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

Mark Thom

December 16th, 2019

CAS–Atlantic

# Why a new Prolog?

- What I want out of a Prolog environment:
  - Strict conformance to the ISO standard
  - Extensive support for extending the core language via metaprogramming:
    - Extensible unification
    - Syntactic macros (term expansion & goal expansion)
    - Logically pure I/O
    - Delimited continuations
    - Tabling

# Why a new Prolog?

- There are two kinds of Prologs developed today:
  - Open source, "free as in beer" Prologs (SWI, ECLiPSe, GNU, Amzi!)
  - Commercial offerings like 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 constraints and other 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 requiring others to pay for it
- .. and that can freely absorb contributions from others
- Scryer mimicks quite a few SICStus features and has strong syntactic compliance with the ISO standard

# Why a new Prolog?

- 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, how Prolog systems have handled arithmetic and I/O are not fully declarative either.. they have a mostly procedural reading.

# Attributed variables

- One constraint is propagated through the `dif/2` predicate
- `dif(X, Y)` is true if `X \== Y`
- Moreover, it plants constraint terms in the heap
- If goals are posted that result in `X` and `Y` being unified, then we hit fail

# Attributed variables

- This is made possible by the attributed variables extension
- Idea: provide backtracking predicates that plant constraints as heap terms, attached to plain logical variables
- Constraint terms are expressed as attributes, which are defined in their own modules/namespaces
- Within those modules, the predicate verify_attributes/3 is defined:

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

# Attributed variables

- If `X` is unified to `Y`, and `X` has attributes attached, the unification is undone, and the `verify_attributes/3` hook is executed
- `verify_attributes/3` is called as a normal Prolog predicate, and is expected to unify `Goals` with a list of terms to be called as goals
- Each goal in the list `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 need something like this:

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

# Declarative arithmetic and SAT solvers

- Markus Triska's clp(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 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.`
- Constraint solvers like these have 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 predicate to know when to generate a choice point, or commit to a single 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 parameter `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` can now be re-written 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, X), dif(a, X), dif(b, X), dif(c, X).
false.
?- member(1, [1,2,3]).
true. % deterministic.
```

# Partial strings

- Prolog systems have had a few traditionally shortcomings with regard to partial strings
- It's often highly convenient / sensible to treat strings as lists of characters
- Scryer Prolog, like SICStus, GNU, etc. is based on the Warren abstract machine
- "abc" represented as a list of chars 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
- The idea of partial strings is to introduce a primitive allowing the user to treat strings as difference lists:

```
?- partial_string("abc", L, L0).

L = [a,b,c|L0].
```

- But the heap representation of partial strings more closely resembles how characters are packed in UTF-8:

```
abc\0 (8-byte address of the heap address of L0)
```

# Scryer is written mostly in the Rust programming language

- … something I feel obliged to mention, because most Prolog systems continue to be written in C.
- Rust is as fast as C, but as expressive as Java.. which is to say, not very expressive, but well ahead of C.
- Unlike Java, Rust is memory safe, lacks GC, and boasts a type system that is more appetizing than ranch dressing on microwaved rice.
- C is mostly good as a DSL for embedding security vulnerabilities in programs; Rust makes some security guarantees out of the box.

# Scryer is written mostly in the Rust programming language

- Rust is most like C++, but Java programmers should also be at home with it
- Rust supplants objects and classes with structs and traits
- Traits provide interface inheritance (trait objects <-> virtual functions <-> dynamic dispatch)
- Memory management in Safe Rust is provided through RAII
- Memory safety is maintained through the borrow system
  - Values can be borrowed any number of times immutably, but only once mutably
- Unsafe Rust allows "dangerous" operations forbidden in Safe Rust: dereference raw pointers and arbitrary casts between types, mostly

# Future directions for Scryer Prolog

- Future directions:
  - Probabilistic logic programming (Prolog clauses fire by the flip of a weighted coin)
  - Tabling via delimited continuations
  - Combined statistical and symbolic methods in AI (ie., Taisuke Sato's PRISM language)
  - Unum computing (unums are an alternative arithmetic format to IEEE 754 floating point)
  - Precise garbage collection

# Resources

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