

# Contents

<b>1</b>	<b>Declarative Programming</b>	<b>1</b>
1.1	Functional programming . . . . .	2
1.1.1	Type polymorphism and higher-order functions . . . . .	3
1.1.2	Lazy evaluation . . . . .	5
1.1.3	Class-based overloading . . . . .	7
1.2	Functional logic programming . . . . .	15
1.2.1	Unbound variables . . . . .	16
1.2.2	Non-determinism . . . . .	17
1.2.3	Lazy non-determinism . . . . .	19
1.2.4	Call-time choice . . . . .	21
1.2.5	Search . . . . .	24
1.2.6	Constraints . . . . .	26
1.3	Chapter notes . . . . .	29
<b>A</b>	<b>Source Code</b>	<b>32</b>
A.1	ASCII versions of mathematical symbols . . . . .	32
A.2	Definitions of used library functions . . . . .	32
<b>B</b>	<b>Proofs</b>	<b>34</b>
B.1	Functor laws for $(a \rightarrow)$ instance . . . . .	34
B.2	Monad laws for <i>Tree</i> instance . . . . .	35



# 1 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, e.g., 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.<sup>1</sup>

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 1.1 with important concepts found in *functional* programming languages, viz., polymorphic typing of higher-order functions, demand-driven evaluation, and type-based overloading. Section 1.2 describes essential features of *logic* programming, viz., non-determinism, unknown values and built-in search

---

<sup>1</sup>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.

### 1.1 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, e.g., demand-driven evaluation. We discuss how demand-driven, so called lazy evaluation can increase the potential for abstraction in Subsection 1.1.2.

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 Subsection 1.1.1.

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 Subsection 1.1.3.

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.

### 1.1.1 Type polymorphism and higher-order functions

Imagine a function *size* that computes the size 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:

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

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*, e.g., `[Int]`, `String`, `[[Bool]]` to name a few.

Type polymorphism 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. Probably the simplest example of a higher-order function is the infix operator `$` for function application:

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

At first sight, this operator seems dispensable, because we can always write `f x` instead of `f $ x`. However, 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<sup>2</sup>:

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

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 the form  $\lambda x \rightarrow e$  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

<sup>2</sup>ASCII representaitons of mathematical symbols are given in Appendix A.1.

function. The body may itself be a function and the notation  $\lambda x y z \rightarrow e$  is short hand for  $\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$ . 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*.<sup>3</sup>

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:

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

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 and the element type of the given list and
- the result type of the given function 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 without looking at its implementation. If we do not provide the type signature, then *type inference* deduces it automatically from the implementation.

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

---

<sup>3</sup>The term *currying* is named after the american mathematician and logician *Haskell B. Curry*.

```

dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x : xs) = if p x then dropWhile p xs else x : xs

```

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 *isSpace :: Char → 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) ∘ length)* drops short lists from a given list of lists, e.g., 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.

### 1.1.2 Lazy evaluation

With lazy evaluation 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:

```
length [⊥, fibonacci 100]
```

Neither the diverging computation  $\perp$  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 different functions are composed sequentially:

```

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

```

This program reads 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]] → [a]` concatenates a given list of lists into a single list<sup>4</sup>. 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, i.e., potentially infinite data. For example, we can compute an approximation of the square root of a number  $x$  as follows:

```
sqrt :: Float → Float
sqrt x = head ∘ dropWhile inaccurate ∘ iterate next $ x
  where inaccurate y = abs (x - y * y) > 0.00001
        next y      = (y + x / y) / 2
```

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, e.g., 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.

---

<sup>4</sup>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*:

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

Conceptually, the call *iterate f x* yields the infinite list

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

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

### 1.1.3 Class-based overloading

Using type polymorphism as described in Subsection 1.1.1 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

$$(\equiv) :: a \rightarrow a \rightarrow \textit{Bool}$$

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

## 1 Declarative Programming

Class-based overloading provides a mechanism to give functions like  $\equiv$  a reasonable type. We can define a *type class* that represents all types that support an equality predicate as follows:

```
class Eq a where  
  ( $\equiv$ ) :: a → a → Bool
```

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  $\equiv$ . After the above declaration, the function  $\equiv$  has the following type:

```
( $\equiv$ ) :: Eq a ⇒ a → a → Bool
```

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

```
( $\neq$ ) :: Eq a ⇒ a → a → Bool  
x  $\neq$  y = ¬ (x  $\equiv$  y)  
elem :: Eq a ⇒ a → [a] → Bool  
x ∈ [] = False  
x ∈ (y : ys) = x  $\equiv$  y ∨ x ∈ ys
```

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

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

```
instance Eq Bool where  
  False  $\equiv$  False = True  
  True  $\equiv$  True = True  
  _  $\equiv$  _ = False
```

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.

```
instance Eq a ⇒ Eq [a] where  
  []  $\equiv$  [] = True  
  (x : xs)  $\equiv$  (y : ys) = x  $\equiv$  y ∧ xs  $\equiv$  ys  
  _  $\equiv$  _ = False
```

Note the class constraint  $Eq\ a$  in the instance declaration for  $Eq\ [a]$ . The first occurrence of  $\equiv$  in the second rule of the definition of  $\equiv$  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  $\equiv$  should be an equivalence relation—reflexive, symmetric and transitive—to aid reasoning about programs that use  $\equiv$ . More specifically, the following properties are usually associated with an equality predicate.

$$\begin{aligned}x &\equiv x \\x \equiv y &\Rightarrow y \equiv x \\x \equiv y \wedge y \equiv z &\Rightarrow x \equiv z\end{aligned}$$

Defining an  $Eq$  instance where  $\equiv$  is no equivalence relation can result in highly unintuitive program behaviour. For example, the *elem* function defined above relies on reflexivity of  $\equiv$ . Using *elem* with a non-reflexive  $Eq$  instance is very likely to be confusing. The inclined reader may check that the definition of  $\equiv$  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*. 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  $map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$  defined in Subsection 1.1.1 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.

## 1 Declarative Programming

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

```
class Functor f where  
  fmap :: (a → b) → f a → f b
```

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

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

Let us check whether the following *Functor* instance for lists satisfies them.

```
instance Functor [] where  
  fmap = map
```

We can prove the first law by induction over the list structure. The base case considers the empty list:

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

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

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

We conclude  $\text{map } id \equiv id$ , 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<sup>5</sup>.

```
data Tree a = Empty | Leaf a | 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*:

```
instance Functor Tree where
  fmap _ Empty    = Empty
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Fork l r) = Fork (fmap f l) (fmap f r)
```

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 ∘ 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*:

```
instance Functor (a  $\rightarrow$ ) where
  fmap = (∘)
```

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

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

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 Subsection 1.1.1. It 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

---

<sup>5</sup>Binary leaf trees are binary trees that store values in their leaves.

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 that can share identical syntax.

### Monads

Monads are a very 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.

```
class Monad m where  
  return :: a → m a  
  (≫=) :: m a → (a → m b) → m b
```

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 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, i.e., 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.

```
getChar :: IO Char  
putChar :: Char → IO ()
```

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:

```
copyChar :: IO ()  
copyChar = getChar ≫= \c → putChar c
```

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.

```
copyChar :: IO ()
copyChar = do c ← getChar
             putChar c
```

In general, **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.

```
instance Monad [] where
  return x = [x]
  l >>= f = concat (map f l)
```

The *return* function for lists yields a singleton list and the  $\gg=$  function maps the given function on 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.

```
pair :: Monad m => m a -> m b -> m (a, b)
pair xs ys = do x ← xs
                y ← ys
                return (x, y)
```

For example, the call `pair [0,1] [True,False]` yields a list of four pairs, viz., `[(0,True),(0,False),(1,True),(1,False)]`. We can write the function *pair* without using the monad operations<sup>6</sup> 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)
```

---

<sup>6</sup> $\lambda xs ys \rightarrow \text{concat} (\text{map} (\lambda x \rightarrow \text{concat} (\text{map} (\lambda y \rightarrow [(x,y)]) ys)) xs)$

## 1 Declarative Programming

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.

$$\begin{aligned} \text{mergeTrees} &:: \text{Tree } (\text{Tree } a) \rightarrow \text{Tree } a \\ \text{mergeTrees } \text{Empty} &= \text{Empty} \\ \text{mergeTrees } (\text{Leaf } t) &= t \\ \text{mergeTrees } (\text{Fork } l r) &= \text{Fork } (\text{mergeTrees } l) (\text{mergeTrees } r) \end{aligned}$$

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.

$$\begin{aligned} &\text{Fork } (\text{Fork } (\text{Leaf } (0, \text{True})) \\ &\quad (\text{Leaf } (0, \text{False}))) \\ &\quad (\text{Fork } (\text{Leaf } (1, \text{True})) \\ &\quad (\text{Leaf } (1, \text{False}))) \end{aligned}$$

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.

$$\begin{aligned} \text{return } x \gg= f &\equiv f x \\ m \gg= \text{return} &\equiv m \\ (m \gg= f) \gg= g &\equiv m \gg= (\lambda x \rightarrow f x \gg= g) \end{aligned}$$

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.



## Summary

In this section we have seen different abstraction mechanisms of functional programming languages that help programmers to write more modular and reusable code. Type polymorphism (Section 1.1.1) 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, e.g., the *map* function on lists. Lazy evaluation (Subsection 1.1.2) 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 (Subsection 1.1.3) 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, viz., 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.

## 1.2 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 two paradigms has shown that their conceptual divide is not as big as one might expect. 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 w.r.t. the pure functional language Haskell are

- unbound variables,
- implicit non-determinism, and

- built-in search.

In the remainder of this section we discuss unbound variables (Section 1.2.1) and non-determinism (Section 1.2.2) and show how they interact with features of functional programming discussed in Section 1.1. We will give a special account to lazy evaluation which allows to relate the concepts of unbound variables and non-determinism in an interesting way (Section 1.2.3) and which forms an intricate combination with implicit non-determinism (Section 1.2.4). Built-in search (Section 1.2.5) allows to enumerate different results of non-deterministic computations and we discuss how programmers can influence the search order by implementing search strategies in Curry.

### 1.2.1 Unbound variables

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

Narrowing w.r.t. patterns is not the only way how unbound 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 1.2.6 but Curry provides a specific kind of constraints that is worth mentioning here: term-equality constraints. The built-in<sup>7</sup> function  $\doteq :: a \rightarrow a \rightarrow Success$  constrains two data terms, which are allowed to contain unbound variables, to be equal. 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 then the corresponding call *fails*, i.e., does not yield a result. We can use constraints—i.e., values of type *Success*—in guards of functions to specify conditions on unbound variables.

---

<sup>7</sup>The fact that  $\doteq$  needs to be built into most Curry systems is due to the lack of type classes in Curry. This is also the reason for the too general type which does not prevent programmers to constrain functions to be equal on the type level. 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  $\doteq$ .

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.

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

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. Unbound Curry 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 unbound 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 unbound 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*.

### 1.2.2 Non-determinism

The built-in search for instantiations of unbound variables can lead to different possible instantiations and, hence, non-deterministic results of computations. Consider, for example, the following definition of *insert*:

$$\begin{aligned} \textit{insert} &:: a \rightarrow [a] \rightarrow [a] \\ \textit{insert} \ x \ l \mid xs \ ++ \ ys &\doteq l \\ &= xs \ ++ \ x : ys \\ \textbf{where } xs, ys &\textbf{ free} \end{aligned}$$

If the argument *l* of *insert* is non-empty then there are different possible bindings for *xs* and *ys* such that *xs ++ ys ≐ l*. Consequently, the result of *insert* may contain *x* at different positions, i.e., 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 non-deterministic results. To avoid the contradictory term *non-deterministic function* we call *insert* (and other defined operations that may have more than one result) *non-deterministic operation*.

## 1 Declarative Programming

Traditionally, Curry systems use backtracking to enumerate all possible results of non-deterministic operations. For example, if we issue the call `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 non-determinism in Curry programs. As the run-time system needs to handle non-determinism anyway, Curry also provides a direct way to define non-deterministic 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 non-deterministically. As a consequence, overlapping rules lead to possibly non-deterministic results.

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

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

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.

The advantage of such implicit non-determinism (as opposed to explicitly using, e.g., lists to represent multiple results) is that the source code does not contain additional combinators to handle non-determinism which eases the composition of more complex non-deterministic operations from simpler ones. For example, we can compute permutations of a given list non-deterministically by recursively inserting all its elements in an empty list.

$$\begin{aligned} \text{permute} &:: [a] \rightarrow [a] \\ \text{permute } [] &= [] \\ \text{permute } (x : xs) &= \text{insert } x \ (\text{permute } xs) \end{aligned}$$

With an explicit representation of non-determinism, we would need to use a separate data structure that models non-deterministic results or use monadic syntax to hide the actual structure used (cf. the definition of the *pair* function in Subsection 1.1.3). Implicit non-determinism helps to focus on algorithmic details because no book-keeping syntax interferes with a non-deterministic algorithm.

### 1.2.3 Lazy non-determinism

In the previous subsection we have seen that the instantiation of unbound variables can lead to non-deterministic computations. We can stress this observation and define an operation  $?$  for non-deterministic choice based on unbound variables:

$$\begin{aligned} (?) &:: a \rightarrow a \rightarrow a \\ x ? y &= \text{ifThenElse } b \ x \ y \\ &\mathbf{\text{where } b \text{ free}} \\ \text{ifThenElse} &:: \text{Bool} \rightarrow a \rightarrow a \rightarrow a \\ \text{ifThenElse True } x \ _ &= x \\ \text{ifThenElse False } \_ \ x &= x \end{aligned}$$

It is well-known for a long time that every Curry program with overlapping rules can be translated into an equivalent Curry program without overlapping rules by using  $?$  for non-deterministic choice. Therefore, one could drop support for direct non-determinism via overlapping rules without restricting the class of programs that can be written.

A more recent discovery is that one can do the opposite too: overlapping rules suffice to model narrowing of unbound variables. This observation essentially relies on laziness and the remainder of this subsection explains the details of combining laziness and non-determinism to model narrowing.

Suppose our language would support non-determinism via overlapping rules but no declarations of unbound variables. We could redefined the operation  $?$  using overlapping rules instead of an unbound variable.

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

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

## 1 Declarative Programming

Narrowing an unbound 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 non-deterministic operation that can be evaluated to every possible binding of the variable. If such an operation—we call it non-deterministic generator—is matched with a pattern then a matching binding will be chosen non-deterministically. An unbound 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.

An unbound variable of type *Bool* can be represented by the non-deterministic generator *bool*.

```
bool :: Bool  
bool = True ? False
```

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* non-deterministically and yields either *False* or *True*. Similarly, the call  $\neg b$  **where** *b* = *bool* evaluates *b* to *True* or *False* non-deterministically 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 unbound variables of type [*Bool*].

```
blist :: [Bool]  
blist = [] ? (bool : blist)
```

The *blist* generator can evaluate to every possible list of Booleans non-deterministically. 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 an unbound variable of the same type is only narrowed as much as demanded.

If we apply the *head* function to the *blist* generator then we obtain two non-deterministic results, viz., *True* or *False*, and the tail of *blist* is not evaluated – just like the tail of an unbound variable of type [*Bool*] is not bound by the *head* function. Besides pointing out the similarity of lazy evaluation and narrowing of unbound variables, this example also demonstrates a difference: unbound variables can be results of computations in Curry. In fact, the result of applying the *head* function to an unbound variable of type [*Bool*] is an unbound 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 unbound variables can be shown to the user without instantiating them with all possible bindings.

Another difference of unbound variables to non-deterministic 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**  $x = \text{bool}$  is solved non-deterministically with two successful derivations. The call  $x \doteq x$  **where**  $x = \text{blist}$  even describes infinitely many non-deterministic successful derivations.

### 1.2.4 Call-time choice

When comparing unbound variables and non-deterministic generators we have made an implicit assumption on the meaning of variable bindings. To illustrate this assumption, consider the call  $\neg x \equiv x$  **where**  $x$  **free**. What are the possible results of this call? From a mathematical point of view,  $\neg x \equiv x$  should clearly be *False* regardless of the instantiation of  $x$ . And indeed, the intuition behind unbound variables is that they denote unknown values as pointed out in Section 1.2.1. There is no possible instantiation of  $x$  to a value such that the call  $\neg x \equiv x$  yields *True*.

For the modeling of narrowing via generators to be correct, it is required that  $\neg x \equiv x$  **where**  $x = \text{bool}$  also cannot yield *True*. But now,  $x$  is not an unbound variable but bound to an expression that can evaluate to *True* or *False* non-deterministically. While  $\text{bool} \equiv \text{bool}$  can evaluate to *True* or *False* non-deterministically 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 and corresponds to an eager evaluation of variable bindings. If we first evaluate the binding of  $x$  to *True* or *False* non-deterministically and then evaluate either  $\neg \text{True} \equiv \text{True}$  or  $\neg \text{False} \equiv \text{False}$  we obtain the result *False* in every non-deterministic 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 1.1.2). If a duplicated variable is bound to an expression, the expression is evaluated at most once and also corresponding non-deterministic choices are performed at most once.

Due to call-time choice, not only unbound variables denote values but every variable—even if duplicated and bound to a non-deterministic expression—denotes one deterministic value in each non-deterministic 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 non-deterministic 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 losing 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 non-deterministically 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

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

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 1.2.2 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. We use the function  $zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$  to pair every queen—i.e., the row in which it should be placed—with its corresponding column.

```
queens :: Int → [Int]
queens n | safe (zip [1..n] qs) = qs
  where qs = permute [1..n]
```



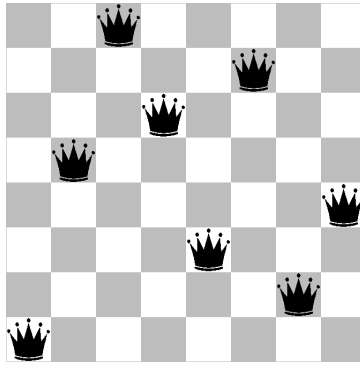


Figure 1.1: Safe placement of 8 queens on an  $8 \times 8$  chessboard

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.

$$\begin{aligned} \text{safe} &:: [(Int, Int)] \rightarrow Bool \\ \text{safe } qs &= \text{and } [j - i \neq \text{abs } (q_j - q_i) \mid (i, q_i) \leftarrow qs, (j, q_j) \leftarrow qs, i < j] \end{aligned}$$

The function  $\text{and} :: [Bool] \rightarrow Bool$  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 1.1.3) and simplify the construction of complex lists.

If we execute *queens 8* in a Curry interpreter it prints a solution—depicted graphically in Figure 1.1—almost instantly:

```
> queens 8
[8,4,1,3,6,2,7,5]
More solutions? [Y(es)/n(o)/a(11)] no
```

The call *queens 10* also has 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 1.2.6) 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, i.e., 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 represent invalid placements. For example, the call *safe*  $((1,1) : (2,2) : \perp)$  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 1.2.2 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 then 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* then we can observe that permutations are not computed completely: we only obtain three results, viz., 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.

### 1.2.5 Search

Implicit non-determinism is convenient to implement non-deterministic algorithms that do not care about the non-deterministic choices they make, i.e., 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 placement of 3 queens on a  $3 \times 3$  chessboard?

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

Therefore Curry supports an operation  $getAllValues :: a \rightarrow IO [a]$  that converts a possibly non-deterministic computation into a list of all its results. The list is returned in the *IO* monad (see Section 1.1.3) because the order of the elements in the computed list is unspecified and may depend on external information, e.g., 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  $3 \times 3$  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 1.2.3).

To overcome this limitation, some Curry implementations provide another search function  $getSearchTree :: a \rightarrow IO (SearchTree a)$  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<sup>8</sup>.

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

The constructor *Failure* represents a computation without results, *Value a* denotes a single result, and *Choice ts* represents a non-deterministic choice between the solutions in the sub trees *ts*. For example, the call  $getSearchTree\ bool$  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

---

<sup>8</sup>The definition of the *SearchTree* type varies among different Curry implementations.

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<sup>9</sup> 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]]`.

```
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 into a deterministic data structure and process them as a whole. With access to a lazy tree representation of the search space, programmers can define their own search strategies and steer the computation towards the parts of the search space that are explored by their traversal function. Especially, they can implement fair search strategies like breadth-first search.

### 1.2.6 Constraints

A key feature of logic programming is the ability to compute with unbound variables that are bound according to conditions. In Subsection 1.2.1 we have seen two ways to bind variables, viz., *narrowing* w.r.t. patterns and term-equality *constraints*. Narrowing can refine the possible values of an unbound variable incrementally while an equality constraint determines a unique binding.

---

<sup>9</sup>which is terminated by itself to enqueue new elements at the end

For specific types, the idea of restricting the possible values of a variable using constraints can be generalised to arbitrary domain specific predicates. In order to solve such domain specific constraints, sophisticated solver implementations can be supplied transparently, i.e., 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 formula where unknown Boolean variables are just represented as unbound variables of this type. Or complex algorithms to solve non-linear equations over real numbers could be integrated to support unbound variables in arithmetic computations.

Finite-domain constraints express equations and in-equations 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 unbound 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 1.2.4 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 unbound 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.

```

queens :: Int → [Int]
queens n | domain qs 1 n
         & all_different qs
         & safe (zip [1..n] qs)
         & labeling [FirstFailConstrained] qs
         = qs
where qs = [unknown | _ ← [1..n]]

```

This definition uses the predefined operation *unknown*—which yields an unbound variable—to build a list of unbound 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.

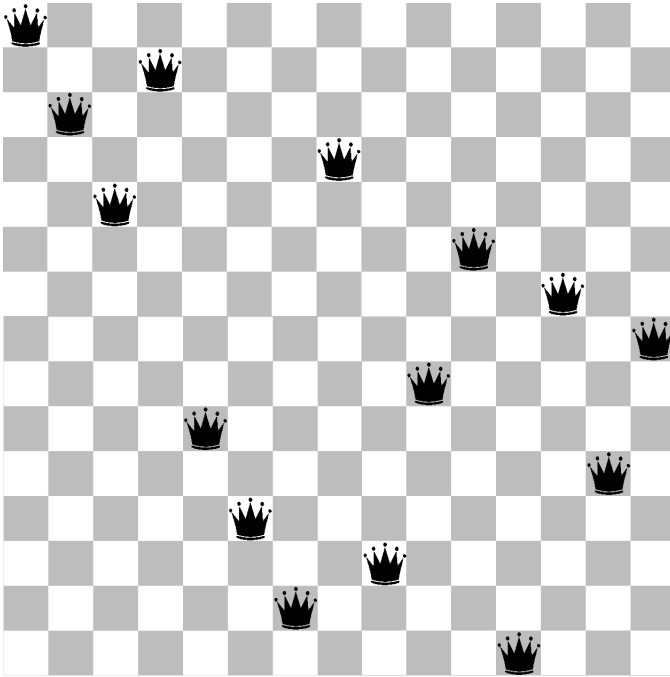


Figure 1.2: Safe placement of 15 queens on an  $15 \times 15$  chessboard

Finally, the *labeling* constraint instantiates the variables with their possible values non-deterministically. The list given as first argument to *labeling* specifies a strategy to use during instantiation, i.e., 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 1.2.4.

$$\begin{aligned}
 \text{safe} &:: [(Int, Int)] \rightarrow \text{Success} \\
 \text{safe } qs &= \text{andC} [ (j - i) \neq (q_j - \# q_i) \ \& \ (j - i) \neq (q_i - \# q_j) \\
 &\quad | (i, q_i) \leftarrow qs, (j, q_j) \leftarrow qs, i < j ]
 \end{aligned}$$

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.

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 1.2.

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 unbound variables.

## Summary

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

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 unbound variables and non-determinism 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 non-determinism that lead to the standard *call-time choice* semantics of functional logic programs.

## 1.3 Chapter notes

Hughes has argued previously why functional programming matters for developing modular and reusable software (1989). The benefits of combining functional and logic programming for program design have been investigated by Antoy and Hanus (2002). They have also shown the correspondence of

## *1 Declarative Programming*

narrowing and lazy non-determinism emphasised in Section 1.2.3 (Antoy and Hanus 2006).



# Bibliography

- 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.
- . 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.
- Hughes, John. 1989. Why functional programming matters. *Computer Journal* 32(2):98–107.

# 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.

$\perp$	undefined	$f \circ g$	$f . g$
$\lambda x \rightarrow e$	<code>\x -&gt; e</code>	<code>do x ← a; b</code>	<code>do x &lt;- a; b</code>
$x \equiv y$	<code>x == y</code>	$x \not\equiv y$	<code>x /= y</code>
$x \leq y$	<code>x &lt;= y</code>	$x \geq y$	<code>x &gt;= y</code>
$x \doteq y$	<code>x := y</code>	$xs \uparrow\uparrow ys$	<code>xs ++ ys</code>
$\neg x$	<code>not x</code>	$x \wedge y$	<code>x &amp;&amp; y</code>
$x \vee y$	<code>x    y</code>	$x \in xs$	<code>x 'elem' xs</code>
$a \ggg f$	<code>a &gt;&gt;= f</code>	$()$	<code>()</code>
$x \not\# y$	<code>x /=# y</code>	$x \# y$	<code>x -# y</code>

## 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  $\perp$  denotes a non-terminating computation.

$$\begin{aligned} \perp &:: a \\ \perp &= \perp \end{aligned}$$

The *concat* function merges a list of lists into a single list.

$$\begin{aligned} \text{concat} &:: [[a]] \rightarrow [a] \\ \text{concat} [] &= [] \\ \text{concat} (l : ls) &= l \uparrow\uparrow \text{concat } ls \end{aligned}$$

The absolute value of an integer can be computed using *abs*.

$$\begin{aligned} \text{abs} &:: \text{Int} \rightarrow \text{Int} \\ \text{abs } n &= \text{if } n \geq 0 \text{ then } n \text{ else } (-n) \end{aligned}$$

There are Boolean functions for negation, conjunction, and disjunction.

$$\begin{aligned}
\neg &:: \text{Bool} \rightarrow \text{Bool} \\
\neg \text{ True} &= \text{False} \\
\neg \text{ False} &= \text{True} \\
(\wedge), (\vee) &:: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \\
\text{True} \wedge x &= x \\
\text{False} \wedge \_ &= \text{False} \\
\text{True} \vee \_ &= \text{True} \\
\text{False} \vee x &= x
\end{aligned}$$

The *zip* function pairs the elements of given lists.

$$\begin{aligned}
\text{zip} &:: [a] \rightarrow [b] \rightarrow [(a, b)] \\
\text{zip} [] \_ &= [] \\
\text{zip} (\_ : \_) [] &= [] \\
\text{zip} (x : xs) (y : ys) &= (x, y) : \text{zip } xs \ ys
\end{aligned}$$

The function *take* selects a prefix of given length from a given list.

$$\begin{aligned}
\text{take} &:: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take} \_ [] &= [] \\
\text{take } n (x : xs) &= \mathbf{if } n \leq 0 \mathbf{ then } [] \mathbf{ else } x : \text{take } (n - 1) \ xs
\end{aligned}$$

The operation *unknown* returns an unbound variable.

$$\begin{aligned}
\text{unknown} &:: a \\
\text{unknown} &= x \mathbf{ where } x \mathbf{ free}
\end{aligned}$$

Three predicates *and* and *andC* implement conjunction on lists of Boolean and constraints respectively.

$$\begin{aligned}
\text{and} &:: [\text{Bool}] \rightarrow \text{Bool} \\
\text{and} [] &= \text{True} \\
\text{and} (b : bs) &= b \wedge \text{and } bs \\
\text{andC} &:: [\text{Success}] \rightarrow \text{Success} \\
\text{andC} [] &= \text{success} \\
\text{andC} (c : cs) &= c \ \& \ \text{andC } cs
\end{aligned}$$

# B Proofs

## B.1 Functor laws for $(a \rightarrow)$ instance

This is a proof of the functor laws

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

for the *Functor* instance

**instance** *Functor*  $(a \rightarrow)$  **where**  
  *fmap* =  $(\circ)$

The laws are a consequence of the fact that functions form a monoid under composition with the identity element *id*.

$$\begin{aligned} &fmap\ id\ h \\ \equiv &\{ \text{definition of } fmap \} \\ &id \circ h \\ \equiv &\{ \text{definition of } (\circ) \} \\ &\lambda x \rightarrow id\ (h\ x) \\ \equiv &\{ \text{definition of } id \} \\ &\lambda x \rightarrow h\ x \\ \equiv &\{ \text{expansion} \} \\ &h \\ \equiv &\{ \text{definition of } id \} \\ &id\ h \end{aligned}$$

This proof makes use of the identity  $(\lambda x \rightarrow f\ x) \equiv f$  for every function *f*. The second law is a bit more involved as it relies on associativity for function composition.

$$\begin{aligned} &fmap\ (f \circ g)\ h \\ \equiv &\{ \text{definition of } fmap \} \\ &(f \circ g) \circ h \\ \equiv &\{ \text{associativity of } (\circ) \} \\ &f \circ (g \circ h) \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{definition of } fmap \text{ (twice)} \} \\
&\quad fmap\ f\ (fmap\ g\ h) \\
&\equiv \{ \text{reduction} \} \\
&\quad (\lambda x \rightarrow fmap\ f\ (fmap\ g\ x))\ h \\
&\equiv \{ \text{definition of } (\circ) \} \\
&\quad (fmap\ f\ \circ\ fmap\ g)\ h
\end{aligned}$$

Now it is only left to verify that function composition is indeed associative:

$$\begin{aligned}
&(f \circ g) \circ h \\
&\equiv \{ \text{definition of } (\circ) \text{ (twice)} \} \\
&\quad \lambda x \rightarrow (\lambda y \rightarrow f\ (g\ y))\ (h\ x) \\
&\equiv \{ \text{reduction} \} \\
&\quad \lambda x \rightarrow f\ (g\ (h\ x)) \\
&\equiv \{ \text{reduction} \} \\
&\quad \lambda x \rightarrow f\ ((\lambda y \rightarrow g\ (h\ y))\ x) \\
&\equiv \{ \text{definition of } (\circ) \text{ (twice)} \} \\
&\quad f \circ (g \circ h)
\end{aligned}$$

## B.2 Monad laws for *Tree* instance

This is a proof of the monad laws

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

for the *Monad* instance

$$\begin{aligned}
&\mathbf{instance\ Monad\ Tree\ where} \\
&\quad return = Leaf \\
&\quad t \gg\! =\ f = mergeTrees\ (fmap\ f\ t) \\
&\quad mergeTrees :: Tree\ (Tree\ a) \rightarrow Tree\ a \\
&\quad mergeTrees\ Empty = Empty \\
&\quad mergeTrees\ (Leaf\ t) = t \\
&\quad mergeTrees\ (Fork\ l\ r) = Fork\ (mergeTrees\ l)\ (mergeTrees\ r)
\end{aligned}$$

for the data type

$$\mathbf{data\ Tree\ a = Empty\ | Leaf\ a\ | Fork\ (Tree\ a)\ (Tree\ a)}$$

The left-identity law follows from the definitions of the functions *return*,  $\gg=$ , *fmap*, and *mergeTrees*.

$$\begin{aligned}
 & \text{return } x \gg= f \\
 \equiv & \quad \{ \text{definitions of } \text{return} \text{ and } \gg= \} \\
 & \text{mergeTrees } (\text{fmap } f \text{ (Leaf } x)) \\
 \equiv & \quad \{ \text{definition of } \text{fmap} \} \\
 & \text{mergeTrees } (\text{Leaf } (f \ x)) \\
 \equiv & \quad \{ \text{definition of } \text{mergeTrees} \} \\
 & f \ x
 \end{aligned}$$

We prove the right-identity law by induction over the structure of *m*. The *Empty* case follows from the observation that  $\text{Empty} \gg= f \equiv \text{Empty}$  for every function *f*, i.e., also for  $f \equiv \text{return}$ .

$$\begin{aligned}
 & \text{Empty} \gg= f \\
 \equiv & \quad \{ \text{definition of } \gg= \} \\
 & \text{mergeTrees } (\text{fmap } f \ \text{Empty}) \\
 \equiv & \quad \{ \text{definition of } \text{fmap} \} \\
 & \text{mergeTrees } \text{Empty} \\
 \equiv & \quad \{ \text{definition of } \text{mergeTrees} \} \\
 & \text{Empty}
 \end{aligned}$$

The *Leaf* case follows from the left-identity law because  $\text{return} \equiv \text{Leaf}$ .

$$\begin{aligned}
 & \text{Leaf } x \gg= \text{return} \\
 \equiv & \quad \{ \text{definition of } \text{return} \} \\
 & \text{return } x \gg= \text{return} \\
 \equiv & \quad \{ \text{first monad law} \} \\
 & \text{return } x \\
 \equiv & \quad \{ \text{definition of } \text{return} \} \\
 & \text{Leaf } x
 \end{aligned}$$

The *Fork* case makes use of the induction hypothesis and the observation that  $\text{Fork } l \ r \gg= f \equiv \text{Fork } (l \gg= f) \ (r \gg= f)$

$$\begin{aligned}
 & \text{Fork } l \ r \gg= f \\
 \equiv & \quad \{ \text{definition of } \gg= \} \\
 & \text{mergeTrees } (\text{fmap } f \ (\text{Fork } l \ r)) \\
 \equiv & \quad \{ \text{definition of } \text{fmap} \} \\
 & \text{mergeTrees } (\text{Fork } (\text{fmap } f \ l) \ (\text{fmap } f \ r)) \\
 \equiv & \quad \{ \text{definition of } \text{mergeTrees} \}
 \end{aligned}$$

$$\begin{aligned}
& \text{Fork } (\text{mergeTrees } (\text{fmap } f \ l)) \ (\text{mergeTrees } (\text{fmap } f \ r)) \\
\equiv & \quad \{ \text{definition of } \gg= \text{ (twice)} \} \\
& \text{Fork } (l \gg= f) \ (r \gg= f)
\end{aligned}$$

Now we can apply the induction hypothesis.

$$\begin{aligned}
& \text{Fork } l \ r \ \gg= \ \text{return} \\
\equiv & \quad \{ \text{previous derivation} \} \\
& \text{Fork } (l \gg= \text{return}) \ (r \gg= \text{return}) \\
\equiv & \quad \{ \text{induction hypothesis (twice)} \} \\
& \text{Fork } l \ r
\end{aligned}$$

Finally we prove associativity of  $\gg=$  by structural induction. The *Empty* case follows from the above observation that  $\text{Empty} \gg= f \equiv \text{Empty}$  for every function  $f$ .

$$\begin{aligned}
& (\text{Empty} \gg= f) \gg= g \\
\equiv & \quad \{ \text{above observation for } \text{Empty} \text{ (twice)} \} \\
& \text{Empty} \\
\equiv & \quad \{ \text{above observation for } \text{Empty} \} \\
& \text{Empty} \gg= (\lambda x \rightarrow f \ x \ \gg= \ g)
\end{aligned}$$

The *Leaf* case follows again from the first monad law.

$$\begin{aligned}
& (\text{Leaf } y \ \gg= \ f) \ \gg= \ g \\
\equiv & \quad \{ \text{definition of } \text{return} \} \\
& (\text{return } y \ \gg= \ f) \ \gg= \ g \\
\equiv & \quad \{ \text{first monad law} \} \\
& f \ y \ \gg= \ g \\
\equiv & \quad \{ \text{first monad law} \} \\
& \text{return } y \ \gg= \ (\lambda x \rightarrow f \ x \ \gg= \ g) \\
\equiv & \quad \{ \text{definition of } \text{return} \} \\
& \text{Leaf } y \ \gg= \ (\lambda x \rightarrow f \ x \ \gg= \ g)
\end{aligned}$$

The *Fork* case uses the identity  $\text{Fork } l \ r \ \gg= \ f \equiv \text{Fork } (l \ \gg= \ f) \ (r \ \gg= \ f)$  that we proved above and the induction hypothesis.

$$\begin{aligned}
& (\text{Fork } l \ r \ \gg= \ f) \ \gg= \ g \\
\equiv & \quad \{ \text{property of } \gg= \} \\
& \text{Fork } (l \ \gg= \ f) \ (r \ \gg= \ f) \ \gg= \ g \\
\equiv & \quad \{ \text{property pf } \gg= \} \\
& \text{Fork } ((l \ \gg= \ f) \ \gg= \ g) \ ((r \ \gg= \ f) \ \gg= \ g) \\
\equiv & \quad \{ \text{induction hypothesis} \}
\end{aligned}$$

## B Proofs

$$\begin{aligned} & \text{Fork } (l \gg= (\lambda x \rightarrow f x \gg= g)) (r \gg= (\lambda x \rightarrow f x \gg= g)) \\ & \equiv \{ \text{property of } \gg= \} \\ & \text{Fork } l r \gg= (\lambda x \rightarrow f x \gg= g) \end{aligned}$$

This finishes the proof of the three monad laws for the *Tree* instance.



# Index

- backtracking, 18
- binary leaf tree, 11
- breadth-first search, 26
  
- call-time choice, 21
- class constraint, 8
- constraint
  - equality, 16
- constraints, 26
  - finite domain, 27
- currying, 4
  
- declarative programming, 1
- do-notation, 13
  
- equational reasoning, 10
- evaluation order, 2
  
- failure, 16
- free variable, *see* unbound variable
- functional logic programming, 15
- functional programming, 2
- functor, 10
  
- generate-and-test, 22
  
- higher-order function, 3
  
- infinite data structure, 6
- input/output, 12
- IO monad, 12
  
- lambda abstraction, 3
- lazy evaluation, 5
- list comprehension, 23
- list monad, 13
  
- logic variable, *see* unbound variable
  
- monad, 12
  - laws, 14
- multi-paradigm language, 15
  
- narrowing, 16
- non-determinism, 17
  - generator, 20
  - laziness, 19
- non-deterministic operation, 17
  
- overloading, 7
  
- partial application, 5
- polymorphism, 3
- programing paradigm, 1
  
- search, 24
- sharing, 7
- side effect, 2
- square root, 6
- structural induction, 10
  
- tree monad, 13
- type class, 8
  - constructor class, 9
  - instance, 8
- type inference, 4
- type variable, 3
  
- unbound variable, 16