n) =
         let ys=decVarsD d in
         if null ys then [z]
         else if head ys==x then [(xs,n)] else []
  upd2 (bv,ls) (HsPatBind _ p (HsUnGuardedRhs _ _) ds) =
        (bv++vs, fst $ maskCtx vs c)
  where vs = varsP p++concatMap decVarsD ds
  upd2 c _ = c

upm (bv,ls) (HsMatch _ n ps (HsUnGuardedRhs _ _) ds) =
    (bv++vs, fst $ maskCtx vs c)
  where vs = n:concatMap varsP ps++concatMap decVarsD ds
  upm c _ = c

In this example, the traversal context, which is given by the collection of bound
variables, is changed whenever a binding (an expression, a declaration, or a match)
is introduced (context functions upe, upd, and upm). The visit function qte checks
for any capture of a free variable if the subexpression being visited is lifted. If it
is not the case, it perform a successful accumulating transformation. The generic
traversal strategy onceAcctransC carries the visit and ensures it is applied only
once. The initial accumulator in this traversal is undefined because the visit itself
does not rely on the initial accumulator value.

5.6 Related Work

Without generic programming, functional programs suffer from a scalability problem. Generic functions whose behavior is defined inductively on the structures of the data can be scaled to large data structures easily without extra effort. They can even be reused for data types that are not yet defined. Our problem domain is program transformation and program generation, in particular, automatic monad introduction [23] and parameterized program generation [20]. Practical problems on large data structures such as the abstract syntax of Haskell and Fortran call for generic term traversals. Various approaches can be used for the purpose of generic term traversals. The program transformation tool Stratego/XT (see Section 2.1.1) implements a set of strategies, many of which are related to generic traversals [86]. However, the language lacks a strong static type system. As explained in Section 2.2.1, Generic Haskell [13, 53] is a language extension to Haskell. It allows one to define purely generic functions. But a generic function is not a first-class citizen in Generic Haskell, which means that we can not define higher-order generic functions.

In [46] and [45], a combinator library (Strafunski) including generic traversal combinators is presented. These papers categorize a strategy into type preserving and type unifying strategies. To some extent, they correspond to the concepts of transformations and accumulators proposed here. A set of traversal schemes is
also defined. These schemes, along with those defined in Stratego [87, 89, 11] are the main inspiration of our categorization of the problem. In [41] Lämmel proposed a highly parameterized generic traversal combinator. We implement these traversal strategies in a statically typed framework proposed in [42, 43, 44]. Hinze, Löh, and Oliveria propose a spine view of data types and use it to define underlying SYB generic functions [32]. Because they are mostly compatible with the original SYB functions other than the embedded type information, this approach could be used to replace the underlying mechanism of creating generic transformations/accumulations as well.

Contextual visits are closely related to scoped dynamic rewrite rules [84, 12]. Dynamic rules are generated at run-time and can access their context. A scope can be imposed to remove rules after they are not valid anymore. One problem with scoped dynamic rules is that it is necessary to inline the definition of the traversal strategy so that the scope can be included in the traversal of subterms. The approach therefore suffers from a modularity problem. In our library, context is abstracted and modularized. It is taken care of by the recursive traversal strategy and passed to the visit so that the visit does not need to worry about the scope.

In [81], van den Brand et al. categorize a traversal into transformation, accumulation, and accumulating transformation. This agrees with our categorization. In fact, we borrowed these terms from [81]. They also identify certain properties of traversals and place them in the corresponding positions in the "traversal cube". We have enriched the cube by extending the coverage axis.
5.7 Summary and Future Work

In this chapter, we have presented an extension to the scrap-your-boilerplate approach proposed by Lämmel and Peyton Jones. We have analyzed the problem domain of generic traversals and have extracted five orthogonal parameters of a traversal. We have defined one universal generic traversal combinator that can be parameterized to cover the whole problem domain space. In summary, these combinators provide the programmers these choices:

- The visit. We can perform a transformation that modifies a node, an accumulation that gathers information from nodes along the traversal, or an accumulating transformation that does both.

- The context. The action might rely on the path from the root node to the current node. A customized context can be maintained by a context updater function and carried to the visit function.

- The vertical traversal order. A traversal can start from the top of the term and moves down or the opposite direction.

- The horizontal traversal order. A traversal can visit from left to right or the opposite direction.

- The coverage. A traversal can visit all the nodes, bypass children of certain nodes, visit along a spine from the root to a leaf, or stop after a successful visit.
The clients can easily choose the appropriate strategy and focus on the “interesting parts”, the recursion is performed by the generic traversal combinators.

In addition to this high-level interface, we have also defined a set of primitive combinators that can be used to define additional recursive traversal strategies.

However, our library only addresses the problem of transformations and accumulations on one term. Problems that involve parallel traversing two terms, such as generic zip [43, 34], cannot be handled. Although our combinators are fairly general, there is still room for improvement. Regardless of the two traversal directions, we always favor the vertical direction over the horizontal direction, which means we always implement a depth-first traversal. One possible extension is to have symmetric breadth-first traversals. Moreover, we only have one and all strategies as our one-layer strategies. We can also consider strategies that visit only some of of direct subterms of a term. We believe these features will extend the traversal space and complement the traversal library.
Chapter 6 – Conclusion

In this dissertation, we have investigated the idea of update programming. The goal of the research was to devise a safe approach to software maintenance. In our context, safety refers to syntax correctness as well as type correctness of the resulting programs. A basic theme throughout the dissertation is to view programs as instances of abstract data types rather than streams of characters. This view, when implemented in a strongly typed language, provides syntax correctness for free. Type correctness, however, is not easy to achieve. We have described an update language HULA and its core functional subset in Chapter 4. We have developed structural criteria for update programs that can guarantee the type correctness of the produced object programs. The goal was to express a wide range of program updates in this update language that would pass the structural criteria check. However, it turned out that the strictness of the criteria imposed significant limitations on update programs. Complex updates that require elaborate analysis of scoping rules cannot be implemented in HULA, at least not with a safety guarantee. Instead, specific approaches need to be designed to accomplish these individual goals. Monadification is such a concrete example. In Chapter 3, we formally defined the problem and proposed a detailed algorithm. The complexity of the problem prevents it from being expressed in HULA. To support the implementation of sophisticated updates like monadification, we have also inves-
tigated an alternative approach based on generic recursion. We have presented a library, which provides a rich collection of generic traversals strategies that work on arbitrary data structures. This library can be used in program updates in general. In particular, it has been successfully employed in the implementation of the monadification algorithm for traversals of terms representing Haskell programs.

In summary, in our search for an approach to update programming we found a trade-off between safety and expressiveness. In this dissertation this result emerges in two different language approaches, the update calculus and the recursion library. Future research might bring about a unified update language that allows update programmers to choose for each individual update a separate safety level.
Bibliography


APPENDICES
Appendix A – Algebraic Data Types

Haskell is a purely functional language. All computations are done via evaluation of expressions to yield values. Each value has an associated type. A type can be roughly regarded as a set of values. Examples of types are Int (integers), Bool (booleans), [Int] (homogeneous lists of integers), (String, Int) (string, integer pairs). Examples of values that have these types are:

```
5 :: Int
True :: Bool
[1,2,3] :: [Int]
("Hello", 42) :: (String, Int)
```

The :: symbol is Haskell’s notation for associating a value with a type. In functional languages, functions are “first class” citizens. Functions can be passed as arguments to functions, returned as results, used as values to construct other values, etc. Functional types are denoted by ->. Int -> Bool is a functional type whose input is an integer and output is a boolean. Haskell also supports polymorphic types, types that are universally quantified. A polymorphic type is a family of types. For example, forall a. [a] represents all list types. In this type, a is a universally quantified type variable. The universal quantifier can usually be omitted. All type variables in a type expression are implicitly universally quantified. Functions can also have polymorphic types. The length function is an example of polymorphic functions. It takes any list and returns its length:
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs

A big advantage of functional language is the flexibility to freely define new algebraic data types. A simple example of data type is `Bool`, defined as follows.

```haskell
data Bool = True | False
```

`True` and `False` are called data constructors, because they are used to construct values. When no confusion could arise, we also refer to data constructors simply as constructors. Such a data type definition whose constructors take no parameters can be regarded as enumerations. Another example of an enumeration type is a data representing colors.

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

The constructors can also take parameters. For example, if we want to model geometric objects, we cannot simply enumerate rectangles, circles, triangles, etc. We must specify the center (a pair of float number representing coordinates) and the radius (a float number) of the circle or the vertices of the triangle. Such a data type can be defined as follows, where `Point` is a type synonym.

```haskell
type Point = (Float,Float)
data Geo = Circle Point Float
       | Triangle Point Point Point
       | Rect Point Point
```

Data types with multiple parameterized constructors are somewhat like tagged unions types in other languages.
Data types can also be recursively defined. For example, suppose we define binary trees. A tree is either a node with children or a leaf. Data might be stored in the nodes and/or leaves. The children are also trees. Consider a simple binary tree which stores only integers in the nodes. Such a tree can be defined as follows.

\[
data \text{ Tree} = \text{ Node Int Tree Tree} \\
| \text{ Leaf}
\]

This is a recursive definition where the data constructor \text{Node} takes an integer and two trees (left and right child tree) as parameters. One can construct a tree by applying constructors to arguments. For example, \text{Node 3 (Node 2 Leaf Leaf)} \text{Leaf} represents a tree visualized in the following figure.

Suppose now we want to store data in both nodes and leaves. The definition of \text{Leaf} has to be adapted for an extra parameter.

\[
data \text{ Tree} = \text{ Node Int Tree Tree} \\
| \text{ Leaf Int}
\]

We might also want to store values other than integers in the tree. Polymorphism allows us use to define parameterized data types. The following definition shows a tree which can store anything in its nodes and leaves.
data Tree a = Node a (Tree a) (Tree a) | Leaf a

Tree a represents a family of trees. Tree is more like a template which can yield infinitely many data types by instantiating a with different types. Therefore, Tree is called a type constructor because it builds types from types.

We can define functions on data types. A function on trees could distinguish the two different cases and extract the data stored with constructors by pattern matching. A pattern is basically a data constructor applied to a list of (free) variables or subpatterns. When a pattern is matched against a value, parts of the value are bound to the pattern variables. Suppose we use an integer tree as a binary search tree. Then we can define a search function as follows.

search :: Int -> Tree Int -> Bool
search n (Leaf x) = n==x
search n (Node x l r) = if x==n
  then True
  else if n<x then search n l
               else search n r

If the parameter is a leaf, the first pattern is matched and the value stored in the leaf is bound to x. In the case where the tree is a node, its stored value and both children are bound to x and l and r, respectively. Recursive search is performed if necessary.

An idea throughout this dissertation is to represent programs as abstract data types. This is done by defining a data type to represent the programs. Every constructor represents a syntax category. For example, lambda calculus has three
syntax categories: variables, applications, and lambda abstractions. These can be defined as the following data type.

```haskell
data Lam = Var String
    | App Lam Lam
    | Abs String Lam
```

In Section 5.3.7, we showed a function for beta reduction on this data type.
Appendix B – Type Classes

Observe the search function for integer trees in Appendix A. Wouldn’t it be great if we could reuse the function for polymorphic trees? In such a case, we can define a search function once and for all. The polymorphic search function would have the following type:

\[
\text{search} :: \text{a} \rightarrow \text{Tree a} \rightarrow \text{Bool}
\]

However, how can we have a binary tree of, say, geometric objects, that cannot be ordered? Generally, the \(\text{==}\) and \(<\) operators are not defined for an arbitrary data type. Therefore, such a polymorphic search function cannot be defined. Do we have to take the pain to define search functions for integers, float numbers, etc. individually? Thanks to type classes of Haskell, the answer is no. With type classes, we can define a “semi-polymorphic” search function. The key issues of the problem is the \(\text{==}\) and \(<\) operators, which are not defined for all data types, but for certain data types that can be ordered. These data types are collected and grouped in a type class. A type class is a set of data types with a common set of functions or variables. \text{Eq} is such a class that has an equality operator \(\text{==}\) defined. In other words, every type that is an instance of \text{Eq} class has an equality operator. The type class declaration is as follows.

\footnote{In fact, \text{Eq} also has an inequality operator \(\neq\). We omit it here for simplicity.}
class Eq a where
     (==) :: a -> a -> Bool

When a data type is defined, it can be declared an instance of Eq by defining ==.

For example, Color can be an instance of Eq:

    instance Eq Color where
       Red == Red   = True
       Green == Green = True
       Blue == Blue  = True
       Yellow == Yellow = True
       _ == _       = False

Any color is equal to itself but nothing else. Such a trivial definition can actually
be derived by Haskell automatically. One only needs to add a deriving clause to
the data type definition:

    data Color = Red | Green | Blue | Yellow
        deriving (Eq)

The Eq class classifies types that the == operator can be defined. However, another
problem with the polymorphic search function is the inequality operator <, which is
also not defined for all data types, even not for some data types that are instances
of Eq. For instance, it is natural to define an == operator for geometry objects by
comparing corresponding parameters, but there is not an obvious way to define
the < operator. Which one is smaller, a circle or a triangle? On the other hand,
if we were able to tell the less than relation between two objects, we can usually
also tell if they are equal. In this sense, we can define a type class Ord that has a
< operator. Ord is a subclass of Eq. A subclass inherits all the members from the
parent class, and usually has more members. The subclass relation is denoted as a type class constraint in the type class declaration.²

\[
\text{class Eq a => Ord a where}
\]

\[
(<) :: a \rightarrow a \rightarrow \text{Bool}
\]

The preceding definition declares a type class \texttt{Ord} with an operator \(<\). A type \(a\) can be an instance of \texttt{Ord} only if it is also an instance of \texttt{Eq}. The \texttt{Ord} type class can also be derived automatically. The default definition assumes the order in which the data constructors are declared. For example, we can derive a default instance of \texttt{Ord} for the data type \texttt{Color}, as follows.

\[
\text{data Color = Red | Green | Blue | Yellow}
\]

\[
\text{deriving (Eq)}
\]

This defines a complete order \texttt{Red} \(<\) \texttt{Green} \(<\) \texttt{Blue} \(<\) \texttt{Yellow}. In the cases where data constructors take arguments, the argument types must all be instances of \texttt{Eq} as well. When two values are compared, their data constructors are first compared. In case of a tie, the first arguments are compared, and so on.

With the \texttt{Ord} class, we can now define a polymorphic search function:

\[
\text{search :: Ord a => a \rightarrow Tree a \rightarrow Bool}
\]

\[
\ldots
\]

The function definition is actually unchanged. The type signature should be read as, \texttt{search} is a function that takes a value of any type \((a)\) and a tree of that type \((\text{Tree } a)\) and returns a boolean, but \(a\) has to be an instance of \texttt{Ord} type class.

²Again, we omit other members of the \texttt{Ord} class for simplicity.
The `<` operator is defined for all types in type class `Ord` with separate instances for each type class. Compared to the `length` function, in which one instance is defined for all types, this kind of polymorphism is called *ad hoc* polymorphism (compared to parameteric polymorphism).
Appendix C – Monads

There are two categories of functional languages: pure and impure. In the rest of this appendix, the term “language” always means functional language unless otherwise specified. Haskell and Miranda are examples of purely functional languages. Impure languages include Scheme and Standard ML. The key difference is that impure languages allow side effects, such as assignments or exceptions. Purely functional languages are simpler and easier to deal with and may benefit from lazy evaluation. Impure languages provide efficiency in practical issues by borrowing features commonly found in imperative languages [66, 92].

The concept of monad bridges the gap. The use of monads integrates impure effects into purely functional languages. The monad can trace its root back to category theory and was introduced by Moggi to structure functional programs [62, 63]. But one does not need a background of category theory, which is rather abstract and difficult, to master monads. In general, monads represent computations (rather than values). Monads are versatile. They can used to model a wide range of computations. In this appendix, we will explore the ideas of monads through the IO monad and the Maybe monad. Both are typical and simple monads.

I/O is one side effect that is often a desirable feature in a purely functional language. Without side effects, one program cannot interact with the user and
cannot get input and output. In [33], Hudak and Sundaresh provide a good survey of functional I/O.

With the introduction of monads, a better solution to I/O, monadic I/O, was made possible. In [66, 92], Peyton Jones gives a comprehensive account of monadic I/O. Essentially, I/O is a function that can be conceptually viewed as having the following type.

\[
\text{type } \text{IO } a = \text{World} \to (a, \text{World})
\]

An IO monad is a computation which takes a initial state of an external world and returns a value and a changed state of the world. In the changed world, either some output has been produced or some input has been consumed. Two simple examples of I/O actions are \texttt{getChar} and \texttt{putChar}, which read (write) a character from (to) the world:

\[
\begin{align*}
\text{getChar} & : \text{IO } \text{Char} \\
\text{putChar} & : \text{Char} \to \text{IO } ()
\end{align*}
\]

Since we do not expect a result from \texttt{putChar}, it simply returns a unit value () of type (). We can compose complex I/O functions by gluing together basic functions. If we want to echo the character read from the console, we need to glue \texttt{getChar} and \texttt{putChar} together to form a new function \texttt{echo}. Such a glue operator is \texttt{>>=}, which reads “bind”:

\[
\text{echo} = \text{getChar} >>= \text{putChar}
\]

The >>= operator takes the result from the first action and passes it to the second one. It glues two computations sequentially. It has the following type:
(>>=) :: IO a -> (a -> IO b) -> IO b

Suppose we need to read two characters and return them as a pair. We could use the bind operator to glue two getChar actions and a function that returns a pair of characters. But this function has to have type IO (Char, Char). Another operator return is provided for the purpose. The return function constructs a function that does no I/O, but returns a result. The type of return is as follows.

return :: a -> IO a

Essentially, the return operator turns a value to a computation that returns the value and does nothing else. With the help from bind and return operators and getChar, we can define our function as follows.

getPairedChar :: IO (Char, Char)
getPairedChar =
  getChar >>=
  (\c1 -> getChar >>=
    (\c2 -> return (c1,c2)))

The first bind operator takes the result from the first getChar and passes it to a function which binds it to variable c1; the second bind operator then binds the second result to variable c2; finally the two characters read are paired and returned as the result of the compound action.

This looks very much like a sequence of imperative actions. The notation, however, quickly gets awkward with the growth of the functions. In Haskell, a syntactic sugar, dubbed do notation, is provided for brevity. With the do notation, the above function can be written as follows.\footnote{The curly brackets {} and the semicolon ; are optional.}


```haskell
getPairedChar :: IO (Char, Char)
getPairedChar = do {c1 <- getChar;
c2 <- getChar;
return (c1,c2)}
```

Note that although this looks more like imperative actions, the do notation is just for syntactic convenience. In the above definition, one should indeed read: the result of `getChar` is bound to variable `c1`, instead of being assigned to memory location `c1`, which would be the case of imperative languages. The do notation can be readily translated into conventional expressions using the following rules [67].

\[
\text{do \{e\}} \quad = \quad e
\]

\[
\text{do \{e; stmts\}} \quad = \quad e \gg= \ \_ \_ \rightarrow \ \text{do \{stmts\}}
\]

\[
\text{do \{x <- e; stmts\}} \quad = \quad e \gg= \ \_\rightarrow \ \text{do \{stmts\}}
\]

In general, a monad in Haskell is a unary type constructor with two\(^2\) associated functions, which is expressed by a type class (more precisely, as a constructor class) `Monad`.

\[
\text{class Monad m where}
\text{ return :: a -> m a}
(\gg=) :: m a -> (a -> m b) -> m b
\]

This definition expresses that any type constructor `m` can be regarded as a monad once these two operations have been defined. In addition, the monadic structure requires `return` to be a left and right unit of `\gg=` and `\gg=` to be associative in

\(^2\)Strictly speaking, `\gg` and `fail` are also members of the `Monad` type class in Haskell. `\gg` is merely a shortcut for a specific instance of `\gg=` and `fail` defaults to ⊥.
a certain sense, that is, the definition of the monad operations should obey the following monad laws:

\[
\begin{align*}
& m \mathbin{>>=} \text{return } = m \\
& \text{return } x \mathbin{>>=} f = f x \\
& (m \mathbin{>>=} f) \mathbin{>>=} g = m \mathbin{>>=} (\lambda x \to f x \mathbin{>>=} g)
\end{align*}
\]

These laws are not enforced by Haskell, it is the programmer’s responsibility to define the operations in such a way that these laws hold.

A seemingly very different kind of computation, error handling, can also be expressed with monads. One simple monad for error handling is the `Maybe` monad, defined as follows.

```haskell
data Maybe a = Just a | Nothing
```

A `Just` constructor denotes a normal state associated with a value of type `a`, while a `Nothing` constructor denotes an error state where no value should be stored. Instances of the two basic monad operations, `return` and `>>=`, are defined for the `Maybe` type as follows.

```haskell
instance Monad Maybe where
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  return    = Just
```

The `>>=` operation works as follows. If the previous computation has produced a proper value (indicated by the enclosing constructor `Just`), the value obtained so far (`x`) is passed on for further computation (`k`). But if an error has occurred (indicated by the constructor `Nothing`), this error state is propagated, regardless of the following computation.
With the *Maybe* monad, programmers can write code in an imperative way to handle computations with potential errors but without worrying about error handling, because it is hidden in the >>= operation. For example, the `eval` function defined in Section 3.1 can be converted to a monadic form as follows.

```haskell
evalM :: Expr -> Maybe Int
evalM (Con x) = return x
evalM (Plus x y) = do i <- evalM x
                  j <- evalM y
                  return (i+j)
 evalM (Div x y) = do i <- evalM x
                 j <- evalM y
                if j==0 then Nothing
                 else return (i `div` j)
```

The only place programmers have to do explicit error handling is where the dividend evaluates to zero. In the cases where evaluation of subexpressions results in errors (indicated by `Nothing`), the error is propagated by the >>= operation, and the following computation will not be carried out at all.

Monads are a purely functional data type. However, it provides functional programmers the same convenience imperative programmers enjoy. Monads find their applications in many areas of functional programming.