Prolog 第四次读书班

桑世杰

July 9,2025

Outline

问题

- 如何增加某个变量的值?
- 如何移除 list 中的某个元素?
- 如何对某个数据结构进行变换?

增加变量值

- 在 C++ 中，这很简单：

$$i = i + 1;$$

增加变量值

- 在 C++ 中，这很简单：

  $$i = i + 1;$$

- 在 Prolog 中，我们需要从新旧状态之间的关系角度进行思考，通过如下语句实现：

  $$I \#= I0 + 1;$$

增加变量值

- 在 C++ 中，这很简单：

  $$i = i + 1;$$

- 在 Prolog 中，我们需要从新旧状态之间的关系角度进行思考，通过如下语句实现：

  $$I \#= I0 + 1;$$

- 这个关系能够在任何方向上使用.

移除列表中的元素 E

- 考虑两个列表的关系：第一个列表中包含元素 E，第二个列表中包含第一个列表中不等于 E 的所有元素；

移除列表中的元素 E

- 考虑两个列表的关系：第一个列表中包含元素 E，第二个列表中包含第一个列表中不等于 E 的所有元素；
- 如上，我们描述了三个实体之间的关系：两个列表和一个元素 E；

移除列表中的元素 E

- 考虑两个列表的关系：第一个列表中包含元素 E，第二个列表中包含第一个列表中不等于 E 的所有元素；
- 如上，我们描述了三个实体之间的关系：两个列表和一个元素 E；
- 这一关系同样能够在任何方向上使用，Prolog 的代码实现如下所示：

```
list1_element_list2([], _, []).
list1_element_list2([E|Ls1], E, Ls2) :-
    list1_element_list2(Ls1, E, Ls2).
list1_element_list2([L|Ls1], E, [L|Ls2]) :-
    dif(L, E),
    list1_element_list2(Ls1, E, Ls2).
```

全局状态

- Prolog 确实支持全局状态的修改，但会破坏逻辑编程的一些很重要的性质：
  - 不能够在所有方向上使用;
  - 相同的查询会得到不同的结果;
  - 无法独立于其它代码片段进行使用或测试.
- 要想充分地利用 pure 的程序的优势，我们需要找到陈述的方法来表达状态的变化.

问题

问题. 有三个水壶 A，B，C，它们的容量分别为 8，5，3，初始状态只有 A 是满的，B，C 都是空的，能否做到让 A，B 中各有 4 个单位的水?

问题

问题. 有三个水壶 A，B，C，它们的容量分别为 8，5，3，初始状态只有 A 是满的，B，C 都是空的，能否做到让 A，B 中各有 4 个单位的水?

显然，在这个问题中，状态应该是每个水壶当中的水量，在 Haskell 中用三元组 (A,B,C) 来表示这一状态：

```
type State = (Int,Int,Int)
```

后继

```
successors :: State -> [State]
successors (a,b,c) =
let ab = min a (5 - b)
    ac = min a (3 - c)
    ba = min b (8 - a)
    bc = min b (3 - c)
    ca = min c (8 - a)
    cb = min c (5 - b)
    ss = [(ab,a-ab,b+ab,c), (ac,a-ac,b,c+ac),
        (ba,a+ba,b-ba,c),
           (bc,a,b-bc,c+bc), (ca,a+ca,b,c-ca),
              (cb,a,b+cb,c-cb)]
in
  [(a',b',c') | (transfer,a',b',c') <- ss,
     transfer > 0]
```

测试

```
Main> successors (8,0,0)
[(3,5,0),(5,0,3)]
```

BFS

```
search :: [State] -> Bool
search (s:ss)
    | s == (4,4,0) = True
    | otherwise = search $ ss ++ successors s
```

BFS

```
search :: [State] -> Bool
search (s:ss)
    | s == (4,4,0) = True
    | otherwise = search $ ss ++ successors s
```

```
search [(8,0,0)]
True
```

路径

```
successors ((a,b,c),path) =
    let ab = min a (5 - b)
        ac = min a (3 - c)
        ba = min b (8 - a)
        bc = min b (3 - c)
        ca = min c (8 - a)
        cb = min c (5 - b)
        ss = [(ab, a-ab, b+ab,    c, path ++ [FromTo A B]),
              (ac, a-ac,    b, c+ac, path ++ [FromTo A C]),
              (ba, a+ba, b-ba,    c, path ++ [FromTo B A]),
              (bc,    a, b-bc, c+bc, path ++ [FromTo B C]),
              (ca, a+ca,    b, c-ca, path ++ [FromTo C A]),
              (cb,    a, b+cb, c-cb, path ++ [FromTo C B])]
    in
      [((a',b',c'), path') | (amount,a',b',c',path') <-
          ss, amount > 0]

search :: [State] -> Path
search (s:ss)
    | fst s == (4,4,0) = snd s
    | otherwise = search $ ss ++ successors s
```

路径

```
search [start]
[FromTo A B,FromTo B C,FromTo C A,
FromTo B C,FromTo A B,FromTo B C,FromTo C A]
```

Prolog version

```
jug_capacity(a, 8).
jug_capacity(b, 5).
jug_capacity(c, 3).

moves(Jugs) -->
    { member(jug(a,4), Jugs),
      member(jug(b,4), Jugs) }.
moves(Jugs0) --> [from_to(From,To)],
    { select(jug(From,FromFill0), Jugs0, Jugs1),
      FromFill0 #> 0,
      select(jug(To,ToFill0), Jugs1, Jugs),
      jug_capacity(To, ToCapacity),
      ToFill0 #< ToCapacity,
      Move #= min(FromFill0, ToCapacity-ToFill0),
      FromFill #= FromFill0 - Move,
      ToFill #= ToFill0 + Move },
    moves([jug(From,FromFill),jug(To,ToFill)|Jugs]).
```

Prolog version

为防止死循环，我们使用迭代深化（iterator deepening）的策略：

```
?- length(Ms, _),
   phrase(moves([jug(a,8),jug(b,0),jug(c,0)]),
   Ms).
Ms = [from_to(a,b),from_to(b,c),from_to(c,a),
      from_to(b,c),from_to(a,b),from_to(b,c),
      from_to(c,a)] .
```

Other Puzzles

- "wolf and goat"
- 8-puzzles
- Escape from Zurg
- "missionary and cannibal"

问题

我们现在想要在 Prolog 中创建一个解释器（interpreter），用来解释简单的对整数进行操作的程序。例如，我们想要对如下计算第四个 catalan 数的程序进行解释：

```
catalan (n) {
    if (n == 0) {
        return 1;
    } else {
        c = catalan(n-1);
        r = 2*(2*n + 1)*c / (n + 2);
        return r;
    }
}

print catalan(4);
```

ASTs

首先，我们用抽象语法树（abstract syntax trees (ASTs)）来对程序进行表征，为了表征上述程序，需要用到如下语句：

```
function(Name, Parameter, Body)
call(Name, Expression)
return(Expression)
assign(Variable, Expression)
if(Condition, Then, Else)
while(Condition, Body)
sequence(First, Second)
```

并且，为了区分变量和常数，我们用函子'v' 表示变量，'n' 表示常数。

Catalan

```
?- string_ast("catalan (n) { if (n == 0) { return 1; } else { c = catalan(n-1);
        r = 2*(2*n + 1)*c / (n + 2); return r; } } print catalan(4);", AST).

AST = sequence(function(catalan, n,
                        if(bin(=, v(n), n(0)),
                           return(n(1)),
                           sequence(assign(c, call(catalan, bin(-, v(n),
                                n(1)))),
                                    sequence(assign(r, bin(/,
                                                           bin(*,
                                                               bin(*,
                                                                   n(2),
                                                                   bin(+,
                                                                       bin(*,
                                                                           n(2),
                                                                           v(n)),
                                                                       n(1))),
                                                               v(c)),
                                                           bin(+, v(n),
                                                                 n(2)))),
                                                    return(v(r)))))),
               print(call(catalan, n(4))))
```

States

在解释上述程序的过程中，我们需要跟踪计算的状态，它由如下两部分组成

- the binding environment for variables
- all encountered function definitions.

这两点，可统称为环境（environment），在环境中，变量名与变量值相关联，函数名与函数体相关联，如此一来，解释语法树的谓词便可以定义环境间的关系，也就是状态间的关系，从而很好地定义函数、引用函数、处理变量。

Eval

为了根据当前环境计算表达式的值，我们用谓词 eval/3：

```
eval(bin(Op,A,B), Env, Value) :-
        eval(A, Env, VA),
        eval(B, Env, VB),
        eval_(Op, VA, VB, Value).
eval(v(V), Env, Value) :-
        env_get_var(Env, V, Value).
eval(n(N), _, N).
eval(call(Name, Arg), Env0, Value) :-
        eval(Arg, Env0, ArgVal),
        env_func_body(Env0, Name, ArgName, Body),
        env_clear_variables(Env0, Env1),
        env_put_var(ArgName, ArgVal, Env1, Env2),
        interpret(Body, Env2, Value).

eval_(+, A, B, V) :- V #= A + B.
eval_(-, A, B, V) :- V #= A - B.
eval_(*, A, B, V) :- V #= A * B.
eval_(/, A, B, V) :- V #= A // B.
eval_(=, A, B, V) :- goal_truth(A #= B, V).
eval_(>, A, B, V) :- goal_truth(A #> B, V).
eval_(<, A, B, V) :- goal_truth(A #< B, V).

goal_truth(Goal, V) :- ( Goal -> V = 1 ; V = 0).
```

Interpret

最终，谓词 interpret/3 具体说明了，语言的每一部分是如何改变环境的：

```
interpret(print(P), Env, Env) :-
    eval(P, Env, Value),
    format("~w\n", [Value]).
interpret(sequence(A,B), Env0, Env) :-
    interpret(A, Env0, Env1),
    (   A = return(_) ->
        Env = Env1
    ;   interpret(B, Env1, Env)
    ).
interpret(call(Name, Arg), Env0, Env0) :-
    eval(Arg, Env0, ArgVal),
    env_func_body(Env0, Name, ArgName, Body),
    env_clear_variables(Env0, Env1),
    env_put_var(ArgName, ArgVal, Env1, Env2),
    interpret(Body, Env2, _).
interpret(function(Name,Arg,Body), Env0, Env) :-
    env_put_func(Name, Arg, Body, Env0, Env).
```

Interpret

```
interpret(if(Cond,Then,Else), Env0, Env) :-
    eval(Cond, Env0, Value),
    (   Value #\= 0 ->
        interpret(Then, Env0, Env)
    ;   interpret(Else, Env0, Env)
    ).
interpret(assign(Var, Expr), Env0, Env) :-
    eval(Expr, Env0, Value),
    env_put_var(Var, Value, Env0, Env).
interpret(while(Cond, Body), Env0, Env) :-
    eval(Cond, Env0, Value),
    (   Value #\= 0 ->
        interpret(Body, Env0, Env1),
        interpret(while(Cond, Body), Env1, Env)
    ;   Env = Env0
    ).
interpret(return(Expr), Env0, Value) :-
    eval(Expr, Env0, Value).
interpret(nop, Env, Env).
```

注意

- print 语句会产生一个副作用（side-effect）：它会在终端上显示输出，而这不能通过转换环境来表达，因此对它的解释不 purely logical。为了解决这个问题，我们可以将“世界 (world)”状态的适当表示纳入我们的环境中，并在遇到 print 时对其进行适当的调整。
- return 语句也有些特殊，其生成的环境由单个值组成，谓词 eval/3 在评估函数调用时使用了这一点。

Run

我们解释一个程序时，需要从一个新的环境开始，并且丢弃结果产生的环境：

```
run(AST) :-
        env_new(Env),
        interpret(AST, Env, _).
```

我们可以通过如下方式运行上述 Catalan 数的例子：

```
?- string_ast("catalan (n) { if (n == 0) {
   return 1; } else { c = catalan(n-1);
        r = 2*(2*n + 1)*c / (n + 2); return r;
           } } print catalan(4);", AST),
           run(AST).
```

栈基虚拟机

为了消除上述解释器在查找变量和函数时的环境查询开销，可以设计一个栈基虚拟机（Stack-based Virtual Machine）及其指令集，用于高效执行编译后的程序。它的核心思想是通过偏移量（offset）寻址实现 O(1) 复杂度的变量访问。

虚拟机代码的设计

那么如何从抽象语法树 (AST) 生成虚拟机 (VM) 代码的系统设计呢？核心是使用 Prolog 的定子句文法 (DCG) 和半上下文表示 (semicontext notation) 来隐式管理编译状态。

编译过程需要跟踪四个状态组件：

- Is：已生成的指令序列（逆序存储）
- Fs：函数名 $\rightarrow$ 地址的映射表
- Vs：变量名 $\rightarrow$ 栈偏移量的映射表
- PC：下一条指令的地址（程序计数器）

并将其封装为四元组：s(Is, Fs, Vs, PC)

隐式状态传递

在这种情况下，只有少数谓词会对状态进行引用和更改，因此此时利用 compile/3 对状态进行传递较为繁琐：

```
compilation(functor(Arg1,Arg2,...,ArgN), State0, State) :-
    compilation(Arg1, State0, State1),
    compilation(Arg2, State1, State2),
        :
        :
    compilation(Argn, State_(N-1), State_N),
    vminstr(instruction_depending_on_functor, State_N,
        State).
```

通过半上下文表示（Prolog semicontext notation）会更加简洁：

```
compilation(functor(Arg1,Arg2,...,ArgN)) -->
    compilation(Arg1),
    compilation(Arg2),
        :
        :
    compilation(Argn),
    vminstr(instruction_depending_on_functor).
```

编译流程

```
ast_vminstrs(AST, VMs) :-
    initial_state(S0),                  % 初始化状态
    phrase(compilation(AST), [S0], [S]), % 编译AST
    state_vminstrs(S, VMs).             % 提取最终指令

initial_state(s([],[],[],0)).
```

```
state_vminstrs(s(Is0,Fs,_,_), Is) :-
    reverse([halt|Is0], Is1),           % 反转指令并添加halt
    maplist(resolve_calls(Fs), Is1, Is). % 解析函数地址

% 将call(函数名)替换为call(实际地址)
resolve_calls(Fs, call(Name), call(Adr)) :-
    memberchk(Name-Adr, Fs).            % 查函数地址表
```

状态操作

```
state(S) --> [S].  % 获取当前状态
state(S0, S) --> [S0], [S]. % 修改当前状态
```

```
% 添加新指令
vminstr(I) -->
    state(s(Is,Fs,Vs,PC0), s([I|Is],Fs,Vs,PC)), % 更新指令列表
    { I =.. [_,Args], length(Args, L), PC is PC0 + L }. % 更新PC

% 函数定义起点
start_function(Name, Arg) -->
    state(s(Is,Fs,_,PC), s(Is,[Name-PC|Fs],[Arg-0],PC)). % 初始化变量表

% 变量偏移量分配
variable_offset(Name, Offset) -->
    state(s(_,_,Vs0,_), s(_,_,Vs,_)),
    { ( memberchk(Name-Offset, Vs0) -> Vs = Vs0  % 已存在
        ; Vs0 = [_-Last|_], Offset is Last + 1,    % 新变量
          Vs = [Name-Offset|Vs0]                   % 添加新映射
      ) }.
```

接下来，我们就能够通过上述内容定义 compilation//1，开始编译了。

定理证明器

Prolog 能够进行定理证明吗？Richard O'Keefe 说过这样一句话：

Prolog is an efficient programming language because it is a very stupid theorem prover.

因此，Prolog 是愚蠢的定理证明器。

愚蠢

为什么说 Prolog 愚蠢?

Prolog 利用深度优先搜索，因此可能导致无限循环从而错过证明或反例。

是

为什么说 Prolog 是定理证明器呢?

Prolog 是图灵完备的编程语言，任何能在电脑上使用的定理证明器都能够在 Prolog 中生效。

例子

```
pl_resolution(Clauses0, Chain) :-
    maplist(sort, Clauses0, Clauses), % remove duplicates
    length(Chain, _),
    pl_derive_empty_clause(Chain, Clauses).

pl_derive_empty_clause([], Clauses) :-
    member([], Clauses).
pl_derive_empty_clause([C|Cs], Clauses) :-
    pl_resolvent(C, Clauses, Rs),
    pl_derive_empty_clause(Cs, [Rs|Clauses]).

pl_resolvent((As0-Bs0) --> Rs, Clauses, Rs) :-
    member(As0, Clauses),
    member(Bs0, Clauses),
    select(Q, As0, As),
    select(not(Q), Bs0, Bs),
    append(As, Bs, Rs0),
    sort(Rs0, Rs), % remove duplicates
    maplist(dif(Rs), Clauses).
```

例子

```
?- Clauses = [[p,not(q)], [not(p),not(s)],
    [s,not(q)], [q]],
   pl_resolution(Clauses, Rs),
   maplist(portray_clause, Rs).
```

```
[p, not(q)]-[not(p), not(s)] --> [not(q),
    not(s)].
[s, not(q)]-[not(q), not(s)] --> [not(q)].
[q]-[not(q)] --> [].
```

注意

- 我们没有使用 Prolog 的内置搜索策略作为证明者的搜索策略，而是使用迭代深化（iterative deepening）来保证反驳的完备性：如果反例存在，那么它一定能够被找到。迭代深化在 Prolog 中是很容易实现的，因为 length/2 能够创建越来越长的列表，从而限制搜索。乍一看，迭代深化似乎是一种非常低效的搜索策略，但实际上，在非常普遍的假设下，它是最优的搜索策略。
- 我们没有使用 Prolog 变量来表示对象级别的变量，而是利用 Prolog 原子代表命题变量。如果我们使用 Prolog 变量，那么我们可以使用布尔约束作为另一种解决方案，来证明公式是不可满足的：

```
?- sat(P + ~Q), sat(~P + ~S), sat(S + ~Q),
   sat(Q).
false.
```

因此，数据表示的选择会显著影响我们的推理。

在讨论使用 Prolog 实现的定理证明器时，我们必须记住，Prolog 内部的工作方式可能与证明器本身的实现行为有很大不同：它的搜索策略、变量的表示、逻辑属性，甚至任何东西都可能与 Prolog 内部的工作方式不同，因此我们只使用 Prolog 作为定理证明器的许多可能实现语言之一。

虽然如此，Prolog 的许多特性使其成为特别合适实现定理证明器的语言，例如：

- Prolog 的内置搜索和回溯在搜索证明和反例时可以很容易地使用
- 迭代深化等完备的搜索策略可以很容易地在 Prolog 中实现
- Prolog 的逻辑变量通常可用于表示对象级别的变量，使我们 absorb built-in Prolog features like unification
- 内置约束允许声明性规范，这使得利用 Prolog 能够写出非常优雅和高效的程序

例子

在 Prolog 中还能够实现：

- Presprover: Proves formulas of Presburger arithmetic
- TRS: Implements a completion procedure for Term Rewriting Systems in Prolog.

Prolog 也能够辅助证明英国彩票问题（证明需要 27 张票保证中奖），发现新李代数。