EECS 390 – Lecture 20

Constraints, Dependencies, and Matching

4/2/24

Constraint Logic Programming

- Extension of logic programming to include constraints on variables

- Basic Prolog includes limited arithmetic constraints that require variables to be instantiated

```
square_sum([N, X, Y, Z]) :-
    N =:= Z * Z, N =:= X * X + Y * Y,
    X > 0, Y > 0, Z > 0, X < Y, N < 1000.
```

```
?- square_sum(S).
ERROR: =:=/2: Arguments are not sufficiently
instantiated
```

CLP(FD)

- The CLP family of libraries provide constraint logic programming as extensions to Prolog
- CLP(FD) is included in SWI-Prolog and works on finite domains (integer subsets)

Import CLP(FD) module

```
:- use_module(library(clpfd)).

square_sum_c([N, X, Y, Z]) :-
    N #= Z * Z, N #= X * X + Y * Y,
    X #> 0, Y #> 0, Z #> 0, X #< Y, N #< 1000,
    label([N, X, Y, Z]).
```

CLP(FD) constraint operator

Require given variables to be *grounded*

```
?- square_sum_c(S).
S = [25, 3, 4, 5] ;
S = [100, 6, 8, 10] ;
S = [169, 5, 12, 13] ;
...
```

Search in CLP

- Search follows the same general strategy as Prolog, except that a ***constraint store*** keeps track of the set of constraints
  - Start with a set of goal terms
  - For first goal term, find a clause whose head can be unified with the term
    - Unification can instantiate or bind variables
  - Insert body terms that are not constraints into the front of the set of goal terms
  - Insert body terms that are constraints into the constraint store
  - Check whether the constraint store is unsatisfiable
    - If so, backtrack
- Search succeeds when no more goal terms remain, and the constraint store is not unsatisfiable

Example: Verbal Arithmetic

- Find a solution to the following such that each digit is distinct, and leading digits are non-zero:

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

- Plain Prolog:

```
money(List) :-
    List = [S, E, N, D, M, O, R, Y],
    maplist(is_digit, List), S =\= 0, M =\= 0, is_set(List),
                     1000 * S + 100 * E + 10 * N + D
    +                1000 * M + 100 * O + 10 * R + E
    =:= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

Higher-order predicate

Takes ~90 seconds to solve in online interpreter

Example: Verbal Arithmetic

- Find a solution to the following such that each digit is distinct, and leading digits are non-zero:

```
      S E N D
+     M O R E
-----------
=   M O N E Y
```

- Prolog + CLP(FD):

```
money_c(List) :-
    List = [S, E, N, D, M, O, R, Y],
    List ins 0 .. 9, S #\= 0, M #\= 0, all_distinct(List),
```

Takes <0.01 second to solve in online interpreter

Require variables in List to be members of set [0, 9]

Constrain variables in List to have distinct values

```
                    1000 * S + 100 * E + 10 * N + D
    +               1000 * M + 100 * O + 10 * R + E
    #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
    label(List).
```

Example: Sudoku

- Sudoku solver:

Higher-order predicate

Partial application

```
sudoku(Rows) :-
  length(Rows, 9), maplist(same_length(Rows), Rows),
  append(Rows, Values), Values ins 1..9,
  maplist(all_distinct, Rows),
  transpose(Rows, Columns),
  maplist(all_distinct, Columns),
  Rows = [Row1,Row2,Row3,Row4,Row5,Row6,Row7,Row8,Row9],
  blocks(Row1, Row2, Row3),
  blocks(Row4, Row5, Row6),
  blocks(Row7, Row8, Row9),
  maplist(label, Rows).

blocks([], [], []).
blocks([N1, N2, N3 | RestRow1],
       [N4, N5, N6 | RestRow2],
       [N7, N8, N9 | RestRow3]) :-
    all_distinct([N1, N2, N3, N4, N5, N6, N7, N8, N9]),
    blocks(RestRow1, RestRow2, RestRow3).
```

Matching

- Regular expressions and grammars specify rules for matching against strings of characters
- Sometimes, we want to match against other kinds of objects
- High-level pattern:

  if *<expr>* **matches** *<pattern1>* **then** *<computation1>*
  else if *<expr>* **matches** *<pattern2>* **then** *<computation2>*
  …
- Some languages have specific constructs to simplify expression of this pattern

Simple Matching

- Switch/case constructs enable matching based on whether an expression produces a particular value
  - Usually, the values specified in the construct must be constants
  - Example in Scheme:

```
(case x
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)
)
```

- Try/catch constructs enable matching based on type

```
try { throw some_object; }
catch (std::invalid_argument &err) { /* … */ }
catch (std::out_of_range &err) { /* … */ }
catch (std::exception &err) { /* … */ }
```

Structural Pattern Matching

- Declarative languages often allow more complex pattern matching based on whether an object has a particular structure
  - Example in Prolog:

```
len([], 0).
len([_First|Rest], Length) :-
  len(Rest, RestLength), Length is 1 + RestLength.
```

- Many functional languages also provide mechanisms for structural pattern matching
  - Example: Scheme macros

```
> (static-length (1 2 3))
3
```

```
(define-syntax static-length
  (syntax-rules ()
    ((static-length ()) 0)
    ((static-length (first rest ...))
     (+ 1 (static-length (rest ...))))
  )
)
```

rest ... matches zero or more items

Scheme only has static pattern matching. Other functional languages such as OCaml have dynamic pattern matching.

Pattern Matching in Python

- Python 3.10 introduced the `match` construct, which can match by value, type, or structure
- Example: match by value

```
def https_error_description(code):
    match code:
        case 400:
            return 'Bad Request'
        case 401:
            return 'Unauthorized'
        case 403:
            return 'Forbidden'
        case 404:
            return 'Not Found'
        case _:
            return f'Unknown code {code}'
```

Anonymous variable

Pattern Matching in Python

- Python 3.10 introduced the `match` construct, which can match by value, type, or structure
- Example: match by structure

```
def length(sequence):
    match sequence:
        case []:
            return 0
        case [_, *rest]:
            return 1 + length(rest)
```

Sequence pattern matches many kinds of sequences, not just lists

Variadic pattern matches zero or more items

- Matching by type should be used judiciously, as it is equivalent to using `isinstance()`
  - Object orientation and dynamic binding is often a better choice for type-based dispatch

Make

- Tool for automating the building of software packages, tracking dependencies between components
- Programming model is a combination of declarative and imperative
- A ***rule*** declares a relation between a target and its dependencies, specifies commands to build the target

```
target: dependencies
        commands
```

Zero or more targets or files

Tab indentation

Sequence of zero or more commands, usually each on its own line

Simple Example

- Rule contained within `Makefile`:

```
hello:
         echo "Hello world!"
```

No dependencies

Build hello target

Build first target in `Makefile`

```
$ make hello
echo "Hello world!"
Hello world!
$ make
echo "Hello world!"
Hello world!
```

Target has no dependencies, so it will always build

Building an Executable

- More complex dependency trees can be specified

```
main.exe: a.o b.o c.o
        g++ -o main.exe a.o b.o c.o

a.o: a.cpp
        g++ --std=c++14 -Wall -pedantic -c a.cpp

b.o: b.cpp
        g++ --std=c++14 -Wall -pedantic -c b.cpp

c.o: c.cpp
        g++ --std=c++14 -Wall -pedantic -c c.cpp
```

```
$ make
g++ --std=c++14 -Wall -pedantic -c a.cpp
g++ --std=c++14 -Wall -pedantic -c b.cpp
g++ --std=c++14 -Wall -pedantic -c c.cpp
g++ -o main.exe a.o b.o c.o
```

Rebuilding a Target

- A target is only rebuilt when one of its dependencies has been modified

Modify timestamp on `b.cpp`

```
$ touch b.cpp
$ ls -l
-rw-r--r--  1 kamil  staff    229 Nov 17 01:01 Makefile
-rw-r--r--  1 kamil  staff     90 Nov 17 00:57 a.cpp
-rw-r--r--  1 kamil  staff   6624 Nov 17 01:01 a.o
-rw-r--r--  1 kamil  staff     31 Nov 17 01:12 b.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 b.o
-rw-r--r--  1 kamil  staff     33 Nov 17 00:58 c.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 c.o
-rwxr-xr-x  1 kamil  staff  15268 Nov 17 01:01 main.exe
$ make
g++ --std=c++14 -Wall -Werror -pedantic -c b.cpp
g++ -o main.exe a.o b.o c.o
```

Only `b.o` is rebuilt

Example: (Old) Makefile for Notes

```
all: foundations functional theory data declarative

foundations: foundations.html foundations.tex

functional: functional.html functional.tex

theory: theory.html theory.tex

data: data.html data.tex

declarative: declarative.html declarative.tex

asynchronous: asynchronous.html asynchronous.tex

metaprogramming: metaprogramming.html metaprogramming.tex

%.html: %.rst
	rst2html.py --stylesheet=../style/style.css $< > $@

%.tex: %.rst
	rst2latex.py --stylesheet=../style/style.sty $< > $@
	pdflatex $@
	pdflatex $@
	pdflatex $@

clean:
	rm -vf *.html *.tex *.pdf *.aux *.log *.out
```

Not built by default

Pattern rule

Build PDF file

Dependencies

Target