Lightweight Functional Logic Meta-Programming

Nada Amin[1], William E. Byrd[2], and Tiark Rompf[3]

[1] Harvard University, USA (namin@seas.harvard.edu)
[2] University of Alabama at Birmingham, USA (webyrd@uab.edu)
[3] Purdue University, USA (tiark@purdue.edu)

Abstract. Meta-interpreters in Prolog are a powerful and elegant way to implement language extensions and non-standard semantics. But how can we bring the benefits of Prolog-style meta-interpreters to systems that combine functional and logic programming? In Prolog, a program can access its own structure via reflection, and meta-interpreters are simple to implement because the "pure" core language is small. Can we achieve similar elegance and power for larger systems that combine different paradigms?

In this paper, we present a particular kind of functional logic meta-programming, based on embedding a small first-order logic system in an expressive host language. Embedded logic engines are not new, as exemplified by various systems including miniKanren in Scheme and LogicT in Haskell. However, previous embedded systems generally lack meta-programming capabilities in the sense of meta-interpretation. Indeed, shallow embeddings usually do not support reflection.

Instead of relying on reflection for meta-programming, we show how to adapt popular multi-stage programming techniques to a logic programming setting and use the embedded logic to generate reified first-order structures, which are again simple to interpret. Our system has an appealing power-to-weight ratio, based on the simple and general notion of dynamically scoped mutable variables.

We also show how, in many cases, non-standard semantics can be realized without explicit reification and interpretation, but instead by customizing program execution through the host language. As a key example, we extend our system with a tabling/memoization facility. The need to interact with mutable variables renders this a highly nontrivial challenge, and the crucial insight is to extract symbolic representations of their side effects from memoized rules. We demonstrate that multiple independent semantic modifications can be combined successfully in our system, for example tabling and tracing.

1 Introduction

An appealing aspect of pure logic programming is its declarative nature. For example, it is easy to take a formal system, expressed as inference rules on paper, and turn it into a logic program. If the formal system describes typing rules, the same logic program might be able to perform type checking, type reconstruction, and type inhabitation. Yet, we want more.

First, we would like to leverage abstractions known from functional programming to structure our logic programs. Where logic programming sports search, nondeterminism, and backwards computation, functional programming excels at parameterization, modularity and abstraction. These strengths are complementary, and there is great value in combining them, as evidenced by a large body of ongoing research. Languages such as Curry [17] focus on integrating functional and logic programming into one coherent declarative paradigm.

Second, we would like to customize the execution of logic programs. For example, we want to be able 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, these tasks can be solved through *meta-programming*, which, in the context of this paper, means to implement a *meta-interpreter* for Prolog clauses. A meta-interpreter for "pure" Prolog clauses, written in Prolog, can customize the search strategy, inspect proof trees or investigate failures [34, 35]. However, for non-trivial applications such as abstract interpretation [10], these meta-interpreters do not usually stick to the "pure" Prolog subset themselves. In many cases, for example if we want to extend the execution logic with tabling or memoization, it is necessary to exploit decidedly un-declarative and imperative features of Prolog—in some sense the "dirty little secret" of logic programming.

In this paper, we present a pragmatic solution to combining functional and logic programming on one hand, and declarative logic programming with restricted notions of state on the other hand. We make the case for a particular style of functional logic meta-programming: embedding a simple, first-order logic programming system in an expressive, impure, higher-order functional host language, optionally supported by best-of-breed external constraint solver engines such as Z3 [25] or CVC4 [2], and providing explicit support for dynamically scoped, i.e., "thread-local" state. In the tradition of miniKanren [4, 6, 5, 13, 14], which embeds logic programming in Scheme, we present Scalogno, a logic programming system embedded in Scala, but designed from the ground up with modularity and customization in mind and with explicit support for dynamically scoped mutable variables.

This paper makes the following contributions:

- We introduce our system, Scalogno, and highlight the benefit of *deep linguistic reuse* in logic programming based on examples, e.g., how higher-order functions of the host language can model higher-order relations (e.g., `map`, `flatMap`, `fold`). The logic engine can remain first order, keeping theory and implementation simple. Scalogno can reuse Scala's type classes (e.g., `Ord`), while the logic engine need not be aware of this feature at all. This flexibility goes beyond dedicated functional logic languages like Curry, which do not support type classes for complexity reasons [24] (Section 2).
- Tracing, proof trees, etc. are examples of a whole class of use cases where a meta-interpreter augments execution with some state. We introduce dynamically scoped mutable variables to capture this design pattern, enabling

modular extensions through the host language as an alternative to explicit interpretation. We discuss the implementation of Scalogno in more detail, and also show how dynamic variables support a generic term reification facility, directly adapting popular multi-stage programming approaches to a logic setting (Section 3).
– We show how we can customize the execution order while maintaining the behavior of other extensions that rely on dynamic mutable state. To this end, we extend our logic engine to implement tabling, i.e., memoization. Unlike most existing Prolog implementations (there are exceptions [11]), the implementation directly corresponds to a high-level description of the tabling process, understood in terms of continuations. A key challenge is to interact with mutable variables, which we solve by extracting symbolic representations of their side effects from memoized rules. To the best of our knowledge, ours is the first logic engine that integrates tabling with mutable state in a predictable way (Section 4).

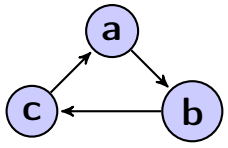
Section 5 discusses related work and Section 6 offers concluding thoughts. Our code is available at `aplas19.namin.net`.

2 Embedded Logic Programming

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 Scalogno in Scala—the embedded logic system is first-order, and re-uses the host language for key features such as naming and structuring logic fragments.

2.1 Relations as Functions

As a running example, we model a graph connecting three nodes a, b, c in a cycle.



In Prolog (on the left), we can model this graph with a relation, `edge`, listing all the possible edges. In Scalogno (on the right), we can define the same relation as a regular Scala function:

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

```
def edge(x: Exp[String],
         y: Exp[String]): Rel =
  (x === "a") && (y === "b") ||
  (x === "b") && (y === "c") ||
  (x === "c") && (y === "a")
```

In Scalogno, infix methods are used for unification (`===`), conjunction (`&&`) and disjunction (`||`). The type `Rel` represents a relation, while the type `Exp[T]` represents a term (possibly including unbound logic variables) of type `T`.

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

```
| ?- edge(X,Y).
↪ X=a,Y=b; X=b,Y=c; X=c,Y=a.
```

```
run[(String,String)] {
  case Pair(x,y) => edge(x,y) }
↪ pair(a,b); pair(b,c); pair(c,a).
```

In Scalogno, we apply the `edge` relation like any ordinary function. The `run` form serves as an interface between the host and the embedded language, returning an answer list of reified values of the variable it scopes. Here, we directly use pattern matching to introduce the variables `x` and `y` as a pair. We can also use the `exists` form to explicitly introduce new logical variables in scope, as in the next example.

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

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

```
def path(x: Exp[T], y: Exp[T]):
  Rel = edge(x,y) ||
  exists[T] { z => edge(x,z) &&
       path(z,y) }

runN[String](10) { q =>
  path("a",q) }
↪ b; c; a; b; c; a; b; c; a; b.
```

```
| ?- path(a,Q).
↪ Q=b; Q=c; ...
```

Here, asking for all answers (with `run` instead of `runN`) would diverge as there are infinitely many paths through the cycle. In Section 3, we show how to cope with this divergence by changing the evaluation semantics through metaprogramming.

2.2 Higher-Order Relations as Higher-Order Functions

In Scalogno, we can exploit higher-order functions (and hence, relations too), for example parameterizing the relation `path` by the relation `edge` so that it works for any graph:

```
def generic_path(edge: (Exp[T],Exp[T]) => Rel)(x: Exp[T], y: Exp[T]): Rel =
  edge(x,y) || exists[T] { z => edge(x,z) && generic_path(edge)(z,y) }
```

We could also recognize that the `path` relation is really just the reflexive transitive closure of the `edge` relation, and since `generic_path` is already parameterized over an arbitrary binary relation, rename it accordingly as `refl_trans_closure`. This enables defining `path` as:

```
val path = refl_trans_closure(edge)
```

The usual higher-order combinators, such as `map`, `flatMap`, and `fold` also have natural higher-order relational counterparts.

2.3 Object-Oriented Encapsulation

To enable additional abstractions that are not present in typical logic programming settings, we can exploit the object-oriented features of the host language:

```
trait Graph[T] {
  def edge(x: Exp[T], y: Exp[T]): Rel // left abstract
  def path(x: Exp[T], y: Exp[T]): Rel = // defined as before
    edge(x,y) || exists[T] { z => edge(x,z) && path(z,y) }}
val g = new Graph[String] {
  def edge(x:Exp[String],y:Exp[String]) = // defined as before
    (x === "a") && (y === "b") ||
    (x === "b") && (y === "c") ||
    (x === "c") && (y === "a") }
```

The object `g` inherits the definition of `path` from `Graph`.

We can also use the pattern known as 'type classes as objects and implicits' [28], for example to support a relational ordering on polymorphic lists.

3 Dynamic Scope as Meta-Interpreter (Design Pattern)

Here is the recipe for Prolog-style meta-interpreters in Scala: a meta-interpreter (a Scalogno relation itself) is configured with a Scalogno meta-relation to build a reified representation of a Scalogno object-relation (e.g. `path`). In other words, we stay completely in the realm of logic programming.

In this section, we consider a different approach: use the host language to augment the execution of logic programs by customizing the logic engine directly. For this approach to be viable, the logic embedding has to be designed with certain kinds of extensions in mind. Within Scalogno, for example, it is difficult to use mutable state because the execution order uses various flavors of interleaving, as opposed to Prolog's deterministic Selective Linear Definite (SLD) clause resolution. But of course interleaving is desirable, so we would like a model that supports a notion of "thread local" state that is attached to a particular execution path, similar to notions of state in Or-parallel logic programming [16].

3.1 Designing Logic Engines for Meta-Programming

In designing the Scalogno implementation, we have put emphasis on modularity and enabling independent extensions of different parts of the system. An overview of the core Scalogno system is shown in Figure 1, and we discuss individual aspects step by step below.

Our starting point is an implementation of a Depth-First Search (DFS) engine, where we reuse the host control flow (stack and exception) to manage the pending goals. Nevertheless, Scalogno is modular and supports a range of search strategies, as well as external solvers.

The engine knows generically about goals and their state. A goal is represented as a thunk of a relation. A relation knows how to execute itself via the

```
val Backtrack = new Exception

// dynamically scoped variables
var dvars: immutable.Map[Int, Any] =
      immutable.Map.empty
case class DVar[T](val id: Int,
    val default: T) extends (() => T) {
  dvars += id -> default
  def apply()  = dvars(id).asInstanceOf[T]
  def :=(v: T) = dvars += id -> v }
var dvarCount = 1
def DVar[T](v: T): DVar[T] = {
  val id = dvarCount
  dvarCount += 1
  new DVar[T](id, v) }

// goals and relations
trait Rel {
  def exec(call: Exec)(k: Cont): Unit }
type Exec = Goal => Cont => Unit
type Cont = () => Unit
type Goal = () => Rel

// unconditional ...
val Yes = new Rel { // ... success
  def exec(call: Exec)(k: Cont) = k() }
val No = new Rel {  // ... failure
  def exec(call: Exec)(k: Cont) =
      throw Backtrack }

def infix_&&(a: => Rel, b: => Rel): Rel =
  new Rel { def exec(call: Exec)(k: Cont) =
    call(() => a) { () => call(() => b)(k) }}
def infix_||(a: => Rel, b: => Rel): Rel =
  new Rel { def exec(call: Exec)(k: Cont) = {
    call(() => a)(k); call(() => b)(k) }}

// logic terms
case class Exp[T](id: Int)
var varCount: Int = 0
def freshId = { var id = varCount;
  varCount += 1; id }
def fresh[T] = Exp(freshId)
```

```
def exists[T](f: Exp[T] => Rel): Rel =
     f(fresh)
def infix_===[T](a: => Exp[T],
                 b: => Exp[T]): Rel = {
  register(IsEqual(a,b)); Yes }
def term[T](key: String, args:
           List[Exp[Any]]): Exp[T] = {
  val e = fresh[T]
  register(IsTerm(e.id, key, args))
  e }

// constraints
abstract class Constraint
case class IsTerm(id: Int, key: String,
                  args: List[Exp[Any]])
    extends Constraint
case class IsEqual(x: Exp[Any], y: Exp[Any])
    extends Constraint

var cstore: immutable.Set[Constraint] =
    immutable.Set.empty
def conflict(cs: Set[Constraint],
             c: Constraint): Boolean = ...
def register(c: Constraint): Unit = {
  if (cstore.contains(c)) return
  if (conflict(cstore,c)) throw Backtrack }

// execution (depth-first)
def run[T](f: Exp[T] => Rel): Seq[String] = {
  def call(e: => Rel)(k: Cont): Unit = {
    // save state
    val cstore1 = cstore; val dvars1 = dvars
    try { e.exec(call)(k)
    } catch { case Backtrack => // OK
    } finally { // restore state
      cstore = cstore1; dvars = dvars1 }}
  val q = fresh[T]
  val res = new mutable.ListBuffer[String]()
  call(() => f(q)) { () =>
    res += extractStr(q) }
  res.toList }
def runN[T](max: Int)(f: Exp[T] => Rel):
     Seq[String] = ...
```

Fig. 1. Scalogno engine implementation

`exec` method, given an executor engine `call` for solving subgoals and a success continuation `k` for returning satisfied. Failure is achieved through throwing a `Backtrack` exception, to backtrack.

Before showing the engine, it's helpful to see a few primitives and means of combination for relations. Unconditional success, `Yes`, immediately successfully continues. Unconditional failure, `No`, immediately throws. The conjunction of two goals, `&&`, executes the first, and successfully continues with the second. The disjunction of two goals, `||`, executes the first, and thereafter through backtracking as defined by the delimited subcall, the second. These goal combinators make use of call-by-name parameters (denoted by an `=>` after the `:` and before the parameter type in Scala).

Finally, our DFS engine, in `call`, pushes the current state on to the stack, runs the goal delegating execution to the underlying relation, catches failures and restores the state upon recursive exits.

This engine cannot do much, because we do not have any constraints to solve yet. So let us introduce a domain of terms, and equality constraints between terms.

A term is uniquely identified. A term constraint `IsTerm(id,key,args)` identifies a term `id` as being bound to a value `key(args)`. An unbound term corresponds to a free logic variable. An equality constraint `IsEqual(x,y)` is introduced by unification, enforcing that two terms, `x` and `y`, have the same structure, that is the same keys and, recursively, arguments.

We define new relations using our constraints. The form `exists` takes a query—a goal with a hole—and fills the hole with a fresh variable. The form `===` unifies two terms by registering an equality constraint with the solver. The form `term` introduces a new term also through constraint registration.

This style of "sea of nodes" construction by side effects is reminiscent of multi-stage programming frameworks like LMS [30]; we will have more to say about this in Section 3.4.

We package the core engine in a runnable interface, which takes a pseudo-goal rather than a thunk, but parameterized by a free logic variable, which is the query variable. The interface `runN` caps the number of returned answers to a given maximum, while `run` is intended to return all answers. (We could also have used a streaming interface.)

We simplistically reify answers into strings. Using polytypic typing as discussed in Section 2, we could improve the model to reify depending on the type of the query variable.

For conflict detection, we keep track of the transitive closure of the set of constraints registered. One can implement a number of performance improvements, including index structures that enable more efficient lookup and matching of constraints.

If we abstract a solver interface, in particular, how state is pushed and restored in the engine, it becomes easy to interface with external SMT solvers.

We are now ready to run some queries.

```
def e(x: Any) = term(x.toString, Nil)
run[Any]{q => q === e(1) || q === e(2)}
↪ List(1,2)
```

As a summary, going back to the basics, what is the essence of a logic programming system? The two main components are 1) search, i.e., nondeterministic execution, and 2) unification and constraints. We implement nondeterministic execution using continuation-passing style (CPS). The class `Rel` comes with implementations for disjunctions and conjunctions, but can be extended for other execution patterns. Method `run` uses an auxiliary `call` to execute individual relations, and the `exec` method of a `Rel` object can invoke its parameter `call` to invoke other relations. The Depth-First Search (DFS) implementation of `call` passes itself to `Rel.exec`. A Breadth-First Search (BFS) implementation would

pass a different method that would just collect the calls in a list. This BFS engine just needs to override the `run` method but can share all other code with the DFS implementation.

The handling of constraints and unification is only sketched in Figure 1. It is a conscious design choice to keep constraints and execution separate as far as possible. The benefit is that both aspects can be extended independently. We model the constraint store `cstore` as a dynamic variable, which keeps its value in a particular execution path (see Section 3.2 below). Invoking the infix method `===` on a logic term registers and checks a new constraint on its arguments in the constraint store of the current execution path.

3.2 An Alternative to Reification and Interpretation

Among the usual use cases for meta-interpreters we find tracing, proof trees and similar extensions. What they all have in common is that they augment a vanilla interpreter to thread a piece of state through the execution.

Let us consider how we can implement such functionality without an explicit meta-interpreter, taking tracing as example. Instead of threading state, we can just use mutable state directly. However there is a catch: we cannot directly use a mutable variable in Scala, because we need to keep apart the state from different nondeterministic branches.

In Scalogno, we provide an abstraction for this: mutable variables with dynamic extent (`DVar`). In contrast to meta-interpreters, these variables can exist side by side, so we can have multiple independent extensions at the same time. Intuitively, dynamic variables have the same extent as the substitution map in miniKanren [6] and the constraint store in cKanren [1], and they correspond to certain realizations of mutable state in Or-parallel logic programming [16].

3.3 Tracing with Dynamic Variables

In the simplest case, we can directly modify the relation we are interested in monitoring:

```
val globalTrace = DVar(nil: Exp[List[List[String]]])
def path(x: Exp[T], y: Exp[T]): Rel = {
  globalTrace := cons(term("path",List(x,y)), globalTrace())
  edge(x,y) || exists[T] { z => edge(x,z) && path(y,b) }}
```

But of course this approach is not very modular. Instead, we can introduce a generic abstract operator for named rules:

```
def rule[T,U](s: String)(f: (Exp[T],Exp[U]) => Rel): (Exp[T],Exp[U]) => Rel
```

Now, we modify the `path` relation to explictly use this `rule` abstraction to indicate that we are indeed defining a named relation, as opposed to just a meta-language abstraction:

```
def path: (Exp[T],Exp[T])=> Rel = rule("path") { (x,y) =>
  edge(x,y) || exists[T] { z => edge(y,z) && path(z,y) }}
```

Instead of modifying the relation directly, we can also build a subclass of `Graph`:

```
trait TracingGraph[T] extends Graph[T] {
  override def path(x:Exp[T],y:Exp[T]) = rule("path")(super.path)(x,y) }
```

In order to implement the actual tracing logic, we define an implementation of the abstract interface as a trait which defines the `rule` method as follows. In Scala, we can mix in this behavior with the otherwise default implementation of the logic engine. We keep the global trace in a variable with dynamic extent.

```
val globalTrace = DVar(nil: Exp[List[List[String]]])
def rule[T,U](s: String)(f: (Exp[T],Exp[U]) => Rel): (Exp[T],Exp[U]) =>
    Rel = { (a,b) =>
  globalTrace := cons(term(s,List(a,b)), globalTrace())
  f(a,b) }
```

We get the same result we would expect:

```
runN[(String,List[String])](5) {
  case Pair(q1,q2) => g.path("a",q1) && globalTrace() === q2 }
↪ pair(b,cons(path(a,b),nil));
   pair(c,cons(path(b,c),cons(path(a,c),nil))); ...
```

We have identified a general design pattern: many meta-interpreters just thread a piece of state. By adding support for this pattern to our engine, we have achieved an alternative implementation approach that removes the need for an entire class of explicit interpreters.

3.4 Clause Reification as Controlled Side Effect

While we have seen that we can often achieve the desired meta-programming effects without explicit meta-interpreters, we may still want explicit interpreters in certain cases. With this goal in mind, we demonstrate another use of dynamic scope: turning logic programs into program generators.

Since we do not want to interpret the meta-language, we need to leverage regular program execution. What can we do? We augment what the program does when run. In an impure language we would use side effects, in a judicious and very controlled way [31]: a `reflect` operation would emit code as side-effect, and a `reify` operation would accumulate code that was produced in its scope. This multi-stage evaluation mechanism is used in program generation frameworks such as LMS [30]. A simple example would be the following:

```
def const(x: Int) = x.toString
def plus(x: String, y: String)  = reflect(s"$x + $y")
def times(x: String, y: String) = reflect(s"$x * $y")
reify { plus(times(const(2), const(3)), times(const(4), const(5))) }
```

↪

```
"val x1 = 2 * 3
 val x2 = 4 * 5
 val x3 = x1 + x2
 x3"
```

Each individual reflected expression generates a val binding, captured by the nearest enclosing reify. The underlying implementation of reify and reflect can be as simple as this:

```
var code: Code
def reify(f: => String) = {
  val temp = code; code = ""
  val res = f
  try (code + res) finally code = temp }
def reflect(rhs: String) = {
  val id = fresh
  code += s"val $id = $rhs\n"
  id }
```

Note how `reify` sets and resets `code` based on the dynamic scope.

How can we adapt this idea to our logic settings? In the place of strings we use a list of goals to accumulate generated terms, based on a dynamic variable to manage scope. The implementation to reflect and reify goals is as follows:

```
val moregoals = DVar(fresh[List[Goal]])
def reifyGoals(goal: => Rel)(goals: Exp[List[Goal]]): Rel = {
  moregoals := goals
  goal && moregoals() === nil }
def reflectGoal(goal: Exp[Goal]): Rel = {
  val hole = moregoals()
  moregoals := fresh
  hole === cons(goal,moregoals()) }
reifyGoals(reflectGoal("path(a,b)") ↪ "cons(path(a,b),nil)"
```

We maintain a global list of clauses, and we can reify clauses given a goal:

```
var allclauses = Map[String,Clause]()
def reifyClause(goal: => Rel)(head: Exp[Goal], body: Exp[List[Goal]]):
  Rel = reifyGoals(goal)(cons(head,nil)) &&
        allclauses(extractKey(head))(head,body)
run[List[Any]] { q =>
  exists[Goal,List[Goal]] { (head,body) =>
    q === cons("to prove", cons(head, cons("prove", cons(body, nil)))) &&
    reifyClause(path(fresh,fresh))(head,body) }}
```

↪

```
cons(to prove,cons(path(a,b),cons(prove,cons(nil,nil)))),
cons(to prove,cons(path(b,c),cons(prove,cons(nil,nil)))),
cons(to prove,cons(path(c,a),cons(prove,cons(nil,nil)))),
cons(to prove,cons(path(a,x0),cons(prove,cons(cons(path(b,x0),nil),nil)))),
cons(to prove,cons(path(b,x0),cons(prove,cons(cons(path(c,x0),nil),nil)))),
cons(to prove,cons(path(c,x0),cons(prove,cons(cons(path(a,x0),nil),nil))))
```

We use the same `rule` abstraction as in the previous section to denote named rules. It adds the clause definition to the global table and reflects the goal as a side effect.

```
def rule[A,B](s: String)(f:(Exp[A], Exp[B]) => Rel) = {
  def goalTerm(a: Exp[A], b: Exp[B]) = term[Goal](s,List(a,b))
  allclauses += s -> { (head: Exp[Goal], body: Exp[List[Goal]]) =>
    exists[A,B] { (a,b) =>
      (head === goalTerm(a,b)) && reifyGoals(f(a,b))(body) }}
  (a: Exp[A], b: Exp[B]) => reflectGoal(goalTerm(a,b))}
```

Finally, we adapt a vanilla interpreter to this new model. This interpreter matches the head of the goal against the global clause table, turned into a disjunction.

```
def allclausesRel: Clause = { (head: Exp[Goal], body: Exp[List[Goal]]) =>
  allclauses.values.foldLeft(No:Rel)((r,c) => r || c(head,body)) }
def vanilla(goal: => Rel): Rel =
  exists[List[Goal]] { goals => reifyGoals(goal)(goals) && vanilla(goals) }
def vanilla(goals: Exp[List[Goal]]): Rel =
  goals === nil || exists[Goal,List[Goal],List[Goal]] { (g, gs, body) =>
    (goals === cons(g,gs)) && allclausesRel(g,body) && vanilla(body) &&
        vanilla(gs) }
```

In the same way, we can implement any other meta-interpreter, such as a tracing interpreter.

4 Tabling as an Alternative Execution Strategy

In this section we show how to implement an alternative evaluation strategy. In functional languages, memoization is a well-known way to speed up computations by reusing intermediate results. The logic programming analogue is known as tabling.

We will implement a `memo` combinator below that can be used as follows to designate particular relations to be tabled:

```
def fib(x:Exp[Int], y:Exp[Int]): Rel = memo(term("fib",List(x,y))) {
  (x === 0) && (y === 1) || (x === 1) && (y === 1) || {
    val x1,x2,y1,y2 = fresh[Int]
    (x === succ(x1)) && (x === (succ(succ(x2)))) &&
    fib(x1,y1) && fib(x2,y2) && add(y1,y2,y) }}
```

The tabled version of `fib` will only compute a linear number of recursive calls instead of an exponential number.

4.1 Implementation: Meta-Programming via the Host Language

Tabling is one of the cases that can not be implemented by a purely declarative meta-interpreter. Instead, imperative features have to be used. Common Prolog implementations are quite intricate, although the concept is simple. The core is

described nicely by Warren [37], which we take as blueprint for our implementation, shown in Figure 2. The evaluation of a logic program forms a search tree for solutions. We can think of exploring this tree either as a nondeterministic process, or as a set of concurrent deterministic processes. In this latter view, multiple processes are active at the same time. When one process reaches a choice point it forks into two new ones, and when it reaches a failure condition, it terminates.

To add tabling or memoization, the first step is to add a global table `callTable` that keeps track of every call to a memoized rule and all the answers returned for it. In contrast to standard functional memoization, though, there may be any number of answers for each call. An *answer* in this context consists of additional constraints that will be applied to the goal as a side effect of executing the rule (details elided in Figure 2). For example, the answer to the goal `fib(5,x0)` will be `fib(5,8)` or equivalently the effect of applying constraint `x0=8` to the goal.

When a process is about to call a memoized rule, it checks the global call table to see if the call has already been made. If not, it adds its continuation to the table and continues evaluating the rule body. When the process is about to return from the call—and this may happen multiple times if the process is forked—then it records the answer it has just computed and resumes all continuations registered for this call with this new answer. If the answer is already in the table, then it is a duplicate, and the process terminates.

When a process calls a memoized rule and the call is already in the table, then the current continuation is invoked once for each recorded answer. The continuation is also registered in the table, since we cannot know if computation of answers has already finished. More answers may become available in the future, and will trigger this continuation again.

4.2 Memoization with Symbolic State Transitions

A key question is how our tabling combinator interacts with state. As a first approximation, we make the input and output state of each call explicit by collecting the values of all dynamic variables. We thus represent a call such as `path(a,b)` as `goal(path(a,b),state0(x0..),state1(x1..))`, where `x0..` are the dynamic variables before the call, and `x1..` the dynamic variables after the call. In other words, we make the state transformation explicit.

However, straightforwardly memoizing these augmented goals would not lead to the desired result. State is often used to accumulate extra contextual information, so it changes all the time. It is rare that a rule is called twice in *exactly* the same state and we would like to be sure that adding a piece of state to the program should not change the memoization behavior.

For this reason, we memoize not based on the augmented goals but on the call patterns only, ignoring input and output state. But how can we describe a rule's state modification independent of a particular input state? To achieve this, we evaluate rule bodies with a *fresh* input state to obtain a *symbolic representation* of the rule's state modification. Implementation-wise, this is easy to achieve because we already maintain a global table of dynamic variables (`dvars`

```
// call table data structures and management
type Answer = (Exp[Any] => Unit)
case class Call(key: String, ...) { ... }
def makeCall(goal: Exp[Any], k: Cont): Call = ...
def makeAnswer(goal: Exp[Any]): Answer = ...
val callTable = new mutable.HashMap[String, mutable.HashMap[String,
    Answer]]
val contTable = new mutable.HashMap[String, List[Call]]
// tabling combinator
def memo[A,B](goal: Exp[Any])(a: => Rel): Rel = new Rel {
  override def exec(call: Exec)(k: Cont): Unit = {
    def resume(cont: Call, ans: Answer) = ...
    val cont = makeCall(goal, k)
    callTable.get(cont.key) match {
      case Some(answers) =>           // call key found:
        for (ans <- answers.values)   //   continue with stored answers
          resume(cont, ans)
      case None =>                    // call key not found:
        val answers = new mutable.HashMap[String, Answer]
        callTable(cont.key) = answers //   add call table entry
        call { () =>
          cont.updateStateBeforeCall()
          a                           //   execute
        } { () =>
          cont.equateStateAfterCall()
          val ansKey = extractStr(goal)
          if (!answers.contains(ansKey)) {
            val ans = cont.makeAnswer()
            ansMap(ansKey) = ans      //   record new answer and
            for (cont1 <- cont.table) //   resume stored continuations
              resume(cont1, ans)
}}}}}
```

Fig. 2. Tabling combinator implementation. Continuations and answers are memoized in global tables.

in Figure 1). Before evaluating the body of a memoized rule, we replace all `dvars` entries with fresh logic variables, which enables us to observe the effects on them when an answer is produced. When resuming a continuation, the symbolic effects need to be unified with the current valuations of the dynamic variables.

With this mechanism in place, we can generate the following answer term for our example of tracing a `path` relation in a graph:

`goal(path(a,b),state0(x0),state1(cons(path(a,b),x0))),`

This term makes explicit that the state after the call—that is, the augmented trace—is the state before the call `x0`, with the current head `cons`ed in front.

Using logic variables to abstract over the state before and after the call ensures that we can represent any kind of relation between the two states that

can be modelled through matching terms. So dropping an element from the front of a list would be easy (match on cons on the left-hand side), recursive predicates such as removing from the middle of collection would be harder.

4.3 Example: Tabled Graph Evaluation

We first note that, as expected, tabling enables left as well as right recursive relations:

```
def pathL(a: Exp[String], b: Exp[String]): Rel =
    memo(term("path",List(a,b))) {
  edge(a,b) || exists[String] { z => pathL(a,z) && edge(z,b) }
```

Furthermore, we can combine tabling with tracing:

```
val globalTrace = DVar(nil: Exp[List[List[String]]])
def pathLT(a: Exp[String], b: Exp[String]): Rel =
    memo(term("path",List(a,b))) {
  globalTrace := cons(term("path",List(a,b)), globalTrace())
  edge(a,b) || exists[String] { z => pathLT(a,z) && edge(z,b) }}
```

And we can verify that the combination works as we would expect. Here is an example query:

```
run[(String,List[String])] { case Pair(q1,q2) => pathLT("a",q1) &&
    globalTrace() === q2 }
↪
pair(b,cons(path(a,b),nil))
pair(c,cons(path(a,b),cons(path(a,c),nil)))
pair(a,cons(path(a,b),cons(path(a,c),cons(path(a,a),nil))))
```

As we can see, the mutable variable `globalTrace` behaves in the way we would expect, recording paths `ab`, `abc`, and `abca` even though we have drastically changed the evaluation order. Here is the execution trace:

```
    goal(path(a,x0),state0(x1,nil),state1(x2,x3))
⟶ goal(path(a,b),state0(x0,x1),state1(x2,cons(path(a,b),x1)))

    goal(path(a,x0),state0(x1,cons(path(a,x2),x3)),state1(x4,x5))
⟶ goal(path(a,b),state0(x0,x1),state1(x2,cons(path(a,b),x1)))

    goal(path(a,x0),state0(x1,nil),state1(x2,x3))
⟶ goal(path(a,c),state0(x0,x1),
    state1(x2,cons(path(a,b),cons(path(a,c),x1))))

    goal(path(a,x0),state0(x1,cons(path(a,x2),x3)),state1(x4,x5))
⟶ goal(path(a,c),state0(x0,x1),
    state1(x2,cons(path(a,b),cons(path(a,c),x1))))

    goal(path(a,x0,state0(x1,nil),state1(x2,x3))
⟶ goal(path(a,a),state0(x0,x1),
    state1(x2,cons(path(a,b),cons(path(a,c),cons(path(a,a),x1)))))
```

```
    goal(path(a,x0),state0(x1,cons(path(a,x2),x3)),state1(x4,x5))
⟶ goal(path(a,a),state0(x0,x1),
    state1(x2,cons(path(a,b),cons(path(a,c),cons(path(a,a),x1)))))
```

Note how `state1` is expressed in terms of `state0`: the first component of `state0`/`state1` is ignored because dynamic var 0 is used internally—dynamic var 1 is the trace.

4.4 Application: Definite Clause Grammar (DCG)

A well-known application of tabling is to turn parsing in logic programming from naive recursive descent strategies to more efficient strategies, variants of Earley's and chart parsing algorithms. As a case study, we consider an example of parsing an arithmetic expression from prior work on tabling in Prolog [7]:

```
expr(S0, S) :- expr(S0, S1), S1 = [+| S2 ], term(S2, S).
expr(S0, S) :- term(S0, S).
term(S0, S) :- term(S0, S1), S1 = [*| S2 ], fact(S2, S).
term(S0, S) :- fact(S0, S).
fact(S0, S) :- S0 = [ '(' | S1 ], expr(S1, S2), S2 = [ ')' | S ].
fact(S0, S) :- S0 = [ N | S ], integer(N).
```

Notably, the grammar is left-recursive, so we cannot use it as a parser in regular Prolog as the standard depth-first resolution strategy would go into an infinite loop. However, in an implementation that supports tabling, the following works and produces expected results:

```
? - expr ([3 , + , 4 , *] , []). ↪ no
? - expr ([3 , + , 4 , * , 7] , []). ↪ yes
? - expr ([ '(' , 3 , + , 4 , ')' , * , 7] , []). ↪ yes
? - E = [_ ,_ ,_ ,_ ,_ ,_ ,_ , _ ] , expr (E , []). ↪ no
```

The Prolog grammar above translates to Scalogno with tabling as follows:

```
def exp(s0: Exp[List[String]], s: Exp[List[String]]): Rel =
    memo(term("exp", List(s0,s))) {
  { val s1,s2 = fresh[List[String]]
    exp(s0,s1) && (s1 === cons("+",s2)) && trm(s2,s) } ||
  trm(s0, s) }
def trm(s0: Exp[List[String]], s: Exp[List[String]]): Rel =
    memo(term("trm", List(s0,s))) {
  { val s1,s2 = fresh[List[String]]
    trm(s0,s1) && (s1 === cons("*",s2)) && fct(s2,s) } ||
  fct(s0, s) }
def fct(s0: Exp[List[String]], s: Exp[List[String]]) = memo(term("fct",
    List(s0,s))) {
  { val s1,s2 = fresh[List[String]]
    s0 === cons("(", s1) && exp(s1, s2) && s2 === cons(")", s) } ||
  { val n = fresh[String]
    s0 === cons(n, s) && dgt(n) }
```

```
}
def dgt(n: Exp[String]) = memo(term("dgt",List(n))) {
  n === "0" || n === "1" || n === "2" || n === "3" || n === "4" ||
  n === "5" || n === "6" || n === "7" || n === "8" || n === "9"
}
```

We obtain the same behavior as in Prolog: without tabling, search diverges, but with the `memo` call in place, we automatically obtain an Earley-style bottom-up parser from the given left-recursive grammar. The embedded setting of Scalogno has the additional advantage that we can easily combine the parser with normal deterministic Scala code that performs IO and/or tokenization:

```
run[List[String]] { q => exp(tokenize("(3+4)*7"), nil) } ↪ x0
```

The result is a single unbounded logic variable that indicates success, without constraining `q`.

5 Related Work

There is a long tradition of meta-programming in Prolog, going back at least to the early 1980s. Warren [36], O'Keefe [27], and Naish [26] 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 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 [32] and HiLog [8] make higher-order programming in those languages awkward. Our host language Scala supports λ-terms and `apply`—we therefore inherit both the expressivity and the complexity of these language features.

According to Martens [23], interest in Prolog meta-interpreters was spurred by two articles [3, 15] from a 1982 collection edited by Clark and Tärnlund. Introductory books on Prolog [34, 27] 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 [19] (and Lloyd's followup language, Escher [22]) is specifically designed for declarative meta-programming. Martens [23] defends 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. Our perspective is that untyped meta-interpreters are clearly useful, as demonstrated by their long history in Prolog; however, when embedding a system similar to

Scalogno in a host language with an expressive static type system (such as Scala, with its type classes), the type system can be put to good use for writing meta-interpreters or achieving similar effects through other means, such as typed variables with dynamic scope. In the spirit of exploiting types but in an orthogonal fashion, OCanren [21] implements an embedding similar to miniKanren while exploiting the type system of OCaml to ensure a well-typed unification from the perspective of the end user.

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 [29, 18, 12], and more recently, in Haskell [20, 33, 9]; to our knowledge, there is no work in the literature on how to best write meta-interpreters for these embedded languages.

6 Conclusion

In this paper, we explored various techniques to meta-program logic programs embedded in a functional host: deep linguistic re-use, reification (of program, and dually, of context), dynamically scoped variables (capturing the common pattern of recording extra information about each run), among others. 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.

Acknowledgments Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR002517, by DARPA under agreement number AFRLFA8750-15-2-0092, by NSF under Awards 1553471, 1564207, 1918483, by the Department of Energy under Award DE-SC0018050, as well as by gifts from Google, Facebook, and VMware. The views expressed are those of the authors and do not reflect the official policy or position of the National Institutes of Health, Department of Defense, Department of Energy, National Science Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Alvis, C.E., Willcock, J.J., Carter, K.M., Byrd, W.E., Friedman, D.P.: cKanren: miniKanren with constraints. In: Scheme (2011)
2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. LNCS, vol. 6806, pp. 171–177 (2011)

3. Bowen, K.A., Kowalski, R.A.: Amalgamating logic and metalanguage in logic programming. In: Clark, K.L., Tärnlund, S.A. (eds.) Logic Programming, pp. 153–172. Academic Press (1982)
4. Byrd, W.E.: Relational Programming in miniKanren: Techniques, Applications, and Implementations. Ph.D. thesis, Indiana University (September 2009)
5. Byrd, W.E., Ballantyne, M., Rosenblatt, G., Might, M.: A unified approach to solving seven programming problems (functional pearl). PACML **1**(ICFP) (2017)
6. Byrd, W.E., Holk, E., Friedman, D.P.: miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In: Scheme (2012)
7. Carro, M., de Guzmàn, P.C.: Tabled logic programming and its applications. `http://cliplab.org/~mcarro/Slides/Misc/intro_to_tabling.pdf` (2011)
8. Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. J. Log. Program. **15**(3), 187–230 (1993)
9. Claessen, K., Ljunglöf, P.: Typed logical variables in Haskell. Electr. Notes Theor. Comput. Sci. **41**(1), 37 (2000), Haskell Workshop
10. Codish, M., Søndergaard, H.: Meta-circular abstract interpretation in Prolog. In: The Essence of Computation. pp. 109–134 (2002)
11. Desouter, B., Van Dooren, M., Schrijvers, T.: Tabling as a library with delimited control. Theory and Practice of Logic Programming **15**(4-5), 419–433 (2015)
12. Felleisen, M.: Transliterating Prolog into Scheme. Tech. Rep. 182, Indiana University Computer Science Department (1985)
13. Friedman, D.P., Byrd, W.E., Kiselyov, O.: The Reasoned Schemer. MIT Press, Cambridge, MA (2005)
14. Friedman, D.P., Byrd, W.E., Kiselyov, O., Hemann, J.: The Reasoned Schemer. MIT Press, Cambridge, MA, second edn. (2018)
15. Gallaire, H., Lasserre, C.: Metalevel control of logic programs. In: Clark, K.L., Tärnlund, S.A. (eds.) Logic Programming, pp. 173–185. Academic Press (1982)
16. Gupta, G., Costa, V.S.: Cuts and side-effects in and-or parallel Prolog. The Journal of logic programming **27**(1), 45–71 (1996)
17. Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. pp. 123–168. LNCS 7797 (2013)
18. Haynes, C.T.: Logic continuations. J. Log. Program. **4**(2), 157–176 (1987)
19. Hill, P.M., Lloyd, J.W.: The Gödel programming language. MIT Press (1994)
20. Hinze, R.: Prological features in a functional setting axioms and implementations. In: FLOPS (1998)
21. Kosarev, D., Boulytchev, D.: Typed embedding of relational language in OCaml. In: 2016 ML Family Workshop (Sep 2016)
22. Lloyd, J.W.: Declarative programming in Escher. Tech. Rep. CSTR-95-013, Department of Computer Science, University of Bristol (June 1995)
23. Martens, B., Schreye, D.D.: Why untyped nonground metaprogramming is not (much of) a problem. Journal of Logic Programming **22**(1), 47–99 (Jan 1995)
24. Martin-Martin, E.: Type classes in functional logic programming. PEPM (2011)
25. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. vol. 4963, pp. 337–340 (2008)
26. Naish, L.: Higher-order logic programming in Prolog. Tech. Rep. 96/2, University of Melbourne (1996)
27. O'Keefe, R.A.: The Craft of Prolog. MIT Press, Cambridge, MA, USA (1990)
28. Oliveira, B.C., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA (2010)

29. Robinson, J.A., Sibert, E.E.: LOGLISP: an alternative to PROLOG. In: Hayes, J., Michie, D., Pao, Y.H. (eds.) Machine Intelligence 10, pp. 399–419. Ellis Horwood Ltd. (1982)
30. Rompf, T.: The essence of multi-stage evaluation in LMS. In: A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 318–335 (2016)
31. Rompf, T., Amin, N., Moors, A., Haller, P., Odersky, M.: Scala-virtualized: linguistic reuse for deep embeddings. Higher-Order and Symbolic Computation **25**(1) (2013)
32. Somogyi, Z., Henderson, F.J., Conway, T.C.: Mercury, an efficient purely declarative logic programming language. In: Proceedings of the Australian Computer Science Conference. pp. 499–512 (1995)
33. Spivey, J.M., Seres, S.: Embedding Prolog in Haskell. In: Meijer, E. (ed.) Proc. of the 1999 Haskell Workshop. Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht (1999)
34. Sterling, L., Shapiro, E.: The Art of Prolog (2nd Ed.): Advanced Programming Techniques. MIT Press, Cambridge, MA, USA (1994)
35. Sterling, L., Yalcinalp, L.U.: Explaining Prolog based expert systems using a layered meta-interpreter. pp. 66–71. IJCAI'89 (1989)
36. Warren, D.H.D.: Higher-order extensions to Prolog: are they needed? In: Hayes, J., Michie, D., Pao, Y.H. (eds.) Machine Intelligence 10, pp. 441–454. Ellis Horwood Ltd. (1982)
37. Warren, D.S.: Memoing for logic programs. Commun. ACM **35**(3), 93–111 (Mar 1992)