Lecture 5. Funclang Implementation

September 27, 2018

Currying

the term Currying is from Haskell Curry

Model multiple argument lambda abstractions as a combination of single argument lambda abstraction

```
(define plus
   (lambda (x y) (+ x y)))

(define plusCurry
   (lambda (x)
    (lambda (y)
      (+ x y)
    )
   )
)
```

Recursive Function

- Recursive function mirror the definition of the input data type

$$List := (list) \mid (cons\ \texttt{val}\ List),\ \texttt{where val} \in \texttt{Value}$$

```
(define append
  (lambda (lst1 lst2)
    (if (null? lst1) lst2
      (if (null? lst2) lst1
        (cons (car lst1) (append (cdr lst1) lst2))
      )
    )
  )
)
```

How to Extend the Semantics for the Grammar?

- Any new types of values to be added?
- Semantic rules?
- How to implement it?

Grammar for FuncLang

Program	::=	DefineDecl* Exp?	*Program*
DefineDecl	::=	(define Identifier Exp)	*Define*
Exp	::=		*Expressions*
		Number	*NumExp*
	|	(+ Exp Exp+)	*AddExp*
	|	(- Exp Exp+)	*SubExp*
	|	(* Exp Exp+)	*MultExp*
	|	(/ Exp Exp+)	*DivExp*
	|	Identifier	*VarExp*
	|	(let ((Identifier Exp)+) Exp)	*LetExp*
	|	(Exp Exp+)	***CallExp*** ★
	|	(lambda (Identifier+) Exp)	***LambdaExp*** ★
Number	::=	Digit	*Number*
	|	DigitNotZero Digit+	
Digit	::=	[0-9]	*Digits*
DigitNotZero	::=	[1-9]	*Non-zero Digits*
Identifier	::=	Letter LetterOrDigit*	*Identifier*
Letter	::=	[a-zA-Z$_]	*Letter*
LetterOrDigit	::=	[a-zA-Z0-9$_]	*LetterOrDigit*

Value of a Lambda Expression

- Lambda expression is function, it has values, and can be passed as parameters, return from a function and stored in the environment

like other values

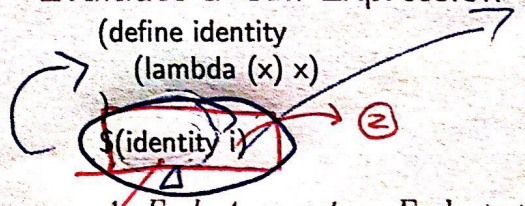
```
Value    ::=
             NumVal                                   Values
           | FunVal                                   Numeric Values
NumVal   ::= (NumVal n)                             Function Values
FunVal   ::= (FunVal var_0,..., var_n e env)            NumVal
             where var_0,..., var_n ∈ Identifier,       FunVal
             e ∈ Exp, env ∈ Env
```

Function value

Value of a Lambda Expression

$$\frac{\textsc{Value of LambdaExp}\quad \texttt{(FunVal var}_i\texttt{,for i = 0...k exp}_b\texttt{ env) = v}}{\texttt{value (LambdaExp var}_i\texttt{,for i = 0...k exp}_b\texttt{) env = v}}$$

Evaluate a Call Expression



(i+j) foo

1. *Evaluate operator.* Evaluate the expression whose value will be the function value. For example, for the call expression `(identity i)` the variable expression `identity`'s value will be the function value.

What is its func-value

2. *Evaluate operands.* For each expression that is in place of a formal parameter, evaluate it to a value. For example, for the call expression `(identity i)` the variable expression `i`'s value will be the only operand value.

3. *Evaluate function body.* This step has three parts.

   a) Find the expression that is the body of the function value,

   b) create a suitable environment for that body to evaluate, and

   c) evaluate the body.

Function: operations

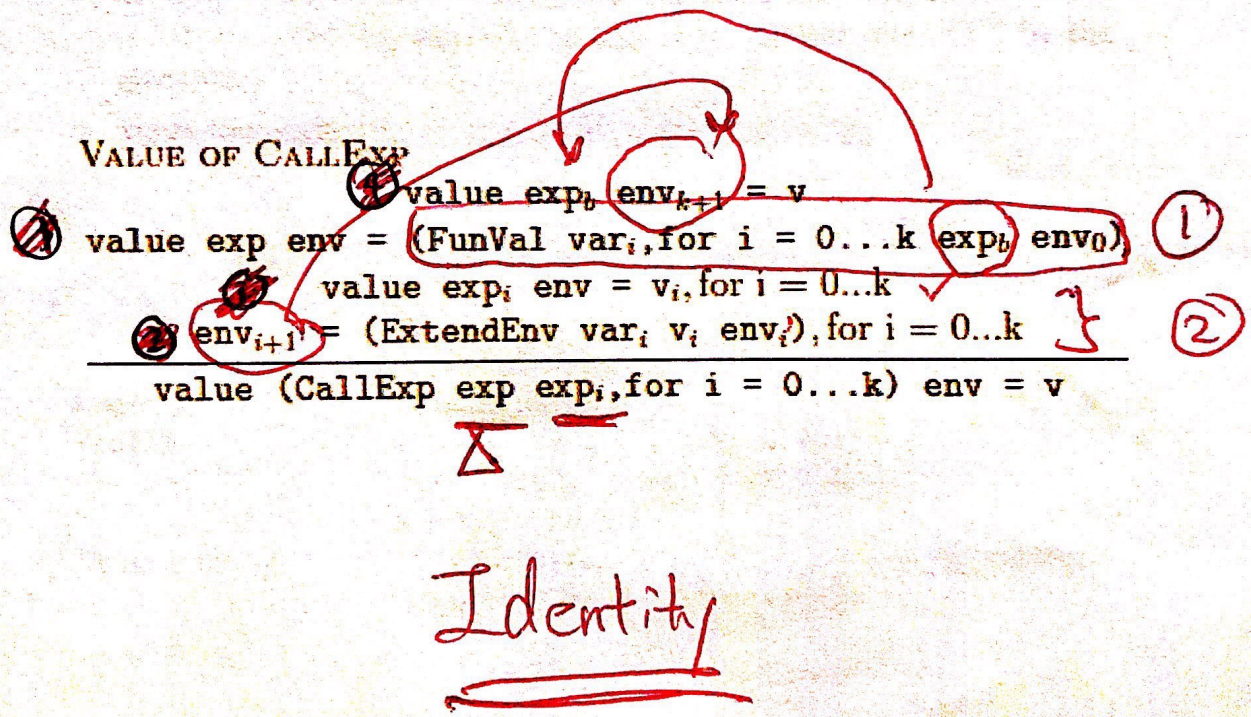
'+'

'-'

special case

Math

Value of a Call Expression



Dynamic Errors

- number of formal parameters and actual parameters do not match (context-sensitivity part of the language, cannot been find by the grammar)
- if exp (operator) does not return a function value

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
	|	`FunVal`	*Function Values*
	|	`DynamicError`	*Dynamic Error*
`NumVal`	::=	`(NumVal n)`	*NumVal*
`FunVal`	::=	`(FunVal` $\mathtt{var}_0, .., \mathtt{var}_n$ `e env)`	*FunVal*
		where $\mathtt{var}_0, .., \mathtt{var}_n \in$ Identifier,	
		$\mathtt{e} \in$ Exp, $\mathtt{env} \in$ Env	
`DynamicError`	::=	`(DynamicError s)`,	*DynamicError*
		where $\mathtt{s} \in$ the set of Java strings	

Identity

Evaluating a Call Expression

```
Value visit(CallExp e, Env env) {
  //Step 1: Evaluate operator
  Object result = e.operator().accept(this, env);

  if (!( result instanceof Value.FunVal))
    return new Value.DynamicError("Operator not a function");
  Value.FunVal operator = (Value.FunVal) result;
  List<Exp> operands = e.operands();

  //Step 2: Evaluate operands
  List<Value> actuals = new ArrayList<Value>(operands.size());
  for(Exp exp : operands)
    actuals.add((Value)exp.accept(this, env));

  //Step 3: Evaluate function body
  List<String> formals = operator.formals();
  if (formals.size()!=actuals.size())
    return new Value.DynamicError("Argument mismatch in call ");
  Env fenv = appendEnv(operator.env(), initEnv);
  for (int i = 0; i < formals.size(); i++)
    fenv = new ExtendEnv(fenv, formals.get(i), actuals.get(i));
  return (Value) operator.body().accept(this, fenv);
}
```

Review and Further Reading

FuncLang: lambda expression and call

- Syntax: lambda expression and call (AST node, visitor interface)
- Semantics: funval, dynamic errors

Further reading:

- Rajan: CH 5, Sebesta Ch 9, 10

Review: Function:

① Write recursive, high order, list-related programs to solve problems

② currying, function as abstration

③ syntax, semantics of functions

How to Further Improve the Expressiveness of FuncLang

Control Structure

- if expression: three mandatory expressions - the condition, then, and else expressions
- comparison expression: $>, <, =$

Control Structure: Extending Value

Value	::=		*Values*
		NumVal	*Numeric Values*
		| BoolVal	*Boolean Values*
		| FunVal	*Function Values*
		| DynamicError	*Dynamic Error*
NumVal	::=	(NumVal n)	*NumVal*
BoolVal	::=	(BoolVal true)	***BoolVal***
		| (BoolVal false)	
FunVal	::=	(FunVal var_0,.., var_n e env)	*FunVal*
		where var_0,.., var_n ∈ Identifier,	
		e ∈ Exp, env ∈ Env	
DynamicError	::=	(DynamicError s),	*DynamicError*
		where s ∈ the set of Java strings	

Figure 5.7: The set of Legal Values for the Funclang Language with new boolean value

Control Structure: Grammar

Exp ::=	*Expressions*
Number	*NumExp*
(+ Exp Exp$^+$)	*AddExp*
(- Exp Exp$^+$)	*SubExp*
(* Exp Exp$^+$)	*MultExp*
(/ Exp Exp$^+$)	*DivExp*
Identifier	*VarExp*
(let ((Identifier Exp)$^+$) Exp)	*LetExp*
(Exp Exp$^+$)	*CallExp*
(lambda (Identifier$^+$) Exp)	*LambdaExp*
(if Exp Exp Exp)	***IfExp***
(< Exp Exp)	***LessExp***
(= Exp Exp)	***EqualExp***
(> Exp Exp)	***GreaterExp***
#t | #f	***BoolExp***

Figure 5.6: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

Semantic Rules

VALUE OF GREATEREXP

$$\frac{\texttt{value}\ \texttt{exp}_0\ \texttt{env} = (\texttt{NumVal}\ n_0) \quad \texttt{value}\ \texttt{exp}_1\ \texttt{env} = (\texttt{NumVal}\ n_1) \quad n_0 > n_1 = b}{\texttt{value}\ (\texttt{GreaterExp}\ \texttt{exp0}\ \texttt{exp}_1)\ \texttt{env} = (\texttt{BoolVal}\ b)}$$

VALUE OF EQUALEXP

$$\frac{\texttt{value}\ \texttt{exp}_0\ \texttt{env} = (\texttt{NumVal}\ n_0) \quad \texttt{value}\ \texttt{exp}_1\ \texttt{env} = (\texttt{NumVal}\ n_1) \quad n_0 == n_1 = b}{\texttt{value}\ (\texttt{EqualExp}\ \texttt{exp0}\ \texttt{exp}_1)\ \texttt{env} = (\texttt{BoolVal}\ b)}$$

VALUE OF LESSEXP

$$\frac{\texttt{value}\ \texttt{exp}_0\ \texttt{env} = (\texttt{NumVal}\ n_0) \quad \texttt{value}\ \texttt{exp}_1\ \texttt{env} = (\texttt{NumVal}\ n_1) \quad n_0 < n_1 = b}{\texttt{value}\ (\texttt{LessExp}\ \texttt{exp0}\ \texttt{exp}_1)\ \texttt{env} = (\texttt{BoolVal}\ b)}$$

> ((lambda (x) x) 2)

2

((lambda (x) x) 2)

2

> ((lambda (x) x) 2)

2

Control Structure: Semantic Rules

$$\frac{\texttt{value}\ \texttt{exp}_{cond}\ \texttt{env} = (\texttt{BoolVal true}) \qquad \texttt{value}\ \texttt{exp}_{then}\ \texttt{env} = \texttt{v}}{\texttt{value (IfExp}\ \texttt{exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{)}\ \texttt{env} = \texttt{v}}\ \textsc{Value of IfExp - True}$$

$$\frac{\texttt{value}\ \texttt{exp}_{cond}\ \texttt{env} = (\texttt{BoolVal false}) \qquad \texttt{value}\ \texttt{exp}_{else}\ \texttt{env} = \texttt{v}}{\texttt{value (IfExp}\ \texttt{exp}_{cond}\ \texttt{exp}_{then}\ \texttt{exp}_{else}\texttt{)}\ \texttt{env} = \texttt{v}}\ \textsc{Value of IfExp - False}$$

dynamic error

```
if (2) ((lambda a (x) x) 2)
```

Grammar with List

Exp ::=	*Expressions*
Number	*NumExp*
(+ Exp Exp$^+$)	*AddExp*
(- Exp Exp$^+$)	*SubExp*
(* Exp Exp$^+$)	*MultExp*
(/ Exp Exp$^+$)	*DivExp*
Identifier	*VarExp*
(let ((Identifier Exp)$^+$) Exp)	*LetExp*
(Exp Exp$^+$)	*CallExp*
(lambda (Identifier$^+$) Exp)	*LambdaExp*
(if Exp Exp Exp)	*IfExp*
(< Exp Exp)	*LessExp*
(= Exp Exp)	*EqualExp*
(> Exp Exp)	*GreaterExp*
#t | #f	*BoolExp*
(car Exp)	***CarExp***
(cdr Exp)	***CdrExp***
(null? Exp)	***NullExp***
(cons Exp Exp)	***ConsExp***
(list Exp*)	***ListExp***

Figure 5.8: Extended Grammar for the Funclang Language. Non-terminals that are not defined in this grammar are same as that in figure 5.1.

design decisions

{ keywords of language
API

Extending the Values

Value	::=		*Values*
		NumVal	*Numeric Values*
	|	BoolVal	*Boolean Values*
	|	FunVal	*Function Values*
	|	PairVal	*Pair Values*
	|	NullVal	*Null Value*
	|	DynamicError	*Dynamic Error*
NumVal	::=	(NumVal n)	*NumVal*
BoolVal	::=	(BoolVal true)	*BoolVal*
	|	(BoolVal false)	
FunVal	::=	(FunVal var_0,.., var_n e env)	*FunVal*
		where var_0,.., var_n ∈ Identifier,	
		e ∈ Exp, env ∈ Env	
PairVal	::=	(PairVal v_0 v_1)	***PairVal***
		where v_0, v_1 ∈ Value	
NullVal	::=	(NullVal)	***NullVal***
DynamicError	::=	(DynamicError s),	*DynamicError*
		where s ∈ the set of Java strings	

Figure 5.9: The set of Legal Values for the Funclang Language with new pair and null values

(1,2)

(1,2)

(cons 1 (1,2))

How to Compute Values

pair val

value (ListExp exp_0 ... exp_n) env = (ListVal val_0 $lval_1$)

where exp_0 ... $exp_n \in$ Exp env $\in$ Env

value exp_0 env = val_0, ..., value exp_n env = val_n

$lval_1$ = (ListVal val_1 $lval_2$), ...,

$lval_n$ = (ListVal val_n (EmptyList))

A corollary of the relation is:

value (ListExp) env = (EmptyList)

(list (lambda (x) x) ( ))

How to Compute Values

The value of a CarExp is given by:

```
value (CarExp exp) env = val

    where exp ∈ Exp   env ∈ Env

value exp env = (ListVal val lval) where lval ∈ ListVal
```

The value of a CdrExp is given by:

```
value (CdrExp exp) env = lval

    where exp ∈ Exp   env ∈ Env

value exp env = (ListVal val lval) where lval ∈ ListVal
```

The value of a ConsExp is given by:

```
value (ConsExp exp exp') env = (ListVal val lval)

    where exp,exp' ∈ Exp   env ∈ Env   value exp env = val
                               value exp' env = lval
```

The value of a NullExp is given by:

```
value (NullExp exp) env = #t if value exp env = (EmptyList)

                value (NullExp exp) env = #f
if value exp env = (ListVal val lval') where lval' ∈ ListVal

    where exp ∈ Exp   env ∈ Env
```