Scryer Prolog: A Modern ISO Prolog (Mostly) Written in Rust

Mark Thom
January 29th, 2020

Why a new Prolog?

- What I want from a Prolog environment
  - Strict conformance to the ISO standard
  - Many different ways of extending the language via metaprogramming:
    - Extensible unification (attributed variables)
    - Syntactic macros (term expansion & goal expansion)
    - Logically pure I/O
    - Delimited continuations
    - Tabling

Why a new Prolog?

- There are two kinds of Prologs available today:
  - Open source, “free as in beer” Prologs (SWI, ECLiPSe, GNU, YAP, etc.)
  - Commercial offerings (SICStus)
- The open source Prologs tend to suffer from these deficits:
  - A lack of, or inconsistency of, support for the ISO standard (hurts portability of programs)
  - A lack of suitably general interfaces for constraint propagators and other kinds of extensions (makes life harder for library implementers)

Why a new Prolog?

- SICStus is very good as a reference ISO implementation
  - … but it costs thousands of Euros and isn’t open source.
- I want a good platform for collaborative research and experimentation in logic programming and related domains!
- … without having to pay for it.
- … and that is able to freely absorb contributions from others.
- Scryer mimicks a few Scryer features and already has strong syntactic compliance with the ISO standard.

Why a new Prolog?

- Also, some limitations plague all popular Prologs
- Cut reduces the generality of Prolog programs and at the same time makes them harder to reason about
- Similarly, traditional arithmetic and I/O mechanisms are not fully declarative either. They have procedural readings.

Attributed variables

- An example of a constraint is found in the `dif/2` predicate
- `dif(X,Y)` is true if and only if `X \== Y`
- `dif(X,Y)` doesn’t simply check that `X` and `Y` are not equal…
- … it constructs a term on the heap representing the constraint
- If goals are posted resulting in `X` and `Y` being unified, the constraint causes the unification to fail.

Attributed variables

- Constraint propagation is made possible through the attributed variables extension
- The extension provides backtracking predicates that plant constraints as heap terms, attached to ordinary logical variables
- Constraint terms are "attributes" whose definition is scoped to a host module/namespace
- Within these modules, the predicate `verify_attributes/3` is defined:

```
verify_attributes(Var, Value, Goals) :-
    …
```

Attributed variables

- If X is unified to Y, and X is found to have attributes attached, the unification is undone, and the corresponding `verify_attributes/3` hook is executed
- `verify_attributes/3` is called as a normal Prolog predicate, and is expected to unify its third argument, `Goals`, with a list of terms to be called as Prolog goals
- Each goal in the `Goals` is called, and if they all succeed, `Var` is unified with `Value` once again

Declarative arithmetic and SAT solvers

- Another source of limitation is moded arithmetic:

```
?- X is -5 + 3 - (2 * 4) // 8.
X = 3.
```

- `is` expects the RHS to be a fully specified expression, and will throw an exception if it isn't
- To be fully declarative, we would expect:

```
?- 3 is X - 2.
X = 5.
```

Declarative arithmetic and SAT solvers

- Markus Triska's clp($\mathbb{Z}$) and clp(B) libraries provide fully moded integer arithmetic and a Boolean SAT solver respectively
- Markus' factorial predicate:

```
n_factorial(0, 1).
n_factorial(N, F) :-
   N #> 0,
   N1 #= N - 1,
   F #= N * F1,
   n_factorial(N1, F1).
```

```
?- n_factorial(6, F).
F = 720.
?- n_factorial(N, 720).
N = 6.
?- n_factorial(N, F).
N = 0, F = 0 ;
N  =  1,  F  =  0  ;
N  =  F,  F  =  1  ;
N = 3, F = 6 ...
```

Declarative arithmetic and SAT solvers

- An example of the clp(B) Boolean constraint solver relevant to integer programming:

```
?- sat(A#B), weighted_maximum([A,B], [1,2],
Maximum).

A = 0, B = 1, Maximum = 2.
```

- `A#B` is a Boolean formula, `A = 0, B = 1` is a solution maximizing the weighted assignment `w(A)= 1, w(B)= 2`
- Integer programming has been used to solve complex problems in scheduling, allocation, verification …

Reifying success/failure

- As a more involved example to consider:

```
member(X, [X|_]).
member(X, [_|Xs]) :-
    member(X, Xs).
```

- member/2 checks that X is a member of a list
- It has several problematic behaviours

Reifying success/failure

```
?- member(1, [1,2,3,4]).
true ;  % 1 is a member of the list.
false.  % we shouldn’t have to check the rest of the list.

?- member(X, [a,a,b,c]).
X = a ; % a is a member of the list.
X = a ; % we already saw that a is a member of the list!
X = b ;
X = c ;
false.
```

Reifying success/failure

- This shows that `member/3` generates redundant choice points, and sometimes, redundant answers
- We expect different behaviours from `member/3` depending on whether `X` is instantiated
- Neumerkel and Kral's paper "Indexing dif/2" introduces the predicate `if_/3`
- `if_/3` defers to a given branching predicate to know when to generate a choice point, or a commit to a certain branch

Reifying success/failure

- A simplified definition of `if_/3`:

```
if_(If, Then, Else) :-
  call(If, T),
  ( T == true -> call(Then)
  ; T == false -> call(Else)
  ; throw(error(_, _))
  ).
```

- The `If` goal takes a final truth argument `T` which is expected to be true or false
- Whether `If` backtracks between distinct true and false values is up to it

Reifying success/failure

```
=(X,Y,T) :-
      (  X == Y -> T = true % commit if we have no choice
      ;  X \= Y -> T = false
      ;  T = true, X = Y % allow backtracking if we do
      ;  T = false, dif(X, Y)
      ).
```

Reifying success/failure

- `member/2` may now be rewritten as:

```
member(E, Xs) :- i_memberd_t(Xs, E, true).


i_memberd_t([], _, false).
i_memberd_t([X|Xs], E, T) :-
     if_(X = E,
         T = true,
         i_memberd_t(Xs, E, T)).
```

Reifying success/failure

```
?- member(X, [a,a,b,Y,c,Z]).
X = a ;
X = b ;
X = Y, dif(a, X), dif(b, X) ;
X = c, dif(Y, c) ;
X = Z, dif(Y, Z), dif(a, X), dif(b, X), dif(c, X) ;
false.
?- member(1, [1,2,3]).
true. % deterministic.
```

Partial strings

- Prolog systems traditionally have a few shortcomings with regard to strings and how they are manipulated.
- For one, it’s often highly convenient to treat strings as lists of characters.
- In the Warren abstract machine, on which Scryer, SICStus & GNU, are based, “abc” represented as a list of characters on the heap would look like:

9	10					
LIS(10)	**CHAR(a)**	**LIS(12)**	**CHAR(b)**	**LIS(14)**	**CHAR(c)**	**CON([])**

Partial strings

9	10					
LIS(10)	CHAR(a)	LIS(12)	CHAR(b)	LIS(14)	CHAR(c)	CON([])

- This wastes a great deal of space and is slow to read and write.
- “Partial strings” are planned to allow the user to treat strings as difference lists of characters:

```
?- partial_string(“abc”, L, L0).
L = [a,b,c|L0].
```

- But! The heap representation of partial strings very closely resembles how characters are packed in UTF-8.

Scryer is written mostly in the Rust programming language

- … something I feel compelled to mention because most Prolog systems continue to be written in C.
- Rust:
  - has the speed of C & the expressivity of Java
  - is memory safe
  - lacks a GC
  - boasts algebraic data types
  - uses UTF-8 as its default character format
  - has a very nice package manager in the Cargo system
  - has hygienic macros..

Scryer is written mostly in the Rust programming language

- Rust is most like C++ with Standard ML’s type system and pattern matching bolted on..
- .. but Java programmers should feel at home with it relatively quickly
- Rust supplants objects and classes with structs and traits
- Memory management is provided through RAII
- Memory safety is maintained through the borrow system
- Unsafe Rust allows “dangerous” operations forbidden in Safe Rust: deferencing unmanaged pointers, arbitrary data casts

Future directions for Scryer Prolog

- Probabilistic logic programming
- Tabling via delimited continuations (on the verge of being finished!)
- On-demand multi-argument indexing of predicates
- Integrated statistical methods (see Taisuke Sato’s PRISM language)
- Unum computing? (unums are an alternative arithmetic format to IEEE 754 floating point)
- Precise garbage collection (see Weilemaker and Neumerkel’s “Precise Garbage Collection in Prolog” for what this would entail)

Resources

- The project page:
  https://github.com/mthom/scryer-prolog
- The Rust programming language:
  https://www.rust-lang.org
- Markus Triska's Power of Prolog textbook:
  https://metalevel.at/prolog