Introduction to Cognitive Robotics

Module 10: Using Turtlesim with CRAM

Lecture 4: Using Prolog for reasoning

David Vernon
Carnegie Mellon University Africa

www.vernon.eu

Using Prolog for Reasoning

Based on Using Prolog for reasoning
http://cram-system.org/tutorials/beginner/cram_prolog

Using Prolog for Reasoning

CRAM has its own Prolog interpreter

- Written in Common Lisp
- Can work natively with the Lisp data structures

Implemented in the **cram_prolog** package

- So, first, we need load the package

```
CL-USER> (asdf:load-system :cram-prolog)
CL-USER> (in-package :cram-prolog)
PROLOG>
```

Using Prolog for Reasoning

Prolog queries are executed by calling the `prolog` function

– The first argument is a list of symbolic data

```
PROLOG> (prolog '(member ?x (1 2 3)))
(((?X . 1)) ⟵ This is the first binding in the lazy list
 . #S(CRAM-UTILITIES::LAZY-CONS-ELEM :GENERATOR #<CLOSURE # {100A55B86B}>))
```

This query means:
find such a binding (i.e. an assignment) for the variable ?x such that the predicate (i.e. the statement) "?x is a member of the list (1 2 3)" is true

Here, there are several bindings for ?x that satisfy the statement, namely, 1, 2 or 3; these are are all valid assignments for ?x

– Since there are possibly very many valid bindings, the CRAM Prolog interpreter returns the bindings as a lazy list and not a normal list

- By default, a lazy list returns one value at a time
- It uses a generator function to extract more values when requested

Using Prolog for Reasoning

We can force a lazy list to be a normal list with the function `force-ll`

If this causes a problem, use `cut:force-ll` (`cut` is the name of the package)

```
PROLOG> (force-ll (prolog '(member ?x (1 2 3))))
(((?X . 1)) ((?X . 2)) ((?X . 3)))
```

← These are all the bindings in the list

Using Prolog for Reasoning

Variables in CRAM Prolog are represented by any symbol that starts with `?`

If a variable in some predicate of a Prolog query has no value assigned to it we say that the variable is unbound

mn

Using Prolog for Reasoning

If there are no solutions for the query Prolog returns NIL:

```
PROLOG> (prolog '(and (member ?x (1 2 3))
                      (< ?x 0)))
NIL
```

mn

Using Prolog for Reasoning

If there are no unbound variables in the query but the query itself is true, Prolog returns a list where bindings are NIL:

```
PROLOG> (force-ll (prolog '(> 4 0)))
(NIL)
```

mn

Using Prolog for Reasoning

The CRAM Prolog interpreter finds solutions by performing a depth first search over the predicate tree:

- It first searches for possible bindings for the first predicate
- It then branches into multiple search paths, one per each assignment (binding)
- It then continues with the second predicate
- and so on

Same for all Prolog interpreters

mn

Using Prolog for Reasoning

The CRAM Prolog interpreter differs from other Prolog interpreters …

- Some predicates can be proven by using a Lisp function
- and not through depth-first search over possible solutions

Use `lisp-fun` and `lisp-pred`

For solving the problem of finding out if 3 is odd or not using a conventional Prolog query the engine would need to have a database of all the odd numbers and would have to search if 3 is in that database.

```
PROLOG> (force-ll (prolog '(lisp-pred oddp 3)))
(NIL)
```

Here, we calculate the result

By calculating the result instead of looking it up through depth first search one can solve continuous and geometric problems such as "where on a table should I put the box so that it doesn't obstruct me from seeing the monitor that is standing on the same table"

Recall: Inference

- Facts can be represented by a list comprising
  - Predicate
  - Zero or more arguments

donald is the parent of nancy

```
(parent donald nancy)
```

- Rules tell what can be inferred from the facts we already have

then-part if-part

```
(<- head body)
```

“If *y* is the parent of *x* then *x* is the child of *y*”
[Alternatively, we can prove any fact of the form (child *x* *y*) by proving (parent *y* *x*)]

```
(<- (child ?x ?y) (parent ?y ?x))
```

Variables are represented as symbols beginning with a question mark

Using Prolog for Reasoning

Defining custom predicates: rules, facts, and fact groups

- In Prolog there can be multiple ways of proving the same predicate (multiple implementations)
- So predicate implementations don't get overwritten but are always being added to
- However, sometimes we do want to replace an old implementation with a new one
- CRAM Prolog does this with fact groups
  - These are the basic "compilation" units of the Prolog engine
  - Each time a fact group is recompiled, all the old implementations inside the group are being replaced with the new ones
  - (Whereas implementations of the same predicates from other fact groups are untouched and stay valid)

Using Prolog for Reasoning

Defining custom predicates: rules, facts, and fact groups

- Predicate definitions are called rules or facts
- CRAM Prolog uses the macro `def-fact-group` with `<-` to define custom Prolog predicates

```
PROLOG> (def-fact-group family-predicates ()
          (<- (grandparent ?grandparent ?grandkid)
            (parent ?grandparent ?x)
            (parent ?x ?grandkid)))
```

Here we define a compilation unit called family-predicates

One rule:
?grandparent is a grandparent of ?grandkid
if
?grandparent is a parent of ?x. and
?x is a parent of ?grandkid

The predicate grandparent is true for two variables ?grandparent and ?grandkid if there is a certain binding for ?x such that ?grandparent is a parent of ?x and ?x is a parent of ?grandkid.

Using Prolog for Reasoning

Defining custom predicates: rules, facts, and fact groups

- Let's extend this to define the parent fact (not a rule because there is no body)

```
PROLOG> (def-fact-group family-predicates ()
        (<- (grandparent ?grandparent ?grandkid)
          (parent ?grandparent ?x)
          (parent ?x ?grandkid))

        (<- (parent my-dad me))
        (<- (parent me my-kid)))
```

Two facts:
my-dad is a parent of me
I am a parent of my-kid

Using Prolog for Reasoning

Defining custom predicates: rules, facts, and fact groups

Let's test this

```
PROLOG> (force-ll (prolog '(parent ?a my-kid)))
(((?A . ME)))
PROLOG> (force-ll (prolog '(grandparent ?a my-kid)))
(((?A . MY-DAD)))
```

Query: Who is the parent of my kid?
Are there any bindings for which this is true?

Me

Query: Who is the grandparent of my kid?
are there any bindings for which this is true?

my-dad

Using Prolog for Reasoning

Defining custom predicates: rules, facts, and fact groups

Let's test this

Query: Are there people who satisfy the grandparent rule?
are there any bindings for which this is true?

```
PROLOG> (force-ll (prolog '(grandparent ?a ?b)))
(((?A . MY-DAD) (?B . MY-KID)))
PROLOG> (force-ll (prolog '(and (parent ?parent-of-me me)
                    (grandparent ?grandparent-of-my-kid my-kid))))
(((?PARENT-OF-ME . MY-DAD) (?GRANDPARENT-OF-MY-KID . MY-DAD)))
```

Yes: my-dad and my-kid satisfy this

Query: Is someone my parent and also the grandparent of my kid?
Are there any bindings for which this is true?

Yes: my-dad satisfies this

CRAM Beginner Tutorials

Create a CRAM Package	http://cram-system.org/tutorials/beginner/package_for_turtlesim
Controlling turtlesim from CRAM	http://cram-system.org/tutorials/beginner/controlling_turtlesim_2
Implementing simple plans to move a turtle	http://cram-system.org/tutorials/beginner/simple_plans
Using Prolog for reasoning	http://cram-system.org/tutorials/beginner/cram_prolog