Prolog-Style Meta-Programming miniKanren

NADA AMIN, Harvard University, USA
WILLIAM E. BYRD, University of Alabama at Birmingham, USA
TIARK ROMPF, Purdue University, USA

We bring external reification and reflection facilities to miniKanren, inspired by the traditional applications in Prolog such as meta-interpreters and partial evaluators. We illustrate meta-programming in relational programming with several examples.

1 INTRODUCTION

Logic programming favors relations over functions, aiming for a high-level declarative style, close to the inference rules one would write on paper. The appeal is evident. For example, we can write a type checker, and might get a type inferencer and even type inhabiter for free.

But sometimes, we also want to augment the evaluation of such programs. For example, we would like to get good error messages when our type inferencer fails. We want to reason about both failures and successes. In case of success, we may want a proof, i.e. a derivation tree, for why the relation holds. In case of failure, feedback is even more important, and yet, by default, a logic program that fails is one that returns no answers.

In Prolog, we can customize the execution of logic programs through meta-programming. In the Prolog community, meta-interpreters have a long tradition. An interpreter for "pure" Prolog clauses, written in Prolog, is easy to implement and to extend to modify the search strategy, inspect proof trees or investigate failures [Sterling and Shapiro 1994; Sterling and Yalcinalp 1989]. More advanced applications of meta-interpreters include partial evaluation [Sahlin 1991] and abstract interpretation [Codish and Søndergaard 2002]. These meta-interpreters do not necessarily stick to the "pure" subset themselves, and often exploit the more imperative features of Prolog.

In this paper, we show how to bring the power of Prolog-style meta-programming to miniKanren, a purely relational logic programming language embedded in Scheme.

We start with an overview of miniKanren, focusing on its embedded nature (Section 2).

We motivate the use of meta-programming: we would like to write high-level declarative programs, and customize their execution in certain ways without changing the code. Examples are adding tracing, computing proof trees, or reasoning about failures. We would also like to chose between different execution models like depth-first search, interleaving, or tabling. In Prolog, this is usually achieved through meta-interpreters (Section 3).

We show that the essence of meta-interpreters also applies in the embedded setting of miniKanren. Since it is not reasonable to self-interpret the full meta-language, we leverage deep linguistic reuse on the meta level: we can use our first-order engine to build program generators that compute first-order "pure" Prolog-like rules, which are again simple to interpret. But some of the appeal of meta-interpreters is lost because programs need to be turned into program generators manually. (Section 4).

We show how to use Scheme macros to automate reflection of the clause structures (Section 5).

We show how it is possible to reflect back the result of a query into reified clauses and show how it applies to partial reduction (Section 6).

We present a type debugger for Simply-Typed Lambda Calculus (STLC) as an application of meta-interpreting a relational type checker. (Section 7).

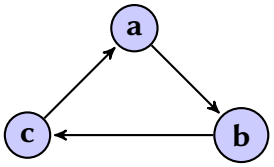
We discuss related work (Section 8) before concluding (Section 9).

Our code is online at https://github.com/namin/metamk.

2 MINIKANREN AS AN EMBEDDED LANGUAGE IN SCHEME

When embedding a language into an expressive host, we benefit from *deep linguistic reuse*: we can keep the embedded language simple by directly exploiting features of the host language. In this section, we illustrate deep linguistic reuse with miniKanren in Scheme, an embedded logic system which is first-order and and re-uses the host language for key features such as naming and structuring logic fragments.

As a running example, we model a graph connecting three nodes *a*, *b*, *c* in a cycle. In Prolog, we can model this graph with a relation, `edgeo`, listing all the possible edges:




```
edgeo(a,b).
edgeo(b,c).
edgeo(c,a).
```

2.1 Relations as Functions

In miniKanren, we can define the same relation as a definition in Scheme:

```
(define (edgeo x y)
  (conde
    ((== x 'a) (== y 'b))
    ((== x 'b) (== y 'c))
    ((== x 'c) (== y 'a))))
```

In miniKanren, the relation == unifies two terms. The special form **conde** is used to produce multiple answers; logically, it is a disjunction of conjunctions: each clause represents a disjunct, and is independent of the other clauses, with the goals within a clause acting as the conjuncts.

We can now run a query on the relation we just defined.

```
(run* (q)
  (fresh (x y)
    (== q `(,x ,y))
    (edgeo x y)))

↪ ((a b) (b c) (c a))
```

The **run** form serves as an interface between the host and the embedded language, returning a list of reified values of the query variable. In miniKanren, the **fresh** form introduces new logical variables and acts as a conjunction.

In Prolog, we can naturally define relations recursively, and so too in miniKanren. For example, the relation `patho` finds all the paths in the graph.

```
(define (patho x y)
  (conde
    ((edgeo x y))
    ((fresh (z)
       (edgeo x z)
       (patho z y)))))

(run 10 (q)
  (patho 'a q))

↪ (b c a b c a b c a b)
```

```
patho(X,Y) :- edgeo(X,Y).
patho(X,Y) :- edgeo(X,Z), patho(Z,Y).
```

Here, the query finds all paths starting with a and ending with the query variable. Because there are infinitely many paths through the cycle, **run*** would diverge. In sections 4, we show how to cope with this divergence by changing the evaluation semantics through meta-programming.

2.2 Higher-Order Relations as Higher-Order Functions

In miniKanren, we can exploit higher-order functions (and hence, relations too), parametrizing the relation `patho` by the relation `edgeo` so that it works for any graph:

```
(define (generic-patho edgeo x y)
  (conde
    ((edgeo x y))
    ((fresh (z)
      (edgeo x z)
      (generic-patho edgeo z y)))))
```

We could also recognize that the `patho` relation is really just the transitive closure of the `edgeo` relation, and use a more general abstraction.

3 WHY META-PROGRAMMING?

Logic programming enables us to concisely describe relations. For example, Figure 1 presents the typing relation of the simply-typed λ-calculus as inference rules as one would write on paper and in miniKanren. The miniKanren relation closely follows the paper rules. Since the relation is pure, it can be queried with logic variables placed anywhere, and so this one logic program can serve many purposes: type checking (provide term and type), type reconstruction (provide term but not type), type inhabitation (provide type but not term), and others. The `rule-case` unifications keep track of which inference rule is applied and is used for debugging as we explain next.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{(VAR)}$$

$$\frac{\Gamma, (x : T_1) \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} \quad \text{(ABS)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \rightarrow T \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T} \quad \text{(APP)}$$

```
(define-rel
  (!-o gamma expr type) (rule-case)
  (matche (expr type)

    ((,x ,T)
     (symbolo x)
     (lookupo `(,x : ,T) gamma)
     (== rule-case 'var))

    (((lambda (,x) ,e) (,T1 -> ,T2))
     (!-o `((,x : ,T1) . ,gamma) e T2)
     (== rule-case 'abs))

    (((,e1 ,e2) ,T)
     (fresh (T1)
       (!-o gamma e1 `(,T1 -> ,T))
       (!-o gamma e2 T1))
     (== rule-case 'app))))
```

Fig. 1. Simply-typed λ-calculus: on paper and in miniKanren

Meta-programming enables us to further extend the uses of a logic program, without compromising its concise description. For example, we can turn our miniKanren typing relation into a type debugger that generates derivation trees showing exactly where failures lie. Figure 2 shows such an auto-generated diagnostic. We type check the program $\lambda a_0.(a_0\ a_0)$ which has a self-application. The diagnostic points out to a failed unification $(T_1 \Rightarrow T_2) \neq T_1$ failing because of the "occurs-check".

$$\dfrac{\dfrac{}{[a_3 \mapsto (T_1 \Rightarrow T_2)\ ] \vdash a_3 : (T_1 \Rightarrow T_2)}\ \textsc{Var} \quad \dfrac{}{[a_3 \mapsto (T_1 \Rightarrow T_2)\ ] \vdash a_3 : (T_1 \Rightarrow T_2)}\ \textsc{Var} \quad (T_1 \Rightarrow T_2) \neq T_1}{[a_3 \mapsto (T_1 \Rightarrow T_2)\ ] \vdash (a_3\ a_3) : T_2}\ \textsc{App}$$
$$\dfrac{[a_3 \mapsto (T_1 \Rightarrow T_2)\ ] \vdash (a_3\ a_3) : T_2}{[\,] \vdash (\lambda a_0.(a_0\ a_0)) : ((T_1 \Rightarrow T_2) \Rightarrow T_2)}\ \textsc{Abs}$$

Fig. 2. Failure of self-application in STLC due to "occurs-check".

In the next section, we explain how to adapt Prolog-style meta-interpreters to miniKanren, which is what enables applications like the type debugger. After learning from the Prolog tradition, we expose meta-programming patterns that are particularly well-suited to the embedded setting.

4 THE ESSENCE OF PROLOG-STYLE META-INTERPRETERS

Meta-circular interpreters for the combined functional logic system seem difficult, since it would be unwieldy to self-interpret the meta-language. The key idea out of this dilemma is that deep linguistic reuse also applies on the meta level. Just as we can get away with a first-order engine for regular computation, we can use our first-order engine to build program generators that compute first-order "pure" Prolog-like rules, which are easy to interpret.

A clause in Prolog consists of a "head" (the left-hand side), and a "tail" or "body" (the right-hand side, possibly empty). One interpretation of a clause reads: to show the "head", it suffices to show the "body". In order to interpret a clause, we need to reify it, i.e. represent it as a data structure. Prolog provides some built-in constructs for this, but we can also do it manually, and partially select what we reify.

The technique we use is to turn individual rules into rule generators, that, when run, produce a reified version of the original rule.

For example, we can choose to reify the `patho` relation, turning the recursive calls into data, while eagerly evaluating the `edgeo` goals within:

```
patho_clause(patho(X,Y), []) :- edgeo(X,Y).
patho_clause(patho(X,Y), [patho(Z,Y)]) :- edgeo(X,Z).
```

Here is the miniKanren version of the same manual conversion of the `patho` program into a program generator, `patho-clause`, that produces the original program as data:

```
(define (patho-clause head tail)
  (fresh (x y)
    (== head `(patho ,x ,y))
    (conde
      ((edgeo x y) (== tail '()))
      ((fresh (z)
         (edgeo x z)
         (== tail `((patho ,z ,y))))))))
```

```
(run* (q) (fresh (head tail)
   (== q '(to prove ,head prove ,tail)) (patho-clause head tail)))
↪ ((to prove (patho a b) prove ())
    (to prove (patho b c) prove ())
    (to prove (patho a _.0) prove ((patho b _.0)))
    (to prove (patho c a) prove ())
    (to prove (patho b _.0) prove ((patho c _.0)))
    (to prove (patho c _.0) prove ((patho a _.0))))
```

We can see that, since edgeo has been left as-is, patho-clause actually produces variants of the patho rules that are partially evaluating with respect to the edgeo relation. Indeed, for example, to show a path from a, it suffices to show a path from b, since there is an edge from a to b. Of course, this manual conversion process is rather tedious, and we discuss how to automate it in Scheme (Section 5).

Reifying both the edgeo and patho relations gives us a direct data analog of the original Prolog clauses for patho:

```
(run* (q) (fresh (head tail)
   (== q '(to prove ,head prove ,tail)) (patho-full-clause head tail)))
↪ ((to prove (patho _.0 _.1) prove ((edgeo _.0 _.1)))
    (to prove (patho _.0 _.1) prove ((edgeo _.0 _.2) (patho _.2 _.1))))
```

Now that we have reified clauses, we can interpret them. We consider a few interpreter implementations for reified clauses next.

4.1 Vanilla Interpreter

The vanilla interpreter takes a clause higher-order relation, such as patho-clause, and returns a solver, which takes a list of reified goals, constraining them to hold. Implementation-wise, the solver is just another miniKanren program: if the list of goals is empty, then we're done; otherwise, we recursively solve the first goal, then the remaining goals.

```
(define (vanilla* clause)
  (define (solve* goals)
    (conde
      ((== goals '()))
      ((fresh (g gs body)
         (== (cons g gs) goals)
         (clause g body)
         (solve* body)
         (solve* gs)))))
  solve*)
```

```
(run 10 (q)
  ((vanilla* patho-clause)
   '((patho a ,q))))
↪ (b c a b c a b c a b)
```

Running the vanilla interpreter on the same query as previously (all the paths from a) gives the same result, cycling through all the nodes ad infinitum.

4.2 Tracing Interpreter

The vanilla interpreter often serves as a starting point to then augment the interpretation with a feature. Here, we add tracing. The parameters trace-in and trace-out accumulates the current trace at the beginning and end of the solve* call.

```
(define (tracer* clause)
  (define (solve* goals trace-in trace-out)
    (conde
      ((== goals '())
       (== trace-in trace-out))
      ((fresh (g gs body trace-out-body)
         (== (cons g gs) goals)
         (clause g body)
         (solve* body (cons g trace-in)
             trace-out-body)
         (solve* gs trace-out-body
             trace-out)))))
  (lambda (goal t)
    (solve* (list goal) '() t)))
```

```
(run 4 (q) (fresh (y t)
  (== q `(,y ,t))
  ((tracer* patho-clause)
   `(patho a ,y) t)))
↪ ((b ((patho a b)))
    (c ((patho b c) (patho a
        c)))
    (a ((patho c a) (patho b a)
        (patho a a)))
    (b ((patho a b) (patho c b)
        (patho b b) (patho a
        b))))
```

4.3 Cycle Detection and Other Extensions

Now, that we have a trace of the goals we step through as we find a path, we can also choose to fail when we are stepping through the same goal again: in effect, detecting cycles. The only change to the tracer interpreter is to add an `absento` constraint, `(absento g trace-in)`, which relationally enforces that its first argument does not occur in its second, just before the `(clause g body)` line.

```
(run* (q) (fresh (y t)
  (== q `(,y ,t))
  ((cycler* patho-clause)
   `(patho a ,y) t)))
```

```
↪ ((b ((patho a b)))
    (c ((patho b c) (patho a c)))
    (a ((patho c a) (patho b a) (patho
        a a))))
```

In a similar way, we can extend the tracing interpreter to build up proof trees or make failures explicit.

5 AUTOMATING REIFICATION

In Prolog, meta-interpreters are convenient thanks to reflective built-ins which automate reification, the process of turning Prolog clauses into Prolog data. How can we achieve the same convenience in our embedded setting?

As a first attempt, we build a Scheme macro, `define-rel`, that lexically reifies a program: it is used around the definition of a miniKanren relation, such as `patho`, to auto-generate reified clauses such as `patho-clause` and `patho-full-clause`.

```
(define-rel (patho x y)
  ((patho-clause patho) (patho-full-clause patho edgeo))
  ()
  (conde ((edgeo x y)) ((fresh (z) (edgeo x z) (patho z y)))))
```

For each relation to reify, the macro lexically redefines it to add a representation of the call to the tail (instead of executing the underlying call):

```
(define-syntax define-rel
  (syntax-rules ()
    ((_ (id a* ...) ((cid* r* ...) ...) (x* ...) body)
     (begin
       (define (id a* ...) (fresh-if-needed (x* ...) body))
       (define (cid* head tail x* ...)
         (define r* (lambda args (snoco tail `(r* . ,args)))) ...
```

```
        (fresh () (fresh (a* ...) (== head '(id ,a* ...)) body) (closeo tail)))
            ...
        (void)))))
```

6 REFLECTION FOR PARTIAL REDUCTION

We reproduce the partial reduction example and system of Sections 18.1 and 18.2 of the Art of Prolog [Sterling and Shapiro 1994].

A partial reduction system resembles a meta-interpreter. It explores the goals, and unfolds them, or folds them, according to user-given directives.

```
(define (preduce* clause)
  (define (solve* goals residues)
    (conde
      ((== goals '())
       (== residues '()))
      ((fresh (g gs)
         (== (cons g gs) goals)
         (conde
           ((fresh (a b ra rb rba rs)
              (== `(<- ,a ,b) g)
              (== `(<- ,ra ,rb) rba)
              (== (cons rba rs) residues)
              (solve* b rb)
              (solve* (list a) ra)
              (solve* gs rs)))
           ((fresh (r rs)
              (== (cons r rs) residues)
              (should-fold g r)
              (solve* gs rs)))
           ((should-unfold g)
            (fresh (b rb rs)
              (clause g b)
              (appendo rb rs residues)
              (solve* b rb)
              (solve* gs rs))))))))
  solve*)
```

The Non-Deterministic Finite Automata (NDFA) example implements a small interpreter that accepts a word according to rules specified per automata using `initial` states, `delta` transitions and `final` states.

```
(define-rel (accept xs)
  ((accept-clause accept accept2 initial delta final)) ()
  (fresh (q)
    (initial q)
    (accept2 xs q)))

(define-rel (accept2 xs q)
  ((accept2-clause accept2 initial delta final)) ()
  (conde
    ((fresh (x xr q1)
       (== (cons x xr) xs)
       (delta q x q1)
```

```
      (accept2 xr q1)))
   ((== xs '())
    (final q))))
```

We build an NDFA that accepts that language `(ab)*`.

```
(define-rel (initial q) ((initial-clause initial)) ()
  (== q 'q0))

(define-rel (final q) ((final-clause final)) ()
  (== q 'q0))

(define-rel (delta qa c qb) ((delta-clause delta)) ()
  (conde
    ((== qa 'q0) (== c 'a) (== qb 'q1))
    ((== qa 'q1) (== c 'b) (== qb 'q0))))
```

For the fold and unfold annotations used by the partial reduction system, we unfold the rules that are specific to the automata, and fold the general rules to be specialized.

```
(define (should-fold g r)
  (fresh (x y)
    (conde
      ((== g `(accept2 ,x ,y))
       (== r `(ab2 ,x ,y)))
      ((== g `(accept ,x))
       (== r `(ab ,x))))))
(define (should-unfold g)
  (fresh (x y z)
    (conde
      ((== g `(initial ,x)))
      ((== g `(final ,x)))
      ((== g `(delta ,x ,y ,z))))))
```

The general clauses that all together form the initial program to partially reduce.

```
(define (ndfa-pclause a b)
  (fresh (x y z)
    (conde
      ((== `(accept ,x) a)
       (accept-clause a b))
      ((== `(accept2 ,x ,y) a)
       (accept2-clause a b)))))
```

We automate the process of reflecting the result of running the clauses back into an object program. Here is the main routine:

```
(define (refl-prog pclause)
  (let ((program-obj
         (run* (q)
           (fresh (a b)
             (== q  (list '<- a b))
             (pclause a b)))))
    (let ((q (reify-prog program-obj)))
      (eval `(lambda (p) ,(q 'p))))))
```

We explain this reflection routine by example:

```
(define (ndfa-program p)
  ((refl-prog ndfa-pclause) p))
ndfa-program ↪
'(((<- (accept _.0) ((initial _.1) (accept2 _.0 _.1)))
   (<- (accept2 () _.2) ((final _.2)))
   (<- (accept2 (_.3 . _.4) _.5)
       ((delta _.5 _.3 _.6) (accept2 _.4 _.6))))))

;; step 1
(run* (q)
  (fresh (a b)
    (== q  (list '<- a b))
    (pclause a b))) ↪
'((<- (accept _.0) ((initial _.1) (accept2 _.0 _.1)))
  (<- (accept2 () _.0) ((final _.0)))
  (<- (accept2 (_.0 . _.1) _.2)
      ((delta _.2 _.0 _.3) (accept2 _.1 _.3))))

;; step 2 by reify-prog on result of step 1
↪
(fresh (_.0 _.1 _.2 _.3 _.4 _.5 _.6)
  (== p
      `((<- (accept ,_.0) ((initial ,_.1) (accept2 ,_.0 ,_.1)))
        (<- (accept2 () ,_.2) ((final ,_.2)))
        (<- (accept2 (,_.3 . ,_.4) ,_.5)
            ((delta _.5 _.3 _.6) (accept2 _.4 _.6))))))
```

Finally, the reduction in action:

```
(run 1 (q)
  (fresh (p)
    (ndfa-program p)
    ((preduce* ndfa-clause)
     p
     q))) ↪
'(((<- ((ab _.0)) ((ab2 _.0 q0)))
   (<- ((ab2 () q0)) ())
   (<- ((ab2 (a . _.1) q0)) ((ab2 _.1 q1))))))
```

The result is a program specialized to a particular NDFA without interpretation overhead.

7 APPLICATION: TYPE DEBUGGER

The advantage of writing a reified relation is that we can now inspect its inner workings through meta-programming.

We present a meta-interpreter that pinpoints a failure in a derivation tree.

It can be used to draw the annotated failed derivation of Figure 2 when ran on the clauses of Figure 1.

```
(define (proofdebug* clause)
  (define (solve* goals proof ok)
    (conde
      ((== goals '())
       (== proof '()))
     ((fresh (first-goal other-goals first-body body-proof other-proof rule-case)
```

```
        (== (cons first-goal other-goals) goals)
        (conda
          ((clause first-goal first-body rule-case)
           (== `((,first-goal ,rule-case <-- ,body-proof) . ,other-proof) proof)
           (solve* first-body body-proof ok))
          ((== `((,first-goal error) . ,other-proof) proof)
           (== ok #f)))
        (solve* other-goals other-proof ok)))))
  solve*)

(define (proofdebug clause)
  (let ((solve* (proofdebug* clause)))
    (lambda (goal proof ok)
      (fresh ()
        (solve* (list goal) proof ok)
        (conda
          ((== ok #t))
          ((== ok #f)))))))
```

Thus meta-interpreters extend the benefits of purely relational programming to reasoning about derivations.

8 RELATED WORK

There is a long tradition of meta-programming in Prolog, going back at least to the early 1980s. Warren [1982], O'Keefe [1990], and Naish [1996] discuss how to express higher-order "meta-predicates" inspired by functional programming, such as `map` and `fold`; O'Keefe uses Prolog's standard `call` operator, while Warren and and Naish advocate using an `apply` operator closer in spirit to Lisp. Warren claims that λ-terms are neither necessary nor desirable for higher-order programming in Prolog, arguing that passing the names of top-level predicates to meta-predicates is the best tradeoff between expressivity and keeping the Prolog language simple. Naish believes that `apply` is a more natural construct for higher-order programming than Prolog's traditional `call` operator, and claims that reliance on `call` by the logic languages Mercury [Somogyi et al. 1995] and HiLog [Chen et al. 1993] make higher-order programming in those languages awkward. Our host language Scheme support λ-terms and `apply`—we therefore inherit both the expressivity and the complexity of these language features.

According to Martens and de Schreye [1995], interest in Prolog meta-interpreters was spurred by two articles [Bowen and Kowalski 1982; Gallaire and Lasserre 1982] from a 1982 collection edited by Clark and Tärnlund. Introductory books on Prolog [O'Keefe 1990; Sterling and Shapiro 1994] further popularized meta-interpreters, which are now considered a standard approach to Prolog meta-programming. Hill and Lloyd claim that meta-interpreters in Prolog are fatally flawed, since they often use non-declarative features, and since it can be difficult to assign a semantics to untyped, unground logic programs; their strongly statically typed functional-logic-constraint language Gödel [Hill and Lloyd 1994] (and Lloyd's followup language, Escher [Lloyd 1995]) is specifically designed for declarative meta-programming. Martens and de Schreye [1995] defend Prolog-style meta-interpreters, arguing that all forms of untyped logic programming have the same issues that Hill and Lloyd point out, but that reasonable semantics can be applied to meta-programming in untyped logic languages.

There is also a long history of trying to combine functional programming and logic programming, once again going back to the early 1980s. There have been many attempts to embed a Prolog-like language in Lisp [Felleisen 1985; Haynes 1987; Robinson and Sibert 1982], and more recently, in

Haskell [Claessen and Ljunglöf 2000; Spivey and Seres 1999]; to our knowledge, there is no work in the literature on how to best write meta-interpeters for these embedded languages.

9 CONCLUSION

In this paper, we have shown how to handle Prolog-style meta-programming in miniKanren. Like in the Prolog tradition of meta-interpreters, these techniques enable transforming the evaluation of a logic program without complicating its description. In the embedded setting, we have the choice of meta-programming within the embedded language, or stepping out to the host language. By embracing this flexibility, we gain simplicity: the embedded logic language remains "pure" and first-order, tailored for relational programming.

REFERENCES

K. A. Bowen and R. A. Kowalski. 1982. Amalgamating Logic and Metalanguage in Logic Programming. In *Logic Programming*, K. L. Clark and S.-. Tärnlund (Eds.). Academic Press, 153–172.

Weidong Chen, Michael Kifer, and David Scott Warren. 1993. HiLog: A Foundation for Higher-Order Logic Programming. *J. Log. Program.* 15, 3 (1993), 187–230. http://dblp.uni-trier.de/db/journals/jlp/jlp15.html#ChenKW93

Koen Claessen and Peter Ljunglöf. 2000. Typed Logical Variables in Haskell. *Electr. Notes Theor. Comput. Sci.* 41, 1 (2000), 37. http://dblp.uni-trier.de/db/journals/entcs/entcs41.html#ClaessenL00

Michael Codish and Harald Søndergaard. 2002. Meta-circular Abstract Interpretation in Prolog. In *The Essence of Computation.* Springer Berlin Heidelberg, 109–134.

Matthias Felleisen. 1985. *Transliterating Prolog into Scheme.* Technical Report 182. Indiana University Computer Science Department.

H. Gallaire and C. Lasserre. 1982. Metalevel control of logic programs. In *Logic Programming*, K. L. Clark and S.-. Tärnlund (Eds.). Academic Press, 173–185.

Christopher T. Haynes. 1987. Logic Continuations. *J. Log. Program.* 4, 2 (1987), 157–176.

Patricia M. Hill and John W. Lloyd. 1994. *The Gödel programming language.* MIT Press.

J. W. Lloyd. 1995. *Declarative Programming in Escher.* Technical Report CSTR-95-013. Department of Computer Science, University of Bristol.

Bern Martens and Danny de Schreye. 1995. Why untyped nonground metaprogramming is not (much of) a problem. *Journal of Logic Programming* 22, 1 (Jan. 1995), 47–99.

Lee Naish. 1996. *Higher-order logic programming in Prolog.* Technical Report 96/2. University of Melbourne.

Richard A. O'Keefe. 1990. *The Craft of Prolog.* MIT Press, Cambridge, MA, USA.

J. A. Robinson and E. E. Sibert. 1982. LOGLISP: an alternative to PROLOG. In *Machine Intelligence 10*, J.E. Hayes, Donald Michie, and Y-H. Pao (Eds.). Ellis Horwood Ltd., 399–419.

Dan Sahlin. 1991. *An automatic partial evaluator for full PROLOG.* Ph.D. Dissertation. The Royal Institute of Technology, Stockholm, Sweden.

Z. Somogyi, F. J. Henderson, and T. C. Conway. 1995. Mercury, an Efficient Purely Declarative Logic Programming Language. In *Proceedings of the Australian Computer Science Conference.* 499–512.

J. M. Spivey and S. Seres. 1999. Embedding Prolog in Haskell. In *Proc. of the 1999 Haskell Workshop (Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht)*, E. Meijer (Ed.).

Leon Sterling and Ehud Shapiro. 1994. *The Art of Prolog (2nd Ed.): Advanced Programming Techniques.* MIT Press, Cambridge, MA, USA.

Leon Sterling and L. Umit Yalcinalp. 1989. Explaining Prolog Based Expert Systems Using a Layered Meta-interpreter *(IJCAI'89).* 66–71.

David H. D. Warren. 1982. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, J.E. Hayes, Donald Michie, and Y-H. Pao (Eds.). Ellis Horwood Ltd., 441–454.