YSC4230: Programming Language Design and Implementation

Week 8: First-Class Functions

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Recap: Parsing in OCaml via Menhir

Practical Issues

- https://github.com/ysc4230/week-07-more-parsing

- Dealing with source file location information
  - In the lexer and parser
  - In the abstract syntax

  - See range.ml, ast.ml
  - Check the parse tree (printing via driver.ml)

- Lexing comments / strings

Menhir output

- You can get verbose parser debugging information by doing:
  - `menhir --explain …`
  - or, if using ocamlbuild:
    `ocamlbuild –use-menhir -yaccflag -–explain …`
- The result is a <parsername>.conflicts file that contains a description of the error
  - The parser items of each state use the '.' just as described above
- The flag --dump generates a full description of the automaton
- Example: see start_parser.mly

Shift/Reduce conflicts

- Conflict 1:
  - Operator precedence
- Conflict 2:
  - Parsing if-then-else statements

Shift/Reduce conflicts

- Conflict 1:
  - Operator precedence (State 13)
  - Resolving by changing the grammar (see good_parser.ml)

- Conflict 2:
  - Parsing if-then-else statements

5.3 Inlining

It is well-known that the following grammar of arithmetic expressions does not work as expected: that is, in spite of the priority declarations, it has shift/reduce conflicts.

```
%token < int > INT
%token PLUS TIMES
%left PLUS
%left TIMES

%%

expression:
    | i = INT { i }
    | e = expression; o = op; f = expression { o e f }
op:
    | PLUS { ( + ) }
    | TIMES { ( * ) }
```

The trouble is, the precedence level of the production *expression* $\rightarrow$ *expression op expression* is undefined, and there is no sensible way of defining it via a **%prec** declaration, since the desired level really depends upon the symbol that was recognized by *op*: was it *PLUS* or *TIMES*?

From Menhir Manual

http://gallium.inria.fr/~fpottier/menhir/manual.pdf

The standard workaround is to abandon the definition of *op* as a separate nonterminal symbol, and to inline its definition into the definition of *expression*, like this:

$$
\begin{array}{l}
\textit{expression}: \\
\quad | \quad i = \textit{INT} \; \{\, i \,\} \\
\quad | \quad e = \textit{expression};\, \textit{PLUS};\, f = \textit{expression} \; \{\, e + f \,\} \\
\quad | \quad e = \textit{expression};\, \textit{TIMES};\, f = \textit{expression} \; \{\, e * f \,\}
\end{array}
$$

This avoids the shift/reduce conflict, but gives up some of the original specification's structure, which, in realistic situations, can be damageable. Fortunately, Menhir offers a way of avoiding the conflict without manually transforming the grammar, by declaring that the nonterminal symbol *op* should be inlined:

$$
\begin{array}{l}
\textit{expression}: \\
\quad | \quad i = \textit{INT} \; \{\, i \,\} \\
\quad | \quad e = \textit{expression};\, o = \textit{op};\, f = \textit{expression} \; \{\, o\; e\; f \,\} \\
\textbf{\%inline}\ \textit{op}: \\
\quad | \quad \textit{PLUS} \; \{\, (+) \,\} \\
\quad | \quad \textit{TIMES} \; \{\, (*) \,\}
\end{array}
$$

The **%inline** keyword causes all references to *op* to be replaced with its definition. In this example, the definition of *op* involves two productions, one that develops to *PLUS* and one that expands to *TIMES*, so every production that refers to *op* is effectively turned into two productions, one that refers to *PLUS* and one that refers to *TIMES*. After inlining, *op* disappears and *expression* has three productions: that is, the result of inlining is exactly the manual workaround shown above.

Precedence and Associativity Declarations

- Parser generators, like menhir often support precedence and associativity declarations.
  - Hints to the parser about how to resolve conflicts.
  - See: good-parser.mly

- Pros:
  - Avoids having to manually resolve those ambiguities by manually introducing extra nonterminals (see parser.mly)
  - Easier to maintain the grammar

- Cons:
  - Can't as easily re-use the same terminal (if associativity differs)
  - Introduces another level of debugging

- Limits:
  - Not always easy to disambiguate the grammar based on just precedence and associativity.

Conflict 2: Ambiguity in Real Languages

- Consider this grammar:

  S $\longmapsto$ `if (E) S`
  S $\longmapsto$ `if (E)` S `else` S
  S $\longmapsto$ X = E
  E $\longmapsto$ …

- Is this grammar OK?

- Consider how to parse:

  `if (`E_1`) if (`E_2`)` S_1 `else` S_2

- This is known as the "dangling else" problem.

- What should the "right" answer be?

- How do we change the grammar?

How to Disambiguate if-then-else

- Want to rule out:

$$\texttt{if } (E_1) \left\{ \texttt{if } (E_2)\ S_1 \right\} \texttt{ else } S_2$$

- Observation: An un-matched ‘`if`’ should not appear as the ‘`then`’ clause of a containing ‘`if`’.

$S \longmapsto M \mid U$ // M = “matched”, U = “unmatched”

$U \longmapsto$ `if (E)` S // Unmatched ‘if’

$U \longmapsto$ `if (E)` M `else` U // Nested if is matched

$M \longmapsto$ `if (E)` M `else` M // Matched ‘if’

$M \longmapsto X = E$ // Other statements

- See: else-resolved-parser.mly

Alternative: Use { }

- Ambiguity arises because the ‘then’ branch is not well bracketed:

  $\texttt{if (E}_1\texttt{) \{ if (E}_2\texttt{) \{ S}_1\texttt{ \} \} else S}_2$ // unambiguous

  $\texttt{if (E}_1\texttt{) \{ if (E}_2\texttt{) \{ S}_1\texttt{ \} else S}_2\texttt{ \}}$ // unambiguous

- So: could just require brackets
  - But requiring them for the else clause too leads to ugly code for chained if-statements:

```
if (c1) {
  …
} else {
  if (c2) {

  } else {
    if (c3) {

    } else {

    }
  }
}
```

How about a compromise? Allow unbracketed else block only if the body is ‘if’:

```
if (c1) {

} else if (c2) {

} else if (c3) {

} else {

}
```

Benefits:

- Less ambiguous
- Easy to parse
- Enforces good style

HW4: Oat v.1

Oat

- Simple C-like Imperative Language
  - supports 64-bit integers, arrays, strings
  - top-level, mutually recursive procedures
  - scoped local, imperative variables

- See examples in *hw4programs* folder

- How to design/specify such a language?

Oat v.1 Language Specification

YSC3208: Programming Language Design and Implementation

1 Grammar

The following grammar defines the Oat syntax. All binary operations are *left associative* with precedence levels indicated numerically. Higher precedence operators bind tighter than lower precedence ones.

prog	::=		prog
	|	$decl_1 .. decl_i$	
decl	::=		global declarations
	|	*gdecl*	
	|	*fdecl*	

Oat Design Considerations

- Resolving parsing errors
- Compiling non-static arrays to LLVMlite

First-Class Functions

Untyped lambda calculus

Substitution

Evaluation

“Functional” languages

- Languages like OCaml, Scala, Haskell, Scheme, Python, C#, Java 8, Swift
- Functions can be passed as arguments (e.g. map or fold)
- Functions can be returned as values (e.g. compose)
- Functions nest: inner function can refer to variables bound in the outer function

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

- How do we implement such functions?
  - in an interpreter? in a compiled language?

(Untyped) Lambda Calculus

- The **lambda calculus** is a minimal programming language.
  - Note: we're writing (fun x -> e) lambda-calculus notation: λ x. e
- It has **variables**, **functions**, and **function application**.
  - That's it!
  - It's Turing Complete.
  - It's the foundation for a *lot* of research in programming languages.
  - Basis for "functional" languages like Scala, OCaml, Haskell, etc.

Abstract syntax in OCaml:

```
type exp =
 | Var of var            (* variables              *)
 | Fun of var * exp   (* functions: fun x → e  *)
 | App of exp * exp  (* function application  *)
```

Concrete syntax:

```
exp ::=
     | x                   variables
     | fun x → exp         functions
     | exp1 exp2           function application
     | ( exp )             parentheses
```

Free Variables and Scoping

```
let add = fun x → fun y → x + y
let inc = add 1
```

- The result of `add 1` is a function
- After calling `add`, we can't throw away its argument (or its local variables) because those are needed in the function returned by add.
- We say that the variable `x` is *free* in `fun y → x + y`
  - Free variables are defined in an outer scope

- We say that the variable `y` is *bound* by "`fun y`" and its scope is the body "`x + y`" in the expression `fun y → x + y`

- A term with no free variables is called *closed*.

- A term with one or more free variables is called *open*.

Values and Substitution

- The only values of the lambda calculus are (closed) functions:

```
val ::=
    | fun x → exp          functions are values
```

- To *substitute* a (closed) value v for some variable x in an expression e
  - Replace all *free occurrences* of x in e by v.
  - In OCaml: written subst v x e
  - In Math: written e{v/x}

- Function application is interpreted by *substitution*:

```
  (fun x → fun y → x + y) 1
= subst 1 x (fun y → x + y)
= (fun y → 1 + y)
```

Note: for the sake of examples we may add integers and arithmetic operations to the "pure" untyped lambda calculus.

Operational Semantics of Lambda Calculus

- Substitution function (in Math):

$x\{v/x\}$	$= v$	*(replace the free x by v)*
$y\{v/x\}$	$= y$	*(assuming $y \neq x$)*
(fun x → exp){v/x}	= (fun x → exp)	*(x is bound in exp)*
(fun y → exp){v/x}	= (fun y → exp{v/x})	*(assuming $y \neq x$)*
$(e_1\ e_2)\{v/x\}$	$= (e_1\{v/x\}\ e_2\{v/x\})$	*(substitute everywhere)*

- Examples:

(x y) {(fun z → z z)/y}

= x (fun z → z z)

(fun x → x y){(fun z → z z)/y}

= fun x → x (fun z → z z)

(fun x → x){(fun z → z z)/x}

= fun x → x // x is not free!

Demo: Programming in Lambda Calculus

- https://github.com/ysc4230/week-08-lambda-2021

- lambda.ml – untyped lambda-calculus
- lambda_int.ml – untyped lambda-calculus with integers
- stlc.ml – simply-typed lambda-calculus

Free Variable Calculation

- An OCaml function to calculate the set of free variables in a lambda expression:

```
let rec free_vars (e:exp) : VarSet.t =
  begin match e with
    | Var x         -> VarSet.singleton x
    | Fun(x, body) -> VarSet.remove x (free_vars body)
    | App(e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
  end
```

- A lambda expression `e` is *closed* if `free_vars e` returns `VarSet.empty`
- In mathematical notation:

$$\mathrm{fv}(x) = \{x\}$$

$$\mathrm{fv}(\textbf{fun}\ x \rightarrow exp) = \mathrm{fv}(exp) \setminus \{x\}$$ *('x' is a bound in exp)*

$$\mathrm{fv}(exp_1\ exp_2) = \mathrm{fv}(exp_1) \cup \mathrm{fv}(exp_2)$$

Variable Capture

- Note that if we try to naively "substitute" an open term, a bound variable might ***capture*** the free variables:

Note: x is *free* in (fun z → x)

(fun x → (x y)){(fun z → x)/y}
= fun x → (x (fun z -> x))

free x is *"captured"!!*

- Usually *not* the desired behaviour
  - This property is sometimes called "dynamic scoping"
    The meaning of "x" is determined by where it is bound dynamically, not where it is bound statically.
  - Some languages (e.g. emacs lisp) are implemented with this as a "feature"
  - But: it leads to hard-to-debug scoping issues

Alpha Equivalence

- Note that the names of bound variables don't matter to the semantics
  - i.e. it doesn't matter which variable names you use, as long as you use them consistently:

(fun x → y x) is the "same" as (fun z → y z)

the choice of "x" or "z" is arbitrary, so long as we consistently rename them

Two terms that differ only by consistent renaming of *bound* variables are called *alpha equivalent*

- The names of *free* variables **do** matter:

(fun x → y x) is *not* the "same" as (fun x → z x)

Intuitively: y an z can refer to different things from some outer scope

Students who cheat by "renaming variables" are trying to exploit alpha equivalence…

Fixing Substitution

- Consider the substitution operation:

$$e_1\{e_2/x\}$$

- To avoid capture, we define substitution to pick an alpha equivalent version of e_1 such that the bound names of e_1 don't mention the free names of e_2.
  - Then do the "naïve" substitution.

For example: (fun x → (x y)){(fun z → x)/y}

= (fun x’ → (x' (fun z → x))

rename x to x'

This is fine:

(fun x → (x y)){(fun x → x)/y}

= (fun x → (x (fun x → x))

= (fun a → (a (fun b → b))

Operational Semantics

- Specified using just two inference rules with judgments of the form $exp \Downarrow val$
  - Read this notation a as "program exp evaluates to value val"
  - This is *call-by-value* semantics: function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \rightarrow exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

Demo: Implementing the Interpreter

- https://github.com/ysc4230/week-08-lambda-2021

- **lambda.ml – untyped lambda-calculus**
- lambda_int.ml – untyped lambda-calculus with integers
- stlc.ml – simply-typed lambda-calculus

Adding Integers to Lambda Calculus

exp ::=	
	…	
	n	*constant integers*
	$exp_1 + exp_2$	*binary arithmetic operation*
val ::=	
	fun x → exp	*functions are values*
	n	*integers are values*
$n\{v/x\} = n$	*constants have no free vars.*
$(e_1 + e_2)\{v/x\} = (e_1\{v/x\} + e_2\{v/x\})$	*substitute everywhere*

$$\frac{exp_1 \Downarrow n_1 \quad exp2 \Downarrow n_2}{exp_1 + exp_2 \Downarrow (n1 [\![+]\!] n2)}$$

Object-level '+'

Meta-level '+'

Next Week

Semantic Analysis via Types