A Family of Unification-oblivious Program Transformations and Their Applications

Paul Tarau

University of North Texas

January 17, 2021

PADL'2021

Overview

- we describe a family of program transformations that compile a Horn Clause program into equivalent programs of a simpler and more regular structure
- our transformations, seen as morphisms between term algebras, commute with unification, clause unfoldings and Prolog's LD-resolution
- $\Rightarrow$ they are composable and preserve the operational semantics of the Horn Clause programs
- the resulting programs have simpler execution algorithms and can be compiled to minimalist instruction sets
- $\Rightarrow$ we sketch out a Python 3 implementation, also ported to Swift, Julia and C, see: `https://github.com/ptarau/LogicTransformers`

Motivations

Prolog's core execution algorithm combines Horn Clause unification and backtracking wrapped together as LD-resolution. Once familiar with it, one is likely to find out that deep down it is a strikingly simple mechanism!

- Two QUESTIONS:
  - can we bring Horn Clause logic to a simple form (e.g., comparable to expressing lambda calculus in terms of the `S` and `K` combinators)?
  - can we derive from where, as it happened in functional programming, an actual implementation?
- The APPROACH:
  - revisit Prolog's core execution algorithm and make it as self-contained as possible while also formally uniform
  - discover some theoretically interesting and at the same time easily implementable reductions to (unusually) simple canonical forms

Term Algebras

- *Term Algebra*: $\mathbb{T}$ a triple $< C, V, T >$ where C is a set of constant symbols, V is set of variable symbols and T is a set of terms for which the following construction rules hold:
  1. if $a \in C$ then $a \in T$
  2. if $X \in V$ then $X \in T$
  3. if $f \in C$ and $T_1, T_2..., T_n \in T$ then $f(T_1, T_2..., T_n) \in T$.
- Binary Term Algebra: $\mathbb{B}$, likewise, a triple $< C, V, B >$:
  1. if $a \in C$ then $a \in B$
  2. if $X \in V$ then $X \in B$
  3. if $B_1, B_2 \in B$ then $B_1 \Rightarrow B_2 \in B$.

Unfolding Monoids

- We denote $u(T_1, T_2)$ the result of applying $\theta = mgu(T_1, T_2)$ to any of them (as $T_1\theta = T_2\theta$) if the *mgu* exists and $\perp$ otherwise. We assume:

$$\forall T\ u(T, \perp) = \perp \ \texttt{ and } \ \texttt{u}(\perp, \texttt{T}) = \perp \tag{1}$$

Theorem

The following associativity property holds:

$$\forall A\ \forall B\ \forall C \quad u(A, u(B, C)) = u(u(A, B), C) \tag{2}$$

The Monoid of Horn Clause Unfoldings

- *LD-resolution* is SLD-resolution, specialized to Prolog's selection of the first atom in the body of clause.
- Its basic step can be described as an *unfolding* operation between two Horn Clauses.

$$(A_0\text{:-}A_1, A_2, \ldots, A_m) \odot (B_0\text{:-}B_1, B_2, \ldots, B_n) = (A'_0\text{:-}B'_1, B'_2, ..B'_n, A'_2, \ldots, A'_m) \quad (3)$$

where A'_i and B'_j are the result of applying the most general unifier of A_1 and B_0 to A_i and B_j, respectively, for all $i = 0, \ldots, m$ and $j = 1, \ldots, n$.

- We assume also:

$$\forall X \quad \bot \odot X = X \odot \bot = \bot \quad (4)$$

continued

Theorem

Horn Clauses form a monoid with the unfolding operation $\odot$. The $\odot$ operation is associative and the clause V :- V where V is a variable acts as its identity element.

- the monoid structure associated to a term algebra $\mathbb{T}$ can also be seen as a *category with a single object* on which its unfolding operations act as morphisms

The Monoid of Binary Clause Unfoldings

$$(A_0 \text{ :- } A_1) \odot (B_0 \text{ :- } B_1) = (A'_0 \text{ :- } B'_1) \tag{5}$$

where A'_0 and B'_1 are the result of applying the *mgu* of A_1 and B_0 to A_0 and B_1, respectively. We assume also:

$$\forall X \quad \bot \odot X = X \odot \bot = \bot \tag{6}$$

Theorem

Binary Clauses form a monoid with the unfolding operation $\odot$. The $\odot$ operation is associative, V :- V (where V is a variable) acts as the identity element and $\bot$ acts as a zero element.

Binarization: from Horn Clauses to Binary Clauses

The binarization transformation $bin_{\mathbb{T}} : HC_{\mathbb{T}} \rightarrow BC_{\mathbb{T}}$ maps a Horn Clause to a continuation passing Binary Clause as follows.

$$bin((A_0 \text{ :- } A_1, A_2, \ldots, A_m)) = (A_0 \triangleright C \text{ :- } A_1 \triangleright A_2 \triangleright \ldots, A_n \triangleright C) \tag{7}$$

where

$$f(A_1, \ldots, A_m) \triangleright B = f(A_1, \ldots, A_m, B) \tag{8}$$

and C is a variable not occurring in A_0 :- $A_1, A_2, \ldots, A_m$.

LD-resolution is Oblivious to Binarization

Theorem

$bin_{\mathbb{T}} : HC_{\mathbb{T}} \rightarrow BC_{\mathbb{T}}$ is a injection and its left inverse bin^{-1} can be used to restore answers computed by the binarized program. Prolog's LD-resolution computation acting on a program P (seen as iterated unfoldings in $HC_{\mathbb{T}}$) is equivalent to LD-resolution of binary clauses (seen as iterated unfoldings in $BC_{\mathbb{T}}$), where $\mathbb{T}$ is the term algebra built with the set of constants and variables occurring P.

$$bin_{\mathbb{T}}(C_1 \odot C_2) = bin_{\mathbb{T}}(C_1) \odot bin_{\mathbb{T}}(C_2) \tag{9}$$

$$C_1 \odot C_2 = bin_{\mathbb{T}}^{-1}(bin_{\mathbb{T}}(C_1) \odot bin_{\mathbb{T}}(C_2)) \tag{10}$$

$\Rightarrow$ *LD-resolution is oblivious to binarization*, in the sense that the same answers will be computed with it or without it!

The LD-resolution-oblivious Bijection between $\mathbb{T}$ and $\mathbb{B}$

We define $bt : \mathbb{T} \rightarrow \mathbb{B}$ as follows:

1. if c is a constant, then $bt(c) = c$
2. if v is a variable, then $bt(v) = v$
3. if $x = f(x_1, \ldots, x_n)$ then
   $bt(x) = bt(x_1) \Rightarrow bt(x_2) \ldots \Rightarrow bt(x_n) \Rightarrow f$

Theorem

The transformation $bt : \mathbb{T} \rightarrow \mathbb{B}$ is a bijection.

Example:

```
bt( f(A,g(a,B),B) ) = A=>(a=>B=>g)=>B=>f.
```

Equational Form in the Binary Term Algebra $\mathbb{B}$

1. if c is a constant then $eqf(X = c)$ is $X = c$
2. if V is a variable then $eqf(X = V)$ is $X = V$
3. if $A = (x_1 \Rightarrow x_2)$ then $eqf(X = A)$ is the nested conjunction $X = (X_1 \Rightarrow X_2)\ ,\ eqf(X_1 = x_1)\ ,\ eqf(X_2 = x_2)$

we extend this to a (binary) Horn Clause as

$$eqf(A_0 \text{:-} A_1) \quad = \quad p(X_0)\text{:-}eqf(X_0 = A_0), eqf(X_1 = A_1), p(X_1) \tag{11}$$

where p is a *new constant symbol*, not occurring in the clause.
Note that we only need a single new symbol "p" for a given (binarized) program.

Theorem

LD-resolution on P computes the same answers as LD-resolution on $eqf(P)$. Moreover, the same applies when permuting the order of the unification equations in any of the clauses.

Composing the Transformations

Theorem

Let ϕ and ψ two transformation that commute with Horn Clause unfoldings (and thus LD-resolution), i.e., such that:

$$C_1 \odot C_2 \ = \ \phi^{-1}(\phi(C_1) \odot \phi(C_2)) \tag{12}$$

and

$$C_1 \odot C_2 \ = \ \psi^{-1}(\psi(C_1) \odot \psi(C_2)) \tag{13}$$

where C_1 and C_2 are Horn Clauses over their respective term algebras. Let $\xi = \phi \ . \ \psi$ be the composition of the two transformations. Then

$$C_1 \odot C_2 \ = \ \xi^{-1}(\xi(C_1) \odot \xi(C_2)) \tag{14}$$

$\Rightarrow$ Thus, composition of our transformations enables *compiling* Horn Clause programs to alternative forms which, when executed, will compute the same answers.

Step by Step, one Transformation after Another ...

The Triplet Normal Form of Horn Clause Programs

Theorem

A Horn Clause program can be transformed into an equivalent Horn Clause program program (we call it its Triplet Normal Form) that has:

1. *a single unary tail recursive predicate with Datalog clauses*
2. *a single binary function symbol*
3. *a single non-Datalog fact of arity 3 referring to the function symbol*

- *Proof sketch* Apply *bin . bt . eqf* (in left to right order). Then, the unification operation "=" can be encapsulated as the predicate `u/3` as the unit clause:

```
u(X,Y,(X=>Y)).
```

- when called as `u(A,B,C)`, it is equivalent to the unification operation `(A=>B)=C`

Example of Triplet Normal Form

The predicate `p/1` with its self-recursive last call.

```
?- Cls = ( a(f(X)):-b(c,X),d(g(X)) ),
     cls2tnf(Cls,TNF),portray_clause(TNF).

p(C) :-
    u(D, f, A),
    u(E, a, B),
    u(A, B, C),
    u(D, g, F),
    u(E, d, G),
    u(F, G, H),
    u(H, b, I),
    u(D, I, J),
    u(c, J, K),
    p(K).
```

The 3-Instruction Assembler

We distinguish between the entry point to a clause with opcode **'d'** and the recursive call to the predicate `p/1` with opcode **'p'** together with the unification operation **'u'/3**.

```
a(f(A)) :- b(c, A), d(g(A))
```

becomes:

```
d A
u B f C
u D a E
u C E A
u B g F
u D d G
u F G H
u H b I
u B I J
u c J K
p K
```

Sketch of a Virtual Machine

- implementations in Python, Swift, Julia and C
- all at: `https://github.com/ptarau/LogicTransformers`
- our run-time instruction interpreter consists of:
  - an inner loop
  - a *trampoline* mechanism that avoids recursion and enables last call optimization (LCO)
  - we borrow garbage collection (GC) from the implementation language (Python, Swift, Julia) or use the conservative Boehm GC (C)
  - $\Rightarrow$ a Horn Clause Prolog VM (extensible with built-ins, IO, etc.) in a few hundred lines of code

The Inner Loop, in Python

```
def step(G, i, code, trail):
  ttop = len(trail)
  while i < len(code):
    unwind(trail, ttop)
    clause,vars = code[i],[]
    i += 1  # next clause
    for instr in clause:
      c = activate(instr, vars)
      op = c[0]
      if op == 'u':
        if not unify((c[1], c[2]), c[3], trail) : break
      elif op == 'd':
        c[1][0] = G
        trail.append(c[1])
      else:  # op==p
        NewG, tg = deref(c[1])
        if NewG == 'true' and CONST == tg: return (DONE, G, ttop, i)
        else: return (DO, NewG, G, ttop, i)
  return (FAIL,)
```

Handling the 3 Virtual Machine Instructions

The inner loop of the `step(G,i)` function, where `G` represents the current goal and `i` represents the next clause index to be tried:

- the **'d'** instruction marks the entry to a new clause to be tried against the current goal `G` and binds its argument (a variable) to it
- the **'u'** instruction, after inlining some special cases of unification, calls the general unification algorithm, mimicking the equivalent of the Prolog `d(A,B,A=>B)` fact
- the **'p'** instruction marks the end of the unification stream and the success of the current clause
  - it returns to the *trampoline* a `DO` instruction implying that there's more work to do
  - or, it returns a `DONE` instruction when no continuation is left to explore and an answer needs to be returned

The Outer Loop and Trampoline Mechanism, in Python

```
def interp(code,goal) :
  todo,trail,l=[(DO,goal,None,0,None)],[],len(code)
  while todo :
    instr=todo.pop()
    op=instr[0]
    if DO==op :
      _,NewG,G,ttop,i=instr
      r=step(NewG,0,code,trail)
      todo.append((UNDO,G,ttop,i))
      todo.append(r)
    elif DONE==op:
      _, G, ttop, i = instr
      todo.append((UNDO,G,ttop,i))
      yield iron(goal)
    elif UNDO==op :
      _, G, ttop, i=instr
      unwind(trail,ttop)
      if i!=None and i<l : todo.append(step(G,i,code,trail))
    else : pass # FAIL == op:
```

A Quick Speed Test

VM Implementation	10 queens program
Python, standard	67.521s
Julia	53.998s
Swift	28.494s
PyPy (JIT compiler for Python)	10.647s
C, with unions and structs	3.530s
C, with tagged 64 bit pointers	**0.461s**
Prolog-in-Prolog VM	1.006s
SWI-Prolog, directly, our baseline	*0.049s*
YAP, directly	0.032s

Conclusions

- composable, unification-oblivious Horn Clause program transformations are useful for succinctly understanding and implementing Prolog's core execution algorithm
- the 3-instruction assembler can help porting unification-based logic programming inference mechanisms to platforms where *resource limitations* or *urgency of implementation* matters
  - IOT devices, small drones, CubeSats
  - smart appliances
  - wearable devices
  - portable or implanted medical devices
- a conjecture: *If a term transformation that has a left inverse and maps variables into variables and constants into constants, then it is oblivious to unification and can be extended into an unfolding monoid on Horn Clauses that commutes with LD-resolution.*