HW1

Aarti Kashyap
CPSC 539B

January 18, 2019

1 Part 1

1.1 The Fi language

$\langle x \rangle$ ::= 0 | 1 |...|9
| a | b |...| z
| A | B |...| Z
| * | ' | + ...

$\langle v \rangle$ ::= x
| True | False
| 0 | 1 | ~~-1 ...~~
| Function x ->e

$\langle e \rangle$ ::= v
| (e)
| ~~e And e | e or e | Not e~~
| ~~e + e | e - e | e = e~~
| ee
| If e then e Else e
| ~~Let x = e In e~~
| ~~Let Rec f x = e In e~~
| list (e, e)
| list-left (e)
| list-right (e)
| record (e)
| record-select (e)

1.2 Explanation of language

e, v and x are the meta-variables of the language which represent expressions, values and variables respectively.

eval (e) = v
Every evaluation of e should result in one of the values v as described above which becomes the elimation form for the grammer.
In the expressions I have defined the value, paranthesis, arithmetic and boolean operations, application expression, let (Ocaml style) anf finally rec using let.

The let expression and rec expression can not be included in the syntax and can be used to encode using logical combinators. Similarly recursion can be expressed through that by combining the existing definitions.

I have tried to define a OCaml style language. However, with simplest operations.

1.3 Expressiveness of Fi Language

1. **List**

   In order to define a list, I am defining a 2-tuple (pairing) construtor. For example, (1,(2,(3,0))) will represent the 4-tuple (1,2,3,0).

$$list\ (e1, e2) \stackrel{\text{def}}{=} (Function\ e1 -> Function\ e2 -> Function\ x -> x\ e1\ e2)\ e1\ e2$$

   I define the following macros for projection. (list-ref)

$$list - left\ (e) \stackrel{\text{def}}{=} e\ (Function\ x -> Function\ y\ -> x)$$

$$list - right\ (e) \stackrel{\text{def}}{=} e\ (Function\ x -> Function\ y\ -> y)$$

   The reason I have defined the list operator as above, is because this way, the e1 and e2 outside don't get bound to the x inside. We retain the values of e1 and e2 just as we want.

   Hence, for the example mentioned in the HW1, we can see that when we project we get the same results.

   But there are many problems with this methodology. Though it solves the purpose of HW1.

   Problem

$$list - left\ (Function\ x -> 0\ )\ => 0$$

   But, it should produce errors, because the function is not a pair.

   Now, however, with these I can implement any types of list functionalities such as constant etc.

Side note: My naming convention is slightly different from list-ref as I have two projections defined right now (list -left and list -right) but main point is that for any type we will get same results.

2. **Sequencing**

   In order to define sequencing for Fi, I can express it in the following way.

$$e;\ e \stackrel{\text{def}}{=} e\ (Function\ n -> e`)$$

   I don't require sequencing here, however, I am still defining it.

3. **Nat-fold**

   So my language syntax contains a definition of recursion function. I will be proving the let and rec relationship further in the next section, where I define my rules.

   However, I have application, let operator (which allows me to define functions), conditional operators and finally a rec operation available in my language.

   Using the syntax:

   Let Rec f x =
   If e1 = 0 Then
   e2
   Else
   e3 In e1

2 Part 2

2.1 Reduction Rules

$$(Addition\ Rule) \qquad n + n` \ \cong \ sum\ of\ n\ and\ n`$$

Similar for -. And, Or, Not and = (the ones present in syntax above.)

$$(Conditional\ Rule) \qquad (If\ True\ Then\ e\ Else\ e`) \cong e$$

$$(Conditional\ Rule\ 2) \qquad (If\ False\ Then\ e\ Else\ e`) \cong e$$

2.2 Conversion and Equivalence Rules

2.2.1 Conversion Rules

$$(Reflexivity\ Rule) \qquad e \cong e$$

$$(Symmetry\ Rule) \qquad e \cong e`\ if e` \ \cong e$$

$$(Transitivity\ Rule) \qquad e \cong e“if e \cong e`and e` \cong e“$$

2.2.2 Equivalence Rules

$$(Equivalence\ Rule\ 1) \qquad ((Function\ x -> e)\ v) \cong (e[v/x])$$

$$(Equivalence\ Rule\ 2) \qquad (Function\ x ->e) \cong ((Function\ z ->Function\ x ->e)z)$$

$$(Equivalence\ Rule\ 2) \qquad (Function\ x ->e) \cong ((Function\ z ->Function\ x ->e)z)$$

2.3 Evaluation rules

$$(Evaluation\ Rule) \qquad eval(e) \ = \ v$$

$$(Congruence\ Rule) \qquad C[e] \cong C[e`]\ if\ e \cong e`$$

2.4 Relation for nat-fold

The nat-fold uses recursion, let, conditionals and equivalence checking in order to evaluate the functionality.
The conditional rule 1 checks for equality for further evaluation.
The equivalence rules are used for further evaluation of the recursive function.

3 Part 3

3.1 Operational semantics

In order to define the rules for the language

$$(ValueRule) \qquad \frac{}{v => v}$$

$$(Not\ Rule) \qquad \frac{e => v}{Not\ e => the\ negation\ of\ v}$$

$$(And\ Rule) \qquad \frac{e1 => v1 \quad e2 => v2}{e1\ And\ e2\ => the\ logical\ and\ of\ v1\ and\ v2}$$

The Or, +, = rules evaluate to a value in a similar way. (Repetative)

$$(If\ Rule) \qquad \frac{e1 => True \quad e2 => v2}{If\ e1\ Then\ e2\ Else\ e3\ => v2}$$

Similar rules for False

$$(Application\ Rule) \qquad \frac{e1\ => Function\ x -> e, e2 => v2,\ e[v2/x] => v}{e1\ e2 => v}$$

Similarly, I can define for list, records, rec and let.

Though they produce valid values. There are side-effects which can be observed from the language I have defined.

4 Part 4

1. To create a dictonary mapping, Fi makes use of records. The records are expressed as (label, expression). The records can be represented as a list of (label, expression).

2. However, my list operation has a hole, which allows it to pass functions as arguments.

3. Another problem is that it contains nested tuples (1,(2,(3,0))). However in order to map like the dictionary I need to represent it as a=5, b=120, c=false

4. So I define a new syntax for record. record (e) = label, expression

5. However, the language seems to have become very messy and mixed.

6. Record

$$(Record\ Rule) \qquad \frac{left\ (e1)\ =>\ v1, right\ (e2) =>v2}{e =>v}$$

   I think I have mixed up the operational semantics with defining constructors.

4.1 Dictionary mapping

1. list [record (a,5), record (b=120), record (c=false)]

2. Projecting list (e e)

3. I define record-select for comparing and then matching the label to its expression.

4. Record-select(e) recursively matches e with left-list(e1) of all existing records and finds the value.

5. Again I have not left the possibility of evaluating bad cases

5 Part 5

5.1 Implement is-zero

eval (is-zero? 0) = true
Expression: (Function x ->If x = 0 Then True Else False) 0
Proof:

$$\frac{\frac{}{(Function\ x\ ->\ If\ 3\ =\ x\ Then\ True)[0/x]\ Function->..}\ \frac{}{0=>0}\ \frac{\frac{\frac{}{0=>0}\ \frac{}{0=>0}}{0\ =0=>\ True}\ \frac{}{False}}{if\ 0=\ 0\ Then\ True\ Else\ False}}{(Function\ x->if\ x=0\ Then\ True\ Else\ False\ )\ 0}$$

5.2 Evaluation

eval (is-zero? 1) = false
Expression: (Function x ->If x = 0 Then True Else False) 1
Proof: A similar type of proof can be constructed for this as shown for the above expression. Exactly same infact except the argument is not zero.

Showing with the series of reductions and conversions, instead of a formal tree, is also quite similar.

1. The first step for proving the above function is use of Application rule ee = >v
2. Applying the application rule e1 becomes the first part of the expression which is the function and e2 is another expression whose evaluation is performed based on the Value rule as stated in Part 2.
3. Now, the expression e1 is evaluated, which is the function, and we use Equivalence relations again as stated in Part 2 to evaluate the function.
4. Finally afte substituion we are have an expression which we can evaluate and get to the answers

eval (is-zero? true) = false
The solution for this is exactly the same as above.

5.3 Part 6

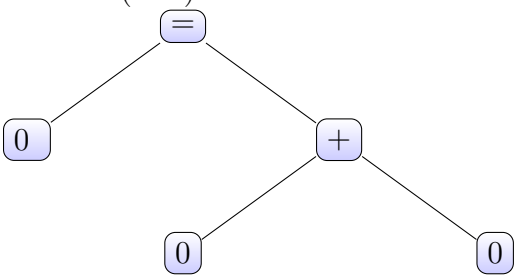
5.4 Derivation tree

According to my language the Plus operation is defined to give the sum of operations according to the rules in Part 2.

The expression is stated in abstract syntax format which is already in the form of syntax which the computer understands.

eval (+ 0 0) = 0

Plus (0 0)



5.5 Equations

eval (+ 0 1) = 1

Direct relation because in my language again I have considered that these are already defined from Part 2. Similarlly for others.

6 References:

1. Harpers "Practical Foundations of Programming Languages", http://www.cs.cmu.edu/ rwh/pfpl/2nded.
2. Grant, Palmer, and Smiths "Principles of Programming Languages", http://pl.cs.jhu.edu/pl/book/book.