On Functional Logic Programming and its Application to Testing

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)

der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Sebastian Fischer

Kiel, 2010

1. Gutachter	Prof. Dr. Michael Hanus
2. Gutachter	Prof. Dr. Herbert Kuchen
3. Gutachter	Priv.-Doz. Dr. Frank Huch
Datum der mündlichen Prüfung	27. Mai 2010

In Deiner Sprache saßen andere Augen.
Klavki, †2009

Contents

1 Introduction **1**
1.1 Syntax of programs 1
1.2 Organisational remarks 4

2 Declarative Programming **5**
2.1 Functional programming 6
2.1.1 Type polymorphism and higher-order functions 7
2.1.2 Lazy evaluation 9
2.1.3 Class-based overloading 11
2.1.4 Summary 19
2.2 Functional logic programming 19
2.2.1 Logic variables 20
2.2.2 Nondeterminism 21
2.2.3 Lazy nondeterminism 23
2.2.4 Call-time choice 25
2.2.5 Search 29
2.2.6 Constraints 31
2.2.7 Summary 34
2.3 Chapter notes 34

3 Generating Tests **35**
3.1 Black-box testing 35
3.1.1 Property-based testing 36
3.1.2 Defining test-case generators 38
3.1.3 Enumerating test cases 42
3.1.4 Experiments 51
3.1.5 Implementation 57
3.1.6 Summary 59
3.2 Glass-box testing 60
3.2.1 Demand-driven testing 61
3.2.2 Fair predicates 67
3.2.3 Practical experiments 70
3.2.4 Summary 75
3.3 Chapter notes 76

4 Code Coverage **79**
4.1 Control flow 81
4.1.1 Rule coverage 82
4.1.2 Call coverage 82
4.2 Data flow 84
4.2.1 Constructors flow to patterns 85
4.2.2 Functions flow to applications 86
4.2.3 Comparison with Control Flow 87
4.3 Monitoring code coverage 88
4.3.1 Lazy coverage collection 88
4.3.2 Combinators for coverage collection 91
4.3.3 Formalisation of program transformation 96
4.3.4 Implementation of coverage combinators 100
4.4 Experimental evaluation 103
4.5 Summary 108
4.6 Chapter notes 109

5 Explicit Nondeterminism **111**
5.1 Nondeterminism monads 111
5.1.1 Laws 112
5.1.2 Backtracking 113
5.1.3 Fair search 120
5.1.4 Experiments 123
5.1.5 Summary 126
5.2 Combining laziness with nondeterminism 127
5.2.1 Explicit sharing 128
5.2.2 Intuitions of explicit sharing 132
5.2.3 Laws of explicit sharing 135
5.2.4 Implementing explicit sharing 140
5.2.5 Generalised, efficient implementation 146
5.2.6 Summary 153
5.3 Chapter notes 154

6 Conclusions **157**
6.1 Declarative programming promotes concepts that increase the potential for abstraction. 158
6.2 The mechanism to execute functional logic programs is tailor made for generating tests. 159
6.3 Declarative programs call for new notions of code coverage. . 162
6.4 Lazy nondeterministic programs can be turned into equivalent purely functional programs. 164

A Source Code **167**
A.1 ASCII versions of mathematical symbols . . . 167
A.2 Definitions of used library functions . . . 167
A.3 Abstract heap data type . . . 169
A.4 Implementation of BlackCheck . . . 171
A.4.1 Input generators . . . 171
A.4.2 Testable types . . . 173
A.4.3 Property combinators . . . 173
A.4.4 Test annotations . . . 174
A.4.5 Testing with different strategies . . . 174
A.4.6 Auxiliary definitions . . . 176
A.5 Implementation of GlassCheck . . . 179
A.5.1 Parallel answers . . . 182
A.5.2 Search strategies . . . 184
A.5.3 Tree search . . . 185
A.6 Implementation of explicit sharing . . . 186
A.7 Benchmarks for explicit sharing . . . 188
A.7.1 Permutation sort . . . 188
A.7.2 Naive reverse . . . 189
A.7.3 Functional logic last . . . 190
A.7.4 Nondeterministic lists . . . 191

B Proofs **192**
B.1 Functor laws for $(a \rightarrow)$ instance . . . 192
B.2 Monad laws for *Tree* instance . . . 193
B.3 Laws for *CPS* type . . . 196
B.3.1 Monad laws . . . 196
B.3.2 *MonadPlus* laws . . . 197

List of Figures

2.1 Safe placement of 8 queens on an 8×8 chessboard 28
2.2 Safe placement of 15 queens on an 15×15 chessboard . . . 33

3.1 Search space for an *arbitrary* value of type $[Bool]$ 40
3.2 Level diagonalisation of $[Bool]$ values 50
3.3 Combined randomised level diagonalisation of $[Bool]$ values . 51

4.1 Computation of data flow for *reverse* $[x, y]$ 90
4.2 Syntax of core Curry programs 97

5.1 The laws of a monad with nondeterminism and sharing . . . 135
5.2 The laws of observing a monad with nondeterminism and sharing in another monad with nondeterminism 137

List of Tables

3.1 Comparison of different strategies for black-box testing 56
3.2 Comparison of black-box and glass-box testing 71

4.1 Data flow in applications of *reverse*. 86
4.2 Experimental results of coverage-based testing 106
4.3 Sizes of reduced sets of tests for different coverage criteria . . 107

5.1 Performance of different monadic search strategies 124

Acknowledgements

I thank my supervisor Michael Hanus for giving me the opportunity to research in his group. I have enjoyed the free and inspiring environment as well as the thoughtful advice you provide which allowed me to find and work on topics that catch my interest. Many interesting discussions with you, Bernd Braßel, Frank Huch, and Fabian Reck have influenced my work beneficially.

Moreover, I thank all colleagues with whom I could collaborate on the work presented in this thesis. Without Herbert Kuchen I would not have started my work on testing. I have enjoyed visiting you and learned a lot from your guidance during our collaboration. I thank Jan Christiansen for pushing me to implement black-box testing. Working with you on EasyCheck was a lot of fun and the beginning of interesting journeys both professionally and personally. I am glad for the opportunity to work with Oleg Kiselyov and Ken Shan. Oleg, programs you write are an invaluable source of ideas and opportunity to learn. Ken, you have an inspiring desire to aspire—and enviable ability to approach—clarity.

I am grateful to everyone who has provided feedback on preliminary drafts of this thesis. Sergio Antoy's comments on Chapter 2 have helped me generally to improve the presentation and specifically to give a more elegant implementation of the n-queens solver. Comments by Jan Christiansen and Frank Kupke have helped to improve the presentation of Chapters 3 and 4, respectively. A discussion with Jan Spitzmann lead to insights in the properties of random search discussed in Section 3.1.3.

1 Introduction

Like good wine, concepts developed in modern programming languages need to ripen before they get applied outside of research communities. Structured programming was not immediately in widespread use after Edsger Dijkstra (1968) considered GOTO statements harmful. The breakthrough of object oriented programming in $C++$ or Java came considerably later than its birth in the language Simula 67. Polymorphic typing—developed long ago in the functional programming community—has entered the mainstream in the disguise of Java generics only recently. Nondeterministic programming and search rarely find their way out of research communities.

In this thesis, we apply functional logic programming (FLP) in the field of software testing. We argue that FLP is tailor made for generating thorough tests for complex algorithms automatically and provide implementations of test tools to support this claim.

1.1 Syntax of programs

All programs in this thesis are, if not stated otherwise, written in the programming languages Haskell (Peyton Jones et al. 2003) or Curry (Hanus 2006). Haskell is a statically typed, lazy, purely functional programming language and Curry is a variant of Haskell that adds logic programming features. We discuss declarative programming in detail in Chapter 2 and only introduce the syntax of Haskell on a very basic level here. The syntax of Curry is identical with respect to this basic introduction.

Basically, a Haskell program declares data types and functions that operate on these types. For example, the type of Boolean values is defined as follows in Haskell.

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

This declaration uses the keyword **data** to define the data type *Bool*. The right-hand side of this equation lists different alternatives to construct values of type *Bool* separated by a vertical bar. Here, two constructors *False* and *True* are defined as the only values of type *Bool*.

Constructors can also have arguments and data types can be recursive. For example, we can define the type of lists of Booleans by distinguishing empty and non-empty lists.

$$\textbf{data}\ \mathit{BList} = \mathit{EmptyBL} \mid \mathit{ConsBL}\ \mathit{Bool}\ \mathit{BList}$$

With this declaration the empty list of Booleans can be constructed using *EmptyBL* and a non-empty list of Booleans can be constructed as *ConsBL b l* where *b* is a value of type *Bool* (either *False* or *True*) which denotes the first element of the constructed list and *l* is a value of type *BList* which denotes the list of remaining elements. For example, a list that contains both Boolean values can be constructed as *ConsBL False* (*ConsBL True EmptyBL*).

Data-type declarations can also be parametrised to define many different types at once. We can generalise the above data type for Boolean lists by using a type parameter.

$$\textbf{data}\ \mathit{List}\ a = \mathit{Empty} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

Here, the type variable *a* denotes an arbitrary type and *List a* denotes the type of lists with elements of type *a*. The type *BList* is equivalent to *List Bool* but we can also use the data types defined so far to define different list types. For example, *List* (*List Bool*) is the type of lists of lists of Booleans.

Haskell functions can be defined using rules that distinguish different cases by *pattern matching*. The function *null* that checks whether a given list is empty can be defined as follows.

$$\begin{aligned}&\mathit{null} :: \mathit{List}\ a \to \mathit{Bool}\\ &\mathit{null}\ \mathit{Empty} \quad\ = \mathit{True}\\ &\mathit{null}\ (\mathit{Cons}\ \_\ \_) = \mathit{False}\end{aligned}$$

The first line is a type declaration that specifies that *null* takes a list as argument and yields a Boolean as result. If we would have left out the type signature, Haskell's type inference mechanism would have determined it automatically from the implementation of *null*. The *null* predicate is implemented by two rules that distinguish the given list by matching them against constructor patterns. If the given list can be matched against the constructor *Empty* then the result of *null* is *True*, if it can be matched against the pattern *Cons* _ _ then the result of *null* is *False*. Underscores in a constructor pattern denote anonymous variables that can be matched against arbitrary values. As the *null* function matches all possible ways how lists can be constructed it is a total function that will never fail with a pattern match error.

We can also use named variables in patterns in order to access the values at the corresponding position of the input. The functions *head* and *tail* select the first element and the list of remaining elements of a non-empty list, respectively.

$$\begin{aligned} &\mathit{head} :: \mathit{List}\ a \to a \\ &\mathit{head}\ (\mathit{Cons}\ x\ \_) = x \\ &\mathit{tail} :: \mathit{List}\ a \to \mathit{List}\ a \\ &\mathit{tail}\ (\mathit{Cons}\ \_\ \mathit{xs}) = \mathit{xs} \end{aligned}$$

Both functions are partial. They fail with a run-time error when applied to the empty list.

Haskell provides an alternative syntax for data-type declarations which allows to define selector functions automatically. The following declaration of the list type is equivalent to the declaration given before but also defines the functions *head* and *tail*.

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Empty} \mid \mathit{Cons}\ \{\mathit{head} :: a, \mathit{tail} :: \mathit{List}\ a\}$$

We can also use this so called labeled-field- or record syntax[1] to construct and update lists. The list that contains only the value *False* can be constructed as follows.

$$\begin{aligned} &\mathit{falseList} :: \mathit{List}\ \mathit{Bool} \\ &\mathit{falseList} = \mathit{Cons}\ \{\mathit{head} = \mathit{False}, \mathit{tail} = \mathit{Empty}\} \end{aligned}$$

Moreover, we can change the tail of *falseList* in order to obtain a list that contains both Boolean values using the following notation.

$$\begin{aligned} &\mathit{boolList} :: \mathit{List}\ \mathit{Bool} \\ &\mathit{boolList} = \mathit{falseList}\ \{\mathit{tail} = \mathit{Cons}\ \mathit{True}\ \mathit{Empty}\} \end{aligned}$$

In a declarative programming language, the values of variables never change. Hence, the above function does not change *falseList* but yields a new updated list. We can access both *falseList* and *boolList* in an interactive Haskell environment which demonstrates that the value of *falseList* does not change.

```
> falseList
Cons { head = False, tail = Empty }
> boolList
Cons { head = False, tail = Cons { head = True, tail = Empty } }
> tail boolList
Cons { head = True, tail = Empty }
```

[1] Record syntax is not officially part of the Curry language but supported by some compilers.

The definition of the tail of *boolList* shows that even if the data-type declaration uses record syntax, values of the corresponding type can still be constructed as if it was defined without labeled fields.

Haskell provides a built-in syntax for lists that is more concise than the syntax presented so far. The empty list is denoted by $[\,]$ and a non-empty list is constructed using the constructor $(:)$ which can be written infix as in $x : xs$.[2] With this syntax, we can write $\mathit{False} : \mathit{True} : [\,]$ instead of *boolList* or even shorter $[\mathit{False}, \mathit{True}]$.

We can also use this notation in patterns as exemplified by the *length* function which is defined as follows.

$$\begin{aligned} &\mathit{length} :: [a] \to \mathit{Int} \\ &\mathit{length}\ [\,] \quad\ = 0 \\ &\mathit{length}\ (\_ : xs) = 1 + \mathit{length}\ xs \end{aligned}$$

The type declaration shows that also the type of lists is written more concisely in Haskell. Finally, the second rule of *length* contains a recursive call which shows that like data types also functions can be recursive.

Haskell provides built-in types, for example *Int* for numbers or *Char* for characters, along with corresponding operations. We do not discuss such types further because they behave as one would expect intuitively.

1.2 Organisational remarks

This thesis is structured as follows. We first discuss advanced concepts of functional programming as well as logic extensions (Chapter 2), then use them to implement automated testing (Chapters 3 and 4), show how to express the employed logic extensions purely functionally (Chapter 5), and finally summarise the presented work (Chapter 6). The appendix contains accompanying source code and proofs.

Most work presented in this thesis has been published previously. Each chapter is concluded by chapter notes in which we mention previous publications and discuss relevant related work. Moreover, we usually provide intermediate summaries after each section. Although the focus of this thesis is on testing, Chapters 2 and 5 have a more general scope and can be understood without considering the developments of the chapters in between.

As demonstrated in Section 1.1 we use a mathematical notation for typesetting source code and typewriter font for terminal sessions. Translations between the mathematical notation and corresponding ASCII syntax are given in an appendix.

[2] The empty list $[\,]$ is usually pronounced *nil* for historical reasons; $(:)$ is pronounced *cons*.

2 Declarative Programming

Programming languages are divided into different paradigms. Programs written in traditional languages like Pascal or C are *imperative* programs that contain instructions to mutate state. Variables in such languages point to memory locations and programmers can modify the contents of variables using assignments. An imperative program contains commands that describe *how* to solve a particular class of problems by describing in detail the steps that are necessary to find a solution.

By contrast, *declarative* programs describe a particular class of problems itself. The task to find a solution is left to the language implementation. Declarative programmers are equipped with tools that allow them to abstract from details of the implementation and concentrate on details of the problem.

Hiding implementation details can be considered a handicap for programmers because access to low-level details provides a high degree of flexibility. However, a lot of flexibility implies a lot of potential for errors, and, more importantly, less potential for abstraction. For example, we can write more flexible programs using assembly language than using C. Yet, writing large software products solely in assembly language is usually considered impractical. Programming languages like Pascal or C limit the flexibility of programmers, for example, by prescribing specific control structures for loops and conditional branches. This limitation increases the potential of abstraction. Structured programs are easier to read and write and, hence, large programs are easier to maintain if they are written in a structured way. Declarative programming is another step in this direction.[1]

The remainder of this chapter describes those features of declarative programming that are preliminary for the developments in this thesis, tools it provides for programmers to structure their code, and concepts that allow writing programs at a higher level of abstraction. We start in Section 2.1 with important concepts found in *functional* programming languages, namely, polymorphic typing of higher-order functions, demand-driven evaluation, and type-based overloading. Section 2.2 describes essential features of *logic* programming, namely, nondeterminism, unknown values and built-in search

[1] Other steps towards a higher level of abstraction have been *modularization* and *object orientation* which we do not discuss here.

and the interaction of these features with those described before. Finally, we show how so called *constraint* programming significantly improves the problem solving capabilities for specific problem domains.

2.1 Functional programming

While running an imperative program means to *execute commands*, running a functional program means to *evaluate expressions*.

Functions in a functional program are functions in a mathematical sense: the result of a function call depends only on the values of the arguments. Functions in imperative programming languages may have access to variables other than their arguments and the result of such a "function" may also depend on those variables. Moreover, the values of such variables may be changed after the function call, thus, the meaning of a function call is not solely determined by the result it returns. Because of such side effects, the meaning of an imperative program may be different depending on the order in which function calls are executed.

An important aspect of functional programs is that they do not have side effects and, hence, the result of evaluating an expression is determined only by the parts of the expression – not by evaluation order. As a consequence, functional programs can be evaluated with different evaluation strategies, like demand-driven evaluation. We discuss how demand-driven, so called lazy evaluation can increase the potential for abstraction in Section 2.1.2.

Beforehand, we discuss another concept found in functional languages that can increase the potential for abstraction: type polymorphism. It provides a mechanism for code reuse that is especially powerful in combination with higher-order functions: in a functional program functions can be arguments and results of other functions and can be manipulated just like data. We discuss these concepts in detail in Section 2.1.1.

Polymorphic typing can be combined with class-based overloading to define similar operations on different types. Overloading of type constructors rather than types is another powerful means for abstraction as we discuss in Section 2.1.3.

We can write purely functional programs in an imperative programming language by simply avoiding the use of side effects. The aspects sketched above, however, cannot be transferred as easily to imperative programming languages. In the remainder of this section we discuss each of these aspects in detail, focusing on the programmers potential to increase the level of abstraction.

2.1.1 Type polymorphism and higher-order functions

Imagine a function *length* that computes the length of a string. In Haskell strings are represented as lists of characters and we could define similar functions for computing the length of a list of numbers or the length of a list of Boolean values. The definition of such length functions is independent of the type of list elements. Instead of repeating the same definition for different types we can define the function *length* once with a type that leaves the type of list elements unspecified:

$$
\begin{aligned}
&length :: [a] \to Int \\
&length\ [\,] \quad\ = 0 \\
&length\ (\_ : l) = 1 + length\ l
\end{aligned}
$$

The type a used as argument to the list type constructor $[\,]$ represents an arbitrary type. There are infinitely many types for lists that we can pass to length, for example, $[Int]$, $String$, and $[[Bool]]$ are some of them.

Type polymorphism (Damas and Milner 1982) allows us to use type variables that represent arbitrary types, which helps to make defined functions more generally applicable. This is especially useful in combination with another feature of functional programming languages: higher-order functions. Functions in a functional program can not only map data to data but may also take functions as arguments or return them as result. In type signatures of higher-order functions, parentheses are used to group functional arguments. Probably the simplest example of a higher-order function is the infix operator \$ for function application:

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

At first sight, this operator seems dispensable, because we can always write $f\ x$ instead of $f\ \$\ x$. However, because of its low precendence, it is often useful to avoid parenthesis because we can write $f\ \$\ g\ \$\ h\ x$ instead of $f\ (g\ (h\ x))$. Another useful operator is function composition[2]:

$$
\begin{aligned}
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c) \\
&f \circ g = \lambda x \to f\ (g\ x)
\end{aligned}
$$

This definition uses a *lambda abstraction* that denotes an anonymous function. The operator for function composition is a function that takes two functions as arguments and yields a function as result. Lambda abstractions have

[2] The ASCII representaitons of all used mathematical symbols are given in Appendix A.1.

the form $\lambda x \to e$ where x is a variable and e is an arbitrary expression. The variable x is the argument and the expression e is the body of the anonymous function. The body may itself be a function and the notation $\lambda x\ y\ z \to e$ is short hand for $\lambda x \to \lambda y \to \lambda z \to e$. While the first of these lambda abstractions looks like a function with three arguments, the second looks like a function that yields a function that yields a function. In Haskell, there is no difference between the two. A function that takes many arguments *is* a function that takes one argument and yields a function that takes the remaining arguments. Representing functions like this is called *currying*.[3]

There are a number of predefined higher-order functions for list processing. In order to get a feeling for the abstraction facilities they provide, we discuss a few of them here.

The *map* function applies a given function to every element of a given list:

$$\begin{aligned} &map :: (a \to b) \to [a] \to [b] \\ &map\ f\ [\,] \quad\ = [\,] \\ &map\ f\ (x : xs) = f\ x : map\ f\ xs \end{aligned}$$

If the given list is empty, then the result is also the empty list. If it contains at least the element x in front of an arbitrary list xs of remaining elements, then the result of calling *map* is a non-empty list where the first element is computed using the given function f and the remaining elements are processed recursively. The type signature of *map* specifies that

- the argument type of the given function f and the element type of the given list and
- the result type of f and the element type of the result list

must be equal. For example, *map length* `["Haskell","Curry"]` is a valid application of *map* because the a in the type signature of *map* can be instantiated with *String* which is defined as $[Char]$ and matches the argument type $[a]$ of *length*. The type b is instantiated with *Int* and, therefore, the returned list has the type $[Int]$. The application *map length* $[7,5]$ would be rejected by the type checker because the argument type $[a]$ of *length* does not match the type *Int* of the elements of the given list.

The type signature is a partial documentation for the function *map* because we get an idea of what *map* does whithout looking at its implementation. If we do not provide the type signature, then *type inference* deduces it automatically from the implementation.

[3] The term *currying* is named after the American mathematician and logician *Haskell B. Curry*.

Another predefined function on lists is *dropWhile* that takes a predicate, which is a function with result type *Bool*, and a list and drops elements from the list as long as they satisfy the given predicate.

$$\begin{aligned}&\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [a] \to [a]\\&\mathit{dropWhile}\ p\ [\,] \quad\ = [\,]\\&\mathit{dropWhile}\ p\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ \mathit{dropWhile}\ p\ xs\ \textbf{else}\ x : xs\end{aligned}$$

The result of *dropWhile* is the longest suffix of the given list that is either empty or starts with an element that does not satisfy the given predicate. We can instantiate the type variable a in the signature of *dropWhile* with many different types. For example, the function *dropWhile isSpace* uses a predefined function $\mathit{isSpace} :: \mathit{Char} \to \mathit{Bool}$ to remove preceding spaces from a string, *dropWhile* (<10) removes a prefix of numbers that are less than 10 from a given list, and *dropWhile* $((<10) \circ \mathit{length})$ drops short lists from a given list of lists, for example, a list of strings. Both functions are defined as so called *partial application* of the function *dropWhile* to a single argument – a programming style made possible by currying.

Polymorphic higher-order functions allow to implement recurring idioms independently of concrete types and to reuse such an implementation on many different concrete types.

2.1.2 Lazy evaluation

With lazy evaluation (Wadsworth 1971) arguments of functions are only computed as much as necessary to compute the result of a function call. Parts of the arguments that are not needed to compute a result are not demanded and may contain divergent and/or expensive computations. For example, we can compute the length of a list without demanding the list elements. In a programming language with lazy evaluation, like Haskell, we can compute the result of the following call to the *length* function:

$$\mathit{length}\ [\bot, \mathit{fibonacci}\ 100]$$

Neither the diverging computation $\bot$ nor the possibly expensive computation *fibonacci* 100 are evaluated to compute the result 2.

This example demonstrates that lazy evaluation can be faster than eager evaluation because unnecessary computations are skipped. Lazy computations may also use less memory when functions are composed sequentially:

```
do contents ← readFile "in.txt"
   writeFile "out.txt" ∘ concat ∘ map addSpace $ contents
  where addSpace c | c == '.'  = ". "
                   | otherwise = [c]
```

This program uses Haskell's **do**-notation (see Section 2.1.3) to read the contents of a file `in.txt`, adds an additional space character after each period, and writes the result to the file `out.txt`. The function $concat :: [[a]] \to [a]$ concatenates a given list of lists into a single list[4]. In an eager language, the functions *map addSpace* and *concat* would both evaluate their arguments completely before returning any result. With lazy evaluation, these functions produce parts of their output from partially known input. As a consequence, the above program runs in constant space and can be applied to gigabytes of input. It does not store the complete file `in.txt` in memory at any time.

In a lazy language, we can build complex functions from simple parts that communicate via intermediate data structures without sacrificing memory efficiency. The simple parts may be reused to form other combinations which increases the modularity of our code.

Infinite data structures

With lazy evaluation we can not only handle large data efficiently, we can even handle unbounded, potentially infinite data (Hudak 1989). For example, we can compute an approximation of the square root of a number x as follows:

$$
\begin{aligned}
&sqrt :: Float \to Float \\
&sqrt\ x = head \circ dropWhile\ inaccurate \circ iterate\ next\ \$\ x \\
&\quad \textbf{where}\ next\ y \quad\ = (y + x\ /\ y)\ /\ 2 \\
&\qquad\quad\ \ inaccurate\ y = abs\ (x - y * y) > 0.00001
\end{aligned}
$$

With lazy evaluation we can split the task of generating an accurate approximation into two sub tasks:

1. generating an unbounded number of increasingly accurate approximations using Newton's formula and
2. selecting a sufficiently accurate one.

Approximations that are more accurate than the one we select are not computed by the function *sqrt*. In this example we use the function *iterate* to generate approximations and *dropWhile* to dismiss inaccurate ones. If we decide to use a different criterion for selecting an appropriate approximation, like the difference of subsequent approximations, then we only need to change the part that selects an approximation. The part of the algorithm that computes them can be reused without change. Again, lazy evaluation promotes modularity and code reuse.

[4] Definitions for library functions that are not defined in the text can be found in Appendix A.2.

In order to see another aspect of lazy evaluation we take a closer look at the definition of the function *iterate*:

$$\begin{aligned} &\mathit{iterate} :: (a \to a) \to a \to [a] \\ &\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

Conceptually, the call $\mathit{iterate}\ f\ x$ yields the infinite list

$$[x, f\ x, f\ (f\ x), f\ (f\ (f\ x)), \ldots]$$

The elements of this list are only computed if they are demanded by the surrounding computation because lazy evaluation is *non-strict*. Although it is duplicated in the right-hand side of *iterate*, the argument x is evaluated at most once because lazy evaluation is *sharing* the values that are bound to variables once they are computed. If we call $\mathit{iterate}\ \mathit{sqrt}\ (\mathit{fibonacci}\ 100)$, then the call $\mathit{fibonacci}\ 100$ is only evaluated once, although it is duplicated by the definition of *iterate*.

Sharing of sub computations ensures that lazy evaluation does not perform more steps than a corresponding eager evaluation because computations bound to duplicated variables are performed only once even if they are demanded after they are duplicated.

2.1.3 Class-based overloading

Using type polymorphism as described in Section 2.1.1, we can define functions that can be applied to values of many different types. This is often useful but sometimes insufficient. Polymorphic functions are agnostic about those values that are represented by type variables in the type signature of the function. For example, the *length* function behaves identically for every instantiation for the element type of the input list. It cannot treat specific element types different from others.

While this is a valuable information about the *length* function, we sometimes want to define a function that works for different types but can still take different instantiations of the polymorphic arguments into account. For example, it would be useful to have an equality test that works for many types. However, the type

$$(==) :: a \to a \to \mathit{Bool}$$

would be a too general type for an equality predicate ==. It requires that we can compare arbitrary types for equality, including functional types which might be difficult or undecidable.

Class-based overloading (Odersky et al. 1995; Wadler and Blott 1989) provides a mechanism to give functions like $==$ a reasonable type. We can define a *type class* that represents all types that support an equality predicate as follows:

$$\begin{array}{l}\textbf{class } \mathit{Eq}\ a \textbf{ where}\\ \quad (==) :: a \to a \to \mathit{Bool}\end{array}$$

This definition defines a type class Eq that can be seen as a predicate on types in the sense that the *class constraint* $Eq\ a$ implies that the type a supports the equality predicate $==$. After the above declaration, the function $==$ has the following type:

$$(==) :: \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}$$

and we can define other functions based on this predicate that inherit the class constraint:

$$\begin{array}{l}(\neq) :: \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}\\ x \neq y = \neg\ (x == y)\\ \mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool}\\ x \in [\,] \quad\ = \mathit{False}\\ x \in (y : ys) = x == y \vee x \in ys\end{array}$$

Here, the notation $x \in xs$ is syntactic sugar for $\mathit{elem}\ x\ xs$, $\neg$ denotes negation and $\vee$ disjunction on Boolean values.

In order to provide implementations of an equality check for them, we can *instantiate* the Eq class for specific types. For example, an Eq instance for Booleans can be defined as follows.

$$\begin{array}{l}\textbf{instance } \mathit{Eq}\ \mathit{Bool} \textbf{ where}\\ \quad \mathit{False} == \mathit{False} = \mathit{True}\\ \quad \mathit{True}\ == \mathit{True}\ = \mathit{True}\\ \quad \_\ \ \ == \_\ \ \ = \mathit{False}\end{array}$$

Even polymorphic types can be given an Eq instance, if appropriate instances are available for the polymorphic components. For example, lists can be compared if their elements can.

$$\begin{array}{l}\textbf{instance } \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ [a] \textbf{ where}\\ \quad [\,] \quad\ == [\,] \quad\ = \mathit{True}\\ \quad (x : xs) == (y : ys) = x == y \wedge xs == ys\\ \quad \_ \quad\ == \_ \quad\ = \mathit{False}\end{array}$$

Note the class constraint $Eq\ a$ in the instance declaration for $Eq\ [a]$. The first occurrence of == in the second rule of the definition of == for lists is the equality predicate for values of type a while the second occurrence is a recursive call to the equality predicate for lists.

Although programmers are free to provide whatever instance declarations they choose, type-class instances are often expected to satisfy certain laws. For example, every definition of == should be an equivalence relation—reflexive, symmetric and transitive—to aid reasoning about programs that use ==. More specifically, the following properties are usually associated with an equality predicate.

$$x == x$$
$$x == y \Rightarrow y == x$$
$$x == y \wedge y == z \Rightarrow x == z$$

Defining an Eq instance where == is no equivalence relation can result in highly unintuitive program behaviour. For example, the *elem* function defined above relies on reflexivity of ==. Using *elem* with a non-reflexive Eq instance is very likely to be confusing. The inclined reader may check that the definition of == for Booleans given above is an equivalence relation and that the Eq instance for lists also satisfies the corresponding laws if the instance for the list elements does.

Class-based overloading provides a mechanism to implement functions that can operate on different types differently. This allows to implement functions like *elem* that are not fully polymorphic but can still be applied to values of many different types. This increases the possibility of code reuse because functions with similar (but not identical) behaviour on different types can be implemented once and reused for every suitable type instead of being implemented again for every different type.

Overloading type constructors

An interesting variation on the ideas discussed in this section are so called *type constructor classes* (Jones 1993). In Haskell, polymorphic type variables can not only abstract from types but also from type constructors. In combination with class-based overloading, this provides a powerful mechanism for abstraction.

Reconsider the function $map :: (a \to b) \to [a] \to [b]$ defined in Section 2.1.1 which takes a polymorphic function and applies it to every element of a given list. Such functionality is not only useful for lists. A similar operation can be implemented for other data types too. In order to abstract

from the data type whose elements are modified, we can use a type variable to represent the corresponding type constructor.

In Haskell, type constructors that support a map operation are called *functors*. The corresponding type class abstracts over such type constructors and defines an operation *fmap* that is a generalised version of the *map* function for lists.

$$\begin{aligned}&\textbf{class } \mathit{Functor}\ f \textbf{ where}\\ &\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b\end{aligned}$$

Like the *Eq* class, the type class *Functor* has a set of associated laws that are usually expected to hold for definitions of *fmap*:

$$\begin{aligned}&\mathit{fmap}\ \mathit{id} &&\equiv \mathit{id}\\ &\mathit{fmap}\ (f \circ g) &&\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g\end{aligned}$$

Here, $\equiv$ denotes semantic equivalence and two functions are equivalent if they agree on all arguments.[5]. Let us check whether the following *Functor* instance for lists satisfies the functor laws.

$$\begin{aligned}&\textbf{instance } \mathit{Functor}\ [\,] \textbf{ where}\\ &\quad \mathit{fmap} = \mathit{map}\end{aligned}$$

We can prove the first law (for all finite lists[6]) by induction over the list structure. The base case considers the empty list:

$$\begin{aligned}&\quad \mathit{map}\ \mathit{id}\ [\,]\\ &\equiv \quad \{\ \text{definition of } \mathit{map}\ \}\\ &\quad [\,]\\ &\equiv \quad \{\ \text{definition of } \mathit{id}\ \}\\ &\quad \mathit{id}\ [\,]\end{aligned}$$

The induction step deals with an arbitrary non-empty list:

$$\begin{aligned}&\quad \mathit{map}\ \mathit{id}\ (x : xs)\\ &\equiv \quad \{\ \text{definition of } \mathit{map}\ \}\\ &\quad \mathit{id}\ x : \mathit{map}\ \mathit{id}\ xs\\ &\equiv \quad \{\ \text{definition of } \mathit{id}\ \}\\ &\quad x : \mathit{map}\ \mathit{id}\ xs\end{aligned}$$

[5] Technically, $\equiv$ can be defined as the smallest equivalence relation that contains the reduction relation of lambda calculus.

[6] During equational reasoning, we gloss over the details of partial or infinite data structures. Danielsson et al. (2006) show that such fast and loose reasoning is morally correct.

$$
\begin{aligned}
&\equiv \quad \{ \text{ induction hypothesis } \} \\
&\quad x : id\ xs \\
&\equiv \quad \{ \text{ definition of } id \text{ (twice) } \} \\
&\quad id\ (x : xs)
\end{aligned}
$$

We conclude $map\ id == id$, hence, the *Functor* instance for lists satisfies the first functor law. The second law can be verified similarly.

As an example for a different data type that also supports a map operation, consider the following definition of binary leaf trees[7].

$$\textbf{data}\ Tree\ a = Empty \mid Leaf\ a \mid Fork\ (Tree\ a)\ (Tree\ a)$$

A binary leaf tree is either empty, a leaf storing an arbitrary element, or an inner node with left and right sub trees. We can apply a polymorphic function to every element stored in a leaf using *fmap*:

$$
\begin{aligned}
&\textbf{instance}\ Functor\ Tree\ \textbf{where} \\
&\quad fmap\ \_\ Empty\ \ \ = Empty \\
&\quad fmap\ f\ (Leaf\ x)\ \ = Leaf\ (f\ x) \\
&\quad fmap\ f\ (Fork\ l\ r) = Fork\ (fmap\ f\ l)\ (fmap\ f\ r)
\end{aligned}
$$

The proof that this definition of *fmap* satisfies the functor laws is left as an exercise. More interesting is the observation that we can now define non-trivial functions that can be applied to both lists and trees. For example, the function $fmap\ (length \circ dropWhile\ isSpace)$ can be used to map a value of type $[String]$ to a value of type $[Int]$ and also to map a value of type $Tree\ String$ to a value of type $Tree\ Int$.

The type class *Functor* can not only be instantiated by polymorphic *data* types. The partially applied type constructor $\rightarrow$ for function types is also an instance of *Functor*:

$$
\begin{aligned}
&\textbf{instance}\ Functor\ ((\rightarrow)\ a)\ \textbf{where} \\
&\quad fmap = (\circ)
\end{aligned}
$$

For $f == (\rightarrow)\ a$ the function *fmap* has the following type.

$$fmap :: (b \rightarrow c) \rightarrow (\rightarrow)\ a\ b \rightarrow (\rightarrow)\ a\ c$$

If we rewrite this type using the more conventional infix notation for $\rightarrow$, we obtain the type $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$ which is exactly the type of the function composition operator $(\circ)$ defined in Section 2.1.1. It

[7] Binary leaf trees are binary trees that store values in their leaves.

is tempting to make use of this coincidence and define the above *Functor* instance without further ado. However, we should check the functor laws in order to gain confidence in this definition. The proofs can be found in Appendix B.1.

Type constructor classes provide powerful means to overload functions. This results in increased potential for code reuse – sometimes to a surprising extend. For example, we can implement an instance of the *Functor* type class for type constructors like $(a \rightarrow)$ where we would not expect such possibility at first sight. The following subsection presents another type class that can be instantiated for many different types leading to a variety of different usage scenarios with identical implementations.

Monads

Monads (Wadler 1990, 1995) are an important abstraction mechanism in Haskell – so important that Haskell provides special syntax to write monadic code. Besides syntax, however, monads are nothing special but instances of an ordinary type class.

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\ &\quad \mathit{return} :: a \rightarrow m\ a\\ &\quad (\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\end{aligned}$$

Like functors, monads are unary type constructors. The *return* function constructs a monadic value of type $m\ a$ from a non-monadic value of type a. The function $\gg\!=$, pronounced *bind*, takes a monadic value of type $m\ a$ and a function that maps the wrapped value to another monadic value of type $m\ b$. The result of applying $\gg\!=$ is a combined monadic value of type $m\ b$.

The first monad that programmers come across when learning Haskell is often the *IO* monad (Peyton-Jones and Wadler 1993) and in fact, a clear separation of pure computations without side effects and input/output operations was the main reason to add monads to Haskell. In Haskell, functions that interact with the outside world return their results in the *IO* monad, that is, their result type is wrapped in the type constructor *IO*. Such functions are often called *IO actions* to emphasise their imperative nature and distinguish them from pure functions. There are predefined *IO* actions *getChar* and *putChar* that read one character from standard input and write one to standard output respectively.

$$\begin{aligned}&\mathit{getChar} :: \mathit{IO}\ \mathit{Char}\\ &\mathit{putChar} :: \mathit{Char} \rightarrow \mathit{IO}\ ()\end{aligned}$$

The *IO* action *putChar* has no meaningful result but is only used for its side effect. Therefore, it returns the value $()$ which is the only value of type $()$.

We can use these simple *IO* actions to demonstrate how to write more complex monadic actions using the functions provided by the type class *Monad*. For example, we can use $\gg\!=$ to sequence the actions that read and write one character:

$$\begin{aligned}&\mathit{echoChar} :: \mathit{IO}\ ()\\&\mathit{echoChar} = \mathit{getChar} \gg\!= \lambda c \rightarrow \mathit{putChar}\ c\end{aligned}$$

This combined action will read one character from standard input and directly write it back to standard output, when it is executed. It can be written more conveniently using Haskell's **do**-notation as follows.

$$\begin{aligned}&\mathit{echoChar} :: \mathit{IO}\ ()\\&\mathit{echoChar} = \mathbf{do}\ c \leftarrow \mathit{getChar}\\&\qquad\qquad\ \ \mathit{putChar}\ c\end{aligned}$$

In general, $\mathbf{do}\ x \leftarrow a; f\ x$ is syntactic sugar for $a \gg\!= \lambda x \rightarrow f\ x$ and arbitrarily many nested calls to $\gg\!=$ can be chained like this in the lines of a **do**-block. The imperative flavour of the special syntax for monadic code highlights the historical importance of input/output for the development of monads in Haskell.

It turns out that monads can do much more than just sequence input/output operations. For example, we can define a *Monad* instance for lists and use **do**-notation to elegantly construct complex lists from simple ones.

$$\begin{aligned}&\mathbf{instance}\ \mathit{Monad}\ [\,]\ \mathbf{where}\\&\quad \mathit{return}\ x = [x]\\&\quad l \gg\!= f\ \ = \mathit{concat}\ (\mathit{map}\ f\ l)\end{aligned}$$

The *return* function for lists yields a singleton list and the $\gg\!=$ function applies the given function to every element of the given list and concatenates all lists in the resulting list of lists. We can employ this instance to compute a list of pairs from all elements in given lists.

$$\begin{aligned}&\mathit{pair} :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ (a,b)\\&\mathit{pair}\ \mathit{xs}\ \mathit{ys} = \mathbf{do}\ x \leftarrow \mathit{xs}\\&\qquad\qquad\qquad\ \ y \leftarrow \mathit{ys}\\&\qquad\qquad\qquad\ \ \mathit{return}\ (x,y)\end{aligned}$$

For example, the call $\mathit{pair}\ [0,1]\ [\mathit{True},\mathit{False}]$ yields a list of four pairs: $[(0,\mathit{True}),(0,\mathit{False}),(1,\mathit{True}),(1,\mathit{False})]$. We can write the function *pair*

without using the monad operations[8] but the definition with **do**-notation is arguably more readable.

The story does not end here. The data type for binary leaf trees also has a natural *Monad* instance:

```
instance Monad Tree where
  return  = Leaf
  t >>= f = mergeTrees (fmap f t)
```

This instance is similar to the *Monad* instance for lists. It uses *fmap* instead of *map* and relies on a function *mergeTrees* that computes a single tree from a tree of trees.

```
mergeTrees :: Tree (Tree a) → Tree a
mergeTrees Empty      = Empty
mergeTrees (Leaf t)   = t
mergeTrees (Fork l r) = Fork (mergeTrees l) (mergeTrees r)
```

Intuitively, this function takes a tree that stores other trees in its leaves and just removes the *Leaf* constructors of the outer tree structure. So, the $\gg=$ operation for trees replaces every leaf of a tree with the result of applying the given function to the stored value.

Now we benefit from our choice to provide such a general type signature for the function *pair*. We can apply the same function *pair* to trees instead of lists to compute a tree of pairs instead of a list of pairs. For example, the call *pair* (*Fork* (*Leaf* 0) (*Leaf* 1)) (*Fork* (*Leaf True*) (*Leaf False*)) yields the following tree with four pairs.

```
Fork (Fork (Leaf (0, True))
           (Leaf (0, False)))
     (Fork (Leaf (1, True))
           (Leaf (1, False)))
```

Like functors, monads allow programmers to define very general functions that they can use on a variety of different data types. Monads are more powerful than functors because the result of the $\gg=$ operation can have a different structure than the argument. When using *fmap* the structure of the result is always the same as the structure of the argument – at least if *fmap* satisfies the functor laws.

The *Monad* type class also has a set of associated laws. The *return* function must be a left- and right-identity for the $\gg=$ operator which needs to satisfy an associative law.

[8] $\lambda xs\ ys \to concat\ (map\ (\lambda x \to concat\ (map\ (\lambda y \to [(x,y)])\ ys))\ xs)$

$$
\begin{aligned}
&return\ x \gg\!= f \quad \equiv f\ x \\
&m \gg\!= return \quad \equiv m \\
&(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

These laws ensure a consistent semantics of the **do**-notation and allow equational reasoning about monadic programs. The verification of the monad laws for the list instance is left as an exercise for the reader. The proof for the *Tree* instance is in Appendix B.2.

2.1.4 Summary

Inspired by Hughes (1989), we have discussed different abstraction mechanisms of functional programming languages that help programmers to write more modular and reusable code. Type polymorphism (Section 2.1.1) allows to write functions that can be applied to a variety of different types because they ignore parts of their input. This feature is especially useful in combination with higher-order functions that allow to abstract from common programming patterns to define custom control structures like the *map* function on lists. Lazy evaluation (Section 2.1.2) increases the modularity of algorithms because demand driven evaluation often avoids storing intermediate results which allows to compute with infinite data. With class-based overloading (Section 2.1.3), programmers can implement one function that has different behaviour on different data types such that code using these functions can be applied in many different scenarios. We have seen two examples for type constructor classes, namely, functors and monads and started to explore the generality of the code they allow to write. Finally, we have seen that equational reasoning is a powerful tool to think about functional programs and their correctness.

2.2 Functional logic programming

Functional programming, discussed in the previous section, is one important branch in the field of declarative programming. Logic programming is another. Despite conceptual differences, research on combining these paradigms has shown that their conceptual divide is not as big as one might expect (Hanus 2007). The programming language Curry unifies lazy functional programming as in Haskell with essential features of logic programming. We use Curry to introduce logic programming features for two reasons:

1. its similarity to Haskell allows us to discuss new concepts using familiar syntax, and
2. the remainder of this thesis builds on features of both functional and logic programming.

Therefore, a multi-paradigm language is a natural and convenient choice. The main extensions of Curry compared to the pure functional language Haskell are

- logic variables,
- implicit nondeterminism, and
- built-in search.

In the remainder of this section we discuss logic variables (Section 2.2.1) and nondeterminism (Section 2.2.2) and show how they interact with features of functional programming discussed in Section 2.1. We will give a special account to lazy evaluation which allows to relate the concepts of logic variables and nondeterminism in an interesting way (Section 2.2.3) and which forms an intricate combination with implicit nondeterminism (Section 2.2.4). Built-in search (Section 2.2.5) allows to enumerate different results of nondeterministic computations and we discuss how programmers can influence the search order by implementing search strategies in Curry.

2.2.1 Logic variables

The most important syntactic extension of Curry compared to Haskell are declarations of logic variables. Instead of binding variables to expressions, Curry programmers can state that the value of a variable is unknown by declaring it **free**. A logic variable will be *bound* during execution according to demand: just like patterns in the left-hand side of functions cause unevaluated expressions to be evaluated, they cause unbound logic variables to be bound. Such instantiation is called *narrowing* (Antoy et al. 2000; Slagle 1974) because the set of values that the variable denotes is narrowed to a smaller set containing only values that match the pattern.

Narrowing according to patterns is not the only way how logic variables can be bound in Curry. We can also use *constraints* to constrain the set of their possible instantiations. We discuss constraint programming in Section 2.2.6 but Curry provides a specific kind of constraints that is worth mentioning here: term-equality constraints. The built-in function $\doteq :: a \to a \to \mathit{Success}$ constrains two data terms, which are allowed to contain logic

variables, to be equal.[9] The type *Success* of constraints is similar to the unit type $()$. There is only one value *success* of type *Success* but we cannot pattern match on this value. If the arguments of $\doteq$ cannot be instantiated to equal terms, the corresponding call *fails*, that is, does not yield a result. We can use constraints—which are values of type *Success*—in guards of functions to specify conditions on logic variables.

We demonstrate both narrowing and equality constraints by means of a simple example. The function *last* which computes the last element of a non-empty list can be defined in Curry as follows.

$$\begin{aligned} &last :: [a] \to a \\ &last\ l \mid xs +\!\!+ [x] \doteq l \\ &\qquad = x \\ &\quad \textbf{where}\ x, xs\ \textbf{free} \end{aligned}$$

Instead of having to write a recursive definition explicitly, we use the *property* that *last* l equals x iff there is a list xs such that $xs +\!\!+ [x]$ equals l. The possibility to use predicates that involve previously defined operations to define new ones improves the possibility of code reuse in functional logic programs. Logic variables are considered existentially quantified and the evaluation mechanism of Curry includes a search for possible instantiations. During the evaluation of a call to *last*, the logic variable xs is *narrowed* by the function $+\!\!+$ to a list that is one element shorter than the list l given to *last*. The result of $+\!\!+$ is a list of logic variables that matches the length of l and whose elements are *constrained* to equal the elements of l. As a consequence, the variable x is bound to the last element of l and then returned by the function *last*.

2.2.2 Nondeterminism

The built-in search for instantiations of logic variables can lead to different possible instantiations and, hence, nondeterministic results of computations (Rabin and Scott 1959). Consider, for example, the following definition of *insert*:

$$\begin{aligned} &insert :: a \to [a] \to [a] \\ &insert\ x\ l \mid xs +\!\!+ ys \doteq l \end{aligned}$$

[9] As Curry does not support type classes, the type of $\doteq$ is too polymorphic because it does not ensure on the type level that its arguments can be constrained to be equal. There is an experimental implementation of the Münster Curry Compiler with type classes which, however, (at the time of this writing) also does not restrict the type of $\doteq$.

$$= xs +\!\!+ x : ys$$
where xs, ys **free**

If the argument l of *insert* is non-empty then there are different possible bindings for xs and ys such that $xs +\!\!+ ys \doteq l$. Consequently, the result of *insert* may contain x at different positions and, thus, there is more than one possible result when applying *insert* to a non-empty list. Mathematically, *insert* does not denote a *function* that maps arguments to deterministic results but a *relation* that specifies a correspondence of arguments to possibly nondeterministic results. To avoid the contradictory term *nondeterministic function* we call *insert* (and other defined operations that may have more than one result) *nondeterministic operation*.

Traditionally, Curry systems use backtracking to enumerate all possible results of nondeterministic operations. For example, if we execute *insert* $1\ [2,3]$ in a Curry system, we can query one solution after the other interactively.

```
> insert 1 [2,3]
[1,2,3]
More solutions? [Y(es)/n(o)/a(ll)] yes
[2,1,3]
More solutions? [Y(es)/n(o)/a(ll)] all
[2,3,1]
```

If we are not interested in all results of a computation, we can just answer `no` to the interactive query, which is especially useful for computations with infinitely many results.

Variable instantiations are not the only source of nondeterminism in Curry programs. As the run-time system needs to handle nondeterminism anyway, Curry also provides a direct way to define nondeterministic operations. Unlike in Haskell, the meaning of defined Curry operations does not depend on the order of their defining rules. While in Haskell the rules of a function are tried from top to bottom committing to the first matching rule, in Curry the rules of an operation are tried nondeterministically. As a consequence, overlapping rules lead to possibly nondeterministic results.

We can use overlapping rules to give an alternative implementation of the *insert* operation.

$$\begin{aligned} &insert :: a \to [a] \to [a] \\ &insert\ x\ l \qquad = x : l \\ &insert\ x\ (y : ys) = y : insert\ x\ ys \end{aligned}$$

This definition *either* inserts the given element x in front of the given list l *or*—if l is non-empty—inserts x in the tail ys of l, leaving the head y in its

original position. This version of *insert* is more lazy than the version that uses a guard shown previously: while the equality constraint in the guard forces the evaluation of both arguments of *insert*, the version with overlapping rules can yield a result without evaluating any of the arguments. In the following we use the second definition of *insert* to benefit from its laziness.

The advantage of implicit nondeterminism (as opposed to explicitly using, for example, lists to represent multiple results) is that the source code does not contain additional combinators to handle nondeterminism which eases the composition of more complex nondeterministic operations from simpler ones. For example, we can compute permutations of a given list nondeterministically by recursively inserting all its elements into an empty list.

```
permute :: [a] → [a]
permute []     = []
permute (x:xs) = insert x (permute xs)
```

With an explicit representation of nondeterminism, we would need to use a separate data structure that models nondeterministic results or use monadic syntax to hide the actual structure used (cf. the definition of the *pair* function in Subsection 2.1.3). Implicit nondeterminism helps to focus on algorithmic details because no book-keeping syntax interferes with a nondeterministic algorithm.

2.2.3 Lazy nondeterminism

In the previous subsection we have seen that the instantiation of logic variables can lead to nondeterministic computations. We can stress this observation and define an operation (?) for nondeterministic choice based on logic variables:

```
(?) :: a → a → a
x ? y = ifThenElse b x y
  where b free

ifThenElse :: Bool → a → a → a
ifThenElse True  x _ = x
ifThenElse False _ x = x
```

It is functional logic folklore, shown by Antoy (2001), that every Curry program with overlapping rules can be translated into an equivalent Curry program without overlapping rules by using (?) for nondeterministic choice. Therefore, one could drop support for direct nondeterminism via overlapping rules without restricting the class of programs that can be written.

A more recent discovery by Antoy and Hanus (2006) is that one can do the opposite too: overlapping rules suffice to model narrowing of logic variables. This observation essentially relies on laziness and the remainder of this subsection explains the details of combining laziness and nondeterminism to model narrowing.

Suppose our language would support nondeterminism via overlapping rules but no declarations of logic variables. We could redefine the operation $(?)$ using overlapping rules instead of a logic variable.

$$
\begin{aligned}
&(?) :: a \to a \to a \\
&x \,?\, \_ = x \\
&\_ \,?\, x = x
\end{aligned}
$$

As there is no pattern matching, both rules are trivially overlapping and, therefore, executed nondeterministically when $(?)$ is called. If the first rule is chosen, $(?)$ returns the first argument; if the second rule is chosen, it returns the second argument.

Narrowing a logic variable means to bind it to a pattern when it is demanded by the evaluation of a defined operation. We can model this process of binding a variable by using a nondeterministic operation that can be evaluated to every possible binding of the variable. If such an operation—we call it nondeterministic generator—is matched with a pattern, a matching binding will be chosen nondeterministically. A logic variable is represented by an unevaluated generator and evaluating a generator to a constructor corresponds to the process of binding the represented variable to this constructor.

A logic variable of type *Bool* can be represented by the nondeterministic generator *bool*.

$$
\begin{aligned}
&bool :: Bool \\
&bool = True \,?\, False
\end{aligned}
$$

Every definition that uses free variables of type *Bool* can use *bool* instead. For example, the call $\neg\ b$ **where** b **free** *narrows* the variable b to *True* or *False* nondeterministically and yields either *False* or *True*. Similarly, the call $\neg\ b$ **where** $b = bool$ *evaluates* b to *True* or *False* nondeterministically and yields either *False* or *True*.

This idea generalises to recursive data, where we see the importance of laziness. For example, we can define an operation *blist* that represents logic variables of type $[Bool]$.

$$
\begin{aligned}
&blist :: [Bool] \\
&blist = [\,] \,?\, (bool : blist)
\end{aligned}
$$

The *blist* generator can evaluate to every possible list of Booleans nondeterministically. Without laziness its evaluation would not terminate, because there are infinitely many such lists. With lazy evaluation, however, it is only evaluated as much as demanded – just like a logic variable of the same type is only narrowed as much as demanded.

If we apply the *head* function to the *blist* generator, we obtain two nondeterministic results, namely, *True* or *False*, and the tail of *blist* is not evaluated – just like the tail of a logic variable of type $[Bool]$ is not bound by the *head* function. Besides pointing out the similarity of lazy evaluation and narrowing of logic variables, this example also demonstrates a difference: logic variables can be results of computations in Curry. In fact, the result of applying the *head* function to a logic variable of type $[Bool]$ is a logic variable of type *Bool*. Unevaluated generators that are part of the result of a computation are evaluated by the eval-print loop of a Curry system whereas logic variables can be shown to the user without instantiating them with all possible bindings.

Another difference of logic variables to nondeterministic generators is that the latter cannot be constrained deterministically using constraint programming. For example, the constraint $x \doteq x$ **where** x **free** can be solved deterministically but $x \doteq x$ **where** $x = bool$ is solved nondeterministically with two successful derivations. The call $x \doteq x$ **where** $x = blist$ even describes infinitely many nondeterministic successful derivations.

2.2.4 Call-time choice

When comparing logic variables and nondeterministic generators, we have made an implicit assumption on the meaning of variable bindings. To illustrate this assumption, consider the call $\neg\ x == x$ **where** x **free**. What are the possible results of this call? From a mathematical point of view, $\neg\ x == x$ should clearly be *False* regardless of the instantiation of x. And indeed, the intuition behind logic variables is that they denote unknown values as pointed out in Section 2.2.1. There is no possible instantiation of x to a value such that the call $\neg\ x == x$ yields *True*.

For the modeling of narrowing via generators to be correct, it is required that $\neg\ x == x$ **where** $x = bool$ also cannot yield *True*. But now, x is not a logic variable but bound to an expression that can evaluate to *True* or *False* nondeterministically. While $bool == bool$ can evaluate to *True* or *False* nondeterministically, because the two occurrences of *bool* denote independent generators, binding the result of a single generator to a variable causes this variable to denote the same *value* wherever it occurs.

This behaviour is called call-time choice (Hennessy and Ashcroft 1977) and corresponds to an eager evaluation of variable bindings. If we first evaluate the binding of x to *True* or *False* nondeterministically and then evaluate either $\neg\ True == True$ or $\neg\ False == False$, we obtain the result *False* in every nondeterministic branch of the computation. In Curry, variable bindings are evaluated on demand but still the computed results are as if they were evaluated eagerly. This behaviour corresponds to *sharing* of variable bindings in Haskell (cf. Section 2.1.2). If a duplicated variable is bound to an expression, the expression is evaluated at most once and also corresponding nondeterministic choices are performed at most once.

Due to call-time choice, not only logic variables denote values but every variable—even if duplicated and bound to a nondeterministic expression—denotes one deterministic value in each nondeterministic branch of the computation.

This property of variables allows to elegantly express search problems using Curry. Because of lazy evaluation with call-time choice, Curry programmers can express search problems in the intuitive *generate-and-test* style whereas they are solved using the more efficient test-of-generate pattern. In the generate-and-test approach, programmers implement a nondeterministic generator for candidate solutions and a test predicate to filter admissible results independently. Due to lazy evaluation, however, candidate solutions are not computed in advance but only as demanded by the test predicate which can lead to a sophisticated interleaving of candidate generation and testing. Being able to compose search programs in this way without loosing the efficiency of an intricate interleaved approach is a major benefit of laziness. In later chapters, we will use this approach to generate test cases on demand but let us first discuss it using a simpler example.

The n-queens problem poses the question how to place n queens on an $n \times n$ chessboard such that no queen can capture another. We can solve this problem elegantly by generating placements of queens nondeterministically and check whether all placed queens are safe. In order to choose a suitable representation of a placement of queens, we observe that it is immediately clear that no two queens are allowed to be in the same row or in the same column of the chess board. Hence, we can represent a placement as permutation of $[1\,..\,n]$ where the number q_i at position i of the permutation denotes that a queen should be placed at row q_i in column i. With such a representation we only need to check whether queens can capture each other diagonally. Two queens are on the same diagonal iff

$$\exists\, 1 \leqslant i < j \leqslant n : j - i = |q_j - q_i|$$

We can solve the n-queens problem in Curry by using this formula to check if placements of queens are safe. We generate candidate placements by reusing the *permute* operation defined in Section 2.2.2 to compute an arbitrary permutation of $[1\,..\,n]$ and return it if all represented queens are placed safely. Clearly, all occurrences of *qs* in the definition of *queens* must denote the same placement of queens. Therefore, this definition is only sensible because of call-time choice semantics.

$$
\begin{aligned}
&\mathit{queens} :: \mathit{Int} \to [\mathit{Int}]\\
&\mathit{queens}\ n \mid \mathit{safe}\ \mathit{qs} = \mathit{qs}\\
&\quad \textbf{where}\ \mathit{qs} = \mathit{permute}\ [1\,..\,n]
\end{aligned}
$$

The predicate *safe* now checks the negation of the above formula: the difference of all columns $i < j$ must not equal the absolute difference of the corresponding rows. We use the function $\mathit{zip} :: [a] \to [b] \to [(a,b)]$ to pair every queen—represented by the row in which it should be placed—with its corresponding column.

$$
\begin{aligned}
&\mathit{safe} :: [\mathit{Int}] \to \mathit{Bool}\\
&\mathit{safe}\ \mathit{qs} = \mathit{and}\ [j - i \neq \mathit{abs}\ (q_j - q_i) \mid (i, q_i) \leftarrow \mathit{iqs}, (j, q_j) \leftarrow \mathit{iqs}, i < j]\\
&\quad \textbf{where}\ \mathit{iqs} = \mathit{zip}\ [1\,..\,]\ \mathit{qs}
\end{aligned}
$$

The function $\mathit{and} :: [\mathit{Bool}] \to \mathit{Bool}$ implements conjunction on lists and we use a *list comprehension* to describe the list of all conditions that need to be checked. List comprehensions are similar to monadic do-notation (see Section 2.1.3) and simplify the construction of complex lists.

If we execute *queens* 8 in a Curry interpreter, it prints a solution—depicted graphically in Figure 2.1—almost instantly:

```
> queens 8
[8,4,1,3,6,2,7,5]
More solutions? [Y(es)/n(o)/a(ll)] no
```

The call *queens* 10 has also acceptable run time but *queens* 15 does not finish within 60 seconds. We will see how to improve the efficiency of this solution (without sacrificing its concise specification) using constraint programming (see Section 2.2.6) but first we investigate the given generate-and-test program.

Although the algorithm reads as if it would generate every permutation of $[1\,..\,n]$, lazy evaluation helps to save at least some of the necessary work to compute them. To see why, we observe that the *safe* predicate is lazy – it does not necessarily demand the whole placement to decide whether it is valid. We can apply *safe* to (some) partial lists and still detect that they

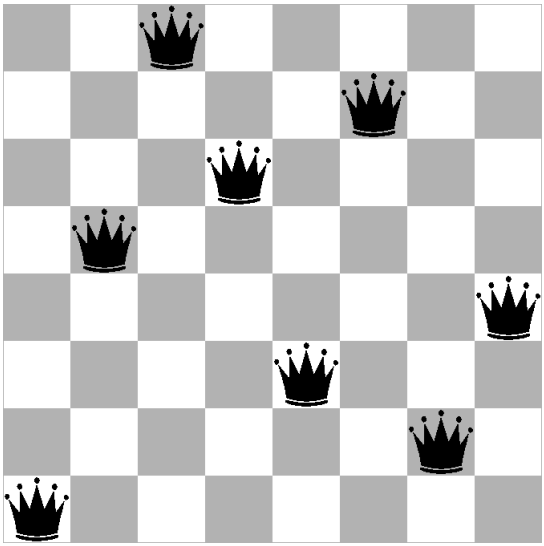


Figure 2.1: Safe placement of 8 queens on an 8×8 chessboard

represent invalid placements. For example, the call *safe* $(1 : 2 : \bot)$ yields *False*. As a consequence, all permutations of $[1 \,..\, n]$ that start with 1 and 2 are rejected simultaneously and not computed any further which saves the work to generate and test $(n - 2)!$ invalid placements.

In order to make this possible, the *permute* operation needs to produce permutations lazily. Thoroughly investigating its source code shown in Section 2.2.2 can convince us that it does generate permutations lazily because the recursive call to *insert* is underneath a $(:)$ constructor and only executed on demand. But we can also check the laziness of *permute* experimentally. If we demand a complete permutation of the list $[1, 2, 3]$ by calling the *length* function on it, we can observe the result 3 six times, once for each permutation. If we use the *head* function, which does not demand the computation of the tail of a permutation, instead of *length*, we can observe that permutations are not computed completely: we only obtain three results, namely, 1, 2, and 3, without duplicates. As a consequence, laziness helps to prune the search space when solving the n-queens problem because the *permute* operation generates placements lazily and the *safe* predicate checks them lazily.

Depending on the exact laziness properties of the generate and test operations, laziness often helps to prune significant parts of the search space

when solving generate-and-test problems. This has sometimes noticeable consequences for the efficiency of a search algorithm although it usually does not improve its theoretical complexity.

2.2.5 Search

Implicit nondeterminism is convenient to implement nondeterministic algorithms that do not care about the nondeterministic choices they make, that is, if programmers are indifferent about which specific solution is computed. But what if we want to answer the following questions by reusing the operations defined previously:

- How many permutations of $[1\,..\,n]$ are there?
- Is there a safe placement of 3 queens on a 3×3 chessboard?

With the tools presented up to now, we cannot compute all solutions of a nondeterministic operation inside a Curry program or even determine if there are any. Without further language support we would need to resort to model nondeterminism explicitly like in a purely functional language, for example, by computing lists of results.

Therefore Curry supports a primitive operation $getAllValues :: a \to IO\ [a]$ that converts a possibly nondeterministic computation into a list of all its results (Braßel et al. 2004; Hanus and Steiner 1998). The list is returned in the *IO* monad (see Section 2.1.3) because the order of the elements in the computed list is unspecified and may depend on external information, like which compiler optimisations are enabled.

We can use *getAllValues* to answer both of the above questions using Curry programs. The following program prints the number 6 as there are six permutations of $[1\,..\,3]$.

```
do ps ← getAllValues (permute [1..3])
   print (length ps)
```

In order to verify that there is no placement of 3 queens on a 3×3 chessboard we can use the following code which prints *True*.

```
do qs ← getAllValues (queens 3)
   print (null qs)
```

The function *getAllValues* uses the default backtracking mechanism to enumerate results. Backtracking corresponds to depth-first search and can be

trapped in infinite branches of the search space. For example, the following program will print $[[\,],[True],[True,True],[True,True,True]]$.

```
do ls ← getAllValues blist
   print (take 4 ls)
```

Indeed, backtracking will never find a list that contains *False* when searching for results of the *blist* generator (see Section 2.2.3).

To overcome this limitation, some Curry implementations provide another search function $getSearchTree :: a \to IO\ (SearchTree\ a)$ which is similar to *getAllValues* but returns a tree representation of the search space instead of a list of results. The search space is modeled as value of type *SearchTree a* which can be defined as follows[10].

```
data SearchTree a = Value a | Choice [SearchTree a]
```

Value a denotes a single result and *Choice ts* represents a nondeterministic choice between the solutions in the sub trees *ts*[11]. The call *getSearchTree bool*, for example, returns the search tree *Choice* $[Value\ True, Value\ False]$.

As *getSearchTree* returns the search tree lazily, we can define Curry functions to traverse a value of type *SearchTree a* to guide the search and steer the computation. If we want to use breadth-first search to compute a fair enumeration of an infinite search space instead of backtracking, we can use the following traversal function.

```
bfs :: SearchTree a → [a]
bfs t = [x | Value x ← queue]
  where queue = t : runBFS 1 queue
runBFS :: Int → [SearchTree a] → [SearchTree a]
runBFS n ts
  | n == 0 = []
  | n > 0 = case ts of
              Choice us : vs → us ++ runBFS (n - 1 + length us) vs
              _ : vs         → runBFS (n - 1) vs
```

The *bfs* function produces a lazy queue[12] containing all nodes of the search tree in level order and selects the *Value* nodes from this list. We can use it to compute a fair enumeration of the results of *blist*. The following program prints the list $[[\,],[True],[False],[True,True],[True,False]]$.

[10] The definition of the *SearchTree* type varies among different Curry implementations.
[11] Failure, that is, a computation without results, can be represented as *Choice* $[\,]$.
[12] which is terminated by itself to enqueue new elements at the end

```
do t ← getSearchTree blist
   print (take 5 (bfs t))
```

Every list of Booleans will be eventually enumerated when searching for results of *blist* using breadth-first search.

Curry's built-in search provides means to reify the results of implicitly non-deterministic computations and process them as a whole. With access to a lazy tree representation of the search space, programmers can steer the computation towards parts of the search space that are explored by their traversal function. Especially, they can implement fair strategies like breadth-first search.

2.2.6 Constraints

A key feature of logic programming is the ability to compute with unbound logic variables that are bound according to conditions. In Subsection 2.2.1 we have seen two ways to bind variables, namely, *narrowing* according to patterns and term-equality *constraints*. Narrowing can refine the possible values of a logic variable incrementally while an equality constraint determines a unique binding.

For specific types, the idea of restricting the possible values of a variable using constraints can be generalised to arbitrary domain specific predicates (Jaffar and Lassez 1987). In order to solve such domain specific constraints, sophisticated solver implementations can be supplied transparently, that is, invisible to programmers. For example, an efficient solver for the Boolean satisfiability problem could be incorporated into Curry together with a type to represent Boolean formulas where unknown Boolean variables are just represented as logic variables of this type. Or complex algorithms to solve non-linear equations over real numbers could be integrated to support logic variables in arithmetic computations.

Finite-domain constraints express equations and inequations over natural numbers with a bounded domain. They can be used to solve many kinds of combinatorial problems efficiently. The key to efficiency is to restrict the size of the search space by incorporating as much as possible information about logic variables deterministically before instantiating them to their remaining possible values.

As an example for constraint programming, we improve the n-queens solver shown in Section 2.2.4 using finite-domain constraints. Instead of generating a huge search space by computing placements of queens and checking them afterwards, we define a placement as list of logic variables,

constrain them such that all placed queens are safe, and search for possible solutions afterwards.

The following *queens* operation implements this idea in Curry.

$$
\begin{aligned}
&\mathit{queens} :: \mathit{Int} \to [\mathit{Int}] \\
&\mathit{queens}\ n \mid \mathit{domain}\ \mathit{qs}\ 1\ n\ \&\ \mathit{all\_different}\ \mathit{qs}\ \&\ \mathit{safe}\ \mathit{qs} \\
&\qquad\quad \&\ \mathit{labeling}\ [\mathit{FirstFailConstrained}]\ \mathit{qs} \\
&\qquad\quad = \mathit{qs} \\
&\quad \textbf{where}\ \mathit{qs} = [\mathit{unknown} \mid \_ \leftarrow [1\,..\,n]]
\end{aligned}
$$

This definition uses the predefined operation *unknown*—which yields a logic variable—to build a list of logic variables of length n. The function $(\&)$ denotes constraint conjunction and is used to specify

- that the variables should have values between 1 and n using the constraint *domain qs* 1 n,
- that all variables should have different values using a predefined constraint *all_different*, and
- that all queens should be placed safely.

Finally, the *labeling* constraint instantiates the variables with their possible values nondeterministically. The list given as first argument to *labeling* specifies a strategy that determines the order in which variables are bound. The strategy *FirstFailConstrained* specifies that a variable with the least number of possible values and the most attached constraints should be instantiated first to detect possible failures early.

The predicate *safe* that checks whether queens are placed safely is similar to the version presented in Section 2.2.4.

$$
\begin{aligned}
&\mathit{safe} :: [\mathit{Int}] \to \mathit{Success} \\
&\mathit{safe}\ \mathit{qs} = \mathit{andC}\ [\ (j-i) \not\#\ (q_j -\#\ q_i)\ \&\ (j-i) \not\#\ (q_i -\#\ q_j) \\
&\qquad\qquad\qquad \mid (i, q_i) \leftarrow \mathit{iqs}, (j, q_j) \leftarrow \mathit{iqs}, i < j] \\
&\quad \textbf{where}\ \mathit{iqs} = \mathit{zip}\ [1\,..]\ \mathit{qs}
\end{aligned}
$$

We use *Success* as result type because the condition is specified as a finite-domain constraint. The function *andC* implements conjunction on lists of constraints. As there is no predefined function to compute the absolute value of finite-domain variables, we express the condition as two dis-equalities. The operations with an attached # work on finite-domain variables and otherwise resemble their counterparts.

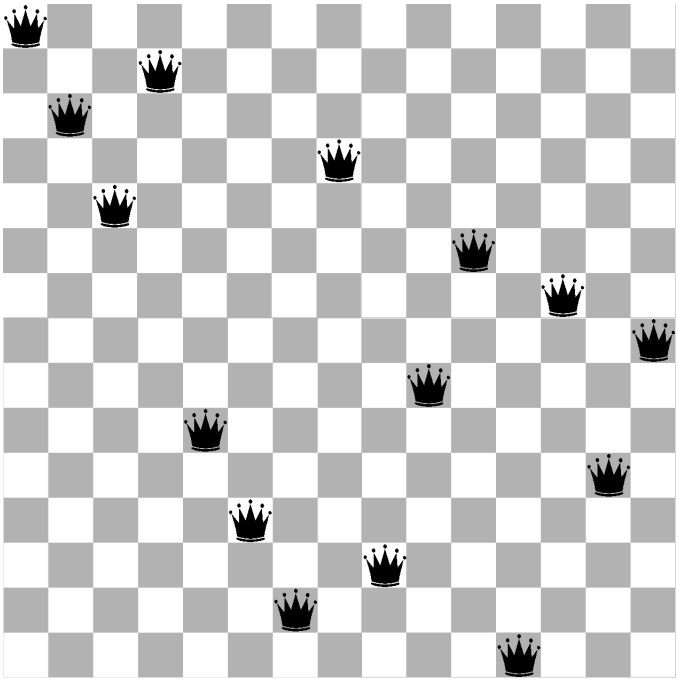


Figure 2.2: Safe placement of 15 queens on an 15×15 chessboard

This definition of *queens* is only slightly more verbose than the one given previously. It uses essentially the same condition to specify which placements of queens are safe. The algorithm that uses this specification to compute safe placements of queens is hidden in the run-time system.

Unlike the generate-and-test implementation, the constraint-based implementation of the n-queens solver yields a solution of the 15-queens problem instantly. Evaluating the call *queens* 15 in the Curry system PAKCS yields the placement $[1,3,5,2,10,12,14,4,13,9,6,15,7,11,8]$ which is depicted graphically in Figure 2.2.

Constraint programming increases the performance of search algorithms significantly, at least in specific problem domains. It is especially useful for arithmetic computations with unknown information because arithmetic operations usually do not support narrowing of logic variables.

2.2.7 Summary

Logic programming complements functional programming in the declarative programming field. It provides programmers with the ability to compute with unknown information represented as logic variables. Such variables are bound during execution by narrowing them according to patterns or by term-equality or other domain specific constraints.

Following Antoy and Hanus (2002), we have emphasised how a combination of functional and logic programming increases the expressive power of programmers. For example, defining operations by conditions on unknown data increases the amount of code reuse in software. Combining features like polymorphic functions from functional languages with logic features like logic variables and nondeterminism allows for a very concise definition of search algorithms.

In this section we have discussed the essential features of logic programming in the context of the multi-paradigm declarative programming language Curry. Like Haskell, Curry uses lazy evaluation and we have discussed the intricacies of combining laziness with nondeterminism that lead to the standard *call-time choice* semantics of functional logic programs.

2.3 Chapter notes

This chapter has been published previously on the authors website for public review (Fischer 2009a). Anonymous comments on Reddit (2009) have helped to improve its presentation.

3 Generating Tests

Software testing is an invaluable means to ensure correctness of code. Even in declarative programs that are amenable to proofs via equational reasoning, thorough testing complements the effort of correctness proofs and helps to identify bugs early. Tools that check a user-defined property for a large number of automatically generated tests often expose bugs quicker than expensive proofs of preliminary code. Moreover, an extensive test suite greatly supports the confidence in refactored code which can be checked automatically using test functions of preliminary code with little effort. In contrast, rewritten code may need a completely new correctness proof that considers implementation details.

Tests can be generated with or without taking implementation details into account. Testing a program only with respect to its interface by generating arbitrary input values of appropriate types is called *black-box testing* because the internals of the tested program remain invisible to the process of test-case generation. Generating test cases for executing specific parts of the tested program is called *glass-box testing*[1] because the implementation of the tested code is visible to—and considered by—the process of test-case generation.

In this chapter we show the benefits of using functional logic programming for generating test cases automatically. The built-in notions of nondeterminism and search support an elegant specification of test-case generators (Section 3.1) and lazy evaluation (see Sections 2.1.2 and 2.2.3) simplifies the demand-driven generation of test cases which avoids generating redundant tests (Section 3.2).

3.1 Black-box testing

In this section we highlight the advantages of functional logic programming in the implementation and use of a library for specification-based black-box testing. Implicit nondeterminism lets programmers define test-case generators elegantly, and an explicit search-tree representation of nondeterministic computations (see Section 2.2.5) lets us enumerate tests according to user-defined generators flexibly.

[1] Glass-box testing is often called *white-box testing* in contrast to black-box testing. We prefer the term glass-box testing because one usually cannot see inside a white box.

We provide an interface for automated, property-based testing of Curry programs (Section 3.1.1), show how to define and use custom nondeterministic test-case generators (Section 3.1.2), present different search strategies to enumerate tests (Section 3.1.3), and compare the distribution of test cases that they produce experimentally (Section 3.1.4). Apart from standard search strategies like depth- and breadth-first search, we define desirable properties of enumeration strategies for test cases and use them to develop and investigate the search strategy randomised level diagonalisation. Finally, we describe the Curry implementation of our library for specification-based black-box testing of Curry programs (Section 3.1.5).

3.1.1 Property-based testing

Automated test tools generate a large number of tests. Having to check all of them manually would probably preclude programmers from testing often. Testing often and quickly, however, is a key ingredient for *test-driven development* where programmers write tests manually *before* writing code that passes them. Test-driven development leads to thoroughly tested programs because no code is written before writing a test that justifies it.

Automated test tools like the one presented in this section advocate *property-driven development* where programmers do not write tests but properties manually. A property is a predicate on test input that can be applied to large amounts of generated test data automatically. With properties, programmers can specify test failure on a high level without having to write individual tests. Like test-driven development, property-driven development leads to extensive test suites that increase the confidence in refactorings. Moreover, writing down properties of algorithms before implementing them can influence the design of their implementation beneficially.

Properties of deterministic functions

As an example for property-based testing, we specify a property that the *dropWhile* function (see Section 2.1.1) must satisfy. The result of the call *dropWhile p xs* should be the longest suffix of *xs* that is empty or starts with an element x such that $p\ x == False$. The following function specifies part of this property.

$$
\begin{aligned}
&dropWhileYieldsValidSuffix :: [Int] \to Bool \\
&dropWhileYieldsValidSuffix\ xs = null\ suffix \vee \neg\ (p\ (head\ suffix)) \\
&\quad \textbf{where}\ suffix = dropWhile\ p\ xs \\
&\qquad\quad p\ x \quad = x == 0
\end{aligned}
$$

This property passes a predicate p that checks if its argument equals 0 to *dropWhile*. It then uses the predefined functions *null* and *head* to check whether the result of applying *dropWhile p* to an arbitrary list of integers is empty or starts with an element that is non-zero. The property does not check that the suffix is as long as possible but we could check this using another property.

We can use the function *blackCheck* to check whether *dropWhile* yields a valid suffix.

```
> blackCheck dropWhileYieldsValidSuffix
100 tests passed.
```

This call generates 100 lists of integers and passes one after the other to the property *dropWhileYieldsValidSuffix*. The *blackCheck* function is overloaded and generates test inputs according to the type of the given property. We discuss its implementation in Section 3.1.5.

This example demonstrates how property-based testing saves us from defining a sufficient number of meaningful tests manually. If 100 different lists do not expose an error, then we can be more or less confident that the implementation satisfies the tested property (depending on the distribution of test cases; see Section 3.1.3).

Properties do not necessarily have the result type *Bool*. There are also library functions for defining more sophisticated properties of type *Property*. We discuss a few of them here and a few more later, when we encounter them in examples.

The implication operator $\Longrightarrow$ is often used to reject test input that is invalid or insignificant. For example, we can refine *dropWhileYieldsValidSuffix* using $\Longrightarrow$ as follows.

$$\begin{aligned}&\mathit{dropWhileYieldsValidSuffix2} :: [\mathit{Int}] \to \mathit{Property}\\&\mathit{dropWhileYieldsValidSuffix2}\ \mathit{xs} =\\&\quad (\mathit{length}\ \mathit{xs} \geqslant 2) \Longrightarrow \mathit{dropWhileYieldsValidSuffix}\ \mathit{xs}\end{aligned}$$

When using *blackCheck* to check this property, 100 lists whose length is at least 2 are checked.

Properties need to be decisive about whether they are satisfied or fail and, hence, must not evaluate to different results nondeterministically. There are, however, combinators to specify properties of nondeterministic operations.

Properties of nondeterministic operations

Functional logic programmers may also want to specify properties of nondeterministic operations like *insert* introduced in Section 2.2.2. We deliber-

ately restrict Boolean properties to succeed only when their result is deterministic because it is questionable whether $(0 \,?\, 1) == 0$ should be satisfied and whether it should be equivalent to $0 == (0 \,?\, 1)$. Nondeterministic properties need to be specific about the sets[2] of different results and we provide three operators[3] for this purpose:

- The property $x \rightharpoonup y$ demands that x evaluates to every possible result of y, that is, the set of possible results of x must be a *superset* of the set of possible results of y.
- The property $x \leftharpoondown y$ is the same as $y \rightharpoonup x$ and demands that the set of possible results of x is a *subset* of the set of possible results of y.
- Finally, $x \rightleftharpoons y$ is satisfied iff the sets of possible results of x and y are *equal*. Note that using $\rightleftharpoons$ differs from using $==$ which demands the result sets to contain exactly one element.

With $\rightharpoonup$ we can specify that among the possible results of the *insert* operation the given element x appears at least as first and last element.

$$\begin{aligned} &\mathit{insertAsFirstOrLast} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Property} \\ &\mathit{insertAsFirstOrLast}\ x\ xs = \mathit{insert}\ x\ xs \rightharpoonup (x : xs \,?\, xs +\!\!+ [x]) \end{aligned}$$

We can again use *blackCheck* to generate test input for this property.

```
> blackCheck insertAsFirstOrLast
100 tests passed.
```

Using $\leftharpoondown$ we can check that the *permute* operation (see Section 2.2.2) preserves the length of its input for every result:

$$\begin{aligned} &\mathit{permutePreservesLength} :: [\mathit{Int}] \to \mathit{Property} \\ &\mathit{permutePreservesLength}\ xs = \mathit{length}\ (\mathit{permute}\ xs) \leftharpoondown \mathit{length}\ xs \end{aligned}$$

We could also use $\rightleftharpoons$ instead of $\leftharpoondown$ but using $==$ would not have the desired effect because $\mathit{length}\ (\mathit{permute}\ xs)$ may be nondeterministic.

3.1.2 Defining test-case generators

In order to check properties automatically, the function *blackCheck* needs to generate values of specific types. Nondeterministic value generators like discussed in Section 2.2.3 serve this purpose elegantly.

[2] We only consider *sets* of results, not *multi sets*. If a result is computed at all, we do not distinguish computations that compute it a different number of times.

[3] The ASCII representations of all used operators are listed in Appendix A.1.

We provide a type class[4] *Arbitrary* with a single operation *arbitrary* that yields test input of the corresponding type nondeterministically.

```
class Arbitrary a where
  arbitrary :: a
```

Implementations of *arbitrary* may yield any value of type *a* or only specific values. We provide instances for some predefined types. For example, the *Arbitrary* instance for *Bool* yields *False* or *True*.

```
instance Arbitrary Bool where
  arbitrary = False ? True
```

We can also define instances for parametrised recursive types that require an *Arbitrary* instance for the type parameters. For example, the instance for lists is defined as follows.

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = []
  arbitrary = arbitrary : arbitrary
```

The first occurrence of *arbitrary* in the right-hand side of the second rule belongs to the *Arbitrary* instance for the type *a* while the second occurrence is a recursive call to the *arbitrary* operation for lists. These definitions generalise the operations *bool* and *blist* defined in Section 2.2.3.

As there are infinitely many lists of type $[Bool]$, the search tree for *arbitrary* of type $[Bool]$ is infinite. The first six levels of this tree are visualised in Figure 3.1. Each inner node represents a nondeterministic choice for a constructor and its outgoing edges are labeled with the chosen constructor. The leaves of the tree are labeled with the corresponding values of type $[Bool]$.

As a more complex example, we define different *Arbitrary* instances for a data type representing heaps which are labeled trees that satisfy the *heap property*: the sequence of labels along any path from the root to a leaf must be non-decreasing. We use the following data type, which does not establish the heap property but can represent arbitrary labeled trees, to represent pairing heaps as described by Okasaki (1996).

```
data Heap a = Empty | Fork a [Heap a]
  deriving Show
```

[4] At the time of this writing an experimental fork of the Münster Curry Compiler provides the only implementation of type classes for Curry.

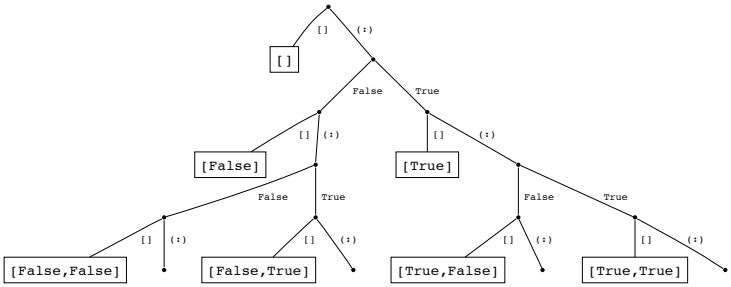


Figure 3.1: Search space for an *arbitrary* value of type $[Bool]$

The **deriving** clause tells the compiler to generate code that converts *Heap* values to a string which allows BlackCheck to print heaps as counter examples for a tested property. To ensure that users can only construct valid heaps, we can make the *Heap* type abstract by hiding its data constructors and provide constructor functions *emptyHeap* and *insertHeap* to construct valid heaps[5].

$$emptyHeap :: Heap\ a$$
$$insertHeap :: Ord\ a \Rightarrow a \to Heap\ a \to Heap\ a$$

We need to be able to compare heap entries to maintain the heap property. The implementations of both functions are given in Appendix A.3.

Simulating logic variables

A straightforward way to define an *Arbitrary* instance for the *Heap* data type is to simulate a logic variable of the same type.

$$\textbf{instance}\ Arbitrary\ a \Rightarrow Arbitrary\ (Heap\ a)\ \textbf{where}$$
$$\quad arbitrary = Empty\ ?\ Fork\ arbitrary\ arbitrary$$

This definition relies on an *Arbitrary* instance for the heap entries and the *Arbitrary* instance for lists described above to generate non-empty heaps. As it uses the hidden *Heap* constructors, it needs to be defined inside the *Heap* module. Another disadvantage is that it yields any value of type *Heap a* which does not necessarily satisfy the heap property. In order to use this definition for generating test input for properties on heaps, we need to use $\Longrightarrow$ in combination with a predicate $isValidHeap :: Ord\ a \Rightarrow Heap\ a \to Bool$ that checks the heap property.

[5] The type-class constraint *Ord a* ensures that the operation $\leqslant$ is defined for values of type *a*.

Generating valid test data

Instead of generating any value of type *Heap a*, we can also use the abstract constructors *emptyHeap* and *insertHeap* to generate only valid heaps. As we cannot define two *Arbitrary* instances for the same type, we use a **newtype** declaration for valid heaps. Such declarations are equivalent[6] to analogous **data** declarations but do not incur run-time overhead for matching their constructors which are not present at run time.

$$\begin{aligned}&\textbf{newtype}\ \mathit{ValidHeap}\ a = \mathit{Valid}\ \{\mathit{validHeap} :: \mathit{Heap}\ a\}\\ &\quad\textbf{deriving}\ \mathit{Show}\end{aligned}$$

We use *record syntax* to define the constructor $\mathit{Valid} :: \mathit{Heap}\ a \to \mathit{ValidHeap}\ a$ and the corresponding selector function $\mathit{validHeap} :: \mathit{ValidHeap}\ a \to \mathit{Heap}\ a$ at the same time. Introducing **newtype**s to define different type class instances for the same underlying type is a common idiom that serves well to provide different test-case generators for the same type. Assuming an *Arbitrary* instance for *ValidHeap a*, we can call *validHeap arbitrary* to generate test input of type *Heap a* according to the definition of *arbitrary* for the type *ValidHeap a*. The *Arbitrary* instance for the *ValidHeap* type uses the abstract constructors *emptyHeap* and *insertHeap* instead of the hidden constructors of the underlying data type and, thus, requires an ordering operation on the heap entries

$$\begin{aligned}&\textbf{instance}\ (\mathit{Arbitrary}\ a, \mathit{Ord}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ValidHeap}\ a)\ \textbf{where}\\ &\quad \mathit{arbitrary} = \mathit{Valid}\ \mathit{emptyHeap}\\ &\quad \mathit{arbitrary} = \mathit{Valid}\ (\mathit{insertHeap}\ \mathit{arbitrary}\ (\mathit{validHeap}\ \mathit{arbitrary}))\end{aligned}$$

When using this instance to generate test input for predicates on heaps, we do not need to check the heap property using $\Longrightarrow$ because all generated heaps are guaranteed to be valid. Of course, this depends on the correctness of *emptyHeap* and *insertHeap* which can be checked using *blackCheck*:

```
> blackCheck (isValidHeap . validHeap :: ValidHeap Int -> Bool)
100 tests passed.
```

We use the function composition operator $\circ$ to define an anonymous property based on the *isValidHeap* predicate and the *validHeap* **newtype** selector. We need to provide an explicit type annotation in order to tell *blackCheck* to generate heap entries of type *Int*.

[6] There is a subtle difference regarding laziness between **data** constructors and **newtype** constructors. The latter behave like *strict* data constructors. For example, $\mathit{Valid}\ \bot$ is equivalent to $\bot$ because **newtype** constructors are eliminated.

Generating custom test data

Instead of generating arbitrary valid heaps, we can also define a **newtype** that represents custom heaps that we consider sufficient for testing our properties. This is often useful to influence the search space such that more significant test input is generated. As an example, we define a type for custom heaps where all labels are small integers.

```
newtype CustomHeap = Custom {customHeap :: Heap Int}
  deriving Show
instance Arbitrary CustomHeap where
  arbitrary = Custom emptyHeap
  arbitrary = Custom (insertHeap digit (customHeap arbitrary))
digit :: Int
digit = 0; digit = 1; digit = 2; digit = 3; digit = 4;
digit = 5; digit = 6; digit = 7; digit = 8; digit = 9;
```

This instance is similar to the *Arbitrary* instance for *ValidHeap a* but instead of using *arbitrary* to generate heap entries we use a custom operation *digit* that yields an integer between 0 and 9.

We will compare the different ways to define test-case generators, especially the distribution of tests that they produce using different search strategies, experimentally in Section 3.1.4.

3.1.3 Enumerating test cases

Infinite search spaces, like those generated by evaluating definitions of the operation *arbitrary*, can be searched using different strategies. Curry systems that support the operation *getSearchTree* to reify the possibly nondeterministic result of an operation into a deterministic tree structure (see Section 2.2.5) usually provide functions to traverse the search tree in depth- or breadth-first order. Neither of these strategies is well-suited to be used for enumerating test cases: depth-first search is easily trapped in infinite branches of the search space and the memory requirements of breadth-first search prohibit its use for enumerating a large number of tests.

In the remainder of this subsection we investigate three different approaches that are better suited to enumerate test input.

1. Iterative deepening depth-first search trades memory requirements for run time to achieve level-wise enumeration like breadth-first search with the memory requirements of depth-first search.

2. Randomly selecting an arbitrary branch at each inner node of the search space instead of a specific one reduces the probability to diverge and leads to more diverse tests compared to depth-first search.

3. Searching the levels of the search space diagonally instead of sequentially leads to faster enumeration of deeper values while preserving good coverage of shallow levels.

We evaluate the different strategies experimentally in Section 3.1.4 and intuitively according to the following criteria:

1. We call an enumeration *complete* if every value in the search space is eventually enumerated. This property allows us to *prove* properties that only involve datatypes with finitely many values. Moreover, completeness implies that any node of an infinite search tree is reached in finite time.

2. It is desirable to obtain reasonably large test cases early in order to avoid numerous trivial test cases. We call a strategy *advancing* if it visits the first node of the n-th level of a search tree after $p(n)$ other nodes where p is a polynomial.

3. We call an enumeration *balanced* if the overall distribution of enumerated values is independent of the order of child trees in branch nodes. Balance is important to obtain diverse test input.

Depth-first search is advancing[7] but incomplete and unbalanced. Breadth-first search is complete and almost balanced but not advancing because it generates an exponential number of small values before larger ones.

Iterative deepening search

Iteratively incrementing the depth limit of depth-bounded depth-first searches enumerates the nodes of a search tree in breadth-first order. Hence, iterative deepening search is a complete strategy and, like breadth-first search, it is balanced but not advancing. The level on which a test case is found corresponds roughly to its size if the *arbitrary* operation is defined inductively and produces larger values in every recursive call. For example, the *arbitrary* operation for Boolean lists produces a search tree where larger lists are at deeper levels as we have seen in Figure 3.1. As iterative deepening search enumerates shallow levels completely before continuing with deeper levels,

[7] not always, however, because it is incomplete

it enumerates small test input exhaustively but does not generate large test input.

This behaviour is unfortunate if there are errors that are only exposed by complex test input. However, the *small scope hypothesis* claims that if there is a bug then it is *almost always* exposed by a simple test case. Or as Runciman et al. (2008) put it: if no simple test case exposes a bug then there *hardly ever* is any. Under this assumption it seems better to enumerate small tests exhaustively than to produce a non-exhaustive sampling of large tests.

We provide functions *depthCheck* and *smallCheck* for depth-bounded and iterative deepening testing. The call *depthCheck d p* checks the property *p* using all tests that can be reached within a depth limit of *d*. The call *smallCheck d p* checks the property *p* incrementally with all tests that can be reached with depth limits from 0 to *d*. The advantage of *smallCheck* is that it finds a smallest counter example first because it checks all levels from 0 to *d* incrementally. We can demonstrate this property by checking if the head of an arbitrary non-empty list of Booleans is always *False*.

```
> smallCheck 10 (\bs -> not (null bs) ==> not (head bs))
2nd test failed, arguments:
[True]
```

The function *depthCheck* is useful to reproduce a test failure if the depth of the smallest failing test is already known: *depthCheck* avoids the repeated enumeration of shallow levels that do not contain a counter example. The following example demonstrates how to use *depthCheck* with a specific depth formerly determined using *smallCheck*.

```
> smallCheck 12 (\l -> (length l>3) ==> (length l>head l))
51 tests passed.
> smallCheck 13 (\l -> (length l>3) ==> (length l>head l))
170th test failed, arguments:
[4,0,0,0]
> depthCheck 13 (\l -> (length l>3) ==> (length l>head l))
160th test failed, arguments:
[4,0,0,0]
```

The call to *depthCheck* 13 runs faster than the call to *smallCheck* 13 which arrives at depth 13 only after checking all tests at level 12 and above. The fact that *depthCheck* 13 finds the counter example ten tests earlier is due to different search order: *depthCheck* 13 enumerates the first 13 levels in depth-first order, *smallCheck* 13 in level order.

Both functions allow to prove properties if their set of possible input is finite. For example, if we check that an arbitrary Boolean value is either *True* or *False* then all two tests are generated.

```
> smallCheck 10 (\b -> b || not b)
2 tests passed.
> depthCheck 10 (\b -> b || not b)
2 tests passed.
```

Non-exhaustive, incomplete strategies like random search may enumerate the same test more than once and fail to detect a finite search space.

The test functions *smallCheck* and *depthCheck* are defined in terms of a function *betweenLevels* that enumerates all values in a *SearchTree* that can be found between the given levels.

$$
\begin{aligned}
&\mathit{betweenLevels} :: \mathit{Int} \to \mathit{Int} \to \mathit{SearchTree}\ a \to [a] \\
&\mathit{betweenLevels}\ \mathit{from}\ \mathit{to} = \mathit{go}\ 0 \\
&\quad \textbf{where}\ \mathit{go}\ \mathit{level}\ (\mathit{Value}\ x) \\
&\qquad\qquad \mid \mathit{from} \leqslant \mathit{level} \wedge \mathit{level} \leqslant \mathit{to} = [x] \\
&\qquad\qquad \mid \mathit{otherwise} \qquad\qquad\quad = [\,] \\
&\qquad\quad \mathit{go}\ \mathit{level}\ (\mathit{Choice}\ \mathit{ts}) \\
&\qquad\qquad \mid \mathit{level} \geqslant \mathit{to} = [\,] \\
&\qquad\qquad \mid \mathit{otherwise} = [x \mid t \leftarrow \mathit{ts}, x \leftarrow \mathit{go}\ (\mathit{level}+1)\ t]
\end{aligned}
$$

A call to *smallCheck d* executes a call to *betweenLevels l l* for each level l between 0 and d. A call to *depthCheck d* executes a single call *betweenLevels* 0 d.

For more fine-grained control of exhaustive testing, we provide a function *iterCheck* that generalises *smallCheck* and *depthCheck* and takes two parameters that specify how many levels to search in each iteration and how many iterations to perform. In fact, *smallCheck d* is defined as *iterCheck d* 1 and *depthCheck d* is defined as *iterCheck* 1 d. The complete implementation of *smallCheck*, *depthCheck*, and *iterCheck* is listed in Appendix A.4.5.

Random search

The role model for all property-based testing tools is QuickCheck (Claessen and Hughes 2000) which produces a random sampling of test input. Compared to exhaustive testing of small values this leads to larger and more diverse test cases.

We also provide a function *quickCheck* for random testing that searches for counter examples in the search space of test input by randomly choosing branches of inner nodes. When using *quickCheck*, tests are not reproducible and a different counter example may be found in different runs.

```
> quickCheck (\l -> (length l>3) ==> (length l>head l))
22nd test failed, arguments:
[118,0,-1,0]
```

```
> quickCheck (\l -> (length l>3) ==> (length l>head l))
12th test failed, arguments:
[10,1,-3,-2]
> quickCheck (\l -> (length l>3) ==> (length l>head l))
15th test failed, arguments:
[17,2,0,-5,-7]
```

Depending on how sparse counter examples are distributed in the search space, *quickCheck* may be unable to find them, even if they are small. On the other hand, it often finds counter examples sooner than exhaustive testing as it generates large values quickly.

The evaluation of random search with respect to completeness, advancement, and balance cannot be absolute but has to take probabilities into account.

If the probability to choose a leaf node as direct descendant of an inner node of the search tree is p, then the probability to reach a leaf on level n (for $n > 0$) is $(1-p)^{n-1} \cdot p$. Whether random search arrives at deeper levels frequently depends on the structure of the search space. If there are few leaves, then random search will arrive at deep levels quickly; if there are many leaves, it will reach deep levels rarely. A closer look reveals that random search is not advancing: The expected number of found leaves on level n (for $n > 0$) using m independent searches equals $m \cdot (1-p)^{n-1} \cdot p$. The expected number of enumerated leafs before reaching the first leaf on level n is the smallest number m such that the expected number of found leaves on level n using m independent searches is greater or equal to one. Hence, we are looking for the smallest m that satisfies the following inequality.

$$m \cdot (1-p)^{n-1} \cdot p \geqslant 1$$

If $p = \frac{1}{q}$, which means that every q-th child of an inner node is a leaf, then we can compute this m as follows.

$$\begin{aligned} m &\geqslant \frac{1}{(1-p)^{n-1} \cdot p} \\ &= \frac{1}{(\frac{q-1}{q})^{n-1} \cdot \frac{1}{q}} \\ &= (\frac{q}{q-1})^n \cdot (q-1) \end{aligned}$$

This derivation shows that the expected number of runs necessary to find one leaf on level n is exponential in n. Depending on p, however, reasonably deep levels are reached frequently. For example, for $p = \frac{1}{3}$, the expected

number of trials necessary to find a leaf on level n is $\lceil 2 \cdot (\frac{3}{2})^n \rceil$ and for $n < 10$ this number is less than n^2. As a consequence, random search behaves like an advancing strategy when enumerating 10^2 leaves of a search tree where every third child of an inner node is a leaf.

According to completeness, random search enumerates every value with a certain probability but it may take very long to reach a specific value if it is hidden deep in the search tree because values at shallow levels are enumerated repeatedly. Moreover, random search has no complete view of the search space and one cannot detect search space exhaustion when searching repeatedly for a single value.

Finally, random search is perfectly balanced. The order of child nodes at inner nodes of the search tree does not affect which results are enumerated by random search which chooses among child nodes randomly.

We can implement random search by shuffling the search tree before searching one value using depth-first search. The following function reorders the branches at inner nodes of a *SearchTree* randomly.

```
shuffleST :: StdGen → SearchTree a → SearchTree a
shuffleST _ (Value x)  = Value x
shuffleST g (Choice ts) = Choice (shuffle r (zipWith shuffleST rs ts))
  where r : rs = splitGen g
```

The type *StdGen* is a type for random number generators and the function $splitGen :: StdGen \to [StdGen]$ splits such a generator into an infinite list of independent generators. The function $shuffle :: StdGen \to [a] \to [a]$ reorders the elements of a list randomly such that every permutation is produced with equal probability (see Appendix A.4.6 for an implementation).

A call to *quickCheck* executes 100 calls to *shuffleST* followed by a depth-first search to find the leftmost leaf in each shuffled tree which is equivalent to following a random path from the root to the first found leaf in the original search tree.

For more fine-grained control of random testing, the function *rndCheck* generalises *quickCheck*. It takes three parameters that specify how many random searches to perform, how many valid tests to collect in each of them, and how many invalid tests to tolerate before giving up on finding a valid one in each random search. Tests are considered invalid if they have an unsatisfied precondition specified using $\Longrightarrow$. The *quickCheck* function is defined as *rndCheck* 100 1 10. It performs 100 random searches, enumerates at most 10 tests in each of them, and yields the first valid one. The *rndCheck* function is useful to generate larger tests at the cost of loosing independence. Enumerating multiple tests in a single run of depth-first search on a shuffled tree will often give similar results of increasing size.

Level diagonalisation

We have developed a new strategy for traversing infinite search spaces that provides stronger guarantees about advancement and completeness than random search and is more advancing than level-wise search. The improvement comes at the cost of higher space complexity than the previous searches. In its simple form our strategy is not balanced. We can combine it with randomisation, however, to get a balanced strategy.

The idea of our strategy is to compute the levels of the search space like breadth-first search but instead of enumerating the levels sequentially process them in a diagonally interleaved order such that deeper levels are reached before processing shallow levels completely. As a result, this strategy will exhaustively check small values but also find some larger values. Due to lazy evaluation (cf. Section 2.1.2) all levels are only produced as much as demanded by the traversal which leads to improved memory requirements compared to breadth-first search.

The following function computes a list of levels from a forest of *SearchTree*s.

$$
\begin{aligned}
&\mathit{levels} :: [\mathit{SearchTree}\ a] \to [[\mathit{SearchTree}\ a]] \\
&\mathit{levels}\ \mathit{ts} \mid \mathit{null}\ \mathit{ts} \quad = [\,] \\
&\qquad\quad \mid \mathit{otherwise} = \mathit{ts} : \mathit{levels}\ [u \mid \mathit{Choice}\ \mathit{us} \leftarrow \mathit{ts}, u \leftarrow \mathit{us}]
\end{aligned}
$$

Each level is a list of sub trees whose root is on the corresponding level of the search tree. Note that the function does not drop inner nodes to enumerate only values. Instead, it enumerates all nodes of the tree which allows us to consume them incrementally and is critical for the memory requirements of our strategy. Filtering a level to produce a list of the values it contains may demand large portions of the level which we avoid consciously.

We consume levels incrementally using the following function for list diagonalisation.

$$
\begin{aligned}
&\mathit{diagonals} :: [[a]] \to [[a]] \\
&\mathit{diagonals}\ [\,] \quad = [\,] \\
&\mathit{diagonals}\ (l : \mathit{ls}) = \mathit{zipCons}\ l\ ([\,] : \mathit{diagonals}\ \mathit{ls})
\end{aligned}
$$

$$
\begin{aligned}
&zipCons :: [a] \to [[a]] \to [[a]] \\
&zipCons\ []\quad ls \quad = ls \\
&zipCons\ (x:xs)\ []\quad = [[y] \mid y \leftarrow x:xs] \\
&zipCons\ (x:xs)\ (l:ls) = (x:l) : zipCons\ xs\ ls
\end{aligned}
$$

This function is best explained by example. Consider the following call and its result which shows that we can employ the *diagonals* function to enumerate an infinite list of infinite lists.

```
> take 3 (diagonals [[ (i,j) | j <- [1..]] | i <- [1..]])
[[(1,1)],[(1,2),(2,1)],[(1,3),(2,2),(3,1)]]
```

We layout this call to *diagonals* slightly differently to observe what happens:

$$
\begin{array}{lll}
diagonals\ [[(1,1),(1,2),(1,3),\ldots & = & [[(1,1)] \\
\quad ,[(2,1),(2,2),(2,3),\ldots & & ,[(1,2),(2,1)] \\
\quad ,[(3,1),(3,2),(3,3),\ldots & & ,[(1,3),(2,2),(3,1)] \\
\quad , \qquad\qquad\qquad \ldots & & ,\ldots]
\end{array}
$$

If we look at the input list as a matrix, then the output list is a list of its diagonals. The first diagonal contains the top-left element, the second diagonal contains the second element of the first row and the first element of the second row and so on.

Note that the first element of the n-th row of the matrix is enumerated after processing $\frac{n\cdot(n-1)}{2}$ elements. If the argument to the *diagonals* function is the list of levels of a search tree, then the first node on the n-th level is reached after visiting $O(n^2)$ other nodes[8]. As n^2 is a polynomial in n, enumerating the levels of search tree diagonally is an advancing search strategy. Its implementation combines the function *levels* and *diagonals* shown previously:

$$
\begin{aligned}
&allValuesDiag :: SearchTree\ a \to [a] \\
&allValuesDiag\ t = [x \mid d \leftarrow diagonals\ (levels\ [t]), Value\ x \leftarrow d]
\end{aligned}
$$

Figure 3.2 shows part of the search tree for arbitrary lists of Booleans. The first 100 nodes of the tree visited by level diagonalisation are highlighted. We can observe that shallow levels are processed completely and larger values are reached as well at the left spine of the tree. The grey colored parts of the search tree have not been visited and, hence, not been computed due to laziness. We can also see that level diagonalisation is complete: it eventually enumerates every leaf. It also enumerates every value only once and, hence, one can detect search space exhaustion. A disadvantage of level

[8]Depth-first search finds the first node on level n after $O(n)$ other nodes, level-wise strategies like breadth-first search or iterative deepening depth-first search after $O(2^n)$ other nodes.

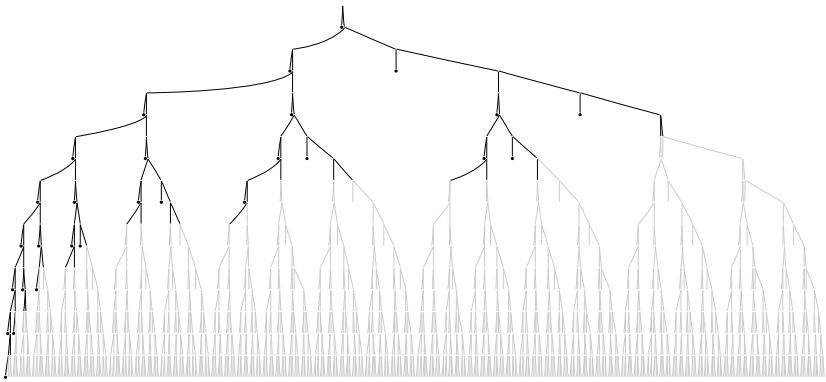


Figure 3.2: Level diagonalisation of $[Bool]$ values

diagonalisation is that most visited nodes are in the left part of the tree which makes this strategy unbalanced.

We can overcome this limitation using randomisation. If we apply the function *shuffleST* before enumerating the values of a search tree with the function *allValuesDiag*, then the enumerated values are independent of the order of child nodes in the original tree. Yet, especially enumerated values from deeper levels come from contiguous parts of the search tree. Large tests generated by level diagonalisation resemble each other even if we use randomisation. We can improve the diversity of tests generated by randomised level diagonalisation by running multiple searches on different randomised versions of the original tree. This comes at the cost that the same value may be enumerated as many times as we run searches—and especially for small values in shallow levels this is likely to happen—but the benefit of searching repeatedly is that more diverse large tests are generated. Figure 3.3 visualises the effect of running randomised level diagonalisation twice and combining the results. Still, the number of visited nodes is quadratic in the depth of the deepest visited node in each run of the search and large parts of the tree do not have to be computed.

The test function *blackCheck* shown previously combines different randomised runs of level diagonalisation to compute test input. As this strategy is complete, we can detect search space exhaustion and prove properties if the search space is finite. Reconsider the Boolean example used with *smallCheck* previously.

```
> blackCheck (\b -> b || not b)
20 tests passed.
```

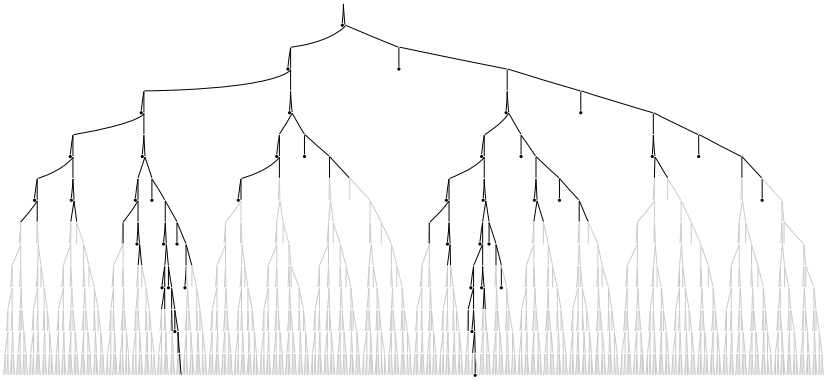


Figure 3.3: Combined randomised level diagonalisation of $[Bool]$ values

This time we need 20 tests to check the property because *blackCheck* combines 10 random searches and enumerates every Boolean value 10 times. Due to randomisation it may require multiple runs to find a counter example using *blackCheck*. For example, a counter example for the list property above is not always found:

```
> blackCheck (\l -> (length l>3) ==> (length l>head l))
21st test failed, arguments:
[635,-6,1,0]
> blackCheck (\l -> (length l>3) ==> (length l>head l))
100 tests passed.
> blackCheck (\l -> (length l>3) ==> (length l>head l))
1st test failed, arguments:
[6,-9,1,0]
```

For more fine-grained control of level diagonalisation, we provide a function *diagCheck* with three parameters that control how many randomised searches to perform, how many valid tests to enumerate in each search and how many invalid tests (tests with a failing precondition) to tolerate before giving up finding a valid one. In fact, *blackCheck* is defined as *diagCheck* 10 10 100 and, thus, performs 10 randomised searches, each time trying to enumerate 10 valid tests, and aborting a search run if it produces 100 invalid tests.

3.1.4 Experiments

We now compare the different strategies presented in Section 3.1.3 experimentally using the heap data type and its corresponding *Arbitrary* instances

from Section 3.1.2 for generating test input. In order to compare test-case distribution, a simple property is sufficient, so we just check the predicate *isValidHeap* as precondition to an always satisfied property.

$$heapProperty :: Heap\ Int \to Property$$
$$heapProperty\ h = isValidHeap\ h \Longrightarrow True$$

BlackCheck provides a combinator *collectAs* to statistically evaluate test input that we use to monitor the size and the depth of generated heaps.

$$heapSize :: Heap\ Int \to Property$$
$$heapSize\ h = collectAs\ \texttt{"size"}\ (size\ h)\ (heapProperty\ h)$$
$$heapDepth :: Heap\ Int \to Property$$
$$heapDepth\ h = collectAs\ \texttt{"depth"}\ (depth\ h)\ (heapProperty\ h)$$

The function $size :: Heap\ a \to Int$ computes the number of labeled nodes in a heap and $depth :: Heap\ a \to Int$ computes the length of a longest path from the root to a leaf.

Exhaustive testing of small values

We can use *smallCheck* to count the size of heaps in the first 10 levels of the search space using the various heap generators. If we use the simplest generator that simulates a logic variable of type *Heap Int*, we obtain the following distribution of test input.

```
> smallCheck 10 heapSize
180 tests passed.
  166 size: 1
   13 size: 2
    1 size: 0
```

The output means that there are 180 valid heaps within depth 10, 166 of them have size 1, and 13 have size 2. There is one heap of size 0 which is the empty heap. The generated heaps are so small that their depth equals their size. Therefore, we omit the results of measuring depth for the first heap generator.

We can check the second heap generator by using the **newtype** selector *validHeap* as follows.

```
> smallCheck 10 (heapSize . validHeap)
289 tests passed.
  129 size: 2
  127 size: 1
```

```
   31 size: 3
    1 size: 4
    1 size: 0
> smallCheck 10 (heapDepth . validHeap)
289 tests passed.
  145 depth: 2
  127 depth: 1
   15 depth: 3
    1 depth: 4
    1 depth: 0
```

Using a generator that generates only valid heaps increases the number of valid heaps within the depth limit 10. Moreover, there are now larger heaps within this depth limit – one has a size and depth of 4. The numbers for heaps of size one and two are similar showing that increasing depth is spent evenly on larger heaps and larger numbers used as labels.

If we restrict the heap labels to be digits using the generator for custom heaps the increasing depth can be spent to generate more large heaps.

```
> smallCheck 10 (heapSize . customHeap)
11111 tests passed.
10000 size: 4
 1000 size: 3
  100 size: 2
   10 size: 1
    1 size: 0
```

Now all heaps of size 0 to 4 that contain only digits are generated. Some of them are generated and, hence, counted multiple times. For example, the call *insertHeap* 0 (*insertHeap* 1 *emptyHeap*) has the same result as the call *insertHeap* 1 (*insertHeap* 0 *emptyHeap*) but both calls are counted individually. The depth of the valid heaps that contain only digits and can be found limiting the depth of the search space to 10 is distributed as follows.

```
> smallCheck 10 (heapDepth . customHeap)
11111 tests passed.
 5050 depth: 2
 4840 depth: 3
 1210 depth: 4
   10 depth: 1
    1 depth: 0
```

Random testing

The test-case distribution varies among different runs of the *quickCheck* function for random testing. Therefore, we perform general observations based on example executions and later point out differences to the other strategies based on multiple runs.

When using the simple generator that simulates a logic variable, there are only 88 valid heaps among 1000 that have been generated before giving up finding valid ones.

```
> quickCheck heapSize
88 tests passed.
   51 size: 0
   32 size: 1
    3 size: 2
    2 size: 3
```

Compared to *smallCheck*, *quickCheck* generates the empty heap in more than half of all valid tests but also generates 2 heaps of size 3. Measuring depth yields similar observations. Larger heaps are generated if we use a generator that produces only valid heaps. Obviously, 100 valid tests can now be found easily but still about half of the generated test input is the empty heap.

```
> quickCheck (heapSize . validHeap)
100 tests passed.
   52 size: 0
   28 size: 1
   12 size: 2
    3 size: 3
    3 size: 4
    2 size: 5
```

Using the custom heap generator to generate heaps that contain only digits has no big effect on the size of heaps generated by *quickCheck* although larger heaps seem to be a bit more likely than with the generator for arbitrary valid heaps.

Level diagonalisation

Finally, we investigate the test-case distribution of the *blackCheck* function that combines 10 randomised runs of level diagonalisation to enumerate test input.

When enumerating results of the simple generator using *blackCheck* the empty heap is generated 10 times and compared to *quickCheck* there are more heaps of all other sizes.

```
> blackCheck heapSize
100 tests passed.
   78 size: 1
   10 size: 0
    9 size: 2
    3 size: 3
```

Most of the generated heaps are of size one and as every heap is enumerated at most 10 times, we know that at least 8 different heaps of size one have been generated.

The depth and size of heaps enumerated using the generators for arbitrary valid heaps and custom heaps are similar. Here is an example call of *blackCheck* to generate arbitrary valid heaps.

```
> blackCheck (heapSize . validHeap)
100 tests passed.
   51 size: 1
   17 size: 2
   17 size: 3
   10 size: 0
    5 size: 4
```

Compared to *quickCheck* there are more heaps of size 1, 2, and 3, but fewer large heaps.

Comparison

In order to compare the different strategies and strengthen the observations made with the example executions, we measure the size and depth of generated heaps by enumerating more valid heaps than in the previous examples. The execution times are similar for all strategies and lie within a few seconds. We execute *smallCheck* with a depth limit of 13 instead of 10 and aim at 1000 instead of 100 valid tests using random search and level diagonalisation. More specifically, we use *rndCheck* 1000 1 10 instead of *quickCheck* and *diagCheck* 10 100 1000 instead of *blackCheck*. For the latter strategies we compute an average of 5 independent runs to account for randomisation.

The results of our experiments are depicted in Table 3.1. There is one column for each strategy that shows the size and depth of generated heaps according to the different heap generators. For example, in the top-right corner we can see that *diagCheck* produces 991 valid test cases using the simple heap generator that simulates a logic variable, 10 of them have size and depth 0 (are empty), 590 have size 1 and 655 have depth 1. If the numbers for size and depth are equal we show a single number instead of

	smallCheck size/depth	*rndCheck* size/depth	*diagCheck* size/depth
Heap Int			
#valid	1627	902/889	991
#0	1	495/494	10
#1	1359/1359	348/336	590/655
#2	260/264	42/50	245/269
#3	7/3	13/8	91/56
#4		2/1	37/1
#5		1/0	12/0
#6		1/0	4/0
#7			2/0
ValidHeap Int			
#valid	4060	1000	1000
#0	1	496/494	10
#1	1023	246/244	562/617
#2	1793/2496	131/196	270/270
#3	1023/486	65/49	107/40
#4	209/5	29/14	17/58
#5	11/4	17/2	17/4
#6		9/1	17/0
#7		4/0	
#8-12		3/0	
CustomHeap			
#valid	111111	1000	
#0	1	500/503	
#1	10	248/252	
#2	100/37741	131/181	
#3	1000/48664	61/50	
#4	10000/21406	28/11	
#5	100000/3289	16/2	
#6		7/1	
#7		5/0	
#8-12		4/0	

Table 3.1: Comparison of different strategies for black-box testing

the same number twice. The reason why there may be different numbers even for size 1 is that they have been obtained in different randomised runs.

Comparing the different generators, we can see that all strategies benefit from using the *Arbitrary* instance that generates values of type *ValidHeap Int*. Exhaustive testing of small values using *smallCheck* shows the most significant improvement, random testing using *rndCheck* produces considerably more heaps of size and depth 2 to 4, and level diagonalisation produces only slightly larger heaps compared to the simple generator.

Only *smallCheck* benefits from using the third generator for custom heaps that contain only digits and produces significantly more heaps of size and depth 4 and 5. The *rndCheck* function shows a similar distribution for the second and third generators. Finally, *diagCheck* 10 100 1000 fails to terminate within several minutes when enumerating test input of type *CustomHeap* and, therefore, there are no table entries for this combination.

Comparing the different strategies, we can see that *smallCheck* produces much more tests than the other strategies that stop after enumerating enough valid heaps. Random testing produces many small heaps – almost half of the generated heaps are empty. Level diagonalisation produces slightly more large heaps although it does not produce heaps larger than 7 which occasionally happens with random search.

From this comparison of strategies it is difficult to identify a clear winner. If a failing test can be found within a small depth bound, then exhaustive testing of small values suffices to expose it. If there are many failing tests and valid input can be generated easily, then random search might find them using significantly fewer tests than exhaustive search. If failing tests are rare, then a strategy that enumerates the search space more systematically than random search but is more advancing than exhaustive search like level diagonalisation can find the error quicker than the other strategies.

As we cannot recommend a single strategy, the BlackCheck tool provides all of them. The different strategies can be beneficially used in combination. For example, one could first try to find a failing test using random search with *quickCheck*, if no error is exposed use level diagonalisation, and, if *blackCheck* is too expensive, employ *smallCheck* to check small values exhaustively.

3.1.5 Implementation

In this subsection we highlight interesting aspects of the implementation of BlackCheck on a high level. The complete implementation is listed in Appendix A.4.

Input generators for primitive types

BlackCheck provides instance declarations for the *Arbitrary* type class for various predefined types. We have seen previously how to define *Arbitrary* instances for algebraic or abstract data types and now describe the instances that BlackCheck provides for the primitive types *Int* for integers and *Char* for characters.

Instead of enumerating integers sequentially we define a generator that creates a balanced tree of numbers.

```
instance Arbitrary Int where
  arbitrary = 0
  arbitrary = nat
  arbitrary = −nat


nat :: Int
nat = 1
nat = 2 ∗ nat
nat = 2 ∗ nat + 1
```

The search space generated by *arbitrary* and *nat* is a tree where each inner node has three children of which one is a leaf. All integers with an absolute value less than 2^n can be found within a depth limit of $n + 1$ in this tree.

We can select arbitrary characters from a list using the *oneOf* function that selects an element from a list nondeterministically.

```
instance Arbitrary Char where
  arbitrary = oneOf (['A'..'z'] ++ ['0'..'9'] ++ " \t\r\n")
oneOf :: [a] → a
oneOf (x : xs) = x ? oneOf xs
```

We restrict the *arbitrary* operation for the *Char* type to only generate specific characters nondeterministically.

Testable types

The testing functions like *blackCheck* are overloaded and take an arbitrary testable property as argument.

```
blackCheck :: Testable a ⇒ a → IO ()
```

The type class *Testable* provides an operation *tests* that takes a testable value and yields a search space of tests.

```
class Testable a where
  tests :: a → Search Test
```

A value of type *Test* represents a single test case storing the test result, a textual representation of the test input, and statistical information about the test. There are *Testable* instance declarations for deterministic Boolean properties and for more complex properties of type *Property*, for example, involving preconditions or properties of nondeterministic operations. The most interesting instance of *Testable*, however, is the instance for function types that uses the *arbitrary* operation to generate test input.

```
instance (Show a, Arbitrary a, Testable b) ⇒ Testable (a → b)
  where tests p = do x ← Search (getSearchTree arbitrary)
                     fmap (λt → t {input = show x : input t})
                          (tests (p x))
```

This instance declaration specifies that a function is testable if its argument type supports test-input generation and its result type is testable. Additionally, it requires the argument type to be an instance of the *Show* class which provides the *show* function to convert corresponding values into a string. The parametrised type *Search* is an instance of the type classes *Monad* and *Functor*. Hence, we can use **do**-notation to combine different searches and *fmap* to modify tests in the search space. The first line generates a search space for test input of type *a* using *getSearchTree arbitrary*. The call to *fmap* adds a textual representation of the input to all results of applying the given function *p* to every input *x*.

3.1.6 Summary

We have presented BlackCheck, a tool for property-based black-box testing of Curry programs. Compared to other approaches to property-based testing, implicit nondeterminism significantly simplifies the specification of test-case generators. We have seen how to define such generators for algebraic data types like lists and trees and for an abstract heap data type that provides custom constructor functions to maintain a data invariant.

The search features built into the functional logic language Curry provide flexible means to implement various strategies for test-case enumeration. We can provide the strategies 'exhaustive testing of small values' (Runciman et al. 2008) and 'random testing' (Claessen and Hughes 2000) that resemble strategies of existing tools for property-based testing and a new strategy, namely, randomised level diagonalisation, in a single framework.

We define desirable properties of search strategies used for testing. *Completeness* allows to proof properties by detecting search space exhaustion, *advancement* ensures the generation of sufficiently large tests early, and *balance* ensures diverse test cases. Exhaustive enumeration is complete and balanced but not advancing, random search is balanced but neither complete nor advancing (although it often behaves as if it was—see Section 3.1.3) and a randomised version of level diagonalisation is complete, advancing, and balanced.

Our experimental comparison suggests that all strategies complement each other and should be used in combination, which is simple using our tool that provides them all. Random testing is useful to find some errors quickly, level diagonalisation tests more thoroughly but also has higher memory requirements which precludes its use on some inputs. Exhaustive testing utilises the small-scope hypothesis to search for bugs systematically but may be slow because it is not advancing.

3.2 Glass-box testing

The BlackCheck tool discussed in Section 3.1 generates test input that is independent of the tested property. Glass-box testing tries to be smarter and generates test input according to the *implementation* of the property. For example, the following property introduced in Section 3.1.3 only demands the first element of the input list – all other elements do not influence whether or not the property fails:

$$\lambda l \to (\mathit{length}\ l > 3) \Longrightarrow (\mathit{length}\ l > \mathit{head}\ l)$$

Generating concrete input that fixes not only the first but all elements of the list is wasteful. In this example, black-box testing tries many lists that differ only in elements behind the first and does not make use of the fact that all those tests are equivalent. As a consequence, the search space for finding a counter example grows unnecessarily.

We can restrict such search-space explosion using lazy evaluation (Section 2.1.2). If we apply the above property to a nondeterministic value of type $[\mathit{Int}]$ only the first list element is evaluated. The nondeterminism hidden in other list elements does not influence the structure of the search space as those elements are not demanded to decide the property. As a result, the search space is much smaller than with black-box testing.

The deficiencies of black-box testing are apparent, when it is difficult to satisfy the precondition of a property. For example, none of the testing func-

tions presented in Section 3.1 manages to find valid inputs for the following slightly modified property.

$$\lambda l \rightarrow (length\ l > 1000) \Longrightarrow (length\ l > head\ l)$$

In this section, we present the design and implementation of a test framework for glass-box testing that finds a counter example to this modified property within a second. The implementation relies heavily on functional logic programming features, namely, lazy nondeterminism and built-in search. Lazy nondeterminism is crucial in order to avoid searching for parts of the test input that is irrelevant to decide a property and the built-in search tree representation of a nondeterministic computation is crucial to define different strategies to traverse the search space.

In the remainder of this section we discuss GlassCheck – our property-based framework for automated glass-box testing of functional logic programs and present an alternative to the sometimes too expensive level diagonalisation strategy (Section 3.2.1), discuss an implementation of fair predicates that increase the laziness of some combined properties (Section 3.2.2), and present practical benchmarks to compare black-box and glass-box testing (Section 3.2.3). The complete implementation of GlassCheck is listed in Appendix A.5.

3.2.1 Demand-driven testing

Unlike black-box testing, glass-box testing generates test input systematically according to the program's demand. Glass-box testing is useful for testing algorithmically complex program units. For example, when testing different implementations of height-balanced binary search trees, black-box testing would use the same tests for each implementation. Glass-box testing will use specific tests to execute the branches of the concrete implementation and is, hence, more likely to test each implementation thoroughly. The execution mechanism of lazy functional logic programming can be used to automate the systematic demand-driven generation of test input.

In Curry, we can apply a function to logic variables as input to bind the input exactly as demanded to compute a result. For example, we can evaluate a call to the predefined *reverse* function to a logic variable *l* and observe the result along with the binding for *l*.

```
> reverse l where l free
{l = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes
{l = [_a]} [_a]
```

```
More solutions? [Y(es)/n(o)/a(ll)] yes
{l = [_b,_c]} [_c,_b]
More solutions? [Y(es)/n(o)/a(ll)] yes
{l = [_d,_e,_f]} [_f,_e,_d]
More solutions? [Y(es)/n(o)/a(ll)] yes
{l = [_g,_h,_i,_j]} [_j,_i,_h,_g]
More solutions? [Y(es)/n(o)/a(ll)] no
```

This call to *reverse* has infinitely many nondeterministic results corresponding to infinitely many bindings of l to finite lists. The input list l is bound according to the demand of *reverse* and the list elements remain unbound.

In practice this approach is of limited use because not all operations bind unbound arguments. Especially, arithmetic operations usually do not bind unbound logic variables as arguments:

```
> map (+1) l where l free
{l = []} []
More solutions? [Y(es)/n(o)/a(ll)] yes
Suspended
More solutions? [Y(es)/n(o)/a(ll)] no
```

This example demonstrates that logic variables do not suffice as test-case generators for all types. Instead of narrowing them, primitive operations often suspend on unbound logic variables which hinders the process of test-case generation. We have already seen the solution to this problem in Section 3.1: The type class *Arbitrary* provides an operation *arbitrary* that can be used instead of a logic variable.

```
> map (+1) (arbitrary :: [Int])
[]
More solutions? [Y(es)/n(o)/a(ll)] yes
[1]
More solutions? [Y(es)/n(o)/a(ll)] yes
[1,1]
More solutions? [Y(es)/n(o)/a(ll)] yes
[1,1,1]
More solutions? [Y(es)/n(o)/a(ll)] no
```

Using the *Arbitrary* type class, it is still possible to use logic variables but we can also provide lazy nondeterministic generators as we did for integers.

Generators defined using the *Arbitrary* class can be evaluated *lazily* and, hence, be used instead of logic variables for demand-driven testing. Operationally, narrowing of logic variables and lazy evaluation of nondeterministic generators is very similar (see Section 2.2.3). Based on this idea, we provide the GlassCheck tool for demand-driven testing of Curry programs.

Implementation of GlassCheck

The GlassCheck tool provides functions similar to *quickCheck*, *smallCheck*, etc. shown previously. The functions are overloaded to allow checking properties of different types, especially properties with a different number of arguments (cf. Section 3.1.5).

$$quickCheck :: Testable\ a \Rightarrow a \to IO\ ()$$

The type class *Testable* is defined differently in GlassCheck. Instead of computing a search space of tests, the associated operation yields a single test nondeterministically.

```
class Testable a where
  test :: a → Test
```

The type *Test* also differs from the type shown previously:

```
type Test = Maybe [String]
```

In GlassCheck, a test is an optional representation of a counter example. A successful test is represented as *Nothing* and a failing test as *Just input* where *input* is a textual representation of the arguments given to the corresponding property that lead to the test failure. We can convert a Boolean value into a test easily.

```
instance Testable Bool where
  test True  = Nothing
  test False = Just []
```

The most important difference of GlassCheck compared to BlackCheck is the definition of the *Testable* instance for function types that handles test input generation.

```
instance (Show a, Arbitrary a, Testable b) ⇒ Testable (a → b)
  where test p = fmap (show x:) (test (p x))
          where x = arbitrary
```

Compared to the corresponding instance in BlackCheck, this definition is remarkably simple. Instead of generating a deterministic representation of the search space for the argument, the nondeterministic result of generating an arbitrary value is passed to the property. No monad is involved to combine the search spaces for the argument and for the result of the property. As the

Maybe type is an instance of *Functor* (see Section 2.1.3), we can use *fmap* to add a textual representation of the argument to the input list stored in failing tests. For successful tests, the argument is ignored.

The benefits of laziness for glass-box testing come at a cost. When passing nondeterministic input to properties, nondeterminism caused by evaluating the argument cannot be distinguished from nondeterminism caused by evaluating nondeterministic operations used in the definition of properties[9]. Therefore, we do not provide combinators for testing nondeterministic operations in GlassCheck.

Another disadvantage comes from the fact that arguments must only be evaluated as much as demanded by the property. We cannot store a textual representation of the arguments along with successful tests because converting an argument into a *String* would evaluate it completely and defeat the goal of demand driven testing. As a consequence, we cannot provide combinators to collect statistical information about arguments of successful tests.

We do provide the implication operator to specify preconditions of properties. In GlassCheck, it is simply defined as partial function, failing on invalid preconditions:

$$(\Longrightarrow) :: Bool \to a \to a$$
$$True \Longrightarrow a = a$$

The effect of using $\Longrightarrow$ is that tests that correspond to input that fails to satisfy the precondition of a property are pruned away from the search space. If the precondition yields *False* without evaluating the test input completely, this rejects many tests—possibly infinitely many—at once.

Different test functions

We provide different test functions for glass-box testing of properties that employ different strategies to search for counter examples. The functions *quickCheck* and *rndCheck* perform a random search. We can use *quickCheck* to search for a counter example to the property shown above.

```
> quickCheck (\xs -> (length xs>1000) ==> (length xs>head xs))
Failure in test 1 for input:
[2165,0,3,0,1,1,-6,2,-234,0,0,-3,0,18,1,-3,0,...
```

[9] Recent research explores how to allow to distinguish between the nondeterminism inherent to an operation and the nondeterminism passed in via arguments (see Section 3.3). However, at the time of this writing these ideas have not been implemented in a Curry compiler.

It finds a counter example within less than a second of which we only show the first 17 elements[10]. Whether or not random search performs well depends on the structure of the search tree, especially on the number of values it contains. The search trees that originate from glass-box testing are much less regular than those encountered during black-box testing because the lazy pruning of preconditions influences the structure of the search space. As invalid tests are not part of the search space, resulting search trees often contain leaves much less frequently. The following example demonstrates that this can be a problem for random search. Instead of using a fixed lower bound for the length of lists, we use an additional parameter to the property.

```
> rndCheck 1 (\x xs -> (length xs>x)==>(length xs>head xs))
Not enough free memory after garbage collection
Current heap size: 1048576000 bytes
```

In this example, even a single run of random search can exhaust gigabytes of memory if it does not arrive at a leaf. Although there are very small counter examples, random search rarely finds one if there are few values in the search tree.

We can use *smallCheck* to find small counter examples. Indeed, it quickly comes up with the following counter example for the same property.

```
> smallCheck 10 (\x xs -> (length xs>x) ==> (length xs>head xs))
iterations: 5
Failure in test 2 for input:
0
[1]
```

On the other hand, searching small levels incrementally can take very long if large test input is required to satisfy a precondition.

```
> smallCheck 10 (\xs -> (length xs>1000) ==> (length xs>head xs))
iterations: 10
OK, passed 0 tests.
> smallCheck 100 (\xs -> (length xs>1000) ==> (length xs>head xs))
iterations: 100
OK, passed 0 tests.
> smallCheck 1000 (\xs -> (length xs>1000) ==> (length xs>head xs))
iterations: 1000
OK, passed 0 tests.
> smallCheck 10000 (\xs -> (length xs>1000)==>(length xs>head xs))
iterations: 1020^C
```

[10] GlassCheck shows the complete counter example but we cut it for space reasons.

The *smallCheck* function processes the first 1000 levels quickly because the precondition rejects all test cases found within this limit without guessing any elements of the input list. Hence, *smallCheck* reports 0 tests for all runs searching up to 1000 levels. For lists of sufficient length, the search slows down noticeably. The 1021st level is not reached within one minute and no counter example is found. We can try *depthCheck* which performs depth-bound search without increasing the depth limit incrementally.

```
> depthCheck 10000 (\xs -> (length xs>1000)==>(length xs>head xs))
Failure in test 12 for input:
[1024,0,0,0,0,0,0,...
```

It finds a counter example for this property in less than a second.

Finally we provide the functions *sparseCheck* and *discrCheck* that perform limited discrepancy search on a randomised search tree either iteratively or not, respectively. Limited discrepancy search is similar to level diagonalisation, as we describe shortly, but requires less memory.

Limited discrepancy search

In Section 3.1.4 we observed the memory requirements of level diagonalisation to be prohibitive when enumerating test input with respect to a nondeterministic generator for custom heaps. Iterative deepening depth-first search enumerates the same results as breadth-first search with the memory requirements of depth-first search which are linear rather than exponential in the depth of the deepest visited node. We can apply a similar approach to define a search strategy which enumerates similar results to level diagonalisation by incremental depth-first searches with a different notion of limit.

Instead of a depth-limit, we use the notion of *discrepancy* introduced by Harvey and Ginsberg (1995). Limited discrepancy search is used with heuristic search methods where children of inner nodes are ordered from left to right according to some preference. The discrepancy of a node in the search tree is the number of choices *against* the heuristics that are necessary on its path from the root, that is, the number of steps to the right in a binary search tree. Hence, limited discrepancy search is robust against a limited number of wrong heuristic choices. Visiting only nodes with limited depth *and* discrepancy leads to exploration of the search space similar as shown in Figure 3.2. Because the limit is spend on both increasing depth and discrepancy, deep nodes are only found in the left part of the tree.

In order to point out the similarities of level diagonalisation and limited depth and discrepancy search, we reconsider the application of the *diagonals* function to the infinite matrix discussed in Section 3.1.3.

$$\begin{array}{lll} \mathit{diagonals}\ [[(1,1),(1,2),(1,3),\ldots & = & [[(1,1)] \\ \quad ,[(2,1),(2,2),(2,3),\ldots & & \quad ,[(1,2),(2,1)] \\ \quad ,[(3,1),(3,2),(3,3),\ldots & & \quad ,[(1,3),(2,2),(3,1)] \\ \quad , \qquad\qquad\qquad\qquad\quad \ldots & & \quad ,\ldots] \end{array}$$

We can also compute the diagonals by enumerating the entries of the matrix in an order determined by the required steps to the right and down.

Instead of computing the levels of the search tree first, we can implement a variant of limited discrepancy search directly on values of type *SearchTree a*.

$$\begin{array}{l} \mathit{allValuesDiscr} :: \mathit{Int} \to \mathit{SearchTree}\ a \to [a] \\ \mathit{allValuesDiscr}\ \_\ (\mathit{Value}\ x) = [x] \\ \mathit{allValuesDiscr}\ d\ (\mathit{Choice}\ ts) \\ \quad \mid d \leqslant 0 \quad\ = [\,] \\ \quad \mid \mathit{otherwise} = \mathit{concat}\ (\mathit{zipWith}\ \mathit{allValuesDiscr}\ [d-1, d-2\,..\,0]\ ts) \end{array}$$

This function limits the depth and discrepancy of visited nodes and, thus, produces an enumeration similar to level diagonalisation. As nodes within the given limit are enumerated using depth-first search, *allValuesDiscr* needs less memory than *allValuesDiag* which often retains the visited parts of levels in memory.

Limited discrepancy search does not produce the same enumeration as level diagonalisation but behaves similarly with respect to the properties discussed in Section 3.1.3. Limited discrepancy search is complete. It is advancing because it makes at most d steps to the right when visiting a node at depth d and, thus, visits $O(d^2)$ nodes before reaching one on level d. As we do not have an underlying heuristics that orders the nodes in the search tree, being left-biased is a disadvantage of limited discrepancy search in our setting. However, we can make it balanced using randomisation.

3.2.2 Fair predicates

The lazier a property checks whether it is satisfied, the smaller is the search space of counter examples. Unevaluated nondeterministic choices in test input do not influence the size of the search space.

The meaning of Boolean conjunction and disjunction is symmetric, yet both are usually implemented sequentially, evaluating their first argument before the second. If the first argument of a conjunction is expensive to compute and the second is false, then evaluating the first argument is worthless effort. If the first argument is nondeterministic and the second is deterministically false, then the result of sequential conjunction is needlessly nondeterministic. Similar observations can be made for sequential disjunction if

the second argument is true. Ideally, implementations of conjunction and disjunction would evaluate both arguments in parallel and stop if one determines the result. Without concurrency, we can evaluate both arguments stepwise interleaved. In this subsection, we discuss the implementation of combinators for fair Boolean conjunction and disjunction. In Section 3.2.3 we will see how they help finding bugs quicker.

We provide combinators to construct primitive answers and functions for negation, conjunction, and disjunction.

$$
\begin{array}{ll}
\mathit{answer} & :: \mathit{Bool} \to \mathit{Answer} \\
\mathit{fromAnswer} & :: \mathit{Answer} \to \mathit{Bool} \\
\mathit{true}, \mathit{false} & :: \mathit{Answer} \\
\mathit{neg} & :: \mathit{Answer} \to \mathit{Answer} \\
(\bigwedge), (\bigvee) & :: \mathit{Answer} \to \mathit{Answer} \to \mathit{Answer}
\end{array}
$$

We also implement some convenience functions to work with answers – see Appendix A.5.1 for the complete interface and its implementation.

Before the implementation idea, we show an example of using fair predicates that demonstrates the fairness of conjunction. The function *indefinite* is an answer that is neither true nor false:

$$
\begin{array}{l}
\mathit{indefinite} :: \mathit{Answer} \\
\mathit{indefinite} = \mathit{neg}\ \mathit{indefinite}
\end{array}
$$

The Boolean counterpart of this function diverges, and indeed,

$$\mathit{fromAnswer}\ \mathit{indefinite}$$

describes a diverging computation. We can, however, use *indefinite* as argument to fair predicate combinators and obtain a useful Boolean result from applying *fromAnswer* to the result. Both

$$\mathit{fromAnswer}\ (\mathit{false} \bigwedge \mathit{indefinite})$$

and

$$\mathit{fromAnswer}\ (\mathit{indefinite} \bigwedge \mathit{false})$$

yield *False* without diverging.

In order to interleave the evaluation of predicates, we need to be able to suspend it. We use lazy evaluation to suspend the computations of answers and, hence, represent answers by a data type that resembles the *Bool* type but has an additional case for suspended answers.

$$\textbf{data}\ \mathit{Answer} = \mathit{Yes} \mid \mathit{No} \mid \mathit{Undecided\ Answer}$$

If an answer is undecided it is only evaluated if we demand the evaluation of the argument of the *Undecided* constructor. The function *fromAnswer* evaluates an answer completely and yields the corresponding Boolean value.

$$\begin{aligned}
&\mathit{fromAnswer} :: \mathit{Answer} \to \mathit{Bool}\\
&\mathit{fromAnswer}\ \mathit{Yes} &&= \mathit{True}\\
&\mathit{fromAnswer}\ \mathit{No} &&= \mathit{False}\\
&\mathit{fromAnswer}\ (\mathit{Undecided}\ a) &&= \mathit{fromAnswer}\ a
\end{aligned}$$

The *Answer* type is abstract. We hide its constructors to prevent users from observing the internal representation of answers.

The function *neg* yields an undecided answer which can be evaluated further to get the result:

$$\begin{aligned}
&\mathit{neg} :: \mathit{Answer} \to \mathit{Answer}\\
&\mathit{neg}\ a = \mathit{Undecided}\ (\mathit{negation}\ a)\\
&\quad \textbf{where}\ \mathit{negation}\ \mathit{Yes} &&= \mathit{No}\\
&\qquad\quad \mathit{negation}\ \mathit{No} &&= \mathit{Yes}\\
&\qquad\quad \mathit{negation}\ (\mathit{Undecided}\ b) &&= \mathit{neg}\ b
\end{aligned}$$

Every call to *neg* introduces an additional occurrence of the *Undecided* construtor which signals that more steps need to be done to compute the result and some other expression may as well be evaluated first. Hence, *indefinite* describes infinitely nested calls to the *Undecided* constructor.

To illustrate the interleaving in binary operations, we discuss the implementation of conjunction.

$$\begin{aligned}
a \bigwedge b = \mathit{Undecided}\ (\textbf{case}\ (a, b)\ \textbf{of}&\\
&(\mathit{Yes} &&, _ &&) \to b\\
&(\mathit{No} &&, _ &&) \to \mathit{No}\\
&(_ &&, \mathit{Yes} &&) \to a\\
&(_ &&, \mathit{No} &&) \to \mathit{No}\\
&(\mathit{Undecided}\ x &&, \mathit{Undecided}\ y) &&\to x \bigwedge y)
\end{aligned}$$

The presence of undecided answers lets us check whether either of the arguments is decided or both are undecided. If one answer is *No*, then the result of $\bigwedge$ is *No*; if one is *Yes*, then the result is the other argument. The implementation is not completely balanced as the left argument is matched before the right but if either argument is *No* and the other is undecided then the result is *No*. Disjunction is implemented similarly, yielding *Yes* if either argument is *Yes*.

3.2.3 Practical experiments

In order to compare the presented functions for black-box and glass-box testing, we have translated standard textbook algorithms to Curry, inserted different kinds of errors, and executed the different test functions on properties that expose the error if a suitable counter example is found. These errors range from deep algorithmic errors (like in Kruskal's algorithm) based on an insufficient understanding of the algorithm to simple oversights such as switched arguments and wrongly selected operations (like *mod* instead of *div* in the algorithm for matrix chain multiplication). Oversights are usually revealed by many counter examples and almost all test functions expose them. Although deep algorithmic errors require sophisticated test cases to expose them, relatively small counter examples can be found. Where appropriate, we define preconditions both using sequential Boolean predicates and fair predicates as discussed in Section 3.2.2 to investigate the benefits and overhead of the latter. In the following, we describe each experiment in detail. See Table 3.2 for a summary.

Heap property

We have introduced an oversight in an internal merge function that is used by the insert function for heaps shown in Section 3.1.2. The erroneous implementation may generate heaps that violate the heap property and we expose the error by specifying that heaps generated by *insertHeap* must satisfy the predicate *isValidHeap* if the original heap satisfies it.

Almost all test functions expose this simple error. The first two columns in Table 3.2 correspond to black-box testing of small and random values respectively. By $+$ we denote that the error has been found and by $-$ that it remained undetected. If the run time exceeds one second, then we additionally show it along with the $+$ or $-$ sign.

The remaining columns correspond to glass-box testing using limited depth (denoted as *small*), randomised limited discrepancy (denoted as *sparse*), or simple random search (denoted as *random*). It turns out that random search is inappropriate for glass-box testing due to the effect described in Section 3.2.1: if the search tree contains only few results, then random search is likely to descend very deeply exhausting available memory quickly.

Heap sort

We have used the correct heap implementation to define a heap-sort function that works by first inserting all elements into the empty heap and then

	black-box testing		glass-box testing					
	small	random	small		sparse		random	
			sequential	fair	sequential	fair	sequential	fair
Heap	+	+	+	+	+	+	− (2s)	− (2s)
HeapSort	+	+	+		+		+	
AVL insert	+ (2s)	− (11s)	+	+	+ (1s)	+ (5s)	− (2s)	− (2s)
AVL delete	− (49s)	− (9s)	+ (1m8s)	+ (1m1s)	+/− (6m)	+/− (7m)	− (2s)	− (2s)
Strassen	+ (4s)	−	+		+ (1s)		− (5s)	
Dijkstra	+ (2s)	−	+	+	+ (7s)	+ (3s)	− (2s)	− (2s)
Kruskal	+ (4m8s)	−	+ (2m59s)	+ (2m4s)	+/− (4m)	+/− (3m)	− (8s)	− (2s)
Dynamic	+	+	+	+	+	+	− (2s)	− (2s)
Train Ticket	− (>20m)	−	+ (24s)		+ (45s)		− (3s)	

Table 3.2: Comparison of black-box and glass-box testing

transforming this heap into a list. To expose the simple oversight in the function that transforms a heap into an ordered list, we define a property that checks whether the resulting list is sorted and contains all elements of the original list.

Every test function quickly exposes the error. In fact, this is the only case where random search works for glass-box testing. The reason is that the heap-sort property does not involve a precondition and, hence, no tests are pruned from the search space. Due to the lack of a precondition, we have only used sequential Boolean predicates and there are no results for fair predicates.

AVL trees

AVL trees (Adelson-Velskii and Landis 1962) are height-balanced binary search trees that rely on sophisticated restructuring operations—so called rotations—to maintain balance after inserting or deleting elements. AVL trees are used in many of the following examples as auxiliary data structures. We have investigated two different algorithmic errors. Both propagate balance information incorrectly – the first one in the insert function and the second one in an auxiliary function used by the delete function.

It turns out that the first error is detected more easily. Apart from those that use random search, all test functions expose the error in the insert function within a few seconds. In this example, random search is insufficient even for black-box testing as it does not generate enough valid AVL trees that satisfy the precondition.

The error in the re-balancing function used by the delete function is one of the few that is not exposed by black-box testing of small values. Without pruning the search space according to the precondition, the error remains undetected. Glass-box testing of small values exposes the error in about a minute and slightly faster if fair predicates are used. Randomised limited discrepancy search finds the error only in some runs and takes significantly longer than glass-box testing of small values. The run times for sparse testing are rough averages of multiple runs incrementing the discrepancy until it reaches a limit of 30.

Strassen's algorithm for matrix multiplication

Strassen's algorithm for matrix multiplication multiplies two $n \times n$ matrices by performing seven multiplications of $n/2 \times n/2$ sub-matrices and adding and subtracting the results in a tricky way. We have introduced an oversight in the part of the algorithm that adds and subtracts sub-matrices and expose

it by comparing the result of the erroneous multiplication function with a function for naive matrix multiplication. As a precondition we require that the matrices generated as test input are square matrices of equal size. This is the only example where we use a custom *Arbitrary* instance rather than reusing predefined instances for base types. Our instance declaration generates only square matrices and an additional precondition checks whether both matrices passed to the multiplication function are of equal size.

The error is exposed quickly by black-box and glass-box testing of small values as well as sparse glass-box testing using discrepancy search. Random search fails to find sufficiently large input matrices of equal size during black-box testing and does not find test results during glass-box testing. We did not express the precondition using fair predicates as it only consists of a simple length comparison which cannot be interleaved.

Dijkstra's shortest path algorithm

Dijkstra's shortest path algorithm computes the distance of every node in a weighted directed graph to some start node. It maintains a heap of nodes ordered by upper bounds for their distance to the start node. In each iteration a node v with minimal upper bound u is extracted from the heap and placed in a set of finished nodes. Each unfinished neighbour v' of v is inserted into the heap with an upper bound computed from u and the weight of its edge from v. Our implementation uses, in addition to the mentioned heaps, (correct) AVL trees for representing weighted directed graphs and distance tables.

We have introduced a deep algorithmic error that leads to incorrect handling of the heap ignoring that it may contain nodes several times with different upper bounds for their distance to the start node. We expose this error using a property that compares the computed distance table with the result of a correct implementation of Dijkstra's algorithm. The property includes a sophisticated precondition that checks whether the test input is a valid representation of a weighted directed graph.

This error is detected quickly by all test functions that do not use random search. Discrepancy search is again significantly slower than depth-bound search and fair predicates cut its run time in half which shows the benefit of interleaved property checking in the presence of complex preconditions.

Kruskal's minimum spanning tree algorithm

We have implemented Kruskal's algorithm to compute a minimum spanning tree of a weighted undirected graph. The algorithm adds a selected edge to

the spanning tree in each iteration. It always selects an edge with minimum weight among all edges that do not introduce a cycle. We use a (correct) heap to maintain the edges and a union-find data structure implemented using (correct) AVL trees to check whether a considered edge would introduce a cycle. The weighted undirected graph is also represented as AVL tree.

Our implementation is based on the Haskell code found in the textbook by Rabhi and Lapalme (1999). Interestingly, the code in the book contains a deep algorithmic error in the use of the union-find data structure that we are able to detect. We expose the error using a property that compares the nodes and the weight of the computed spanning tree with the nodes and weight computed by a correct implementation. The property includes a sophisticated precondition that checks whether the test input is a valid representation of a weighted undirected graph.

This error turns out most difficult to find among those checked in our experiments. Black-box testing of small values reveals it after about four minutes, glass-box testing of small values with a precondition expressed via sequential Boolean predicates finds a counter example after three minutes, and when using fair predicates the run time can be reduced further to about two minutes. Whether or not randomised discrepancy search finds the error depends on the particular program run. The run times shown for sparse testing are rough averages of multiple runs iteratively increasing the discrepancy up to the limit 30. Random search does not expose the error using black-box or glass-box testing.

Dynamic programming

As an example for dynamic programming, we implement a higher-order function for tabled computations and use it to solve the matrix chain multiplication problem: given a sequence of matrices find the most efficient way to multiply them. As matrix multiplication is associative, the different matrices can be multiplied in different orders which requires a varying number of primitive operations depending on the involved dimensions. Our erroneous implementation spuriously uses the operation *mod* instead of *div*. We expose this oversight by comparing the computed result with the result of a correct implementation of the problem. The property involves a simple precondition that checks that all given dimensions are positive.

This error is detected quickly by almost all testing functions. Only glass-box testing with random search does not find a counter example.

Train ticket price

Finally, we have implemented a function that computes the price of the cheapest ticket for the German railway company. After many simplifications of the price structure, the managers of Deutsche Bahn have installed a system where the price of a ticket depends on just 16 parameters. Implementing this system leads to a sophisticated branching structure and some of the branches are executed rarely. We have introduced an oversight into one of these branches and exposed it by comparing the computed result with the result of a correct implementation. The property includes a simple precondition that ensures that the 16 input parameters are assigned sensibly.

Black-box testing does not expose the error hidden in one branch of the computation. Even after 20 minutes of exhaustive testing of small values no counter example is found. Glass-box testing reveals the error after half a minute when checking small values exhaustively, randomised discrepancy search takes a bit longer. As the run time of sparse glass-box testing varies, we have computed an average of multiple runs.

The search space in this example is huge because many of the input parameters are numbers which are evaluated nondeterministically to all possible values. An alternative representation of numbers as logic variables in combination with constraint programming to bind them could significantly reduce the size of the search space in this example and others that rely heavily on arithmetic computations.

3.2.4 Summary

Glass-box testing, as opposed to black-box testing, is the systematic generation of test input considering the concrete implementation of the tested code. It is often performed by manually investigating the source code and its branching structure to devise test input that executes specific parts of the program. Lazy functional logic programming allows to automatically generate test input according to the demand of a program, which results in test input that is tailor made for executing different branches of its implementation.

One possibility to leverage the execution mechanism of functional logic programming to generate test input is to call the tested function with logic variables as input. As a result of executing such a call in a functional logic programming environment, the input variables are bound exactly as much as demanded by the tested function. A disadvantage of this approach is that it does not generalise to arithmetic computations. Arithmetic functions usually do not bind unbound arguments. Therefore, generating test cases by binding logic variables is only possible with programs that do not use

arithmetic functions or use an implementation of numbers as algebraic data types (Braßel et al. 2008).

We generalise the idea of narrowing-driven test-case generation by using class-based overloading which allows user-defined implementations of test-case generators. The simplest form of nondeterministic generators simulates the binding of unbound logic variables of algebraic data types using lazy nondeterminism (see Section 2.2.3) but different implementations are possible. For example, we provide various implementations of generators for an abstract heap data type and a generator for numbers that nondeterministically evaluates to every integer.

An alternative implementation for a generator of arbitrary type could simply yield a logic variable. This requires, however, that the operations that demand the test input bind unbound arguments appropriately. For numbers this requires constraint programming which could complement our test framework beneficially.

Demand-driven testing is useful for testing properties with preconditions. While black-box testing may generate many tests that violate a precondition, demand-driven testing can reject partially evaluated test input that represents many invalid tests at once. Complex and restrictive preconditions often *improve* the performance of glass-box testing as they prune the search space.

We have introduced errors in nine algorithmically complex programs to compare black-box and glass-box testing with different strategies. We did not consider level diagonalisation for black-box testing because of its performance problems. Limited discrepancy search is a more efficient alternative that we use for glass-box testing where it is outperformed by iterative deepening depth-first search. Simple random search is rarely useful for glass-box testing at least if the properties specify preconditions. In our experiments, glass-box testing with depth-bound search outperforms all other strategies.

3.3 Chapter notes

The basis of the work presented in this chapter has been published previously. The idea of using Curry to systematically generate glass-box tests has been published in the Proceedings of the ninth ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (Fischer and Kuchen 2007). The design and implementation of a tool for black-box testing of Curry programs appeared in the Proceedings of the 9th International Symposium on Functional and Logic Programming (Christiansen and Fischer 2008). The benchmarks used in the comparison of black-box and glass-box testing (Section 3.2.3) resemble those presented in work on code

coverage (see Chapter 4) which has been published in the Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Fischer and Kuchen 2008). The implementation of fair, interleaving predicates (Section 3.2.2) has been published online (Fischer 2009b) and is available via the Haskell package database (Hackage).

The level diagonalisation search strategy that we have used for black-box testing has been first presented by Christiansen and Fischer (2008). Only later, we discovered the similarities to limited discrepancy search (Harvey and Ginsberg 1995) which has similar properties but is more efficient. For random search we use an approach similar to *iterative sampling* (Langley 1992) with the difference that our search algorithm does not stop at dead ends in the search tree but backtracks until it finds the first result in each randomised run.

Related work

Early tools for black-box testing of functional programs generate test input randomly or deduce them from a specification (Claessen and Hughes 2000; Koopman et al. 2002). Since they do not take the implementation into account, they cannot ensure that all parts of the program are actually executed by the test cases. Hence, errors in uncovered parts of the program may remain undetected by these tools.

Simultaneously with our developments, Lindblad (2007) and Runciman et al. (2008) have developed new tools for demand-driven testing of Haskell programs. Instead of using it, they simulate functional logic programming in Haskell either by using a dedicated library (Naylor et al. 2007) or by exploiting a tricky combination of laziness and exceptions to simulate logic variables. As these tools do not use a functional logic programming language but simulate their features, their implementation is not as simple as the implementation of our glass-box test tool (see Section 3.2.1).

In Section 3.2.1 we have mentioned that we cannot provide combinators for nondeterministic properties when using glass-box testing. The reason is that we cannot distinguish whether a nondeterministic choice is caused by demanding (or narrowing) an unknown argument or by nondeterministic operations used in the property. Nondeterminism in the arguments corresponds to different test cases whereas nondeterminism in the property corresponds to different results of a single test case. Recently, Antoy and Hanus (2009) have proposed *set functions* that allow to make exactly the missing distinction. The set function f_S of a function f encapsulates only the nondeterminism in the definition of f but not that in arguments. If there was a Curry system that would support both set functions and encapsulated search,

then we could support nondeterministic properties during glass-box testing by encapsulating the nondeterministic results of calling the corresponding set functions of properties.

An approach for generating glass-box test cases for Java has been presented by Lembeck et al. (2004); Müller et al. (2004). They incorporate techniques known from logic programming into a symbolic Java virtual machine in order to implement code-based test-case generation. A similar approach based on a Prolog simulation of a Java Virtual Machine is presented in Albert et al. (2007). A related approach to test-case generation for logic programs is discussed in Degrave et al. (2009). Here, test cases are not generated by executing the program but by first computing constraints on input arguments that correspond to an execution path and then solving these constraints to obtain test inputs that cover the corresponding path.

4 Code Coverage

In the previous chapter we have distinguished back-box testing and glass-box testing. Glass-box testing aims at executing different parts of the source code of a tested program systematically. We can use the notion of code coverage to describe in detail how tests execute the tested program.

For imperative programming languages there are a number of established code-coverage criteria. For example,

function coverage describes which functions, procedures, or methods of the tested program have been executed by a test,

statement coverage describes which statements have been executed, and

condition/decision coverage describes to which results Boolean expressions have been evaluated and which branches of conditional control structures like if-then-else statements or while-loops have been executed.

More complex criteria investigate the control or data flow of the tested program. They allow to detect which edges of—or paths through—the control-flow graph of a program are executed or which assignments of variables affect which of their uses in a specific program run.

Code coverage information serves different purposes. Most importantly, it increases the confidence in tested code, if the tests cover it (almost) completely. It is important to note, however, that no coverage criterion can guarantee the absence of errors even if full coverage is achieved. A finite set of *covered items* can only approximate the possibly infinite variety of program behaviours. Coverage criteria differ in how accurately they distinguish interesting differences in program runs and how well they expose errors when full coverage is achieved.

Another possibility to leverage code coverage information is to present programmers information about the items covered during failing tests. This can narrow down the amount of code programmers need to review when searching for the cause of test failure. For example, the information which statements have been executed in a failing test and which branches were taken in certain decisions may hint at where the error is.

Finally, code coverage can also be used to automatically minimise a large set of generated tests. Tests that induce code coverage which is already obtained by other tests can be considered redundant and eliminated from the

test suite. Finding a small set of test cases that achieve the same code coverage as a much larger set of automatically generated tests is useful for regression testing where tests are re-executed after changes of the implementation. Also, if the result of tests must be evaluated manually by programmers, it is important not to present them with too many tests but show only a small number of relevant ones.

Expression coverage

Although well investigated for imperative programming languages, code coverage has received little attention in the area of declarative programming. Simultaneously with our developments, Gill and Runciman (2007) have developed Haskell Program Coverage (HPC) which implements *expression coverage*—a coverage criterion for the functional programming language Haskell inspired by statement coverage. As an introduction to declarative code coverage we review this coverage criterion and discuss its benefits and limitations.

Expression coverage describes which expressions in a program are demanded. Coverage is monitored for every sub-expression of combined expressions and, hence, allows to observe coverage at a fine granularity: when executing the call $takeWhile\ (>0)\ (3:2:1:0:0:0:[\,])$ HPC would mark the first four list constructors and the first four elements as being demanded and the remaining zeros as well as the constructor for the empty list as unevaluated. HPC is useful to get an overview which parts of a program are executed by a program run. It can be combined with automated test tools like QuickCheck and SmallCheck to get hints where testing can be improved.

Expression coverage can be easily presented to the user in the form of colourised source code that shows which parts of a program have been executed. Hence, in order to track down a bug, users can run a failing test and highlight the executed parts of the program to find the error more quickly.

It turns out, however, that expression coverage is by no means a thorough criterion. There are small programs—like the following—where HPC reports 100% code coverage for a program run that executes an erroneous function without exposing the fault.

$$
\begin{aligned}
&reverse\ [\,] &&= [\,]\\
&reverse\ (x:xs) &&= [x] +\!\!+ reverse\ xs\\
&[\,] +\!\!+ ys &&= ys\\
&(x:xs) +\!\!+ ys &&= x:xs +\!\!+ ys\\
&test\ xs\ ys &&= reverse\ xs +\!\!+ reverse\ ys == reverse\ (ys +\!\!+ xs)\\
&main &&= print\ (test\ [1]\ [\,])
\end{aligned}
$$

This implementation of *reverse* is faulty: the arguments of $+\!\!+$ in the second rule of *reverse* are swapped. Yet checking the anti-distributive property of *reverse* with a singleton and an empty list demands every expression in the program without revealing the bug. The result of the shown call *test* $[1]$ $[]$ is incidentally *True* because *reverse* is never called with a list with more than one element (on such short lists the given definition of *reverse* is correct). Calling *test* $[1]$ $[2]$ reveals the error—and automated test tools come up with this or a similar test—but this call is not necessary to achieve full expression coverage.

This qualifies the adequacy of expression coverage for two of the aforementioned purposes of code coverage. Such a weak criterion does not provide strong confidence in the correctness of tested code even if 100% coverage is achieved and it is certainly inappropriate to minimise a set of automatically generated test cases with respect to expression coverage because too many relevant tests would be considered redundant.

In this chapter we present more thorough criteria for control- and data-flow coverage in declarative programs (Sections 4.1 and 4.2). After introducing the different notions of coverage intuitively, we formally describe how to compute such information for functional logic programs (Section 4.3). Finally, we evaluate the different criteria experimentally (Section 4.4).

4.1 Control flow

Lazy declarative languages like Curry have no assignments and a rather complicated control-flow (due to laziness), which cannot easily be represented by a control-flow graph. Therefore, we cannot simply transfer the code coverage criteria from the imperative to the declarative world, but need adapted notions. In this section, we will present two different coverage criteria: *Rule Coverage* (Section 4.1.1) and *Call Coverage* (Section 4.1.2) which correspond both to variants of control-flow coverage in imperative languages.

In imperative languages, control-flow coverage describes which branches are taken in control structures like *if-then-else* statements. In declarative languages, control-flow is often expressed using pattern matching in multiple rules of function definitions. Hence, our control flow criteria consider the different rules of functions. Note that our criteria apply to *if-then-else* expressions which are just function calls in Haskell or Curry.

4.1.1 Rule coverage

Rule coverage describes which rules of which functions have been executed during a specific program run. The idea is to label the rules of each function uniquely and collect all labels that correspond to executed rules. If labels corresponding to all rules of the tested function and all rules of directly or indirectly called functions are collected during testing, then full rule coverage is achieved. Note that rule coverage is subsumed by expression coverage because each rule of a program is executed if and only if its right-hand side is demanded.

In glass-box testing for imperative languages, typically only code that is part of a single function or procedure declaration is considered. Due to control structures like loops present in imperative languages, there is often no need to consider more than one function to obtain interesting test cases. In declarative programming, (recursive) function calls are used to express control structures and to replace loops. Since the functions in a declarative program are typically very small and consist of a few lines only, it is not sufficient in practice to cover only the code of the function to be tested. Thus, we aim at covering the code of all the directly and indirectly called functions, too.

For example, a call to *main* defined in the introduction to this chapter executes every rule of the shown program. Bugs can hide easily in rules of a program despite testing with full rule coverage. If fact, all rules of the *reverse* and $+\!\!+$ functions are already executed by calling *reverse* $[1]$: the second rule of *reverse* is executed by the initial call, the recursive call to *reverse* executes the first rule, the call to $+\!\!+$ in *reverse* executes the second rule of $+\!\!+$, and the recursive call to $+\!\!+$ in the second rule of $+\!\!+$ executes the first rule of $+\!\!+$ in this example.

Rule coverage is similar to expressions coverage: easy to explain and visualise but insufficient for building confidence in tested code or eliminating redundant tests. It is even more basic than expression coverage as it does not describe coverage of sub-expressions.

4.1.2 Call coverage

Rule coverage has two disadvantages: it does not ensure thorough testing and the number of rules to be covered may be quite large if the tested function depends on many other functions. We can modify the notion of rule coverage to overcome both disadvantages with a novel coverage criterion, namely, call coverage. Full call coverage with respect to a function f is achieved if every reachable call to f executes every rule of f.

Call coverage ensures more thorough testing than simple rule coverage because every call to a function needs to execute every rule individually. It is not enough that all different calls in a program execute all rules of a function together. Moreover, the number of possible items to be covered is smaller than with simple rule coverage if we only aim at call coverage for the tested function f: the reachable calls of f can only be in f or in functions that are mutually recursive to f. Such functions are usually within a more limited fraction of the code than all reachable functions.

We can illustrate the benefits of call coverage by reconsidering the erroneous *reverse* implementation. The call to *main* does not cause full call coverage with respect to *reverse* because there are calls to *reverse* which do not execute every rule of *reverse*:

- the recursive call in the second rule of *reverse* only executes the first rule and
- each call to *reverse* in *test* executes only one rule of *reverse*.

We cannot achieve full call coverage with respect to *reverse* with a single call to *test*. The following calls to the *test* function form a minimal set of test cases that achieves full call coverage with respect to *reverse* and reveals the bug.

$$\begin{array}{l} \mathit{test}\ [\,]\ \ [\,] \\ \mathit{test}\ [1]\ [2] \end{array}$$

Using these tests, each call to *reverse* in *test* executes both rules and also the recursive call to *reverse* in the second rule of *reverse* executes not only the first but also the second rule because the third call to *reverse* in *test* is applied to a list with two elements by the second test.

It is not always possible to achieve full call coverage. For example, we cannot achieve full call coverage with respect to $+\!\!+$ in a program that calls the erroneous definition of *reverse*. The call to $+\!\!+$ in the second rule of *reverse* can never execute the first rule because $+\!\!+$ is applied to a non-empty list explicitly.

Moreover, even full call coverage does not guarantee the absence of errors. For example, the following set of calls to the *test* function also achieves full call coverage with respect to *reverse* but does not reveal the error.

$$\begin{array}{l} \mathit{test}\ [\,]\ \ [\,] \\ \mathit{test}\ [1]\ [\,] \\ \mathit{test}\ [\,]\ \ [1,2] \end{array}$$

Although it is less likely to miss a bug with full call coverage compared to rule coverage, it is not impossible which hints at the incomplete nature of every coverage criterion. In Section 4.4 we evaluate experimentally that call coverage works quite well to reveal bugs using practical example programs.

4.2 Data flow

Data-flow testing intends to provide a system of test cases which ensures that each computed value reaches every possible place, where it is used. In imperative languages, data-flow testing is usually based on covering all so-called def-use chains. A *def-use chain* is a triple consisting of a variable, a statement, where a value for this variable is computed (defined), and a statement, where this value is used and where this value has not been overwritten in between. In the following C code, for instance, the assignments in lines 1 and 4 form a def-use chain for variable `x`.

```
1 x = 1;
2 z = 2;
3 if (p()) {
4   y = x;
5 }
```

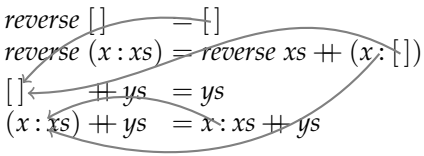
If `p()` never delivers `true` (which is undecidable in general), the def-use chain is purely syntactic, but cannot be passed by a test case and it is hence irrelevant for testing.

To the best of our knowledge there is no notion of data-flow coverage for (lazy) declarative languages. The imperative criteria cannot be simply transferred, since there are no assignments in declarative languages and laziness has to be taken into account. In this section, we propose a novel notion of data flow in declarative programs. We do not adapt traditional criteria from imperative programming languages that refer to definitions and uses of variables. Rather, our notion is based on algebraic data types and pattern matching – central abstraction mechanisms of declarative languages.

We define a declarative def-use chain as a pair of source code positions that correspond to the creation and elimination of data. There are two kinds of data that can flow through a declarative program: terms are created by constructors and eliminated by pattern matching and functions are created by partial applications or lambda abstractions and eliminated by applications. Hence, we regard constructors and partial applications or lambda abstractions as definitions, and patterns or applications as uses in a declarative def-use chain.

4.2.1 Constructors flow to patterns

Consider the following (correct) implementation of the naive reverse function in Haskell.



We use the notation $x : [\,]$ instead of $[x]$ in the second rule of *reverse* to be able to distinguish the different list constructors and indicate possible data flow as arrows from a constructor to a corresponding pattern. The function *reverse* introduces three constructors, namely $[\,]$ in the first and second rule and $(:)$ in the second rule. The function $+\!\!+$ also introduces the constructor $(:)$ in its second rule. It does not introduce new constructors in its first rule but yields the value given as second argument.

For a given expression, we are interested in the data flow caused by its lazy evaluation. We want to know, which values are matched against which patterns, that is, to which patterns introduced *data* constructors *flow*. For example, the expression *reverse* $[\,]$ causes the constructor $[\,]$ that is introduced in this expression to flow to the pattern in the first rule of *reverse* – in fact, it is matched immediately.

The expression *reverse* $(42 : [\,])$ causes a more complex data flow. The introduced constructor $(:)$ is again matched immediately by *reverse*. The number 42 is not matched during the execution and the constructor $[\,]$ is matched by *reverse* in the recursive call. However, there is more data flow in this execution. The result of the recursive call to *reverse* is matched by the function $+\!\!+$. During the execution of *reverse* $(42 : [\,])$ there is only one recursive call to *reverse* and its result is the constructor $[\,]$ introduced in the first rule of *reverse*. Therefore, this constructor flows to the pattern in the first rule of $+\!\!+$.

In the previous examples, we have observed the data flow that is caused by *specific* applications of *reverse*. If we are interested in the flow of data that can be caused by *any* call to *reverse*, we can apply *reverse* to a logic variable and observe which constructors in the program flow to which case expressions. The results are depicted in Table 4.1. We refer to the constructors introduced by *reverse* as `[]-rev1`, `[]-rev2` and `(:)-rev` and the constructor introduced by $+\!\!+$ as `(:)-app`. We refer to the patterns of the functions as `rev-[]`, `rev-(:)`, `app-[]`, and `app-(:)`.

Narrowed Call	Result	Def-Use Chains
reverse $[\,]$	$[\,]$	—
reverse $[a]$	$[a]$	`[]-rev1` $\rightarrow$ `app-[]`

reverse $[a,b]$	$[b,a]$	`[]-rev1` $\rightarrow$ `app-[]`
		`[]-rev2` $\rightarrow$ `app-[]`
		`(:)-rev` $\rightarrow$ `app-(:)`

reverse $[a,b,c]$	$[c,b,a]$	`[]-rev1` $\rightarrow$ `app-[]`
		`[]-rev2` $\rightarrow$ `app-[]`
		`(:)-rev` $\rightarrow$ `app-(:)`
		`(:)-app` $\rightarrow$ `app-(:)`

reverse $[a,b,c,d]$	$[d,c,b,a]$	`[]-rev1` $\rightarrow$ `app-[]`
		`[]-rev2` $\rightarrow$ `app-[]`
		`(:)-rev` $\rightarrow$ `app-(:)`
		`(:)-app` $\rightarrow$ `app-(:)`

Table 4.1: Data flow in applications of *reverse*.

An introduced constructor c and a pattern p where c is matched form a *def-use chain* denoted by $c \rightarrow p$. For def-use chains, only constructors introduced in the program are considered. Constructors that are part of the initial expression do not contribute to the data flow of a program. We can observe that none of the constructors introduced in the program is matched by *reverse* and all are matched by $+\!\!+$. It may be surprising that even the constructor $(:)$ introduced by $+\!\!+$ can be matched by $+\!\!+$ itself. The reason is that this $(:)$ is the result of a call to *reverse* with at least two arguments and, therefore, it flows to the first argument of $+\!\!+$ in a call to *reverse* with at least three arguments.

4.2.2 Functions flow to applications

Another source for data flow in declarative programs are higher-order functions. Partial applications and lambda abstractions flow through a program like constructed terms and, therefore, we treat them as data too. The difference is that functions are not matched in case expressions but *applied*. Hence, a def-use chain can also be a pair of a functional value and an occurrence of an application where the functional value is applied.

Consider a more efficient implementation of the reverse function.

$$
\begin{array}{ll}
\mathit{rev}\ l & = \mathit{foldl}\ (\mathit{flip}\ (:))\ [\,]\ l \\
\mathit{foldl}\ \_\ \ e\ [\,] & = e \\
\mathit{foldl}\ \mathit{op}\ e\ (x : \mathit{xs}) & = \mathit{foldl}\ \mathit{op}\ (\mathit{op}\ e\ x)\ \mathit{xs} \\
\mathit{flip}\ f\ y\ z & = f\ z\ y
\end{array}
$$

Apart from the constructors, the definition of *rev* introduces two partial applications – to *flip* and to $(:)$. When *rev* is applied to a non-empty list the partial application of *flip* to the argument $(:)$ flows to the application of *op* to e and x in the definition of *foldl*. The partial application of $(:)$ to zero arguments flows to the application of f to z and y in the definition of *flip*.

4.2.3 Comparison with Control Flow

Control Flow and Data Flow are complementary notions of code coverage. None of them subsumes the other. We will substantiate this claim by means of practical experiments in Section 4.4 but can also give an intuitive explanation by means of simple examples.

The identity function can be covered with respect to rule coverage (see Section 4.1.1) with one test case that executes it.

$$
\begin{array}{l}
\mathit{id} :: a \to a \\
\mathit{id}\ x = x
\end{array}
$$

Full dataflow coverage is already achieved with zero tests because it does not contain any construtors.

We have already seen an example of why call coverage (Section 4.1.2) is more difficult to achieve than rule coverage. The function *test* contains three different calls to *reverse* which need to be covered separately when considering call coverage whereas for rule coverage it would be enough if one call executes the first rule of *reverse* and another executes the second.

Call coverage subsumes rule coverage and is, therefore, not subsumed by data flow coverage, as the example with the identity function shows. That data flow coverage is also not subsumed by rule coverage is again apparent from the previous example: covering both rules in all calls to reverse, even in the recursive call, requires only a call to *reverse* with an argument of length two. However, to achieve full data-flow coverage, that is, to induce the flow of the $(:)$ constructor in $+\!\!+$ to the pattern of $+\!\!+$, requires a call to *reverse* with an argument of length three.

4.3 Monitoring code coverage

We have implemented a program transformation to collect coverage information during the execution of a declarative program.

4.3.1 Lazy coverage collection

A distinguishing property of our approach is that instrumented programs do not use side effects to collect information about the execution of the original program. The instrumented program computes, along with the original result, a set of *items* that are covered during the computation. Such items represent information about the control- or data-flow of the execution.

In lazy declarative languages expressions are only evaluated as much as demanded. If an expression is matched by a pattern, it is evaluated until the top-level constructor is known but its arguments may remain unevaluated. If a functional expression is applied to another expression, it is evaluated until its argument variable and body is known and then applied to the unevaluated argument. Partly evaluated expressions like constructors with possibly unevaluated arguments or functions with unevaluated bodies are called (weak) *head-normal forms*.

When implementing a program transformation for coverage collection in lazy declarative programs, we need to take laziness into account. The fact that we monitor the coverage that is induced by the evaluation of certain expressions must not influence the way in which those expressions are evaluated. The execution order of the instrumented program must reflect the execution order of the original program in order to not distort the results.

As an example, consider the following application of the *head* function to a result of calling $+\!\!+$.

$$\begin{aligned} & \mathit{head}\ ([1,2] +\!\!+ [3]) \\ = {} & \mathit{head}\ (1 : ([2] +\!\!+ [3])) \\ = {} & 1 \end{aligned}$$

During this computation, the call to $+\!\!+$ is only evaluated to head-normal form – the tail of the result, $[2] +\!\!+ [3]$, remains unevaluated. Consequently, the first rule of $+\!\!+$ which matches the empty list is not executed in order to compute the result 1 of the shown call to *head*.

In order to preserve the evaluation order of the original program and collect coverage information that corresponds to lazy computations, we attach a set of covered items to every computed constructor rather than to the complete result as a whole. As a result, coverage information that belongs to

unevaluated computations is not collected. For example, the covered items attached to the result of $[2] +\!\!+ [3]$ can be discarded in the instrumented version of the above call to *head*.

Figure 4.1 shows the lazy computation of *main* defined as follows[1]:

$$main = reverse\ [x, y]$$

For the sake of clarity, we only collect data-flow information in this example and omit control-flow coverage. We write constructors in prefix notation and give a subscript to every constructor in the program in order to distinguish different occurrences of the same constructor. The data flow of a computation is denoted by a superscript to corresponding expressions: a set of def-use chains that is attached to an expression denotes exactly those def-use chains that belong to the computation of its head-normal form.

After the unfolding of *main*, the initial call to *reverse* is unfolded. In this call, the first occurrence of the constructor $(:)$ in function *main* is matched by *reverse*. This data flow is denoted as $(:)_{main_1} \rightarrow rev$ and attached to the result of the unfolding. The next step is again an unfolding of *reverse*. In this call the second occurrence $(:)_{main_2}$ of $(:)$ in *main* is matched and we attach the corresponding data flow to the result of the unfolding. In its last recursive call, *reverse* is applied to the empty list $[]_{main}$ introduced in the right-hand side of *main*. The result of this call is the first occurrence $[]_{rev_1}$ of $[\,]$ in *reverse* and we attach the corresponding data flow $[]_{main} \rightarrow rev$. The empty list that was the result of the previous call is now matched in a call to the function $+\!\!+$. Note that the matched value $[]_{rev_1}$ has an attached def-use chain $[]_{main} \rightarrow rev$ and that we attach this def-use chain to the result of the call to $+\!\!+$. Additionally, the new def-use chain $[]_{rev_1} \rightarrow +\!\!+$ is recorded. The result of this call is again matched in a call to $+\!\!+$, and, again, we attach the coverage information of the matched value to the result of the call and record the new def-use chain $(:)_{rev} \rightarrow +\!\!+$. Now the computation to head-normal form is finished and the data flow caused by this computation is attached to the computed head-normal form. If we continue computing underneath the top-level constructor $(:)_{+\!\!+}$, there is one step left that unfolds the last call to $+\!\!+$. The data flow $[]_{rev} \rightarrow +\!\!+$ that is caused by this call is attached to the singleton list underneath the top-level constructor $(:)_{+\!\!+}$. Attaching coverage information to each sub term individually allows to collect data flow of lazy computations. If the tail of the resulting list, computed in Figure 4.1, would be discarded (for example, by projecting to the head) then the def-use chain $[]_{rev_2} \rightarrow +\!\!+$ would be discarded as well.

[1] In the definition of *main*, the variables x and y denote arbitrary values.

$$
\begin{aligned}
&\mathit{main} \to \mathit{reverse}\ ((:)_{\mathit{main}_1}\ x\ ((:)_{\mathit{main}_2}\ y\ []_{\mathit{main}})) \\
&\quad \to (\mathit{reverse}\ ((:)_{\mathit{main}_2}\ y\ []_{\mathit{main}}) +\!\!+ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_1} \to \mathit{rev}\}} \\
&\quad \to ((\mathit{reverse}\ []_{\mathit{main}} +\!\!+ ((:)_{\mathit{rev}}\ y\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_2} \to \mathit{rev}\}} +\!\!+ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_1} \to \mathit{rev}\}} \\
&\quad \to (([]_{\mathit{rev}_1}^{\{[]_{\mathit{main}} \to \mathit{rev}\}} +\!\!+ ((:)_{\mathit{rev}}\ y\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_2} \to \mathit{rev}\}} +\!\!+ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_1} \to \mathit{rev}\}} \\
&\quad \to (((:)_{\mathit{rev}}\ y\ []_{\mathit{rev}_2})^{\{(:)_{\mathit{main}_2} \to \mathit{rev},\ []_{\mathit{main}} \to \mathit{rev},\ []_{\mathit{rev}_1} \to +\!\!+\}} +\!\!+ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2}))^{\{(:)_{\mathit{main}_1} \to \mathit{rev}\}} \\
&\quad \to ((:)\ +\!\!+\ y\ ([]_{\mathit{rev}_2} +\!\!+ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2})))^{\{(:)_{\mathit{main}_1} \to \mathit{rev},\ (:)_{\mathit{main}_2} \to \mathit{rev},\ []_{\mathit{main}} \to \mathit{rev},\ []_{\mathit{rev}_1} \to +\!\!+,\ (:)_{\mathit{rev}} \to +\!\!+\}} \\
&\quad \to ((:)\ +\!\!+\ y\ ((:)_{\mathit{rev}}\ x\ []_{\mathit{rev}_2})^{\{[]_{\mathit{rev}_2} \to +\!\!+\}})^{\{(:)_{\mathit{main}_1} \to \mathit{rev},\ (:)_{\mathit{main}_2} \to \mathit{rev},\ []_{\mathit{main}} \to \mathit{rev},\ []_{\mathit{rev}_1} \to +\!\!+,\ (:)_{\mathit{rev}} \to +\!\!+\}}
\end{aligned}
$$

Figure 4.1: Computation of data flow for *reverse* $[x, y]$

4.3.2 Combinators for coverage collection

We now describe our program transformation by example. We introduce types to represent values with attached coverage information and combinators that operate on values of such types starting with simple versions for rule coverage that we subsequently refine to incorporate call coverage and data-flow coverage.

Wrapping constructors

We represent values with attached coverage information as values of the parametrised type $C\ a$. A number with attached coverage information has type $C\ Int$, a string with attached coverage information has type $C\ String$ and a list with elements of an arbitrary type a with attached coverage information has type $C\ [a]$. The implementation of the type C and its operations is presented in Section 4.3.4 – for now we confine ourselves to discuss its interface.

In order to construct nested values with attached coverage information, we can use the following operations.

$$cons :: a \to C\ a$$
$$(\circledast) :: C\ (a \to b) \to C\ a \to C\ b$$

The combinator *cons* wraps an arbitrary value into the C type attaching an empty set of coverage information, and the apply operation $\circledast$ applies a wrapped function or constructor to a wrapped argument. We only use $\circledast$ with constructors to construct nested values with attached coverage information. For example, a wrapped representation of $[42]$ is constructed by $cons\ (:) \circledast cons\ 42 \circledast cons\ [\,]$. The combinator $\circledast$ binds left associatively and we can decompose this nested application into individual parts to verify that it is type correct. The term $cons\ (:)$ has type $C\ (a \to [a] \to [a])$ and $cons\ 42$ has type $C\ Int$, hence, $cons\ (:) \circledast cons\ 42$ has type $C\ ([Int] \to [Int])$. The term $cons\ [\,]$ has type $C\ [a]$ and, thus, the nested application shown above has type $C\ [Int]$. The resulting value stores distinct sets of coverage information for each of the three constructors, $(:)$, 42, and $[\,]$.

Monitoring control flow

We use the correct implementation of the naive *reverse* function from Section 4.2.1 to illustrate how we collect coverage information. We start with rule coverage. In order to translate pattern matching, we need combinators

that allow us to match values wrapped with coverage information. The combinator $match :: C\ a \to (a \to C\ b) \to C\ b$ serves half of this purpose[2]: it takes a wrapped value to be matched, supplies the unwrapped value to the given function and yields the result of this call. The function given as second argument to *match* can match the original value without attached coverage information. It computes the result of type $C\ b$ using another combinator $rule_n$ that takes a representation of a rule as well as the arguments of the matched constructor, and supplies wrapped arguments to the given function.

$$rule_n :: RuleID \to C\ a \to a_1 \to ...\ a_n \to (C\ a_1 \to ...\ C\ a_n \to C\ b) \to C\ b$$

An instrumented version of the *reverse* function using these combinators can be defined as follows.

```
reverse :: C [a] → C [a]
reverse l = match l rules where
  rules []       = rule0 "rev-[]" l (cons [])
  rules (x' : xs') = rule2 "rev-(:)" l x' xs'
                     (λx xs → reverse xs ++ (cons (:) ⊛ x ⊛ cons []))
(++) :: C [a] → C [a] → C [a]
l ++ ys = match l rules where
  rules []       = rule0 "app-[]" l ys
  rules (x' : xs') = rule2 "app-(:)" l x' xs'
                     (λx xs → cons (:) ⊛ x ⊛ (xs ++ ys))
```

We have instrumented both the *reverse* function and the $+\!\!+$ function used by *reverse*. The types of both functions have changed to account for coverage information attached to arguments and results.

The *match* combinator is applied to the matched argument l and a function *rules* that performs the matching on the unwrapped argument in both functions. The right-hand side of *rules* is defined in terms of a $rule_n$ combinator[3] that takes a rule identifier, the matched value with attached coverage information, and the arguments of the unwrapped value without coverage information. The last argument to $rule_n$ is a function that takes wrapped representations of the arguments of the matched constructor and uses them in the transformed right-hand side of the original rule. The coverage information attached to the wrapped arguments x and xs is extracted from the wrapped representation of l and attached to the unwrapped arguments x' and xs' by the $rule_2$ combinator. In this example, we use strings as rule

[2] The fact that the type of *match* hints at monadic bind is no coincidence (cf. Chapter 5).

[3] We provide implementations of $rule_n$ for different n.

identifiers for simplicity. Our implementation uses a different representation. The $rule_n$ combinators add the given rule identifier to the coverage information attached to the head-normal form of their result to monitor the corresponding rule coverage.

In order to monitor call coverage, we add an additional call identifier to every function call. This argument is passed to the $rule_n$ combinators to monitor not only which rule has been executed but also to which call of the corresponding function it belongs. The instrumentation of the *reverse* function can be extended to monitor call coverage as follows. The only differences compared to the previous version are the additional argument *callID* and that $+\!\!+$ is written in prefix notation.

$$
\begin{aligned}
&\mathit{reverse} :: \mathit{CallID} \to C\ [a] \to C\ [a] \\
&\mathit{reverse}\ \mathit{callID}\ l = \ \mathit{match}\ l\ \mathit{rules}\ \mathbf{where} \\
&\quad \mathit{rules}\ [\,] \qquad = \mathit{rule}_0\ \mathit{callID}\ \texttt{"rev-[]"}\ l\ (\mathit{cons}\ [\,]) \\
&\quad \mathit{rules}\ (x' : xs') = \mathit{rule}_2\ \mathit{callID}\ \texttt{"rev-(:)"}\ l\ x'\ xs' \\
&\qquad\qquad\qquad (\lambda x\ xs \to (+\!\!+)\ \texttt{"app-rev"} \\
&\qquad\qquad\qquad\qquad (\mathit{reverse}\ \texttt{"rev-rev"}\ xs) \\
&\qquad\qquad\qquad\qquad (\mathit{cons}\ (:) \circledast x \circledast \mathit{cons}\ [\,]))
\end{aligned}
$$

$$
\begin{aligned}
&(+\!\!+) :: \mathit{CallID} \to C\ [a] \to C\ [a] \to C\ [a] \\
&(+\!\!+)\ \mathit{callID}\ l\ ys = \mathit{match}\ l\ \mathit{rules}\ \mathbf{where} \\
&\quad \mathit{rules}\ [\,] \qquad = \mathit{rule}_0\ \mathit{callID}\ \texttt{"app-[]"}\ l\ ys \\
&\quad \mathit{rules}\ (x' : xs') = \mathit{rule}_2\ \mathit{callID}\ \texttt{"app-(:)"}\ l\ x'\ xs' \\
&\qquad\qquad\qquad (\lambda x\ xs \to \mathit{cons}\ (:) \\
&\qquad\qquad\qquad\qquad \circledast x \circledast ((+\!\!+)\ \texttt{"app-app"}\ xs\ ys))
\end{aligned}
$$

Monitoring data flow

We now extend the presented approach such that not only control-flow coverage can be monitored but also data-flow coverage. The key extension is to not only label data constructors with coverage information but also with an identifier that uniquely represents the occurrence of a constructor in the source code. In order to label constructors, we change the *cons* combinator such that it takes an additional identifier.

$$\mathit{cons} :: \mathit{ConsID} \to a \to C\ a$$

This is the only extension that is necessary to support the collection of def-use chains from constructors to patterns: The $rule_n$ combinators can query the label of the wrapped matched argument and collect a def-use chain from this label to the given rule identifier.

Here is the final version of the instrumented *reverse* function that supports monitoring of both control-flow and data-flow coverage. The only difference compared to the previous version is the additional identifier in calls of the *cons* combinator.

$$
\begin{aligned}
&\mathit{reverse} :: \mathit{CallID} \to C\ [a] \to C\ [a] \\
&\mathit{reverse}\ \mathit{callID}\ l = \ \mathit{match}\ l\ \mathit{rules}\ \mathbf{where} \\
&\quad \mathit{rules}\ [\,] \qquad = \mathit{rule}_0\ \mathit{callID}\ \texttt{"rev-[]"}\ l\ (\mathit{cons}\ \texttt{"[]-rev1"}\ [\,]) \\
&\quad \mathit{rules}\ (x' : xs') = \mathit{rule}_2\ \mathit{callID}\ \texttt{"rev-(:)"}\ l\ x'\ xs' \\
&\qquad\qquad\qquad (\lambda x\ xs \to (+\!\!+)\ \texttt{"app-rev"} \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{reverse}\ \texttt{"rev-rev"}\ xs) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{cons}\ \texttt{"(:)-rev"}\ (:) \\
&\qquad\qquad\qquad\qquad\qquad\ \circledast x \circledast \mathit{cons}\ \texttt{"[]-rev2"}\ [\,]))) \\
&(+\!\!+) :: \mathit{CallID} \to C\ [a] \to C\ [a] \to C\ [a] \\
&(+\!\!+)\ \mathit{callID}\ l\ ys = \mathit{match}\ l\ \mathit{rules}\ \mathbf{where} \\
&\quad \mathit{rules}\ [\,] \qquad = \mathit{rule}_0\ \mathit{callID}\ \texttt{"app-[]"}\ l\ ys \\
&\quad \mathit{rules}\ (x' : xs') = \mathit{rule}_2\ \mathit{callID}\ \texttt{"app-(:)"}\ l\ x'\ xs' \\
&\qquad\qquad\qquad (\lambda x\ xs \to \mathit{cons}\ \texttt{"(:)-app"}\ (:) \\
&\qquad\qquad\qquad\qquad\qquad \circledast x \circledast ((+\!\!+)\ \texttt{"app-app"}\ xs\ ys)
\end{aligned}
$$

This version of the instrumented *reverse* function can be used to monitor the def-use chains depicted in Table 4.1.

As final example for an instrumented program, we consider the efficient reverse function defined in terms of higher-order functions. We provide an additional combinator to transform applications of higher-order functions:

$$\mathit{apply} :: \mathit{AppID} \to C\ (C\ a \to C\ b) \to C\ a \to C\ b$$

As we consider functions as data that flows through a program they are also wrapped in the C type and have, hence, associated coverage information and an identifier that represents their source code location.

Partial applications are transformed into possibly nested applications of the *cons* combinator to lambda abstractions. For example, the call $\mathit{flip}\ f$ can be translated as follows:

$$\mathit{cons}\ i_1\ (\lambda x \to \mathit{cons}\ i_2\ (\lambda y \to \mathit{flip}\ i\ f\ x\ y))$$

If the original expression $\mathit{flip}\ f$ is of type $a \to b \to c$ then the instrumented version shown above, which uses instrumented versions of *flip* and f, is of type $C\ (C\ a \to C\ (C\ b \to C\ c))$. The identifiers i_1 and i_2 represent the positions of the different lambda abstractions and can be the definition part in def-use chains to the position of an application where the corresponding

lambda abstraction is applied. The two lambda abstractions can be applied in completely different program positions, hence, we distinguish them in order to monitor separate def-use chains for each application. The identifier i represents the position of *flip* and is used to monitor call coverage.

Partial applications of constructors need to be handled differently because constructors are not instrumented like functions and do not take wrapped arguments. In order to transform a partial constructor application, we use additional calls to the combinator $\circledast$. For example, the partial application of the $(:)$ constructor to zero arguments can be transformed as follows:

$$cons\ i_1\ (\lambda x \to cons\ i_2\ (\lambda xs \to cons\ i\ (:) \circledast x \circledast xs))$$

The original expression $(:)$ has type $a \to [a] \to [a]$ and the instrumented version shown above has the type $C\ (C\ a \to C\ (C\ [a] \to C\ [a]))$. The identifiers i_1 and i_2 can be recorded in def-use chains to applications while the identifier i will be recorded in a def-use chain to every pattern that matches this occurrence of $(:)$.

We are now ready to instrument the definition of the reverse function shown in Section 4.2.2.

```
rev :: CallID → C [a] → C [a]
rev callID l = rule callID "rev"
                 (foldl "foldl-rev" pflipcons (cons "[]-rev" []) l)
  where pflipcons = cons "flip-rev1"
                         (λx → cons "flip-rev2"
                                    (λy → flip "flip-rev" pcons x y))
        pcons     = cons "(:)-rev1"
                         (λx → cons "(:)-rev2"
                                    (λxs → cons "(:)-rev" (:) ⊛ x ⊛ xs))
```

The function *rule* is a variant of $rule_0$ that does not take a matched argument and is used to instrument rules without pattern matching. We have introduced local declarations for the involved partial applications to increase readability. Our implementation would generate code as if the occurrences of *pflipcons* and *pcons* would have been substituted by their right-hand sides. Finally, we show how to instrument the definitions of *foldl* and *flip*.

$$
\begin{aligned}
&\mathit{foldl} :: \mathit{CallID} \to C\ (C\ b \to C\ (C\ a \to C\ b)) \to C\ b \to C\ [a] \to C\ b\\
&\mathit{foldl}\ \mathit{callID}\ \mathit{op}\ e\ l = \mathit{match}\ l\ \mathit{rules}\ \textbf{where}\\
&\quad \mathit{rules}\ [\,] \quad\ = \mathit{rule}_0\ \mathit{callID}\ \texttt{"foldl-[]"}\ l\ e\\
&\quad \mathit{rules}\ (x' : xs') = \mathit{rule}_2\ \mathit{callID}\ \texttt{"foldl-(:)"}\ l\ x'\ xs'\\
&\qquad\qquad (\lambda x\ xs \to \mathit{foldl}\ \texttt{"foldl-foldl"}\ \mathit{op}\\
&\qquad\qquad\qquad (\mathit{apply}\ \texttt{"foldl-op2"}\\
&\qquad\qquad\qquad\quad (\mathit{apply}\ \texttt{"foldl-op1"}\ \mathit{op}\ e)\ x)\\
&\qquad\qquad\quad xs)
\end{aligned}
$$

The instrumented version of *foldl* uses the *match* combinator for pattern matching and the *apply* combinator to apply the given wrapped function *op* to the two wrapped arguments e and x. The instrumented version of *flip* also uses *apply* to apply the given wrapped function to the given wrapped arguments.

$$
\begin{aligned}
&\mathit{flip} :: \mathit{CallID} \to C\ (C\ a \to C\ (C\ b \to C\ c)) \to C\ b \to C\ a \to C\ c\\
&\mathit{flip}\ \mathit{callID}\ f\ y\ z = \mathit{rule}\ \mathit{callID}\ \texttt{"flip"}\\
&\qquad\qquad\qquad (\mathit{apply}\ \texttt{"flip-f2"}\ (\mathit{apply}\ \texttt{"flip-f1"}\ f\ z)\ y)
\end{aligned}
$$

The reason why we need the *apply* combinator for some applications (for example, those in *flip*) but not for others (for example, the applications of *foldl*) will become apparent in the following section where we formalise our transformation. Intuitively, *foldl* cannot be applied using *apply* as it is not wrapped inside the C type, hence, using *apply* would be a type error.

4.3.3 Formalisation of program transformation

We formalise our program transformation in terms of a simplified *core* representation of Curry programs. Every Curry program can be translated into an equivalent core program with considerably simplified syntax. For example, functions are defined by a single rule and pattern matching is expressed using sequences of flat case expressions where patterns cannot be nested.

Such a restrictive representation of programs is inconvenient for programmers but well suited for tools that operate on programs because programs are much simpler, and, hence, fewer special cases have to be considered. The disadvantage of using a simplified core language is that it is sometimes difficult to relate the code to the original source code. Although a simple core language simplifies the implementation of an analysis tool, it may complicate the presentation of analysis results to the user.

The syntax of core programs is depicted in Figure 4.2. A core program is a sequence of equations that define operations. Each definition consists of a left- and a right-hand side where the left-hand side is an application of

Program	P	$::=$	$D_1 \ldots D_m$	
Definition	D	$::=$	$f\ x_1\ \ldots\ x_n = e$	
Expression	e	$::=$	x	(variable)
		$\mid$	c	(constructor)
		$\mid$	f	(defined operation)
		$\mid$	$e_1\ e_2$	(application)
		$\mid$	$\lambda\ x\ \rightarrow\ e$	(abstraction)
		$\mid$	$let\ \{x_1/e_1; \ldots; x_k/e_k\}\ in\ e$	(local declarations)
		$\mid$	$case\ x\ of\ \{p_1 \rightarrow e_1; \ldots; p_k \rightarrow e_k\}$	(case distinction)
		$\mid$	$e_1\ or\ e_2$	(nondeterminism)
Pattern	p	$::=$	$c\ x_1\ \ldots\ x_n$	

Figure 4.2: Syntax of core Curry programs

the defined operation to variables and the right-hand side is an expression. The number of arguments is called the *arity* of the defined operation. There are different kinds of expressions. We denote variables using the letters x and y possibly with subscripts, constructors are denoted by c, and defined operations by f. As usual, application binds left associatively, that is, $f\ x\ y$ means $(f\ x)\ y$, and we occasionally write lambda abstractions with more than one argument instead of nested lambda abstractions. In addition to syntactic constructs for abstraction, local declarations, and case distinction, we also provide nondeterministic choice to model functional logic programming. Justified by the observations in Section 2.2.3, we do not consider unbound logic variables which can be simulated using nondeterminism. Note that arguments of case expressions are always variables and patterns in case expressions are flat which means that all arguments of matched constructors are also variables. In patterns the number of arguments must match the arity of the constructor c which we assume is given implicitly.

Transforming first-order programs

In a first step, we assume that there are no lambda abstractions and all applications apply defined operations or constructors according to their arity. In a second step, we will drop these assumptions and describe the transformation of lambda abstractions and arbitrary applications.

Each defined operation of a program is transformed into another operation by applying a mapping τ to its definition. The transformation adds an additional argument y for the call position, a call to the *rule* combinator, and transforms the right-hand side of the operation recursively.

$$\tau(f\ x_1\ \ldots\ x_n = e) = (f\ y\ x_1\ \ldots\ x_n = rule\ y\ \boxed{\text{f}}\ \tau(e))$$

In this transformation, y denotes a fresh variable name and $\boxed{\text{f}}$ is a unique identifier that represents the transformed rule. This definition of the transformation τ adds more calls to *rule* combinators than necessary. In the examples in Section 4.3.2, we have introduced them only after pattern matching and our transformation also avoids nested calls to *rule* combinators. We do not discuss the technical details which would complicate the presentation unnecessarily.

According to the transformation of right-hand sides, we have already seen that the most interesting part—at least for first-order programs—is concerned with constructor applications and case expressions. In fact, everything else is left unchanged:

$$\begin{aligned}
\tau(x) &= x && (x \text{ variable})\\
\tau(e_1\ or\ e_2) &= \tau(e_1)\ or\ \tau(e_2)\\
\tau(f\ e_1\ \ldots\ e_k) &= f\ \boxed{\text{f}}\ \tau(e_1)\ \ldots\ \tau(e_k) && (f \text{ defined operation of arity } k)
\end{aligned}$$

$$\tau(let\ \{x_1/e_1;\ldots;x_k/e_k\}\ in\ e) = let\ \{x_1/\tau(e_1);\ldots;x_k/\tau(e_k)\}\ in\ \tau(e)$$

Here, $\boxed{\text{f}}$ denotes a unique identifier that represents the call position of the operation f. Constructor applications are transformed using the auxiliary functions *cons* and $\circledast$:

$$\tau(c\ e_1\ \ldots\ e_k) = cons\ \boxed{\text{c}}\ c \circledast \tau(e_1) \ldots \circledast \tau(e_k)$$

Note that a constructor c without arguments is just represented as $cons\ \boxed{\text{c}}\ c$. Here, $\boxed{\text{c}}$ denotes a unique identifier of this occurrence of the constructor c.

The transformation of case expressions introduces calls to the auxiliary functions *match* and $rule_n$.

$$\begin{aligned}
&\tau(case\ x\ of\ \{\ldots(c_i\ x_{i_1}\ \ldots\ x_{i_n}) \to e_i;\ldots\}) =\\
&\quad match\ x\ (\lambda\ x'\ \to\\
&\quad case\ x'\ of\{\ldots\\
&\qquad c_i\ x'_{i_1}\ \ldots\ x'_{i_n}\ \to\\
&\qquad\quad rule_n\ \boxed{\text{r}}\ x\ x'_{i_1}\ \ldots\ x'_{i_n}\ (\lambda\ x_{i_1}\ \ldots\ x_{i_n}\ \to\ \tau(e_i));\\
&\qquad \ldots\})
\end{aligned}$$

Here, $x', x'_{i_1}, \ldots, x'_{i_n}$ are fresh variable names. The transformation of a case expression applies the auxiliary function *match* to the matched variable x and a lambda expression that introduces a fresh variable x'. In the original case expression all pattern variables x_{i_k} are renamed to fresh variables x'_{i_k} and passed—along with a unique identifier $\boxed{\mathrm{r}}$ and the matched variable x—to the auxiliary function $rule_n$. The original pattern variables $x_{i_1}, \ldots, x_{i_n}$ are used as variables in the lambda expression that is also passed to $rule_n$. Hence, we do not need to rename variables in the right-hand sides e_i of the case branches.

Transforming higher-order programs

The transformation described so far assumes a first-order program, that is, there are no lambda abstractions and all defined operations and constructors are applied according to their arity. We now describe the transformation of lambda abstractions and arbitrary applications. We assume that there are no partial applications of defined operations or constructors. Partial applications (where the number of supplied arguments is less than the arity of the applied operation or constructor) can be eliminated using lambda abstractions. For example, if f is a defined operation of arity two, then the partial application $f\ e$ can be replaced by $(\lambda x \to \lambda y \to f\ x\ y)\ e$ where x and y are fresh variable names. Note, that we do not use the simpler replacement $\lambda y \to f\ e\ y$ which has a different semantics in Curry because e is not shared if the lambda abstraction is duplicated.

Lambda abstractions are definitions that may be part of data flow and, hence, transformed using the combinator *cons*.

$$\tau(\lambda\ x\ \to\ e) = cons\ \boxed{\lambda}\ (\lambda\ x\ \to\ \tau(e))$$

Here, $\boxed{\lambda}$ denotes a unique identifier for this lambda abstraction. Note that if the original abstraction has type $a \to b$ then the transformed abstraction has type $C\ (C\ a \to C\ b')$ where $C\ b'$ is the type of the transformed body of the abstraction. Transformed lambda abstractions need to be applied with the *apply* combinator which is introduced when transforming applications.

An expression of the form $e_1\ e_2$ is transformed using the auxiliary function *apply* if it is not part of an application of a defined operation or constructor according to its arity:

$$\tau(e_1\ e_2) = apply\ \boxed{\mathrm{apply}}\ \tau(e_1)\ \tau(e_2)$$

Here, $\boxed{\mathrm{apply}}$ is a unique identifier for the transformed application.

4.3.4 Implementation of coverage combinators

The program transformation presented in the previous subsections introduces calls to auxiliary functions that operate on values of type $C\ a$. In this subsection we discuss the implementation of all used data types and functions.

The type $C\ a$ is defined as follows:

$$\begin{aligned}&\textbf{data}\ C\ a\ = C\ \mathit{Info}\ a\\&\textbf{data}\ \mathit{Info} = \mathit{Info}\ \mathit{ConsID}\ \mathit{Coverage}\ [\mathit{Info}]\end{aligned}$$

The type $C\ a$ associates an unranked tree of type *Info* to a value of type a. The structure of this tree reflects the structure of the corresponding data term, and, hence, associates an identifier and a set of covered items to each constructor. If the original value is a function, the associated tree has no children.

We assume an abstract data type *Coverage* with the following interface:

$$\begin{aligned}&\mathit{noCoverage} &&:: \mathit{Coverage}\\&\mathit{mergeCoverage} &&:: \mathit{Coverage} \to \mathit{Coverage} \to \mathit{Coverage}\\&\mathit{addControlFlow} &&:: \mathit{CallID} \to \mathit{RuleID} \to \mathit{Coverage} \to \mathit{Coverage}\\&\mathit{addDataFlow} &&:: \mathit{ConsID} \to \mathit{RuleID} \to \mathit{Coverage} \to \mathit{Coverage}\end{aligned}$$

We do not discuss the implementation of this interface. It is straightforward to implement a set of coverage items using efficient, purely functional implementations of map data structures.

To understand the correspondence between the structure of a value and its associated tree of information we discuss the implementation of the functions *cons* and $\circledast$ first. The function *cons* attaches a tree with the given identifier, no coverage, and no children to the given value.

$$\begin{aligned}&\mathit{cons} :: \mathit{ConsID} \to a \to C\ a\\&\mathit{cons}\ i\ x = C\ (\mathit{Info}\ i\ \mathit{noCoverage}\ [\,])\ x\end{aligned}$$

The identifier i uniquely identifies the constructor x. The function $\circledast$ is used to apply a wrapped constructor and build complex terms with attached information:

$$\begin{aligned}&(\circledast) :: C\ (a \to b) \to C\ a \to C\ b\\&cf \circledast cx = C\ (\mathit{Info}\ i\ c\ (t : ts))\ (f\ x)\\&\quad \textbf{where}\ C\ (\mathit{Info}\ i\ c\ ts)\ f = cf\\&\qquad\quad\ \ C\ t\ x\qquad\qquad\ = cx\end{aligned}$$

The wrapped constructor and argument are unwrapped by pattern matching in a local declaration. Hence, the arguments of $\circledast$ are evaluated only on demand which is important to maintain the laziness of the original program. The result of $\circledast$ is constructed from the application of the constructor to its argument and a new tree of type *Info*. The new tree is identical to the tree that was associated to the constructor before but has an additional child tree – the one that was associated to the argument. We extend the list of child trees at the head to achieve constant run time. Hence, the trees of information that correspond to the arguments of a constructor appear in reverse order.

As an example for a complex term with associated information consider the following expression:

```
cons "c1" (:) ⊛ cons "t" True
              ⊛ (cons "c2" (:) ⊛ cons "f" False
                               ⊛ cons "n" [])
```

This expression is equivalent to:

```
C (Info "c1" noCoverage
        [Info "c2" noCoverage
              [Info "n" noCoverage []
              ,Info "f" noCoverage []]
        ,Info "t" noCoverage []])
  [True,False]
```

The shown tree of type *Info* reflects the structure of the corresponding value ((:) *True* ((:) *False* [])), which is [*True*,*False*], while arguments appear in reverse order. The unique identifier for each constructor can be found at a corresponding position in the tree, and, initially, no coverage information has been collected.

Every case expression is transformed using the function *match* which is defined as follows.

```
match :: C a → (a → C b) → C b
match (C (Info _ cx _) x) f = cover (f x)
  where cover (C (Info i cy ts) y) =
              C (Info i (mergeCoverage cx cy)) ts) y
```

The function *match* selects the originally matched value x from its wrapped representation and provides it as an argument to the function f. The coverage information associated with the result of this call is updated to include

the coverage associated with the top-most constructor of the matched value. We need to collect the coverage of the top-most constructor because its evaluation is demanded by the case expression for the computation of the result. The function *match* lies at the heart of our program transformation because this implementation carefully merges exactly the coverage information of a lazy computation without modifying the laziness of the original program.

A careful reader might have noticed that the *match* function ignores some of the information that is associated with the matched argument. The identifier of the constructor and the information associated to its arguments are discarded. This information is not redundant but used by a combinator $rule_n$ in each branch of a case expression. Here, n is the arity of the constructor matched in the corresponding rule. As an example, we consider the implementation of $rule_2$:

$$
\begin{aligned}
&rule_2 \;::\; CallID \to RuleID \\
&\quad \to C\ a \to a1 \to a2 \to (C\ a1 \to C\ a2 \to C\ b) \to C\ b \\
&rule_2\ callID\ ruleID\ (C\ (Info\ consID\ \_\ [t2,t1])\ \_)\ x1\ x2\ f \\
&\quad = cover\ (f\ (C\ t1\ x1)\ (C\ t2\ x2)) \\
&\quad \textbf{where}\ cover\ (C\ (Info\ i\ c\ ts)\ y) = \\
&\qquad\qquad C\ (Info\ i\ (addControlFlow\ callID\ ruleID \\
&\qquad\qquad\qquad (addDataFlow\ consID\ ruleID\ c))\ ts)\ y
\end{aligned}
$$

This definition selects the information stored in the sub trees associated to the matched value and associates it to the corresponding unwrapped arguments. Note that the argument trees are stored in reverse order in the tree that is associated to the matched value. Finally, control- and data-flow coverage is associated to the head-normal form of the result.

We now turn to higher-order programs. The *apply* combinator is used to apply a wrapped partial application to another argument:

$$
\begin{aligned}
&apply :: AppID \to C\ (C\ a \to C\ b) \to C\ a \to C\ b \\
&apply\ appID\ fun\ arg = C\ (Info\ i\ (addDataFlow\ funID\ appID \\
&\qquad\qquad (mergeCoverage\ cf\ cres))\ ts)\ res \\
&\quad \textbf{where}\ C\ (Info\ funID\ cf\ [\,])\ f\ = fun \\
&\qquad\quad C\ (Info\ i\ cres\ ts)\ res\ \ = f\ arg
\end{aligned}
$$

This combinator selects the partially applied function from its wrapped representation and applies it to the wrapped argument. The coverage information of the result is updated twice:

- the coverage associated with the partial application is added to the coverage of the result, and

- a def-use chain from the partial application to the position where it is applied is included.

The coverage information associated with the partial application needs to be added to the coverage information of the result because an application demands the evaluation of its left argument. We treat partial applications as *data* and consider the flow of a partially applied function or constructor to its application as data flow.

4.4 Experimental evaluation

We have implemented the described program transformation in a prototypical tool for coverage-based test-case generation. The tool uses narrowing to bind initially uninstantiated input of a tested operation as described in Chapter 3. Alternatively, the user can provide custom input generators as nondeterministic generators (see Section 2.2.3).

We now investigate how coverage-based testing behaves for example programs that resemble those presented in Section 3.2.3. The considered examples all contain non-trivial algorithms which makes it hard to cover every relevant behavior. Coverage-based testing shows which items corresponding to the considered criterion are covered by a generated test suite.

Our experiments have been performed on a VMware virtual machine with 700 MB memory running Suse Linux 10.2 on a machine with a 3.6 GHz Xeon processor. We have compared control-flow coverage, data-flow coverage, and a combined control- and data-flow coverage. The results of our experiments are depicted in Table 4.2. We refer to control-flow coverage as CFC, to data-flow coverage as DFC, and to combined control- and data-flow coverage as CDFC.

For each of these three *coverage approaches*, we have investigated two *coverage scopes*. The first of them, *Module*, only considers coverable items inside the considered module. The second, *Global*, also considers control- and/or data-flow between different modules and inside auxiliary modules. In the following, a combination of coverage approach and coverage scope will be called *coverage criterion*.

We have used *breadth-first search* (BF) as a search strategy for exploring the search space in most experiments. The search is stopped, if n test cases have been generated without covering any new items with respect to to the considered coverage criterion. If n is chosen sufficiently large, one can be relatively sure that all coverable items will be covered and that the generated test cases really achieve the desired coverage. In most of our experiments we

have used $n = 200$. This is often a conservative choice. With smaller n, the run time can be reduced considerably while increasing the risk of missing a coverable item.

Fixing n as above only makes sense with breadth-first search or similar fair strategies that ensure that the progress in exploring the search space is distributed over the whole breadth of the tree. It would not make sense with depth-first search. Breadth-first search is sufficient if there is enough memory. In some cases we have experienced memory problems. For example, for Kruskal with CDFC/*Global*. In such cases, one can reduce n or use a less space-consuming search strategy such as iterative-deepening depth-first search.

Our tool usually generates a large number of test cases nondeterministically and then minimises the set of generated tests according a coverage criterion. We compare for different criteria how well they preserve failing tests. In our experiments, the introduced bugs are always exposed by a generated test case. Whether a test case that exposes the error remains after eliminating tests with redundant coverage often depends on the employed coverage criterion.

Table 4.2 shows for each considered example application and coverage criterion:

- which search strategy has been used,
- whether the error is exposed by a test case in the minimal covering test suite (+/-),
- whether the error is exposed by a test case before redundancy elimination (+/-, in fact always +),
- how many test cases are necessary in order to fulfill the considered coverage criterion,
- how many items (control- or data-flows) have been covered,
- how long the search has taken (in seconds),
- and how long the elimination of redundant test-cases has taken (in seconds).

problem	coverage criterion	search strategy	error covered	error reached	# test cases	# items covered	search time (s)	elimination time (s)
AVL insert	CFC/Module	BF200	+	+	5	25	17.8	0.5
	CFC/Global	BF200	+	+	9	78	21.8	3.6
	DFC/Module	BF200	+	+	7	51	29.3	2.0
	DFC/Global	BF200	+	+	12	165	33.8	10.9
	CDFC/Module	BF200	+	+	8	45+51	32.0	4.4
	CDFC/Global	BF200	+	+	20	165+165	45.0	26.6
AVL delete	CFC/Module	BF200	-	+	9	41	29.2	1.2
	CFC/Global	BF200	-	+	9	88	33.3	4.4
	DFC/Module	BF200	+	+	19	90	47.6	2.9
	DFC/Global	BF200	+	+	23	223	57.5	16.2
	CDFC/Module	BF200	+	+	24	77+90	53.8	6.7
	CDFC/Global	BF200	+	+	27	204+223	70.5	36.8
Heap insert	CFC/Module	BF200	+	+	2	9	2.8	0.0
	CFC/Global	BF200	+	+	8	38	4.8	0.5
	DFC/Module	BF200	+	+	2	10	2.7	0.1
	DFC/Global	BF200	+	+	7	25	4.1	0.3
	CDFC/Module	BF200	+	+	5	17+10	4.3	0.4
	CDFC/Global	BF200	+	+	23	72+25	8.6	1.7
Heap delete	CFC/Module	BF200	+	+	2	9	1.1	0.0
	CFC/Global	BF200	+	+	8	38	3.2	0.3
	DFC/Module	BF200	+	+	2	6	1.0	0.0
	DFC/Global	BF200	+	+	10	25	3.2	0.1
	CDFC/Module	BF200	+	+	4	14+6	1.2	0.1
	CDFC/Global	BF200	+	+	24	76+25	4.5	1.2
Heapsort	CFC/Module	BF200	+	+	1	15	60.2	0.5
	CFC/Global	BF200	+	+	6	47	74.0	1.7
	DFC/Module	BF200	+	+	2	27	104.9	1.5
	DFC/Global	BF200	+	+	7	59	106.3	4.0
	CDFC/Module	BF200	+	+	5	31+27	243.8	6.4
	CDFC/Global	BF200	+	+	16	90+63	234.2	17.5
Strassen	CFC/Module	BF50	+	+	1	12	45.8	0.2
	CFC/Global	BF50	+	+	4	94	102.8	8.4
	DFC/Module	BF50	+	+	1	6	46.8	0.0
	DFC/Global	BF50	+	+	13	240	170.3	31.7
	CDFC/Module	BF50	+	+	2	35+6	44.7	0.7
	CDFC/Global	BF50	+	+	18	233+240	198.7	74.0

problem	coverage criterion	search strategy	error covered	error reached	# test cases	# items covered	search time (s)	elimination time (s)
Dijkstra	CFC/Module	BF200	-	+	1	7	39.0	0.2
	CFC/Global	BF200	+	+	6	110	101.6	13.7
	DFC/Module	BF200	-	+	2	7	37.4	0.2
	DFC/Global	BF200	+	+	9	192	119.6	23.3
	CDFC/Module	BF200	-	+	2	8+7	39.6	0.4
	CDFC/Global	BF200	+	+	10	202+192	140.4	62.2
Kruskal	CFC/Module	BF200	-	+	1	19	1823.0	0.8
	CFC/Global	BF200	-	+	4	161	2709.0	39.1
	DFC/Module	BF200	-	+	1	18	1782.7	0.7
	DFC/Global	BF200	+	+	9	370	2737.4	127.7
	CDFC/Module	BF200	-	+	1	22+18	1796.5	2.3
	CDFC/Global	BF100	+	+	10	328+370	2539.1	224.6
Matrix CM	CFC/Module	BF200	+	+	1	9	17.4	0.1
	CFC/Global	BF200	+	+	2	125	33.8	17.5
	DFC/Module	BF200	+	+	1	3	17.4	0.0
	DFC/Global	BF200	+	+	7	596	112.9	364.2
	CDFC/Module	BF200	+	+	1	9+3	18.3	0.2
	CDFC/Global	BF190	+	+	4	284+555	73.5	257.3
Train Ticket	CFC/Module	BF100	+	+	3	19	50.0	0.4
	CFC/Global	BF100	+	+	3	94	57.0	13.2
	DFC/Module	BF100	+	+	4	13	56.7	0.2
	DFC/Global	BF100	+	+	7	445	130.7	193.3
	CDFC/Module	BF100	+	+	5	22+13	68.2	1.1
	CDFC/Global	BF100	+	+	9	248+445	194.2	322.1

Table 4.2: Experimental results of coverage-based testing

	Control Flow	Data Flow	Control + Data Flow
AVL insert	9	12	20
AVL delete	9	23	27
Heap insert	8	7	23
Heap delete	8	10	24
Heapsort	6	7	16
Strassen	4	13	18
Dijkstra	6	9	10
Kruskal	4	9	10
Matrix CM	2	7	4
Train Ticket	3	7	9

Table 4.3: Sizes of reduced sets of tests for different coverage criteria

Generating test input on demand does not ensure a minimal set of test cases. In order to get a minimal set, we need an additional step which removes redundant tests. Obviously, this problem is the set covering problem which is known to be NP-complete (Cormen et al. 1990). Since it is not essential in our context that we really find a minimal solution, we are happy with any heuristic that produces a small solution.

If the error is inside the search space, this means that our tool has considered a test case exposing the error. However, it may happen that this test case is deleted during the elimination of redundant test cases, since the considered coverage criterion does not ensure it to be preserved. Except for the coverage criterion, the tool has no means to tell which test cases are relevant.

As our experiments show and as one might expect, CDFC reaches a better coverage than CFC and DFC, while needing more time (up to a factor of roughly two compared to CFC) and space. If using coverage scope *Global*, CDFC is able to cover every error in our example applications. With coverage scope *Module*, CDFC could find all errors except those in the Dijkstra and Kruskal algorithms.

CFC does not cover the error in the Kruskal algorithm. In general, DFC requires in most cases more time (up to a factor of roughly two) and space than CFC. Interestingly, DFC/*Global* is able to expose all introduced errors.

Coverage scope *Module* is often insufficient. If the control- and data-flows of auxiliary modules are taken into account, far more errors can be exposed.

Table 4.4 summarizes the benchmark results for the coverage scope *Global*. It shows for each example the number of tests that remains after eliminating redundant tests with respect to the different coverage criteria.

Although data-flow coverage exposes all errors in our examples, the table shows that additionally requiring control-flow coverage leads to more thorough testing. Almost always is the number of required tests for combined control- and data-flow coverage larger than the number of required tests for data-flow coverage alone. Only for matrix chain multiplication were actually *fewer* tests necessary to achieve control- and data-flow coverage as were necessary to achieve data-flow coverage only. The reason is that our tool generated larger tests when aiming at combined coverage and the elimination program then picked few large test cases instead many small ones.

The run time of our tool is in almost all cases good or acceptable, namely a few seconds or minutes. Only in the Kruskal example, almost half an hour was needed. But even this is acceptable taking into account that the generation of test cases does not have to be repeated often and that it can be performed in the background while doing other work. As a result of our experiments, we recommend the coverage criterion CDFC/*Global*. It produces the best coverage and exposes most errors, while the overhead compared to other criteria is acceptable.

4.5 Summary

We have developed a tool that generates a minimal set of test cases that satisfy different coverage criteria. We have introduced novel notions of control- and data-flow coverage for declarative programs to improve the quality of test cases selected by our tool. We have presented a program transformation that instruments declarative programs such that they compute coverage according to their execution along with the original result. The generated program is a pure declarative program – it does not use side effects to monitor coverage.

Coverage-based testing is especially useful for algorithmically complex program units. We have selected a couple of such examples and compared the quality of selected tests using different coverage criteria. For this purpose, we have introduced a bug in each example and checked whether this bug was exposed by the generated minimal set of test cases. It turns out that most thorough testing is achieved by a combination of control and data flow. During our experiments, our tool has exposed a subtle error in the description of Kruskal's minimum spanning tree algorithm given in a textbook by Rabhi and Lapalme (1999).

4.6 Chapter notes

Work on the code coverage criteria presented in this chapter has been published previously. The presented control-flow coverage criteria *rule coverage* and *call coverage* have been published in the Proceedings of the ninth ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (Fischer and Kuchen 2007) under different names: previously, rule coverage has been called branch coverage and call coverage has been called function coverage. In hindsight, we prefer to use the term *rule* (of an operation) rather than *branch* (in a control-flow graph) and emphasise that different *calls* of the same *function* need to be covered individually.

Our data-flow coverage criteria have been published in the Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Fischer and Kuchen 2008). The program transformation presented in Section 4.3 is an extended version of the transformation presented previously that incorporates control-flow coverage and is applicable to functional logic rather than only functional programs. Unlike Albert et al. (2005), we also incorporate lambda abstractions in our core language which significantly simplifies the transformation of partial applications compared to our previous formalisation. The previously published program transformation relied on lambda lifting to eliminate lambda abstractions and incorporated complex rules for transforming partial applications with a variable number of missing arguments.

Related work

Code coverage for declarative programs has gained interest only recently. Simultaneously with our developments, Gill and Runciman (2007) have developed Haskell Program Coverage (HPC) – a tool that monitors code coverage for Haskell programs. HPC implements expression coverage discussed in the introduction to this chapter and does not provide more sophisticated notions of control- or data-flow. Coverage information as monitored by HPC can be beneficially shown to the user in the form of colourised source code but is of little value to ensure thorough testing or select a sufficient number of interesting tests from a large set of automatically generated tests. Our more complex criteria are useful to evaluate the significance of tests but cannot be visualised easily. For example, denoting data flow by arrows like in Section 4.2.1 does not scale to larger programs.

Obtaining information about a computation by instrumenting a declarative program such that it collects this information along with the original result is not new. For example, the Haskell Tracer Hat (Chitil et al. 2003) trans-

forms Haskell programs in order to obtain a computation trace that can be used for various debugging tools. Our transformation differs from others in this area in that it does not introduce impure features like side effects. Rather, the result of our program transformation is a pure declarative program. Using side effects simplifies the collection of additional information about the execution of programs. Especially, the original evaluation order is maintained automatically. We show that this is also possible with a pure approach. In fact, using side effects was not easily possible in our approach because of nondeterminism. We generate test cases nondeterministically and need to collect coverage information that corresponds to individual tests. With side effects we could collect coverage information globally, for all tests together, but it is non-trivial to associate globally collected information to individual tests generated in different nondeterministic branches of the computation. The disadvantage of our pure approach is that it incurs more run-time overhead. However, our experiments show that this overhead is acceptable at least for small-scale unit testing.

5 Explicit Nondeterminism

Logic programming languages like Prolog and functional logic languages like Curry support nondeterminism implicitly. Prolog uses backtracking in order to try different alternatives when proving goals; Curry evaluates different rules of defined operations nondeterministically (see Section 2.2).

(Functional) logic programming can be modeled in a purely functional programming language by representing nondeterministic choice explicitly. For example, if we represent different results of a nondeterministic computation by a lazy list of results then list concatenation expresses nondeterministic choice and evaluating the list of all results is backtracking, that is, depth-first search. Different implementations of explicit nondeterminism are possible and lead to different search strategies.

In Section 5.1 we discuss nondeterminism monads, an abstraction that provides a common interface to different implementations of explicit nondeterminism and present a framework that simplifies the definition of different search strategies by focusing on failure and choice. We then observe that monadic nondeterministic programs sacrifice laziness compared to corresponding functional logic programs in Curry. Laziness is important to implement generate-and-test algorithms more efficiently as we have mentioned in Section 2.2.4. We further motivate this importance and show how to elegantly and efficiently express laziness in monadic nondeterministic programs in Section 5.2.

5.1 Nondeterminism monads

Logic programming functionality can be incorporated into Haskell by expressing nondeterminism explicitly as a computational effect modeled in the framework of monads (see Section 2.1.3). Nondeterministic computations can be expressed monadically using two additional monadic combinators for failure and choice.

$$\begin{aligned} \emptyset &:: m\ a \\ (\oplus) &:: m\ a \to m\ a \to m\ a \end{aligned}$$

The four monadic combinators can be interpreted in the context of nondeterminism.

- $\emptyset$ represents a failing computation, that is, one without results;
- *return* x represents a computation with the single result x;
- $a \oplus b$ represents a computation that yields either a result of the computation a or one of the computation b; and
- $a \gg\!= f$ applies the nondeterministic operation f to any result of a and yields any result of such an application.

In Section 5.1.1 we discuss laws that describe how these combinators interact. We can use them to define a function that yields an arbitrary element of a given list.

$$\begin{aligned} &\mathit{anyof} :: \mathit{MonadPlus}\ m \Rightarrow [a] \to m\ a \\ &\mathit{anyof}\ [\,] \quad\ = \emptyset \\ &\mathit{anyof}\ (x : xs) = \mathit{anyof}\ xs \oplus \mathit{return}\ x \end{aligned}$$

The type signature specifies that the result of *anyof* can be expressed using a parametrised type m that is an instance of the type class *MonadPlus* (a monad for nondeterminism that supports the shown four operations). The first rule of *anyof* uses the failing computation $\emptyset$ to indicate that no result can be returned if the given list is empty. If it is nonempty, the second rule either returns a result of the recursive call to the tail of the list or the first element.

In Section 5.1.2, we introduce different implementations of parametrised types m that can be used to compute results of the nondeterministic operation *anyof*. Starting with an intuitive but inefficient implementation, we subsequently refine it using standard techniques. Specifically, we use difference lists to improve the asymptotic complexity of list concatenation and transform computations to continuation-passing style, which provides an implementation of monadic bind for free. Finally, we arrive at a well-known efficient implementation of backtracking by combining these techniques. We show in Section 5.1.3 how to use the developed ideas to find novel implementations of breadth-first search and iterative deepening depth-first search and compare different search strategies experimentally in Section 5.1.4.

5.1.1 Laws

Instances of the *MonadPlus* type class are usually required to satisfy laws that describe how the new combinators $\emptyset$ and $\oplus$ interact with the monadic combinators *return* and $\gg\!=$. In addition to the laws that only involve *return* and

$\gg\!=$ discussed in Section 2.1.3, the following identities need to be satisfied for a valid *MonadPlus* instance.

$$\begin{aligned}&\emptyset \qquad \gg\!= f \equiv \emptyset\\ &(a \oplus b) \gg\!= f \equiv (a \gg\!= f) \oplus (b \gg\!= f)\end{aligned}$$

These laws are supported by the intuition about the $\gg\!=$ combinator mentioned above. If the first argument to $\gg\!=$ is a failing computation then there are no results to which the given nondeterministic operation f could be applied and, hence, the result is also a failing computation. If the first argument to $\gg\!=$ is a nondeterministic choice between a and b then f is either applied to a result of a or to one of b independently.

In addition to the above laws that describe the interaction of the combinators for failure and choice with the monadic combinators, there are also laws that specify properties of $\emptyset$ and $\oplus$ alone. Often, nondeterministic monadic actions are required to form a monoid:

$$\begin{aligned}&\emptyset \oplus a \qquad \equiv a\\ &a \oplus \emptyset \qquad \equiv a\\ &(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)\end{aligned}$$

The $\emptyset$ combinator should be an identity for $\oplus$ which should be associative.

5.1.2 Backtracking

The most intuitive implementation of the *MonadPlus* type class uses *lists of successes* to represent results of nondeterministic computations. The four monadic operations are implemented on lists as follows. The failing computation is represented as empty list.

$$\begin{aligned}&\emptyset :: [a]\\ &\emptyset = [\,]\end{aligned}$$

A deterministic computation with result x is represented as list with a single element x.

$$\begin{aligned}&return :: a \to [a]\\ &return\ x = [x]\end{aligned}$$

To choose from the results of two nondeterministic computations, the results of both are concatenated using the append function $+\!\!+$.

$$\begin{aligned}&(\oplus) :: [a] \to [a] \to [a]\\ &xs \oplus ys = xs +\!\!+ ys\end{aligned}$$

Nondeterministic operations can be applied to any result of a given computation, for example, by using a *list comprehension*.

$$(\gg\!=) :: [a] \to (a \to [b]) \to [b]$$
$$xs \gg\!= f = [y \mid x \leftarrow xs, y \leftarrow f\ x]$$

It is easy to verify that these implementations indeed satisfy the laws given above. As Haskell lists implement the interface of the *MonadPlus* type class, we can use lists to compute results of the nondeterministic operation *anyof*. For example, we can apply it to a list of numbers in order to get another list of the numbers that are contained in the list.

```
> anyof [1..10] :: [Int]
[10,9,8,7,6,5,4,3,2,1]
```

We provide an explicit type annotation in the call above which specifies that we want to use the list monad to compute the results. The resulting list does indeed contain each number of the given list but in reverse order. As different results of nondeterministic computations are independent their order is irrelevant, at least from a declarative point of view. We should expect different search strategies to enumerate results of nondeterministic computations in different orders.

If we compute results of *anyof* for long lists, we recognise that the list monad scales badly on this example. This is because of the specific implementation of *anyof* that uses a recursive call in the left argument of *mplus*. Actually, if we use the list monad then the implementation of *anyof* is the naive reverse function and, thus, has quadratic run time. We could change the implementation of *anyof* to avoid left recursion by swapping the arguments of *mplus*. However, we refrain from doing so and rather strive for a monad that can handle it gracefully.

Difference lists

The reason why the list monad scales so badly in case of left associative use of $\oplus$ is that the function $+\!\!+$ for list concatenation used for implementing $\oplus$ has linear run time in the length of its first argument. The standard technique to avoid this complexity is to use so called difference lists. A difference list is a function which takes a list as argument and yields a possibly longer list that ends with the given list. We can define the type of difference lists using Haskell's record syntax as follows.

$$\textbf{newtype}\ \mathit{DiffList}\ a = \mathit{DiffList}\ \{\,(\not\gg) :: [a] \to [a]\,\}$$

This declaration automatically generates a selector function

$$(\twoheadrightarrow) :: \mathit{DiffList}\ a \to [a] \to [a]$$

that can be used to append an ordinary list to a difference list. As an interface to difference lists, we need a function to construct the empty difference list;

$$\begin{aligned}&\mathit{empty} :: \mathit{DiffList}\ a\\&\mathit{empty} = \mathit{DiffList}\ \{\,(\twoheadrightarrow) = \mathit{id}\,\}\end{aligned}$$

a function to construct a difference list with a single element;

$$\begin{aligned}&\mathit{singleton} :: a \to \mathit{DiffList}\ a\\&\mathit{singleton}\ x = \mathit{DiffList}\ \{\,(\twoheadrightarrow) = (x:)\,\}\end{aligned}$$

and a function to concatenate two difference lists.

$$\begin{aligned}&(+\!\!+\!\!+) :: \mathit{DiffList}\ a \to \mathit{DiffList}\ a \to \mathit{DiffList}\ a\\&a +\!\!+\!\!+ b = \mathit{DiffList}\ \{\,(\twoheadrightarrow) = (a\twoheadrightarrow) \circ (b\twoheadrightarrow)\,\}\end{aligned}$$

The function $+\!\!+\!\!+$ is implemented via the function composition operator $\circ$ and has, thus, constant run time, which is a critical advantage compared to ordinary lists. The disadvantage of difference lists is that we cannot perform pattern matching without converting them back to ordinary lists. Such conversion can be performed by using $\twoheadrightarrow$ to stick the empty list at the end of a difference list.

$$\begin{aligned}&\mathit{toList} :: \mathit{DiffList}\ a \to [a]\\&\mathit{toList}\ a = a \twoheadrightarrow [\,]\end{aligned}$$

The constant run time of difference-list concatenation is useful because the *toList* function has linear run time, regardless of the structure of the calls to $+\!\!+\!\!+$ that were used to build the difference list. As a consequence of this improvement, the call $\mathit{toList}\ ((\mathit{singleton}\ 1 +\!\!+\!\!+ \mathit{singleton}\ 2) +\!\!+\!\!+ \mathit{singleton}\ 3)$ does not traverse a prefix of the result repeatedly like the call $([1] +\!\!+ [2]) +\!\!+ [3]$ does. If we drop the **newtype** constructors and selectors then the above call is evaluated as follows.

$$\begin{aligned}&\quad \mathit{toList}\ (((1:) \circ (2:)) \circ (3:))\\&\equiv (((1:) \circ (2:)) \circ (3:))\ [\,]\\&\equiv ((1:) \circ (2:))\ (3 : [\,])\\&\equiv (1 : 2 : 3 : [\,])\end{aligned}$$

The number of steps in this derivation is linear in the number of list elements although concatenation has been nested left associatively.

The three functions *empty*, *singleton*, and $+\!\!+$ correspond exactly to the monadic combinators $\emptyset$, *return*, and $\oplus$ respectively. If we inline their definitions in the definition of *anyof* we obtain the following definition of *reverse*.

$$\begin{aligned}&\mathit{reverse} :: [a] \to [a] \to [a]\\&\mathit{reverse}\ [\,] \quad\ = \mathit{id}\\&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs \circ (x:)\end{aligned}$$

This is the well-known linear-time implementation of the reverse function which uses an accumulating parameter to avoid repeated list concatenation. The functions *empty* and $+\!\!+$ are implemented as identity function and function composition respectively and, hence, satisfy the monoid laws (see Section 5.1.1). Nevertheless, we cannot instantiate the type parameter m of *anyof* with the type *DiffList* which is no instance of *MonadPlus*. In order to make the type of difference lists a monad for nondeterminism, we would need to implement the bind operator. Unfortunately, this is only possible by converting back and forth between difference and ordinary lists[1] which is unsatisfactory. We insist on a more elegant solution.

Continuation-passing style

To achieve a more elegant solution, we need another well-known technique, namely, continuation-passing style. A function in continuation-passing style does not yield its result to the caller but is called with an additional function—a so called continuation—that expects the computed result as argument. For example, we could define integer addition in continuation-passing style as follows.

$$\begin{aligned}&\mathit{plusCPS} :: \mathit{Int} \to \mathit{Int} \to (\mathit{Int} \to a) \to a\\&\mathit{plusCPS}\ m\ n\ k = k\ (m + n)\end{aligned}$$

In general, if the result type of an ordinary function is a then the result type of the same function in continuation-passing style is $(a \to b) \to b$. The result type of the continuation is polymorphic. For example, we can pass the *IO* operation $\mathit{print} :: \mathit{Show}\ a \Rightarrow a \to \mathit{IO}\ ()$ as a continuation to *plusCPS* to print the computed result on the standard output.

```
> plusCPS 17 4 print
21
```

We want to combine continuation-passing style with different effects modeled by a parametrised type that represents computations. For this purpose,

[1] $xs \gg\!= f = \mathit{DiffList}\ \{(\not\Vdash) = ([y \mid x \leftarrow \mathit{toList}\ xs, y \leftarrow \mathit{toList}\ (f\ x)]+\!\!+)\}$

it turns out beneficial to restrict the result type of continuations to use some parametrised type c. The type of so restricted computations in continuation-passing style is defined as follows.

$$\textbf{newtype}\ \mathit{CPS}\ c\ a = \mathit{CPS}\ \{(\gg\!-) :: \forall b.(a \to c\ b) \to c\ b\}$$

The *CPS* type uses so called rank-2 polymorphism to introduce the type variable b used in the result type of the continuation. We use Haskell's record syntax again to get the following selector function.

$$(\gg\!-) :: \mathit{CPS}\ c\ a \to (a \to c\ b) \to c\ b$$

A value of type $\mathit{CPS}\ c\ a$ can be converted into a value of type $c\ a$ by passing it a continuation of type $a \to c\ a$ using $\gg\!-$. We define a type class *Computation* for parametrised types that can represent computations and support an operation *yield* that resembles the monadic operation *return*.

$$\begin{array}{l}\textbf{class}\ \mathit{Computation}\ c\ \textbf{where}\\ \quad \mathit{yield} :: a \to c\ a\end{array}$$

We can now pass the operation *yield* as continuation using $\gg\!-$ to run *CPS* values.

$$\begin{array}{l}\mathit{runCPS} :: \mathit{Computation}\ c \Rightarrow \mathit{CPS}\ c\ a \to c\ a\\ \mathit{runCPS}\ a = a \gg\!- \mathit{yield}\end{array}$$

Remarkably, *CPS c is a monad* for *any* parametrised type c. We get implementations of monadic operations *for free*.

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ (\mathit{CPS}\ c)\ \textbf{where}\\ \quad \mathit{return}\ x = \mathit{CPS}\ \{(\gg\!-) = \lambda k \to k\ x\}\\ \quad a \gg\!= f\ \ = \mathit{CPS}\ \{(\gg\!-) = \lambda k \to a \gg\!- \lambda x \to f\ x \gg\!- k\}\end{array}$$

The last definition looks very clever. Fortunately, we do not need to invent it ourselves. It is the standard definition of monadic bind for continuation monads.[2]

Monads for nondeterminism need to support the additional operations $\emptyset$ and $\oplus$. We define another type class for parametrised types, this time to model computations that support failure and choice.

$$\begin{array}{l}\textbf{class}\ \mathit{Nondet}\ n\ \textbf{where}\\ \quad \mathit{failure} :: n\ a\\ \quad \mathit{choice} :: n\ a \to n\ a \to n\ a\end{array}$$

[2] Compare this with the implementation of the *Cont* monad in Haskell's monad transformer library.

This type class is similar to the *MonadPlus* type class; *failure* resembles $\emptyset$ and *choice* resembles $\oplus$. However, the class *Nondet* does not require the parametrised type n to implement monadic bind, which the *MonadPlus* type class does. As we get monadic bind for free from the *CPS* type, we don't need to require it for types that represent nondeterministic computations.

CPS c is not only a monad for any c. If n is an instance of *Nondet* then *CPS n* is an instance of *MonadPlus*.

$$\begin{aligned}&\textbf{instance } \mathit{Nondet}\ n \Rightarrow \mathit{MonadPlus}\ (\mathit{CPS}\ n) \textbf{ where}\\ &\quad \emptyset \quad\ = \mathit{CPS}\ \{(\gg\!-) = \lambda\_ \to \mathit{failure}\}\\ &\quad a \oplus b = \mathit{CPS}\ \{(\gg\!-) = \lambda k \to \mathit{choice}\ (a \gg\!- k)\ (b \gg\!- k)\}\end{aligned}$$

In order to implement the operations for failure and choice we can simply dispatch to the corresponding operations of the *Nondet* class.

Efficient backtracking

Now we combine difference lists and continuation-passing style. We use the type *DiffList* for difference lists and wrap it inside *CPS* to get an efficient implementation of the *MonadPlus* type class. Note that we do not need to implement $\gg\!=$ on difference lists in order to obtain a monad on top of *DiffList*. We only need to implement the functions *failure*, *yield*, and *choice* that correspond to the monadic operations $\emptyset$, *return*, and $\oplus$ respectively. In order to be able to unwrap the *DiffList* type from *CPS*, we need to provide an instance of *Computation* for *DiffList* and to make *CPS DiffList* an instance of *MonadPlus*, we need to provide an instance of *Nondet*. Both instance declarations reuse operations for difference lists defined in Section 5.1.2.

$$\begin{aligned}&\textbf{instance } \mathit{Computation}\ \mathit{DiffList} \textbf{ where}\\ &\quad \mathit{yield} \ = \mathit{singleton}\\ &\textbf{instance } \mathit{Nondet}\ \mathit{DiffList} \textbf{ where}\\ &\quad \mathit{failure} = \mathit{empty}\\ &\quad \mathit{choice} \ = (+\!\!+)\end{aligned}$$

We can now define efficient backtracking for nondeterministic computations.

$$\begin{aligned}&\mathit{backtrack} :: \mathit{CPS}\ \mathit{DiffList}\ a \to [a]\\ &\mathit{backtrack} = \mathit{toList} \circ \mathit{runCPS}\end{aligned}$$

If we inline the **newtype** declarations *DiffList* and *CPS*, we can see that the type *CPS DiffList a* is the same as the following type.

$$\mathit{CPS\ DiffList}\ a \approx \forall b.(a \to [b] \to [b]) \to [b] \to [b]$$

This type is the well-known type used for two-continuation-based depth-first search (Hinze 2000). The first argument of type $a \to [b] \to [b]$ is called *success continuation* and the second argument of type $[b]$ is the so called *failure continuation*. If we inline the monadic operations, we can see that they resemble the operations that are usually used to implement efficient backtracking. The operation $\emptyset$ yields the failure continuation.

$$\emptyset\ \mathit{succ}\ \mathit{fail} = \mathit{fail}$$

The *return* function passes the given argument to the success continuation and also passes the failure continuation for backtracking.

$$\mathit{return}\ x\ \mathit{succ}\ \mathit{fail} = \mathit{succ}\ x\ \mathit{fail}$$

The operation $\oplus$ passes the success continuation to both computations given as arguments and uses the results of the second computation as failure continuation of the first computation.

$$(a \oplus b)\ \mathit{succ}\ \mathit{fail} = a\ \mathit{succ}\ (b\ \mathit{succ}\ \mathit{fail})$$

The bind operation builds a success continuation that passes the result of the first computation to the given function.

$$(a \gg\!= f)\ \mathit{succ}\ \mathit{fail} = a\ (\lambda x\ \mathit{fail} \to f\ x\ \mathit{succ}\ \mathit{fail})\ \mathit{fail}$$

These definitions have been devised from scratch earlier. We have obtained them by combining difference lists with continuation-passing style.

Using *backtrack* to enumerate the results of calling *anyof* produces the same order of results as using the list monad.

```
> backtrack (anyof [1..10])
[10,9,8,7,6,5,4,3,2,1]
```

When using *backtrack*, however, the resulting list is computed more efficiently. The function $\mathit{backtrack} \circ \mathit{anyof}$ is a linear-time implementation of the reverse function. We can inline the monadic operations into the definition of *anyof* to verify this observation.

$$\begin{aligned}
&\mathit{reverse}' :: [a] \to (a \to [b] \to [b]) \to [b] \to [b] \\
&\mathit{reverse}'\ [\,]\qquad \mathit{succ}\ \mathit{fail} = \mathit{fail} \\
&\mathit{reverse}'\ (x : xs)\ \mathit{succ}\ \mathit{fail} = \mathit{reverse}'\ xs\ \mathit{succ}\ (\mathit{succ}\ x\ \mathit{fail})
\end{aligned}$$

If we specialise this definition for $succ = (:)$ then we obtain again the implementation of the reverse function that uses an accumulating parameter to achieve linear run time. The advantage of *CPS DiffList* over *DiffList* is that it has a natural implementation of monadic bind and can, hence, be used to execute monadic computations.

5.1.3 Fair search

In Section 5.1.2 we have seen how to reinvent an existing implementation of monadic backtracking. In this section we develop implementations of breadth-first search and iterative deepening depth-first search that we have not been aware of previously.

Breadth-first search

Backtracking can be trapped if the search space is infinite. If we apply *anyof* to an infinite list then the function *backtrack* diverges without producing a result. Breadth-first search enumerates the search space in level order which results in a fair enumeration of all results.

$$\textbf{newtype}\ \mathit{Levels}\ n\ a = \mathit{Levels}\ \{\mathit{levels} :: [n\ a]\}$$

The parameter n of *Levels* is used to hold the elements of the different levels. If n is an instance of *Nondet*, we can merge them.

$$\begin{aligned}&\mathit{runLevels} :: \mathit{Nondet}\ n \Rightarrow \mathit{Levels}\ n\ a \to n\ a\\&\mathit{runLevels} = \mathit{foldr}\ \mathit{choice}\ \mathit{failure} \circ \mathit{levels}\end{aligned}$$

We could later use lists to represent individual levels but we use difference lists to benefit from more efficient concatenation. Thus, we define breadth-first search as follows.

$$\begin{aligned}&\mathit{bfs} :: \mathit{CPS}\ (\mathit{Levels}\ \mathit{DiffList})\ a \to [a]\\&\mathit{bfs} = \mathit{toList} \circ \mathit{runLevels} \circ \mathit{runCPS}\end{aligned}$$

We only need to provide instances of the type classes *Computation* and *Nondet* such that *bfs* can be applied to nondeterministic monadic computations. The definition of *yield* creates a single level that contains the given argument wrapped in the type n.

$$\begin{aligned}&\textbf{instance}\ \mathit{Computation}\ n \Rightarrow \mathit{Computation}\ (\mathit{Levels}\ n)\ \textbf{where}\\&\quad \mathit{yield}\ x = \mathit{Levels}\ \{\mathit{levels} = [\mathit{yield}\ x]\}\end{aligned}$$

The function *failure* is implemented as an empty list of levels and *choice* creates a new empty level (using the *failure* operation of the underlying parametrised type n) in front of the merged levels of the given computations.

$$\begin{aligned}&\textbf{instance}\ \mathit{Nondet}\ n \Rightarrow \mathit{Nondet}\ (\mathit{Levels}\ n)\ \textbf{where}\\&\quad \mathit{failure}\ \ \ \ = \mathit{Levels}\ \{\mathit{levels} = [\,]\}\\&\quad \mathit{choice}\ a\ b = \mathit{Levels}\ \{\mathit{levels} = \mathit{failure} : \mathit{merge}\ (\mathit{levels}\ a)\ (\mathit{levels}\ b)\}\end{aligned}$$

The use of *failure* in the implementation of *choice* is crucial to achieve breadth-first search because it delays deeper levels which are combined using *merge*.

$$\begin{aligned}&\mathit{merge} :: \mathit{Nondet}\ n \Rightarrow [n\ a] \to [n\ a] \to [n\ a]\\&\mathit{merge}\ [\,]\ \ \ \ \ ys\ \ \ \ \ \ = ys\\&\mathit{merge}\ xs\ \ \ \ \ [\,]\ \ \ \ \ \ = xs\\&\mathit{merge}\ (x : xs)\ (y : ys) = \mathit{choice}\ x\ y : \mathit{merge}\ xs\ ys\end{aligned}$$

We could generalise the type *Levels* to use an arbitrary parametrised type instead of lists to represent the collection of levels. Such a type would need to provide a *zip* operation to implement *merge* which we could require using another type class like *Zipable*. We favour a simpler description using lists.

Iterative deepening depth-first search

Breadth-first search has an advantage when compared with depth-first search because it is fair. However, there is also a disadvantage. It needs a huge amount of memory to store complete levels of the search space. We can trade memory requirements for run time by using depth-first search to enumerate all results of the search space that are reachable within a certain depth limit and incrementally repeat the search with increasing depth limits.

We can define a type for depth-bounded search as a function that takes a depth limit and yields results that can be found within the given limit.

$$\textbf{newtype}\ \mathit{Bounded}\ n\ a = \mathit{Bounded}\ \{(!) :: \mathit{Int} \to n\ a\}$$

The type parameter n is later required to be an instance of *Computation* and *Nondet* and holds the results of depth-bounded search. We use ordinary lists that can be made an instance of these type classes as follows.

$$\begin{aligned}&\textbf{instance}\ \mathit{Computation}\ [\,]\ \textbf{where}\\&\quad \mathit{yield}\ x = [x]\\&\textbf{instance}\ \mathit{Nondet}\ [\,]\ \textbf{where}\\&\quad \mathit{failure}\ = [\,]\\&\quad \mathit{choice}\ \ = (+\!\!+)\end{aligned}$$

We can define an instance of *Nondet* for *Bounded n* as follows. The implementation of *failure* uses the *failure* operation of the underlying type *n*.

```
instance Nondet n ⇒ Nondet (Bounded n) where
  failure    = Bounded {(!) = λ_ → failure}
  choice a b = Bounded
                 {(!) = λd → if d == 0 then failure
                             else choice (a ! (d − 1)) (b ! (d − 1))}
```

The *choice* operation fails if the depth limit is exhausted. Otherwise, it calls the underlying *choice* operation on the given arguments with a decreased depth limit, reflecting that a choice descends one level in the search space.

Iteratively increasing the depth limit of a depth-bounded computation yields a list of levels.

```
levelIter :: (Computation n, Nondet n) ⇒
             Int → CPS (Bounded n) a → Levels n a
levelIter n a = Levels {levels = [(a ≫− yieldB) ! d | d ← [0, n . .]]}
  where
    yieldB x = Bounded
                 {(!) = λd → if d < n then yield x else failure}
```

Between different searches the depth limit is incremented by *n*. If *n* equals one then the returned levels are really the levels of the search space. If it is greater then multiple levels of the search space are collected in a single level of the result.

Instead of *runCPS* (which is defined as (≫−*yield*)) we use a custom function *yieldB* and pass it as continuation to the given computation. This allows us to yield only those results where the remaining depth limit is small enough which are those that have not been enumerated in a previous search. We merge the different levels of iterative deepening search using an arbitrary instance of *Nondet*—using lists results in iterative deepening depth-first search.

```
idfs :: (Computation n, Nondet n) ⇒ Int → CPS (Bounded n) a → n a
idfs n = foldr choice failure ∘ levels ∘ levelIter n
```

This implementation of iterative deepening depth-first search is remarkable because it does not require the depth limit to be returned by depth-bound computations. If we wanted to implement ≫= directly on the type *Bounded n* we would need an updated depth limit as result of executing the first argument (see Spivey (2006)). We don't need to thread the depth limit explicitly when using the bind operation of the *CPS* type.

Both *bfs* and *idfs* can enumerate arbitrary infinite search spaces lazily.

```
> take 10 (bfs (anyof [1..])) == take 10 (idfs 1 (anyof [1..]))
True
```

In fact, when using lists for the results, *idfs* 1 always returns them in the same order as *bfs* because it enumerates one level after the other from left to right.

5.1.4 Experiments

Using the types developed in the previous sections we can build numerous variants of the presented search strategies. In this section we compare experimentally three different versions of depth-first search and two versions of both breadth-first and iterative deepening depth-first search. All presented implementations can be built from the types developed in the previous sections: the parametrised types $[\,]$, *CPS* $[\,]$, and *CPS DiffList* are all instances of *MonadPlus* that implement depth-first search. The types *CPS* (*Levels* $[\,]$) and *CPS* (*Levels DiffList*) implement breadth-first search whereas *CPS* (*Bounded* $[\,]$) and *CPS* (*Bounded DiffList*) implement iterative deepening depth-first search.

What if we keep following this pattern further? We can also build the types $CPS\ (x\ (y\ z))$ with $x, y \in \{Levels, Bounded\}$ and $z \in \{[\,], DiffList\}$. We can stack arbitrarily many layers of *Levels* and *Bounded* between *CPS* and $[\,]$ or *DiffList*. If we define instances $Computation\ (CPS\ c)$ and $Nondet\ c \Rightarrow Nondet\ (CPS\ c)$ similar to the *Monad* and *MonadPlus* instances for *CPS* then we can also include multiple layers of *CPS* between *Levels* and *Bounded*. The inclined reader may investigate these types and the performance properties of the resulting strategies. We include some of them in our comparison.

Pythagorean triples

We measure run time and memory requirements of the different nondeterminism monads using the *anyof* function and a slightly more complex action that returns Pythagorean triples nondeterministically. A Pythagorean triple is an increasing sequence of three positive numbers a, b, and c such that $a^2 + b^2 = c^2$.

$$
\begin{aligned}
&pytriple :: MonadPlus\ m \Rightarrow m\ (Int, Int, Int)\\
&pytriple = \mathbf{do}\ a \leftarrow anyof\ [1\,..]\\
&\qquad\qquad b \leftarrow anyof\ [a\,..]\\
&\qquad\qquad c \leftarrow anyof\ [b\,..]\\
&\qquad\qquad guard\ (a * a + b * b == c * c)\\
&\qquad\qquad return\ (a, b, c)
\end{aligned}
$$

	anyof	*pytriple*	$pytriple_{\leqslant}$
$[\,]$	179s/ 9MB	—/—	44s/ 2MB
CPS $[\,]$	196s/11MB	—/—	4s/ 2MB
CPS DiffList	0s/ 6MB	—/—	10s/ 2MB
CPS (*Levels* $[\,]$)	0s/ 1MB	21s/ 966MB	12s/ 966MB
CPS (*Levels DiffList*)	0s/ 1MB	23s/ 966MB	13s/ 966MB
CPS (*Bounded* $[\,]$)	223s/17MB	38s/ 2MB	16s/ 2MB
CPS (*Bounded DiffList*)	7s/14MB	54s/ 2MB	25s/ 2MB
CPS (*Bounded* (*CPS* $[\,]$))	200s/19MB	47s/ 2MB	20s/ 2MB
CPS (*Levels* (*Levels DiffList*))	0s/ 1MB	1206s/2041MB	24s/1929MB

Table 5.1: Performance of different monadic search strategies

The predefined function $guard :: MonadPlus\ m \Rightarrow Bool \rightarrow m\ ()$ fails if its argument is *False* and we use it to filter Pythagorean triples from arbitrary strictly increasing sequences of three positive numbers.

That is a concise declarative specification of Pythagorean triples but can we execute it efficiently? It turns out that (unbounded) depth-first search is trapped in infinite branches of the search space and diverges without returning a result. We need a complete search strategy like breadth-first search or iterative deepening search to execute *pytriple*. In order to be able to compare those strategies with unbounded depth-first search, we use a variant $pytriple_{\leqslant} :: MonadPlus\ m \Rightarrow Int \rightarrow m\ (Int, Int, Int)$ that computes Pythagorean triples where all components are less or equal a given number. For this task the search space is finite and can also be explored using incomplete strategies.

Benchmark results

The run time and memory requirements of the different strategies are depicted in Table 5.1. The three columns correspond to three benchmarks:

anyof executes the call *anyof* $[1\,..\,50000]$ and enumerates the results according to the strategies depicted in the leftmost column of the table.

pytriple enumerates 500 Pythagorean triples without an upper bound for their components. This benchmark can only be executed using complete strategies, there are no results for unbounded depth-first search.

$pytriple_{\leqslant}$ enumerates all 386 Pythagorean triples with an upper bound of 500 using all search strategies.

All benchmarks were executed on an Apple MacBook 2.2 GHz Intel Core 2 Duo with 4 GB RAM using a single core. We have used the Glasgow Haskell Compiler (GHC, version 6.10.3) with optimisations (-O -fno-full-laziness) to compile the code. When executing breadth-first search, we have provided an initial heap of 1 GB (+RTS -H1G). We have increased the depth-limit of iterative deepening search by 100 between different searches.

The *anyof* benchmark demonstrates the quadratic complexity of depth-first search strategies based on list concatenation. The corresponding search space is degenerated as it is a narrow but deep tree. Hence, there is noticeable overhead when performing iterative deepening search.

The search space for enumerating Pythagorean triples is more realistic. With and without an upper limit, breadth-first search is faster than iterative deepening depth-first search but uses significantly more memory. Using difference lists instead of ordinary lists does not improve the performance of breadth-first search in our benchmarks. We have observed the memory requirements of iterative deepening depth-first search to be constant only when we disabled *let floating* by turning off the *full-laziness optimisation* of GHC. This optimisation increases sharing in a way that defeats the purpose of iteratively exploring the search space by recomputing it on purpose. Iterative deepening depth-first search incurs noticeable overhead compared to ordinary depth-first search which, however, can only be applied if the search space is finite.

We have tested two more strategies, namely, $\mathit{CPS}\ (\mathit{Bounded}\ (\mathit{CPS}\ [\,]))$ and $\mathit{CPS}\ (\mathit{Levels}\ (\mathit{Levels}\ \mathit{DiffList}))$. The former demonstrates that even wrapping the list type under multiple layers of CPS does not improve on the quadratic complexity of the $\oplus$ operation when nested left associatively. Moreover, the extra *CPS* layer causes moderate overhead compared to ordinary iterative-deepening depth-first search. Using two layers of *Levels* for breadth-first search blows up the memory requirements even more. Although we have run this specific benchmark with 2 GB initial heap, the memory requirements are so huge that reclaiming memory is sometimes a significant performance penalty. The large difference in run time between the *pytriple* and the $\mathit{pytriple}_{\leqslant}$ benchmarks is suspicious. Probably, the slowdown is caused by limiting memory by the option -M2G.

The experiments suggest to use the monad *CPS DiffList* if the search space is known to be finite and $\mathit{CPS}\ (\mathit{Bounded}\ \mathit{DiffList})$ if it is not. Although there is a moderate overhead of difference list compared to usual lists, the latter perform much worse in case of left associative uses of $\oplus$. The memory requirements of breadth-first search prohibit its use in algorithms that require extensive search.

5.1.5 Summary

We have presented monads for nondeterminism in Haskell and a novel approach to define them. By using our method, their implementation is split into two independent parts: one is identical for every implementation, the other captures the essence of the employed search strategy. The part that is identical for every implementation includes the monadic bind operation, which does not have to be reimplemented for every nondeterminism monad. Programmers only need to define notions of *failure* and *choice* and can wrap these definitions in a parametrised type to obtain a monad for nondeterminism.

As the implementation of monadic bind is usually the most involved part in the implementation of nondeterminism monads, our approach gives rise to simpler implementations of search strategies that required a more complex implementation before. For example, difference lists are a natural choice to represent nondeterministic computations efficiently but do not support a natural implementation of monadic bind. We show that wrapping the type for difference lists in a continuation monad results in the well-known two-continuation-based backtracking monad. Using different base types—which both lack a natural implementation of monadic bind—we obtain novel implementations of breadth-first search and iterative deepening depth-first search.

We have compared variations of the presented search strategies experimentally and found that the two-continuation-based backtracking monad outperforms the other strategies. Iterative deepening search, which requires little space compared to other complete search strategies, is also suitable for infinite search spaces.

Monads for nondeterminism are usually expected to satisfy certain laws. Instances of *MonadPlus* derived with the presented approach satisfy the monad laws (see Sections 2.1.3 and 5.1.1) by construction because the implementations of *return* and $\gg\!=$ are always those of the continuation monad. Whether the derived instances satisfy the monoid laws for the *MonadPlus* operations depends on the employed instance of *Nondet*.[3] The strategies presented in Section 5.1.3 do not satisfy the monoid laws, however, manipulating a nondeterministic program according to these laws has no effect on which results are computed—it only affects their order.

[3] We prove all mentioned laws for *CPS* monads in Appendix B.3

5.2 Combining laziness with nondeterminism

At first sight, Haskell seems to provide all necessary features for functional logic programming without further ado: it has

- lazy evaluation which turned out useful for solving generate-and-test problems efficiently and
- nondeterminism, albeit explicitly via monads, which allows to describe search problems elegantly.

On closer inspection, however, it turns out that laziness and nondeterminism cannot be combined easily in Haskell. Nondeterminism monads are not lazy.

A monad is called *lazy* if its bind operation satisfies the following law for every function f.

$$\bot \gg\!= f \equiv f\ \bot$$

As a counter example, the implementation of $\gg\!=$ for the list monad (see Section 5.1.2) performs pattern matching on its first argument. Hence, in the list monad $\bot \gg\!= f \equiv \bot$ and, thus, the list monad is not lazy.

Substituting $f = \lambda\_ \to a$ for some monadic action a leads to an intuitive interpretation of the lazy monad property: If the *result* of a monadic action is ignored, then its *effects* are ignored too. This property of lazy monads is in conflict with the laws for nondeterminism monads described in Section 5.1.1. For example, the law

$$\emptyset \gg\!= f \equiv \emptyset$$

requires that the effect of $\emptyset$ is *not* ignored even if f ignores its argument. Nondeterminism monads account for the effects (failure and choice) of monadic actions even if their results are irrelevant. This property allows, for example, to define an auxiliary function *guard* for nondeterminism monads:

$$\begin{aligned} &guard :: MonadPlus\ m \Rightarrow Bool \to m\ () \\ &guard\ False = \emptyset \\ &guard\ True\ \ = return\ () \end{aligned}$$

This function was used in Section 5.1.4 to prune the search for Pythagorean triples and would not have this desirable effect in a lazy monad.

The functional logic programming community has developed a sound combination of laziness and nondeterminism, namely, *call-time choice* (see

Section 2.2.4). Roughly, call-time choice makes lazy nondeterministic programs predictable and comprehensible because their declarative meanings can be described in terms of (and are often the same as) the meanings of eager nondeterministic programs.

In Section 5.2.1 we exemplify the problems of combining laziness (non-strictness with sharing) and nondeterminism in more detail to motivate the introduction of a monadic combinator for explicit sharing that solves the discussed problems. We clarify the intuitions of explicit sharing in Section 5.2.2 and introduce equational laws to reason about explicit sharing in Section 5.2.3. In Section 5.2.4 we develop an easy to understand implementation in several steps which we generalise and speed up in Section 5.2.5.

5.2.1 Explicit sharing

In this subsection, we revisit non-strictness, sharing, and nondeterminism and explain why combining them is useful and non-trivial.

Consider the following Haskell predicate which checks whether a given list of numbers is sorted:

$$
\begin{aligned}
&\mathit{isSorted} :: [\mathit{Int}] \to \mathit{Bool} \\
&\mathit{isSorted}\ (x : y : zs) = (x \leqslant y) \wedge \mathit{isSorted}\ (y : zs) \\
&\mathit{isSorted}\ \_ \qquad\quad = \mathit{True}
\end{aligned}
$$

The predicate *isSorted* only demands the complete input list if it is sorted. If the list is not sorted then it is only demanded up to the first two elements that are out of order.

As a consequence, we can apply *isSorted* to infinite lists and it will yield *False* if they are unsorted. The *iterate* function defined in Section 2.1.2 produces an infinite list. Nevertheless, the test $\mathit{isSorted}\ (\mathit{iterate}\ (`\mathit{div}`2)\ n)$ yields *False* if $n > 0$. It does not terminate if $n \leqslant 0$ because an infinite list cannot be identified as being sorted without considering each of its elements.

A *lazy* evaluation strategy is not only non-strict. Additionally, it evaluates each expression bound to a variable at most once. If the variable occurs more than once then the result of its first evaluation is *shared*.

Consider the following monadic variant of the *permute* operation defined in Section 2.2.2.

$$
\begin{aligned}
&\mathit{permute} :: \mathit{MonadPlus}\ m \Rightarrow [a] \to m\ [a] \\
&\mathit{permute}\ [\,] \qquad = \mathit{return}\ [\,] \\
&\mathit{permute}\ (x : xs) = \mathbf{do}\ ys \leftarrow \mathit{permute}\ xs \\
&\qquad\qquad\qquad\quad\ \ zs \leftarrow \mathit{insert}\ x\ ys \\
&\qquad\qquad\qquad\quad\ \ \mathit{return}\ zs
\end{aligned}
$$

```
insert :: MonadPlus m ⇒ a → [a] → m [a]
insert x xs = return (x : xs)
            ⊕ case xs of
                [ ]     → ∅
                (y : ys) → do zs ← insert x ys
                              return (y : zs)
```

We demonstrate the advantage of combining laziness with nondeterminism to solve generate-and-test style search algorithms with a toy example, namely, permutation sort. Below is a simple declarative specification of sorting. In words, to sort a list is to compute a permutation of the list that is sorted.

```
sort :: MonadPlus m ⇒ [Int] → m [Int]
sort xs = do ys ← permute xs
             guard (isSorted ys)
             return ys
```

Unfortunately, this program is grossly inefficient, because it iterates through every permutation of the list. It takes about a second to sort 10 elements, more than 10 seconds to sort 11 elements, and more than 3 minutes to sort 12 elements. The inefficiency is mostly because we do not use the non-strictness of the predicate *isSorted*. Although *isSorted* rejects a permutation as soon as it sees two elements out of order, *sort* generates a complete permutation before passing it to *isSorted*. Even if the first two elements of a permutation are already out of order, exponentially many permutations of the remaining elements are computed.

As discussed in the introduction to this chapter, the usual, naive monadic encoding of nondeterminism in Haskell loses non-strictness.

Retaining non-strictness by sacrificing sharing

The problem with the naive monadic encoding of nondeterminism is that the arguments to a constructor must be deterministic. If these arguments are themselves results of nondeterministic computations, these computations must be performed completely before we can apply the constructor to build a nondeterministic result.

To overcome this limitation, we can redefine all data structures such that their components may be nondeterministic. A data type for lists with nondeterministic components is as follows:

```
data List m a = Nil | Cons (m a) (m (List m a))
```

We define operations to construct such lists conveniently:

```
nil :: Monad m ⇒ m (List m a)
nil = return Nil
cons :: Monad m ⇒ m a → m (List m a) → m (List m a)
cons x y = return (Cons x y)
```

We redefine the non-strict *isSorted* to test nondeterministic lists:

```
isSorted :: MonadPlus m ⇒ m (List m a) → m Bool
isSorted ml =
  do l ← ml
     case l of
       Cons mx mxs →
         do xs ← mxs
            case xs of
              Cons my mys →
                do x ← mx
                   y ← my
                   if x ⩽ y then isSorted (cons (return y) mys)
                            else return False
              _ → return True
       _ → return True
```

By generating lists with nondeterministic arguments, we can define a lazier version of the permutation algorithm.

```
permute :: MonadPlus m ⇒ m (List m a) → m (List m a)
permute ml = do l ← ml
                case l of
                  Nil          → nil
                  Cons mx mxs → insert mx (permute mxs)
```

Note that we no longer evaluate (bind) the recursive call of *permute* in order to pass the result to the operation *insert*, because *insert* now takes a nondeterministic list as its second argument.

```
insert :: MonadPlus m ⇒ m a → m (List m a) → m (List m a)
insert mx mxs = cons mx mxs
              ⊕ do Cons my mys ← mxs
                   cons my (insert mx mys)
```

The operation *insert either* creates a new list with the nondeterministic *mx* in front of the nondeterministic *mxs or* it inserts *mx* somewhere in the tail

of *mxs*. Here, the pattern match in the **do**-expression binding is non-exhaustive. If the computation *mxs* returns *Nil*, the pattern-match failure is a failing computation.

Now, we can define a permutation-sort algorithm that lazily checks whether generated permutations are sorted:

```
sort :: MonadPlus m ⇒ m (List m Int) → m (List m Int)
sort xs = let ys = permute xs in
          do True ← isSorted ys
             ys
```

Unfortunately, this version of the algorithm does not sort. It yields every permutation of its input, not only the sorted permutations. This is because the shared variable *ys* in the new definition of *sort* is bound to the nondeterministic *computation* yielding a permutation of the input rather than to the result of this computation. Consequently, *isSorted* checks whether there is a sorted permutation and, if so, *sort* yields an arbitrary permutation.

The presence of nondeterministic components in data structures conflicts with the intuition that shared variables such as *ys* denote *values*, fully determined if not yet fully computed. In order for *sort* to work, the shared nondeterministic computation *ys*, used twice in *sort*, must yield the same result each time.

Our new approach to nondeterminism is lazy in that it preserves both non-strictness and sharing. We provide a combinator *share* for explicit sharing, which can be used to introduce variables for nondeterministic computations that represent *values* rather than computations. The combinator *share* has the signature[4]

```
share :: m a → m (m a)
```

where *m* is an instance of *MonadPlus* that supports explicit sharing. (We describe the implementation of explicit sharing in Sections 5.2.4–5.2.5.) The function *sort* can then be redefined to actually sort:

```
sort xs = do ys ← share (permute xs)
             True ← isSorted ys
             ys
```

In this version of *sort*, the variable *ys* denotes the same permutation wherever it occurs but is nevertheless only computed as much as demanded by the predicate *isSorted*.

[4] In fact, the signature has additional class constraints; see Section 5.2.5.

5.2.2 Intuitions of explicit sharing

In this subsection we present a series of small examples to clarify how to use *share* and what *share* does. We define two simple programs. The computation *coin* flips a coin and nondeterministically returns either 0 or 1.

$$\begin{aligned}&\mathit{coin} :: \mathit{MonadPlus}\ m \Rightarrow m\ \mathit{Int}\\ &\mathit{coin} = \mathit{return}\ 0 \oplus \mathit{return}\ 1\end{aligned}$$

The function *duplicate* evaluates a given computation a twice and returns a pair of the results.

$$\begin{aligned}&\mathit{duplicate} :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ (a,a)\\ &\mathit{duplicate}\ a = \mathbf{do}\ u \leftarrow a\\ &\qquad\qquad\qquad\quad\ v \leftarrow a\\ &\qquad\qquad\qquad\quad\ \mathit{return}\ (u,v)\end{aligned}$$

Sharing enforces call-time choice

We contrast three ways to bind x:

$$\begin{aligned}\mathit{dup\_coin\_let} &= \mathbf{let}\ x = \mathit{coin}\ \mathbf{in} && \mathit{duplicate}\ x\\ \mathit{dup\_coin\_bind} &= \mathbf{do}\ x \leftarrow \mathit{coin}; && \mathit{duplicate}\ (\mathit{return}\ x)\\ \mathit{dup\_coin\_share} &= \mathbf{do}\ x \leftarrow \mathit{share}\ \mathit{coin}; && \mathit{duplicate}\ x\end{aligned}$$

The operations *dup_coin_let* and *dup_coin_bind* do not use *share*, so we can understand their results by treating m as any nondeterminism monad. The operation *dup_coin_let* binds the variable x to the nondeterministic computation *coin*. The function *duplicate* executes x twice—performing two independent coin flips—so *dup_coin_let* yields four answers, namely $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. In contrast, *dup_coin_bind* binds x, of type *Int*, to share not the *coin* computation but its result. The function *duplicate* receives a *deterministic* computation *return* x, whose two evaluations yield the same result, so *dup_coin_bind* yields only $(0,0)$ and $(1,1)$.

The shared computation x in the definition of *dup_coin_share* behaves like *return* x in *dup_coin_bind*: both arguments to *duplicate* are deterministic computations, which yield the same results even when evaluated multiple times. As in Section 5.2.1, we wish to share the results of computations, and we wish variables to denote values. In *dup_coin_bind*, x has the type *Int* and indeed represents an integer. In *dup_coin_share*, x has the type m *Int*, yet it represents one integer rather than multiple nondeterministic integers. Thus *dup_coin_share* yields the same two results as *dup_coin_bind*.

Sharing preserves non-strictness

Shared computations, like the lazy evaluation of pure Haskell expressions, take place only when their results are needed. In particular, if the program can finish without a result from a shared computation, then that computation never happens. The sorting example in Section 5.2.1 shows how non-strictness can improve performance dramatically. Here, we illustrate non-strictness with two shorter examples:

$$\begin{aligned} \mathit{strict\_bind} &= \mathbf{do}\ x \leftarrow \bot :: m\ \mathit{Int} \\ &\qquad\quad \mathit{duplicate}\ (\mathit{const}\ (\mathit{return}\ 2)\ (\mathit{return}\ x)) \\ \mathit{lazy\_share} &= \mathbf{do}\ x \leftarrow \mathit{share}\ (\bot :: m\ \mathit{Int}) \\ &\qquad\quad \mathit{duplicate}\ (\mathit{const}\ (\mathit{return}\ 2)\ x) \end{aligned}$$

The evaluation of *strict_bind* diverges, whereas *lazy_share* yields $(2,2)$. Of course, real programs do not contain $\bot$ or other intentionally divergent computations. We use $\bot$ above to stand for an expensive search whose results are unused.

Alternatively, $\bot$ above may stand for an expensive search that in the end fails to find any solution. If the rest of the program does not need any result from the search, then the shared search is not executed at all. Thus, if we replace $\bot$ with $\emptyset$ in the examples above, *strict_bind* would fail, whereas *lazy_share* would yield $(2,2)$ as before.

Sharing recurs on nondeterministic components

We turn to data types that contain nondeterministic computations, such as *List m a* introduced in Section 5.2.1. We define two functions for illustration: the function *first* takes the first element of a *List*; the function *dupl* builds a *List* with the same two elements.

$$\begin{aligned} &\mathit{first} :: \mathit{MonadPlus}\ m \Rightarrow m\ (\mathit{List}\ m\ a) \rightarrow m\ a \\ &\mathit{first}\ l = l \gg\!= \lambda(\mathit{Cons}\ x\ \mathit{xs}) \rightarrow x \\ &\mathit{dupl} :: \mathit{Monad}\ m \Rightarrow m\ a \rightarrow m\ (\mathit{List}\ m\ a) \\ &\mathit{dupl}\ x = \mathit{cons}\ x\ (\mathit{cons}\ x\ \mathit{nil}) \end{aligned}$$

The function *dupl* is subtly different from *duplicate*: whereas *duplicate* runs a computation twice and returns a data structure with the results, *dupl* returns a data structure containing the same computation twice without running it.

The following two examples illustrate the benefit of data structures with nondeterministic components.

$$
\begin{aligned}
&\mathit{heads\_bind} \;= \mathbf{do}\; l \leftarrow \mathit{cons\ coin}\ \bot \\
&\qquad\qquad\qquad\quad \mathit{dupl}\ (\mathit{first}\ (\mathit{return}\ l)) \\
&\mathit{heads\_share} = \mathbf{do}\; l \leftarrow \mathit{share}\ (\mathit{cons\ coin}\ \bot) \\
&\qquad\qquad\qquad\quad \mathit{dupl}\ (\mathit{first}\ l)
\end{aligned}
$$

Despite the presence of $\bot$, the evaluation of both examples terminates and yields defined results. Since only the head of the list l is needed, the undefined tail of the list is not evaluated.

The expression *cons coin* $\bot$ above denotes a deterministic computation that returns a data structure containing a nondeterministic computation *coin*. The monadic bind in the definition of *heads_bind* shares this data structure, *coin* and all, but not the result of *coin*. The monad laws entail that *heads_bind* yields $\mathit{cons\ coin}\ (\mathit{cons\ coin\ nil})$. When we later execute the latent computations (to print the result, for example), the two copies of *coin* will run independently and yield four outcomes $[0,0]$, $[0,1]$, $[1,0]$, $[1,1]$, so *heads_bind* is like *dup_coin_let* above. Informally, monadic bind performs only shallow sharing, which is not enough for data with nondeterministic components.

Our *share* combinator performs *deep* sharing: all components of a shared data structure are shared as well.[5] For example, the variable l in the definition of *heads_share* stands for a fully determined list with no latent nondeterminism. Thus, *heads_share* yields only two outcomes, $[0,0]$ and $[1,1]$.

Sharing applies to unbounded data structures

Our final example involves a list of nondeterministic, unbounded length, whose elements are each also nondeterministic. The set of possible lists is infinite, yet non-strictness lets us compute with it.

$$
\begin{aligned}
&\mathit{coins} :: \mathit{MonadPlus}\ m \Rightarrow m\ (\mathit{List}\ m\ \mathit{Int}) \\
&\mathit{coins} = \mathit{nil} \oplus \mathit{cons\ coin\ coins} \\
&\mathit{dup\_first\_coin} = \mathbf{do}\ \mathit{cs} \leftarrow \mathit{share\ coins} \\
&\qquad\qquad\qquad\quad \mathit{dupl}\ (\mathit{first}\ \mathit{cs})
\end{aligned}
$$

The nondeterministic computation *coins* yields every finite list of zeroes and ones. Unlike the examples above using $\bot$, each possible list is fully defined and finite, but there are an infinite number of possible lists, and generating every list requires an unbounded number of choices. Even though, as discussed above, the shared variable *cs* represents the fully determined result

[5] Applying *share* to a function does not cause any nondeterminism in its body to be shared. This behavior matches the intuition that invoking a function creates a copy of its body by substitution.

$$
\begin{aligned}
\mathit{return}\ x \gg\!= f &\equiv f\ x && \text{(Lret)}\\
a \gg\!= \mathit{return} &\equiv a && \text{(Rret)}\\
(a \gg\!= f) \gg\!= g &\equiv a \gg\!= \lambda x \to f\ x \gg\!= g && \text{(Bassoc)}\\
\emptyset \gg\!= f &\equiv \emptyset && \text{(Lzero)}\\
(a \oplus b) \gg\!= f &\equiv (a \gg\!= f) \oplus (b \gg\!= f) && \text{(Ldistr)}\\
\mathit{share}\ (a \oplus b) &\equiv \mathit{share}\ a \oplus \mathit{share}\ b && \text{(Choice)}\\
\mathit{share}\ \emptyset &\equiv \mathit{return}\ \emptyset && \text{(Fail)}\\
\mathit{share}\ \bot &\equiv \mathit{return}\ \bot && \text{(Bot)}\\
\mathit{share}\ (\mathit{return}\ (C\ x_1 \ldots x_n)) &\equiv \mathit{share}\ x_1 \gg\!= \lambda y_1 \to \ldots && \text{(HNF)}\\
&\qquad \mathit{share}\ x_n \gg\!= \lambda y_n \to\\
&\qquad \mathit{return}\ (\mathit{return}\ (C\ y_1 \ldots y_n))
\end{aligned}
$$

where C is a constructor with n nondeterministic components

Figure 5.1: The laws of a monad with nondeterminism and sharing

of such an unbounded number of choices, computing *dup_first_coin* only makes the few choices demanded by *dupl* (*first cs*). In particular, *first cs* represents the first element and is demanded twice, each time giving the same result, but no other element is demanded. Thus, *dup_first_coin* produces two results, $[0,0]$ and $[1,1]$.

5.2.3 Laws of explicit sharing

We now formalise the intuitions illustrated above in a set of equational laws that hold up to observation as detailed below. We show here how to use the laws to reason about—in particular, predict the results of—nondeterministic computations with *share*, such as the examples above. In Section 5.2.4, we further use the laws to guide an implementation.

The laws of our monad are shown in Figure 5.1. First of all, our monad satisfies the monad laws, (Lret), (Rret) and (Bassoc) which we have discussed in Section 2.1.3. Our monad is also a *MonadPlus* instance and we include the first two laws described in Section 5.1.1, (Ldistr) and (Lzero). We do not however require that $\oplus$ be associative or that $\emptyset$ be a left or right unit of $\oplus$, so that our monad can be a probabilistic nondeterminism monad, for which $\oplus$ computes the average of probabilities.

Using the laws in Figure 5.1, we can reduce a computation expression in our monad to an expression like $(\emptyset \oplus \dots) \oplus (\mathit{return}\ v \oplus \dots)$, a (potentially infinite) tree whose branches are $\oplus$ and whose leaves are $\bot$, $\emptyset$, or $\mathit{return}\ v$. To observe the computation, we apply a function *run* to convert it to another *MonadPlus* instance. Figure 5.2 gives the laws of *run*. The right-hand sides use primes (return', $\gg\!=\!'$, $\emptyset'$, $\oplus'$) to refer to operations of the target monad.

Using the (Lret) and (Ldistr) laws, we can compute the result of the example *dup_coin_bind* above, which does not use *share*.

$$
\begin{aligned}
& (\mathit{return}\ 0 \oplus \mathit{return}\ 1) \gg\!= \lambda x \to \\
& \quad \mathit{return}\ x \gg\!= \lambda u \to \mathit{return}\ x \gg\!= \lambda v \to \mathit{return}\ (u, v) \\
\equiv\ & \quad \{\ \text{(Lret) twice}\ \} \\
& (\mathit{return}\ 0 \oplus \mathit{return}\ 1) \gg\!= \lambda x \to \mathit{return}\ (x, x) \\
\equiv\ & \quad \{\ \text{(Ldistr)}\ \} \\
& (\mathit{return}\ 0 \gg\!= \lambda x \to \mathit{return}\ (x, x)) \\
& \quad \oplus (\mathit{return}\ 1 \gg\!= \lambda x \to \mathit{return}\ (x, x)) \\
\equiv\ & \quad \{\ \text{(Lret) twice}\ \} \\
& \mathit{return}\ (0, 0) \oplus \mathit{return}\ (1, 1)
\end{aligned}
$$

To show how the laws enforce call-time choice, we derive the same result for *dup_coin_share*, which is

$$
\begin{aligned}
& \mathit{share}\ (\mathit{return}\ 0 \oplus \mathit{return}\ 1) \gg\!= \lambda x \to \\
& \quad x \gg\!= \lambda u \to x \gg\!= \lambda v \to \mathit{return}\ (u, v)
\end{aligned}
$$

We first use the (Choice) law to reduce the expression $\mathit{share}\ (\mathit{return}\ 0 \oplus \mathit{return}\ 1)$ to $\mathit{share}\ (\mathit{return}\ 0) \oplus \mathit{share}\ (\mathit{return}\ 1)$. We then reduce the expression $\mathit{share}\ (\mathit{return}\ 0)$ to $\mathit{return}\ (\mathit{return}\ 0)$ (and $\mathit{share}\ (\mathit{return}\ 1)$ to $\mathit{return}\ (\mathit{return}\ 1)$) using the (HNF) law (HNF is short for "head normal form"). In the law, C stands for a constructor with n nondeterministic components. Since 0 has no nondeterministic components, $n = 0$ and we have

$$
\begin{aligned}
& \mathit{return}\ (\mathit{return}\ 0) \gg\!= \lambda x \to \\
& \quad x \gg\!= \lambda u \to x \gg\!= \lambda v \to \mathit{return}\ (u, v) \\
\equiv\ & \quad \{\ \text{(Lret)}\ \} \\
& \mathit{return}\ 0 \gg\!= \lambda u \to \mathit{return}\ 0 \gg\!= \lambda v \to \mathit{return}\ (u, v) \\
\equiv\ & \quad \{\ \text{(Lret) twice}\ \} \\
& \mathit{return}\ (0, 0)
\end{aligned}
$$

The overall result is thus the same as that for *dup_coin_bind*.

The preservation of non-strictness is illustrated by *lazy_share*. After reducing $\mathit{const}\ (\mathit{return}\ 2)\ x$ there to $\mathit{return}\ 2$, we obtain

$$run\ \emptyset \equiv \emptyset' \qquad \text{(rZero)}$$

$$run\ (a \oplus b) \equiv run\ a \oplus' run\ b \qquad \text{(rPlus)}$$

$$\begin{aligned} run\ (return\ (C\ x_1 \ldots x_n)) \equiv\ & run\ x_1 \gg\!=' \lambda y_1 \to \ldots \\ & run\ x_n \gg\!=' \lambda y_n \to \\ & return'\ (C\ (return'\ y_1) \ldots (return'\ y_n)) \end{aligned} \qquad \text{(rRet)}$$

Figure 5.2: The laws of observing a monad with nondeterminism and sharing in another monad with nondeterminism

$$share\ \bot \gg\!= \lambda x \to duplicate\ (return\ 2)$$

Because x is unused, the result can be computed without evaluating the shared expression. And it is, as assured by the (Bot) law, which reduces *share* $\bot$ to *return* $\bot$ (which is observably different from $\bot$). The (Lret) law then reduces the expression to *duplicate* (*return* 2), and the final result is *return* $(2,2)$. The (Bot) law is discussed further shortly. The (Fail) law works similarly.

We turn to data structures with nondeterministic components. *heads_bind* reduces to *return* (*Cons coin* (*return* (*Cons coin* (*return Nil*)))) by monad laws, where the constructor *Cons* takes two nondeterministic computations as arguments. Whereas applying *run* to observe results without nondeterministic components is a trivial matter of replacing $\emptyset$ by $\emptyset'$, $\oplus$ by $\oplus'$, and *return* by *return*′ (so trivial as to be glossed over above), observing the result of *heads_bind* requires using the (rRet) law in Figure 5.2 in a non-trivial way, with *C* being *Cons* and *n* being 2. The result is

$$\begin{aligned} & run\ coin \gg\!=' \lambda y_1 \to \\ & run\ (return\ (Cons\ coin\ (return\ Nil))) \gg\!=' \lambda y_2 \to \\ & return'\ (Cons\ (return'\ y_1)\ (return'\ y_2)) \end{aligned}$$

which eventually yields four solutions due to two independent observations of *coin*. In general, (rRet) ensures that the final observation yields only fully determined values.

To predict the result of *heads_share*, we need to apply the (HNF) law in a non-trivial way, with *C* being *Cons* and *n* being 2:

$$\begin{aligned} & share\ (return\ (Cons\ coin\ \bot)) \\ \equiv\ & \quad \{\ \text{(HNF)}\ \} \\ & share\ coin \gg\!= \lambda y_1 \to share\ \bot \end{aligned}$$

$$
\begin{array}{ll}
 & \gg\!= \lambda y_2 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1\ y_2)) \\
\equiv & \quad \{\ \text{(Bot) and (Lret)}\ \} \\
 & \mathit{share}\ \mathit{coin} \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1\ \bot)) \\
\equiv & \quad \{\ \text{definition of } \mathit{coin}\ \} \\
 & \mathit{share}\ (\mathit{return}\ 1 \oplus \mathit{return}\ 1) \\
 & \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1\ \bot)) \\
\equiv & \quad \{\ \text{(Choice)}\ \} \\
 & (\mathit{share}\ (\mathit{return}\ 0) \oplus \mathit{share}\ (\mathit{return}\ 1)) \\
 & \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1\ \bot)) \\
\equiv & \quad \{\ \text{(HNF)}\ \} \\
 & (\mathit{return}\ (\mathit{return}\ 0) \oplus \mathit{return}\ (\mathit{return}\ 1)) \\
 & \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1\ \bot)) \\
\equiv & \quad \{\ \text{(Ldistr}\ \} \\
 & (\mathit{return}\ (\mathit{return}\ 0) \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1)\ \bot)) \\
 & \oplus (\mathit{return}\ (\mathit{return}\ 1) \gg\!= \lambda y_1 \to \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ y_1)\ \bot)) \\
\equiv & \quad \{\ \text{(Lret)}\ \} \\
 & \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ 0)\ \bot)) \\
 & \oplus \mathit{return}\ (\mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ 1)\ \bot))
\end{array}
$$

This derivation shows that applying *share* to $\mathit{return}\ (\mathit{Cons}\ \mathit{coin}\ \bot)$ exposes and lifts the latent choice *coin* in the list to the top level. Therefore, sharing a list that contains a choice is equivalent to sharing a choice of a list, so *heads_share* yields only two outcomes.

The (Bot) law and our discussion of *lazy_share* above suggest a more general law

$$\mathit{share}\ a \gg\!= \lambda\_ \to b \equiv b, \qquad \text{(Ignore)}$$

which says that any unused shared computation, not just $\bot$, can simply be skipped. This law implies that $\oplus$ is *idempotent*: $b \oplus b \equiv b$. The proof of the implication is that

$$
\begin{array}{ll}
 & b \oplus b \\
\equiv & \quad \{\ \text{(HNF) and (Lret)}\ \} \\
 & (\mathit{share}\ (\mathit{return}\ 0) \gg\!= \lambda\_ \to b) \oplus (\mathit{share}\ (\mathit{return}\ 1) \gg\!= \lambda\_ \to b) \\
\equiv & \quad \{\ \text{definition of } \mathit{coin} \text{ and (Choice)}\ \} \\
 & \mathit{share}\ \mathit{coin} \gg\!= \lambda\_ \to b \\
\equiv & \quad \{\ \text{(Ignore)}\ \} \\
 & b
\end{array}
$$

Idempotence is justified if we observe a nondeterministic computation as a *set* of outcomes, that is, if we care only *whether* a particular result is produced, not how many times or in what order.

The (Ignore) law enables a simpler analysis of our last example program *dup_first_coin*, which creates an infinite number of choices but demands only a few of them. Without (Ignore), we can only reduce the program using (Choice) and (HNF) to $\emptyset \oplus (a \oplus b)$, where $a \equiv d_0 \oplus (a \oplus a)$, $b \equiv d_1 \oplus (b \oplus b)$, and

$$d_i \equiv \mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ i)\ (\mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ i)\ (\mathit{return}\ \mathit{Nil})))).$$

Using (Ignore), we can arrive at a simpler result with no duplicate solutions, namely $\emptyset \oplus (d_0 \oplus d_1)$.

Intuitions behind our laws

Call-time choice makes shared nondeterminism feel like familiar call-by-value evaluation in monadic style, except $\emptyset$ and $\bot$ are treated like values. Indeed, the laws (Fail) and (Bot) would be subsumed by (HNF) if $\bot$ and $\emptyset$ were *return c* for some *c*. The intuition of treating divergence like a value to express laziness guides standard formalisations of functional logic programming that inspired our laws (González-Moreno et al. 1999; López-Fraguas et al. 2007).

Still, for an equational law, (Bot) is unusual in two ways. First, (Bot) is not constructive: its left-hand side matches a computation that diverges, which is in general not decidable. Therefore, it does not correspond to a clause in the implementation of *share*, as we detail in Sections 5.2.4 and 5.2.5 below. Second, the function *share* is computable and thus monotonic in the domain-theoretic sense, so (Bot) entails that $\mathit{share}\ a \geqslant \mathit{share}\ \bot \equiv \mathit{return}\ \bot$ for all *a*. In particular, we have by (Choice) that $\mathit{share}\ a \oplus \mathit{share}\ b \equiv \mathit{share}\ (a \oplus b) \geqslant \mathit{return}\ \bot$. How can a nondeterministic value $\mathit{share}\ a \oplus \mathit{share}\ b$ possibly be as defined as the deterministic value $\mathit{return}\ \bot$?

The key is that our laws hold only *up to observation*. That is, they only say that replacing certain expressions by others does not affect the (non)termination of well-typed programs[6] when the monad type constructors are held abstract (Hinze 2000; Lin 2006). Observing a computation in our monad requires applying *run* then observing the result in the target monad. Each of the two steps may identify many computations. For example, the order of choices may be unobservable because the target monad is the set monad.[7] Also, we may be unable to disprove that $\mathit{share}\ a \oplus \mathit{share}\ b$ is as defined as $\mathit{return}\ \bot$ because $\mathit{run}\ (\mathit{return}\ \bot)$ diverges.

[6] without selective strictness via *seq*

[7] The set monad can be implemented in Haskell just like the list monad, with the usual *Monad* and *MonadPlus* instances that do not depend on *Eq* or *Ord*, as long as computations can only be observed using the *null* predicate.

A positive example of our laws holding up to observation lies in Constructor-Based Rewriting Logic (CRWL) (González-Moreno et al. 1999), a standard formalisation of functional logic programming. To every term e (which we assume is closed in this informal explanation), CRWL assigns a denotation $[\![e]\!]$, the set of *partial values* that e can *reduce* to. A partial value is built up using constructors such as *Cons* and *Nil*, but any part can be replaced by $\bot$ to form a lesser value. A denotation is a downward-closed set of partial values.

Theorem 1 of López-Fraguas et al. (2008) is a fundamental property of call-time choice. It states that, for every context C and term e, the denotation $[\![C[e]]\!]$ equals the denotation $\bigcup_{t \in [\![e]\!]} [\![C[t]]\!]$ which is the union of denotations of $C[t]$ where t is drawn from the denotation of e. Even if e is nondeterministic, the denotation of a large term that contains e can be obtained by considering each partial value e can reduce to. Especially, if e is an argument to a function that duplicates its argument, this argument denotes the same value wherever it occurs. The monadic operation $\oplus$ for nondeterministic choice resembles the operation ? defined in Section 2.2.3. Using the theorem above, we conclude that $[\![C[a?b]]\!] = [\![C[a]?C[b]]\!]$, which inspired our (Choice) law.

5.2.4 Implementing explicit sharing

We start to implement *share* in this section. We begin with a very specific version and generalise it step by step. Revisiting the equational laws for *share*, we show how memoisation can be used to achieve the desired properties. First, we consider values without nondeterministic components, namely values of type *Int*. We then extend the approach to nondeterministic components, namely lists of numbers. An implementation for arbitrary user-defined nondeterministic types in terms of a transformer for arbitrary instances of *MonadPlus* is given in Section 5.2.5.

The tension between late demand and early choice

Lazy evaluation means to evaluate expressions at most once and not until they are demanded. The law (Ignore) from the previous subsection, or more specifically, the laws (Fail) and (Bot) from Figure 5.1 formalise late demand. In order to satisfy these laws, we could be tempted to implement *share* as follows:

$$\begin{aligned} &\mathit{share} :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ (m\ a) \\ &\mathit{share}\ a = \mathit{return}\ a \end{aligned}$$

and so *share* $\bot$ is trivially *return* $\bot$, just as the law (Bot) requires; (Fail) is similarly satisfied. But (Choice) fails, because *return* $(a \oplus b)$ is not equal to *return* $a \oplus$ *return* b. For example, if we take *dup_coin_share* from Section 5.2.2 and replace *share* with *return*, we obtain *dup_coin_let*—which, as explained there, shares only a nondeterministic computation, not its result as desired. Instead of re-making the choices in a shared monadic value each time it is demanded, we must make the choices only once and reuse them for duplicated occurrences.

We could be tempted to try a different implementation of *share* that ensures that choices are performed immediately:

$$\begin{aligned} &\mathit{share} :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ (m\ a) \\ &\mathit{share}\ a = a \gg\!= \lambda x \to \mathit{return}\ (\mathit{return}\ x) \end{aligned}$$

This implementation satisfies the (Choice) law, but it does not satisfy the (Fail) and (Bot) laws. The (Lzero) law of *MonadPlus* shows that this implementation renders *share* $\emptyset$ equal to $\emptyset$, which is observationally different from the *return* $\emptyset$ required by (Fail). This attempt ensures early choice using early demand, so we get eager sharing, rather than lazy sharing as desired.

Memoisation

We can combine late demand and early choice using memoisation. The idea is to delay the choice until it is demanded, and to remember the choice when it is made for the first time so as to not make it again if it is demanded again.

To demonstrate the idea, we define a very specific version of *share* that fixes the monad and the type of shared values. We use a state monad to remember shared monadic values. A state monad is an instance of the following type class, which defines operations to query and update a threaded state component.

$$\begin{aligned} &\textbf{class}\ \mathit{MonadState}\ s\ m\ \textbf{where} \\ &\quad \mathit{get} :: m\ s \\ &\quad \mathit{put} :: s \to m\ () \end{aligned}$$

In our case, the threaded state is a list of *thunks* that can be either unevaluated or evaluated.

$$\textbf{data}\ \mathit{Thunk}\ a = \mathit{Uneval}\ (\mathit{Memo}\ a) \mid \mathit{Eval}\ a$$

Here, *Memo* is the name of our monad. It threads a list of *Thunk*s through nondeterministic computations represented as lists.

```
newtype Memo a =
  Memo {unMemo :: [Thunk Int] → [(a, [Thunk Int])]}
```

The instance declarations of *Memo* for the type classes *Monad*, *MonadState*, and *MonadPlus* are as follows:

```
instance Monad Memo where
  return x = Memo (λts → [(x, ts)])
  m >>= f  = Memo (concatMap (λ(x, ts) → unMemo (f x) ts)
                             ∘ unMemo m)
instance MonadState [Thunk Int] Memo where
  get      = Memo (λts → [(ts, ts)])
  put ts   = Memo (λ_ → [((), ts)])
instance MonadPlus Memo where
  ∅        = Memo (const [])
  a ⊕ b    = Memo (λts → unMemo a ts ++ unMemo b ts)
```

It is crucial that the thunks are passed to both alternatives separately in the implementation of *mplus*. Using mutable global state to store the thunks would not suffice because thunks are created and evaluated differently in different nondeterministic branches.

We can implement a very specific version of *share* that works for integers in the *Memo* monad.

```
share :: Memo Int → Memo (Memo Int)
share a = memo a

memo a =
  do thunks ← get
     let index = length thunks
     put (thunks ++ [Uneval a])
     return (do thunks ← get
                case thunks !! index of
                  Eval x   → return x
                  Uneval a →
                    do x ← a
                       thunks ← get
                       let (xs, _ : ys) = splitAt index thunks
                       put (xs ++ [Eval x] ++ ys)
                       return x)
```

This implementation of *share* adds an unevaluated thunk to the current store and returns a monadic action that, when executed, queries the store and either returns the already evaluated result or evaluates the unevaluated thunk

before updating the threaded state. The argument *a* given to *share* is not demanded until the inner action is performed. Hence, this implementation of *share* satisfies the (Fail) and (Bot) laws. Furthermore, the argument is only evaluated once, followed by an update of the state to remember the computed value. Hence, this implementation of *share* satisfies the (Choice) law (up to observation). If the inner action is duplicated and evaluated more than once, then subsequent calls will yield the same result as the first call due to memoisation.

Nondeterministic components

The version of *share* just developed memoises only integers. However, we want to memoise data with nondeterministic components, such as permuted lists that are computed on demand. So instead of thunks that evaluate to numbers, we redefine the *Memo* monad to store thunks that evaluate to lists of numbers now.

```
newtype Memo a =
  Memo {unMemo :: [Thunk (List Memo Int)]
                -> [(a, [Thunk (List Memo Int)])]}
```

The instance declarations of *Memo* for *Monad* and *MonadPlus* stay the same. In the *MonadState* instance only the state type needs to be adapted. We also reuse the *memo* function, which has now a different type. We could try to define *share* simply as a renaming for *memo* again:

```
share :: Memo (List Memo Int) -> Memo (Memo (List Memo Int))
share a = memo a
```

However, with this definition lists are not shared deeply. This behavior corresponds to the expression *heads_bind* where the head and the tail of the demanded list are still executed whenever they are demanded and may hence yield different results when duplicated. This implementation does not satisfy the (HNF) law.

We can remedy this situation by recursively memoising the head and the tail of a shared list:

```
share :: Memo (List Memo Int) -> Memo (Memo (List Memo Int))
share a = memo (do l <- a
                   case l of
                     Nil        -> nil
                     Cons x xs -> do y <- share x
                                     ys <- share xs
                                     cons y ys)
```

This implementation of *share* memoises data containing nondeterministic components as deeply as demanded by the computation. Each component is evaluated at most once and memoised individually in the list of stored thunks.[8]

Observing nondeterministic results

In order to observe the results of a computation that contains nondeterministic components, we need a function (such as *run* in Figure 5.2) that evaluates all the components and combines the resulting alternatives to compute a nondeterministic choice of deterministic results. For example, we can define a function *eval* that computes all results from a nondeterministic list of numbers.

$$
\begin{aligned}
&\mathit{eval} :: \mathit{List\ Memo\ Int} \to \mathit{Memo}\ (\mathit{List\ Memo\ Int}) \\
&\mathit{eval}\ \mathit{Nil} \qquad\quad = \mathit{return}\ \mathit{Nil} \\
&\mathit{eval}\ (\mathit{Cons}\ x\ xs) = \mathbf{do}\ y \leftarrow x \gg\!= \mathit{eval} \\
&\qquad\qquad\qquad\qquad\quad\ \ ys \leftarrow xs \gg\!= \mathit{eval} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ y)\ (\mathit{return}\ ys))
\end{aligned}
$$

The lists returned by *eval* are fully determined. Using *eval*, we can define an operation *run* that computes the results of a nondeterministic computation:

$$
\begin{aligned}
&\mathit{run} :: \mathit{Memo}\ (\mathit{List\ Memo\ Int}) \to [\mathit{List\ Memo\ Int}] \\
&\mathit{run}\ m = \mathit{map}\ \mathit{fst}\ (\mathit{unMemo}\ (m \gg\!= \mathit{eval})\ [\,])
\end{aligned}
$$

In order to guarantee that the *observed* results correspond to *predicted* results according to the laws in Section 5.2.3, we place two requirements on the monad used to observe the computation ($[\,]$ above). (In contrast, the laws in Section 5.2.3 constrain the monad used to express the computation (*Memo* above).)

Idempotence of choice The (Choice) law predicts that executing the call $\mathit{run}\ (\mathit{share}\ \mathit{coin} \gg\!= \lambda\_ \to \mathit{return}\ \mathit{Nil})$ gives $\mathit{return}'\ \mathit{Nil} \oplus' \mathit{return}'\ \mathit{Nil}$. However, our implementation gives a single solution $\mathit{return}'\ \mathit{Nil}$ (following the (Ignore) law, as it turns out). Hence, we require $\oplus'$ to be *idempotent*; that is, $a \oplus' a \equiv a$.

[8] This implementation of *share* does not actually type-check because *share x* in the body needs to invoke the previous version of *share*, for the type *Int*, rather than this version, for the type *List Memo Int*. The two versions can be made to coexist, each maintaining its own state, but we develop a polymorphic *share* combinator in Section 5.2.5 below, so the issue is moot.

This requirement is satisfied if we abstract from the multiplicity of results (considering $[\,]$ as the set monad rather than the list monad), as is common practice in functional logic programming, or if we treat $\oplus'$ as averaging the weights of results, as is useful for probabilistic inference.

Distributivity of bind over choice According to the (Choice) law, the result of the computation

$$\begin{aligned}&\mathit{run}\ (\mathit{share}\ \mathit{coin} \gg\!= \lambda c \to \mathit{coin} \gg\!= \lambda y \to c \gg\!= \lambda x \to \\ &\qquad \mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ x)\ (\mathit{return}\ (\mathit{Cons}\ (\mathit{return}\ y)\ (\mathit{return}\ \mathit{Nil})))))\end{aligned}$$

is the following nondeterministic choice of lists (we write $\langle x, y\rangle$ to denote $\mathit{return}'\ (\mathit{Cons}\ (\mathit{return}'\ x)\ (\mathit{return}'\ (\mathit{Cons}\ (\mathit{return}'\ y)\ (\mathit{return}'\ \mathit{Nil}))))$.

$$(\langle 0,0\rangle \oplus' \langle 0,1\rangle) \oplus' (\langle 1,0\rangle \oplus' \langle 1,1\rangle)$$

However, our implementation yields

$$(\langle 0,0\rangle \oplus' \langle 1,0\rangle) \oplus' (\langle 0,1\rangle \oplus' \langle 1,1\rangle).$$

In order to equate these two trees, we require the following distributive law between $\gg\!='$ and $\oplus'$.

$$a \gg\!=' \lambda x \to (f\ x \oplus' g\ x) \equiv (a \gg\!=' f) \oplus' (a \gg\!=' g)$$

If the observation monad satisfies this law, then the two expressions above are equal (we write coin' to denote $\mathit{return}'\ 0 \oplus' \mathit{return}'\ 1$:

$$\begin{aligned}&(\langle 0,0\rangle \oplus' \langle 0,1\rangle) \oplus' (\langle 1,0\rangle \oplus' \langle 1,1\rangle) \\ &\quad \equiv (\mathit{coin}' \gg\!=' \lambda y \to \langle 0, y\rangle) \oplus' (\mathit{coin}' \gg\!=' \lambda y \to \langle 1, y\rangle) \\ &\quad \equiv \mathit{coin}' \gg\!=' \lambda y \to (\langle 0, y\rangle \oplus' \langle 1, y\rangle) \\ &\quad \equiv (\langle 0,0\rangle \oplus' \langle 1,0\rangle) \oplus' (\langle 0,1\rangle \oplus' \langle 1,1\rangle).\end{aligned}$$

Hence, the intuition behind distributivity is that the observation monad does not care about the order in which choices are made. This intuition captures the essence of implementing call-time choice: we can perform choices on demand and the results are as if we performed them eagerly.

In general, it is fine to use our approach with an observation monad that does not match our requirements, as long as we are willing to abstract from the mismatch. For example, the list monad satisfies neither idempotence nor distributivity, yet our equational laws are useful in combination with the list monad if we abstract from the order and multiplicities of results. We also do not require that $\oplus'$ be associative or that $\varnothing'$ be a left or right unit of $\oplus'$.

5.2.5 Generalised, efficient implementation

In this subsection, we generalise the implementation ideas described in the previous section such that

1. arbitrary user-defined types with nondeterministic components can be passed as arguments to the combinator *share*, and
2. arbitrary instances of *MonadPlus* can be used as the underlying search strategy.

We achieve the first goal by introducing a type class with the interface to process nondeterministic data. We achieve the second goal by defining a monad transformer *Lazy* that adds sharing to any instance of *MonadPlus*. After describing a straightforward implementation of this monad transformer, we show how to implement it differently in order to improve performance significantly.

Both of these generalisations are motivated by practical applications in nondeterministic programming.

1. The ability to work with user-defined types makes it easier to compose deterministic and nondeterministic code and to draw on the sophisticated type and module systems of existing functional languages.
2. The ability to plug in different underlying monads makes it possible to employ different search strategies like those presented in Section 5.1.

The implementation of our monad transformer is given in Appendix A.6.

Nondeterministic data

We have seen in the previous section that in order to share nested, nondeterministic data deeply, we need to traverse it and apply the combinator *share* recursively to every nondeterministic component. We have implemented deep sharing for the type of nondeterministic lists, but want to generalise this implementation to support arbitrary user-defined types with nondeterministic components. It turns out that the following interface to nondeterministic data is sufficient:

$$\textbf{class}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{Nondet}\ m\ a\ \textbf{where}$$
$$\quad \mathit{mapNondet} :: (\forall b.\mathit{Nondet}\ m\ b \Rightarrow m\ b \rightarrow m\ (m\ b)) \rightarrow a \rightarrow m\ a$$

A nondeterministic type a with nondeterministic components wrapped in the monad m can be made an instance of $\mathit{Nondet}\ m$ by implementing the

function *mapNondet*, which applies a monadic transformation to each nondeterministic component. The type of *mapNondet* is a rank-2 type: the first argument is a polymorphic function that can be applied to nondeterministic data of any type.

We can make the type *List m Int*, of nondeterministic number lists, an instance of *Nondet* as follows.

```
instance MonadPlus m ⇒ Nondet m Int where
  mapNondet _ c            = return c
instance Nondet m a ⇒ Nondet m (List m a) where
  mapNondet _ Nil          = return Nil
  mapNondet f (Cons x xs) = do y  ← f x
                               ys ← f xs
                               return (Cons y ys)
```

The implementation mechanically applies the given transformation to the nondeterministic arguments of each constructor.

An example for the use of *mapNondet* is the following operation, which computes the fully determined values from a nondeterministic value.

```
eval :: Nondet m a ⇒ a → m a
eval = mapNondet (λa → a >>= eval >>= return ∘ return)
```

This operation generalises the specific version for lists given in Section 5.2.4. In order to determine a value, we determine values for the arguments and combine the results. The bind operation of the monad nicely takes care of the combination.

Our original motivation for abstracting over the interface of nondeterministic data was to define the operation *share* with a more general type. In order to generalise the type of *share* to allow not only different types of shared values but also different monad type constructors, we define another type class.

```
class MonadPlus m ⇒ Sharing m where
  share :: Nondet m a ⇒ m a → m (m a)
```

Nondeterminism monads that support the operation *share* are instances of this class. We next define an instance of *Sharing* with the implementation of *share* for arbitrary nondeterministic types.

State monad transformer

The implementation of memoisation in Section 5.2.4 uses a state monad to thread a list of thunks through nondeterministic computations. The straight-

forward generalisation is to use a state monad transformer to thread thunks through computations in arbitrary monads. A state monad transformer adds the operations defined by the type class *MonadState* to an arbitrary base monad.

The type for *Thunk*s generalises easily to an arbitrary monad:

```
data Thunk m a = Uneval (m a) | Eval a
```

Instead of using a list of thunks, we use a *ThunkStore* with the following interface. Note that the operations *lookupThunk* and *insertThunk* deal with thunks of arbitrary type.

```
emptyThunks :: ThunkStore
getFreshKey  :: MonadState ThunkStore m ⇒ m Int
lookupThunk  :: MonadState ThunkStore m ⇒ Int → m (Thunk m a)
insertThunk  :: MonadState ThunkStore m ⇒ Int → Thunk m a → m ()
```

There are different options to implement this interface. We have implemented thunk stores using the generic programming features of the modules *Data.Typeable* and *Data.Dynamic* but omit corresponding class contexts for the sake of clarity.

Lazy monadic computations can now be performed in a monad that threads a *ThunkStore*. We obtain such a monad by applying the *StateT* monad transformer to an arbitrary instance of *MonadPlus*.

```
type Lazy m = StateT ThunkStore m
```

For any instance *m* of *MonadPlus*, the type constructor *Lazy m* is an instance of *Monad*, *MonadPlus*, and *MonadState ThunkStore*. We only need to define the instance of *Sharing* ourselves, which implements the operation *share*.

```
instance MonadPlus m ⇒ Sharing (Lazy m) where
  share a = memo (a >>= mapNondet share)
```

The implementation of *share* uses the operation *memo* to memoise the argument and the operation *mapNondet* to apply *share* recursively to the nondeterministic components of the given value. The function *memo* resembles the specific version given in Section 5.2.4 but has a more general type.

```
memo :: MonadState ThunkStore m ⇒ m a → m (m a)
memo a = do key ← getFreshKey
            insertThunk key (Uneval a)
            return (do thunk ← lookupThunk key
                       case thunk of
                         Eval x   → return x
                         Uneval b → do x ← b
                                       insertThunk key (Eval x)
                                       return x)
```

The only difference in this implementation of *memo* from before is that it uses more efficient thunk stores instead of lists of thunks.

In order to observe a lazy nondeterministic computation, we use the functions *eval* to compute fully determined values and *evalStateT* to execute actions in the transformed state monad.

```
run :: Nondet (Lazy m) a ⇒ Lazy m a → m a
run a = evalStateT (a >>= eval) emptyThunks
```

This function is the generalisation of the *run* function to arbitrary data types with nondeterministic components that are expressed in an arbitrary instance of *MonadPlus*.

This completes an implementation of our monad transformer for lazy nondeterminism, with all of the functionality motivated in Sections 5.2.1 and 5.2.2.

Optimising performance

We have applied some optimisations that improve the performance of our implementation significantly. We use the permutation sort in Section 5.2.1 with the list monad as underlying nondeterminism monad for a rough measure of performance. The implementation just presented exhausts the search space for sorting a list of length 20 in about 5 minutes.[9] The optimisations described below reduce the run time to 7.5 seconds. All implementations run permutation sort in constant space (5 MB or less) and the final implementation executes permutation sort on a list of length 20 roughly three times faster than the fastest available compiler for Curry, the Münster Curry Compiler (MCC).

[9] We performed our experiments on an Apple MacBook with a 2.2 GHz Intel Core 2 Duo processor using GHC with optimisations (`-O2`).

As detailed below, we achieve this competitive performance by

1. reducing the amount of pattern matching in invocations of the monadic bind operation,
2. reducing the number of store operations when storing shared results, and
3. manually inlining and optimising library code.

Less pattern matching The *Monad* instance for the *StateT* monad transformer performs pattern matching in every call to $\gg\!=$ in order to thread the store through the computation. This is wasteful especially during computations that do not access the store because they do not perform explicit sharing. We can avoid this pattern matching by using a different instance of *MonadState*.

We define the continuation monad transformer *ContT*:[10]

$$\begin{aligned}&\textbf{newtype}\ \mathit{ContT}\ m\ a = C\ \{\mathit{unC} :: \forall w.(a \to m\ w) \to m\ w\}\\&\mathit{runContT} :: \mathit{Monad}\ m \Rightarrow \mathit{ContT}\ m\ a \to m\ a\\&\mathit{runContT}\ m = \mathit{unC}\ m\ \mathit{return}\end{aligned}$$

We can make *ContT m* an instance of the type class *Monad* without using operations from the underlying monad *m*:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ (\mathit{ContT}\ m)\ \textbf{where}\\&\quad \mathit{return}\ x = C\ (\lambda c \to c\ x)\\&\quad m \gg\!= k = C\ (\lambda c \to \mathit{unC}\ m\ (\lambda x \to \mathit{unC}\ (k\ x)\ c))\end{aligned}$$

An instance for *MonadPlus* can be easily defined using the corresponding operations of the underlying monad. The interesting exercise is to define an instance of *MonadState* using *ContT*. When using continuations, a reader monad—a monad where actions are functions that take an environment as input but do not yield one as output—can be used to pass state. More specifically, we need the following operations of reader monads:

$$\begin{aligned}&\mathit{ask} :: \mathit{MonadReader}\ s\ m \Rightarrow m\ s\\&\mathit{local} :: \mathit{MonadReader}\ s\ m \Rightarrow (s \to s) \to m\ a \to m\ a\end{aligned}$$

The function *ask* queries the current environment, and the function *local* executes a monadic action in a modified environment. In combination with

[10] This implementation differs from the definition shipped with GHC in that the result type w for continuations is higher-rank polymorphic.

a continuation monad transformer, the function *local* is enough to implement state updates:

```
instance Monad m ⇒ MonadState s (ContT (ReaderT s m))
  where
    get   = C (λc → ask ≫= c)
    put s = C (λc → local (const s) (c ()))
```

With these definitions, we can define our monad transformer *Lazy*:

```
type Lazy m = ContT (ReaderT ThunkStore m)
```

We can reuse from above the definition of the *Sharing* instance and of the *memo* function used to define *share*.

After this optimisation, searching all sorted permutations of a list of length 20 takes about two minutes rather than five.

Fewer state manipulations The function *memo* just defined performs two state updates for each shared value that is demanded: one to insert the unevaluated shared computation and one to insert the evaluated result. We can save half of these manipulations by inserting only evaluated head-normal forms and using lexical scope to access unevaluated computations. We use a different interface to stores now, again abstracting away the details of how to implement this interface in a type-safe manner.

```
emptyStore  :: Store
getFreshKey :: MonadState Store m ⇒ m Int
lookupHNF   :: MonadState Store m ⇒ Int → m (Maybe a)
insertHNF   :: MonadState Store m ⇒ Int → a → m ()
```

Based on this interface, we can define a variant of *memo* that only stores evaluated head normal forms.

```
memo :: MonadState Store m ⇒ m a → m (m a)
memo a = do key ← getFreshKey
            return (do hnf ← lookupHNF key
                       case hnf of
                         Just x  → return x
                         Nothing → do x ← a
                                      insertHNF key x
                                      return x)
```

Instead of retrieving a thunk from the store on demand if it is not yet evaluated, we can use the action *a* directly because it is in scope. After this

optimisation, searching all sorted permutations of a list of length 20 takes 1.5 minutes rather than two.

Mechanical simplifications The final optimisation is to

1. expand the types in *ContT* (*ReaderT State m*),
2. inline all definitions of monadic operations,
3. simplify them according to monad laws, and
4. provide a specialised version of *memo* that is not overloaded.

This optimisation, like the previous ones, affects only our library code and not its clients; for instance, we did not inline any definitions into our benchmark code. Afterwards, searching all sorted permutations of a list of length 20 takes 7.5 seconds rather than 1.5 minutes. This is the most impressive speedup during this sequence of optimisations, even though it is completely mechanical and should ideally be performed by the compiler.

Surprisingly, our still high-level and very modular implementation (it works with arbitrary monads for nondeterminism and arbitrary types for nested, nondeterministic data) outperforms the fastest available Curry compiler. A Curry program for permutation sort, equivalent to the program we used for our benchmarks, runs for 25 seconds when compiled with MCC and $-O2$ optimisations.

We have also compared our performance on *deterministic* monadic computations against corresponding non-monadic programs in Haskell and Curry. Our benchmark is to call the naive reverse function on long lists, which involves a lot of deterministic pattern-matching. In this benchmark, the monadic code is roughly 20% faster than the corresponding Curry code in MCC. The overhead compared to a non-monadic Haskell program is about the same order of magnitude.

Our library does not directly support narrowing and unification of logic variables but can emulate it by means of lazy nondeterminism. We have measured the overhead of such emulation using the functional logic implementation of the *last* function given in Section 2.2.1. This Curry function uses narrowing to bind *xs* to the spine of the init of *l* and unification to bind *x* and the elements of *xs* to the elements of *l*. We can translate it to Haskell by replacing *x* and *xs* with nondeterministic generators and implementing the unification operator $\doteq$ as equality check. When applying *last* to a list of determined values, the monadic Haskell code is about six times faster than the Curry version in MCC. The advantage of unification shows up when *last*

is applied to a list of logic variables: in Curry, $\doteq$ can unify two logic variables deterministically, while an equality check on nondeterministic generators is nondeterministic and leads to search-space explosion. More efficient unification could be implemented using an underlying monad of equality constraints.

All programs used for benchmarking are listed in Appendix A.7.

5.2.6 Summary

We have presented an equational specification and an efficient implementation of non-strictness, sharing, and nondeterminism embedded in a pure functional language.

Our specification (Figure 5.1) formalizes *call-time choice*, a combination of these three features that has been developed in the FLP community. This combination is intuitive and predictable because the results of computations resemble results of corresponding eager computations and shared variables represent fully determined values as opposed to possibly nondeterministic computations. Our equational laws for lazy nondeterminism can be used to reason about the meaning of nondeterministic programs on a high level. They differ from previous formalizations of call-time choice, which use proof calculi, rewriting, or operational semantics. We describe intuitively the correspondence of López-Fraguas et al.'s formalizations (2007, 2008) with our laws as well as why our implementation satisfies them.

Our implementation is novel in working with custom monadic data types and search strategies; in expressing the sharing of nondeterministic choices explicitly; and in implementing the sharing using first-class stores of typed data.

Our high-level monadic interface was crucial in order to optimize our implementation as described in Section 5.2.5. Initial comparisons of monadic computations with corresponding computations in Curry that use nondeterminism, narrowing, and unification are very promising. We outperform the currently fastest Curry compiler (MCC) on the highly nondeterministic permutation sort algorithm. In our deterministic benchmark we incur acceptable overhead compared to pure Haskell. Simulated narrowing turned out competitive while simulated unification can lead to search space explosion. Our results suggest that our work can be used as a simple, high-level, and efficient implementation target for FLP languages.

5.3 Chapter notes

The work on nondeterminism monads and how to combine them with laziness by explicit sharing has been published previously. The approach to modularly compose monads for nondeterminism using continuations has been presented at a German workshop on programming languages (Fischer 2009c). The formalisation and implementation of explicit sharing to model purely functional lazy nondeterministic programming has been published in the Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (Fischer et al. 2009).

Haskell implementations of the described ideas are available in the Haskell package database (Hackage) as packages *level-monad* and *explicit-sharing*.

Related work

Wadler (1995) has introduced monads as a means to modularly incorporate effects like global state, exception handling, output, or nondeterminism into purely functional programs. Hinze (2000) derived monad transformers for backtracking from equational specifications. We have reinvented the two-continuation-based backtracking by Hinze as a combination of difference lists and continuation-passing style. Using different base types in our approach, we have found implementations of breadth-first search and iterative deepening depth-first search that we have not been aware of previously.

The latter strategies have also been implemented in Haskell by Spivey (2006) but not as instances of the *MonadPlus* type class. Spivey uses a slightly different interface with an operation $\oplus$ for nondeterministic choice and an additional operation *wrap* to increase the search depth by one level. Our implementations of breadth-first search and depth-bounded search use a single operation *choice* that could be expressed as combination of $\oplus$ and *wrap* in Spivey's framework and allows us to implement both strategies in the *MonadPlus* framework. Both implementations differ from the corresponding implementations given by Spivey due to the use of a continuation monad. Unlike Spivey's implementation, we can use difference list to represent levels for breadth-first search and don't need to return updated depth limits in depth-bounded search.

Kiselyov et al. (2005) also improved the search strategy in monadic computations to avoid the deficiencies of depth-first search. However, we are the first to introduce *laziness* in nondeterministic computations modeled using monads in Haskell. Any instance of *MonadPlus*—including those developed in the mentioned work—can be used in combination with our approach.

The interaction of non-strict and nondeterministic evaluation has been studied in the FLP community, leading to different semantic frameworks and implementations. They all establish *call-time choice*, which ensures that computed results correspond to strict evaluation. An alternative interpretation of call-time choice is that variables denote *values* rather than (possibly nondeterministic) computations. As call-time choice has turned out to be the most intuitive model for lazy nondeterminism, we also adopt it.

Unlike approaches discussed below, however, we do not define a new programming language but implement our approach in Haskell. In fact, functional logic programs in Curry or Toy can be compiled to Haskell programs that use our library.

There are different approaches to formalising the semantics of FLP. CRWL (González-Moreno et al. 1999) is a proof calculus with a denotational flavor that allows to reason about functional logic programs using inference rules and to prove program equivalence. Let Rewriting (López-Fraguas et al. 2007, 2008) defines rewrite rules that are shown to be equivalent to CRWL. It is more operational than CRWL but does not define a constructive strategy to evaluate programs. Deterministic procedures to run functional logic programs are described by Albert et al. (2005) in the form of operational big-step and small-step semantics.

We define equational laws for monadic, lazy, nondeterministic computations that resemble let rewriting in that they do not fix an evaluation strategy. However, we provide an efficient implementation of our equational specification that can be executed using an arbitrary *MonadPlus* instance. Hence, our approach is a step towards closing the gap between let rewriting and the operational semantics, as it can be seen as a monadic let calculus that can be executed but does not fix a search strategy.

There are different compilers for FLP languages that are partly based on the semantic frameworks discussed above. Moreover, the operational semantics by Albert et al. (2005) has been implemented as Haskell interpreters by Tolmach and Antoy (2003) and Tolmach et al. (2004). We do not define a compiler that translates an FLP language; nor do we define an interpreter in Haskell. We rather define a monadic language for lazy FLP *within* Haskell. Instead of defining data types for every language construct as the interpreters do, we only need to define new types for data with nondeterministic components. Instead of using an untyped representation for nondeterministic data, our approach is typed.

This tight integration with Haskell lets us be much more efficient than is possible using an interpreter. The KiCS compiler from Curry to Haskell (Braßel and Huch 2009) also aims to exploit the fact that many functional logic programs contain large deterministic parts. Unlike our approach, KiCS

does not use monads to implement sharing but generates unique identifiers using impure features that prevent compiler optimisations on the generated Haskell code.

Naylor et al. (2007) implement a library for functional logic programming in Haskell which handles logic variables explicitly and can hence implement a more efficient version of unification. It does not support data types with nondeterministic components or user-defined search strategies. The authors discuss the conflict between laziness and nondeterminism in Section 5.4 without resolving it.

6 Conclusions

Functional logic programming is well suited for describing and implementing automated testing. Finding appropriate tests is a search problem, nondeterminism of logic programming languages supports an elegant description of search problems, and abstraction mechanisms of functional programming languages support a modular implementation of nondeterminism.

Selecting test input automatically from a, generally infinite, set of possibilities incorporates search. Regardless, whether a tool enumerates small test input exhaustively, arbitrary test input randomly, or significant test input on demand – it always searches an implicit or explicit search space of test inputs. Different approaches to generate test input differ in the search strategy used to explore this search space.

Search spaces can be described elegantly using nondeterministic algorithms. Instead of representing different nondeterministic alternatives explicitly and manipulating them *as a whole*, nondeterministic algorithms support nondeterministic choice implicitly which allows to focus on a *single* computation path instead. Only after the implicitly nondeterministic description of a search space is it made explicit to enumerate different results. The expressive power of implicit nondeterminism is not only observable in the context of logic programming. It is also apparent in other areas of computing science. For example, nondeterministic finite automata (that recognise regular languages) are often simpler than equivalent deterministic ones.

The advanced overloading mechanism of Haskell supports a very general and modular implementation of nondeterminism. Nondeterministic programs can be overloaded with respect to the search strategy such that, for example, the same definition of a test case generator can be used to enumerate test input according to different strategies. Moreover, lazy evaluation plays a crucial role in order to enumerate a search space according to different strategies. Parts of the search space that are not demanded by an enumeration function do not need to be computed.

In this thesis we have

- introduced important declarative programming features (Chapter 2),
- shown how to use them to implement automated testing (Chapter 3),

- developed novel code coverage criteria for the declarative programming paradigm (Chapter 4), and
- presented a lightweight approach to express lazy functional logic programming purely functionally (Chapter 5).

We now summarise the main ideas laid out in the previous chapters, discuss advantages and shortcomings, and propose possible future work.

6.1 Declarative programming promotes concepts that increase the potential for abstraction.

In Chapter 2 we have given an introduction to declarative programming emphasising the ability of programmers to increase the level of abstraction by writing more modular and, thus, reusable code. We have introduced features of purely functional programming languages, namely, polymorphic higher-order functions, lazy evaluation, and class-based overloading as well as features of logic and combined functional logic programming languages, namely, logic variables, implicit nondeterminism, search, and constraints.

Polymorphic type systems provide a means to make functions more generally applicable without sacrificing static type safety. Parts of the input to a function that are not considered by its definition do not need to be given a specific type but can be abstracted using type variables. This is especially useful in combination with higher-order functions (which are functions that take functions as arguments or yield functions as result) to abstract and reuse recurring patterns. Instead of using a predefined set of control structures, declarative programmers can use and define higher-order functions like the *map* function that applies a given function to every element of a list and, thus, abstracts a specific form of a loop.

Lazy evaluation allows to split algorithms into modular parts that communicate via intermediate data structures which are not evaluated completely in between. Optimising compilers may even eliminate the data structure used for communication. Using intermediate data structures conceptually but not actually computing them makes practical a programming style that separates concerns which would otherwise need to be entangled. The use of infinite data structures to separate the computation and selection of square root approximations exemplifies this idea.

Class-based overloading—at first sight not much more than a mechanism to reuse common operation symbols for different types—turns out to increase the generality of declarative programs to an intriguing extend. It enables

polymorphic functions that behave differently on different instantiations of the overloaded type variables which turns out extraordinarily useful when abstracting over type constructors. Executing nondeterministic programs with different search strategies by changing not more than their type is one application of overloaded type constructors. Overloaded operations often come with equational laws that enable programmers to reason about overloaded programs without considering specific instantiations of the overloaded type variables.

The most intriguing feature of logic programming compared to functional programming is the ability to compute with unknown information in form of logic variables. Instantiating logic variables with possible bindings leads to an implicitly nondeterministic execution and it turns out that support for nondeterminism and logic variables can be defined in terms of each other in a combined lazy functional logic programming language. This insight is based on the idea to evaluate nondeterministic operations lazily in order to simulate the demand driven instantiation of logic variables. This idea lies at the heart of our approach to generate tests for declarative programs.

Functional logic programming languages often provide a built-in operation to reify the results of a nondeterministic computation in a deterministic data structure. This feature is crucial in order to compute a test suite from nondeterministically generated tests. We rely on the ability to compute a tree representation of the search space that can be processed lazily to implement different search strategies for enumerating tests.

The easiest way to determine possible bindings of logic variables is to narrow them according to patterns when supplied as arguments to functions. Constraint programming provides an alternative way to restrict the possible bindings of logic variables. For specific problem domains, constraint solvers find possible solutions of a complex search problem much faster than the generic narrowing approach. To exemplify the advantage of constraint solving over narrowing we have compared corresponding solvers for the n-queens problem expressed via predicates on integers with finite domain.

6.2 The mechanism to execute functional logic programs is tailor made for generating tests.

In Chapter 3 we have leveraged the execution mechanism of lazy functional programming languages to generate test input for declarative programs. We have shown how to generate test input for black-box testing based on the

argument types of the tested function and how to employ lazy evaluation in order to generate glass-box tests according to the demand of the tested program. We have defined different strategies to enumerate the search space of generated tests, investigated them according to formal properties, and compared them experimentally on algorithmically complex example programs.

Nondeterministic evaluation of expressions that contain unknown information generates test input for free: if we apply a function to logic variables as input then a result of this call is computed nondeterministically and the input variables are bound to values that correspond to the computed result. We can then check whether the computed result is the intended output for the generated input and later—for example after optimising its implementation—apply the function to the input again to test whether it still computes the intended result.

Although there are programs where checking the generated tests manually is inevitable (for example, if a formal specification is almost identical to the tested program and a test with respect to a verbal specification is preferable) it is often more convenient to automate not only the generation of tests but also their verification. Property-based testing allows to define functions that check for arbitrary test input whether it leads to output that satisfies arbitrary user-defined properties. Often, properties are simply functions with result type *Bool* but property-based test tools usually provide additional combinators to specify more sophisticated properties.

We transfer property-based testing, which has been previously developed for functional programming languages, to the functional logic programming paradigm. In addition to the usual combinators to test deterministic functions we also provide combinators to specify properties of nondeterministic operations. We provide two implementations of property-based testing for functional logic programs that differ in how they generate input for properties. BlackCheck implements black-box testing and GlassCheck implements glass-box testing.

For black-box testing, test input is generated by only considering the type of test input and not the implementation of the property or the tested function. Type-based test input generators are defined via overloading and specify appropriate test input nondeterministically. We have discussed different approaches to define nondeterministic test case generators: simulating logic variables, using abstract data constructors to maintain invariants of test input, and generating custom test data to restrict the search space. We enumerate the results of such generators using different search strategies and supply the generated input to tested properties.

For glass-box testing, test input is not only generated according to its type but also according to the implementations of the property and the tested

function. Instead of enumerating the results of a test case generator before passing them as input to a tested property, we pass the nondeterministic input generator to the property directly and enumerate the results of this call instead. Because of lazy evaluation, the input generator is only evaluated as much as demanded by the property which is especially useful in combination with preconditions: if a precondition rejects test input without demanding it completely, the unevaluated parts of the nondeterministic input generator do not contribute to the search space which often speeds up the search for valid tests. The drawback of enumerating the results of properties rather than their arguments is that we cannot distinguish nondeterminism in the tested operation from nondeterminism in the input generators. As a consequence, the additional combinators for nondeterministic operations provided by BlackCheck are not provided by GlassCheck.

We have presented and compared different search strategies to enumerate tests for black-box and glass-box testing based on a lazy tree representation of the search space. In order to compare different strategies for generating tests, we have defined the following distinguishing criteria:

- completeness ensures that every test in the search space is eventually enumerated if only enough tests are computed,
- advancement ensures that reasonably complex tests are enumerated in acceptable time, and
- balance ensures that the search is not biased towards a specific part of the search space.

In addition to these formal criteria we have used experiments to compare different strategies.

Depth-first search is only feasible with a depth limit because it is otherwise trapped in infinite branches of the search space and often diverges. Iteratively increasing the depth limit enumerates results similar to breadth-first search requiring less space but more time. Despite the run-time overhead due to repeated searches, iterative deepening depth-first search is well suited for testing. It is not advancing and, hence, enumerates many small tests during black-box testing but has been the most efficient strategy in our experiments for glass-box testing.

Random search by repeatedly computing the first result of depth-first search in a shuffled search tree generates very diverse tests and is an improvement over iterative deepening depth-first search for black-box testing because it enumerates larger tests. As different results are computed independently, random search enumerates small tests repeatedly. For glass-box testing, random search turned out impractical. In a search space that has been lazily

pruned via preconditions the test density is often so small that random search has little chance to find a test before descending impractically deep into the search space.

We have developed a novel strategy, namely randomised level diagonalisation, that has all of the properties mentioned above. For black-box testing, the corresponding distribution of generated tests is superior to the other strategies. However, its memory requirements are prohibitive for larger examples. It turns out that iterative discrepancy search has similar properties as level diagonalisation but is more memory efficient and we have used it instead of level diagonalisation in our experiments for glass-box testing. Although iterative discrepancy search, unlike iterative depth-first search, is advancing it did not outperform the latter strategy during glass-box testing.

In summary, our investigations suggest to use glass-box testing in combination with iterative deepening depth-first search for testing deterministic functions and black-box testing with all investigated search strategies for testing nondeterministic operations. Glass-box testing can sometimes be improved considerably by specifying lazy preconditions and we provide a library for lazy predicates to support them.

6.3 Declarative programs call for new notions of code coverage.

In Chapter 4 we have developed novel code coverage criteria for declarative programming languages, shown how to implement them by transforming functional logic programs, and described experiments to evaluate how well they expose errors in algorithmically complex programs. We have presented criteria for control-flow coverage, namely rule coverage and call coverage as well as two different forms of data-flow coverage. We present a program transformation based on a core language for functional logic programs that augments a program such that code coverage information is computed along with the original result. Our experiments show that most thorough testing is achieved with a combination of control- and data-flow coverage.

Our notions of control-flow coverage for declarative programs are based on rules and call positions of defined operations. Rule coverage describes which rules of which functions are executed during a specific program run. Call coverage refines this criterion by distinguishing different call positions when collecting covered rules of an operation. Rather than only ensuring that every rule of an operation is executed eventually, this refined criterion ensures that every rule of an operation is executed at each syntactic occur-

rence of an operation.[1] Call coverage is more difficult to establish than rule coverage and, thus, ensures more thorough testing.

Our notions of data-flow coverage for declarative programs are based on pattern matching and on higher-order application. We define a *def-use chain* as a pair of source code locations where the first represents the definition and the second a use of data. Definitions are the introduction forms of data (constructors or abstractions) and uses are the elimination forms (pattern matching and application). We explicitly do not only consider first-order values as data but also lambda abstractions or partially applied operations.

We have shown that our coverage criteria are well suited to automatically evaluate the significance of tests but also recognised that the more complex criteria are hard to visualise intuitively. For example, the use of arrows to visualise data flow as in Section 4.2.1 does not scale to larger programs where data can flow between different modules.

The presented program transformation supports the collection of control- and data-flow information without using side effects and without changing the (lazy) evaluation order of the original program. Our transformation is formalised based on a core language for functional logic programs that is similar to other core languages for functional and functional logic programming languages. Compared to other core languages for functional languages we add a language construct for nondeterminism and compared to other core languages for functional logic languages we include lambda abstractions to avoid the necessity of lambda lifting. It turns out that by incorporating lambda abstractions in our core language we can significantly simplify the formalisation of our program transformation.[2]

Justified by the observation that logic variables can be simulated by lazy nondeterministic generators (Antoy and Hanus 2006), we do not include logic variables in our core language. As a consequence, our implementation relies on a separate program transformation that eliminates logic variables according to this idea. In fact, it is not easily possible to extend our transformation such that it can handle logic variables because the structure of the coverage information attached to a logic variable (which corresponds to the structure of the data term to which the variable will be bound during the computation) is unknown when the logic variable is declared. The problems with handling logic variables in our program transformation have been an important motivation for the work presented in Chapter 5 which can pro-

[1] We do not distinguish different dynamic evaluations of the same syntactic occurrence of a function call which could lead to infinitely many coverable items.

[2] Comparing the program transformation in Section 4.3.3 with the transformation published previously by Fischer and Kuchen (2008) reveals the simplifying effect of transforming lambda abstractions instead of arbitrary partial applications.

vide more control over nondeterminism and logic variables by representing them explicitly.

We have measured how well the presented code coverage criteria expose bugs in the example programs introduced in Section 3.2.3. It turns out that a combination of control- and data-flow coverage is able to ensure that all errors in the tested programs remain after eliminating redundant tests based on coverage information. We have observed performance problems due to search space explosion—especially when testing arithmetic computations—that could be alleviated using arithmetic constraint solvers. Unfortunately, there is currently no implementation of a functional logic programming language that supports both constraint solving and complete search strategies which has been another important motivation for the work presented in Chapter 5. The monadic implementation of lazy nondeterminism presented in Section 5.2 can be combined with recent work on monadic constraint solving to provide functional logic programming with constraints and user defined search strategies.

6.4 Lazy nondeterministic programs can be turned into equivalent purely functional programs.

In Chapter 5 we have discussed monads for nondeterminism, presented a new technique to modularly implement various search strategies as nondeterminism monads, and shown how to express lazy nondeterminism monadically. We have used a combination of a continuation monad transformer with difference lists to reinvent a well-known monad for efficient backtracking. We have then shown how breadth-first search and iterative deepening depth-first search can be implemented using the same technique by replacing difference lists with something else. After observing that the usual monadic representation of nondeterminism destroys laziness, we have presented an approach to model sharing explicitly to restore laziness, provided laws to reason about explicit sharing, and developed an efficient implementation.

Monads provide a framework to model various computational effects like hidden state, exceptions, or nondeterminism explicitly. Nondeterminism can be expressed monadically with new monadic combinators to model failure and choice. A straightforward but sometimes inefficient way to implement nondeterminism monadically is to use lists in order to represent the results of a nondeterministic computation. The list monad performs backtracking because it traverses the implicit tree of choice operations in depth-first order. The problem with using lists is that the implementation of

list concatenation can lead to quadratic run time in the number of choices when they are nested left associatively.

Difference lists provide more efficient concatenation but do not support a natural implementation of monadic bind. Instead of implementing monadic bind for difference lists directly (or changing the type such that we can implement monadic bind efficiently), we wrap the type for difference lists in a continuation monad transformer which provides monadic bind for free. The result is equivalent to the well-known two-continuation based monad that uses success and failure continuations to backtrack efficiently. Our technique separates the implementation of failure and choice from the implementation of monadic bind and the continuation monad transformer which provides monadic bind can be reused without change to implement different search strategies modularly. Only the underlying type which provides implementations of failure and choice needs to be adapted.

Monadic nondeterminism is usually not hidden in nested components of structured data. For example, when representing nondeterministic lists of numbers as values of type $m\ [Int]$ for some nondeterminism monad m, the head and tail of each represented list need to be deterministic. The absence of nondeterministic components in structured data impedes laziness because algorithms that do not demand certain components of a nondeterministic value still need to evaluate these components – at least to check that their evaluation yields a result.

By using data types with nested monadic components one can hide nested nondeterminism underneath constructors of nondeterministic values to implement lazy nondeterministic algorithms. However, without further ado an explicit representation of nested nondeterminism impedes sharing because shared nondeterministic data represents multiple nondeterministic values. In functional logic programs a shared nondeterministic value denotes the same deterministic value wherever it occurs even if it is not yet evaluated and this property is crucial in order to implement lazy generate-and-test algorithms such as generating glass-box tests for properties with preconditions.

We have presented an approach to express sharing of unevaluated nondeterministic structured data explicitly by using a monad transformer that can be combined with user-defined types for nondeterministic data and any nondeterminism monad. With our approach explicitly shared nondeterministic data evaluates to the same deterministic value in every nondeterministic branch of the computation but is still evaluated on demand. This behaviour corresponds to the notion of call-time choice that has been developed in the functional logic programming community to describe the meaning of lazy nondeterministic programs.

We have formalised the meaning of our combinator for explicit sharing as equational laws that hold *up to observation*. We have discussed intuitively how our laws correspond to similar properties of lazy nondeterministic programs that inspired our approach. In order to *observe* the results of lazy nondeterministic computations in our approach one has to apply an observation function that lifts latent nondeterminism out of nested components of structured data and returns the results in the base monad to which our monad transformer is applied.

We have developed an implementation of explicit sharing in several steps. We have started with a specialised inefficient version that illustrates the underlying idea and then generalised and optimised it in order to support arbitrary user-defined nondeterministic data types and nondeterminism monads. Our implementation uses a store of shared computations that is updated when stored computations are evaluated and passes it through the computation of nondeterministic programs. We have performed experiments to compare our approach with corresponding Curry programs that use nondeterminism, narrowing, and unification. We outperform the Münster Curry compiler in our experiments which suggests that our approach can be used to implement a new compiler for functional logic programming languages.

As mentioned before, our work on expressing lazy nondeterminism purely functionally has been motivated by a lack of control over nondeterminism and logic variables in our program transformation for code coverage collection as well as by missing support for functional logic constraint programming with user defined search strategies. However, we have not applied our monadic implementation of lazy nondeterminism to coverage-based testing or integrated it with monadic constraint solving which both offers possibilities for future work.

A Source Code

A.1 ASCII versions of mathematical symbols

The source code in this thesis is typeset using mathematical notation. The following table shows how to type the non-standard symbols.

$\bot$	`undefined`	$f \circ g$	`f . g`
$\lambda x \to e$	`\x -> e`	**do** $x \leftarrow a; b$	`do x <- a; b`
$x == y$	`x == y`	$x \neq y$	`x /= y`
$x \leqslant y$	`x <= y`	$x \geqslant y$	`x >= y`
$x := y$	`x =:= y`	$xs +\!\!+ ys$	`xs ++ ys`
$\neg x$	`not x`	$x \wedge y$	`x && y`
$x \vee y$	`x || y`	$x \in xs$	`x 'elem' xs`
$a \gg\!= f$	`a >>= f`	$()$	`()`
$x \neq\!\# y$	`x /=# y`	$x -\!\!\# y$	`x -# y`
$\Longrightarrow$	`==>`	$\Longleftarrow$	`<==`
$\doteq$	`-=-`	$\rightleftharpoons$	`<~>`
$\rightharpoonup$	`~>`	$\leftharpoondown$	`<~`
$\bigwedge$	`/\`	$\bigvee$	`\/`
$\circledast$	`<*>`	$\circledcirc$	`<.>`
$\emptyset$	`mzero`	$a \oplus b$	`a 'mplus' b`
$\nrightarrow$	`+>+`	$+\!\!+\!\!+$	`+++`
$\gg\!-$	`>>-`		

A.2 Definitions of used library functions

Here we list definitions of library functions that are used but not defined in the text.

The function $\bot$ denotes a non-terminating computation.

$$\bot :: a$$
$$\bot = \bot$$

The *concat* function merges a list of lists into a single list.

$$
\begin{aligned}
&\mathit{concat} :: [[a]] \to [a] \\
&\mathit{concat}\ [\,] \quad\ = [\,] \\
&\mathit{concat}\ (l : ls) = l +\!\!+ \mathit{concat}\ ls
\end{aligned}
$$

The absolute value of an integer can be computed using *abs*.

$$
\begin{aligned}
&\mathit{abs} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{abs}\ n = \mathbf{if}\ n \geqslant 0\ \mathbf{then}\ n\ \mathbf{else}\ (-n)
\end{aligned}
$$

There are Boolean functions for negation, conjunction, and disjunction.

$$
\begin{aligned}
&\neg :: \mathit{Bool} \to \mathit{Bool} \\
&\neg\ \mathit{True}\ = \mathit{False} \\
&\neg\ \mathit{False} = \mathit{True} \\
&(\wedge), (\vee) :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \\
&\mathit{True}\ \wedge x = x \\
&\mathit{False} \wedge \_ = \mathit{False} \\
&\mathit{True}\ \vee \_ = \mathit{True} \\
&\mathit{False} \vee x = x
\end{aligned}
$$

The function *otherwise* is usually used in the last alternative of guarded rules.

$$
\begin{aligned}
&\mathit{otherwise} :: \mathit{Bool} \\
&\mathit{otherwise} = \mathit{True}
\end{aligned}
$$

The *zip* function pairs the elements of given lists.

$$
\begin{aligned}
&\mathit{zip} :: [a] \to [b] \to [(a, b)] \\
&\mathit{zip}\ [\,] \quad\ \_ \quad\ = [\,] \\
&\mathit{zip}\ (\_ : \_)\ [\,] \quad\ = [\,] \\
&\mathit{zip}\ (x : xs)\ (y : ys) = (x, y) : \mathit{zip}\ xs\ ys
\end{aligned}
$$

The function *take* selects a prefix of given length from a given list, the $(!!)$ function selects the element with the specified index.

$$
\begin{aligned}
&\mathit{take} :: \mathit{Int} \to [a] \to [a] \\
&\mathit{take}\ \_\ [\,] \quad\ = [\,] \\
&\mathit{take}\ n\ (x : xs) = \mathbf{if}\ n \leqslant 0\ \mathbf{then}\ [\,]\ \mathbf{else}\ x : \mathit{take}\ (n - 1)\ xs \\
&(!!) :: \mathit{Int} \to [a] \to a \\
&n\ !!\ (x : xs) = \mathbf{if}\ n == 0\ \mathbf{then}\ x\ \mathbf{else}\ (xs\ !!\ n)
\end{aligned}
$$

The function *concatMap* is a combination of *concat* and *map*.

```
concatMap :: (a → [b]) → [a] → [b]
concatMap = concat ∘ map
```

The operation *unknown* returns an unbound logic variable.

```
unknown :: a
unknown = x where x free
```

Thre predicates *and* and *andC* implement conjunction on lists of Booleans and constraints respectively.

```
and :: [Bool] → Bool
and []     = True
and (b:bs) = b ∧ and bs

andC :: [Success] → Success
andC []     = success
andC (c:cs) = c & andC cs
```

The parametrised type *Maybe* represents optional values.

```
data Maybe a = Nothing | Just a
```

A.3 Abstract heap data type

This is the complete implementation of an abstract data type for heaps, i.e., labeled trees where each sequence of labels from the root to a leaf is non-decreasing. We use so called *pairing heaps* as described by Okasaki (1996).

```
module Heap (
  Heap, emptyHeap, insertHeap,
  isEmptyHeap, splitMinHeap, isValidHeap,
  size, depth
  ) where
import Arbitrary
import Answer
  -- abstract data type for heaps (constructors not exported)
data Heap a = Empty | Fork a [Heap a]
  deriving Show
  -- defined here, because it uses Heap constructors
```

```
instance Arbitrary a ⇒ Arbitrary (Heap a) where
  arbitrary = Empty ? Fork arbitrary arbitrary

isEmptyHeap :: Heap a → Bool
isEmptyHeap Empty      = True
isEmptyHeap (Fork _ _) = False

  -- constructor functions
emptyHeap :: Heap a
emptyHeap = Empty

insertHeap :: Ord a ⇒ a → Heap a → Heap a
insertHeap x = merge (Fork x [])

  -- selector function
splitMinHeap :: Ord a ⇒ Heap a → (a, Heap a)
splitMinHeap (Fork x xs) = (x, mergeAll xs)

  -- auxiliary functions
merge :: Ord a ⇒ Heap a → Heap a → Heap a
merge Empty      h                    = h
merge (Fork x xs) Empty                = Fork x xs
merge (Fork x xs) (Fork y ys) | x ⩽ y      = Fork x (Fork y ys : xs)
                              | otherwise = Fork y (Fork x xs : ys)

  -- check heap property
isValidHeap :: Ord a ⇒ Heap a → Bool
isValidHeap = fromAnswer ∘ isValid

  -- check heap property using fair predicates
isValid :: Ord a ⇒ Heap a → Answer
isValid Empty      = true
isValid (Fork x hs) = Answer.all (answer ∘ (x ⩽)) [y | Fork y _ ← hs]
                    ⋀ Answer.all isValid hs

  -- Boolean version for benchmarks
isValidHeap' :: Ord a ⇒ Heap a → Bool
isValidHeap' Empty      =  True
isValidHeap' (Fork x hs) =  Prelude.all (x ⩽) [y | Fork y _ ← hs]
                         ∧  Prelude.all isValidHeap' hs

mergeAll :: Ord a ⇒ [Heap a] → Heap a
mergeAll []       = emptyHeap
mergeAll [x]      = x
mergeAll (x : y : zs) = merge (merge x y) (mergeAll zs)

  -- compute size and depth of a heap
size, depth :: Heap a → Int
```

```
size  = traverseHeap 0 (succ ∘ foldr (+) 0)
depth = traverseHeap 0 (succ ∘ foldr max 0)

traverseHeap :: a → ([a] → a) → Heap b → a
traverseHeap x _ Empty       = x
traverseHeap x f (Fork _ hs) = f (map (traverseHeap x f) hs)
```

A.4 Implementation of BlackCheck

Blackcheck is a property-based test framework for automated generation of black-box tests for functional logic programs. This section presents its implementation. A more high-level description can be found in Section 3.1.

A.4.1 Input generators

The type class *Arbitrary* specifies types that support an operation to yield test input nondeterministically. We provide default instances for some predefined types like Booleans, lists, numbers, and characters.

```
module Arbitrary where

class     Arbitrary a     where arbitrary :: a
instance  Arbitrary ()    where arbitrary = ()
instance  Arbitrary Bool where arbitrary = False ? True

instance Arbitrary Int where
  arbitrary = 0
  arbitrary = nat
  arbitrary = -nat

  -- we produce a balanced tree of natural numbers
nat = 1; nat = 2 * nat; nat = 2 * nat + 1

  -- we generate upper- and lower-case characters
  -- as well as spaces and newlines
instance Arbitrary Char where
  arbitrary = oneOf (['A'..'z'] ++ ['0'..'9'] ++ " \t\r\n")

oneOf :: [a] → a
oneOf (x:xs) = x ? oneOf xs

instance Arbitrary a ⇒ Arbitrary [a] where
  arbitrary = [] ? (arbitrary : arbitrary)

instance Arbitrary a ⇒ Arbitrary (Maybe a) where
  arbitrary = Nothing ? Just arbitrary

instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (Either a b) where
```

```
    arbitrary = Left arbitrary ? Right arbitrary
instance (Arbitrary a, Arbitrary b) ⇒ Arbitrary (a, b) where
    arbitrary = (arbitrary, arbitrary)
instance (Arbitrary a, Arbitrary b, Arbitrary c) ⇒ Arbitrary (a, b, c) where
    arbitrary = (arbitrary, arbitrary, arbitrary)
```

We provide auxiliary functions to construct *Arbitrary* instances from constructors.

```
cons1 c = c arbitrary
cons2 c = cons1 (c arbitrary)
cons3 c = cons2 (c arbitrary)
cons4 c = cons3 (c arbitrary)
cons5 c = cons4 (c arbitrary)
cons6 c = cons5 (c arbitrary)
cons7 c = cons6 (c arbitrary)
cons8 c = cons7 (c arbitrary)
cons9 c = cons8 (c arbitrary)
cons10 c = cons9 (c arbitrary)
cons11 c = cons10 (c arbitrary)
cons12 c = cons11 (c arbitrary)
cons13 c = cons12 (c arbitrary)
cons14 c = cons13 (c arbitrary)
cons15 c = cons14 (c arbitrary)
cons16 c = cons15 (c arbitrary)
cons17 c = cons16 (c arbitrary)
cons18 c = cons17 (c arbitrary)
cons19 c = cons18 (c arbitrary)
cons20 c = cons19 (c arbitrary)
```

```
module BlackCheck (
    Arbitrary (..),
    Property, (⟹), (⇁), (↽), (⇌),
    label, classify, trivial, collect, collectAs,
    blackCheck, quickCheck, smallCheck,
    depthCheck, iterCheck, rndCheck, diagCheck
    ) where
import Arbitrary
```

```
import List
import AllSolutions
import Random
  -- We use lazy IO to handle infinite search trees
  -- in the Search monad.
import Unsafe (unsafeInterleaveIO)
```

A.4.2 Testable types

We provide a type class *Testable* for types that can be used as argument to test functions like *blackCheck*. Such types need to provide an operation to generate a search space of test cases. A single test case stores a representation of the corresponding test input, a test result, and a list of attached information. The *Search* type is a monad so we can use **do**-notation to construct search spaces.

```
newtype Search a = Search {search :: IO (SearchTree a)}
newtype Property = Property {propTests :: Search Test}
data Test = Test {input :: [String], result :: Maybe Bool, info :: [String]}
noTest :: Test
noTest = Test [] Nothing []
class Testable a where
  tests :: a → Search Test
instance Testable Property where
  tests = propTests
instance Testable Bool where
  tests b = test1 b (λxs → case xs of [x] → x
                                      _   → False)
instance (Show a, Arbitrary a, Testable b) ⇒ Testable (a → b) where
  tests p = do x ← Search (getSearchTree arbitrary)
               fmap (λt → t {input = show x : input t}) (tests (p x))
  -- BlackCheck can also check I/O properties.
instance Testable a ⇒ Testable (IO a) where
  tests a = liftIO a >>= tests
```

A.4.3 Property combinators

We provide combinators to construct complex properties from simpler ones. The implication operation $\Longrightarrow$ is usually employed to ignore tests with in-

valid input. The combinators $\rightharpoonup$, $\leftharpoondown$, and $\rightleftharpoons$ allow to define properties involving nondeterministic operations.

```
(⟹) :: Testable a ⇒ Bool → a → Property
False ⟹ _ = Property (return noTest)
True  ⟹ a = Property (tests a)

(⇀), (↽), (⇌) :: Eq a ⇒ a → a → Property
x ⇀ y = y ↽ x
x ↽ y = Property (test2 isSubsetOf x y)
x ⇌ y = Property (test2 (λa b → a `isSubsetOf` b
                              ∧ b `isSubsetOf` a)
                        x y)
```

A.4.4 Test annotations

We provide combinators to collect statistical information.

```
label :: Testable a ⇒ String → a → Property
label s = Property ∘ fmap (λt → t {info = s : info t}) ∘ tests

classify :: Testable a ⇒ Bool → String → a → Property
classify True  name = label name
classify False _    = Property ∘ tests

trivial :: Testable a ⇒ Bool → a → Property
trivial = (`classify` "trivial")

collect :: (Show s, Testable a) ⇒ s → a → Property
collect = label ∘ show

collectAs :: (Show s, Testable a) ⇒ String → s → a → Property
collectAs name = label ∘ ((name ++ ": ") ++) ∘ show
```

A.4.5 Testing with different strategies

No we can define the *check* operation which is parametrised by a strategy and used to implement the other test functions like *blackCheck*. A strategy maps a search space of tests to a list of traversals.

```
type Strategy = Search Test → [IO [Test]]
check :: Testable a ⇒ Strategy → a → IO ()
check s a = checkTraversals (s (tests a))
```

Based on this function, we can define functions *smallCheck* for exhaustive testing of small test input, *quickCheck* for random testing, and *blackCheck* for testing with randomised level diagonalisation.

Eshaustive testing of small values

We implement iterative deepening on the search space of the test input to implement functions *depthCheck* and *smallCheck* similar to SmallCheck (Runciman et al. 2008). The function *smallCheck* iteratively searches for tests at increasing depths up to the given depth and will find small counter examples before larger ones. The function *depthCheck* directly enumerates all tests up to the given depth in the search space, which is useful to reproduce test failure if the depth of a small counter example is known.

```
depthCheck :: Testable a ⇒ Int → a → IO ()
depthCheck = iterCheck 1

smallCheck :: Testable a ⇒ Int → a → IO ()
smallCheck = flip iterCheck 1

iterCheck :: Testable a ⇒ Int → Int → a → IO ()
iterCheck m n = check (iterDepth m n)

iterDepth :: Int → Int → Strategy
iterDepth m n a = map (λk → searchBetween k (k + n − 1) a)
                      (take m [0, n ..])

searchBetween :: Int → Int → Search a → IO [a]
searchBetween from to = fmap (betweenLevels from to) ∘ search

betweenLevels :: Int → Int → SearchTree a → [a]
betweenLevels from to = go 0
  where go level Fail = []
        go level (Or ts)
          | level ⩾ to = []
          | otherwise = concatMap (go (level + 1)) ts
        go level (Val x)
          | from ⩽ level ∧ level ⩽ to = [x]
          | otherwise                 = []
```

Random testing

We can also implement random testing similar to Claessen and Hughes (2000) by shuffling the search tree before searching it using depth-first search.

```
quickCheck :: Testable a ⇒ a → IO ()
quickCheck = rndCheck 100 1 10

rndCheck :: Testable a ⇒ Int → Int → Int → a → IO ()
rndCheck = checkPasses allValuesD

checkPasses :: Testable a
            ⇒ (SearchTree Test → [Test]) → Int → Int → Int
            → a → IO ()
checkPasses traverse m n max p =
    do gs ← fmap splitGen getStdGen
       check (λa → [values g a | g ← take m gs]) p
  where
    values g a = fmap (takeTests ∘ traverse ∘ shuffleST g) (search a)
    takeTests  = take n ∘ filter isValid ∘ take max
```

Randomised level diagonalisation

Finally, we provide a function *blackCheck* that implements our own strategy randomised level diagonalisation.

```
blackCheck :: Testable a ⇒ a → IO ()
blackCheck = diagCheck 10 10 100

diagCheck :: Testable a ⇒ Int → Int → Int → a → IO ()
diagCheck = checkPasses allValuesDiag

allValuesDiag :: SearchTree a → [a]
allValuesDiag t = [x | Val x ← concat (diagonals (levels [t]))]

levels :: [SearchTree a] → [[SearchTree a]]
levels ts | null ts      = []
          | otherwise    = ts : levels [u | Or us ← ts, u ← us]

diagonals :: [[a]] → [[a]]
diagonals []       = []
diagonals (l : ls) = zipCons l ([] : diagonals ls)

zipCons :: [a] → [[a]] → [[a]]
zipCons []       ls       = ls
zipCons (x : xs) []       = [[y] | y ← x : xs]
zipCons (x : xs) (l : ls) = (x : l) : zipCons xs ls
```

A.4.6 Auxiliary definitions

The remaining definitions are helper functions used previously.

```
isSubsetOf :: Eq a ⇒ [a] → [a] → Bool
isSubsetOf = flip (all ∘ flip elem)

isValid :: Test → Bool
isValid = maybe False (const True) ∘ result

test1 :: Eq a ⇒ a → ([a] → Bool) → Search Test
test1 x p = do xs ← getValues x
               return (noTest {result = Just (p (nub xs))})

test2 :: Eq a ⇒ ([a] → [a] → Bool) → a → a → Search Test
test2 p x y = do ys ← getValues y
                 test1 x (`p`nub ys)

getValues :: a → Search [a]
getValues = liftIO ∘ fmap allValuesB ∘ getSearchTree

liftIO :: IO a → Search a
liftIO a = Search (a >>= return ∘ Val)

instance Functor SearchTree where
  fmap _ Fail      = Fail
  fmap f (Val x)   = Val (f x)
  fmap f (Or ts)   = Or (map (fmap f) ts)

instance Functor Search where
  fmap f = Search ∘ fmap (fmap f) ∘ search

instance Monad Search where
  return  = Search ∘ return ∘ Val
  a >>= f = Search (search a >>= fmap joinST ∘ sequenceST ∘ fmap (search ∘ f))

joinST :: SearchTree (SearchTree a) → SearchTree a
joinST Fail      = Fail
joinST (Val x)   = x
joinST (Or ts)   = Or (map joinST ts)

  -- here we need lazy IO to be able to handle infinite search trees
sequenceST :: SearchTree (IO a) → IO (SearchTree a)
sequenceST Fail      = return Fail
sequenceST (Val x)   = fmap Val x
sequenceST (Or ts)   = fmap Or (mapM (unsafeInterleaveIO ∘ sequenceST) ts)

splitGen :: StdGen → [StdGen]
splitGen g = l : splitGen r
  where (l, r) = split g

shuffle :: StdGen → [a] → [a]
shuffle g l = shuffleWithLen (randoms g) (length l) l

shuffleWithLen :: [Int] → Int → [a] → [a]
shuffleWithLen (r : rs) len xs
  | len == 0    = []
  | otherwise = z : shuffleWithLen rs (len − 1) (ys ++ zs)
```

```
  where
    (ys, z : zs) = splitAt (abs r `mod` len) xs
shuffleST :: StdGen → SearchTree a → SearchTree a
shuffleST _ Fail     = Fail
shuffleST _ (Val x) = Val x
shuffleST g (Or ts) = Or (shuffle r (zipWith shuffleST rs ts))
  where r : rs = splitGen g
checkTraversals :: [IO [Test]] → IO ()
checkTraversals ts = do putStr "0"
                        go ts (0, [])
  where go [ ]       (_, is) = passedOK is
        go (x : xs) nis     = x >>= checkTests nis >>= maybe done (go xs)
passedOK :: [[String]] → IO ()
passedOK is = do putStrLn " tests passed."
                 mapM_ putStrLn (table is)
  where table  = map entry
               ∘ sortBy (flip compare `on` fst)
               ∘ runLength
               ∘ sort
               ∘ filter (¬ ∘ null)
        entry (n, i) = pad 5 (show n) ++ ' ' : concat (intersperse ", " i)
pad :: Int → String → String
pad n s = replicate (n − length s) ' ' ++ s
on :: (b → b → c) → (a → b) → a → a → c
(.∗.) `on` f = λx y → f x .∗. f y
runLength :: Eq a ⇒ [a] → [(Int, a)]
runLength = map (length &&& head) ∘ group
(&&&) :: (a → b) → (a → c) → a → (b, c)
(f &&& g) x = (f x, g x)
checkTests :: (Int, [[String]]) → [Test] → IO (Maybe (Int, [[String]]))
checkTests nis l = go l nis
  where go [ ]       nis = return (Just nis)
        go (t : ts) nis = checkTest nis t >>= maybe (return Nothing) (go ts)
checkTest :: (Int, [[String]]) → Test → IO (Maybe (Int, [[String]]))
checkTest (n, is) t = do putStr (replicate (length count) (chr 8))
                         putStr count
                         maybe (return (Just (n, is)))
                               (notify (info t))
                               (result t)
  where count          = show (n + 1)
        notify i True  = return (Just (n + 1, i : is))
        notify i False = do putStr (nth ++ " test failed")
```

```
                    if null (input t) then putStrLn "."
                      else do putStrLn ", arguments:"
                              mapM_ putStrLn (input t)
                    return Nothing
nth | ((n + 1) `mod` 100) ∈ [11, 12, 13] = "th"
    | otherwise = maybe "th" id (lookup ((n + 1) `mod` 10)
      [(1, "st"), (2, "nd"), (3, "rd")])
```

A.5 Implementation of GlassCheck

GlassCheck is a property-based test framework for automated generation of glass-box tests for Curry programs. Unlike BlackCheck it does not generate test input in advance but applies properties to unevaluated nondeterministic generators.

The disadvantage of this approach is that GlassCheck cannot distinguish between nondeterminism caused by operations used in the definition of the property and nondeterminism in the arguments passed to it. However, this approach also has an important advantage: input is only evaluated as much as demanded by the property. Parts of the input that are not demanded but hide nondeterministic choices cannot blow up the search space. As a consequence, GlassCheck is much more efficient than BlackCheck if properties are lazy, but also more limited—it does not support properties of nondeterministic operations or statistical evaluation of arguments.

It turns out that having only one level of nondeterminism also significantly simplifies the implementation of GlassCheck compared to BlackCheck.

```
module GlassCheck (
  module Arbitrary, module Answer,
  (⟹),
  quickCheck, rndCheck, smallCheck,
  depthCheck, sparseCheck, discrCheck
  ) where
```

We import the *Arbitrary* class with instances as well as search strategies and fair predicates from separate modules.

```
import Arbitrary
import SearchStrategies
import Answer
```

```
import IO (hSetBuffering, stdout, BufferMode (..))
```

The only combinator for constructing properties provided by GlassCheck is the implication operator that prunes away parts of the search space using a precondition. This combinator is the key to the efficiency of GlassCheck because it avoids to search for test input that can be detected invalid lazily.

```
(⟹) :: Bool → a → a
True ⟹ a = a
```

Types that can be passed to the test functions need to support an operation that generates tests for this type nondeterministically. A test is an optional counterexample, i.e., success is represented as *Nothing*.

```
type Test = Maybe [String]   -- list of arguments that lead to failure
class Testable a where
  test :: a → Test
instance Testable Bool where
  test True  = Nothing
  test False = Just []
instance Testable Answer where
  test = test ∘ fromAnswer
```

The most interesting *Testable* instance is the one for function types that guesses arbitrary input an passes it to the property. Unlike the corresponding definition in BlackCheck, the input is not enumerated but the property takes nondeterministic input.

```
instance (Show a, Arbitrary a, Testable b) ⇒ Testable (a → b)
  where test p = fmap (show x:) (test (p x))
          where x = arbitrary
```

Now we define the test functions based on strategies defined in the module *SearchStrategies*. The *quickCheck* function performs 100 random tests to find a counter example.

```
quickCheck :: Testable a ⇒ a → IO ()
quickCheck = rndCheck 100
rndCheck :: Testable a ⇒ Int → a → IO ()
rndCheck = check
        ∘ randomised (strategy ∘ ((take 1 ∘ dfs)∘) ∘ shuffledST)
```

The *smallCheck* function traverses the search space in level order an is thus guaranteed to find a counter example at the lowest level, i.e., a smallest counter example. The *depthCheck* function performs depth bound search without enumerating levels incrementally and is useful to reproduce test failure without repeating unsuccessful searches at lower levels.

```
smallCheck :: Testable a ⇒ Int → a → IO ()
smallCheck = check ∘ iterative (strategy ∘ (dfs∘) ∘ depthSlice 5)

depthCheck :: Testable a ⇒ Int → a → IO ()
depthCheck = check ∘ strategy ∘ (dfs∘) ∘ (λd → depthSlice d d)
```

The *sparseCheck* function performs incremental limited discrepancy search on a randomly shuffled search space. Like *smallCheck* it is useful if there are very few counter examples but unlike *smallCheck* it is advancing and generates large test cases more quickly. The *discrCheck* function performs limited discrepancy search without increasing the discrepancy limit incrementally.

```
sparseCheck :: Testable a ⇒ Int → a → IO ()
sparseCheck = check ∘ iterative (λd →
                        randomised (λr →
                        strategy (dfs ∘ discrSlice 0 d ∘ shuffledST r)) 10)
discrCheck :: Testable a ⇒ Int → a → IO ()
discrCheck b = check (randomised (λr →
                        strategy (dfs ∘ discrSlice b b ∘ shuffledST r)) 10)
```

All test functions are defined in terms of *check* which takes a strategy as argument, employs it to enumerate tests, and processes the result to print found counter examples.

```
check :: Testable a ⇒ Strategy Test → a → IO ()
check s a = do hSetBuffering stdout NoBuffering
               s (getSearchTree (test a)) >>= checkAll 0
checkAll :: Int → [Test] → IO ()
checkAll n []     = putStrLn $ "\nOK, passed "
                               ++ show n ++ " tests."
checkAll n (t : ts) = maybe ((checkAll $! (n + 1)) ts) (printFail n) t
printFail :: Int → [String] → IO ()
printFail n args =
  do putStr ("\nFailure in test " ++ show (n + 1))
     if null args then putStrLn "."
                  else do putStrLn " for input:"
                          mapM_ putStrLn args
```

A.5.1 Parallel answers

This module provides combinators to constructs fair predicates. The binary operations for conjunction and disjunction are not sequential but evaluate both arguments stepwise interleaved. Properties formulated using these combinators are lazier than corresponding Boolean properties if one argument suffices to determine the result of a conjunction or disjunction.

```
module Answer (
  Answer, answer, fromAnswer,
  true, false, neg, (∧), (∨),
  and, or, all, any, elem, allDifferent
  ) where
import Prelude hiding (and, or, all, any, elem)
infixr 3 ∧
infixr 2 ∨
```

Internally, answers are represented by a data type similar to *Bool* but with an additional case for undecided answers that allows to suspend the evaluation of an answer. The data type is abstract and the exact structure of an answer is not directly observable from the outside.

```
data Answer = Yes | No | Undecided Answer
instance Show Answer where
  show a = if fromAnswer a then "true" else "false"
instance Eq Answer where
  a == b = fromAnswer a == fromAnswer b
```

The functions *answer* and *fromAnswer* convert between answers and Booleans, the latter demands the complete evaluation of an answer by stripping off all suspensions.

```
answer :: Bool → Answer
answer b = Undecided (if b then true else false)
fromAnswer :: Answer → Bool
fromAnswer Yes          = True
fromAnswer No           = False
fromAnswer (Undecided a) = fromAnswer a
```

There are combinators to construct primitive answers and for negation, conjunction, and disjunction.

```
true, false :: Answer
true = Yes; false = No

neg :: Answer → Answer
neg a = Undecided (negation a)
  where negation Yes           = No
        negation No            = Yes
        negation (Undecided a) = neg a
```

Conjunction and disjunction evaluate both arguments in an interleaved order.

```
(∧), (∨) :: Answer → Answer → Answer
a ∧ b = Undecided $ case (a, b) of
                      (Yes          , _          ) → b
                      (No           , _          ) → No
                      (_            , Yes        ) → a
                      (_            , No         ) → No
                      (Undecided x, Undecided y) → x ∧ y
a ∨ b = Undecided $ case (a, b) of
                      (Yes          , _          ) → Yes
                      (No           , _          ) → b
                      (_            , Yes        ) → Yes
                      (_            , No         ) → a
                      (Undecided x, Undecided y) → x ∨ y
```

Finally, we provide convenience functions to process lists of answers.

```
and, or :: [Answer] → Answer
and = foldr (∧) true
or  = foldr (∨) false

all, any :: (a → Answer) → [a] → Answer
all p = and ∘ map p
any p = or  ∘ map p

elem :: Eq a ⇒ a → [a] → Answer
elem x = any (answer ∘ (x==))

allDifferent :: Eq a ⇒ [a] → Answer
allDifferent []       = true
allDifferent (x : xs) = all (answer ∘ (x≠)) xs ∧ allDifferent xs
```

A.5.2 Search strategies

This module provides search strategies used by the GlassCheck tool to enumerate tests. Rather than defining fixed strategies, we define strategy combinators that can be combined with strategies from the *TreeSearch* module to perform bounded searches with an incrementally increasing limit or to run the same search on different shuffled versions of a tree.

We use lazy IO in the form of *unsafeInterleaveIO* to return the result of combined searches lazily.

```
module SearchStrategies (
  module TreeSearch,
  Strategy, strategy, iterative, randomised
  ) where
import Monad
import TreeSearch
import Unsafe (unsafeInterleaveIO)
type Strategy a = IO (SearchTree a) → IO [a]
strategy :: (SearchTree a → [a]) → Strategy a
strategy = fmap
iterative :: (Int → Strategy a) → Int → Strategy a
  -- iterative f n = runAll . zipWith f [0..n] . repeat
iterative f n a = do putStr "iterations:     "
                     runAll (map pass [0 .. n])
  where pass k =
          do putStr (replicate (length (show (k − 1))) (chr 8)
                       ++ show k)
             f k a
randomised :: (StdGen → Strategy a) → Int → Strategy a
randomised f n a = do rs ← fmap (take n ∘ splitGen) getStdGen
                      runAll (zipWith f rs (repeat a))
runAll :: [IO [a]] → IO [a]
runAll = fmap concat ∘ sequenceLazy
sequenceLazy :: [IO a] → IO [a]
sequenceLazy [ ]       = return [ ]
sequenceLazy (m : ms) =
  liftM2 (:) m (unsafeInterleaveIO (sequenceLazy ms))
```

A.5.3 Tree search

This module provides different functions to traverse search trees.

```
module TreeSearch (
  SearchTree, getSearchTree,
  StdGen, getStdGen, splitGen, shuffled,
  dfs, depthSlice, discrSlice, shuffledST
  ) where
import AllSolutions
import Random
```

We reexport depth-first search under a shorter name for convenience.

```
dfs :: SearchTree a → [a]
dfs = allValuesD
```

There are combinators to prune a search tree w.r.t. depth- or discrepancy and to shuffle a search tree.

```
depthSlice :: Int → Int → SearchTree a → SearchTree a
depthSlice _ _ Fail      = Fail
depthSlice w d (Val x)  | d ⩽ w     = Val x
                        | otherwise = Fail
depthSlice w d (Or ts)
  | d ⩽ 0     = Fail
  | otherwise = Or (map (depthSlice w (d − 1)) ts)
discrSlice :: Int → Int → SearchTree a → SearchTree a
discrSlice _ _ Fail      = Fail
discrSlice w d (Val x)  | d ⩽ w     = Val x
                        | otherwise = Fail
discrSlice w d (Or ts)
  | d ⩽ 0     = Fail
  | otherwise = Or (zipWith (discrSlice w) [d − 1, d − 2 .. 0] ts)
```

The shuffling function comes with auxiliary functions to split a random generator and to shuffle a list.

```
shuffledST :: StdGen → SearchTree a → SearchTree a
shuffledST _ Fail      = Fail
shuffledST _ (Val x) = Val x
```

```
shuffledST g (Or ts) = Or (shuffled r (zipWith shuffledST rs ts))
  where r : rs = splitGen g
splitGen :: StdGen → [StdGen]
splitGen g = l : splitGen r
  where (l, r) = split g
shuffled :: StdGen → [a] → [a]
shuffled g l = shuffleWithLen (randoms g) (length l) l
shuffleWithLen :: [Int] → Int → [a] → [a]
shuffleWithLen (r : rs) len xs
  | len == 0 = []
  | otherwise = z : shuffleWithLen rs (len − 1) (ys ++ zs)
where
  (ys, z : zs) = splitAt (abs r `mod` len) xs
```

A.6 Implementation of explicit sharing

In this section we present the efficient implementation of explicit sharing as described in Section 5.2.5. It is also available via the Haskell package database (Hackage) and the source code can be found along with the benchmark programs listed in Appendix A.7 online.[1]

```
module ExplicitSharing where
import Data.IntMap
import Control.Monad
import Unsafe.Coerce
```

Our approach is based on two type classes *Nondet* and *Sharing* which are interfaces to data with nondeterministic components and nondeterminism monads that support explicit sharing of such data respectively.

```
class MonadPlus m ⇒ Sharing m where
  share :: Nondet m a ⇒ m a → m (m a)
class MonadPlus m ⇒ Nondet m a where
  mapNondet :: (∀b.Nondet m b ⇒ m b → m (m b)) → a → m a
```

The *eval* function employs the *Nondet* class to expose all nested nondeterminism to the top level.

[1] `http://github.com/sebfisch/explicit-sharing/tree/0.1.1`

```
eval :: Nondet m a ⇒ a → m a
eval = mapNondet (λa → a >>= eval >>= return ∘ return)
```

We define a monad transformer *Lazy* as an optimised version of a continuation monad transformer applied to a reader monad transformer along with a function to evaluate *Lazy* computations in the base monad.

```
newtype Lazy m a =
  Lazy {fromLazy :: ∀w.(a → Store → m w) → Store → m w}
evalLazy :: Monad m ⇒ Nondet (Lazy m) a ⇒ Lazy m a → m a
evalLazy m = runLazy (m >>= eval)
runLazy :: Monad m ⇒ Lazy m a → m a
runLazy m = fromLazy m (λa _ → return a) (Store 1 empty)
```

The type constructor *Lazy m* is a monad and an instance of *MonadPlus* if the base type *m* is.

```
instance Monad m ⇒ Monad (Lazy m) where
  return x = Lazy (λc → c x)
  a >>= k  = Lazy (λc s → fromLazy a (λx → fromLazy (k x) c) s)
  fail msg = Lazy (λ_ _ → fail msg)
instance MonadPlus m ⇒ MonadPlus (Lazy m) where
  ∅       = Lazy (λ_ _ → ∅)
  a ⊕ b = Lazy (λc s → fromLazy a c s ⊕ fromLazy b c s)
```

The *Sharing* instance for the *Lazy* monad transformer uses the *memo* function to memoise nested nondeterministic data. This version of *memo* is a specialised version of the overloaded *memo* function shown in Section 5.2.5.

```
instance MonadPlus m ⇒ Sharing (Lazy m) where
  share a = memo (a >>= mapNondet share)
memo :: Lazy m a → Lazy m (Lazy m a)
memo a =
  Lazy (λc (Store key heap) →
    c (Lazy (λc s@(Store _ heap) →
        case Data.IntMap.lookup key heap of
          Just x    → c (typed x) s
          Nothing → fromLazy a
            (λx (Store other heap) →
              c x (Store other (insert key (Untyped x) heap))) s))
      (Store (key + 1) heap))
```

A value of type *Store* stores explicitly shared thunks and is threaded through *Lazy* computations.

$$\textbf{data}\ \mathit{Store} = \mathit{Store}\ \mathit{Int}\ (\mathit{IntMap}\ \mathit{Untyped})$$

In order to be able to store values of arbitrary types, we use an existential type to hide the actual type of stored values.

$$\begin{aligned}&\textbf{data}\ \mathit{Untyped} = \forall a.\mathit{Untyped}\ a\\&\mathit{typed} :: \mathit{Untyped} \to a\\&\mathit{typed}\ (\mathit{Untyped}\ x) = \mathit{unsafeCoerce}\ x\end{aligned}$$

A.7 Benchmarks for explicit sharing

We show Curry and Haskell versions of three different programs to evaluate the performance of our approach to lazy nondeterminism in Haskell. Permutation sort is a standard example for a generate-and-test algorithm, the naive reverse function performs a lot of deterministic pattern matching, and the functional logic version of the last function lets us examine how well logic variables and unification can be simulated via lazy nondeterminism.

A.7.1 Permutation sort

The Curry implementation of the permutation sort algorithm used for our experiments is as follows.

$$\begin{aligned}&\mathit{sort}\ l \mid \mathit{isSorted}\ p = p\\&\quad \textbf{where}\ p = \mathit{permute}\ l\\&\mathit{isSorted}\ [\,] = \mathit{True}\\&\mathit{isSorted}\ (x : xs) = \mathit{isSorted}'\ x\ xs\\&\mathit{isSorted}'\ \_\ [\,] = \mathit{True}\\&\mathit{isSorted}'\ x\ (y : ys) = x \leqslant y \wedge \mathit{isSorted}'\ y\ ys\\&\mathit{permute}\ [\,] = [\,]\\&\mathit{permute}\ (x : xs) = \mathit{insert}\ x\ (\mathit{permute}\ xs)\\&\mathit{insert}\ x\ xs = x : xs\\&\mathit{insert}\ x\ (y : ys) = y : \mathit{insert}\ x\ ys\end{aligned}$$

For the Haskell version we need to import the explicit sharing module defined in Appendix A.6.

```
import Control.Monad
import ExplicitSharing
import Prelude hiding (reverse)
```

We define monadic variants for all presented Curry operations. We occasionally use a flipped version =<< of the bind operator >>= which resembles monadic application.

```
sort :: Sharing m => m (List m Int) -> m (List m Int)
sort l = do xs <- share (perm =<< l)
            True <- isSorted =<< xs
            xs

isSorted :: Monad m => List m Int -> m Bool
isSorted Nil = return True
isSorted (Cons mx mxs) = isSorted' mx =<< mxs

isSorted' :: Monad m => m Int -> List m Int -> m Bool
isSorted' _ Nil                = return True
isSorted' mx (Cons my mys) =
  do x <- mx
     y <- my
     if x <= y then isSorted' (return y) =<< mys
               else return False

perm :: MonadPlus m => List m a -> m (List m a)
perm Nil         = nil
perm (Cons x xs) = insert x (perm =<< xs)

insert :: MonadPlus m => m a -> m (List m a) -> m (List m a)
insert e l = cons e l
         ++ do Cons x xs <- l
               cons x (insert e xs)
```

A.7.2 Naive reverse

We use the naive reverse function to measure the overhead of our approach for purely functional computations. In Curry, we use the usual recursive implementation in terms of the append function ++.

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

The monadic Haskell version is as follows.

```
reverse :: Monad m ⇒ List m a → m (List m a)
reverse Nil          = nil
reverse (Cons x xs) = append (reverse =≪ xs) (cons x nil)

append :: Monad m ⇒ m (List m a) → m (List m a) → m (List m a)
append mxs ys = do xs ← mxs; appendLists xs ys

appendLists :: Monad m ⇒ List m a → m (List m a) → m (List m a)
appendLists Nil          ys = ys
appendLists (Cons x xs) ys = cons x (append xs ys)
```

A.7.3 Functional logic last

Finally, we can investigate the efficiency of simulating logic variables and unification via lazy nondeterminism by means of the functional logic implementation of the last function:

```
last :: [a] → a
last l | l =:= xs ++ [x]
       = x
  where x, xs free
```

The monadic Haskell version of *last* uses a nondeterministic generator for lists of Booleans instead of a logic variable.

```
last :: Sharing m ⇒ m (List m Bool) → m Bool
last l = do x ← share freeBool
            l =:= append freeBoolList (cons x nil)
            x

freeBool :: MonadPlus m ⇒ m Bool
freeBool = return False ⊕ return True

freeBoolList :: MonadPlus m ⇒ m (List m Bool)
freeBoolList = nil ⊕ cons freeBool freeBoolList
```

The constraint equality operator =:= is implemented as monadic equality.

```
(=:=) :: MonadPlus m ⇒ m (List m Bool) → m (List m Bool) → m ()
mxs =:= mys = do xs ← mxs; ys ← mys; eqBoolList xs ys

eqBoolList :: MonadPlus m ⇒ List m Bool → List m Bool → m ()
eqBoolList Nil          Nil          = return ()
eqBoolList (Cons x xs) (Cons y ys) = do True ← liftM2 (==) x y
                                        xs =:= ys
eqBoolList _            _            = ∅
```

A.7.4 Nondeterministic lists

The Haskell versions of our benchmarks all use a data type for nondeterministic lists which is defined as follows.

```
data List m a = Nil | Cons (m a) (m (List m a))
nil :: Monad m ⇒ m (List m a)
nil = return Nil
cons :: Monad m ⇒ m a → m (List m a) → m (List m a)
cons x xs = return (Cons x xs)
```

In order to be able to explicitly share nondeterministic lists of numbers or Booleans we need appropriate *Nondet* instances.

```
instance Nondet m a ⇒ Nondet m (List m a) where
  mapNondet _ Nil          = return Nil
  mapNondet f (Cons x xs) = do y ← f x
                               ys ← f xs
                               return (Cons y ys)
instance MonadPlus m ⇒ Nondet m Int where
  mapNondet _ = return
instance MonadPlus m ⇒ Nondet m Bool where
  mapNondet _ = return
```

B Proofs

B.1 Functor laws for $(a \to)$ instance

This is a proof of the functor laws

$$\begin{aligned} &\mathit{fmap}\ \mathit{id} &&\equiv \mathit{id} \\ &\mathit{fmap}\ (f \circ g) &&\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g \end{aligned}$$

for the *Functor* instance

$$\begin{aligned} &\textbf{instance}\ \mathit{Functor}\ (a \to)\ \textbf{where} \\ &\quad \mathit{fmap} = (\circ) \end{aligned}$$

The laws are a consequence of the fact that functions form a monoid under composition with the identity element id.

$$\begin{aligned} & \mathit{fmap}\ \mathit{id}\ h \\ \equiv & \quad \{\ \text{definition of } \mathit{fmap}\ \} \\ & \mathit{id} \circ h \\ \equiv & \quad \{\ \text{definition of } (\circ)\ \} \\ & \lambda x \to \mathit{id}\ (h\ x) \\ \equiv & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & \lambda x \to h\ x \\ \equiv & \quad \{\ \text{expansion}\ \} \\ & h \\ \equiv & \quad \{\ \text{definition of } \mathit{id}\ \} \\ & \mathit{id}\ h \end{aligned}$$

This proof makes use of the identity $(\lambda x \to f\ x) \equiv f$ for every function f. The second law is a bit more involved as it relies on associativity for function composition.

$$\begin{aligned} & \mathit{fmap}\ (f \circ g)\ h \\ \equiv & \quad \{\ \text{definition of } \mathit{fmap}\ \} \\ & (f \circ g) \circ h \\ \equiv & \quad \{\ \text{associativity of } (\circ)\ \} \\ & f \circ (g \circ h) \end{aligned}$$

$$
\begin{aligned}
&\equiv \quad \{ \text{ definition of } \mathit{fmap} \text{ (twice) } \} \\
&\quad \mathit{fmap}\ f\ (\mathit{fmap}\ g\ h) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad (\lambda x \to \mathit{fmap}\ f\ (\mathit{fmap}\ g\ x))\ h \\
&\equiv \quad \{ \text{ definition of } (\circ) \} \\
&\quad (\mathit{fmap}\ f \circ \mathit{fmap}\ g)\ h
\end{aligned}
$$

Now it is only left to verify that function composition is indeed associative:

$$
\begin{aligned}
&\quad (f \circ g) \circ h \\
&\equiv \quad \{ \text{ definition of } (\circ) \text{ (twice) } \} \\
&\quad \lambda x \to (\lambda y \to f\ (g\ y))\ (h\ x) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad \lambda x \to f\ (g\ (h\ x)) \\
&\equiv \quad \{ \text{ reduction } \} \\
&\quad \lambda x \to f\ ((\lambda y \to g\ (h\ y))\ x) \\
&\equiv \quad \{ \text{ definition of } (\circ) \text{ (twice) } \} \\
&\quad f \circ (g \circ h)
\end{aligned}
$$

B.2 Monad laws for *Tree* instance

This is a proof of the monad laws

$$
\begin{aligned}
&\mathit{return}\ x \gg\!= f \quad \equiv f\ x \\
&m \gg\!= \mathit{return} \quad \equiv m \\
&(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{aligned}
$$

for the *Monad* instance

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad\ Tree}\ \textbf{where} \\
&\quad \mathit{return} = \mathit{Leaf} \\
&\quad t \gg\!= f = \mathit{mergeTrees}\ (\mathit{fmap}\ f\ t) \\
&\mathit{mergeTrees} :: \mathit{Tree}\ (\mathit{Tree}\ a) \to \mathit{Tree}\ a \\
&\mathit{mergeTrees}\ \mathit{Empty} \quad = \mathit{Empty} \\
&\mathit{mergeTrees}\ (\mathit{Leaf}\ t) \quad = t \\
&\mathit{mergeTrees}\ (\mathit{Fork}\ l\ r) = \mathit{Fork}\ (\mathit{mergeTrees}\ l)\ (\mathit{mergeTrees}\ r)
\end{aligned}
$$

for the data type

$$
\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)
$$

The left-identity law follows from the definitions of the functions *return*, $\gg\!=$, *fmap*, and *mergeTrees*.

$$
\begin{aligned}
& \mathit{return}\ x \gg\!= f \\
\equiv\ & \quad \{\ \text{definitions of } \mathit{return} \text{ and } \gg\!= \ \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ (\mathit{Leaf}\ x)) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{fmap} \ \} \\
& \mathit{mergeTrees}\ (\mathit{Leaf}\ (f\ x)) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{mergeTrees} \ \} \\
& f\ x
\end{aligned}
$$

We prove the right-identity law by induction over the structure of m. The *Empty* case follows from the observation that $\mathit{Empty} \gg\!= f \equiv \mathit{Empty}$ for every function f, i.e., also for $f \equiv \mathit{return}$.

$$
\begin{aligned}
& \mathit{Empty} \gg\!= f \\
\equiv\ & \quad \{\ \text{definition of } \gg\!= \ \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ \mathit{Empty}) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{fmap} \ \} \\
& \mathit{mergeTrees}\ \mathit{Empty} \\
\equiv\ & \quad \{\ \text{definition of } \mathit{mergeTrees} \ \} \\
& \mathit{Empty}
\end{aligned}
$$

The *Leaf* case follows from the left-identity law because $\mathit{return} \equiv \mathit{Leaf}$.

$$
\begin{aligned}
& \mathit{Leaf}\ x \gg\!= \mathit{return} \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return} \ \} \\
& \mathit{return}\ x \gg\!= \mathit{return} \\
\equiv\ & \quad \{\ \text{first monad law} \ \} \\
& \mathit{return}\ x \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return} \ \} \\
& \mathit{Leaf}\ x
\end{aligned}
$$

The *Fork* case makes use of the induction hypothesis and the observation that $\mathit{Fork}\ l\ r \gg\!= f \equiv \mathit{Fork}\ (l \gg\!= f)\ (r \gg\!= f)$

$$
\begin{aligned}
& \mathit{Fork}\ l\ r \gg\!= f \\
\equiv\ & \quad \{\ \text{definition of } \gg\!= \ \} \\
& \mathit{mergeTrees}\ (\mathit{fmap}\ f\ (\mathit{Fork}\ l\ r)) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{fmap} \ \} \\
& \mathit{mergeTrees}\ (\mathit{Fork}\ (\mathit{fmap}\ f\ l)\ (\mathit{fmap}\ f\ r)) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{mergeTrees} \ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Fork}\ (\mathit{mergeTrees}\ (\mathit{fmap}\ f\ l))\ (\mathit{mergeTrees}\ (\mathit{fmap}\ f\ r)) \\
\equiv\ & \quad \{\ \text{definition of } \gg= \text{ (twice)}\ \} \\
& \mathit{Fork}\ (l \gg= f)\ (r \gg= f)
\end{aligned}
$$

Now we can apply the induction hypothesis.

$$
\begin{aligned}
& \mathit{Fork}\ l\ r \gg= \mathit{return} \\
\equiv\ & \quad \{\ \text{previous derivation}\ \} \\
& \mathit{Fork}\ (l \gg= \mathit{return})\ (r \gg= \mathit{return}) \\
\equiv\ & \quad \{\ \text{induction hypothesis (twice)}\ \} \\
& \mathit{Fork}\ l\ r
\end{aligned}
$$

Finally we prove associativity of $\gg=$ by structural induction. The *Empty* case follows from the above observation that $\mathit{Empty} \gg= f \equiv \mathit{Empty}$ for every function f.

$$
\begin{aligned}
& (\mathit{Empty} \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{above observation for } \mathit{Empty} \text{ (twice)}\ \} \\
& \mathit{Empty} \\
\equiv\ & \quad \{\ \text{above observation for } \mathit{Empty}\ \} \\
& \mathit{Empty} \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}
$$

The *Leaf* case follows again from the first monad law.

$$
\begin{aligned}
& (\mathit{Leaf}\ y \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return}\ \} \\
& (\mathit{return}\ y \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{first monad law}\ \} \\
& f\ y \gg= g \\
\equiv\ & \quad \{\ \text{first monad law}\ \} \\
& \mathit{return}\ y \gg= (\lambda x \to f\ x \gg= g) \\
\equiv\ & \quad \{\ \text{definition of } \mathit{return}\ \} \\
& \mathit{Leaf}\ y \gg= (\lambda x \to f\ x \gg= g)
\end{aligned}
$$

The *Fork* case uses the identity $\mathit{Fork}\ l\ r \gg= f \equiv \mathit{Fork}\ (l \gg= f)\ (r \gg= f)$ that we proved above and the induction hypothesis.

$$
\begin{aligned}
& (\mathit{Fork}\ l\ r \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{property of } \gg=\ \} \\
& \mathit{Fork}\ (l \gg= f)\ (r \gg= f) \gg= g \\
\equiv\ & \quad \{\ \text{property pf } \gg=\ \} \\
& \mathit{Fork}\ ((l \gg= f) \gg= g)\ ((r \gg= f) \gg= g) \\
\equiv\ & \quad \{\ \text{induction hypothesis}\ \}
\end{aligned}
$$

$$
\begin{array}{ll}
 & \mathit{Fork}\ (l \gg\!= (\lambda x \to f\ x \gg\!= g))\ (r \gg\!= (\lambda x \to f\ x \gg\!= g)) \\
\equiv & \quad \{\ \text{property of } \gg\!= \ \} \\
 & \mathit{Fork}\ l\ r \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{array}
$$

This finishes the proof of the three monad laws for the *Tree* instance.

B.3 Laws for *CPS* type

In this section we prove various laws about the *CPS* type defined in Section 5.1.2 as follows.

$$\textbf{newtype}\ \mathit{CPS}\ c\ a = \mathit{CPS}\ \{\ (\gg\!-) :: \forall b.(a \to c\ b) \to c\ b\ \}$$

We start with a preliminary observation which we will use in the subsequent proofs. For all $a :: \mathit{CPS}\ c\ a$ we have the identity $\mathit{CPS}\ (\lambda k \to a \gg\!- k) \equiv a$. Let $f :: (a \to c\ b) \to c\ b$ such that $a = \mathit{CPS}\ f$:

$$
\begin{array}{ll}
 & \mathit{CPS}\ (\lambda k \to a \gg\!- k) \\
\equiv & \quad \{\ a = \mathit{CPS}\ f\ \} \\
 & \mathit{CPS}\ (\lambda k \to \mathit{CPS}\ f \gg\!- k) \\
\equiv & \quad \{\ \text{record selection}\ \} \\
 & \mathit{CPS}\ (\lambda k \to f\ k) \\
\equiv & \quad \{\ \text{expansion}\ \} \\
 & \mathit{CPS}\ f \\
\equiv & \quad \{\ a = \mathit{CPS}\ f\ \} \\
 & a
\end{array}
$$

B.3.1 Monad laws

Now, we verify the monad laws

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

for the *Monad* instance

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ (\mathit{CPS}\ c)\ \textbf{where} \\
\quad \mathit{return}\ x = \mathit{CPS}\ (\lambda k \to k\ x) \\
\quad a \gg\!= f\ \ = \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x \to f\ x \gg\!- k)
\end{array}
$$

We use the definitions of *return* and $\gg\!=$ to prove the first monad law.

$$
\begin{array}{ll}
 & \mathit{return}\ x \gg\!= f \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & \mathit{CPS}\ (\lambda k \to \mathit{return}\ x \gg\!- \lambda y \to f\ y \gg\!- k) \\
\equiv & \quad \{\ \text{definition of } \mathit{return} \ \} \\
 & \mathit{CPS}\ (\lambda k \to (\mathit{CPS}\ (\lambda k' \to k'\ x)) \gg\!- \lambda y \to f\ y \gg\!- k) \\
\equiv & \quad \{\ \text{record selection and reduction} \ \} \\
 & \mathit{CPS}\ (\lambda k \to f\ x \gg\!- k) \\
\equiv & \quad \{\ \text{preliminary observation} \ \} \\
 & f\ x
\end{array}
$$

The second monad law is verified similarly:

$$
\begin{array}{ll}
 & a \gg\!= \mathit{return} \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x \to \mathit{return}\ x \gg\!- k) \\
\equiv & \quad \{\ \text{definition of } \mathit{return} \text{ and record selection} \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x \to k\ x) \\
\equiv & \quad \{\ \text{expansion} \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- k) \\
\equiv & \quad \{\ \text{preliminary observation} \ \} \\
 & a
\end{array}
$$

Finally, we show the associative law for $\gg\!=$.

$$
\begin{array}{ll}
 & (a \gg\!= f) \gg\!= g \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & \mathit{CPS}\ (\lambda k \to (a \gg\!= f) \gg\!- \lambda x \to g\ x \gg\!- k) \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & \mathit{CPS}\ (\lambda k \to \mathit{CPS}\ (\lambda k' \to a \gg\!- \lambda x' \to f\ x' \gg\!- k') \gg\!- \lambda x \to g\ x \gg\!- k) \\
\equiv & \quad \{\ \text{record selection and reduction} \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x' \to f\ x' \gg\!- \lambda x \to g\ x \gg\!- k) \\
\equiv & \quad \{\ \text{record selection and reduction} \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x' \to \mathit{CPS}\ (\lambda k' \to f\ x' \gg\!- \lambda x \to g\ x \gg\!- k') \gg\!- k) \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & \mathit{CPS}\ (\lambda k \to a \gg\!- \lambda x' \to (f\ x' \gg\!= g) \gg\!- k) \\
\equiv & \quad \{\ \text{definition of } \gg\!= \ \} \\
 & a \gg\!= (\lambda x \to f\ x \gg\!= g)
\end{array}
$$

B.3.2 *MonadPlus* laws

We now consider the *MonadPlus* instance for the *CPS* type.

$$\begin{aligned}&\textbf{instance}\ \mathit{Nondet}\ n \Rightarrow \mathit{MonadPlus}\ (\mathit{CPS}\ n)\ \textbf{where}\\ &\quad \emptyset \quad = \mathit{CPS}\ (\lambda\_ \rightarrow \mathit{failure})\\ &\quad a \oplus b = \mathit{CPS}\ (\lambda k \rightarrow \mathit{choice}\ (a \gg\!- k)\ (b \gg\!- k))\end{aligned}$$

We first address the interaction of $\emptyset$ and $\oplus$ with $\gg\!=$:

$$\begin{aligned}&\emptyset \quad\quad \gg\!= f \equiv \emptyset\\ &(a \oplus b) \gg\!= f \equiv (a \gg\!= f) \oplus (b \gg\!= f)\end{aligned}$$

The $\emptyset$ combinator causes $\gg\!=$ to fail.

$$\begin{aligned}&\quad \emptyset \gg\!= f\\ &\equiv \quad \{\ \text{definition of } \gg\!=\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \emptyset \gg\!- \lambda x \rightarrow f\ x \gg\!- k)\\ &\equiv \quad \{\ \text{definition of } \emptyset\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{CPS}\ (\lambda\_ \rightarrow \mathit{failure}) \gg\!- \lambda x \rightarrow f\ x \gg\!- k)\\ &\equiv \quad \{\ \text{record selection and reduction}\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{failure})\\ &\equiv \quad \{\ \text{definition of } \emptyset\ \}\\ &\quad \emptyset\end{aligned}$$

The $\gg\!=$ combinator distributes over $\oplus$.

$$\begin{aligned}&\quad (a \oplus b) \gg\!= f\\ &\equiv \quad \{\ \text{definition of } \gg\!=\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow (a \oplus b) \gg\!- \lambda x \rightarrow f\ x \gg\!- k)\\ &\equiv \quad \{\ \text{definition of } \oplus\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{CPS}\ (\lambda k' \rightarrow \mathit{choice}\ (a \gg\!- k')\ (b \gg\!- k')) \gg\!- \lambda x \rightarrow f\ x \gg\!- k)\\ &\equiv \quad \{\ \text{record selection and reduction}\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{choice}\ (a \gg\!- \lambda x \rightarrow f\ x > -\ k)\ (b \gg\!- \lambda x \rightarrow f\ x \gg\!- k))\\ &\equiv \quad \{\ \text{record selection and reduction}\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{choice}\ (\mathit{CPS}\ (\lambda k' \rightarrow a \gg\!- \lambda x \rightarrow f\ x \gg\!- k') \gg\!- k)\\ &\qquad\qquad\qquad\qquad (\mathit{CPS}\ (\lambda k' \rightarrow b \gg\!- \lambda x \rightarrow f\ x \gg\!- k') \gg\!- k))\\ &\equiv \quad \{\ \text{definition of } \gg\!=\ \text{(twice)}\ \}\\ &\quad \mathit{CPS}\ (\lambda k \rightarrow \mathit{choice}\ ((a \gg\!= f) \gg\!- k)\ ((b \gg\!= f) \gg\!- k))\\ &\equiv \quad \{\ \text{definition of } \oplus\ \}\\ &\quad (a \gg\!= f) \oplus (b \gg\!= f)\end{aligned}$$

The *CPS* type inherits the monoid properties from the underlying *Nondet* instance.

$$\begin{aligned}&\emptyset \oplus a \quad\quad \equiv a\\ &a \oplus \emptyset \quad\quad \equiv a\\ &(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)\end{aligned}$$

According to the identity laws, we only show that $\emptyset$ is a left identity of $\oplus$ if *failure* is a left identity for *choice*. The proof for the right identity is analogous.

$$
\begin{aligned}
& \emptyset \oplus a \\
\equiv \quad & \{ \text{ definition of } \oplus \ \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (\emptyset \gg\!- k)\ (a \gg\!- k)) \\
\equiv \quad & \{ \text{ definition of } \emptyset \ \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (\mathit{CPS}\ (\lambda \_ \to \mathit{failure}) \gg\!- k)\ (a \gg\!- k)) \\
\equiv \quad & \{ \text{ record selection and reduction } \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ \mathit{failure}\ (a \gg\!- k)) \\
\equiv \quad & \{ \mathit{failure} \text{ is left identity for } \mathit{choice} \ \} \\
& \mathit{CPS}\ (\lambda k \to a \gg\!- k) \\
\equiv \quad & \{ \text{ preliminary observation } \} \\
& a
\end{aligned}
$$

If *choice* is associative then $\oplus$ also is.

$$
\begin{aligned}
& (a \oplus b) \oplus c \\
\equiv \quad & \{ \text{ definition of } \oplus \ \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ ((a \oplus b) \gg\!- k)\ (c \gg\!- k)) \\
\equiv \quad & \{ \text{ definition of } \oplus \ \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (\mathit{CPS}\ (\lambda k' \to \mathit{choice}\ (a \gg\!- k')\ (b \gg\!- k')) \gg\!- k) \\
& \qquad\qquad\qquad (c \gg\!- k)) \\
\equiv \quad & \{ \text{ record selection and reduction } \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (\mathit{choice}\ (a \gg\!- k)\ (b \gg\!- k))\ (c \gg\!- k)) \\
\equiv \quad & \{ \mathit{choice} \text{ is associative } \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (a \gg\!- k)\ (\mathit{choice}\ (b \gg\!- k)\ (c \gg\!- k))) \\
\equiv \quad & \{ \text{ record selection and reduction } \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (a \gg\!- k) \\
& \qquad\qquad\qquad (\mathit{CPS}\ (\lambda k' \to \mathit{choice}\ (b \gg\!- k')\ (c \gg\!- k')) \gg\!- k)) \\
\equiv \quad & \{ \text{ definition of } \oplus \ \} \\
& \mathit{CPS}\ (\lambda k \to \mathit{choice}\ (a \gg\!- k)\ ((b \oplus c) \gg\!- k)) \\
\equiv \quad & \{ \text{ definition of } \oplus \ \} \\
& a \oplus (b \oplus c)
\end{aligned}
$$

Bibliography

Adelson-Velskii, Georgii Maximovich, and Evgenii Mikhailovich Landis. 1962. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, vol. 146, 263–266.

Albert, Elvira, Miguel Gómez-Zamalloa, Laurent Hubert, and German Puebla. 2007. Verification of Java bytecode using analysis and transformation of logic programs. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*. LNCS, Springer-Verlag.

Albert, Elvira, Michael Hanus, Frank Huch, Javier Oliver, and German Vidal. 2005. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation* 40(1):795–829.

Antoy, Sergio. 2001. Constructor-based conditional narrowing. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 199–206. ACM Press.

Antoy, Sergio, Rachid Echahed, and Michael Hanus. 2000. A needed narrowing strategy. *Journal of the ACM* 47(4):776–822.

Antoy, Sergio, and Michael Hanus. 2002. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, 67–87. Springer LNCS 2441.

Antoy, Sergio, and Michael Hanus. 2006. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the International Conference on Logic Programming (ICLP 2006)*, 87–101. Springer LNCS 4079.

Antoy, Sergio, and Michael Hanus. 2009. Set functions for functional logic programming. In *PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, 73–82. New York, NY, USA: ACM.

Braßel, Bernd, Sebastian Fischer, and Frank Huch. 2008. Declaring numbers. *Electronic Notes in Theoretical Computer Science* 216:111 – 124. Proceedings of the 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2007).

Braßel, Bernd, Michael Hanus, and Frank Huch. 2004. Encapsulating non-determinism in functional logic computations. In *Journal of functional and logic programming*, vol. 6. EAPLS.

Braßel, Bernd, and Frank Huch. 2009. The Kiel Curry System KiCS. In *WLP 2007*, 195–205.

Chitil, Olaf, Colin Runciman, and Malcolm Wallace. 2003. Transforming Haskell for tracing. In *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, ed. Ricardo Pena and Thomas Arts, 165–181. LNCS 2670. Madrid, Spain, 16–18 September 2002.

Christiansen, Jan, and Sebastian Fischer. 2008. EasyCheck – test data for free. In *FLOPS '08: Proceedings of the 9th International Symposium on Functional and Logic Programming*. Springer LNCS 4989.

Claessen, Koen, and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, 268–279. New York, NY, USA: ACM.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to algorithms*. The MIT Press, Cambridge, MA.

Damas, Luis, and Robin Milner. 1982. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 207–212. New York, NY, USA: ACM.

Danielsson, Nils Anders, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 206–217. New York, NY, USA: ACM.

Degrave, François, Tom Schrijvers, and Wim Vanhoof. 2009. Automatic generation of test inputs for Mercury. In *Logic-Based Program Synthesis and Transformation: 18th International Symposium, LOPSTR 2008, Revised Selected Papers*, 71–86. Springer-Verlag.

Dijkstra, Edsger W. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11(3):147–148.

Fischer, Sebastian. 2009a. Declarative programming overview. `http://www-ps.informatik.uni-kiel.de/~sebf/haskell/fp-overview.html`, `http://www-ps.informatik.uni-kiel.de/~sebf/haskell/dp-overview.html`.

Fischer, Sebastian. 2009b. Fair predicates library. available online at: `http://sebfisch.github.com/fair-predicates/`.

Fischer, Sebastian. 2009c. Reinventing Haskell backtracking. In *4. Arbeitstagung Programmiersprachen (ATPS'09)*.

Fischer, Sebastian, Oleg Kiselyov, and Chung-chieh Shan. 2009. Purely functional lazy non-deterministic programming. In *ICFP'09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM Press.

Fischer, Sebastian, and Herbert Kuchen. 2007. Systematic generation of glass-box test cases for functional logic programs. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 63–74. New York, NY, USA: ACM Press.

Fischer, Sebastian, and Herbert Kuchen. 2008. Data-flow testing of declarative programs. In *ICFP'08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM Press.

Gill, Andy, and Colin Runciman. 2007. Haskell Program Coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, 1–12. New York, NY, USA: ACM.

González-Moreno, Juan Carlos, Maria Teresa Hortalá-González, Francisco Javier López-Fraguas, and Mario Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40(1):47–87.

Hackage. Haskell package database. `http://hackage.haskell.org/`.

Hanus, Michael. 2006. Curry: An integrated functional logic language (vers. 0.8.2). `http://www.curry-language.org`.

Hanus, Michael. 2007. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, 45–75. Springer LNCS 4670.

Hanus, Michael, and Frank Steiner. 1998. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, 374–390. Springer LNCS 1490.

Harvey, William D., and Matthew L. Ginsberg. 1995. Limited discrepancy search. 607–613. Morgan Kaufmann.

Hennessy, Matthew, and Edward A. Ashcroft. 1977. Parameter-passing mechanisms and nondeterminism. In *STOC '77: Proceedings of the ninth annual ACM Symposium on Theory of Computing*, 306–311. New York, NY, USA: ACM.

Hinze, Ralf. 2000. Deriving backtracking monad transformers (functional pearl). In *ICFP '00: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, 186–197. New York, NY, USA: ACM.

Hudak, Paul. 1989. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.* 21(3):359–411.

Hughes, John. 1989. Why functional programming matters. *Computer Journal* 32(2):98–107.

Jaffar, Joxan, and Jean-Louis Lassez. 1987. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 111–119. New York, NY, USA: ACM.

Jones, Mark P. 1993. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 52–61. New York, NY, USA: ACM.

Kiselyov, Oleg, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP'05: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, 192–203.

Koopman, Pieter, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. 2002. Gast: Generic automated software testing. In *The 14th International workshop on the Implementation of Functional Languages, IFL'02, Selected Papers*, ed. R. Peña, vol. 2670 of *LNCS*, 84–100. Madrid, Spain, Springer.

Langley, Pat. 1992. Systematic and nonsystematic search strategies. In *Proceedings of the first International Conference on Artificial Intelligence Planning Systems*, 145–152. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Lembeck, Christoph, Rafael Caballero, Roger Müller, and Herbert Kuchen. 2004. Constraint solving for generating glass-box test cases. In *Proceedings of International Workshop on Functional and (Constraint) Logic Programming (WFLP)*, 19–32.

Lin, Chuan-kai. 2006. Programming monads operationally with Unimo. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, 274–285.

Lindblad, Fredrik. 2007. Property directed generation of first-order test data. `http://cs.shu.edu/tfp2007/drafts/71.pdf`.

López-Fraguas, Francisco Javier, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2007. A simple rewrite notion for call-time choice semantics. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 197–208. New York, NY, USA: ACM Press.

López-Fraguas, Francisco Javier, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández. 2008. Rewriting and call-time choice: The HO case. In *FLOPS '08: Proceedings of the 9th International Symposium on Functional and Logic Programming*, 147–162. Springer LNCS 4989.

Müller, Roger, Christoph Lembeck, and Herbert Kuchen. 2004. A symbolic Java virtual machine for test-case generation. In *Proceedings IASTED*.

Naylor, Matthew, Emil Axelsson, and Colin Runciman. 2007. A functional-logic library for wired. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, 37–48. New York, NY, USA: ACM.

Odersky, Martin, Philip Wadler, and Martin Wehr. 1995. A second look at overloading. In *FPCA '95: Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, 135–146. New York, NY, USA: ACM.

Okasaki, Chris. 1996. Purely functional data structures. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Available as Technical Report CMU-CS-96-177.

Peyton Jones, Simon, et al. 2003. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming* 13(1):0–255. `http://www.haskell.org/definition/`.

Peyton-Jones, Simon L., and Philip Wadler. 1993. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 71–84. New York, NY, USA: ACM.

Rabhi, Fethi, and Guy Lapalme. 1999. *Algorithms – a functional programming approach*. Addison Wesley.

Rabin, Michael O., and Dana Scott. 1959. Finite automata and their decision problems. *IBM Journal of Research and Development* 3:114–125.

Reddit. 2009. Comments on declarative programming overview. `http://www.reddit.com/r/haskell/comments/8xzsx/sebastian_-fischer_a_brief_overview_on_functional/`, `http://www.reddit.com/r/haskell/comments/8zqm4/sebfisch_-declarative_programming_overview/`.

Runciman, Colin, Matthew Naylor, and Fredrik Lindblad. 2008. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN Symposium on Haskell*, 37–48. New York, NY, USA: ACM.

Slagle, James R. 1974. Automated theorem-proving for theories with simplifiers commutativity, and associativity. *Journal of the ACM* 21(4):622–642.

Spivey, Michael. 2006. Algebras for combinatorial search. In *Workshop on mathematically structured functional programming*.

Tolmach, Andrew, and Sergio Antoy. 2003. A monadic semantics for core Curry. In *WFLP 2003*, 33–46. Valencia, Spain.

Tolmach, Andrew, Sergio Antoy, and Marius Nita. 2004. Implementing functional logic languages using multiple threads and stores. In *ICFP'04: Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming*, 90–102.

Wadler, Philip. 1990. Comprehending monads. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 61–78. New York, NY, USA: ACM.

Wadler, Philip. 1995. Monads for functional programming. In *Advanced functional programming, first international spring school on advanced functional programming techniques—tutorial text*, 24–52. London, UK: Springer-Verlag.

Wadler, Philip, and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 60–76. New York, NY, USA: ACM.

Wadsworth, Christopher Peter. 1971. Semantics and pragmatics of the lambda calculus. Ph.D. thesis, Programming Research Group, Oxford University.

Index

abstraction, 5, 6, 8, 13, 16, 111, 157
advancing strategy, 43
anonymous function, *7*
approximation, 10
assembly language, 5
assignment, 5
AVL tree, *72*

backtracking, 22, 29, 111, 113, 118
balanced strategy, 43
binary leaf tree, 15
black-box tests, 35, *70*
Boolean values, 1
breadth-first search, 30, 42, 120

call-time choice, 25, 26, 127, 132, 136, 139
class constraint, 12
code coverage, *79*
  call coverage, 82, 93
  criterion, *79*, 104
  rule coverage, 82
code reuse, 9, 10, 13, 16, 21, 33
complete strategy, 43
computational effect, 111
constraints, 31
  equality, 20
  finite domain, 31
constructor, 1
continuation, 116, 150
control flow, 81, 91, 104
core language, 96
Curry, 1, 19
currying, 8

data flow, 84, 89, 93, 104
data type, 1
declarative programming, 5
def-use chain, 84, 86, 89, 93, 94

depth-bound search, 66
depth-first search, 29, 42, 111
diagonalisation, 48
difference list, 114, 118
discrepancy search, 66
diverging computation, 9
do-notation, 17, 27, 59
dynamic programming, *74*

equational reasoning, 13, 14, 19, 35, 135
evaluation order, 6
explicit sharing, 128, 131, 132
expression coverage, 80

failure, 21
failure continuation, 119
fair predicates, *67*
free variable, *see* logic variable
function, 1
function application, *7*
function composition, *7*, 15
functional logic programming, 19
functional programming, 6
functor, 14

generate-and-test, 26, 127, 129
glass-box tests, 35, 60, *70*, *79*

Haskell, 1
head-normal form, 88
heap, 39, 51, *70*
heap sort, *70*
higher-order function, *7*, 74, 86, 99, 102

infinite data structure, 10, 128, 134
input/output, 16
IO monad, 16
iterative deepening search, 43, 121

iterative sampling, 77

Kruskal's algorithm, 73

labeled field, *see* record syntax
lambda abstraction, 7, 94, 99
lazy evaluation, 9, 22, 26, 48, 60, 64, 68, 88, 127, 128, 157
level diagonalisation, 48
list comprehension, 27
list monad, 17
logic variable, 20, 31, 40, 52
logic variables, 23, 62

matrix multiplication, 72
memoisation, 141
minimum spanning tree, 73
modularity, 10, 157
monad, 16, 111, 117, 127
  laws, 18, 112, 135, 137
  laziness, 127
multi-paradigm language, 20

narrowing, 20, 23
newtype declaration, 41
non-strictness, 128, 129, 133, 136
nondeterminism, 21, 35, 111, 127, 128, 157
  generator, 24, 38, 52, 62, 157
  laziness, 23, 61, 67
nondeterministic operation, 22, 37

overlapping rules, 22, 23
overloading, 11, 157

partial application, 9, 94, 99
partial function, 3
pattern matching, 2
permutation, 23, 26, 38, 129
polymorphism, 7
probability, 46
program transformation, 88, 97
programing paradigm, 5
property-driven development, 36
Pythagorean triples, 123

railway, 75
random testing, 45, 54, 64
randomised level diagonalisation, 36, 54, 66
readability, 17
record syntax, 3, 41, 114
recursion, 2, 4, 8, 24, 39, 82, 133
refactoring, 35
rule coverage, 92

search, 26, 28, 61, 157
  heuristic, 66
search space, 28, 42, 60, 67
search strategy, 30, 104, 111
set functions, 77
set monad, 139
sharing, 11, 128, 129
shortest path, 73
side effect, 6, 16, 88
small scope hypothesis, 44
square root, 10
state monad, 141, 147
state monad transformer, 148
Strassen's algorithm, 72
structural induction, 14
success continuation, 119
syntax, 1

test-case generation, 35
test-driven development, 36
testing, 35
ticket price, 75
total function, 2
tree monad, 18
type annotation, 112, 114
type class, 12, 39, 58, 63
  constructor class, 13
  instance, 12, 39, 59, 63
type constructors, 16
type declaration, 2
type inference, 8
type parameter, *see* type variable
type variable, 2, 7
type-class constraint, 40
type-class deriving, 40

Lebenslauf

Persönliche Daten

Sebastian Fischer

geboren am 16. März 1980 in Gießen,
ledig

Beruf

seit Sep 2005	wissenschaftlicher Angestellter am Institut für Informatik der CAU Kiel

Studium

16. Aug 2005	Diplom in Informatik
9. Okt 2002	Vordiplom in Informatik
Okt 00 - Aug 05	Studium der Informatik an der CAU Kiel

Zivildienst

Aug 99 - Jun 00	Malteser Hilfsdienst Kiel

Schulzeit

3. Juli 1999	Abitur
1990 - 1999	Ernst-Barlach-Gymnasium Kiel
1986 - 1990	Reventlouschule Kiel

Kiel, 2010