Lightweight Functional Logic Meta-Programming

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

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—not so, in 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.

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 is 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 (see Figure 1). 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 [Hanus 2013] 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 [Sterling and Shapiro 1994; Sterling and Yalcinalp 1989]. However, for non-trivial applications such as abstract

Authors' addresses: Nada Amin, Computer Laboratory, University of Cambridge, William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK, first.last@cl.cam.ac.uk; William E. Byrd, Department of Computer Science, University of Alabama at Birmingham, USA, webyrd@uab.edu; Tiark Rompf, Department of Computer Science, Purdue University, 305 N. University Street, West Lafayette, IN, 47907, USA, first@purdue.edu.

interpretation [Codish and Søndergaard 2002], 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 [de Moura and Bjørner 2008] or CVC4 [Barrett et al. 2011], and providing explicit support for dynamically scoped, i.e., "thread-local" state. In the tradition of miniKanren [Byrd 2009a; Byrd et al. 2017, 2012; Friedman et al. 2005, 2018], 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.

We further extend logic programming to more general constraints by delegating solving to SMT (Satisfiability Modulo Theory) engines such as Z3 [de Moura and Bjørner 2008] or CVC4 [Barrett et al. 2011]. We explore multiple ways to combine the power of each style (functional, high-level search, constraint solving).

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 [Martin-Martin 2011] (Section 2).
- We motivate the use of meta-programming in logic programming: it is often desirable 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, a concept which we review and apply to Scalogno using term reification instead of reflective inspection (Section 3).
- 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 4).
- 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 [Desouter et al. 2015]), 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 5).
- We extend constraint solving by delegating queries to an external SMT solver. We show that we can use the DFS search strategy to interactively submit assertions to a solver, and push and pop solver state when backtracking naturally. We discuss issues of committed choice: it is essential to let the solver check satisfiability without committing to a model, as the model might get invalidated further along. We discuss how abstraction layers at the meta-level can be blurred away in the solver, and how deferring committed choice enforces abstraction. We show several simple examples of program synthesis that use the solver (Section 6).

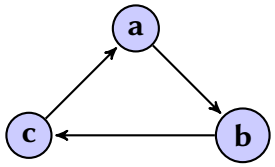
Section 7 discusses related work and Section 8 offers concluding thoughts.

2 EMBEDDED LOGIC PROGRAMMING FOR DEEP LINGUISTIC REUSE

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-use 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 (`&&`) disjunction (`||`). The type `Rel` represents a relation, the type `Exp[T]` 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).

| ?- path(a,Q).
↪ Q=b; Q=c; ...
```

```
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.
```

Here, asking for all answers (with `run` instead of `runN`) would diverge as there are infinitely many paths through the cycle. In Sections 3 and 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 Scalogno, we can exploit higher-order functions (and hence, relations too), for example parametrizing 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`, `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`.

2.4 Type Classes

As another example of deep linguistic reuse, we show how Scalogno benefits from Scala's type classes as objects and implicits [Oliveira et al. 2010].

If we want to capture the property of a datatype to be ordered, we can define a type class interface `Ord[T]`, an infix operation `<` on ordered types, and a type class instance `Ord[Int]` for natural numbers:

```
trait Ord[T] {
  def lt(x:Exp[T],y:Exp[T]): Rel
}
implicit class OrdOps[T:Ord](x:Exp[T]) {
  def <(y:Exp[T]): Rel = implicitly[Ord[T]].lt(x,y)
}
implicit val ordNat = new Ord[Int] {
  def lt(x:Exp[Int],y:Exp[Int]): Rel = ... /*elided*/
}
```

Given our type class instance for natural numbers we can run queries like the following:

```
run[Int] { q => q < 4 }  ↪ 0,1,2,3
```

Internally, this will invoke the `lt` method on `ordNat`.
We now define a lexicographic ordering on polymorphic lists

```
implicit def ordList[T:Ord] = new Ord[List[T]] {
  def lt(as:Exp[List[T]],bs:Exp[List[T]]): Rel =
    exists[T,List[T]] { (b,bs1) =>
      (bs === cons(b,bs1)) && { (as === nil) || exists[T,List[T]] { (a,as1) =>
          (as === cons(a,as1)) && { (a < b) || (a === b) && (as1 < bs1) }}}}
```

The occurrence of `a < b` uses the type class instance `Ord[T]` on list elements, whereas `as1 < bs1` is a recursive call to the relation on `Ord[List[T]]`.

A simple query returns all the lists lexicographically smaller than `List(0,1,2)`. Some lists may have arbitrary length, denoted by a logic var `x0` in the output:

```
run[List[Int]] { q => q < List(0,1,2) }
↪ nil; cons(z,nil); cons(z,cons(z,x0)); cons(z,cons(s(z),nil));
   cons(z,cons(s(z),cons(z,x0))); cons(z,cons(s(z),cons(s(z),x0))).
```

Without going into the details, we can add other data types, like binary trees that implement a dictionary. The only requirement for keys is to be ordered. For example, we can use lists of numbers as keys, and run queries forwards and backwards:

```
run[String] { q => val t = tree(
  List(1,1,1) -> "a", List(1,2,2) -> "b", List(2,1,1) -> "c")
  lookup(t,List(1,2,2),q) } ↪ b.
run[Int] { q => val t = tree(
  List(1,1,1) -> "a", cons(q,List(2,2)) -> "b", List(2,2,2) -> "c")
  lookup(t,List(1,2,2),"b") } ↪ 1.
```

3 THE ESSENCE OF PROLOG-STYLE META-INTERPRETERS

Logic programming enables us to concisely describe relations. For example, Figure 1 presents the typing relation of the simply-typed λ-calculus on paper and in Prolog. The Prolog rules closely follow the paper rules, deviating only by adding a cut (`!`) and `unify_with_occurs_check` to make some informal conventions explicit. In particular, `unify_with_occurs_check` ensures that the result type is finitely expressible, for example preventing self-application. The Prolog relation 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.

Meta-programming enables us to further extend the uses of a logic program, without modifying its concise description. For example, we can turn our Prolog typing relation into a type debugger,

$$\frac{(x : T_A) \in \Gamma}{\Gamma \vdash x : T_A} \quad (\textsc{var})$$

```
in((X,A),G) :- member((X,A),G), !.
ty(G,v(X),A) :- in((X,A),G).
```

$$\frac{\Gamma, (x : T_A) \vdash e_M : T_B}{\Gamma \vdash \lambda x.e_M : T_A \rightarrow T_B} \quad (\textsc{abs})$$

```
ty(G,l(X,M),to(A,B)) :-
  ty([(X,A)|G],M,B).
```

$$\frac{\Gamma \vdash e_M : T_A \rightarrow T_B \quad \Gamma \vdash e_N : T_A}{\Gamma \vdash (e_M\ e_N) : T_B} \quad (\textsc{app})$$

```
ty(G,a(M,N),B) :-
  ty(G,M,C), ty(G,N,A),
  unify_with_occurs_check(C,to(A,B)).
```

Fig. 1. Simply-typed λ-calculus: in formal notation and in Prolog.

that generates derivation trees showing exactly where failures lie. Figure 2 shows such an auto-generated diagnostic.

$$\dfrac{\dfrac{}{[x \mapsto (A \Rightarrow B)\ ] \vdash x : (A \Rightarrow B)}\ \textsc{Var} \quad \dfrac{}{[x \mapsto (A \Rightarrow B)\ ] \vdash x : (A \Rightarrow B)}\ \textsc{Var} \quad (A \Rightarrow B) \neq A}{\dfrac{[x \mapsto (A \Rightarrow B)\ ] \vdash (x\ x) : B}{[] \vdash (\lambda x.(x\ x)) : ((A \Rightarrow B) \Rightarrow B)}\ \textsc{Abs}}\ \textsc{App}$$

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

We now review Prolog meta-interpreters [O'Keefe 1990; Sterling and Shapiro 1994], 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 an embedded setting, with examples in Scalogno, in Section 4.

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-ins for this, but we can also do it manually, and partially select what we reify.

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

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

Here is the Scalogno version of the same manual conversion of the `path` program into a program generator, `path_clause`, that produces the original program as data:

```
def path_clause[T](g: Graph[T])(head: Exp[Goal], body: Exp[List[Goal]]) = {
  exists[T,T] { (a,b) =>
    (head === path_term(a,b)) && {
```

```
      (g.edge(a,b) && (body === nil)) ||
      exists[T] { z =>
        g.edge(a,z) && (body === cons(path_term(z,b),nil)) }}}}
```

Here, the type `Goal` is a phantom type and the function `path_term` serves as constructor for `Exp[Goal]` terms, implemented using Scalogno's primitive term constructor `term` (see Figure 3):

```
trait Goal
def path_term[T](a: Exp[T], b: Exp[T]) = term[Goal]("path",List(a,b))
```

We can observe the behavior of `path_clause` by running a query as follows:

```
run[List[Any]] { q =>
  exists[Goal,List[Goal]] { (head,body) =>
    q === cons("to prove", cons(head, cons("prove", cons(body, nil)))) &&
    path_clause(g)(head,body)
  }
}
↪
  (to prove path(a,b ) prove ());
  (to prove path(b,c ) prove ());
  (to prove path(a,x0) prove (path(b,x0)));
  (to prove path(c,a ) prove ());
  (to prove path(b,x0) prove (path(c,x0)));
  (to prove path(c,x0) prove (path(a,x0))).
```

We can see that, since `edge` has been left as-is, `path_clause` actually produces variants of the `path` rules that are partially evaluated with respect to the `edge` 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 show ways to automate it in Section 4.5.

Reifying both the `edge` and `path` relations gives us a direct data analog of the original Prolog clauses for `path`:

```
def edge_term[T](a: Exp[T], b: Exp[T]) = term[Goal]("edge",List(a,b))
def path_full_clause[T](g: Graph[T])(head: Exp[Goal], body: Exp[List[Goal]]) = {
  exists[T,T] { (a,b) =>
    ((head === path_term(a,b)) && (
      (body === cons(edge_term(a,b),nil)) ||
      exists[T] { z =>
        body === cons(edge_term(a,z), cons(path_term(z,b),nil))
      }
    )) ||
    ((head === edge_term(a,b)) && g.edge(a,b))
  }
}
run[List[Any]] { q =>
  exists[Goal,List[Goal]] { (head,body) =>
    exists[String,String] { (a,b) => pathTerm(a,b) === head } &&
      q === cons("to prove", cons(head, cons("prove", cons(body, nil)))) &&
      path_full_clause(g)(head,body)
  }
}
↪
  (to prove path(x0,x1) prove (edge(x0,x1)));
  (to prove path(x0,x1) prove (edge(x0,x2) path(x2,x1))).
```

We consider a few interpreter implementations for reified clauses next.

3.1 Vanilla Interpreter

The vanilla interpreter takes a `clause` higher-order relation, such as `path_clause`, and returns a solver, which takes a list of reified goals, constraining them to hold. Implementation-wise, the solver is just another logic program: if the list of goals is empty, then we're done; otherwise, we recursively solve the first goal, then the remaining goals. In Prolog, we hard-code the `clause` relation to be `path_clause`, though in usual Prolog meta-interpreters the clauses would come through reflecting on the goal.

```
vanilla([]).

vanilla([G|GS]) :-
  path_clause(G,BODY),
  vanilla(BODY),
  vanilla(GS).
```

```
type Clause = (Exp[Goal],
     Exp[List[Goal]]) => Rel

def
     vanilla(clause:Clause)(goals:Exp[List[Goal]]):Rel
     =
  goals === nil ||
  exists[Goal,List[Goal],List[Goal]]
       { (g, gs, body) =>
    (goals === cons(g,gs)) &&
    clause(g,body) &&
    vanilla(clause)(body) &&
    vanilla(clause)(gs)
  }
```

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.

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

```
runN[String](10) { q =>
  vanilla(path_clause(g))(cons(path_term("a",q),nil))
       }
↪ b; c; a; b; c; a; b; c; a;
     b.
```

3.2 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 `in` and `out` accumulate the current trace using a logic difference list [Clocksin 1997].

```
tracer([], T, T).


tracer([G|GS], IN, OUT) :-
  path_clause(G,BODY),
  tracer(BODY, [G|IN],
      OUT_BODY),
  tracer(GS, OUT_BODY, OUT).
```

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

```
def tracer(clause: Clause)
  (in: Exp[List[Goal]], out:
       Exp[List[Goal]])
  (goals: Exp[List[Goal]]): Rel =
  ((goals === nil) && (in === out))
       ||
  exists[Goal,List[Goal],List[Goal],List[Goal]]
       {
    (g, gs, body, out_body) =>
    (goals === cons(g,gs)) &&
    clause(g,body) &&
    tracer(clause)(cons(g,in),out_body)(body)
        &&
    tracer(clause)(out_body,out)(gs)
  }

runN[(List[Any],List[Goal])](4) {
     case Pair(q,t) =>
  tracer(path_clause(g))(nil,t)(
    cons(path_term("a",q),nil)) }
↪
  pair(b, (path(a,b)));
  pair(c, (path(b,c) path(a,c)));
  pair(a, (path(c,a) path(b,a)
       path(a,a)));
  pair(b, (path(a,b) path(c,b)
       path(b,b) path(a,b))).
```

3.3 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 enforce that the current goal does not occur in the current trace. While there are ways to implement such a constraint purely relationally, for example the `absento` form in miniKanren [Byrd et al. 2012], we postpone a solution in Scalogno to Section 5 because we do not want to extend the purely logical core with extra-logical features. In Prolog, we rely on the *not provable* operator `\+` which implements negation as failure.

```
cycler([], T, T).
cycler([G|GS], T_IN, T_OUT) :-
    path_clause(G,BODY),
    \+ member(G, T_IN),
    cycler(BODY, [G|T_IN], T_OUT_BODY),
    cycler(GS, T_OUT_BODY, T_OUT).

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

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

4 DYNAMIC SCOPE AS META-INTERPRETER (DESIGN PATTERN)

In Section 3, we explored Prolog-style meta-interpreters in Scala: a meta-interpreter (a Scalogno relation itself) is configured with a Scalogno meta-relation (e.g. `path_clause`) to build a reified representation of a Scalogno object-relation (e.g. `path`). In other words, we stayed 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 [Gupta and Costa 1996].

4.1 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 3, 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.

```
type Goal = () => Rel
```

A relation knows how to execute itself, given an executor engine for solving subgoals and a success continuation for returning satisfied.

```
trait Rel { def exec(call: Exec)(k: Cont): Unit }
type Exec = Goal => Cont => Unit
type Cont = () => Unit
```

Failure is achieved through throwing an exception, to backtrack.

```
val Backtrack = new Exception
```

Before showing the engine, it's helpful to see a few primitives and means of combination for relations.

Unconditional success immediately successfully continues.

```
val Yes = new Rel {
  def exec(call: Exec)(k: Cont) = k() }
```

Unconditional failure immediately throws.

```
val No = new Rel {
  def exec(call: Exec)(k: Cont) = throw Backtrack }
```

The conjunction of two goals executes the first, and successfully continues with the second.

```
val Backtrack = new Exception

// dynamically scoped variables
var dvars: immutable.Map[Int, Any] = 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

val Yes = new Rel {
  def exec(call: Exec)(k: Cont) = k() }

val No = new Rel {
  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 = {
    val restore = solver.push()
    try {
      e.exec(call)(k)
    } catch {
      case Backtrack => // OK
    } finally {
      solver.pop(restore)
    }
  }
  val q = fresh[T]
  val res = new mutable.ListBuffer[solver.Model]()
  call(() => f(q)) { () =>
    res += solver.extractModel(q)
  }
  res.toList
}
def runN[T](max: Int)(f: Exp[T] => Rel):
      Seq[solver.Model] = ...

// solver instance (see text below)
val solver: Solver
```

Fig. 3. Scalogno engine implementation

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

The disjunction of two goals executes the first, and thereafter through backtracking as defined by the delimited subcall, the second.

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

```
    call(() => a)(k); call(() => b)(k) }}
```

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

```
def call(g: Goal)(k: Cont): Unit = {
  val restore = solver.push()
  try {
    g().exec(call)(k)
  } catch {
    case Backtrack => // OK
  } finally {
    solver.pop(restore)
  }
}
```

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 constraint `IsTerm(id, key, args)` identifies a logic variable `id` as being bound to the term `key(args)`.

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

```
case class Exp[+T](id: Int)
def fresh[T] = Exp(freshId)

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
```

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.

```
def exists[T](f: Exp[T] => Rel): Rel = f(solver.fresh)
```

The form `===` unifies two terms by registering an equality constraint with the solver.

```
def infix_===[T](a: => Exp[T], b: => Exp[T]): Rel = {
  solver.register(IsEqual(a,b)); Yes }
```

The form `term` introduces a new term also through constraint registration.

```
def term[T](key: String, args: List[Exp[Any]]): Exp[T] = {
  val e = solver.fresh[T];
  solver.register(IsTerm(e.id, key, args)); e }
```

This style of "sea of nodes" construction by side effects is reminiscent of multi-stage programming framework like LMS [Rompf 2016], and we will have more to say about this in Section 4.5.

We package the core engine in a runnable interface, which takes a pseudo-goal, not a thunk, but parametrized by a free logic variable – the query variable. The interfae `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.)

```
def runN[T](max: Int)(f: Exp[T] => Rel): Seq[solver.Model] = {
  val q = solver.fresh[T]
  val res = mutable.ListBuffer[solver.Model]()
```

```
  val Done = new Exception
  try { call(() => f(q)) { () =>
    res += solver.extractModel(q)
    if (res.length >= max) throw Done
  }} catch { case Done => }
  res.toList
}
def run[T](f: Exp[T] => Rel): Seq[solver.Model] = runN(scala.Int.MaxValue)(f)
```

Now, we need to provide a solver. As used by the engine, the interface abstracts over state, fresh, constraint registration and model extraction.

```
abstract class Solver {
  type State
  type Model
  def push(): State
  def pop(restore: State): Unit
  def fresh[T]: Exp[T]
  def register(c: Constraint): Unit
  def extractModel(x: Exp[Any]): Model
}
```

Let us implement a vanilla solver, which keeps track of the transitive closure of the set of constraints registered.

```
class VanillaSolver extends Solver {
  override type State = immutable.Set[Constraint]
  var cstore: immutable.Set[Constraint] = immutable.Set.empty
  override def push(): State = cstore
  override def pop(restore: State): Unit = { cstore = restore }
  override def register(c: Constraint): Unit = {
    if (cstore.contains(c)) return
    if (conflict(cstore,c)) throw Backtrack
  }
  def conflict(cs: Set[Constraint], c: Constraint): Boolean = {
    def prop(c1: Constraint, c2: Constraint)(fail: () => Nothing): List[Constraint] = (c1,c2)
        match {
      ...
      case _ => Nil
    }
    val fail = () => throw Backtrack
    val cn = cs flatMap { c2 => prop(c, c2)(fail) }
    cstore += c
    cn foreach register
    false
  }
  override type Model = String
  def extractModel(x: Exp[Any]): Model = { ... }
}
```

For the model, we can 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. The actual built-in Scalogno solver implements a number of performance improvements, including

index structures that enable more efficient lookup and matching of constraints. Based on this solver interface, it is easy to interface with external SMT solvers. We discuss an example in Section 6.

To complete the high-level interface, we use implicit classes to support infix method syntax in Scala:

```
implicit class RelOps(a: => Rel) {
  def &&(b: => Rel) = infix_&&(a,b)
  def ||(b: => Rel) = infix_||(a,b)
}
implicit class ExpOps[T](a: Exp[T]) {
  def ===[U](b: Exp[U]) = infix_===(a,b)
}
```

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 conjuntions, 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 relation. The Depth-First Search (DFS) implementation of `call` passes itself to `Rel.exec`. A Breath-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 3. 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 (type `DVar`), which keeps its value in a particular execution path (see Section 4.2 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.

4.2 An Alternative to Reification and Interpretation

Among the use cases for meta-interpreters we have considered were 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 [Byrd et al. 2012] and the constraint store in cKanren [Alvis et al. 2011], and they correspond to certain realizations of mutable state in Or-parallel logic programming [Gupta and Costa 1996].

4.3 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.

4.4 Probabilistic Logic Computation

To give another example, dynamic variables can neatly capture probabilistic logic computation: at each random flip, each branch updates the overall probability.

```
val theprob = DVar(1.0)
def flip(p: Double)(a: => Rel)(b: => Rel): Rel =
  { theprob := theprob() * p; a } || { theprob := theprob() * (1.0 - p); b }

runN[(Boolean,Double)](3) { case Pair(c,p) => flip(0.2,c) && { p === theprob() } }
↪ pair(true,0.2), pair(false,0.8)
```

4.5 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 [Rompf et al. 2013]: 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 [Rompf 2016]. 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, captures 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 var 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 the vanilla interpreter from Section 3.1 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 the tracing interpreter from Section 3.2

5 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: (Exp[Int], Exp[Int]) => Rel = memo("fib") { (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.

5.1 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 [1992], which we take as blueprint for our implementation, shown in Figure 4. 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 in additional constraints that will be applied to the goal as a side effect of executing the rule (details elided in Figure 4). 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.

5.2 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.

```
// call table data structures and management
case class Call(key: String) { ... }
case class Answer(key: String) { ... }
def makeCall(name: String, args: Exp[Any]*): Call = ...
def makeAnswer(name: String, args: Exp[Any]*): Answer = ...
val callTable = new mutable.HashMap[String, (mutable.HashSet[Call], mutable.HashSet[Answer])]
// tabling combinator
def memo[A,B](name: String)(f: (Exp[A], Exp[B]) => Rel)(a: Exp[A], b: Exp[B]): Rel = new Rel {
  override def exec(call: Exec)(k: Cont): Unit = {
    def resume(cont: Call, ans: Answer) = ...
    val cont = makeCall(name, a, b)
    callTable.get(cont.key) match {
      case Some((conts, answers)) =>                        // call key found:
        conts += cont                                       //   save continuation for later
        for (ans <- answers.toList) resume(cont, ans)       //   continue with stored answers
      case None =>                                          // call key not found:
        val answers = new mutable.HashSet[Answer]           //   add table entry
        val conts   = new mutable.HashSet[Call]
        callTable(cont.key) = (conts,answers)
        conts += cont                                       //   store continuation
        call { () => f(a,b) } { () =>                       //   execute rule body
          val ans = makeAnswer(name, a, b)
          if (!answers.contains(ans)) {
            answers += ans                                  //   record each new answer and
            for (cont1 <- conts.toList) resume(cont1, ans)  //   resume stored continuations
} } }   } }
```

Fig. 4. Tabling combinator implementation. Continuations and answers are memoized in a global call table.

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` in Figure 3). 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 `consed` in front. A larger example follows.

5.3 Example: Tabled Graph Evaluation

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

```
def pathL: (Exp[String], Exp[String]) => Rel = memo("path") { (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: (Exp[String], Exp[String]) => Rel = memo("path") { (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.

5.4 Application: Definite Clause Grammar (DFG)

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 [Carro and de Guzmàn 2011]:

```
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 expr: (Exp[List[String]], Exp[List[String]]) => Rel = memo("expr") { (s0, s) =>
  { val s1,s2 = fresh[List[String]]
    expr(s0,s1) && (s1 === cons("+",s2)) && term(s2,s) } ||
    term(s0, s) }
def term: (Exp[List[String]], Exp[List[String]]) => Rel = memo("term") { (s0, s) =>
  { val s1,s2 = fresh[List[String]]
    term(s0,s1) && (s1 === cons("*",s2)) && fact(s2,s) } ||
    fact(s0, s) } }
def fact: (Exp[List[String]], Exp[List[String]]) => Rel = memo("fact") { (s0, s) =>
  { val s1,s2 = fresh[List[String]]
    s0 === cons("(", s1) && expr(s1, s2) && s2 === cons(")", s) } ||
  { val n = fresh[String]
    s0 === cons(n, s) && digit(n) } }
def digit: Exp[String] => Rel = memo("digit") {
  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 =>
  expr(tokenize("(3+4)*7"), nil)
}
↪ x0
```

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

6 SYNTHESIS WITH CLP(SMT)

Coming back to end of Section 4.1,we now implement another solver "backend" based on SMT rather than (or in addition to) rolling our own.

SMT solvers such as Z3 and CVC4 implement a common interface SMTLIB based on Lisp S-expression. We write a low-level Scala process which interactively interacts with such a tool. It has the following interface, and a possible implementation is left as an exercise for the reader. Instead of using an interactive mode, we also can use a batch mode; the advantage is that it is not tied into our search strategy (Depth-First Strategy), but is slower.

An important question is when to commit the model back to the engine. A natural idea is to commit the model every time we check satisfiability, but this solution commits us too early to a solution.

A Depth-First Strategy naturally fits with the pushing and popping features available in an SMT solver. So we illustrate the interactive fast mode here, while we've also experimented with other strategies, requiring batch mode like interleaving.

When extracting the model, we make sure to extract a solution from the SMT solver, using the underlying facility. We do this by adding equality constraints.

```
solver.extractModel({(x,v) =>
  register(IsEqual(Exp(x),term(v.toString, Nil)))
})
```

In the engine search execution, pushing and popping also pushes and pops a frame in the SMT engine.

We can now use our extended facility to add relations that target the SMT solver. This is what the assertion relation looks like. The `seenvars` bookkeeping is to ensure we declare the variables in the proper scope when generating SMT queries.

```
def zAssert(p: P[Boolean]): Rel = {
  seenvars = seenvars0
  val c = P("assert", List(p)).toString
  seenvars.foreach{solver.decl}
  solver.add(c)
  if (!solver.checkSat()) throw Backtrack
  Yes
}
```

The key idea is to check satisfiability, but commit to a model late so that we always pick a good one.

Here is a possible domain for dealing with constraints on integers.

```
abstract class Z[+T]
case class A[+T](x: Exp[T]) extends Z[T]
case class P[+T](s: String, args: List[Z[Any]]) extends Z[T]

implicit object InjectInt extends Inject[Int] {
  def toTerm(i: Int): Exp[Int] = term(i.toString,Nil)
}
implicit def int2ZInt(e: Int): Z[Int] = toZInt(InjectInt.toTerm(e))
implicit def toZInt(e: Exp[Int]): Z[Int] = A(e)
implicit def toZIntOps(e: Exp[Int]) = ZIntOps(A(e))
implicit class ZIntOps(a: Z[Int]) {
  def ==?(b: Z[Int]): Rel = zAssert(P("=", List(a, b)))
  def !=?(b: Z[Int]): Rel = zAssert(P("not", List(P("=", List(a, b)))))
  def >(b: Z[Int]): Rel = zAssert(P(">", List(a, b)))
  def >=(b: Z[Int]): Rel = zAssert(P(">=", List(a, b)))
  def -(b: Z[Int]): Z[Int] = P("-", List(a, b))
```

```
  def *(b: Z[Int]): Z[Int] = P("*", List(a, b))
  def +(b: Z[Int]): Z[Int] = P("+", List(a, b))
}
```

6.1 A Few Simple Synthesis Examples

Let's use Scalogno's SMT extensions to solve several simple problems from a blog post by James Bornholt [Bornholt 2018], describing how to use use the Rosette language for program synthesis [Torlak and Bodik 2013]. Bornholt's later examples use a simple arithmetic DSL; Bornholt begins by defining an `interpret` function in Rosette (which is itself embedded in Racket).

```
(define (interpret p)
  (match p
    [(plus a b)  (+ (interpret a) (interpret b))]
    [(mul a b)   (* (interpret a) (interpret b))]
    [(square a)  (expt (interpret a) 2)]
    [_ p]))
```

A call to the `interpret` function uses Rosette as an interface to an underlying SMT solver, which solves the constraints accumulated during interpretation of the arithmetic expression.

```
(interpret (plus (square 7) 3))
↪
52
```

A more interesting use of `interpret` is to perform "angelic execution" [Bodik et al. 2010].For example, we can ask Rosette can find an integer y such that $(y + 2)^2$ evaluates to 25.

```
(solve
  (assert
    (= (interpret (square (plus y 2))) 25)))
```

We can write our own version of `interpret` in Scalogno, as the two-argument relation `interpreto`, rather than as a one-argument function. The `interpreto` relation uses unification rather than pattern matching, logic variables rather than pattern variables, and must unnest the recursive calls, using the transformation from functional program to relational program described in [Byrd 2009b] and [Friedman et al. 2018]. Arithmetic constraints are specified using the `zAssert` extension to Scalogno, and are solved by the underlying SMT solver. A `run` query for the angelic execution example above terminates after returning two values for y, 3 and -7, demonstrating that no other solutions exist.

The resulting `interpreto` relation is written in a low level and verbose style—it is several times longer than the Rosette `interpret` function, and has to explicitly add SMT constraints using `zAssert`, exposing the underlying constraint solving technology. One way to create a higher-level interface to the SMT solver in Scalogno is by writing an interpreter for a Turing-complete programming language as a relation, and recasting synthesis problems in terms of that programming language. This is similar to the idea of the `interpret` function described by Bornholt, but can allow for additional abstraction, such as functions, lists, and recursion. We have extended an interpreter for a Turing-complete subset of Racket—similar in spirit to the relational interpreter described in [Byrd et al. 2017]—with arithmetic operators that are implemented using `zAssert`. The user of the interpreter, however, need not be concerned with the details of how the interpreter implements arithmetic, and can instead write programs in a subset of Racket, but which can contain holes represented by fresh logic variables. Our experience is that this approach provides a very convenient interface to the SMT solver.

For example, the angelic execution problem above can be solved in Scalogno by specifying that the Racket expression

```
(let ((plus (lambda (a b) (+ a b))))
  (let ((mul (lambda (a b) (* a b))))
    (let ((square (lambda (a) (* a a))))
      (square (+ y 2))))),
```

where y is a logic variable representing an integer, evaluates to 25 under the `evalo` relation.

The `evalo` relation also allows us to easily solve Bornholt's first example, which is to find all integers y whose absolute value is 5. Here is Bornholt's Rosette solution.

```
(define (absv x)
  (if (< x 0) (- x) x))

(define-symbolic y integer?)

(solve (assert (= (absv y) 5)))
```

We can solve the same problem in Scalogno by specifying that the Racket expression

```
(let ((absv (lambda (x)
              (if (< x 0) (- x) x))))
  (absv y)),
```

where y is a logic variable representing an integer, evaluates to 5 under the `evalo` relation. As expected, the query produces 5 and -5 as possible values of y. We can solve Bornholt's second problem (showing there is no integer y whose absolute value is less than zero) using a slight variant of the above Racket expression.

We cannot handle Bornholt's last two Rosette examples, our implementation of Scalogno does not currently support universally quantified variables.

7 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 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 Scala supports λ-terms and `apply`—we therefore inherit both the expressivity and the complexity of these language features.

According to Martens and 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 [1995]) is specifically designed for declarative meta-programming. Martens and 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. 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 [Kosarev and Boulytchev 2016] 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 [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.

8 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.

REFERENCES

Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, and Daniel P. Friedman. 2011. cKanren: miniKanren with Constraints. In *Workshop on Scheme and Functional Programming*.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *CAV (Lecture Notes in Computer Science)*, Vol. 6806. Springer, 171–177.

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 339–352. https://doi.org/10.1145/1706299.1706339

James Bornholt. 2018. Building a Program Synthesizer. https://homes.cs.washington.edu/ bornholt/post/building-synthesizer.html. (2018).

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.

William E. Byrd. 2009a. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.

William E. Byrd. 2009b. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages.

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Workshop on Scheme and Functional Programming*.

Manuel Carro and Pablo Chico de Guzmàn. 2011. Tabled Logic Programming and Its Applications. http://cliplab.org/~mcarro/Slides/Misc/intro_to_tabling.pdf. (2011).

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.

Koen Claessen and Peter Ljunglöf. 2000. Typed Logical Variables in Haskell. *Electr. Notes Theor. Comput. Sci.* 41, 1 (2000), 37.

William F. Clocksin. 1997. *Clause and Effect: Prolog Programming and the Working Programmer.* Springer.

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

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science)*, Vol. 4963. Springer, 337–340.

Benoit Desouter, Marko Van Dooren, and Tom Schrijvers. 2015. Tabling as a library with delimited control. *Theory and Practice of Logic Programming* 15, 4-5 (2015), 419–433.

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

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* MIT Press, Cambridge, MA.

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (second ed.). MIT Press, Cambridge, MA.

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.

Gopal Gupta and Vitor Santos Costa. 1996. Cuts and side-effects in and-or parallel prolog. *The Journal of logic programming* 27, 1 (1996), 45–71.

M. Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger.* LNCS 7797, 123–168.

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.

Dmitri Kosarev and Dmitri Boulytchev. 2016. Typed Embedding of Relational Language in OCaml. In *2016 ML Family Workshop.*

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.

Enrique Martin-Martin. 2011. Type Classes in Functional Logic Programming *(PEPM).*

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.

Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes As Objects and Implicits. In *OOPSLA.*

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.

Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World (Lecture Notes in Computer Science)*, Vol. 9600. Springer, 318–335.

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2013. Scala-Virtualized: linguistic reuse for deep embeddings. *Higher-Order and Symbolic Computation* (2013).

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.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013).* ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

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.

David S. Warren. 1992. Memoing for Logic Programs. *Commun. ACM* 35, 3 (March 1992), 93–111.