Introduction to Docker and Prolog

September 23, 2020

1 Introduction to Docker and how to use it.

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines.

Docker is free to download.

1.1 Install Docker

1.1.1 How to install docker in windows.

For this purpose it is better to install Ubuntu as a subsystem for windows. Then we can run docker using the terminal of the Subsystem we installed. Lets have a look and see how we can do this.

1.1.2 How to install docker in Mac.

Installing docker desktop in Mac is straightforward. Lets have look at this process as well.

1.1.3 How to install docker in ubuntu

Installing docker on Ubuntu

1.2 Interact with Docker

In order to use Docker we, need to first generate images with an OS kernel and configuration that necessary for an specific operation. We can do this by defining the `Dockerfile` Lets have a close look at an example to understand how it works:

```
# This Dockerfile has two required ARGs to determine which base image
# to use for the JDK and which sbt version to install.

# Define the argument for openjdk version
ARG OPENJDK_TAG=8u232
# Do the packaging based on openjdk
FROM openjdk:8u232
```

```
# Set the name of the maintainers
MAINTAINER Habib Ghaffari Hadigheh, Mark Armstrong <ghaffh1@mcmaster.ca, armstmp@mcmaster.ca>

RUN apt-get update && \
  apt-get install scala -y && \
  apt-get install -y curl && \
  sh -c '(echo "#!/usr/bin/env sh" && \
  curl -L https://github.com/lihaoyi/Ammonite/releases/download/2.1.1/2.12-2.1.1) > /usr/local/b
  chmod +x /usr/local/bin/amm'

# Set the working directory
WORKDIR /opt/h1
```

1.3 Basic commands to work with Docker

If you want to use docker as tool for implementation, you may need to know lots of commands, here is cheat sheet for this purpose. But in this course it is not necessery to learn and remeber many complicated processes. Here is a short list of command you may/may not use in this course:

- Build

```
docker build [OPTIONS] PATH | URL | -
```

- Run

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

- See the list of images:

```
docker images
```

- See the list of containers

```
docker ps [OPTIONS]
```

Learning all the commands and optoins take time. The best resource for biggners to start with docker is this online free course.

1.4 How to use it for the purpose of this course

We already defined the `Dockerfile` with all necessary packages and library necassery for you. We also definde a `docker-compose.yml` file that will help you in the process of building the images, as well as running/stoping the containers.

Lets see how you can do that.

Here is the link of the Docker test for h1.

Now we are going to build and run the test to see how it works.

h1 test using docker

2 Learning Inference Rules

We are using inference rules in this course for two main reasons:

1. To understand the semantics of programming languages.
2. Discuss the type systems.

- Prolog is a programming language that is all about inference rules. That is why we are covering it at the begging of the course this year.

2.1 Prolog

Prolog programs are simply databases of inference rules, also called clauses. Let have a look at a simple inference rule.

$$\frac{A_1 \; A_2 \; ... \; A_n}{B}$$

Lets see how we can write it down in Prolog:

```
b :- a1, a2, a3, a4, a5.
```

We can seperate the permisis by `,`. This rule states that b is true if all of the a_i are true. So we can think of `:-` as $\leftarrow$. Notice the **period** ending the rule.

Now lets have a look at a very simple axiom:

$$\frac{}{C}$$

We can write this in prolog as

```
c :- true.
```

or more simply,

```
c.
```

How we can get output from sequence of inference rules?

2.2 Get output from sequence of inference rules

To interact with prolog programs, we make queries, to which Prolog reponds by checking its inference rule database to determine possible anwsers. Lets have a look at one example:

2.2.1 List catenatoin

List catenation in SWI prolog is a **ternary predicate**. Here is how it's written

```
append(X,Y,Z)
```

the rule of which enforce that `Z` is the result of catenating `X` and `Y`.

lets have a look and see which kinds of queries we cand make:

- Is `[1,2,3,4]` the result of catenating `[1,2]`, and `[3,4]`?

[1]:
```
%%script swipl -q
append([1,2],[3,4],[1,2,3,4]).
```

```
true.
```

- What are the possible values of Z for which Z is the catenation of `[1,2,3]` and `[4,5,6]`?

[2]:
```
%%script swipl -q
append([1,2,3],[4,5,6],Z).
```

```
Z = [1, 2, 3, 4, 5, 6].
```

What are the possible values of X and Y so that, when they are catenated, the result is `[1,2,3,4,5,6]`?

[3]:
```
%%script swipl -q
append(X,Y, [1,2,3,4,5,6]).
```

```
X = [],
Y = [1, 2, 3, 4, 5, 6]


ERROR: Type error: `character_code' expected, found `-1' (an integer)
ERROR: In:
ERROR:   [11] char_code(_23930,-1)
ERROR:   [10] '$in_reply'(-1,'?h') at /usr/local/Cellar/swi-
prolog/8.2.1/libexec/lib/swipl/boot/init.pl:911
```

The reponse will be:

```
X = [],
Y = [1, 2, 3, 4, 5, 6] ;
X = [1],
Y = [2, 3, 4, 5, 6] ;
X = [1, 2],
Y = [3, 4, 5, 6] ;
X = [1, 2, 3],
Y = [4, 5, 6] ;
X = [1, 2, 3, 4],
Y = [5, 6] ;
X = [1, 2, 3, 4, 5],
Y = [6] ;
X = [1, 2, 3, 4, 5, 6],
Y = [] ;
```

Note: In this way, we get several "functions" from one predicate, depending upon what question(s) we ask!

2.3 Names, kinds of tems

- In Prolog, any **name** begining with an upper case letter denotes a variable.
- Names which begin with lower case letters are **atoms**. which are a type of constant value. Atoms may be used as the name of predicates.

Note: You should avoid calling atome functions, because they are relations. Functions are special kind of relations/predicates that are single-value.

- Character strings surrounded by single quotes, are also atoms. So we can write

```
'is an empty list'([]).
```

- As you would expect, Prolog also has numerical constants, such as `1` or `3.14`.
- Aside from what described above, the remianing kind or Prolog term is a *structure*, which has the fome below:

```
atom(term1,...,terml)
```

As you can see the syntax is simple, the main important and probably problmatic part of Prolog is simantics of it.

2.4 Interacting with Prolog

As we've said, a Prolog program consist of clauses (inference rules.) As an example have a look at this clause:

```
head(X) :- body(X,Y)
```

Which can be interpreted as below:

$\forall X, head(X) \leftarrow (\exists Y, body(X, Y))$

Then during computation, given this cluase and the goal `head(X)`, the Prolog runtime is tasked with finding a substitution for `Y` which makes `body(X,Y)` true.

We provide Prolog with goals through queries, usually by loading our programs into the interactive query REPL, either by running

```
swipl my_program.pl
```

from the command line, or

```
?- consult('my_program.pl')
```

once runing SWI Prolog.

We can also `assert` or `retract` rules in the query REPL. if needed. Let have look at that.

[4]:
```
%%script swipl -q
assert(c).
c.
retract(c).
c.
assert(d).
assert(c :- d).
c.
```

```
true.

true.

true.

false.

true.

true.

true.
```

Also, use `listing` or `listing(name)` to see all given clauses or clauses about the `name` predicate.

2.5 Unification

The computation of model of Prolog involves *unification* of terms. Terms unify if either: 1. They are equal, or 1. They contain variables that can be **instantiated** in a way that makes the terms equal.

We saw an example of this when we are using `append(X,Y,[1,2,3,4])`. Prolog tries to find us a possible bining. If it cannot then it reply false.

So in general, unification involves searching for possible variable binings, by making use of the clauses, and modus ponens

$(P \wedge P \Rightarrow Q) \Rightarrow Q$

2.5.1 The goal list

Through this process, the single goal presented by a quey will usually turn into a collection of goals; For instance if we query `?- P(5).` and the search uses a clause

```
p(X) :- Q(X), R(X).
```

then we now have as goals `q(X)` and `r(X)`.

Now the goals will change to `q(5)` and `r(5)`.

2.5.2 Backtracking

Now we are defining a small program:

$a \leftarrow b \wedge c$

$a \leftarrow b$

$b = \top$

$c = \bot$

Here is how we write it in prolog:

```
a :- b, c.
a :- b.

b.
c :- false.
```

This will be the result of `?- trace.` command:

```
[trace]  ?- a.
   Call: (10) a ? creep
   Call: (11) b ? creep
   Exit: (11) b ? creep
   Call: (11) c ? creep
   Call: (12) false ? creep
   Fail: (12) false ? creep
   Fail: (11) c ? creep
   Redo: (10) a ? creep
   Call: (11) b ? creep
   Exit: (11) b ? creep
   Exit: (10) a ? creep
true.
```

As Prolog runtime tries to prove `a`, it will use the first rule, and `fail` (because in trying to prove `c`, it reaches `false`, which it cannot prove.). At that point it has to **backtrack**, and try a different clause to prove `a`.

In general runtime will backtrac serveral times during a proof, and it keeps track of which clauses have been tried.

2.5.3 SWI Prolog's search strategy

1. Attempt to apply clauses in order from top to bottom (as in the source code).

- Only backtrack and try other clauses after success and failure.

1. Perform a depth first search to prove each goal.

- So if the current goal are `b` and `c`, try to prove `b` before considering `c`.

2.5.4 Examining the search strategy

In order to interactively see the process Prologe is using during unification, use the trace. command in the REPL. Then each query will result in a log of calls made and failures encoutered.

2.6 Equality

Prolog has an equality comparision, written simply = (not ==). **However**, this equality does not simplification. So. for instance, if a variable X has been unified to value 5

```
X = 5
```

would be `true`. but

```
X= 2 + 3
```

would be `false`.

Lets try it out very quickly:

[5]:
```
%%script swipl -q
5 = 2 + 3.
```

```
false.
```

How we can do this comparison?

This non-simplifying equality allows us to conisder construction of terms, rather that just their value. But in case we want to actually carry out arithmetic or calculate other values, we can use the `is` comparison.

[6]:
```
%%script swipl -q
5 is 2 + 3.
```

```
true.
```

2.7 Exerting control over the search; the cut operator

In part because Prolog's searching mechanism can be naive, the programmer is given a certain amount of control over the search.

The most important mechanism for controlling the search that we will see is the cut.

A cut is written `!`, and can be understood as "no backtracking is allowed to go beyond this point

2.8 Checking for divisors of a number

```
hasDivisorLessThanOrEqualTo(_,1) :- !, false.
hasDivisorLessThanOrEqualTo(X,Y) :- 0 is X mod Y, !.
hasDivisorLessThanOrEqualTo(X,Y) :- Z is Y - 1, hasDivisorLessThanOrEqualTo(X,Z).
```

Lets try this out to see what is the results: