Functional Programming

WS 2009/10

Christian Sternagel (VO)
Friedrich Neurauter (PS) Sarah Winkler (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

4 December 2009

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. The linear sequence may be a text in some natural language, a computer program, a web site, a piece of music, a sequence of genes, . . .

In the Following . . .

Use

- linear sequence: `'t list` (list of tokens)
- structure: some user-defined type
- 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

The Programming Language $\mathcal{PL}$

Grammar

$$
\begin{array}{rcl}
\langle exp \rangle & \stackrel{\text{def}}{=} & \langle call \rangle \mid \langle const \rangle \mid \langle def \rangle \mid (\langle exp \rangle) \\
& \mid & \textbf{if } \langle exp \rangle \textbf{ then } \langle exp \rangle \textbf{ else } \langle exp \rangle \\
\\
\langle call \rangle & \stackrel{\text{def}}{=} & x \mid f(\langle exp \rangle, \ldots, \langle exp \rangle) \\
\\
\langle def \rangle & \stackrel{\text{def}}{=} & f(x_1, \ldots, x_n) = \langle exp \rangle \\
& & \langle exp \rangle \\
& \mid & x = \langle exp \rangle \\
& & \langle exp \rangle \\
\\
\langle const \rangle & \stackrel{\text{def}}{=} & -^{?}(0 \mid \cdots \mid 9)^{+}
\end{array}
$$

Computing Fibonacci Numbers in $\mathcal{PL}$

```
# We assume that we may use the two functions
# -(x,y) for subtracting y from x, as well as
# leq(x,y) for checking whether x is less
# than or equal to y.
p(x) = -(x,1)

fib(x) = if leq(x,1)
  then x
  else +(fib(p(x)),fib(p(p(x))))

fib(10)
```

fib.txt

Parsers

First Attempt

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

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)

Usual Two Phases - Lexing and Parsing

Lexing

- divide original input (list of `chars`) into tokens
- white space and comments may be dropped at this stage

Parsing

- work on list of tokens
- produce an abstract syntax tree (AST)

Tokens and ASTs of $\mathcal{Pl}$

Tokens

```
type token = Comma | Comment | Lparen | Rparen
  | Keyword of Strng.t
  | Number of Strng.t
  | Ident of Strng.t
```

ASTs

```
type sym = Strng.t

type ast =
  | Const of Strng.t
  | Ite of (ast * ast * ast)
  | Call of (sym * ast list)
  | Def of (sym * sym list * ast * ast)
```

Primitive Parsers

Checking for End of Input

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

Reading a Single Token

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

Parsers

Any Token Satisfying a Condition

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

Character Parsers

Reading a Given Character

```
let any_char i = sat (fun _ -> true) i

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

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"
```

Turning Values Into Parsers

Return

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

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

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

Parser Combinators

Bind - Binding a Parser to the Result of Another

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

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

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

Parser Combinators (cont'd)

Choice

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

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

Example

- $p ::= \mathtt{a} \mid \mathtt{b}$
- `let p = char 'a' <|> char 'b'`
- i.e., `(<|>)` corresponds to $\mid$ in BNF

Parser Combinators (cont'd)

Optional Input With Default Result

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

```
let (?>) p d = p <|> return d
```

- `?> p d` tries to apply parser `p`
- if possible the result of `p` is returned
- otherwise the default value `d`

Parser Combinators (cont'd)

Many

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

Example

- $p ::= \epsilon \mid \mathtt{a}\ p$
- `let p = many(char 'a')`

Running a Parser on Some Input

Parse

```
let parse p input = match p input with None       -> None
                                     | Some(x,_) -> Some x
```

Test

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

Lexing $\mathcal{Pl}$

Comments

```
let comment =
  char '#'            >>= fun _ ->
  many(noneof "\n") >>= fun _ ->
  return Comment
```

Integer Literals

```
let number =
  ?> (oneof "+-") '+' >>= fun sign ->
  many1 digit         >>= fun ds   ->
  return(Number(sign::ds))
```

Lexing $\mathcal{Pl}$ (cont'd)

Identifiers and Keywords

```
let keywords =
  Lst.map Strng.of_string ["if";"then";"else";"="]

let ident =
  many1(noneof "(),␣\n\r\t") >>= fun cs ->
  if Lst.mem cs keywords then return(Keyword cs)
                         else return(Ident cs)
```

Lexing $\mathcal{PL}$ (cont'd)

Delimiters

```
let lparen = char '(' >>= fun _ -> return Lparen

let rparen = char ')' >>= fun _ -> return Rparen

let comma  = char ',' >>= fun _ -> return Comma
```

Removing Trailing White Space

```
let lex p =
  p       >>= fun x ->
  spaces >>= fun _ ->
  return x
```

Lexing $\mathcal{Pl}$ (cont'd)

Reading a Single Token

```
let token = lex(
  comment <|> number <|> ident <|>
  lparen  <|> rparen <|> comma
)
```

Split All of the Input Into Tokens

```
let tokenize =
  spaces     >>= fun _  ->
  many token >>= fun ts ->
  eoi        >>= fun _  ->
  return ts
```

Example - Lexing `fib.txt`

```
[Comment; Comment; Comment; Comment; Ident "p"; Lparen;
 Ident "x"; Rparen; Keyword "="; Ident "-"; Lparen;
 Ident "x"; Comma; Number "+1"; Rparen; Ident "fib";
 Lparen; Ident "x"; Rparen; Keyword "=";
 Keyword "if"; Ident "leq"; Lparen; Ident "x"; Comma;
 Number "+1"; Rparen; Keyword "then"; Ident "x";
 Keyword "else"; Ident "+"; Lparen; Ident "fib"; Lparen;
 Ident "p"; Lparen; Ident "x"; Rparen; Rparen; Comma;
 Ident "fib"; Lparen; Ident "p"; Lparen;
 Ident "p"; Lparen; Ident "x"; Rparen; Rparen; Rparen;
 Rparen; Ident "fib"; Lparen; Number "+10"; Rparen]
```

Parsing $\mathcal{Pl}$

Functions for Recognizing Tokens

```
let kwd s  = token(function
  | Keyword k when k = Strng.of_string s -> Some k
  | _ -> None)

let number = token(function
  | Number n -> Some n
  | _ -> None)

let ident  = token(function Ident i -> Some i | _ -> None)

let lparen = token(function Lparen -> Some() | _ -> None)

let rparen = token(function Rparen -> Some() | _ -> None)

let comma  = token(function Comma -> Some() | _ -> None)
```

Parsing $\mathcal{Pl}$ (cont'd)

Parsing Integer Literals

```
let const = number >>= fun n -> return(Const n)
```

2 Auxiliary Functions

```
let par p = between lparen p rparen
```

```
let parm p = between lparen (sep_by comma p) rparen
```

Parsing $\mathcal{Pl}$ (cont'd)

Definitions

```
let rec def() =
  ident                  >>= fun f  ->
  ?> (parm ident) []     >>= fun xs ->
  kwd "="                >>= fun _  ->
  exp()                  >>= fun e1 ->
  exp()                  >>= fun e2 ->
  return(Def(f,xs,e1,e2))
```

Parsing $\mathcal{Pl}$ (cont'd)

Function Calls

```
and call() =
  ident                  >>= fun f  ->
  ?> (parm(exp())) []    >>= fun es ->
  return(Call(f,es))
```

Parsing $\mathcal{Pl}$ (cont'd)

Conditional Expressions

```
and ite() =
  kwd "if"   >>= fun _ ->
  exp()      >>= fun b ->
  kwd "then" >>= fun _ ->
  exp()      >>= fun t ->
  kwd "else" >>= fun _ ->
  exp()      >>= fun e ->
  return(Ite(b,t,e))
```

Parsing $\mathcal{Pl}$ (cont'd)

Arbitrary Expressions

```
and exp() =
  def() <|> ite() <|> call() <|> const <|> (
    lparen >>= fun _ ->
    exp()  >>= fun e ->
    rparen >>= fun _ ->
    return e
)
```

Parsing $\mathcal{Pl}$ (cont'd)

The Abstract Syntax Tree of $\mathcal{Pl}$

```
let ast =
  exp() >>= fun e ->
  eoi   >>= fun _ ->
  return e
```

Example - Parsing `fib.txt`

```
(Def("p",["x"],Call("-",[Call("x",[]);Const "1"]),
 Def("fib",["x"],
   Ite(Call("leq",[Call("x",[]);Const "1"]),
     Call("x",[]),
     Call("+",
       [Call("fib",[Call("p",[Call("x",[])])]);
        Call("fib",[Call("p",[Call("p",[Call("x",[])])])])])))),

Call("fib",[Const "+10"])))
```