Constraint Logic Programming

1 Motivation

Early in the course, we introduced arithmetic operations, including those below:

```
?- 3 > 4.
false.

?- X is 1 + 2.
X = 3.

?- X = 7, X < 10.
X = 7.
```

The meaning of these operations should be fairly straightforward.

However, there is a very obnoxious restriction on the above operations: they only work with known values. That is, any variables involved **must** have known values by the time the arithmetic constraint is reached, and it is considered an error otherwise. For example, all the following queries produce errors:

```
?- X > 2.
ERROR: >/2: Arguments are not sufficiently instantiated

?- X is Y + 3.
ERROR: is/2: Arguments are not sufficiently instantiated

?- X < 10, X = 7.
ERROR: </2: Arguments are not sufficiently instantiated
```

The last query is particularly annoying, since it works if we change the order of the conjunction. That is, if we instead do the `X = 7` part first, the (then subsequent) `X < 10` would succeed.

This requirement that variables have known values breaks the logical model. The fact that proper conjunction ordering can mean the difference between the program working correctly and a fatal error is similarly broken. Now we have to think more from an operational standpoint, that is, *how* Prolog will execute our code, as opposed to fundamentally *what* the problem is we're trying to solve. This isn't very Prolog-like.

This problem of needing variables to have values with arithmetic constraints is not fundamental, but rather a restriction Prolog imposes to make implementations of the language easier. As such, it's possible to overcome this limitation with some extra work.

2 Digression: Trying to Solve this Problem Directly in Prolog

> ***Sidenote:*** This entire section can be safely skipped. It's merely here to illustrate that numeric representation is the fundamental problem here, as opposed to something deeper about numbers. If you like math, you'll probably like this section.

We can solve this problem, at least partially, using only the Prolog we've already seen. In order to come to this solution, we need to ask a very basic question: what is a number? This is fundamental to solving this problem, as the answer will define what a solution looks like. Prolog's answer to this is that a number is some internally-represented value, most likely a string of bits. Bit strings are but one way of encoding numbers, which happens to be convenient on most modern computers. There exist other, more fundamental encodings, as well.

The particular encoding we will use here is based on the *Peano axioms*, which were written in the 1800's as a logic-based way to describe natural numbers (i.e., integers $\geq$ zero). The Peano axioms define natural numbers inductively, like so:

$$n \in \mathit{NaturalNumber} ::= \textbf{zero} \mid \textbf{successor}(n)$$

That is, a natural number is either:

1. Zero
2. The successor to another natural number n. Phrased another way, if n is a natural number, then $n + 1$ is also a natural number. We intentionally don't write it with $+$, because we haven't yet defined what $+$ does on this definition, so we treat the idea of a number succeeding another number as primitive.

Note that the above definition looks *very* much like the inductive definition of lists we've seen before; the only real difference is that lists hold a list element on the **successsor** case.

Numbers defined as above are often referred to as *Peano numbers*. Using the above definitions, we can define common arithmetic operations over Peano numbers, such as addition:

$$\begin{aligned} \textbf{zero} + n &= n \\ \textbf{successor}(n_1) + n_2 &= \textbf{successor}(n_1 + n_2) \end{aligned}$$

In plain English, the above definition states (on a per-line basis):

1. Adding zero (**zero**) to any other number (n) yields n.
2. If adding a successor to another number (**successor**(n_1), where n_1 is the other number) to yet another number (n_2), this yields the successor of whatever the two numbers added together yields (**successor**($n_1 + n_2$)). Using $+1$ instead of **successor**, this could be written as: $(n_1 + 1) + n_2 = (n_1 + n_2) + 1$. However, because of the way we've defined things with **successor**, this definition will eventually reach the base case involving **zero**, so this is well-founded (it won't trigger infinite recursion).

Similarly, we can define $<$:

$$\begin{aligned} \textbf{successor}(n) < \textbf{zero} &= \textbf{false} \\ \textbf{zero} < \textbf{successor}(n) &= \textbf{true} \\ \textbf{successor}(n_1) < \textbf{successor}(n_2) &= n_1 < n_2 \end{aligned}$$

An English description of the above definition on a per-line basis follows:

1. The successor to any number n is never less than **zero**. Phrased another way, a positive number is never less than zero.
2. **zero** is always less than the successor to any number n. Phrased another way, **zero** is always less than positive numbers.
3. The successor to some number n_1 is less than the successor to some other number n_2, as long as $n_1 < n_2$. This acts as a recursive case, breaking down both arguments to $<$ into smaller pieces. Using $+1$ instead of successor, this could be read as: $(n_1 + 1) < (n_2 + 1) = n_1 < n_2$.

Using $<$, we can define $\leq$ in a straightforward manner, shown below:

$$\begin{aligned} n \leq n &= \textbf{true} \\ n_1 \leq n_2 &= n_1 < n2, \text{ iff } n_1 \neq n_2 \end{aligned}$$

The iff part above is to state that the two cases are mutually exclusive.

The nice thing about the above definitions is that they can be translated directly to Prolog quite easily. Such a translation is shown below:

```
1  add(zero, Num, Num).
2  add(successor(Num1), Num2, successor(Rest)) :-
3      add(Num1, Num2, Rest).
4
5  lessThan(zero, successor(_)).
6  lessThan(successor(Num1), successor(Num2)) :-
7      lessThan(Num1, Num2).
8
9  lessThanOrEqual(Num, Num).
10 lessThanOrEqual(Num1, Num2) :-
11     lessThan(Num1, Num2).
```

This translation should be straightforward, so we won't go into detail behind it.

With the translation in hand, we can start to issue Prolog queries involving natural numbers, first, let's add three and two, to get things started:

```
?- add(successor(successor(successor(zero))), successor(successor(zero)), Result).
Result = successor(successor(successor(successor(successor(zero))))).
```

As shown, three is the successor to the successor to the successor of zero (`successor(successor(successor(zero)))`). Similarly, two is the successor to the successor of zero (`successor(successor(zero))`). The result ends up being (deep breath) the successor to the successor to the successor to the successor to the successor of zero, or successor^5 of zero. All of this amounts to $3+2=5$, **but** we didn't use `+` anywhere.

Here is where the magic of Prolog comes in. Let's ask a different query: is there such an X such that $X+1=3$? This is shown below:

```
?- add(X, successor(zero), successor(successor(successor(zero)))).
X = successor(successor(zero)) ;
false.
```

As shown, our definition of `add` (implementing +) doesn't require us to know the inputs to the addition; we can have variables anywhere. This determines that X must be two (`successor(successor(zero))`), and this is the only answer. All of this comes without any concept of algebraic manipulation.

We can ask even more complex queries. For example: is there such an X and Y such that $X+Y=3$? This is shown below:

```
?- add(X, Y, successor(successor(successor(zero)))).
X = zero,
Y = successor(successor(successor(zero))) ;
X = successor(zero),
Y = successor(successor(zero)) ;
X = successor(successor(zero)),
Y = successor(zero) ;
X = successor(successor(successor(zero))),
Y = zero ;
false.
```

As shown, there are multiple answers, corresponding to the following (in order of when the answers were shown):

1. $0+3=3$
2. $1+2=3$
3. $2+1=3$
4. $3+0=3$

We can combine `add` (+) with `lessThan` (<). For example, let's revisit the above query, and change it to the following: is there such an X and Y such that $X + Y = 3$, **and** $X < Y$? This is shown below:

```
?- add(X, Y, successor(successor(successor(zero)))), lessThan(X, Y).
X = zero,
Y = successor(successor(successor(zero))) ;
X = successor(zero),
Y = successor(successor(zero)) ;
false.
```

The above query ends up forcing only the $0 + 3 = 3$ and $1 + 2 = 3$ answers to exist, since those are the only answers for which $X < Y$.

Clearly, the use of arithmetic based on the Peano axioms (or just *Peano arithmetic*) gets us much further than what Prolog gives us outside of the box. However, this solution is still problematic. For one, all the operations shown operate in $\mathcal{O}(N)$, where N is the size of the number. This is a whole lot worse than the $\mathcal{O}(1)$ we're used to.

Moreover, this solution *still* has a certain sensitivity to conjunctions. For example, let's change the order of the conjunctions in the above query, shown below:

```
?- lessThan(X, Y), add(X, Y, successor(successor(successor(zero)))).X = zero,
Y = successor(successor(successor(zero))) ;
X = successor(zero),
Y = successor(successor(zero)) ;
% significant hanging occurs here
ERROR: Out of global stack
```

As shown, we end up causing infinite recursion here. This is because the query `lessThan(X, Y)` has an infinite number of solutions, and it will end up trying **all** of them with the subsequent `add`. We don't see this problem in the previous query because `X` and `Y` have known values (sound familiar?) by the time `add` is reached. As such, the `lessThan` call in that case only every operates on known data, and won't attempt to enumerate all solutions.

While the Peano arithmetic approach gets us further, we still need a better solution that can overcome these sort of problems with infinite recursion.

3 A Better Way: Constraint Logic Programming

Towards solving this problem, there is a major insight to make: Prolog itself is a constraint solver over one particular *logic* (yes, there exist different kinds of logics, which each reason about different things). Prolog's logic is focused around *equality constraints*: that is, are two things equal to each other? This is the premise behind unification, and why we use = for unification. We can extend Prolog's logic to handle other kinds of constraints as well, generalizing Prolog to *constraint logic programming* (CLP). With CLP, we can add in *arithmetic constraints* like $x < y$, $a = 2 + b$, etc. Because Prolog is based on logic, it's fairly natural to add these sort of constraints, and they don't fundamentally change how we write code.

> ***Sidenote:*** Don't worry if this insight doesn't make sense or is non-obvious. Prolog existed for at least 15 years before CLP was discovered, specifically by Jaffar and Lassez in their POPL'87 publication "Constraint Logic Programming". The rest of this section is based around a pragmatic, applications-oriented view of constraint logic programming.

With constraint logic programming, we can naturally express the same sort of queries as we were trying to before. In order to do this with SWI-PL, we first need to write the following:

```
?- use_module(library(clpfd)).
true.
```

The code above effectively imports the related libraries, which are not included by default. From here, we can issue queries, such as the equivalent of $X < Y$ shown below:

```
?- X #< Y.
X#=<Y+ -1.
```

As shown above, we must use `#<` instead of `<` to use CLP. This is because `<` is already taken by the usual definition that won't work on variables. This effectively asks: is X less than Y?. This succeeds, and gives a strange-looking result. You don't need to worry about this particular result (it shows some mostly internal components of the related library handling CLP), only that it succeeded.

In saying `X #< Y`, Prolog effectively records that `X` must be less than `Y`. However, in general, it does not pick any particular values for `X` and `Y` at this point. Instead, these are represented *symbolically*, and will be replaced with concrete numbers later on.

To perform this sort of replacement with concrete numbers, we need to use `label/1`. This procedure takes a list of variables involved in arithmetic constraints, and will replace the variables with concrete numbers. There exist multiple satisfying solutions, in which case `label/1` will nondeterministically return all possible answers. An example using `label/1` is shown below:

```
?- X #> 0, X #< 5, label([X]).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4.
```

The `label/1` procedure is a little finicky, in that it won't try to enumerate any variables which it thinks have an infinite number of values. In such a case, it will instead produce an error. An example of this is shown below:

```
?- X #>= X, label([X]).
ERROR: Arguments are not sufficiently instantiated
```

In the above example, since there are an infinite number of values of X such that $X \geq X$, `label/1` refuses to try to enumerate this.

Probably the simplest way to get around this problem is to bound the values that variables can take on. This can be done using constraints you have already seen, like so:

```
?- X #>= X, X #>= 0, X #=< 4, label([X]).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
X = 4.
```

We an also use a new constraint, namely `in`, to do this same sort of bounding succinctly:

```
?- X #>= X, X in 0..4, label([X]).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
X = 4.
```

When working with multiple variables, a variant of `in` (namely `ins`) is particularly effective:

```
?- X #> Y, [X, Y] ins 0..2, label([X, Y]).
X = 1,
Y = 0 ;
X = 2,
Y = 0 ;
X = 2,
Y = 1.
```

The above query states that variable `X` must be greater than variable `Y`, and that both `X` and `Y` are in the range $[0, 2]$. This is equivalent to the following (must more verbose) query:

```
?- X #> Y, X #>= 0, X #=< 2, Y #>= 0, Y #=< 2, label([X, Y]).
X = 1,
Y = 0 ;
X = 2,
Y = 0 ;
X = 2,
Y = 1.
```

3.1 Practical Example: Solving Sudoku

We've discussed the problem of solving Sudoku before, and it was used as an example of a problem where optimization was practically necessary. A refresher on the problem with solving Sudoku in Prolog follows, along with a discussion of how CLP can help us out here.

3.1.1 Sudoku Refresher

Recall that the main difference between a naive and an optimized Sudoku solver is in how conjunctions were applied. A naive solver took an approach like the following (using pseudocode):

```
naiveSolve(Board) :-
    allRowsUnique(Board),
    allColumnsUnique(Board),
    allSquaresUnique(Board).
```

... where `allRowsUnique` checked that all rows contained distinct values, `allColumnsUnique` checked that all columns contained distinct values, and `allSquaresUnique` checked that all 3x3 squares contained distinct values. In practice, this setup was incredibly inefficient, since `allRowsUnique` would fill in the entire board. We would then filter-out invalid solutions with the subsequent calls to `allColumnsUnique` and `allSquaresUnique`. We'd expect that nearly all solutions would be filtered-out, since `allRowsUnique` completely ignores column and 3x3 square constraints (which are handled by `allColumnsUnique` and `allSquaresUnique`, respectively).

In contrast, an optimized solver will check row, column, and 3x3 square constraints for each value entered into the board. This fundamentally still checks the same row, column, and 3x3 square constraints as with the naive solver, but now it interleaves the checks for each value instead of performing them separately in batch. This optimization is key, and it makes the difference between a Sudoku solver that can practically handle only several unknowns, and a Sudoku solver than can rapidly solve boards with dozens of unknowns (as with real problems). However, it requires us to restructure code in a way that makes it far less clear, and itmay force us to use tricky operations like `var/1` which behave in non-logical ways.

3.1.2 Solving Sudoku with CLP

With CLP, we can effectively write a solver that *looks* like the naive solver, but performs more like the optimized solver. This leads to a solver which is far more straightforward and concise.

Sudoku is such a well-known problem that the documentation for the CLP libraries in SWI-PL include a complete solver, available here: `http://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku`. The source code for this solver is reproduced below, and forms a complete solver:

```
1  blocks([], [], []).
2  blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
3      all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
4      blocks(Ns1, Ns2, Ns3).
5
6  sudoku(Rows) :-
7      length(Rows, 9),
8      maplist(same_length(Rows), Rows),
9      append(Rows, Vs),
10     Vs ins 1..9,
11     maplist(all_distinct, Rows),
12     transpose(Rows, Columns),
13     maplist(all_distinct, Columns),
14     Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
15     blocks(As, Bs, Cs),
16     blocks(Ds, Es, Fs),
17     blocks(Gs, Hs, Is).
```

A line-by-line explanation of this solution follows:

- **Lines 1-4**: Definition of the helper procedure `blocks/3` which is used to handle constraints on 3x3 squares. The `all_distinct/1` constraint used on line 3 succeeds only if all variables in the given list are distinct from each other. This is a CLP constraint which we haven't discussed, but is nonetheless straightforward in its capability.
- **Line 7**: There are nine rows in a Sudoku board.
- **Line 8**: Each row contains nine elements. This makes use of `maplist/2`, which applies a given constraint (the first parameter) to each element of a list (the second parameter). The particular constraint applied is `same_length/2`, which succeeds if it is given two lists of the same length. In this specific case, this is stating that each row needs to contain the same number of rows as the board as a whole. Hence, this is stating that each row has nine elements, since the board contains nine rows. Note that this line does not use CLP constraints; both `maplist/2` and `same_length/2` are defined in other standard libraries of SWI-PL.
- **Line 9**: Gather each element in the board into a single list of elements `Vs`. This makes use of `append/2`, which behaves a lot like `flatten/2`, but only goes one level deep (appropriate since the board is represented as a list of lists of numbers). Notably, `append/2` is defined outside of the CLP libraries.
- **Line 10**: Each element in the board must be in the range $[1, 9]$, corresponding to the fact that each square in Sudoku can only hold between one through nine, inclusive.
- **Line 11**: The elements of each row must be distinct.
- **Line 12**: Gather the elements in the rows into a series of columns. For example, consider the following query:

  ```
  ?- transpose([[1, 2, 3], [4, 5, 6], [7, 8, 9]], Columns).
  Columns = [[1, 4, 7], [2, 5, 8], [3, 6, 9]].
  ```

  Note that while `transpose/2` is defined in the CLP libraries (seemingly just to make Sudoku solvers easier to write), it does not need to use arithmetic constraints.
- **Line 13**: The values in each column are distinct. As with the naive Sudoku solver, this treats all columns in batch as opposed to applying constraints on a per-element basis, as the optimized Sudoku solver does.
- **Line 14-17**: The values in each 3x3 square must be distinct. Again, this works like the naive Sudoku solver, applying constraints on 3x3 squares in batch.

While the above code may *look* like a naive solver, it performs far more quickly. Using an example from the same Webpage containing a whopping 64 unknowns, the above solver is able to find a solution in under 0.2 seconds on a relatively slow computer. Such performance is more in line with how an optimized solver is expected to perform; a problem of this size is well outside of the capabilities of a naive solver.

3.2 How is CLP Performance so Good?

A major question may come to mind when considering the performance of the CLP-based Sudoku solver: how can this possibly be so fast, especially when we write code that *looks* naive? The short answer to this question is that these sort of arthmetic constraints are solved by a custom, high-performance solver which operates behind the scenes with the CLP libraries. Exactly how this solver works is beyond the scope of this class, though it works somewhat like the optimized Sudoku solver in that it will carefully choose variable values satisfying all arithmetic constraints (e.g., `X #> Y`) in an incremental way, as opposed to naively brute-forcing different values for different variables. We could easily spend an entire course on how these solvers work, and different solvers work in fundamentally different ways. As such, we will leave off of the discussion there, and simply say that these solvers are fast.

> ***Sidenote:*** If you're still wondering how these arithmetic solvers work, in particular the solver in SWI-PL, you may be interested in reading the Ph.D. dissertation of Markus Triska, the main developer behind the solver in SWI-PL. The dissertation is titled "Correctness Considerations in CLP(FD) Systems", and a direct link follows: `https://www.metalevel.at/drt.pdf`. The fact that this solver is behind an entire Ph.D. dissertation should give an idea of its complexity; I cannot do justice to it with a short discussion, and I honestly am unfamiliar with most of the details myself.