Fortgeschrittene funktionale Programmierung LVA 185.A05, VU 2.0, ECTS 3.0 SS 2020 (Stand: 18.05.2020)

Jens Knoop



Technische Universität Wien Information Systems Engineering Compilers and Languages



Lecture 6

Detailed Outline

Chap. 15

Chap. 16

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

Vote

Lecture 6

Part V: Applications

- Chapter 15: Parsing
- Chapter 16: Logic Programming Functionally

Outline in more Detail (1)

Part V: Applications

- ► Chap. 15: Parsing
 - 15.1 Motivation
 - 15.2 Combinator Parsing
 - 15.2.1 Primitive Parsers
 - 15.2.2 Parser Combinators
 - 15.2.3 Universal Combinator Parser Basis
 - 15.2.4 Structure of Combinator Parsers
 - 15.2.5 Writing Combinator Parsers: Examples
 - 15.3 Monadic Parsing
 - 15.3.1 The Parser Monad
 - 15.3.2 Parsers as Monadic Operations
 - 15.3.3 Universal Monadic Parser Basis
 - 15.3.4 Utility Parsers
 - 15.3.5 Structure of a Monadic Parser
 - 15.3.6 Writing Monadic Parsers: Examples
 - 15.4 Summary
 - 15.5 References, Further Reading

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

Outline in more Detail (2)

- ► Chap. 16: Logic Programming Functionally
 - 16.1 Motivation
 - 16.1.1 On the Evolution of Programming Languages
 - 16.1.2 Functional vs. Logic Languages
 - 16.1.3 A Curry Appetizer
 - 16.1.4 Outline
 - 16.2 The Combinator Approach
 - 16.2.1 Introduction
 - 16.2.2 Diagonalization
 - 16.2.3 Diagonalization with Monads
 - 16.2.4 Filtering with Conditions
 - 16.2.5 Indicating Search Progress
 - 16.2.6 Selecting a Search Strategy
 - 16.2.7 Terms, Substitutions, Unification, and Predicates
 - 16.2.8 Combinators for Logic Programs
 - 16.2.9 Writing Logic Programs: Two Examples
 - 16.3 In Closing
 - 16.4 References, Further Reading

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3 15.4

13.5

Note

Chapter 15 Parsing

Parsing: Lexical and Syntactical Analysis

Parsing

- basic task of a compiler.
- umbrella term for the lexical and syntactical analysis of the structure of text, e.g., source code text of programs.
- enjoys a long history, see e.g.
 - William H. Burge. Recursive Programming Techniques. Addison-Wesley, 1975.

as an example of an early text book concerned with parsing.

Last but not least

 an application often used for demonstrating the power and elegance of functional programming.

Lecture 6

Detailed Outline

Chap. 15

15.2	
15.3	
15.4	
15.5	
	1

Functional Approaches for Parsing

...two different but conceptually related approaches are:

- 1. Combinator parsing
 - Graham Hutton. Higher-Order Functions for Parsing. Journal of Functional Programming 2(3):323-343, 1992.
- 2. Monadic parsing
 - Graham Hutton, Erik Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.
 - Graham Hutton, Erik Meijer. Monadic Parsing in Haskell. Journal of Functional Programming 8(4):437-444, 1998.

which are both well-suited for building recursive descent parsers.

Lecture 6

Detailed Outline

Chap. 15

15.2 15.3 15.4 15.5 Chap. 1

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3

5.4

Chap. 16

Note

Chapter 15.1 Motivation

Informally

... the parsing problem is the following:

- 1. Read a sequence of objects/values of a type a.
- 2. Yield an object/value or a sequence of objects/values of a type b.

Illustration:

1. Read a sequence of values of type Char:

 $\langle \text{if n mod} = 0 \text{ then } 2*n \text{ else } 2*n+1 \text{ fi} \rangle$

2. Yield a sequence of pairs of tokens and strings:

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3 15.4 15.5

```
Note
```

Parsing Arithmetic Expressions

 $\ldots a \ parser \ p \ for \ arithmetic \ expressions \ could \ be \ assumed \ to$

- 1. read strings representing well-formed arithmetic expression
- 2. yield the Exp values matching the strings read with:

data Exp = Lit Int | Var Char | Op Ops Exp Exp
data Ops = Add | Sub | Mul | Div | Mod

Example:

p "((2+b)*5)"
 ->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3 15.4 15.5

Chap. 16

Note

 \dots such a parser p for arithmetic expressions were

▶ the reverse of the show function:

show Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
 ->> "((2+b)*5)"

p "((2+b)*5)"

->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)

- similar to the automatically derived read function for Exp values, differing, however, in the kind of arguments they accept
 - p: Strings of the form "((2+b)*5)":

p "((2+b)*5)"

->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)
- read: Strings of the form "Op Mul (Add (Lit ...)":
 read "Op Mul (Add (Lit 2) (Var 'b')) (Lit 5)"
 ->> Op Mul (Op Add (Lit 2) (Var 'b')) (Lit 5)

Lecture 6

Detailed Dutline

Chap. 1. 15.1 15.2 15.3 15.4 15.5 Chap. 1.

Towards the Type of Parser Functions (1)

...considering parsing as

- 1. reading of sequences of objects of some type a
- 2. yielding objects or sequences of objects of some type b

suggests naively the type of parser functions should be:

type Parse_naive a b = [a] -> b

This, however, raises some questions. Assume, bracket and number are parser functions recognizing brackets and numbers, respectively:

Parser Input What shall be the output? bracket "(xyz" ->> '('? If so, what to do w/"xyz"? number "234" ->> 2? Or: 23? Or: 234? bracket "234" ->> No result? Failure?

Lecture 6

Detailed Dutline

Chap. 15

Towards the Type of a Parser Function (2)

...this means, we have to answer:

How shall a parser function behave if

- (i) the input is not completely read?
- (ii) there are multiple results?
- (iii) there is a failure?

The latter two questions suggest the following type refinement:

type Parse_refined a b = [a] -> [b]

which allows for the previous example the following output:

Parser	Input		Output
bracket	"(xyz"	->>	['(']
number	"234"	->>	[2,23,234]
bracket	"234"	->>	[]

Lecture 6

Detailed Dutline

Chap. 15

15.1 15.2 15.3 15.4

Chap. 16

Towards the Type of a Parser Function (3)

...we are left with answering:

(i) What a parser function shall do with the part of the input that is not read?

Answering this question leads finally to the definite definition of the type of parser functions:

...which enables as output lists of pairs of recognized objects and left-over inputs:

Parser Input Output bracket "(xyz" ->> [('(',"xyz")] number "234" ->> [(2,"34"),(23,"4"), (234,"")] bracket "234" ->> []

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3 15.4

Chap. 16

```
Note
```

Informally

- ... if a parser function delivers
 - the empty list, this signals failure of the analysis.
 - a non-empty list, this signals success of the analysis: Every list element represents the result of a successful parse.

In the success case, every list element is a pair, whose

- first component is the identified object (token)
- second component is the remaining input which must still be analyzed.

Note, delivering multiple results by means of lists

- is known as the so-called list of successes technique (Philip Wadler, 1985).
- enables parsers to also analyze ambiguous grammars.

Lecture 6

Detailed Outline

Chap. 15

15.2 15.3 15.4 15.5

Chap. 16

Reference

...the following presentation is based on:

- Simon Thompson. Haskell The Craft of Functional Programming, Addison-Wesley/Pearson, 2nd edition, 1999, Chapter 17.
- Graham Hutton, Erik Meijer. Monadic Parsing in Haskell. Journal of Functional Programming 8(4):437-444, 1998.

Lecture 6

Detailed Outline

Chap. 15

15.1 15.2 15.3 15.4 15.5 Chap. 1

Lecture 6

Detailed Outline

Chap. 15

15.2 15.2.1 15.2.2 15.2.3

Chapter 15.2

Combinator Parsing

15.2.4

.4

5.5

Chap. 16

Objective

...developing a combinator library for parsing composed of

- Four primitive parser functions
 1.&2. Two input-independent ones (none, succeed)
 3.&4. Two input-dependent ones (token, spot)
- Three parser combinators for
 - 1. Alternatives (alt)
 - 2. Sequencing ((>*>))
 - 3. Transforming (build)

...forming a universal parser basis, which allows to construct parser functions at will, i.e., according to what is required by a parsing problem. 15.2

Chapter 15.2.1 Primitive Parsers

Lecture 6

Detailed Outline

Chap. 15

.2

15.2.1

.2.3

5.2.5

5.3

15.5

hap. 16

The two Input-independent Primitive Parsers Recall:

type Parse a b = [a] -> [(b,[a])]

- 1. none, the always failing parser
 none :: Parse a b
 none _ = []
- 2. succeed, the always succeeding parser
 succeed :: b -> Parse a b
 succeed val inp = [(val,inp)]

Note:

- Parser none always fails. It does not accept anything.
- Parser succeed always succeeds without consuming its input or parts of it. In BNF-notation this corresponds to the symbol ε representing the empty word.

Lecture 6

Detailed Outline

Chap. 15

15.2 15.2.1

5.2.3

.3

The two Input-dependent Primitive Parsers

token, the parser recognizing single objects (so-called tokens):

token :: Eq a => a -> Parse a a
token t (x:xs)
 | t == x = [(t,x)]
 | otherwise = []
token t [] = []

spot, the parser recognizing single objects enjoying some property:

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x = [(x,xs)]
  | otherwise = []
spot p [] = []
```

Lecture 6

Detailed Dutline

15.1 15.2 15.2.1 15.2.2

5.4

hap. 16

Example: Using the Primitive Parsers

...for constructing parsers for simple parsing problems:

```
bracket = token '('
dig = spot isDigit
isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')</pre>
```

Note: The parser functions token and bracket could also be defined using spot:

Lecture 6

Detailed Dutline

15.1 15.2 15.2.1 15.2.2

15.2.5 15.3

15.4

Chap. 16

Chapter 15.2.2 Parser Combinators

Lecture 6

Detailed Outline

hap. 15

.2

15.2.2

.2.3

5.2.5

15.3

1.0.1

Chap. 16

Parser Combinators

...to write more complex and powerful parser functions, we need in addition to primitive parsers

- parser-combining functions (or parser combinators)

which are re-usable higher-order polymorphic functions.

Detailed Outline

15.1 15.2 15.2.1 15.2.2

15.2.3

15.2.4

15.3

15.4

.nap. 10

The Parser Combinator for Alternatives

Combining parsers as alternatives:

1. alt, the parser combining parsers as alternatives:

alt :: Parse a b -> Parse a b -> Parse a b alt p1 p2 input = p1 input ++ p2 input

Intuitively: alt combines the results of the parses of p1 and p2. The success of either of them is a success of their combination.

Lecture 6

Detailed Outline

> nap. 18 5.1 5.2

15.2.2

15.2.4 15.2.5

15.3

15.5

Chap. 16

Example: Alternatively Combining Parsers

(bracket 'alt' dig) "234" ->> [] ++ [(2,"34")] ->> [(2,"34")]

...reflecting that numbers might start with a bracket or a digit.

(lit 'alt' var 'alt' opexp) "(234+7)" ->> ...

...reflecting that expressions are either literals, or variables or complex expressions starting with an operator.

Lecture 6

Detailed Outline

15.1 15.2

15.2.2 15.2.3 15.2.4 15.2.5

15.3 15.4

Chap. 16

The Parser Combinator for Sequential Comp.

Combining parsers sequentially:

Note:

The values (y,rem1) run through the results of parser p1 applied to input. Parser p2 is applied to the part rem1 of the input that is unconsumed by p1 in every particular case. The results of the successful parses of p1 and p2, y and z, are returned as a pair.

Detailed Outline

Example: Sequentially Composing Parsers

...evaluating number "24(" yields a list of two parse results [(2, "4("), (24, "(")]. We thus get for the composition of the parsers number and bracket applied to input "24(":

(number >*> bracket) "24("
->> [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],</pre>

->> [((2,z),rem2) | (z,rem2) <- bracket "4("] ++ [((24,z),rem2) | (z,rem2) <- bracket "("]

(z.rem2) <- bracket rem1]

->> [] ++ [((24,z),rem2) | (z,rem2) <- bracket "("]

- ->> [((24,z),rem2) | (z,rem2) <- bracket "("]
- ->> [((24,z),rem2) | (z,rem2) <- [('(',"")]]

->> [((24,'('),"")]

Lecture 6

Detailed Outline

15.1 15.2 15.2.1 15.2.2 15.2.3

15.2.5 15.3

Chap. 16

The Parser Combinator for Transformations

Combining a parser with a map transforming the parse results:

3. build, the parser transforming obtained parse results: build :: Parse a $b \rightarrow (b \rightarrow c) \rightarrow$ Parse a c build p f input = [(f x,rem) | (x,rem) <- p input]</pre>

Intuitively: The map argument **f** of build transforms the items returned by its parser argument: It builds something from it.

15 2 2

Example: Transforming Parse Results

...the parser digList is assumed to return a list of digit lists, whose elements are transformed by digsToNum into the numbers whose values they represent:

Lecture 6

Detailed Outline

15.1 15.2 15.2.1 15.2.2 15.2.3 15.2.4 15.2.5 15.3

Chap. 16

Chapter 15.2.3 Universal Combinator Parser Basis

Lecture 6

Detailed Outline

> ар. 15 .1

.5.2.1

15.2.3

15.2.5

15.3

15.4

hap. 16

Universal Combinator Parser Basis

build any parser we might be in need of.

Lecture 6

Detailed Outline

...together, the four primitive parsers
1.,2.,3.,4.: none, succeed, token, spot
and the three parser combinators
1.,2.,3.: alt, (>*>), build
form a universal combinator parser basis, i.e., they allow us to

The Universal Parser Basis at a Glance (1)	
The priority of the sequencing operator:	
infixr 5 >*>	
The two of more functions	15.2
The type of parser functions:	15.2.1 15.2.2
	15.2.3
type Parse a b = $[a] \rightarrow [(b, [a])]$	15.2.4
	15.3
The insert independent substitute second functions.	15.4 15.5
Two input-independent primitive parser functions:	
1. The always failing parser function:	
1. The always failing parser function.	Note
none :: Parse a b	
none _ = []	
2. The always succeeding parser function:	
succeed :: b -> Parse a b	
<pre>succeed val input = [(val,input)]</pre>	

The Universal Parser Basis at a Glance (2)

Two input-dependent primitive parser functions:

3. The parser for recognizing single objects:

token :: Eq a => a -> Parse a a
token t = spot (==t)

4. The parser for recognizing single objects satisfying some property:

Lecture 6

Detailed Outline

> .5.1 5.2 15.2.1 15.2.2

1523

5.5

Chap. 16

The Universal Parser Basis at a Glance (3) Three parser combinators: 5. Alternatives alt :: Parse a b \rightarrow Parse a b \rightarrow Parse a b alt p1 p2 input = p1 input ++ p2 input 6. Sequencing (>*>) :: Parse a b -> Parse a c -> Parse a (b,c) (>*>) p1 p2 input = [((y,z),rem2) | (y,rem1) <- p1 input, (z,rem2) <- p2 rem1]

7. Transformation

build :: Parse a b -> (b -> c) -> Parse a c build p f input = [(fx,rem) | (x,rem) <- p input]</pre>

1523

Chapter 15.2.4 Structure of Combinator Parsers

Lecture 6

Detailed Outline

.1

.2.1

15.2.4

15.2.5

15.4

15.5

Chap. 16

The Structure of Combinator Parsers

... is usually as follows:

type Pars	se a b = [a] -> [(b,[a])]	
J I		
nono	:: Parse a b	15.1 15.2
none	Faise a D	15.2.1
succeed	:: b -> Parse a b	15.2.2
succeeu		15.2.3
+ - 1		15.2.4 15.2.5
token	:: Eq a => a -> Parse a a	15.3
anot	:: (a -> Bool) -> Parse a a	15.4
spot	$(a \rightarrow bool) \rightarrow raise a a$	15.5
alt	:: Parse a b -> Parse a b -> Parse a b	
alt	: Parse a D -> Parse a D -> Parse a D	Note
(>*>)	:: Parse a b -> Parse a c -> Parse a (b,c)	
build	:: Parse a b -> (b -> c) -> Parse a c	
list	:: Parse a b -> Parse a [b]	
topLevel	:: Parse a b \rightarrow [a] \rightarrow b $$ see Exam. 2,	
-	Chap. 15.2.5	
	Ollap. 10.2.0	

Combinator Parsers

... are well-suited for writing so-called recursive descent parsers.

This is because the parser functions (summarized on the previous slide)

- are structurally similar to grammars in BNF-form.
- provide for every operator of the BNF-grammar a corresponding (higher-order) parser function.

These (higher-order) parser functions allow

- combining simple(r) parsers to (more) complex ones.
- are therefore called combining forms, or, as a short hand, combinators (cf. Graham Hutton. Higher-order Functions for Parsing. Journal of Functional Programming 2(3), 323-343, 1992).

hap. 15 5.1 5.2 15.2.1 15.2.2

15.2.4 15.2.5 15.3

Chapter 15.2.5 Writing Combinator Parsers: Examples

Lecture 6

Detailed Outline

> nap. 15 5.1 5.2

5.2.1 5.2.2

15.2.3

15.2.5

15.3

15.4

Chap. 16

Using the Parser Basis

...for constructing (more) complex parser functions.

A parser

- 1. recognizing a list of objects (example 1).
- 2. transforming a string expression into a value of a suitable algebraic data type for expressions (example 2).

Lecture 6

Detailed Outline

hap. 15

.2 5.2.1

.2.3

15.2.5

.....

15.5

hap. 16

Example 1: Parsing a List of Objects

...let p be a parser recognizing single objects. Then list applied to p is a parser recognizing lists of objects:

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
                                ((p >*> list p) 'build' (uncurry (:)))
```

Intuitively

- A list of objects can be empty: This is recognized by the parser succeed called with [].
- A list of objects can be non-empty, i.e., it consists of an object followed by a list of objects: This is recognized by the sequentially composed parsers p and (list p):
 (p >*> list p).
- The parser build, finally, is used to turn a pair (x,xs) into the list (x:xs).

Lecture 6

Detailed Outline

Example 2: Parsing Arithm. Expressions (1) ...parsing arithmetic expressions like "(234+~42)*b", we shall construct the corresponding value of the algebraic data type: data Expr = Lit Int | Var Char | Op Ops Expr Expr data Ops = Add | Sub | Mul | Div | Mod Parsing "(234+~42)*b", e.g., shall yield the Exp-value:

Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')

...according to the below assumptions for string expressions:

- Variables are the lower case characters from 'a' to 'z'.
- Literals are of the form 67, ${\sim}89,$ etc., where ${\sim}$ is used for unary minus.
- Binary operators are +, *, -, /, %, where / and % represent integer division and modulo operation, respectively.
- Expressions are fully bracketed.
- White space is not permitted.

Lecture 6

Detailed Dutline

Example 2: Parsing Arithm. Expressions (2)

The parser for string expressions:

parser :: Parse Char Expr parser = nameParse 'alt' litParse 'alt' opExpParse

...is composed of three parsers reflecting the three kinds of expressions:

- variables (or variable names)
- literals (or numerals)
- fully bracketed operator expressions.

Lecture 6

Detailed Outline

15.1 15.2 15.2.1 15.2.2 15.2.3 15.2.4 15.2.5

15.3

15.5

Chap. 16

Example 2: Parsing Arithm. Expressions (3)

Parsing variable names:

nameParse :: Parse Char Expr nameParse = spot isName 'build' Var

isName :: Char -> Bool -- A variable name isName x = ('a' <= x && x <= 'z') -- must be a lower -- case character

Parsing literals (numerals):

litParse :: Parse Char Expr litParse

Lecture 6

Detailed Outline

```
Chap. 18
15.1
15.2
15.2.1
15.2.2
15.2.3
15.2.4
15.2.5
15.3
```

```
15.5
Than 16
```

```
Note
```

-- A literal starts

Example 2: Parsing Arithm. Expressions (4)

Parsing fully bracketed operator expressions:

optExpParse :: Parse Char Expr opExpParse -- A non-trivial expression = (token '(' >*> -- must start with an opening bracket, parser >*> -- must be followed by an expression, spot isOp >*> -- must be followed by an operator, parser >*> -- must be followed by an expression, token ')') -- must end with a closing bracket. 'build' makeExpr

Lecture 6

Detailed Outline

> 15.1 15.2 15.2.1 15.2.2

15.2.5 15.3

15.4 15.5

Chap. 16

Example 2: Parsing Arithm. Expressions (5) ...required supporting parser functions: neList :: Parse a b -> Parse a [b] optional :: Parse a b -> Parse a [b] where - neList p recognizes a non-empty list of the objects recognized by p. - optional p recognizes an object recognized by p or succeeds immediately.

Note: neList, optional, and some other supporting func-

tions including

- isOp
- charlistToExpr

are still be defined, left here as an exercise.

15 2 5

Example 2: Parsing Arithm. Expressions (6)

...we are left with defining a top-level parser function, which converts a string into an expression when called with parser:

Converting a string into the expression it represents:

topLevel :: Parse a b -> [a] -> b
topLevel p input
 = case results of
 [] -> error "parse unsuccessful"
 _ -> head results
 where

results = [found | (found, []) <- p input]</pre>

Note:

- The parse of an input is successful, if the result contains at least one parse, in which all the input has been read.
- topLevel parser "(234+~42)*b)" ->> Op Mul (Op Add (Lit 234) (Lit -42)) (Var 'b')

Detailed Outline

15.5

15.2.5

Chap. 10

Chapter 15.3 Monadic Parsing Lecture 6

Outline

Chap. 15

15.2 15.3

.3.2

5.3.5

.4

Chap. 16

Monadic Parsing

...complements the concept of combining forms underlying combinator parsing with the one of monads.

Since monads are 1-ary type constructors, the type of parser functions must be adjusted accordingly:

At the same time, we re-use the convention of Chapter 13.2 that delivery of the

- empty list signals failure of a parsing analysis.
- non-empty list signals success of a parsing analysis: each element of the list is a pair, whose first component is the identified object (token) and whose second component the input which is still to be parsed.

Lecture 6

Detailed Outline

15.1 15.2 **15.3** 15.3.1 15.3.2 15.3.3 15.3.4 15.3.5 15.3.6 15.4

Chap. 16

Chapter 15.3.1 The Parser Monad Lecture 6

Detailed Outline

.hap. 15

.2

15.3.1

.3.3

5.3.5

j.3.6

.

han 16

The Parser Monad

Recalling the definition of type class Monad:

where

```
parse :: (Parser a) -> (String -> [(a,String)])
parse (Parse p) = p
```

Remarks on the Parser Monad

Intuitively:

- The parser (return a) succeeds without consuming any of the argument string, and returns the single value a.
- parse denotes a deconstructor function for parsers defined by parse (Parse p) = p.
- The parser sequence p >>= f applies first parser (parse p) to the argument string cs yielding a list of results of the form (a, cs'), where a is a value and cs' is a string. For each such pair the parser (parse (f a)) is applied to the unconsumed input string cs'. The result is a list of lists which is concatenated to give the final list of results.

Lecture 6

Proof Obligation: The Monad Laws

...Parser satisfies the monad laws and is thus a valid instance of Monad. We have:

Lemma 15.3.1.1 (Soundness of Parser Monad)

- 1. return a >>= f = f a
- 2. $p \gg return = p$

3. p >>=
$$(a \rightarrow (f a \rightarrow g)) = (p \rightarrow (a \rightarrow f a)) \rightarrow =$$

Note:

- (>>=) being associative allows suppression of parentheses when parsers are applied sequentially.
- return being left-unit and right-unit for (>>=) allows some parser definitions to be simplified.

1531

g

Chapter 15.3.2 Parsers as Monadic Operations

15.3.2

Monadic Operations as Parsers

...Parser as an instance of Monad provides us already with two important parser functions, a primitive parser and a (monadic) parser combinator:

- 1. return, the always succeeding parser
- 6. (>>=), a combinator for sequentially combining parsers

which are the monadic counterparts of the combinator parsers



6. (>*>)

of Chapter 15.2.1 and 15.2.2, respectively.

The MonadPlus instance of Parser will give us two more parser functions... Lecture 6

Detailed Outline Chap. 15 15.1 15.3 15.3.1 15.3.2 15.3.3 15.3.4 15.3.5 15.3.6 15.4 15.5 Chap. 16

In more Detail

...the MonadPlus (cf. Chapter 12.6) instance of Parser:

class Monad m => MonadPlus m where
mzero :: m a
mplus :: m a -> m a -> m a

will provide us with the parser functions:

- 2. mzero, the always failing parser
- 5. mplus (via (++)), the parser for alternatives (or non-deterministic choice)

which are the monadic counterparts of the parser combinators

- 2. none
- 5. alt

of Chapter 15.2.1 and 15.2.2, respectively.

Lecture 6

Detailed Outline

> 5.1 5.2 5.3

15.3.2 15.3.3 15.3.4

15.3.6 15.4

Chap. 16

The Parser Monad-Plus

... yields the new parser functions mzero and mplus:

instance MonadPlus Parser where
-- The always failing parser
mzero = Parse (\cs -> [])

-- The parser combinator for alternatives: p 'mplus' q = Parse (\cs -> parse p cs ++ parse q cs)¹⁵⁴

Note: mplus can yield more than one result; the value of (parse p cs ++ parse q cs) can be a list of any length. In this sense mplus is considered to explore parsers alternatively (or, in this sense, non-deterministically). 1532

Proof Obligations: The Monad-Plus Laws

...we can prove that Parser satisfies the Monad-Plus laws:

Lemma 15.3.2.1 (Soundness of Parser Monad-Plus)

- 1. $p \gg (mzero) = mzero$
- 2. mzero >>= p = mzero
- 3. mzero 'mplus' p = p
- 4. p 'mplus' mzero = p

This means:

- mzero is left-zero and right-zero for (>>=).
- mzero is left-unit and right-unit for mplus.

Moreover

...we can prove the following laws:

Lemma 15.3.2.2 1. p 'mplus' (q 'mplus' r) = (p 'mplus' q) 'mplus' r $\frac{15.31}{15.31}$ 2. (p 'mplus' q) >>= f = (p >>= f) 'mplus' (q >>= $\frac{15.5}{15.51}$ 3. p >>= (\a -> f a 'mplus' g a) = (p >>= f) 'mplus' (p >>= $\frac{15.5}{15.52}$

This means:

- mplus is associative.
- (>>=) distributes through mplus.

Chapter 15.3.3 Universal Monadic Parser Basis

Lecture 6

Detailed Outline

> .1 .2

.3.1 .3.2

15.3.3 15.3.4

5.3.6

5.4

Chap 16

Towards a Universal Monadic Parser Basis

...in order to arrive at a universal monadic parser basis as in Chapter 15.2.3 we are left with defining monadic counterparts of the

- 3.,4. primitive parsers token and spot.
- 6. parser combinator build.

Lecture 6

Detailed Outline

15.1 15.2 15.3 15.3.1 15.3.2 **15.3.3**

> 15.3.5 15.3.6

15.4 15.5

Chap. 16

The Monadic Counterpart of Parser spot

...parser sat recognizes single characters satisfying a given property:

```
sat :: (Char -> Bool) -> Parser Char
sat p =
do {c <- item; if p c then return c else zero}</pre>
```

sat is the monadic counterpart of the parser function spot of Chapter 15.2.1.

Lecture 6

Detailed Dutline

5.1 5.2 5.3 15.3.1

15.3.3 15.3.4

15.3.6

Chap. 16

The Monadic Counterpart of Parser token

...parser char recognizes single characters; it is defined in terms of parser sat:

char :: Char -> Parser Char
char c = sat (== c)

char is the monadic counterpart of the parser function token of Chapter 15.2.1.

Lecture 6

Detailed Outline

Chap. 15

5.2

5.3.1

15.3.3

15.2.5

15.3.6

5

Chap. 16

The Universal Monadic Parser Basis (1) The type of parser functions: newtype Parser a = Parse (String -> [(a, String)]) Two input-independent primitive parser functions: 1533 1. The always succeeding parser function: return :: a \rightarrow Parser a return a = Parse (\cs \rightarrow [(a,cs)]) 2. The always failing parser function: mzero :: Parser a mzero = Parse (\cs \rightarrow [])

The Universal Monadic Parser Basis (2)

Lecture 6

Detailed Dutline

- Two input-dependent primitive parser functions:
 - 3. The parser for recognizing single objects:
 - char :: Char -> Parser Char
 char c = sat (== c)
 - The parser for recognizing single objects satisfying some property:

```
sat :: (Char -> Bool) -> Parser Char
sat p =
   do {c <- item; if p c then return c else zero}</pre>
```

1533

The Universal Monadic Parser Basis (3) Three parser combinators: 5 Alternatives mplus :: Parser a -> Parser a -> Parser a p 'mplus' q = Parse (\cs -> parse p cs ++ parse q cs) 1533 6. Sequencing (>>=) :: Parser a -> (a -> Parser b) -> Parser b p >>= f = Parse (\cs -> concat [(parse (f a)) cs'] $(a,cs') \leftarrow (parse p) cs])$ 7. Transformation mbuild :: Parser a \rightarrow (a \rightarrow b) \rightarrow Parser b mbuild p f inp = ... (completion left as homework)

Lecture 6

Detailed Outline

3.1

15.3.3 15.3.4

Chapter 15.3.4

Utility Parsers

.3.5

3.6

hap. 16

Utility Parsers (1)

Consuming the first character of an input string, if it is nonempty, and failing otherwise:

Parsing a specific string:

string :: String -> Parser String
string "" = return ""
string (c:cs) = do char c; string cs; return (c:cs)

1534

Utility Parsers (2)

The deterministically selecting parser:

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q
= Parse (\cs -> case parse (p 'mplus' q) cs of
        [] -> []
        (x:xs) -> [x])
```

Note:

- (+++) shows the same behavior as mplus, but yields at most one result (in this sense 'deterministically'), whereas mplus can yield several ones (in this sense 'non-deterministically')
- (+++) satisfies all of the previously listed properties of mplus.

Lecture 6

Detailed Outline

Utility Parsers (3)

Applying a parser p repeatedly:		
	15.1 15.2 15.3	
zero or more applications of p	15.3.1 15.3.2	
<pre>many :: Parser a -> Parser [a] many p = many1 p +++ return []</pre>	15.3.3 15.3.4	
	15.3.5 15.3.6	
one or more applications of p	15.4 15.5	
<pre>many1 :: Parser a -> Parser [a] many1 p = do a <- p; as <- many p; return (a:as)</pre>	Chap. 16 Note	
$\begin{array}{c} \text{many } p = u \\ u = u \\ p, a \\ u = v \\ u = u \\ u = v \\ u = v \\ u = u \\ u = v \\$	Note	

Note: As above, useful parsers are often recursively defined.

Utility Parsers (4)

A variant of the parser many with interspersed applications of parser sep, whose result values are thrown away:

Lecture 6

Detailed Outline

15.1 15.2 15.3 15.3.1 15.3.2 15.3.3 15.3.4 15.3.5 15.3.6 15.4 15.5

Utility Parsers (5)

Repeated applications of a parser p separated by applications of a parser op, whose result value is an operator which is assumed to associate to the left, and used to combine the results from the p parsers in chainl and chainl1:

15.3.4

Utility Parsers (6)

Handling white space, tabs, newlines, etc.

- Parsing a string with blanks, tabs, and newlines: space :: Parser String space = many (sat isSpace)
- Parsing a token by means of a parser p skipping any 'trailing' space:

token :: Parser a -> Parser a
token p = do {a <- p; space; return a}</pre>

- Parsing a symbolic token:

symb :: String -> Parser String
symb cs = token (string cs)

- Applying a parser p and throwing away any leading space: apply :: Parser a -> String -> [(a,String)] apply p = parse (do {space; p}) Lecture 6

Detailed Outline

> 15.1 15.2 15.3 15.3.1 15.3.2 15.3.3 15.3.4

NI. . .

Note

...parsers handling comments or keywords can be defined in a similar fashion allowing together avoidance of a dedicated lexical analysis (for token recognition), which typically precedes parsing.

Lecture 6

Detailed Outline

15.3.4

Chapter 15.3.5 Structure of a Monadic Parser

Lecture 6

Detailed Outline

Chap. 16

15.3.5

The Typical Structure of a Monadic Parser

...using the sequencing operator (>>=) or the syntactically sugared do-notation:

p1 >>= \a1 ->	do a1 <- p1
p2 >>= \a2 ->	a2 <- p2
	•••
pn >>= \an ->	an <- pn
f a1 a2 an	f a1 a2 an

...the latter one equivalently expressed in just one line, if so desired:

do {a1 <- p1; a2 <- p2;...; an <- pn; f a1 a2...an}

Recall: The expressions ai <- pi are called generators (since they generate values for the variables ai). Generators of the form ai <- pi can be replaced by pi, if the generated value will not be used afterwards.

Lecture 6

Detailed Outline

> 5.1 5.2 5.3

15.3.5 15.3.6

Chap. 1

Note

— ...

...the intuitive, natural operational reading of such a monadic parser:

- Apply parser p1 and call its result value a1.
- Apply subsequently parser p2 and call its result value a2.
- Apply subsequently parser ${\tt pn}$ and call its result value an.
- Combine finally the intermediate results by applying an appropiate function f.

Note, most typically f = return (g a1 a2 ... an); for an exception see parser chainl1 in Chapter 15.3.4, which needs to parse 'more of the argument string' before it can return a result.

Chapter 15.3.6 Writing Monadic Parsers: Examples

Lecture 6

Detailed Outline

hap. 15

.2 .3 5.3.1

15.3.3

15.3.5

15.3.6

15.4

Chap 16

Example 1: A Simple Parser

15.3.6

writing a parser p which
 reads three characters,
 drops the second character of these, and
- returns the first and the third character as a pair.
Implementation:

p :: Parser (Char, Char) p = do c <- item; item; d <- item; return (c,d)</pre>

Example 2: Parsing Arithm. Expressions (1)

...built up from single digits, the operators +, -, *, /, and parentheses, respecting the usual precedence rules for additive and multiplicative operators.

Grammar for arithmetic expressions:

expr	::=	expr addop term term
term	::=	term mulop factor factor
factor	::=	digit (expr)
digit	::=	0 1 9
addop	::=	+ -
mulop	::=	* /

Lecture 6

Detailed Outline

> 15.1 15.2 15.3 15.3.1 15.3.2 15.3.3 15.3.4 15.3.5 **15.3.6**

5.5

Chap. 16

Example 2: Parsing Arithm. Expressions (2)

The Parsing Problem:

Parsing expressions and evaluating them on-the-fly (yielding their integer values) using the chainl1 combinator of Chapter 15.3.4 to implement the left-recursive production rules for expr and term.

Lecture 6

Detailed Outline

> .nap. 13 15.1 15.2 15.3

15.3.2

15.3.4

1536

15.4

Chap. 16

Example 2: Parsing Arithm. Expressions (3)

The implementation of the parser expr: expr :: Parser Int addop :: Parser (Int -> Int -> Int) mulop :: Parser (Int -> Int -> Int) expr = term 'chainl1' addop term = factor 'chainl1' mulop 1536 factor =digit +++ do {symb "("; n <- expr; symb ")"; return n}</pre> digit = do {x <- token (sat isDIgit); return (ord x - ord '0')}</pre> addop = do {symb "+"; return (+)} +++ do {symb "-"; return (-)} mulop = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}

Example 2: Parsing Arithm. Expressions (4) ...using the parser. Parsing and evaluating the string "1 - 2 * 3 + 4 " onthe-fly by calling: apply expr " 1 - 2 * 3 + 4 " yields the singleton list: [(-1, "")]

which is the desired result.

15.3.6

Lecture 6

Detailed Dutline

Chap. 15

15.1 15.2 15.3

15.4

Chap. 16

Note

Chapter 15.4 Summary

In Conclusion

...combinator and monadic parsing rely (in part) on different language features but are quite similar in spirit as becomes obvious when opposing their primitives and combinators:

	Combinator Parsing	Monadic Parsing
Primitive	none	mzero
Parsers	succeed	return
	token	char
	spot	sat
Parser	alt	mplus
Combinators	(>*>)	(>>=)
	build	mbuild

Lecture 6

Detailed Outline

Chap. 15

15.2

15.3

15.5

Chap 16

Invaluable

... for combinator (as well as monadic) parsing are:

- Higher-order functions: Parse a b (like Parser a) is of a functional type; all parser combinators are thus higherorder functions.
- Polymorphism: The type Parse a b is polymorphic: We do need to be specific about either the input or the output type of the parsers we build. Hence, the parser combinators mentioned above can immediately be reused for tokens of any other data type (in the examples, these were lists and pairs, characters, and expressions).
- Lazy evaluation: 'On demand' generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

Lecture 6

Outline Chap. 1 15.1 15.2 15.3 15.4 15.5

Chap. 16

Chapter 15.5 References, Further Reading

Lecture 6

Detailed Outline

Chap. 15

15.2

15.4

15.5

Chap. 16

Chapter 15.2: Basic Reading

- Steve Hill. Combinators for Parsing Expressions. Journal of Functional Programming 6(3):445-464, 1996.
- Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- Simon Thompson. Haskell The Craft of Functional Programming. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 17.5, Case study: parsing expressions)
- Philip Wadler. How to Replace Failure with a List of Successes. In Proceedings of the 4th International Conference on Functional Programming and Computer Architecture (FPCA'85), Springer-V., LNCS 201, 113-128, 1985.

Lecture 6

Detailed Outline

15.2 15.3 15.4 15.5

1

Chapter 15.3: Basic Reading

- Graham Hutton. Programming in Haskell. Cambridge University Press, 2007. (Chapter 8, Functional parsers)
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 8(4):437-444, 1998.
- Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.

Lecture 6

Detailed Outline

Chap. 15 15.1

15.2 15.3 15.4 15.5

Chap. 16

Chapter 15: Selected Further Reading (1)

- Richard Bird. Introduction to Functional Programming using Haskell. Prentice-Hall, 2nd edition, 1998. (Chapter 11, Parsing)
- Jeroen Fokker. Functional Parsers. In Johan Jeuring, Erik Meijer (Eds.), Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques. Springer-V., LNCS 925, 1-23, 1995.
- Pieter W.M. Koopman, Marinus J. Plasmeijer. Efficient Combinator Parsers. In Proceedings of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Selected Papers, Springer-V., LNCS 1595, 120-136, 1999.

Lecture 6

Detailed Outline

Chap. 13 15.1 15.2 15.3

15.5

Chapter 15: Selected Further Reading (2)

- Andy Gill, Simon Marlow. Happy The Parser Generator for Haskell. University of Glasgow, 1995. www.haskell.org/happy
- Daan Leijen. Parsec, a free Monadic Parser Combinator Library for Haskell, 2003. legacy.cs.uu.nl/daan/parsec.html
- S. Doaitse Swierstra. Combinator Parsing: A Short Tutorial. In Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Revised Tutorial Lectures. Springer-V., LNCS 5520, 252-300, 2009.

Lecture 6

Detailed Outline

15.1 15.2

Chap. 16

Chapter 15: Selected Further Reading (3)

- S. Doaitse Swierstra, P. Azero Alcocer. Fast, Error Correcting Parser Combinators: A Short Tutorial. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer-V., LNCS 1725, 111-129, 1999.
 - Matthew Might, David Darais, Daniel Spiewak. Parsing with Derivatives – A Functional Pearl. In Proceedings of the 16th ACM International Conference on Functional Programming (ICFP 2011), 189-195, 2011.

Lecture 6

Detailed Outline

Chap. 15 15.1

15.3

15.5

Chap. 16

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

16.1 16.2 16.3

Chapter 16 Logic Programming Functionally

Logic Programming Functionally

Declarative programming

- Characterizing: Programs are declarative assertions about a problem rather than imperative solution procedures.
- Hence: Emphasizes the 'what,' rather than the 'how.'
- Important styles: Functional and logic programming.
- If each of these two styles is appealing for itself
 - (features of) functional and logic programming

uniformly combined in just one language should be even more appealing.

Question

- Can and shall (features of) functional and logic programming be uniformly combined? Lecture 6

Detailed Outline

Chap. 15

Chap. 16

16.1 16.2 16.3 16.4

...a recent article highlights important benefits of combining the paradigm features of functional and logic programming

Sergio Antoy, Michael Hanus. Functional Logic Programming. Communications of the ACM 53(4):74-85, 2010.

shedding thereby some light on this question.

...part of it is summarized in Chapter 16.1.

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

16.1	
16.2	
16.3	
16.4	

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

Chapter 16.1 Motivation

Chapter 16.1.1 On the Evolution of Programming Languages

Lecture o

Outline

Chap. 15

Chap. 16

16.1

16.1.1

16.1.3

16.1.4

16.2

16.2

16.4

The Evolution of Programming Languages (1)

...a continous and ongoing process of hiding the computer hardware and the details of program execution by the stepwise introduction of abstractions.

- Assembly languages
 - introduce mnemonic instructions and symbolic labels for hiding machine codes and addresses.
- FORTRAN
 - introduces arrays and expressions in standard mathematical notation for hiding registers.
- ALGOL-like languages
 - introduce structured statements for hiding gotos and jump labels.
- **Object-oriented** languages
 - introduce visibility levels and encapsulation for hiding the representation of data and the management of memory.

Lecture 6 Detailed Outline Chap. 16 16.1 16.1.1 16.1.3 16.1.4 16.2 16.3 16.4

Evolution of Programming Languages (2)

Declarative languages, most prominently functional and logic languages

- remove assignment and other control statements for hiding the order of evaluation.
 - A declarative program is a set of logic statements describing properties of the application domain.
 - The execution of a declarative program is the computation of the value(s) of an expression wrt these properties.

This way:

- The programming effort in a declarative language shifts from encoding the steps for computing a result to structuring the application data and the relationships between application components.
- Declarative languages are similar to formal specification languages but executable.

Lecture 6

Detailed Outline

Chap. 15

Chapter 16.1.2 Functional vs. Logic Languages

Lecture 6

Detailed Outline

Chap. 15

.nap. 10

16.1.2

6.1.3

16.2

10.5

Functional vs. Logic Languages

Functional languages

- are based on the notion of mathematical function.
- programs are sets of functions that operate on data structures and are defined by equations using case distinction and recursion.
- provide efficient, demand-driven evaluation strategies that support infinite structures.

Logic languages

- are based on predicate logic.
- programs are sets of predicates defined by restricted forms of logic formulas, such as Horn clauses (implications).
- provide non-determinism and predicates with multiple input/output modes that offer code reuse.

Lecture 6

Detailed Outline

Chap. 15

Functional Logic Languages (1)

...there are many: Curry, TOY, Mercury, Escher, Oz, HAL,...

Some of them in more detail:

- Curry

Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. Curry: A Truly Functional Logic Language. In Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming, 95-107, 1995.

See also: Michael Hanus (Ed.). Curry: An Integrated Functional Logic Language (vers. 0.8.2, 2006). http://www.curry-language.org/ Lecture 6

Detailed Dutline

Chap. 15

16.1 16.1.1 16.1.2 16.1.3 16.1.4 16.2 16.3 16.4

Functional Logic Languages (2)

- TOY

Francisco J. López-Fraguas, Jaime Sánchez-Hernández. TOY: A Multi-paradigm Declarative System. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), Springer-V., LNCS 1631, 244-247, 1999.

- Mercury

Zoltan Somogyi, Fergus Henderson, Thomas Conway. The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language. Journal of Logic Programming 29(1-3):17-64, 1996.

See also: The Mercury Programming Language http://www.mercurylang.org

Lecture 6

	16.1
	16.1.
Chapter 16.1.3	
	16.1.
	16.2
A Curry Appetizer	16.3 16.4
	10.4
	Note

A Curry Appetizer (1)	
Two important Curry operators:	
 – ?, denoting nondeterministic choice. 	
- =:=, indicating that an equation is to be solved rather	
than an operation to be defined.	Chap. 16 16.1 16.1.1
Example: Regular expressions and their semantics	16.1.2 16.1.3
data RE a = Lit a	16.1.4 16.2
Alt (RE a) (RE a)	16.3 16.4
Conc (RE a) (RE a)	Note
Star (RE a)	
sem :: RE a -> [a]	
sem (Lit c) = [c]	
sem (Alt r s) = sem r ? sem s	
sem (Conc r s) = sem r ++ sem s	
sem (Star r) = [] ? sem (Conc r (Star r))	

A Curry Appetizer (2)

 Evaluating the semantics of the regular expression abstar:

non-deterministically
sem abstar ->> ["a","ab","abb"]
where abstar = Conc (Lit 'a') (Star (Lit 'b'))

 Checking whether some word w is in the language of a regular expression re:

sem re =:= w

 Checking whether some string s contains a word generated by a regular expression re (similar to Unix's grep utility):

xs ++ sem re ++ ys =:= s Note: xs and ys are free! Lecture 6

Detailed Outline

Chap. 15

16.1 16.1.1 16.1.2 **16.1.3** 16.1.4 16.2 16.3

Chapter 16.1.4 Outline

Lecture 6

Detailed Outline

Chap. 15

hap. 10

5.1.1

6.1.3

16.1.4

10.0

16.4

16.1.4

- Less ambitious: Implementing an interpreter for one style using the other style.
- Even less ambitious: Developing a combinator library allowing us to write logic programs in Haskell.

Combining Functional and Logic Programming

...some principal approaches for combining their features:

- Ambitious: Designing a new programming language enjoying features of both programming styles (e.g., Curry, Mercury, etc.).

108/200

Here

...we follow the last approach as proposed by Michael Spivey and Silvija Seres in:

 Michael Spivey, Silvija Seres. Combinators for Logic Programming. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.

Central are:

- Combinators
- Monads
- Combinator and monadic programming.

Lecture 6

Detailed Outline

Chap. 15

Chap. 10 16.1 16.1.1 16.1.2 16.1.3 16.1.4 16.2

Note

Benefits and Limitations

...of this combinator approach compared to approaches striving for fully functional/logic programming languages:

- Less costly

but also

- less expressive and (likely) less performant.

Lecture 6

Detailed Outline

Chap. 15

.hap. 16 16.1 16.1.1

16.1.2

16.1.4

0.2

16.4

Note

Chapter 16.2 The Combinator Approach

Lecture o

Outline

Chap. 15

16.2

5.2.1

.2.4

0.2.6

16.2.8

16.2.9

6.4

Note

Chapter 16.2.1 Introduction

Lecture 6

Detailed Outline

Chap. 15

.1

.2.2

16.2.1

6.2.5

o.2.7

16.2.9

6.4

NUMBER

Three Key Problems

... are to be solved in the course of developing this approach:

Modelling

- 1. logic programs yielding (possibly) multiple answers → using the lists of successes technique
- 2. the evaluation/search strategy inherent to logic programs → encapsulating the search strategy in 'search monads'
- logical variables (no distinction between input and output variables)
 → realizing unification

Lecture 6

Detailed Outline

Chap. 15

Key Problem 1: Multiple Answers

...can easily be handled (re-) using the technique of

- lists of successes (lazy lists) (Philip Wadler, 1985)

Intuitively

- Any function of type (a -> b) can be replaced by a function of type (a -> [b]).
- Lazy evaluation ensures that the elements of the result list (i.e., the list of successes) are provided as they are found, rather than as a complete list after termination of the computation.

Lecture 6

Chap. 15

16.1
 16.2
 16.2.2
 16.2.3
 16.2.4
 16.2.5
 16.2.6
 16.2.7
 16.2.8
 16.2.9
 16.3
 16.4

Key Problem 2: Evaluation/Search Strategies

...dealt with investigating an illustrating running example.

This is factoring of natural numbers: Decomposing a positive integer into the set of pairs of its factors, e.g.:

Obviously, this problem (instance) is solved by:

In fact, we get:

factor 24 ->> [(1,24),(2,12),(3,8),(4,6),(6,4),(8,3),(12,2),(24,1)]

Note

When implementing the 'obvious' solution we exploit explicit domain knowledge:

 Most importantly the domain fact:
 - r * s = n ⇒ r ≤ n ∧ s ≤ n which allows us to restrict our search to a finite space: [1..24]×[1..24]

Often, however, such knowledge is not available:

Generally, the search space cannot be restricted a priori!

In the following, we thus consider the factoring problem as a

search problem over the infinite 2-dimensional search space:

 $[1..] \times [1..]$

Lecture 6

Detailed Outline

Chap. 15

Illustrating the Search Space $[1..] \times [1..]$

Lecture 6

Detailed Dutline

	1	2	3	4	5	6	7	8	9	 Chap. 15
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	 Chap. 16 16.1
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	 16.2 16.2.1
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	 16.2.2
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	 16.2.3 16.2.4
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	 16.2.5 16.2.6
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	 16.2.7
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	 16.2.8 16.2.9
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	 16.3
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	 Note

Back to the Running Example

...adapting function factor straightforward to the infinite search space $[1..] \times [1..]$ yields:

Applying factor to the argument 24 yields:

factor 24 ->> [(1,24)

...followed by an infinite wait.

This is useless and of no practical value!

Lecture 6

Detailed Outline

Chap. 15

16.2.1 16.2.1 16.2.2 16.2.3 16.2.4 16.2.5

118/200

The Problem: Unfair Depth-first Search

...the two-dimensional space is searched in a depth-first order:

2 3 5 8 9 1 4 6 7 Chap. 16 1 (1,1)(1.2)(1,3)(1,4)(1,5)(1.6)(1.7)(1,8)(1,9)2 (2,3) (2,1)(2,2)(2, 4)2,5) 2,6 [2,7] 2,8) 2,9 . . . 16.2.1 3 (3,1)(3,4) (3,9) 3.2) 3.3) 3.5) 3.6 3,7 3.8) . . . (4,3) 4 (4,1)4,2) (4,4) 4,5) 4,6 (4,7) 4,8) (4,9) 5 5.2) (5,3) (5,4) (5.5)5.6 5.7 5.8) 5.9 5.16 (6,1)(6, 2)(6,3)(6, 4)(6,5)6,6 (6,7)6,8) 6,9 7 7,1) 7.2) 7.3) (7.5) 7.8) (7,4) 7.6 7,7 (7.9) 8 (8,1)(8,2) 8.3) (8.4)8,5) 8,6 (8,7) (8,8) 8,9 . . . 9 9.1)(9.2) 9.3) (9.4) (9.5) 9.6) (9.7) (9.8) (9,9) Note . . .

This search order is unfair: Pairs in rows 2 onwards will never be reached and considered for being a factor pair.

	16.1
	16.2
Chapter 16.2.2	16.2.1
	16.2.2 16.2.3
	16.2.3
Distance in the set	16.2.5
Diagonalization	16.2.6
Diagonalization	16.2.7
	16.2.8
	16.2.9

Diagonalization to the Rescue (1)

...searching the infinite number of finite diagonals ensures fairness, i.e., every pair will deterministically be visited after a finite number of steps:

	1	2	3	4	5	6	7	8	9	 Outline Chap. 15
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	 Chap. 16
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	 16.1
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	 -16.2 16.2.1
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	 16.2.2 16.2.3
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9)	 16.2.4
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)	(6,9)	 - 16.2.5 16.2.6
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9)	 16.2.7 16.2.8
8	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9)	 16.2.9
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)	 - 16.3 16.4
										 Note

- Diagonal 1: [(1,1)]
- Diagonal 2: [(1,2),(2,1)]
- Diagonal 3: [(1,3),(2,2),(3,1)]
- Diagonal 4: [(1,4),(2,3),(3,2),(4,1)]
- Diagonal 5: [(1,5),(2,4),(3,3),((4,2),(5,1)]

Diagonalization to the Rescue (2)

In fact, on visiting the infinite number of finite diagonals, every pair (i, j) of the infinite 2-dimensional search space $[1..] \times [1..]$ is deterministically reached after a finite number of steps as illustrated below:

								16.1
	1	2	3	4	5	6	7	 16.2 16.2.1
1	$(1,1)_{1}$	(1,2) ₂	(1,3)4	(1,4)7	(1,5) ₁₁	(1,6) ₁₆	$(1,7)_{22}$	 16.2.2 16.2.3
2	(2,1) 3	(2,2)5	(2,3) 8	$(2,4)_{12}$	(2,5)17	$(2,6)_{23}$	(2,7) ₃₀	 16.2.4
3	(3,1) ₆	(3,2)g	(3,3) ₁₃	(3,4) ₁₈	(3,5) ₂₄	(3,6) ₃₁	(3,7) ₃₉	 16.2.5 16.2.6
4	(4,1) ₁₀	$(4,2)_{14}$	(4,3) ₁₉	$(4,4)_{25}$	$(4,5)_{32}$	(4,6) ₄₀	(4,7) 49	 16.2.7 16.2.8
5	$(5,1)_{15}$	(5,2) ₂₀	(5,3) ₂₆	(5,4)33	(5,5)41	(5,6) 50	$(5,7)_{60}$	 16.2.9
6	$(6,1)_{21}$	(6,2)27	(6,3) ₃₄	(6,4) ₄₂	(6,5) 51	(6,6) ₆₁	(6,7) ₇₂	 16.3 16.4
7	$(7,1)_{28}$	(7,2)35	(7,3) ₄₃	(7,4) ₅₂	$(7,5)_{62}$	(7,6) ₇₃	$(7,7)_{85}$	 Note
8	(8,1) ₃₆	(8,2)44	(8,3) 53	(8,4) ₆₃	(8,5)74	(8,6) ₈₆	(8,7)99	
9	(9,1)45	(9,2) ₅₄	(9,3) ₆₄	(9,4) ₇₅	(9,5) ₈₇	(9,6) ₁₀₀	(9,7) ₁₁₄	

Implementing Diagonalization (1)

...function diagprod realizes the diagonalization strategy: It enumerates the cartesian product of its argument lists in a fair order, i.e., every element is enumerated after some finite amount of time:

E.g., applied to the infinite 2-dimensional space $[1..] \times [1..]$, diagprod ejects every pair (x,y) of $[1..] \times [1..]$ in finite time:

 $[(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), (4,1), (1,5), (2,4), (3,3), (4,2), (5,1), (1,6), (2,5), \dots, (6,1), (1,7), (2,6), \dots, (7,1), \dots$

Implementing Diagonalization (2)

diagprod :: [a] -> [b] -> [(a,b)]
diagprod xs ys = [(xs!!i, ys!!(n-i)) | n<-[0..], i<-[0..n]]</pre>

n	i	n-i	(xs!!i, ys!!(n-i))	([1]!!i, [1]!!(n-i))	#	Diag. #	
0	0	0	(xs!! <mark>0</mark> ,ys!! <mark>0</mark>)	(1,1)	1	1	
1	0	1	(xs!! <mark>0</mark> ,ys!!1)	(1,2)	2	2	16.1
1	1	0	(xs!! <mark>1</mark> ,ys!! <mark>0</mark>)	(2,1)	3		16.2 16.2.1
2	0	2	(xs!! <mark>0</mark> ,ys!!2)	(1,3)	4	3	16.2.2 16.2.3
2	1	1	(xs!! 1 ,ys!! 1)	(2,2)	5		16.2.4
2	2	0	(xs!! <mark>2</mark> ,ys!! <mark>0</mark>)	(3,1)	6		16.2.5 16.2.6
3	0	3	(xs!! <mark>0</mark> ,ys!! <mark>3</mark>)	(1,4)	7	4	16.2.7 16.2.8
3	1	2	(xs!!1,ys!!2)	(2,3)	8		16.2.9
3	2	1	(xs!!2,ys!!1)	(3,2)	9		16.3 16.4
3	3	0	(xs!! <mark>3</mark> ,ys!! <mark>0</mark>)	(4,1)	10		Note
4	0	4	(xs!! <mark>0</mark> ,ys!! <mark>4</mark>)	(1,5)	11	5	
4	1	3	(xs!! 1 ,ys!! 3)	(2,4)	12		
4	2	2	(xs!! <mark>2</mark> ,ys!! <mark>2</mark>)	(3,3)	13		
4	3	1	(xs!! <mark>3</mark> ,ys!!1)	(4,2)	14		
4	4	0	(xs!! <mark>4</mark> ,ys!! <mark>0</mark>)	(5,1)	15		
							124/200

Back to the Running Example

...let's adjust factor in a way such that it explores the search space of pairs in a fair order using diagonalization:

Applying now factor to the argument 24, we obtain:

factor 24 ->> [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)

...i.e., we obtain all results, followed by an infinite wait.

Of course, this is not surprising, since the search space is infinite.

Chapter 16.2.3 Diagonalization with Monads

Lecture 6

Detailed Outline

Chap. 15

Finite Lists, Infinite Streams, Monads

...in the following we conceptually distinguish between:

- [a]: Finite lists.

- Stream a: Infinite lists defined as type alias by:

type Stream a = [a]

Note: Distinguishing between (Stream a) for infinite lists and [a] for finite lists is conceptually and notationally only as is made explicit by defining (Stream a) as a type alias of [a].

Like [], Stream is a 1-ary type constructor and can thus be made an instance of type class Monad:

```
class Monad m where
return :: a -> m a
(>>=) :: m a -> (a -> m b) -> m b
```

The Stream Monad

...since (Stream a) is a type alias of [a], the stream and the list monad coincide; the bind (>>=) and return operation of the stream monad

- (>>=) :: Stream a -> (a -> Stream b) -> Stream b - return :: a -> Stream a

are thus defined as in Chapter 12.4.2:

instance Monad Stream where
 xs >>= f = concat (map f xs)
 return x = [x] -- yields the singleton list

Note: The monad operations (>>) and fail are not relevant in the following, and thus omitted.

Notational Benefit (1)

...the monad operations return and (>>=) for lists and streams allow us to avoid or replace list comprehension:

E.g., the expression

[(x,y) | x <- [1..], y <- [10..]]

using list comprehension is equivalent to the expression

 $[1..] >>= (\langle x -> [10..] >>= (\langle y -> return (x,y)))$

using monad operations; this is is made explicit by stepwise unfolding the monadic expression yielding first the equivalent expression:

concat (map ($x \rightarrow [(x,y) | y <- [10..]])[1..]$) and second the equivalent expression:

concat

(map (\x -> concat (map (\y -> [(x,y)])[10..]))[1..])

Lecture 6

Detailed Outline

Chap. 15

16.1 16.2 16.2.1 16.2.2 **16.2.3** 16.2.4 16.2.5 16.2.6 16.2.7 16.2.8 16.2.9 16.3

)

Notational Benefit (2)

Lecture 6

By exploiting the general rule that do x1 <- e1; x2 <- e2; ...; xn <- en; e is a shorthand for $e1 \gg = (\x1 \rightarrow e2 \gg = (\x2 \rightarrow ... \gg) = (\xn \rightarrow e)...))$

lent representation:

do x <- [1..]; y <- [10..]; return (x,y)

Note

...exploring the pairs of the search space using the stream monad is not yet fair.

E.g., the expression:

do x <- [1..]; y <- [10..]; return (x,y)

yields the infinite list (i.e., stream):

 $[(1,10),(1,11),(1,12),(1,13),(1,14),\ldots]$

Note

16.2.3

.. the fairness issue is only handled by defining another monad.

Towards a Fair Binding Operation (>>=)

...idea: Embedding diagonalization into (>>=).

To this end, we introduce a new polymorphic type Diag:

newtype Diag a = MkDiag (Stream a) deriving Show

together with a utility function for stripping off the data constructor MkDiag:

unDiag :: Diag a -> a unDiag (MkDiag xs) = xs

Lecture 6

Detailed Outline

Chap. 15

The Diag(onalization) Monad

...making Diag an instance of the type constructor class Monad:

instance Monad Diag where return x = MkDiag [x] MkDiag xs >>= f = MkDiag (concat (diag (map (unDiag . f) xs)))

where diag rearranges the values into a fair order:

diag :: Stream (Stream a) -> Stream [a] diag [] = [] diag (xs:xss) = lzw (++) [[x] | x <- xs] ([] : diag xss)</pre>

Utility Function 1zw

...using itself the utility function lzw ('like zipWith.'):

Note: lzw equals zipWith except that the non-empty remainder of a non-empty argument list is attached, if one of the argument lists gets empty.

Lecture 6

Detailed Outline

Chap. 15

Note

- ...for monad Diag holds:
 - return yields the singleton list.
 - undiag strips off the constructor added by the function f :: a -> Diag b.
 - diag arranges the elements of the list into a fair order (and works equally well for finite and infinite lists).

Detailed

Chap. 15

Illustrating

... the idea underlying the map diag: Transform an infinite list of infinite lists: [x11,x12,x13,x14,..], [x21,x22,x23,..], [x31,x32,..], ...]into an infinite list of finite diagonals: [[x11], [x12, x21], [x13, x22, x31], [x14, x23, x32, ...], ...]This way, we get: do x <- MkDiag [1..]; y <- MkDiag [10..]; return (x,y) ->> MkDiag [(1,10),(1,11),(2,10),(1,12),(2,11), $(3,10),(1,13),\ldots$

which means, we are done:

- The pairs are delivered in a fair order!

Back to the Factoring Problem

... the current status of our approach:

- Generating pairs (in a fair order): Done.
- Selecting the pairs being part of the solution: Still open.

Next, we are going to tackle the selection problem, i.e., filtering out the pairs (r, s) satisfying the equality $r \times s = n$, by:

Filtering with conditions!

To this end, we introduce a new type constructor class Bunch.

Detailed Outline

Chap. 15

16.1 16.2 16.2.1 16.2.2 **16.2.3** 16.2.4 16.2.5 16.2.6 16.2.7 16.2.8 16.2.9 16.3 16.4

Lecture 6

Detailed Outline

Chap. 15

16.2.4

Chapter 16.2.4 Filtering with Conditions

138/200

The Type Constructor Class Bunch

```
... is defined by:
 class Monad m \Rightarrow Bunch m where
  -- Empty result (or no answer)
 zero :: m a
  -- All answers in xm or ym
  alt :: m a -> m a -> m a
 -- Answers yielded by 'auxiliary calculations'
 -- (for now, think of wrap in terms of the
 -- identity, i.e., wrap = id)
 wrap :: m a -> m a
```

Note: zero allows to express that a set of answers is empty; alt allows to join two sets of answers.

1624

```
Making [] and Diag Instances of Bunch
 ...making (lazy) lists and Diag instances of Bunch:
  instance Bunch [] where
  zero = []
  alt xs ys = xs ++ ys
  wrap xs = xs
  instance Bunch Diag where
                                                         16.2.4
  zero = MkDiag []
  alt (MkDiag xs) (MkDiag ys) -- shuffle in the
           = MkDiag (shuffle xs ys) -- interest of
                                      -- fairness
  wrap xm = xm
  shuffle :: [a] \rightarrow [a] \rightarrow [a]
  shuffle [] ys = ys
  shuffle (x:xs) ys = x : shuffle ys xs
 Note: wrap will only be used be used in Chapter 16.2.5 on-
wards
```

Filtering with Conditions using test

Using zero, the function test, which might not look useful at first sight, yields the key for filtering:

test :: Bunch m => Bool -> m () -- () type $idf_{chap.16}$ test b = if b then return () else zero -- () value $idf_{chap.16}^{chap.16}$

In fact, all do-expressions filter as desired, i.e., the multiples of 3 from the streams [1..] and MkDiag [1..], respectively:

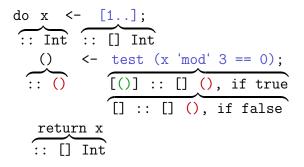
do x <- [1..]; () <- test (x 'mod' 3 == 0); return x
->> [3,6,9,12,15,18,21,24,27,30,33,...

do x <- [1..]; test (x 'mod' 3 == 0); return x ->> [3,6,9,12,15,18,21,24,27,30,33,..

do x <- MkDiag [1..]; test (x 'mod' 3 == 0); return x
->> MkDiag [3,6,9,12,15,18,21,24,27,30,33,..

A note on test

In more detail:



...if test evaluates to true, it returns the value (), and the rest of the program is evaluated. If it evaluates to false, it returns zero, and the rest of the program is skipped for this value of x. This means, return x is only reached and evaluated for those values of x with x 'mod' 3 equals 0.

Lecture 6

Detailed Outline

Chap. 15

Nonetheless

...we are not yet done as the below example shows:

<pre>do r <- MkDiag [1]; s <- MkDiag [1];</pre>
test (r*s==24); return (r,s)
->> MkDiag [(1,24)

...followed again by an infinite wait.

Why is that?

The above expression is equivalent to:

Why is that? (1)

...this means the generator for y is merged with the subsequent test to the (sub-) expression:

Intuitively

- This expression yields for a given value of x all values of y with x * y = 24.
- For x = 1 the answer (1, 24) will be found, in order to then search in vain for further fitting values of y.
- For x = 5 we thus would not observe any output, since an infinite search would be initiated for values of y satisfying 5 * y = 24.

Why is that? (2)

...the deeper reason for this (undesired) behaviour:

The bind operation (>>=) of Diag is not associative, i.e., $xm \Rightarrow (x \rightarrow f x \Rightarrow g) = (xm \Rightarrow f) \Rightarrow g$...does not hold! Or, equivalently expressed using do:

...does not hold.

Lecture 6

Detailed Dutline

Chap. 15

16.1
16.2
16.2.1
16.2.2
16.2.3
16.2.4
16.2.5
16.2.6
16.2.7
16.2.8
16.2.9
16.3
16.4

Overcoming the Problem

...frankly, Diag is not a valid instance of Monad, since it fails the monad law of associativity for (>>=). The order of applying generators is thus essential.

For taking this into account, the generators are explicitly pairwise grouped together to ensure they are treated fairly by diagonalization:

...yields now all results, followed, of course, by an infinite wait (due to an infinite search space).

This means, the problem is fixed. We are done.

Note

Getting all results followed by an infinite wait is

the best we can hope for if the search space is infinite.

Explicit grouping is

only required because Diag is not a valid instance of Monad since its bind operation (>>=) fails to be associative. If it were, both expressions would be equivalent and explicit grouping unnecessary.

Next, we will strive for

avoiding/replacing infinite waiting by indicating search progress, i.e., by indicating from time to time that a(nother) result has not (yet) been found.

Lecture 6

Detailed Outline

Chap. 15

Chapter 16.2.5 Indicating Search Progress

Indicating Search Progress

...to this end, we introduce a new type Matrix together with a cost-guided diagonalization search, a true breadth search.

Intuitively

- Values of type Matrix: Infinite lists of finite lists.
- Goal: A program which yields a matrix of answers, where row *i* contains all answers which can be computed with costs c(*i*) specific for row *i*.
- Indicating progress: If the list returned as row k is the empty list, this means 'nothing found,' i.e., the set of solutions which can be computed with costs c(k) is empty.

Lecture 6

Detailed Outline Chap. 15 Chap. 16 16.1

The Type Matrix

_

The new type Matrix:			
newtype Matrix a =	=		
MkMatrix (Stream	[a])	deriving	Show

...and a utility function for stripping off the data constructor:

unMatrix	:: Matrix a -> a	
unMatrix	(MkMatrix xm) = xm	

Lecture 6

Detailed Outline

Chap. 15

16.2.1 16.2.2 16.2.3 16.2.4 **16.2.5** 16.2.6

16.2.7 16.2.8

16.2.9

. .

Vote

Towards Matrix an Instance of Bunch (1)

...preliminary reasoning about the required operations and their properties:

```
-- Matrix with a single row
return x = MkMatrix [[x]]
-- Matrix without rows
zero = MkMatrix []
-- Concatenating corresponding rows
alt (MkMatrix xm) (MkMatrix ym) =
 MkMatrix (lzw (++) xm ym)
-- Taking care of the cost management!
wrap (MkMatrix xm) = MkMatrix ([]:xm)
```

Lecture 6

Detailed Outline

Chap. 15

```
16.1
16.2
16.2.1
16.2.2
16.2.3
16.2.4
16.2.5
16.2.6
16.2.7
16.2.8
16.2.9
16.3
```

Towards Matrix an Instance of Bunch (2)

{- (>>=) is essentially defined in terms of bindm; it
handles the data constructor MkMatrix which is not
done by bindm. -}

(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix f))

{- bindm is almost the same as (>>=) but without bothering about MkMatrix; it applies f to all the values in xm and collects together the results in a matrix according to their total cost: these are the costs of the argument of f given by xm plus the cost of computing its result. -}

bindm :: Stream[a] -> (a -> Stream[b]) -> Stream [b] bindm xm f = map concat (diag (map (concatAll . map f) xm))

{- A variant of the concat function using lzw. -}
concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (++)) []

Making Matrix an Instance of Bunch

...now we are ready to make Matrix an instance of the type constructor classes Monad and Bunch:

```
instance Monad Matrix where
                     = MkMatrix [[x]]
return x
 (MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix.f))
instance Bunch Matrix where
                                 = MkMatrix []
zero
                                                          16.2.5
alt( MkMatrix xm) (MkMatrix ym) =
  MkMatrix (lzw (++) xm ym)
wrap (MkMatrix xm) = -- 'wrap xm' yields a matrix w/
 MkMatrix ([]:xm) -- the same answers but each
                        -- with a cost one higher than
                        -- its cost in 'xm'
intMat = MkMatrix [[n] | n <- [1..]] -- intMat replaces</pre>
                                       -- stream [1..]
```

Using intMat and Matrix

...consider the expression:

Intuitively

- Diagonals 1 to 8: No factor pairs of 24 were found (indicated by []).
- Diagonal 9: The factor pairs (4,6) and (6,4) were found.
- Diagonal 10: The factor pairs (3,8) and (8,3) were found.
- Diagonals 11 to 12: No factor pairs of 24 were found (ind'd by []).
- Diagonal 13: The factor pairs (2,12) and (12,2) were found.

...if a diagonal d does not contain a valid factor pair, we get []; otherwise we get the list of valid factor pairs located in d.

I.e., we are done: Infinite waiting is replaced by progress indication!

Illustrating the Location

...of the factor pairs of 24 in the diagonals of the search space by $!(\cdot, \cdot)!$:

	1	2	3	4	5	6	7	8	9 Chap. 16
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9) 10.2
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9) 6.2.2
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	!(3,8)!	(3,9) 6.2.4
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	!(4,6)!	(4,7)	(4,8)	(4,9) 6.2.5
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)	(5,9) 6.2.7
6	(6,1)	(6,2)	(6,3)	!(6,4)!	(6,5)	(6,6)	(6,7)	(6,8)	(6,9) 16.2.9
7	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)	(7,9) 16.4
8	(8,1)	(8,2)	!(8,3)!	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)	(8,9) Note
9	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)

Lecture 6

Detailed Outline

.nap. 15

Chapter 16.2.6 Selecting a Search Strategy

Lecture 6

Detailed Outline

Chap. 15

2.2.1

.2.5

16.2.6

629

An Array of Search Strategies

... is now at our disposal, namely

- 1. Depth search ([1..])
- 2. Diagonalization (MkDiag [[n] | n<-[1..]])
- 3. Breadth search (MkMatrix [[n] | n<-[1..]])

...and we can choose each of them at the very last moment, just by picking the right monad when calling a function:

-- Picking the desired search strategy by choos--- ing m accordingly at the time of calling factor factor :: Bunch m => Int -> m (Int, Int) factor n = do r <- choose [1..]; s <- choose [1..]; test (r*s==n); return (r,s)

choose :: Bunch m => Stream a -> m a
choose (x:xs) = wrap (return x 'alt' choose xs)

Picking a Search Strategy at Call Time	
specifying the result type of factor when calling it selects	
the search monad and thus the search strategy applied.	
Illustrated in terms of our running example:	
Depth Search: Picking Stream	16.1 16.2 16.2.1
<pre>factor 24 :: Stream (Int,Int) ->> [(1,24)</pre>	16.2.2 16.2.3 16.2.4
Diagonalization Search: Picking Diag factor 24 :: Diag (Int, Int)	16.2.5 16.2.6 16.2.7 16.2.8
->> MkDiag [(4,6),(6,4),(3,8),(8,3),(2,12),(12,2), (1,24),(24,1)	16.2.9 16.3 16.4
Breadth Search w/ Progress Indication: Picking Matri	Note X
<pre>factor 24 :: Matrix (Int, Int) ->> MkMatrix [[],[],[],[],[],[],[],[],[],[],[],[],[],</pre>	

Summarizing our Progress so Far

...recall the 3 key problems we have or had to deal with.

Modelling

- logic programs yielding (possibly) multiple answers: Done (using lazy lists).
- 2. the evaluation strategy inherent to logic programs: Done.
 - The search strategy implicit to logic programming languages has been made explicit. The type constructors and type classes of Haskell allow even different search strategies and to pick one conveniently at call time.
- 3. logical variables (i.e., no distinction between input and output variables): Still open!

Lecture 6

Detailed Outline

Chap. 15

Next

...we tackle this third problem, i.e.:

Modelling

 logical variables (i.e., no distinction between input and output variables).

Common for evaluating logic programs

...not a pure simplification of an initially completely given expression but a simplification of an expression containing variables, for which appropriate values have to be determined. In the course of the computation, variables can be replaced by other subexpressions containing variables themselves, for which then appropriate values have to be found.

Fundamental: Substitution, unification.

Lecture 6

Detailed Outline

Chap. 15

Lhap, 10 16.1 16.2 16.2.1 16.2.2 16.2.3 16.2.4 16.2.5 16.2.6 16.2.7 16.2.8 16.2.9 16.3 16.4

Chapter 16.2.7 Terms, Substitutions, Unification, and Predicates

Terms (1)

towards logical variables — we introduce a type for terms:	
Terms	
data Term = Int Int Nil Cons Term Term Var Variable deriving Eq	Chap. 1 16.1 16.2 16.2.1 16.2.2 16.2.3 16.2.4 16.2.5
will describe values of logic variables.	16.2.6 16.2.7 16.2.8 16.2.9
Named variables and generated variables data Variable = Named String Generated Int deriving (Show, Eq)	16.3 16.4 Note
will be used for formulating queries, respectively, evolve in the course of the computation.	

Terms (2)

Utility functions for transforming

 a string into a named variable: var :: String -> Term var s = Var (Named s)
 a list of integers into a term: list :: [Int] -> Term

16.1
16.2
16.2.1
16.2.2
16.2.3
16.2.4
16.2.5
16.2.6
16.2.7
16.2.8
16.2.9
16.3

list xs = foldr Cons Nil (map Int xs)

Substitutions (1)	
Substitutions	
newtype Subst = MkSubst [(Var,Term)]	
essentially mappings from variables to terms.	Chap. 16 16.1 16.2
Support functions for substitutions:	16.2.1 16.2.2 16.2.3 16.2.4
unSubst :: Subst -> [(Var,Term)]	16.2.5 16.2.6 16.2.7
unSubst (MkSubst s) = s	16.2.8 16.2.9 16.3
idsubst :: Subst idsubst = MkSubst []	16.4 Note
extend :: Var -> Term -> Subst -> Subst	
extend x t (MkSubst s) = MkSubst ((x:t):s)	

Substitutions (2)

Applying a substitution:

```
apply :: Subst -> Term -> Term
apply s t = -- Replace each variable
case deref s t of -- in t by its image under s
Cons x xs -> Cons (apply s x) (apply s xs)
t' -> t'
```

where

```
deref :: Subst -> Term -> Term
deref s (Var v) =
   case lookup v (unSubst s) of
    Just t -> deref s t
    Nothing -> Var v
deref s t = t
```

Lecture 6

Detailed Outline

Chap. 15

16.1 16.2 16.2.1 16.2.2 16.2.3 16.2.4 16.2.5 16.2.6 16.2.7

16.2.9

16.4

Term Unification (1)	
unifying terms:	Lecture 6
unify :: (Term, Term) -> Subst -> Maybe Subst	Detailed Outline
unify (t,u) s =	Chap. 15 Chap. 16
case (deref s t, deref s u) of	16.1
(Nil, Nil) -> Just s (Cons x xs, Cons y ys) ->	16.2.1 16.2.2
unify (x,y) s >>= unify (xs, ys)	16.2.3 16.2.4
(Int n, Int m) (n==m) -> Just s	16.2.5 16.2.6
(Var x, Var y) (x==y) -> Just s	16.2.7 16.2.8
(Var x, t) -> if occurs x t s	16.2.9 16.3
then Nothing	16.4
else Just (extend x t s)	Note
(t, Var x) -> if occurs x t s	
then Nothing	
else Just (extend x t s)	
(_,_) -> Nothing	

Term Unification (2)

where

Outline

Chap. 15

16.2.5 16.2.6 16.2.7 16.2.8 16.2.9

16.4

Predicates: Modelling Logic Programs (1)

...in our scenario m is of type bunch.

```
Logic programs are of type:
```

```
type Pred m = Answer -> m Answer
```

...intuitively, applied to an 'input' answer which provides the information that is already decided about the values of variables, an array of new answers is computed, each of them satisfying the constraints expressed in the program.

Answers are of type:

```
newtype Answer = MkAnswer (Subst,Int)
```

...intuitively, the substitution carries the information about the values of variables; the integer value counts how many variables have been generated so far allowing to generate fresh variables when needed.

Lecture 6

Detailed Outline

Chap. 15

Predicates: Modelling Logic Programs (2)	
Initial 'input' answer:	
initial :: Answer	
initial = MkAnswer (idsubst, 0)	
Logical program run: Predicate p as query is applied to the initial 'input' answer:	16.1 16.2 16.2.1 16.2.2 16.2.3
run :: Bunch m => Pred m -> m Answer run p = p initial	16.2.4 16.2.5 16.2.6 16.2.7 16.2.8
Example: Choosing Stream for m allows evaluating the predi- cate append (defined later):	16.2.9 16.3 16.4 Note
<pre>run (append (list [1,2],list [3,4],var "z"))</pre>	
->> [{z=[1,2,3,4]}] an appropriate show function is assumed	

Chapter 16.2.8 Combinators for Logic Programs

Combinator (=:=): Equality

...combinator (=:=) ('equality' of terms) allows us to build simple predicates, e.g.:

Implementation of (=:=) by means of unify:

Intuitively: If the argument terms t and u can be unified wrt the input answer MkAnswer (s,n), the most general unifier is returned as the output answer; otherwise there is no answer.

Combinator (&&&): Conjunction

...combinator (&&&) allows us to connect predicates conjunctively, e.g.:

->> [{x=3,y=4}] run (var "x" =:= Int 3 &&& var "x" =:= Int 4) :: Stream Answer

->> []

Implementation of (&&&) by means of the bind operation
(>>=) of monad bunch:

(&&&) :: Bunch m => Pred m -> Pred m -> Pred m (p &&& q) s = p s >>= q

-- or equivalently using the do-notation: do t <- p s; u <- q t; return u</pre> Lecture 6

Detailed Outline

Chap. 16 16.1 16.2

```
16.2.2
16.2.3
16.2.4
16.2.5
```

lote

Combinator (|||): Disjunction

...combinator (|||) allows us to connect predicates disjunctively, e.g.:

Implementation of (|||) by means of the alt operation of monad bunch:

(|||) :: Bunch m => Pred m -> Pred m -> Pred m (p ||| q) s = alt (p s) (q s)

Lecture 6

Assigning Priorities to (=:=), (&&&), (|||)

Lecture 6

Outline

Chap. 15

hap. 10 6.1 6.2 16.2.1 16.2.2 16.2.3 16.2.4 16.2.5 16.2.6 16.2.7

16.2.8 16.2.9

16.4

NI . . .

... is done as follows:

infixr 4 =:=
infixr 3 &&&
infixr 2 |||

Combinator exists: Existential Quantificat.

...a combinator allowing the introduction of new variables in predicates (exploiting the Int component of answers).

Existential quantification: Introducing local variables in recursive predicates

exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
p (Var (Generated n)) (MkAnswer (s,n+1))

Note:

- The term exists (\x -> ...x...) has the same meaning as the predicate ...x... but with x denoting a fresh variable which is different from all the other variables used by the program; n+1 in MkAnswer (s,n+1) ensures that never the same variable is introduced by nested calls of exists.
- The function exists thus takes as its argument a function, which expects a term and produces a predicate; it invents a fresh variable and applies the given function to that variable.

Lecture 6

Detailed Outline

Chap. 15

Named vs. Generated Variables

...illustrating the difference:

- Example 1): The named variable y is set to the head of the list, which is the value of x. The value of the generated variable t is not output.
- Example 2): The same as above but now t denotes a named variable, whose value is output.

Cost Management of Recursive Predicates

...ensuring that in connection with the bunch type Matrix the costs per unfolding of the recursive predicate increase by 1 using wrap:

step :: Bunch m => Pred m -> Pred m
step p s = wrap (p s)

Illustrating the usage and effect of step:

```
run (var "x" =:= Int 0) :: Matrix Answer
->> MkMatrix [[{x=0}]] -- Without step: Just
-- the result.
```

Chapter 16.2.9 Writing Logic Programs: Two Examples

Writing Logic Programs: Two Examples

We consider two examples:

- 1. Concatenating lists: The predicate append.
- 2. Testing and constructing 'good' sequences: The predicate good.

Lecture 6

Detailed Outline

Chap. 15

Example 1: List Concatenation (1)

...implementing a predicate append (a,b,c), where a, b denote lists and c the concatenation of a and b.

The implementation of predicate append:

Lecture 6

Detailed Dutline Chap. 15 Chap. 16

16.2.3 16.2.4 16.2.5 16.2.6 16.2.7 16.2.8 16.2.9 16.3

Example 1: List Concatenation (2)

```
...in more detail:
append :: Bunch m => (Term, Term, Term) -> Pred m
append (p,q,r) =
-- Case 1
step (p =:= Nil &&& q =:= r
|||
-- Case 2
exists (\x -> exists (\a -> exists (\b ->
p =:= Cons x a &&& r =:= Cons x b &&& append (a,q,b)))))
ic20
ic21
ic22
ic21
ic22
ic21
ic23
ic24
ic23
ic24
ic25
ic24
ic25
ic26
ic27
```

Intuitively

- Case 1: If p is Nil, then r must be the same as q.
- Case 2: If p has the form Cons x a, then r must have the form Cons x b, where b is obtained by recursively concatenating a with the unchanged q.
- Termination: Is ensured since the third argument is getting smaller in each recursive call of append.

Example 1: List Concatenation (3)

...as common for logic programs, there is no difference between input and output variables. Hence, multiple usages of append are possible, e.g.:

a) Using append for concatenating two lists:

```
->> [{z=[1,2,3,4]}]
```

-- An appropriate implementation of show -- generating the above output is assumed.

-- More closely related to the internal structure -- of the value of z would be an output like:

->> Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))

Lecture 6

Example 1: List Concatenation (4)

Using append for computing the set of lists which equal a given list

b) ...when concatenated:

run (append (var "x", var "y", list [1,2,3])) :: Stream Answer \rightarrow [{x = Nil, y = [1,2,3]}, $\{x = [1], y = [2,3]\},\$ $\{x = [1,2], y = [3]\},\$ $\{x = [1,2,3], y = Nil\}\}$ c) ... when concatenated with another given list: run (append (var "x", list [2,3], list [1,2,3])) :: Stream Answer

$$\rightarrow$$
 [{x = [1]}]

Example 2: 'Good' Sequences (1)

...implementing a predicate good allowing to

- generate sequences of 0s and 1s, which are considered 'good.'
- check, if a sequence of 0s and 1s is 'good.'

We define:

- 1. The sequence [0] is good.
- 2. If the sequences s1 and s2 are good, then also the sequence [1] ++ s1 ++ s2.
- 3. There is no other good sequence except of those formed in accordance to the above two rules.

Example 2: 'Good' Sequences (2)

Examples:	
'Good' sequences	
	16.1
ΓΟ]	16.2
[0]	16.2.1
[1] + [0] + [0] = [100]	16.2.2
	16.2.3
[1] + + [0] + + [100] = [10100]	16.2.4
	16.2.5
[1] + + [100] + + [0] = [11000]	16.2.6
	16.2.7 16.2.8
[1] ++ [100] ++ [10100] = [110010100]	16.2.9
	16.3
	16.4
Pad' convences	Note
'Bad' sequences	
[1], [11], [110], [000], [010100], [1010101],	

Example 2: 'Good' Sequences (3)

Lemma 16.2.9.1 (Properties of 'Good' Sequences)

If a sequence *s* is good, then

1. the length of s is odd

2. s = [0] or there is a sequence t with s = [1] + t + [00]

Note: The converse implication of Lemma 16.2.9.1(2) does not hold: the sequence [11100] = [1]++[11]++[00], e.g., is bad.

Lecture 6

Detailed Outline

Chap. 15

Example 2: 'Good' Sequences (4)

Lecture 6

Outline

Chap. 15

16.1
16.2
16.2.1
16.2.2
16.2.3
16.2.4
16.2.5
16.2.6
16.2.7
16.2.8
16.2.9
16.3

16.4

```
The implementation of predicate good:
good :: Bunch m => Term -> Pred m
good (s) =
step (s =:= Cons (Int 0) Nil
||| exists (\t -> exists (\q -> exists (\r ->
s =:= Cons (Int 1) t
&&& append (q,r,t)
&&& good (q)
&&& good (r)))))
```

Example 2: 'Good' Sequences (5)

```
... in more detail:
 good :: Bunch m => Term -> Pred m
 good(s) =
  step (
  -- Case 1
   s =:= Cons (Int 0) Nil
    111
     -- Case 2
     exist (t \rightarrow exists (q \rightarrow exists (r \rightarrow
      s =:= Cons (Int 1) t
      &&& append (q,r,t) &&& good (q) &&& good (r)))))
                                                                         16.2.9
```

Intuitively

- Case 1: Checks if s is [0].
- Case 2: If s has the form [1]++t for some sequence t, all ways are checked of splitting t into two sequences q and r with q++r==t and q and r are good sequences themselves.
- Termination: Is ensured, since t gets smaller in every recursive call and the number of its splittings is finite.

Example 2: 'Good' Sequences (6) Using predicate good. 1) Checking if a sequence is good using Stream: run (good (list [1,0,1,1,0,0,1,0,0])) :: Stream Answer ->> [{}] -- Returning the empty set as answer, -- if the argument list is good. run (good (list [1,0,1,1,0,0,1,0,1])) :: Stream Answer ->> [] -- Returning no answer, if the argument -- list is bad.

Note: The "empty answer" and the "no answer" correspond to the answers "yes" and "no" of a Prolog system.

Example 2: 'Good' Sequences (7)

2a) Constructing good sequences using Stream:

```
run (good (var "s")) :: Stream Answer
->> [{s=[0]},
        {s=[1,0,0]},
        {s=[1,0,1,0,0]},
        {s=[1,0,1,0,1,0,0]},
        {s=[1,0,1,0,1,0,1]},...
```

...some answers will not be generated, since the depth search induced by Stream is not fair. The computation is thus likely to get stuck at some point.

Example 2: 'Good' Sequences (8)	
2b) Constructing good sequences using Diag:	
run (good (var "s")) :: Diag Answer	
\rightarrow Diag [{s=[0]},	
{s=[1,0,0]},	
$\{s=[1,0,1,0,0]\},\$	16.2 16.2.1
{s=[1,0,1,0,1,0,0]},	16.2.2
$\{s=[1,1,0,0,0]\},\$	16.2.4
{s=[1,0,1,0,1,0,1,0,0]},	16.2.6
{s=[1,1,0,0,1,0,0]},	16.2.8 16.2.9
$\{s=[1,0,1,1,0,0,0]\},\$	16.3
{s=[1,1,0,0,1,0,1,0,0]},	Note

...eventually all answers will be generated, since the diagonalization search induced by Diag is fair. However, the output order can hardly be predicted due to the interaction of diagonalization and shuffling. Example 2: 'Good' Sequences (9) 2c) Constructing good sequences using Matrix: run (good (var "s")) :: Matrix Answer ->> MkMatrix [[]. [{s=[0]}],[],[],[], [{s=[1,0,0]}],[],[],[]. $[{s=[1,0,1,0,0]}],[],$ $[{s=[1,1,0,0,0]}],[],$ $[{s=[1,0,1,0,1,0,0]}],[],$ 16.2.9 $[{s=[1,0,1,1,0,0,0]}, {s=[1,1,0,0,1,0,0]}], [],$using the cost-guided 'true' breadth search induced by

Matrix, the output order of results seems more 'predictable' than for the search induced by Diag. Additionally, we get 'progress notifications.'

Exercise 16.2.9.2: Adding Missing Code

Note, code for

- 1. pretty printing terms and answers
- making the types Term, Subst, and Answer instances of the type class Show
- is missing and must be provided before using the approach.

Add the missing code.

Lecture 6

Detailed Dutline

Chap. 15

16.1
 16.2
 16.2.1
 16.2.2
 16.2.3
 16.2.4
 16.2.5
 16.2.6
 16.2.7
 16.2.8
 16.2.9

16.4

Lecture 6

Detailed Dutline

Chap. 15

Chap. 16

6.1

16.3

.

NI . . .

Chapter 16.3 In Closing

In Closing

Current functional logic languages aim at balancing

- generality (in terms of paradigm integration).
- efficiency of implementations.

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language.
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency.

Overall: Functional logic programming is

- an emerging paradigm with appealing features.

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

16.2

16.3

10.4

lote

Lecture 6

Detailed Outline

Chap. 15

Chap. 16

6.1

- 2

.....

16.4

Note

Chapter 16.4 References, Further Reading

Chapter 16: Basic Reading

- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 177-199, 2003.
- Silvija Seres, Michael Spivey. Embedding Prolog in Haskell. In Proceedings of the 1999 Haskell Workshop (Haskell'99), Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, 25-38, 1999.
- Norbert Eisinger, Tim Geisler, Sven Panne. Logic Implemented Functionally. In Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97), Springer-V., LNCS 1292, 351-368, 1997.
 - Peter Pepper, Petra Hofstedt. *Funktionale Programmierung.* Springer-V., 2006. (Kapitel 22, Integration von Konzepten anderer Programmiersprachen)

Lecture 6

Outline Chap. 15 Chap. 16 16.1 16.2 16.3 16.4 Note

Chapter 16: Selected Further Reading (1)

- Hassan Ait-Kaci, Roger Nasr. Integrating Logic and Functional Programming. Lisp and Symbolic Computation 2(1):51-89, 1989.
- Sergio Antoy, Michael Hanus. Compiling Multi-Paradigm Declarative Languages into Prolog. In Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS 2000), Springer-V., LNCS 1794, 171-185, 2000.
- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM 53(4):74-85, 2010.
- Michael Hanus. Functional Logic Programming: From Theory to Curry. In Programming Logics – Essays in Memory of Harald Ganzinger. Springer-V., LNCS 7797, 123-168, 2013.

Lecture 6

Detailed Outline

Chap. 15

Chap. 16 16.1 16.2 16.3 16.4

Chapter 16: Selected Further Reading (2)

- Michael Hanus. Multi-paradigm Declarative Languages. In Proceedings of the 23rd International Conference on Logic Programming (ICLP 2007), Springer-V., LNCS 4670, 45-75, 2007.
- John W. Lloyd. *Programming in an Integrated Functional and Logic Language*. Journal of Functional and Logic Programming 1999(3), 49 pages, MIT Press, 1999.

Lecture 6

Detailed Outline

Chap. 15

Chap. 16 16.1 16.2

16.4

...for additional information and details refer to

full course notes

available at the homepage of the course at:

 Lecture 6

Detailed Dutline

Chap. 15

Chap. 16