The mini-prolog Language

Kyle Dewey and Ben Hardekopf

1 Overall Language Design

mini-prolog is a small, Prolog-like logic programming language which has been designed to be simple to understand and implement. It still contains the core features of Prolog, namely:

- Unification as a primitive of computation.
- Backtracking and nondeterministic execution.
- A depth-first search strategy for executing nondeterministic programs.

For the sake of simplicity, certain Prolog features are intentionally missing:

- Negation as failure (\+ or not).
- The cut operator (!).
- Predicates to dynamically alter the clause database, such as assert and retract.
- Meta-programming capabilities such as forall and bagof.
- The standard built-in predicates (e.g., member and append).

The user-facing syntax is intended to be as Prolog-like as possible. Many existing Prolog programs will run unmodified as mini-prolog programs, as long as they do not use the missing features. Any mini-prolog program will run unmodified on a standards-compliant Prolog engine.

2 Internal Syntax

The user-facing, Prolog-like syntax of `mini-prolog` is automatically translated into an simpler internal syntax for execution. This translation is responsible for the following tasks:

- Converting lists and list operations into structures and operations on structures. For example, the list `[1,2,3]` in user-facing syntax is represented in the internal syntax as the structure `cons(1, cons(2, cons(3, nil)))`, as per the standard inductive list definition. This simplifies the language, which need not understand lists directly.
- Lifting clause-local and query-local variables into a list of local variables at the head of each clause or query. This simplifies the language evaluation rules by declaring all variables used in a clause up-front.
- Translating pattern-matching into unification predicates. This allows the language evaluation rules to concentrate on unification and ignore pattern-matching.
- Translating unification between arbitrary values to a series of variable bindings (via the ← operator) and unifications on variables (via the **unify** operator). By permitting unification only between variables, the unification algorithm is greatly simplified.

Here is the definition of the internal syntax:

$$x \in \mathit{Variable} \qquad n \in \mathbb{Z} \qquad \mathit{sym} \in \mathit{Symbol}$$

$$\begin{aligned}
\mathit{prog} \in \mathit{Program} &::= \overrightarrow{\mathit{clause}}\ \mathit{query} \\
\mathit{clause} \in \mathit{Clause} &::= \mathit{sym}(\vec{x_1})\ \{\vec{x_2}\}\ \ {:}{-}\ \ \mathit{body} \\
\mathit{query} \in \mathit{Query} &::= \{\vec{x}\}\ \ {?}{-}\ \ \mathit{body} \\
\mathit{body} \in \mathit{Body} &::= \mathit{body}_1 \wedge \mathit{body}_2 \ \mid\ \mathit{body}_1 \vee \mathit{body}_2 \ \mid\ \textsf{\textbf{check}}\ \mathit{sym}(\vec{x}) \ \mid\ x_1 \equiv x_2 \ \mid\ x_1 \bowtie x_2 \ \mid\ x \leftarrow \mathit{rhs} \ \mid\ \textsf{\textbf{true}} \ \mid\ \textsf{\textbf{false}} \\
\mathit{rhs} \in \mathit{Rhs} &::= n \ \mid\ \mathit{sym}(\vec{x}) \ \mid\ \mathit{exp} \\
\mathit{exp} \in \mathit{ArithExp} &::= x \ \mid\ \mathit{exp}_1 \oplus \mathit{exp}_2 \\
\oplus \in \mathit{ArithOp} &::= + \ \mid\ - \ \mid\ \times \ \mid\ \div \\
\bowtie \in \mathit{RelationalOp} &::= < \mid \leq \mid > \mid \geq \mid = \mid \neq
\end{aligned}$$

The vector notation indicates an ordered sequence; for example, $\overrightarrow{\mathit{clause}}$ means a list of clauses and $\vec{x}$ means a list of variables. A `mini-prolog` program *prog* ∈ *Program* is a list of clauses followed by a query. A clause consists of two parts: (1) a head $\mathit{sym}(\vec{x_1})\ \{\vec{x_2}\}$ that gives the name and parameters of the clause (the $\mathit{sym}(\vec{x_1})$ part, i.e., the clause name *sym* and the list of parameters $\vec{x_1}$) and the set of local variables used in the body of the clause (the $\{\vec{x_2}\}$ part); and (2) the clause body that defines when the clause is true. A query is like a clause except it doesn't have a name or parameters, only local variables and a body. A body is a series of conjunctions ∧ and disjunctions ∨ of the following: (1) **check** $\mathit{sym}(\vec{x})$ calls the given clause with the given arguments (which are all variables), yielding either **true** or **false** depending on what that predicate returns; (2) $x_1 \equiv x_2$ tries to unify the values of the given variables, returning **true** if that is possible, otherwise **false**; (3) $x_1 \bowtie x_2$ checks whether the values of x_1 and x_2 (which are assumed to be numbers) have the given relation; (4) $x \leftarrow \mathit{rhs}$ evaluates *rhs* to a value and binds the result to x; (5) **true** and **false**, which have the obvious meanings. Note that variable values can be numbers or ground terms, but the arithmetic and relational operators only work on numbers.

2.1 Syntax Examples

Here is a `mini-prolog` progam in user-facing (i.e., Prolog-like) syntax that represents ordered binary trees. '_' is a wildcard that matches any argument; it is used for convenience when the body of the clause doesn't need to refer to that argument. '%' indicates a comment, which extends to the end of the line.

```
% allLess/2: Value, List. Ensures that all values in the list are < Value.
allLess(_, []).
allLess(V1, [V2 | Rest]) :-
        V2 < V1,
        allLess(V1, Rest).
```

```
% allGreater/2: Value, List. Ensures that all values in the list are > Value.
allGreater(_, []).
allGreater(V1, [V2 | Rest]) :-
        V2 > V1,
        allGreater(V1, Rest).

% isBST/3: node, LT, GT
% LT - everything the node's value must be <
% GT - everything the node's value must be >
isBST(nodenil, _, _).
isBST(node(Value, Left, Right), LT, GT) :-
        allLess(Value, LT),
        allGreater(Value, GT),
        isBST(Left, [Value | LT], GT),
        isBST(Right, LT, [Value | GT]).
```

Here is the same program after translation to the internal syntax. The Tn variables were introduced during the translation process. Note the absence of built-in lists, pattern-matching, and wildcards.

```
allLess(T1, T2) {T3} :-
        T3 ← nil() ∧
        T2 ≡ T3

allLess(V1, T1) {V2, Rest, T2} :-
        T2 ← cons(V2, Rest) ∧
        T1 ≡ T2 ∧
        V2 < V1 ∧
        check allLess(V1, Rest)

allGreater(T1, T2) {T3} :-
        T3 ← nil() ∧
        T2 ≡ T3

allGreater(V1, T1) {V2, Rest, T2} :-
        T2 ← cons(V2, Rest) ∧
        T1 ≡ T2 ∧
        V2 > V1 ∧
        check allGreater(V1, Rest)

isBST(T1, T2, T3) {T4} :-
        T4 ← nodenil() ∧
        T1 ≡ T4

isBST(T1, LT, GT) {Value, Left, Right, T2, T3, T4} :-
        T2 ← node(Value, Left, Right) ∧
        T1 ≡ T2 ∧
        check allLess(Value, LT) ∧
        check allGreater(Value, GT) ∧
        T3 ← cons(Value, LT) ∧
        check isBST(Left, T3, GT) ∧
        T4 ← cons(Value, GT) ∧
        check isBST(Right, LT, T4)
```

3 Language Implementation

To implement the `mini-prolog` runtime engine, we need a procedure to take a program (i.e., a list of clauses and a query) and try to satisfy the query if possible. Section 3.1 describes this procedure using pseudocode. The pseudocode references several data structures and helper functions which are described in Sections 3.2 and 3.3 respectively.

3.1 Engine Pseudocode

This pseudocode takes a `mini-prolog` program `program` and finds a satisfying assignment of query variables to values (if one exists). When the while loop terminates (if it does), either `env` is empty, which means there is no solution, or `env` maps the query's variables to satisfying values. If we want to find another satisfying assignment, we can just push **false** onto `goalStack` and re-enter the while loop. If we want to find all solutions, we can repeat this process until there are no solutions left.

The case for binding (i.e., $x \leftarrow rhs$) exploits a property guaranteed by the translator, namely that the variable involved maps to a placeholder which has not yet been used.

```
initialize the data structures:
  db is the clause database computed from program
  env is the initial empty environment
  equiv is the initial empty equivalence relation between values
  goalStack is the initial empty stack of goals
  choiceStack is the initial empty stack of choices

set env to newEnv(program.query.vars, [])
push program.query.body onto goalStack

while goalStack is not empty:
  pop the top goal from goalStack
  match the goal with one of the following:

    case $body_1 \wedge body_2$ $\Rightarrow$
      push $body_2$ and then $body_1$ onto goalStack, so $body_1$ is on the top

    case $body_1 \vee body_2$ $\Rightarrow$
      push ($body_2$, env, equiv, goalStack) onto choiceStack
      push $body_1$ onto goalStack

    case $x_1 \equiv x_2$ $\Rightarrow$
      look up $x_1$ and $x_2$ in env to get their values $v_1$ and $v_2$
      update equiv to unify $v_1$ and $v_2$; if they cannot be unified, push false onto goalStack

    case check $sym(\vec{x})$ $\Rightarrow$
      let $\overrightarrow{clause}$ be the value of db($sym$, $|\vec{x}|$), where $|\vec{x}|$ is the number of arguments
      if $\overrightarrow{clause}$ is empty, abort execution
      else:
        push 'restore env' onto goalStack
        look up each argument $x$ in env to get its value $v$
        let $clause_1$ be the first clause in $\overrightarrow{clause}$
        for each remaining clause $clause_i$ from $\overrightarrow{clause}$ in reverse order:
          let envC = newEnv($clause_i$.localVars, $clause_i$.params $zip$ $\vec{v}$), where $\vec{v}$ are the argument values
          push ($clause_i$.body, envC, equiv, goalStack) onto choiceStack
        let envC = newEnv($clause_1$.localVars, $clause_1$.params $zip$ $\vec{v}$)
        set env to envC
        push $clause_1$.body onto goalStack
    case restore envR $\Rightarrow$
      set env to envR

    case $x_1 \bowtie x_2$ $\Rightarrow$
      look up $x_1$ and $x_2$ in env to get their values $v_1$ and $v_2$
      if $v_1$ or $v_2$ is not a number, abort execution
      else if $v_1 \bowtie v_2$ is false, push false onto goalStack

    case $x \leftarrow rhs$ $\Rightarrow$
      look up $x$ in env to get its value $v_1$
      evaluate $rhs$ to get its value $v_2$
      set equiv to equiv$[v_1 \mapsto v_2]$

    case true $\Rightarrow$
      do nothing

    case false $\Rightarrow$
      if choiceStack is empty, set env and goalStack to empty
      else:
        pop ($body$, envC, equivC, goalStackC) from choiceStack
        set env to envC, equiv to equivC, and goalStack to goalStackC
        push $body$ onto goalStack
end while
```

3.2 Data Structures

The engine uses the following data structures: the clause database, the environment, the equivalence relation, the goal stack, and the choice stack. Each data structure is described below.

3.2.1 The Clause Database

The clause database is a map from clause names and their arities to a list of matching clauses. It is used when evaluating a **check** expression to find the relevant clauses being called. The database is created once at the very beginning and never changes during program execution. In the internal syntax example from Section 2.1, the clause database would map `allLess/2` to a list of two clauses, `allGreater/2` to a list of two clauses, and `isBST/3` to a list of two clauses. It is important that for each *(name, arity)* entry in the clause database, the relevant clauses are listed in the order that they appear in the program—this gives the programmer control over the order the clauses are evaluated in.

3.2.2 The Environment

The environment maps variables to values. There are no global variables; all variables are local to a particular clause. Clauses can be recursive—just as for recursive functions in imperative programs, the local variables for one instance of the clause are distinct from the local variables of any other instance of that clause. We ensure this by using a different environment for each clause invocation. The new environments are created when evaluating a **check** expression; how to create a new environment for a clause invocation is explained in Section 3.3.2. This is also why we have added the goal **restore** `env` in addition to the regular goals; the restore goal is used to restore the original environment after we've finished evaluating a **check** expression and don't need its environment anymore.

Values are either (1) numbers; (2) ground terms; or (3) placeholder values. Ground terms are explained in Section 3.3.3. Placeholder values are an implementation convenience—it is useful to ensure that variables are always mapped to some value, even if we don't know what that value should be. A placeholder value is essentially a dummy value that stands for some unknown real value (either a number or ground term). When we create new environments, we always map all of the local variables to placeholder values; this makes binding and unification simple because we don't have to worry about special cases that depend on whether a variable has been given a value already or not.

Note that when we look up a variable in the environment to get its value, rather than just return the value directly we should map that value to its set representative using the equivalence relation on values (as explained in the next subsection, 3.2.3) and return that set representative.

3.2.3 The Equivalence Relation

There are two ways to give variables values: either binding (via $\leftarrow$) or unification (via $\equiv$). Because of the way we're using placeholder values, we can implement both very simply by (1) evaluating the left- and right-hand sides to values; and (2) using unification to try and put those values in the same equivalence class. The process of unification is explained in Section 3.3.1. Here we simply describe the data structure we use to record equivalence classes.

An equivalence relation is a partition of values into equivalence classes; in other words, it groups values into sets such that each value is in exactly one set and all values in the same set are equivalent to each other. A given equivalence class is denoted by its *set representative*, i.e., a distinguished member of the set that stands for all of the other members. The equivalence relation data structure is used to find, for each value, what its particular set representative is. This means that the equivalence relation data structure is a map from values to values; we'll call that map `equiv`. It turns out that, due to the way we implement unification, the values in the domain of the map are always placeholder values. To determine the set representative of a value v, we do the following:

1. Determine whether v is in the domain of `equiv`. If not, then v is a set representative and we're done.
2. If v is in the domain, then get the value that v maps to; call that value v'. Recursively determine the set representative of v' by going to step 1.

3.2.4 The Goal Stack

The goal stack keeps track of the remaining things that we need to satisfy in order to find a solution for the program. It consists mostly of *body* terms (as defined in the language syntax); the only exception is that we also have a **restore** goal that is used to restore environments after calling another clause (as explained above in the environment section).

3.2.5 The Choice Stack

When we call a clause that has more than one definition or we evaluate a disjunction, we must nondeterministically pick a goal to satisfy among the available choices. However, if we pick the wrong choice, we must be able to backtrack and try a different choice. The choice stack keeps track of the remaining available choices that we didn't pick, allowing us to do that backtracking when necessary. An element of the choice stack is a tuple of (*body*, environment, equivalence relation, goal stack). This tuple completely captures the state of the execution when we made our original choice, allowing us to restore that state if we need to make a new choice.

3.3 Helper Functions

The engine needs helper functions for the following: unifying variables, computing new environments, and evaluating *rhs* expressions. Each helper function is described below.

3.3.1 Unification

Unification is a method of determining whether two values are equivalent or not; alternatively, we can view it as a way of making two values equivalent if possible. The current equivalence relation is maintained in the equivalence relation data structure `equiv`, described above in Section 3.2.3. The affect of trying to unify two values is either (1) success, with an updated `equiv`; or (2) failure. We can represent these possibilities with an `Option` type, where `None` represents failure. The unification procedure pseudocode is below, where p means a placeholder value:

$$\texttt{unify}(v_1, v_2, \texttt{equiv}) =$$

$$\begin{cases} \texttt{Some(equiv)} & \text{if } v_1 = v_2 \\ \texttt{Some(equiv}[p \mapsto v_2]\texttt{)} & \text{if } v_1 = p \\ \texttt{Some(equiv}[p \mapsto v_1]\texttt{)} & \text{if } v_2 = p \\ \texttt{unifyTerms}(\vec{v_3}\ \textit{zip}\ \vec{v_4}, \texttt{equiv}) & \text{if } v_1 = \textit{sym}(\vec{v_3}),\ v_2 = \textit{sym}(\vec{v_4}),\ |\vec{v_3}| = |\vec{v_4}| \\ \texttt{None} & \text{otherwise} \end{cases}$$

```
unifyTerms($\overrightarrow{pairs}$, equiv) = $\overrightarrow{pairs}$.foldLeft(Some(equiv))(fun)
    where fun($acc$, ($v_1$, $v_2$)) =
      $acc$ match {
        case None ⇒ None
        case Some(newEquiv) ⇒ unify($v_1$, $v_2$, newEquiv)
      }
```

3.3.2 New Environments

New environments are constructed whenever we need to call a clause (or at the very beginning of program execution, when we need to evaluate the program's query). We will first discuss creating environments for clauses; then creating environments for queries follows as a special case. Given a clause $sym(\vec{x_1})\ \{\vec{x_2}\}$:− *body* that we're calling along with a list of values $\vec{v}$ of arguments that we're passing, the new environment is $(\vec{x_1}\ \mathit{zip}\ \vec{v})$ ++ $(\vec{x_2}\ \mathit{zip}\ \vec{p})$, where ++ unions two maps and $\vec{p}$ is a list of fresh placeholder values equal in length to $\vec{x_2}$.

Queries are handled in a similar manner. However, since queries do not take any arguments, the sequences represented by the variables $\vec{x_1}$ and $\vec{v}$ shown in the above expression are always empty.

3.3.3 Expression Evaluation

Expressions on the right of ← are either numbers n, arithmetic expressions on numbers and number-valued variables exp, or terms $sym(\vec{x})$. A number evaluates to itself. A binary arithmetic expression is evaluated recursively: evaluate the left-hand side to a number, the right-hand side to a number, then apply the given operation ($+$, $-$, $\times$, or $\div$). These arithmetic operations are only defined on numbers; if a value used in an arithmetic operation is not a number (or if the operation is division by zero) then the program execution aborts with an error. Terms are evaluated to *ground terms*, which simply means terms that do not contain any variables. To evaluate a term $sym(\vec{x})$, we evaluate each argument x to its value v and the corresponding ground term value is $sym(\vec{v})$.