A Family of Unification-oblivious Program Transformations and Their Applications

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
paul.tarau@unt.edu

Abstract. 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, thus preserving the operational semantics of the Horn Clause programs.
As applications, the resulting programs have simpler execution algorithms and can be compiled to minimalist instruction sets.

Keywords: term algebras, Horn Clause unfoldings, program transformations, minimalist canonical forms of logic programs, simplified instruction sets, lightweight Prolog run-time systems

1 Introduction

Prolog's core execution algorithm works as a search mechanism on Horn Clauses programs. It combines unification, backtracking and it is wrapped together as LD-resolution, a specialization of SLD resolution [7, 8] selecting for expansion the leftmost goal in the body of clause, in depth-first search order. It is this core execution algorithm that makes Prolog a general-purpose, Turing-complete programming language by contrast to SAT, SMT or ASP solvers. Adding built-ins, tabling and constraints or extending execution to incorporate sound handling of negation and co-inductive constructs entail orthogonal (and often minor) changes, from an implementor's perspective, to the underlying core execution mechanism.

Program transformations (including partial evaluation) operating on Horn Clause programs also relate to this core execution mechanism. In particular, unfolding of the leftmost goal in the body of a Horn clause mimics a single step in Prolog's LD-resolution.

Multiple equivalent views of Prolog's semantics e.g., via Herbrand models / fixpoints [7], working with ground terms or non-ground terms (e.g., S-models, C-models) [4] can all be seen as derived from this core execution mechanism.

Most of the literature covering such transformations, at least in terms of discovering new ones, dates from the 80's and 90's. The same applies to actual implementation models, initially derived empirically [18], although (sometime) consolidated later as formally accurate [11].

Our goal in this paper, after revisiting Prolog's core execution algorithm and making it as self-contained as possible while also formally uniform, is to discover some theoretically interesting and at the same time easily implementable reductions to unusually simple canonical forms. In a way, our program transformations will bring Horn Clause logic to a simplicity comparable to expressing lambda calculus in terms of the `S` and `K` combinators, from where, as it happened in functional programming [16], an actual implementation can be derived. As with combinator-based implementations in functional programming, this simplicity has a performance cost. We will show that in our case this performance hit can be significantly reduced.

As applications, we derive extremely lightweight proof-of-concept implementations, in a *few hundred lines* in **Python**, **Swift**, **Julia** and **C**, usable as starting points for deploying logic programming systems in resource-limited environments (e.g., IOT or wearable devices).

The SWI-Prolog [19] implementation of the transformers as well as open-source runtime-systems written in Prolog, Swift, Julia and C are available online[1].

We will proceed as follows. First, we simplify as much as possible the theory behind LD-resolution by reworking the data types inherited from predicate calculus in formally lightweight term algebras. As part of this "refactoring" we transform terms to an equivalent binary tree form via a bijection that commutes with unification. Next, starting from a set of Horn Clauses, via a sequence of semantics-preserving program transformations, we derive simpler equivalent Horn Clause programs. In particular, we revisit the binarization transformation [15] that we use to derive clauses with at most one goal in their body. Finally we convert to an equational form in which unification operations are expressed as triplets, ready to be turned into virtual machine instructions. Together with a *define* and a *proceed* instruction, we obtain our minimalist assembly language of 3-instructions, for we will describe a proof-of-concept Python implementation and discuss aspects of porting it to programming languages like Swift, Julia and C, while being able to delegate last call optimization and garbage collection to the underlying implementation-language.

The rest of the paper is organized as follows: Section 2 introduces the term algebras used in our transformations. Section 3 introduces the unfolding monoids defining our LD-resolution computation steps. Section 4 covers several composable transformations on term algebras and Horn Clause programs. Section 5 overviews applications to lightweight run-time system implementations, based on our transformations. Section 6 discusses related work and section 7 concludes the paper.

2 Term Algebras

2.1 The General Term Algebra

We define a *Term Algebra* $\mathbb{T}$ as 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] `https://github.com/ptarau/LogicTransformers`

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 $f(T_1, T_2..., T_n) \in T$.

Note that, as this is the norm in logic programming languages, we accept overloading of function symbols with different arities. Note also that, for the same reason we will use constant symbols interchangeably as predicate and term constructors and have variables as goals in atom in clause bodies. When needed, we also assume that an additional $\perp$ element, that's not part of its set of constants or variables, is added to a term algebra.

2.2 The Binary Term Algebra

We define a *Binary Term Algebra* $\mathbb{B}$ as a triple $< C, V, B >$ where C is a set of constant symbols, V is set of variable symbols and B is a set of terms for which the following construction rules hold:

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 $B_1 \Rightarrow B_2 \in B$.

For convenience, we assume that the "$\Rightarrow$" constructor is right associative i.e., we will write $A \Rightarrow B \Rightarrow C$ instead of $A \Rightarrow (B \Rightarrow C)$. In fact, one can see a Binary Term Algebra simply as a term algebra where terms are only built by using the symbol "$\Rightarrow$" with exactly two terms as its arguments. Note that the resulting language is essentially that of types for simply typed lambda terms + constants, or equivalently, via the Curry-Howard isomorphism [5], that of the implicational subset of intuitionistic propositional calculus. That also hints to our (otherwise arbitrary) choice of "$\Rightarrow$" as the binary constructor symbol for the algebra $\mathbb{B}$.

3 Unfolding Monoids

We assume that the reader is familiar with the most general unifier *mgu* of two terms [8]. 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 also that

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

Theorem 1. *The following associativity property holds:*

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

Proof sketch It follows from known properties of the *mgu* (see [8]) and assumption (1).

3.1 The Monoid of Horn Clause Unfoldings

LD-resolution is SLD-resolution [8], 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 *Horn Clause* $(A_0 \text{ :- } A_1,\ldots,A_m)$ with *head* A_0 and (non- empty) *body* $A_1,\ldots,A_m$ is built of terms $A_i \in \mathbb{T}$ where $\mathbb{T}$ is a term algebra.

Horn Clause *unfolding* is defined as as follows:

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

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 define the result of $\odot$ as the special term $\perp$ if the unification of A_1 and B_0 fails. We assume also that $\perp$ acts as a *zero element* (more generally, an *absorbing element*):

$$\forall X \;\; \perp \odot X = X \odot \perp = \perp \quad (4)$$

Computing an answer to a program for a query G can be seen as iterated unfoldings, starting from a clause G :- G, with fresh instances of clauses in the program. At the end, variable bindings in G correspond to answers to the query and reaching G :- $true$ marks a successful derivation.

Theorem 2. *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.*

Proof sketch: As u is associative (2) it follows that $\odot$ is associative. Propagation of failure follows from (4), given the action of $\odot$ on $\perp$.

As an intuition, note that the structure is similar to that on $\mathbb{N}$ with multiplication, where 1 acts as identity and 0 acts as a zero element, except, of course, for commutativity of multiplication in $\mathbb{N}$.

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.

3.2 The Monoid of Binary Clause Unfoldings

We consider a term algebra $\mathbb{T}$ extended with a bottom element $\perp$. Binary Clauses are Horn Clauses of the form H :- B with $H,B \in \mathbb{T}$.

A composition operation on binary clauses is defined as unfolding, specialized to binary clauses:

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

where A'_0 and B'_1 are the result of applying the most general unifier (mgu) of A_1 and B_0 to A_0 and B_1, respectively. We define the result of $\odot$ as the special term $\perp$ if the

unification of A_1 and B_0 fails. We assume also that $\perp$ acts as a *zero element*:

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

Theorem 3. *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 $\perp$ acts as a zero element.*

Proof sketch: It follows from theorem 2, as a special case.

The monoid can be also seen as a category with a single object with morphisms provided by the $\odot$ operation.

4 The Logic Program Transformers

We will now focus on program transformations that can be see as *morphisms* between our term algebras or as morphisms between the Horn Clause programs built on them.

4.1 Binarization: from Horn Clauses to Binary Clauses

Let $HC_{\mathbb{T}}$ denote the set of Horn Clauses with terms in a given term algebra $\mathbb{T}$ and $BC_{\mathbb{T}}$ the set of Binary Clauses (clauses having at most one term in their body).

The binarization transformation $bin_{\mathbb{T}} : HC_{\mathbb{T}} \to BC_{\mathbb{T}}$ 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 :- $\mathtt{A_1}, \mathtt{A_2}, \ldots, \mathtt{A_m}$.

Example 1 *Binarization of rules and facts. Note that $\triangleright$ embeds the second operand as last argument of the first.*

```
bin(a:-b,c,d) = a(Cont) :-b(c(d(Cont)))

bin(a(f(A)):-b(c,A),d(g(A))) = a(f(A), Cont):-b(c,A,d(g(A), Cont))

bin(f(a,b)) = f(a,b,Cont):-Cont
```

Note also that in the case of a fact, the continuation forming the body of its transformation via *bin* is a variable.

Theorem 4. *$bin_{\mathbb{T}} : HC_{\mathbb{T}} \to BC_{\mathbb{T}}$ 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.*

Proof sketch With computations proceeding via $\odot$ on the two sides, note that each computation in $BC_{\mathbb{T}}$ corresponds via bin^{-1} to a computation step in $HC_{\mathbb{T}}$ as the morphism $bin_{\mathbb{T}} : HC_{\mathbb{T}} \to BC_{\mathbb{T}}$ (also seen as a functor between the corresponding one-object categories) obeys the following relations:

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

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

To summarize, this means that *LD-resolution is oblivious to binarization*, in the sense that the same answers will be computed with it or without it.

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

Let $\mathbb{T}$ be a Term Algebra and $\mathbb{B}$ a Binary Term Algebra on the same set of constants and variables.

We define a bijection $bt : \mathbb{T} \to \mathbb{B}$ 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 5. *The transformation $bt : \mathbb{T} \to \mathbb{B}$ is a bijection.*

Proof sketch By induction on the structure of $\mathbb{T}$ and the structure of $\mathbb{B}$ for its inverse bt^{-1}.

Example 2 *Bijective transformation to binary tree*

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

Theorem 6. *Let us denote the result of unifying two terms and applying the unifier (in the respective term algebras) to either as the binary operator u. Then*
$bt(u(A,B)) = u(bt(A), bt(B))$

Proof Sketch Note that on both sides, we have the same variables. Thus a substitution of A with a term X on one side corresponds to a substitution with term $bt(X)$ on the other side. One can proceed by induction on size of the terms, keeping in mind that the multiway trees representing terms of $\mathbb{T}$ are in bijection with the binary trees representing them in $\mathbb{B}$, and that unifications are expressed as compositions of substitutions.

As an intuition, *bt* is similar to (and inspired by) *currying* in functional programming, in the sense that applying a function symbol to its arguments is equivalent to applying closures repeatedly.

Theorem 7. *Let $\mathbb{T}$ be a Term Algebra and $\mathbb{B}$ a Binary Term Algebra sharing the same sets of constants and variables. Let $HC_{\mathbb{T}}$ be a set of Horn Clauses and $HC_{\mathbb{B}}$ the result of applying bt to each A_i in each clause A_0 :- $A_1, \ldots, A_n$ in $HC_{\mathbb{T}}$. Then for $C_1, C_2 \in HC_{\mathbb{T}}$ the following holds:*

$$bt(C_1 \odot C_2) = bt(C_1) \odot bt(C_2) \tag{11}$$

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

Proof sketch It follows from theorems 8 and 6 by applying, inductively, the fact that *bt* commutes with unifications.

4.3 The Lifted One-Function Transformation

Another way to push function symbols symbols into constants is to mark all compound terms with a single functions symbol, not occurring in the term algebra, say \$. Let $T^{\$}$ be the term algebra $\mathbb{T}$ extended with the constant symbol \$. We define a function $hl : \mathbb{T} \to \mathbb{T}^{\$}$ as follows:

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

Theorem 8. *The transformation $hl : \mathbb{T} \to \mathbb{T}^{\$}$ is injective and its has a left inverse hl^{-1}. It commutes with unification, and when extended to Horn Clauses, it commutes with LD-resolution and the following relations hold:*

$$hl(C_1 \odot C_2) = hl(C_1) \odot hl(C_2) \tag{13}$$

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

where C_1 and C_2 are Horn Clauses.

Proof sketch By induction on the structure of $\mathbb{T}$ and the structure of $\mathbb{T}^{\$}$ for its inverse hl^{-1}.

Example 3 *Transformation to lifted one-function terms.*

```
hl( f(A,g(a,B),B) ) = $(f,A,$(f,a,B),B).
```

As all terms have only "\$" in function symbol positions, by omitting them, one can see these terms as multi-way trees with variable or constant labels only at leaf nodes.

4.4 Equational Forms

Equational Forms in Term Algebra $\mathbb{T}$ Given a term $A \in \mathbb{T}$ and an equation of the form $X = A$, it can be deconstructed to an canonical equational form $eqf(X = A)$ as follows:

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 $A = f(x_1,\ldots,x_n)$ then $eqf(X = A)$ is the nested conjunction
   $X = f(X_1,\ldots,X_n)$, $eqf(X_1 = x_1)$, ... , $eqf(X_n = x_n)$

where "=" stands for the unification operation and X_i are variables not occurring in A. Note that equations of the form $X = V$ with both terms variables can be eliminated by uniformly substituting X with V in all the equations it occurs. The same applies to equations of the form $X = c$ with c a constant.

Example 4 *Equational form*

```
eqf(X=f(a,g(V,b,h(c)),V)) =  X=f(a,X2,V),X2=g(V,b,X3),X3=h(c)
```

Equational Form in the Binary Term Algebra $\mathbb{B}$ In particular, given a term $A \in \mathbb{B}$ and an equation of the form $X = A$, its equational form is defined as follows:

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
   $X = (X_1 \Rightarrow X_2)$, $eqf(X_1 = x_1)$, $eqf(X_2 = x_2)$

Note that this suggest applying the composition of the transformations bt and eqf to reduce Prolog terms to a structurally simpler equational form.

Equational Form of a Horn Clause We extend this to a Horn Clause as

$$eqf(A_0 \text{ :- } A_1, \ldots, A_n) \; = \tag{15}$$

$$p(X_0) \text{ :- } eqf(X_0 = A_0), eqf(X_1 = A_1), \ldots, eqf(X_n, A_n), p(X_1), \ldots, p(X_n) \tag{16}$$

where p is a *new constant symbol*, not occurring in the clause. Note that the conjunctions resulting from eqf on the right side can be assumed to be flattened.

Given a program P defined as a set of Horn clauses, its equational form $eqf(P)$ is the set of the equational forms of its clauses.

Theorem 9. *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.*

Proof sketch It follows from an equational rewriting of the unification algorithm.

4.5 Composing the Transformations

Theorem 10. *Let ϕ and ψ two transformation that commute with Horn Clause unfoldings (and thus LD-resolution), i.e., such that:*

$$C_1 \odot C_2 \; = \; \phi^{-1}(\phi(C_1) \odot \phi(C_2)) \tag{17}$$

and

$$C_1 \odot C_2 \; = \; \psi^{-1}(\psi(C_1) \odot \psi(C_2)) \tag{18}$$

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

$$C_1 \odot C_2 \; = \; \xi^{-1}(\xi(C_1) \odot \xi(C_2)) \tag{19}$$

Proof sketch Trivial, using the fact that $\xi^{-1} = \psi^{-1} \; . \; \phi^{-1}$.

Thus, composition of our transformations enables *compiling* Horn Clause programs to alternative forms which, when executed, will compute the same answers. Moreover, with tweaks for built-ins, IO and other orthogonal language extensions, these can be used to derive actual, possibly much simpler implementations of unification-based logic languages.

5 Applications

5.1 The Triplet Normal Form of Horn Clause Programs

We will show next that composing the transformations like *bin*, *bt* and eqf results in a dramatically simple and uniform canonical form for Horn Clause programs.

Theorem 11. *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 follows:

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

Clearly, when called as `u(A,B,C)`, it is equivalent to the unification operation `(A=>B)=C` covering the equational form of a binarized program with terms in $\mathbb{B}$.

Example 5 *Triplet Normal Form of a Horn Clause*

```
?- Cls=(a:-b,c,d),to_tnf(Cls,TopVars,TNF).
Cls =  (a:-b, c, d),
TopVars = [B, E],
TNF = [u(A, a, B)]:[u(A, d, C), u(C, c, D), u(D, b, E)].
```

Note that our code has separated in the result `TNF` with "`:`" unification operations expressing the structures of the clause head and clause body.

Note also that `u(X,Y,Z)` can be split as:

```
u(X,Y,Z):-l(X,Z),r(X,Z).
```

assuming the two binary facts

```
l(X,X=>_).
r(Y,_=>Y).
```

Thus, we can have, alternatively, 2 binary predicates replacing the ternary `u/3`, given that one can define `d/3` as:

```
d(A,B,AB):-l(A,AB),r(B,AB).
```

Theorem 12. *Any reordering of the $u/3$ (or the $l/2$ and $r/2$) operations preserves the LD-resolution semantics of the resulting program.*

Proof sketch It follows from commutativity of conjunction in the equational form and commutativity and associativity of unifications (see [8]).

5.2 Deriving a Prolog Virtual Machine

Specifying the VM as Prolog Program We can now use the top variable corresponding to the head to define our unique Datalog predicate `p/1` and the top variable corresponding to the body as the recursive last call to the predicate itself.

Example 6 *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).
```

Note that one could separate the `u/3` unification triplets into ones coming from the head and the ones coming from the body when generating the "assembler" as these are meant exclusively to build terms on the heap with no call to unifications.

By putting the clauses together the resulting code *can be run as a Prolog program*, by adding the definition `u(A,B,A=>B)` as well as a first clause `p(X):-var(X),!` or equivalently, by introducing, like in [12, 13], a special symbol `$true`, assumed not to occur in the program, to mark the fact that no continuations are left to unwrap and execute. Reaching this special symbol or unbound variable continuation would also mean that a solution has actually been found, a hint on what a runtime system in a language other than Prolog should do.

The 3-Instruction Assembler We derive our 3 instruction assembler simply by distinguishing between the entry point to a clause with opcode **'d'** and the recursive call to the predicate `p/1` with opcode **'p'**. Together with our unification triples `u/3`, marked as instructions with opcode **'u'** we obtain our 3-instruction assembler code, ready to run on a VM designed for a procedural implementation language. Thus, the code for the clause:

```
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
```

5.3 Sketch of a Virtual Machine, in Python

Our implementation sketch takes advantage of the simplified instruction set generated through the composition of 3 program transformations, resulting in the Ternary Normal Form of a Horn Clause program as stated in theorem 11, in its 3-assembler instruction form. We refer to the actual implementation[2] for details not covered by this succinct description, focussed on the main ideas behind it.

The general implementation concept To keep things simple we avoid creating our own data types[3]. Variables are represented as Python lists of size 1 (or 2, if keeping names around). We represent the equivalent of Prolog's `A=>B` term as the the Python pair `(A,B)`. Constants could be any other Python types, as all we assume is that they are not Python tuples or lists and have a well defined "==" operation. In practice, we currently have as constants strings, ints and floats as these are also meaningful on the Prolog side where we compile the code.

Unification and dereferencing are standard, except that we use `None` rather than self-reference to mark unbound variables. Our unification algorithm uses its own local stack and trails variables when "older" than those originating from the current clause.

The main code interpreter consists of an inner loop (the function `step()` in our code snippet) and a *trampoline* mechanism (`function interp()` in our code snippet) that avoids recursion and enables last call optimization (LCO).

Besides the *trail*, we use a *todo* stack, accumulating things to do and undo. It is initialized with a term *goal(Answer,Continuation)*. By convention, we assume that the predicate `goal/1` has a definition in the sources. The *todo* stack combines goal stack and choice-point stack functions. It manages the current goal and the current next clause index to be tried (if any).

Instead of using a *heap*, for which we would need to write a garbage collector, we assume that the underlying language has a garbage collector, to which we gladly delegate all memory management tasks. Besides Python, languages like Swift, Julia, go, Java or C-sharp qualify, as well as plain C or C++ with the Boehm conservative garbage collector [1] enabled.

As our sketch of Python code will show , we can implement a runtime system supporting the core execution mechanism of Prolog, with key features as LCO and garbage collection "for free", in a host language that automates memory management.

---

[2] `https://raw.githubusercontent.com/ptarau/LogicTransformers/main/tnf2okAssocUnif.py`

[3] For most of our Python implementation variants. However, we define appropriate data types in Julia, Swift and C either as combinations of `struct` and `union` equivalents or by using more efficient tagged unsigned 64 bit integers for both values and array indices or pointers.

The inner clause selection loop As we do not actually manage a heap, the concept of *age* of a variable is undefined. Thus we need to know which variables originate from the current clause, to avoid unnecessary trailing. We achieve that with a lazily built set of new variables (as returned by Python's `id()` function, held in a register array (“‘`vars`“‘ in our code snippet).

Trying out a new clause involves creating its data objects from a *template*. We do that using the `activate` function that lazily copies and relocates elements of the template, while ensuring that variables with the same name correspond to the same variable object, with scopes local to each clause. The inner loop of the `step(G,i)` function, where `G` represents the current goal and `i` represents the clause index next to be tried, decodes our 3-instruction assembler as follows:

- the **'d'** instruction marks the entry to a new clause to be tried against the current goal `G`. It simply binds its argument, known to be a variable, to the current goal.
- 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 getting to the end of the unification stream marking the success of the current clause. It returns to the *trampoline* a `DO` instruction implying that there's more work to do or a `DONE` instruction when no continuation is left to explore and an answer needs to be returned. Note the presence of the `NewG` variable (to be explored next) as well as the `G` variable, with which execution would have to resume in case of failure.

At the end of the `step()` function's loop, if no more clauses are available, a `FAIL` operation is emitted, usable for tracing purposes but otherwise resulting in no action on the receiving side.

```
FAIL, DONE, DO, UNDO = 0, 1, 2, 3
```

```
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': #assert VAR == type_of(c[1])
        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,)
```

The main interpreter and its "trampoline" mechanism The "trampoline" loop implemented by the function `interp()` calls the `step()` function and handles the `FAIL, DONE, DO` operations returned by it, as well as the `UNDO` operations scheduled for later execution by its `DO` and `DONE` operations. Besides the trail, it uses the `todo` stack that is popped at each step, until empty. The stack holds "trampoline" instructions for scheduling new goals (`DO`), return answers (`DONE`) or to move control to the next unexplored clause (`UNDO`). Its key steps are:

- `DO`: it executes the recently popped instruction from the `todo` stack, initialized by the original `goal` term (containing the answer pattern as its first argument) and fed by returns from calling the `step()` function. It also schedules a clean-up `UNDO` operation by pushing it on the `todo` stack. Note that for a sequence of successful `DO` operations, their corresponding `UNDO` instructions will get executed in the right order, ensured by the stack's last-in first-out policy.
- `DONE`: works similarly to `DO`, except that an answer is emitted and no call to the `step()` function is performed. In this case, the `answer` variable (by convention, the first argument of the goal given to the VM) is "ironed" by the function `iron()` into a fully dereferenced term, ready to be yield to the user and insensitive to undoing the binding that lead to it.
- `UNDO`: unwinds the trail one level below the current one, as instructed by the argument "`ttop`" and passes control via the `step()` function to the next clause, as instructed by its "`i`" argument. Note that each new clause to be tried out in `step()` cleans up the mess left by the previous ones, by calling `unwind_trail`. Leaving this to `UNDO` would be too late and result in missing solutions. On the other hand, `UNDO` will be the right instruction to unwind the trail to its appropriate level, as transmitted by the corresponding `DO` instruction.
- `FAIL`: a no-operation instruction, useful only for tracing. While this is emitted when no clause is left to match a given goal, its clean-up tasks have been delegated to the `step()` function and the `UNDO` instruction as part of the process enabling them to ensure LCO.

```
# trampoline, ensures LCO and eliminates recursion
def interp(code,goal) :
  l=len(code)
  todo,trail=[(DO,goal,None,0,None)],[]
  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:
```

Note that with the *trampoline* mechanism we only rely on Python's `yield` operation for passing an answer from the trampoline to the user, something that can be replaced by printing it out, adding it to a queue or sending it over a socket.

Best target implementations are fast, GC-enabled programming languages like Swift, go, Java, C-sharp or Julia. Another possibility is C or C++ relying on the Boehm-Demers-Weiser garbage collector [1]. As applications, this implementation mechanism can enable running Logic Programming systems on network edges, smart IOT, routers, portable devices and GPU-threads.

Note that our proof-of-concept implementations have *no indexing* and the complete program is hosted in *a single predicate* precluding any common Prolog optimizations. Nevertheless, we will list here the progressively better results on computing all solutions for a **10-queens** problem for a few program variants and programming languages, from slower to faster (running on an iMacPro with a Xeon W processor):

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

While only about twice as fast as running the resulting assembler on an emulated VM in Prolog and one order of magnitude below the native execution speed in SWI-Prolog, our slightly optimized C-implementation shows that the performance penalty for deriving a compiler and a virtual machine for it from our program transformations is manageable. As a side note, the timings might also be an indicator about the price some high-level modern programming languages are willing to pay for language constructs supporting the data types and algorithms needed for executing the core of an unification-based logic programming language.

6 Related work

Scholarly work on theory and implementation of unification-based logic programming languages and their program transformations covers at this time more than 50 years. We will refer to classic work like [8] for the foundations of the field and [9, 10] for a comprehensive overview of some of the transformations we have taken inspiration from. As

we have revisited those essential concepts, we have used a uniform and (arguably) much simpler notation, focusing on the commonalities involved in morphing with their help Prolog's core execution mechanism into equivalent alternatives, including the triplet normal form used for deriving our 3-instruction assembler code.

The transformation between $\mathbb{T}$ and $\mathbb{B}$ is new and it was surprising to see that it has not been discovered previously as a general multi-way tree to binary tree bijection. Among possible alternatives, we have looked also at the well-known "Left-child right-sibling binary tree" transformer originating in [6], Section. 2.3.2, which is prone to move variables to function symbol positions and thus it is not commuting with unification operations. At the same time, we have passed also on LISP-based or Micro-Prolog's [3] "everything is a list" representations that, while commuting with unification, are more memory intensive and also prone to conflate Prolog's list data type with a list representation of terms. The uniform treatment of function and constant symbols as a result of the *bt* or *hl* transformations can enable similar operations as the use of arbitrary terms in function symbol positions [2], although we have not used this property here to work with higher-order programming constructs, contrary to [14], where such a representation is used. On the other hand, in [14] a representation derived from an equational form similar to eqf is used, but with a goal stacking mechanism instead of binarization and without the simplification brought by the transformation $bt : \mathbb{T} \to \mathbb{B}$ used here. The binarization transformation is introduced in [15], resulting in the BinProlog system [13], but its implementation is based on a simplified Warren Abstract Machine. More general continuation passing transformations, with goal atoms in argument position, as those raising from binarization as well as several examples of program optimizations using them are described in [10] and an extension to disjunctive continuations is explored via a metainterpreter in [17].

7 Conclusions and future work

We have presented a uniform view of a family of unification-oblivious Horn Clause program transformations and sketched an application to build lightweight implementations in several programming languages. The derived triplet normal form and the resulting 3-instruction assembler have opened the door for new implementation techniques for porting unification-based logic programming systems to provide reasoning capabilities on platforms where resource limitations or urgency of implementation might matter, ranging from IOT devices, small drones, CubeSats and smart appliances to wearable devices and portable or implanted medical devices.

An interesting theoretical outcome relies on the following *conjecture*, subject to possible future work.

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.

Any such a transformation, besides being ready to be used as the "inner language" of a Prolog implementation, would have applications to mechanisms for code serialization, code obfuscation or code encryption.

References

1. Boehm, H., Weiser, M.: Garbage collection in an uncooperative environment. Software — Practice and Experience **18**(9), 807–820 (1988)
2. Chen, W., Kifer, M., Warren, D.: HiLog: A first-order semantics for higher-order logic programming constructs. In: Lusk, E., Overbeek, R. (eds.) 1st North American Conf. Logic Programming. pp. 1090–1114. MIT Press, Cleveland, OH (1989)
3. Clark, K.L., McCabe, F.G.: Micro-Prolog - programming in logic. Prentice Hall international series in computer science, Prentice Hall (1984)
4. Falaschi, M., Levi, G., Martelli, M., Palamidessi, C.: A new declarative semantics for logic languages. In: Kowalski, R.A., Bowen, K.A. (eds.) Proceedings of the Fifth International Conference and Symposium on Logic Programming. pp. 993–1005. The MIT Press (1988)
5. Howard, W.: The Formulae-as-types Notion of Construction. In: Seldin, J., Hindley, J. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 479–490. Academic Press, London (1980)
6. Knuth, D.E.: The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition. Addison-Wesley (1997), `https://www.worldcat.org/oclc/312910844`
7. Kowalski, R., Emden, M.V.: The semantics of predicate logic as a programming language. JACM **23**(4), 733–743 (Oct 1976)
8. Lloyd, J.W.: Foundations of Logic Programming, 1st Edition. Springer (1984)
9. Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. J. Log. Program. **19/20**, 261–320 (1994). https://doi.org/10.1016/0743-1066(94)90028-0, `https://doi.org/10.1016/0743-1066(94)90028-0`
10. Pettorossi, A., Proietti, M.: Transformations of logic programs with goals as arguments. Theory Pract. Log. Program. **4**(4), 495–537 (2004). https://doi.org/10.1017/S147106840400198X, `https://doi.org/10.1017/S147106840400198X`
11. Russinoff, D.M.: A Verified Prolog Compiler for the Warren Abstract Machine. Journal of Logic Programming **13**, 367–412 (1992)
12. Tarau, P.: A Simplified Abstract Machine for the Execution of Binary Metaprograms. In: Proceedings of the Logic Programming Conference'91. pp. 119–128. ICOT, Tokyo (7 1991)
13. Tarau, P.: The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. Theory and Practice of Logic Programming **12**(1-2), 97–126 (2012)
14. Tarau, P.: A Hitchhiker's Guide to Reinventing a Prolog Machine. In: Rocha, R., Son, T.C., Mears, C., Saeedloei, N. (eds.) Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017). OpenAccess Series in Informatics (OASIcs), vol. 58, pp. 10:1–10:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). https://doi.org/10.4230/OASIcs.ICLP.2017.10, `http://drops.dagstuhl.de/opus/volltexte/2018/8453`
15. Tarau, P., Boyer, M.: Elementary Logic Programs. In: Deransart, P., Maluszyński, J. (eds.) Proceedings of Programming Language Implementation and Logic Programming. pp. 159–173. No. 456 in Lecture Notes in Computer Science, Springer, Berlin Heidelberg (Aug 1990)
16. Turner, D.A.: A new implementation technique for applicative languages. Software: Practice and Experience **9**(1), 31–49 (1979)
17. Vandenbroucke, A., Schrijvers, T.: Disjunctive Delimited Control (2020), available at: `https://arxiv.org/abs/2009.04909`
18. Warren, D.: An abstract Prolog instruction set. Technical Note 309, SRI International, Stanford, Ca (1983)
19. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**, 67–96 (1 2012). https://doi.org/10.1017/S1471068411000494