Functional Programming

WS 2011/12

Harald Zankl (VO)
Thomas Bauereiß (PS)   Thomas Sternagel (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

week 9

Efficiency of Functional Programs

Avoid unnecessary recomputations by ...

- tupling

Introduce tail recursion by ...

- parameter accumulation

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, . . .

What is Parsing?

Parsing is the decomposition of a linear sequence into a structure, given by a grammar. This linear sequence may be a text in some natural language, a computer program, a web site, a piece of music, a sequence of genes, . . .

- linear sequence: `'t list` (list of tokens)
- structure: some user-defined type (abstract syntax tree)
- grammar: BNF (Backus-Naur form)

Note

- BNF can express context-free grammars (CFG)
- combinator parsers can parse context-sensitive grammars
- however, for the purpose of this lecture CFG suffice

Usual Two Phases - Lexing and Parsing

~~Lexing~~

- ~~divide original input (list of `char`s) into tokens~~
- ~~white space and comments may be dropped at this stage~~
- ~~syntactic check~~

Parsing

- work on list of tokens
- check if list of tokens corresponds to grammar
- produce an abstract syntax tree (AST)
- semantic check

Propositional Formulas

Grammar

$$\phi ::= p \mid (!\,\phi) \mid (\phi \,\&\, \phi)$$

Lexing

token	corresponds to
LPAR	(
RPAR	)
NOT	!
AND	&
ID of string	propositional atom

Example

$"(a\ \ \&(!b)\ )" \xrightarrow{lexing} [LPAR; ID"a"; AND; LPAR; NOT; ID"b"; RPAR; RPAR]$

$"(a\ \ a\ )" \xrightarrow{lexing} [LPAR; ID"a"; ID"a"; RPAR]$

Propositional Formulas (cont'd)

Grammar

$$\phi ::= p \mid (!\,\phi) \mid (\phi \,\&\, \phi)$$

Parsing (AST)

```
type t = Atom of string
       | And of t * t
       | Not of t
```

Example

$[LPAR; ID\text{"}a\text{"}; AND; LPAR; NOT; ID\text{"}b\text{"}; RPAR; RPAR] \xrightarrow{parsing}$
$And(Atom\text{"}a\text{"}, Not(Atom\text{"}b\text{"}))$
$[LPAR; ID\text{"}a\text{"}; ID\text{"}a\text{"}; RPAR] \xrightarrow{parsing} \times$

Parsers

First Attempt

- functions of type `'t list -> ('a * 't list)`
- e.g., `digit ['1';'2']` results in `('1',['2'])`
- but what about errors, i.e, `digit ['x';'y'] =` ?

Type of parsers

```
type ('a,'t)t = 't list -> ('a * 't list)option
```

- a parser works on a list of tokens of arbitrary type `'t`
- a successful parse yields `Some(x,ts)` with result `x` and remaining tokens `ts`
- a parse error is represented by a result of `None` (i.e., we do not get any further information about the error)

Preparation

Applying a Parser

```
let test p s = match p(Strng.of_string s) with
  | None       -> failwith "parse␣error"
  | Some(x,ts) -> (x,ts)
```

Example

- `letter` ... parses a single letter (see page 12)
  ```
  # test letter "hello␣world"
  - char * char list = ('h', "ello␣world")
  # test letter "1234"
  Exception: Failure "parse␣error".
  ```
- `digit` ... parses a single digit (see page 12)
  ```
  # test digit "1234"
  - char * char list = ('1', "234")
  ```

Preparation (cont'd)

Type of a Parser

- type of parser: `'t list -> ('a * 't list)`
- often we want to process single tokens only, i.e., `'t -> 'a`

Operating on a Single Token

```
let token f = function
  | []    -> None
  | t::ts -> match f t with
    | Some x -> Some(x,ts)
    | None   -> None
```

Primitive Parsers

Any Token Satisfying a Condition

```
let sat p ts = token(fun t -> if p t then Some t else None) ts
```

Checking for End of Input

```
let eoi = function [] -> Some((),[])
                 | _  -> None
```

Example

```
# test (sat ((=) 'h')) "hello␣world"
- char * char list = ('h', "ello␣world")
# test (sat ((=) 'e')) "hello␣world"
Exception Failure "parse␣error".
# test eoi ""
- : unit * char list = ((), "")
```

Character Parsers

Reading a Given Character

```
let any ts = sat (fun _ -> true) ts

let char c ts = sat ((=)c) ts
```

Reading Letters and Digits

```
let letter =
 sat(fun c -> ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z'))

let digit = sat(fun c -> '0' <= c && c <= '9')
```

Character Parsers (cont'd)

Choosing From a List of Tokens

```
let oneof s = sat(fun c -> Lst.mem c (Strng.of_string s))
```

Every Token Except a Given List

```
let noneof s = sat(fun c -> not(Lst.mem c (Strng.of_string s)))
```

White Space

```
let space = oneof "␣\n\r\t"
```

Parser Combinators

Choice

```
('a,'t)t -> ('a,'t)t -> ('a,'t)t
```

```
let (<|>) p q ts = match p ts with None          -> q ts
                                   | Some _ as r -> r
```

- choice takes two parsers `p` and `q` of same result type
- if `p` is successful on `ts` then its result is returned
- otherwise `q` is applied on `ts`

Example

```
# test (letter <|> digit) "hello␣world"
- : char * char list = ('h', "hello␣world")
# test (letter <|> digit) "1234"
- : char * char list = ('1', "234")
```

Parser Combinators (cont'd)

Bind - Binding a Parser to the Result of Another

`('a,'t)t -> ('a -> ('b,'t)t) -> ('b,'t)t`

```
let (>>=) p f ts = match p ts with None       -> None
                                 | Some(x,ts) -> f x ts
```

- bind takes two arguments
- first a parser `p` with results of type `'a`
- then a function `f` taking an `'a` and producing a parser with results of type `'b`
- `p >>= f` executes `p` and then feeds the function `f` with its result
- since `f x` is a parser, the result of `p >>= f` is a parser

Parser Combinators (cont'd)

Then - Sequential Composition of Parsers

```
('a,'t)t -> ('b,'t)t -> ('b,'t)t
```

```
let (>>) p q = p >>= (fun _ -> q)
```

- then takes two parsers `p` and `q`
- first a parser `p` with results of type `'a`
- then a parser `q` with result type `'b`
- `p >> q` first executes `p` and then executes `q`
- the result of `p` is ignored, the result of `q` is returned

Example

```
# test (space >> space) "␣␣␣hello␣world"
- : char * char list = (' ', "␣hello␣world")
```

Turning Values Into Parsers

Return

```
'a -> ('a,'t)t
```

```
let return x = fun ts -> Some(x,ts)
```

- `return x` takes the value `x` and yields a parser that returns `x` without consuming any input

Example

```
# test (return 3) "hello␣world"
- : int * char list = (3, "hello␣world")
```

Counting Spaces

First try

```
let rec count_spaces =
 (space >>
  count_spaces >>= fun i -> return (i+1))
 <|>
 (any >>
  count_spaces >>= fun i -> return i)
 <|>
 (eoi >> return 0)
```

Problem

- OCaml interpreter detects that `count_spaces` would not terminate (eager evaluation)
- Error: This kind of expression is not allowed as right-hand ...

Counting Spaces (cont'd)

Second try (dummy argument)

```
let rec count_spaces () =
 (space >>
  count_spaces () >>= fun i -> return (i+1))
 <|>
 (any >>
  count_spaces () >>= fun i -> return i)
 <|>
 (eoi >> return 0)
```

Problem

- OCaml interpreter doesn't detect nontermination (eager evaluation)
- `# test (count_spaces ()) "hello␣world"`
  Stack overflow during evaluation (looping recursion?).

Counting Spaces (cont'd)

Third try (enforce evaluation)

```
let rec count_spaces () =
  (space >>= fun _ ->
   count_spaces () >>= fun i -> return (i+1))
  <|>
  (any >>= fun _ ->
   count_spaces () >>= fun i -> return i)
  <|>
  (eoi >> return 0)
```

Success

```
# test (count_spaces ()) "hello␣world"
- : int * char list = (1, "")
# test (count_spaces ()) "␣␣␣hello␣world"
- : int * char list = (4, "")
```

Parser Combinators (cont'd)

Many

```
let rec many p = (
  p      >>= fun x ->
  many p >>= fun xs ->
  return(x::xs)
) <|> return []
```

- many p applies p zero or more times
- result is list of results of p
- greedy (as many applications of p as possible)

Example

```
# test (many space) "␣␣␣hello␣world"
- : char list * char list = ("␣␣␣", "hello␣world")
```

Arithmetic

A Grammar for Arithmetic

$$e ::= e + t \mid t \qquad t ::= t * f \mid f \qquad f ::= (e) \mid n$$

$$n ::= d\ n \mid d \qquad d ::= 0 \mid \ldots \mid 9$$

Type of the AST

```
type arith = Num of Strng.t
           | Add of arith * arith
           | Mul of arith * arith
```

Example

```
# test ??? "3+2*5"
- : arith * char list = (Add (Num "3", Mul (Num "2", Num "5")), "")
```

A Parser for Arithmetic (does not terminate)

```
let rec e() =
 (e() >>= fun e1 -> char '+' >> t() >>= fun e2 ->
  return(Add(e1,e2)))
 <|> (t())
and t() =
 (t() >>= fun t1 -> char '*' >> f() >>= fun t2 ->
 return(Mul(t1,t2)))
 <|> (f())
and f() = (
  char '(' >>= fun _ ->
  e()      >>= fun e1 ->
  char ')' >>
  return e1
) <|> n
and n = many1 digit >>= fun r -> return(Num r)
```

Problem & Solution

- problem: left recursive grammar
- solution: eliminate left recursion

Non-left recursive Grammar for Arithmetic

$$
\begin{aligned}
e &::= t\ e' \\
e' &::= +\ t\ e' \mid \epsilon \\
t &::= f\ t' \\
t' &::= *\ f\ t' \mid \epsilon \\
f &::= (\ e\ ) \mid n \\
n &::= d\ n \mid d \\
d &::= 0 \mid \ldots \mid 9
\end{aligned}
$$

A Parser for Arithmetic

```
let rec e() = t() >>= e'
and e' term = (char '+' >>= fun _ -> t() >>= e' >>= fun t2 ->
  return(Add(term,t2))
) <|> return term
and t() = f() >>= t'
and t' fact = (char '*' >>= fun _ -> f() >>= t' >>= fun f2 ->
 return(Mul(fact,f2))
) <|> return fact
and f() = (char '(' >>= fun _ -> e() >>= fun e1 -> char ')' >>
 return e1
) <|> n
and n = many1 digit >>= fun r -> return(Num r)
```