Implementing a Simple Logic Inference Engine in F#

Libo Yin, 2116 4183
The University of Western Australia

October 12, 2013

1 High-Level Strategy

The problem solving process is modeled as a depth-first search tree. In contrast to a traditional search tree, this tree has following properties:

1. There are two types of nodes: `QUERY` nodes and `RULE` nodes
2. There are two types of edges: `AND` edges and `OR` edges.
3. A `QUERY` node is branched to `RULE` nodes through `OR` edges. A `QUERY` node succeeds if *any* of its children `RULE` nodes succeeds.
4. A `RULE` node is branched to `QUERY` nodes though `AND` edges. A `RULE` node succeeds if it can be applied to its parent `QUERY` node, and *all* of its children `QUERY` nodes succeeds.
5. An empty `QUERY` node succeeds.
6. The root of the tree is a `QUERY` node.

With a more problem-specific terminology, a node in the search tree is a `sufficency` in type definition: `type rule = Rule of sufficency * (sufficency list)`. A `RULE` node corresponds to the former one, i.e. the goal, while its children `QUERY` node corresponds to the latter one, i.e. the subgoals. A `sufficency` is satisfied if it can be unified with the goal of *any* rules, and satisfy *all* subgoals of that rule.

Figure 1 shows an examplary search tree of query: `Mix(A,A)→Mix(B,B)` using rule set: `x→x if [], A→B if [], Mix(x1,x2)→Mix(y1,y2) if [x1→y1,x2→y2]`.

2 Handling Unknowns

One distinct feature of logical programming is the existence of unknowns. Unknowns may appear in rules and queries, and may refer to expressions that either have already appeared before, or will only be known later. Unknowns are handled with the classic approach of unification, i.e. association of unknowns to known expressions. To assist the unification function, the concept of dictionary is introduced.

Dictionary is the cumulative result of successful unifications, containing mappings from unknowns to stable expressions. A stable expression is defined as a known expression, an unknown that is not in the dictionary, or a combination of stable expressions.

Dictionary is used in unification twice. Prior to the unification, a dictionary lookup is performed on each of the two expressions to be unified, reducing the number of unknowns. After the unification, new mappings are added to the dictionary. If the unification fails, no change is made to the dictionary.

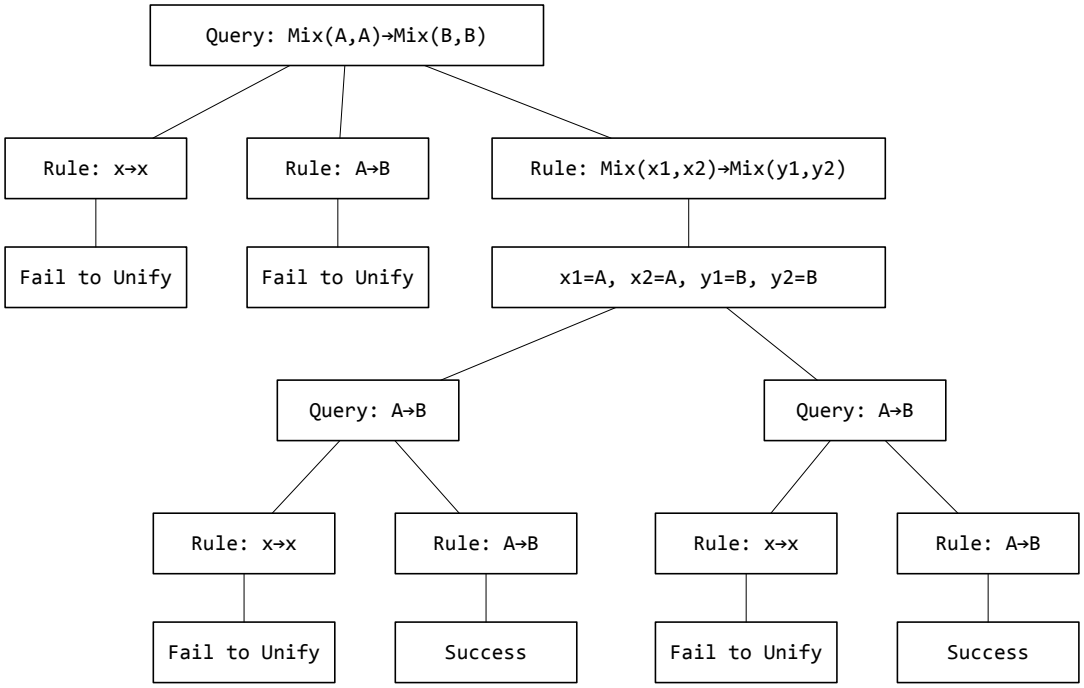

Figure 1: Examplary search tree of query: `Mix(A,A)→Mix(B,B)` using rule set: `x→x if [], A→B if [], Mix(x1,x2)→Mix(y1,y2) if [x1→y1,x2→y2]`.

Another issue concerning unknowns is that the same rule can be applied to different queries at different depths of the search tree. In such occations, the same unknown may refer to different expressions. This problem is solved by normalizing name of unknowns: Before unification, the current depth in the search tree is appendeded to the end of all unknowns in a rule.

3 Program Structure

A state in the solution process is defined as a 4-tuple `dictionary * queries * rules * depth`. `Queries` is a list containing all unsolved `QUERY` nodes of their parent `RULE` node. Likewise, `rules` is a list containing all unsolved `RULE` nodes of their parent `QUERY` node. Since a `QUERY` node branches through `OR` edges, an empty `rules` indicates the failure of its `QUERY` parent. On the other hand, since a `RULE` node branches through `AND` edges, an empty `queries` indicates the success of its `RULE` parent.

If neither `queries` nor `rules` is empty, the heads of them are taken as the current query and rule. The current rule is then normalized into a 2-tuple `goal * subgoals`. Afterwards, the current query is unified with the current goal. If the unification succeeds, which means an augmented dictionary is returned, the current subgoals are solved recursively with the augmented dictionary and all rules. If *all* subgoals succeed, the current query succeeds, and the solver continues with the tail of `queries`, the augmented dictionary, and all rules. On the other hand, if *any* subgoal fails, or if the unification fails, the solver tries to solve the current query again with other rules, i.e. the tail of `rules`, and the original dictionary.

A Uncommented Source Code

Please refer to the source code submission for commented source code.

```
let rec MapVar (dict:Map<string,exp>) e =
    match e with
    | A | B -> e
    | Mix(e1,e2) -> Mix(MapVar dict e1, MapVar dict e2)
    | Var(key) ->
       match dict.TryFind key with
       | Some(value) -> if value=e then e else MapVar dict value
       | None -> e

let rec Contains e vn =
    match e with
    | A | B -> false
    | Mix(e1,e2) -> Contains e1 vn || Contains e2 vn
    | Var(vn') -> vn=vn'

let rec UnifyExp dict e1 e2 =
    let (x1,x2) = (MapVar dict e1,MapVar dict e2)
    match (x1,x2) with
    | _ when x1=x2 -> Some(dict)
    | (Var(vn),e) | (e,Var(vn)) ->
       if Contains e vn then None else Some(dict.Add(vn,e))
    | (Mix(e1a,e1b),Mix(e2a,e2b)) ->
       match UnifyExp dict e1a e2a with
       | Some(dict') -> UnifyExp dict' e1b e2b
       | None -> None
    | _ -> None

let UnifySuff dict s1 s2 =
    match UnifyExp dict (fst s1) (fst s2) with
    | Some(dict') -> UnifyExp dict' (snd s1) (snd s2)
    | None -> None

let rec NormExp n e =
    match e with
    | A | B -> e
    | Mix(e1,e2) -> Mix(NormExp n e1, NormExp n e2)
    | Var(vn) -> Var(vn + string n)

let NormSuff n suff = (NormExp n (fst suff),NormExp n (snd suff))

let NormRule n (Rule(goal,subgoals)) =
    (NormSuff n goal,List.map (NormSuff n) subgoals)
```

```
43 let rec Solve ar dict queries rules depth =
44     if depth=8 then false else
45     match queries with
46     | [] -> true
47     | query::qTail ->
48         match rules with
49         | [] -> false
50         | rule::rTail ->
51             let (goal,subgoals) = NormRule depth rule
52             match UnifySuff dict query goal with
53             | Some(dict') ->
54                 if Solve ar dict' subgoals ar (depth+1)
55                 then Solve ar dict' qTail ar depth
56                 else Solve ar dict queries rTail depth
57             | None -> Solve ar dict queries rTail depth
58
59 let suffices ruleGens (exp1,exp2) =
60     let rules = List.map (fun f -> f()) ruleGens
61     Solve rules (Map.empty<string,exp>) [(exp1,exp2)] rules 0
```