The `mini-prolog-fd` Language

Kyle Dewey and Ben Hardekopf

1 Overall Language Design

`mini-prolog-fd` is an extension of `mini-prolog` to include finite-domain symbolic constraints and a constraint solver. The symbolic constraints assume that all integer values are between 0 and `Int.MaxValue`, inclusive. We simplify the constraint solver by only allowing symbolic constraints between two variables, rather than between arbitrary arithmetic expressions. This makes the language less expressive, but easier to implement.

2 Internal Syntax

`mini-prolog-fd` extends the syntax of `mini-prolog` to include symbolic relational operators (with a '#' superscript) and an expression `fd_labeling(X)` to retrieve variable values from the constraint solver. Here is the definition of the extended syntax:

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

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

The only differences from the `mini-prolog` syntax are (1) the new bodies $x_1 \bowtie^{\#} x_2$ and **fd_labeling**(x), and (2) the six new symbolic relational operators. The expression $x_1 \bowtie^{\#} x_2$ sends a new constraint to the constraint solver and checks for satisfiability; the expression **fd_labeling**(x) takes the value v of variable x and queries the constraint solver to bind any symbolic values contained in v to satisfying integers.

3 Symbolic Values

We extend the possible values for a variable to include *symbolic placeholders*, signified with a # superscript like so: $p^{\#}$. Thus, the possible values of a variable are (1) a ground term; (2) a number; (3) a placeholder; or (4) a symbolic placeholder. Symbolic placeholders are used to interact with the constraint solver; they are the solver's variables that it is trying to find satisfying values for. Because symbolic placeholders are handled using the constraint solver, they must be treated specially by the `mini-prolog-fd` interpreter to avoid messing things up; this is described further below.

4 Language Implementation

To implement the `mini-prolog-fd` runtime engine, we extend the specification of `mini-prolog`. We treat the constraint solver as a black box; its implementation is specified in a separate document.

4.1 Engine Pseudocode

```
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
  constraintStore is the initial empty store of constraints

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

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

    case $x_1 \equiv x_2$ ⇒
      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
      -- see the helper functions section for how unification should be modified for CLP

    case **check** $sym(\vec{x})$ ⇒
      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, constraintStore) 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 ⇒
      set env to envR

    case $x_1 \bowtie x_2$ ⇒
      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 $v_1$, which will be a placeholder value
      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, constraintStoreC) from choiceStack
        set env to envC, equiv to equivC, goalStack to goalStackC, and constraintStore to constraintStoreC
        push $body$ onto goalStack

    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 either value is a ground term or if both are numbers then treat exactly like $\bowtie$
      else:
        if either $v_1$ or $v_2$ are placeholders, unify them with fresh symbolic placeholders
        call the resulting values $v_1'$ and $v_2'$
        let result = constraintStore insert $v_1' \bowtie^{\#} v_2'$
        match result with one of the following:
          None $\Rightarrow$ push **false** onto goalStack
          Some(cs) $\Rightarrow$ set constraintStore to cs

    case **fd_labeling**$(x)$ $\Rightarrow$
      look up $x$ in env to get its value $v$
      get the list of symbolic placeholder values $\overrightarrow{p^{\#}}$ in $v$
      query constraintStore for satisfying assignments $\vec{n}$ for $\overrightarrow{p^{\#}}$
      insert new constraints setting each symbolic placeholder equal to its satisfying number
      update equiv to replace any symbolic placeholder in $\overrightarrow{p^{\#}}$ with its satisfying number
end while
```

4.2 Data Structures

The only new data structure is the constraint store. We also slightly modify the elements of the choice stack. Each data structure is described below.

4.2.1 The Constraint Store

The constraint store holds the current set of symbolic constraints and contains a constraint solver that can check the constraints for satisfiability. This functionality is described in a separate document. From the engine's perspective, the constraint store's interface should allow:

- Inserting a new constraint. The constraint store will return an Option that is either `None` (the new constraint is inconsistent with existing constraints) or `Some(constraintStoreN)`, where `constraintStoreN` is an updated constraint store containing the newly inserted constraint. In the latter case, the new constraint is consistent with existing constraints, meaning there is guaranteed to be a satisfying solution.

- Querying for satisfying solutions. Given a list of symbolic placeholder values, the constraint store will return an assignment of symbolic placeholders to numbers that satisfies all of the constraints.

4.2.2 The Choice Stack

Previously, an element of the choice stack was a tuple of (*body*, environment, equivalence relation, goal stack). We need to extend that tuple to now include the constraint store. 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.

4.3 Helper Functions

The unification helper should be modified for CLP.

4.3.1 Unification

Because the constraint solver is handling symbolic placeholders, we need to treat them specially for unification. In particular, if we're unifying two values such that either (1) one value is a symbolic placeholder and the other is a number, or (2) both values are symbolic placeholders, then rather than unifying them we instead insert an appropriate constraint into the constraint store. If the result is satisfiable the "unification" succeeds, otherwise it fails. If we unify a placeholder and a symbolic placeholder, the new set representative should be the symbolic placeholder. If we unify a symbolic placeholder with a ground term then the unification should fail.