Assignment 1

Professor: Dr. Constantinos Constantinides

Course: Soen 331

Alawadi Mustafa and Randall-M.Charles

February 27 2024

1

Since we have blue $\rightarrow$ prime

The equivalent is the contrapositive $\neg$prime $\rightarrow$ $\neg$blue

Given the numbers 9 and 11, and the colors blue and yellow, we need to flip the cards that will help us verify the contrapositive.

1. **Blue Card**: Since the proposition states that blue cards must have prime numbers on the other side, we need to verify if the blue card has a prime number on its back. If it doesn't, it would confirm the contrapositive.

2. **Number 9 Card**: Since we know that 9 is not a prime number, flipping this card will help us determine if a non-prime number corresponds to the color blue, which is necessary to confirm the contrapositive.

Therefore, the cards to turn over are the blue card and the number 9 card.

2

2.1

1)

The predicate logic formulas are:

Planet $= \forall\, x, y : \text{Object}(x) \wedge \text{Mass}(x, y) \wedge \text{Orbits}(x, sun) \rightarrow \text{Planet}(x)$

is_satelitte_of $= \forall x, y : \text{Object}(x) \wedge \text{Planet}(y) \wedge \text{Orbits}(x, y) \rightarrow \text{is\_satellite\_of}(x, y)$

2)

```
:- consult('solar.pl').

planet(X) :- mass(X, Y), Y > 0.33, orbits(X, sun).

is_satellite_of(X, Y) :- orbits(X, Y), planet(Y).


?- planet(P).       %% non-ground query
P = venus ;
P = earth ;
P = mars ;
P = jupiter ;
P = saturn ;
P = uranus ;
P = neptune ;
false.

?- planet(sun).     %% ground query
false.

?- planet(mercury). %% ground query
false.
```

3)

```
obtain_all_satellites(X, L) :- findall(Y, is_satellite_of(Y, X), L).

?- obtain_all_satellites(earth, Satellite).      %% non-ground query
Satellite = [moon] ;
false.
```

2.2

1)

$\exists x(\text{number}(x) \wedge \neg\text{composite}(x))$ Type O

2)

$\forall x(\text{number}(x) \rightarrow \text{composite}(x))$ Type A

3)

$\exists\, x(\text{number}(x) \wedge \text{composite}(x))$ Type I

4)

$\forall\, x(\text{number}(x) \rightarrow \neg\text{composite}(x))$ Type E

2.3

1)

The negation is : $\exists\, s : S \mid s \notin P$ We know that the proposition O is in the form: $\exists\, x[\neg p(x)]$. Therefore, the negation of A is logically equivalent to the O proposition.

2)

The E proposition is in the form $\forall\, x[\neg p(x)]$ or in this case $\forall\, s : S \mid s \notin P$. If we take the negation we get $\exists\, s : S \mid s \in P$ and since the proposition I is in the form $\exists\, s : S \mid s \in P$ then E and I are logically equivalent.

3

3.1

1)

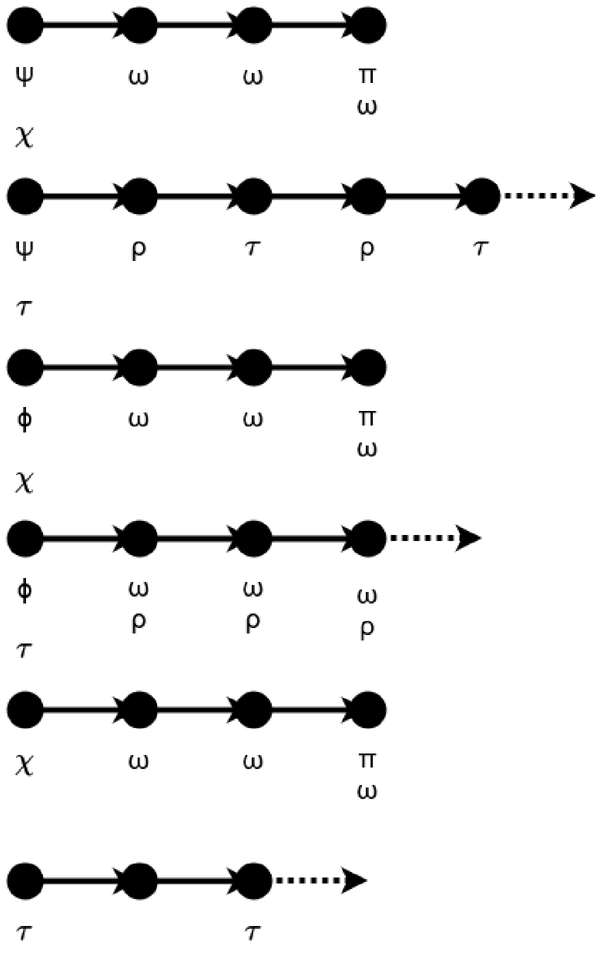

Figure 1: These are the temporal logic visualizations.

2)

The first visualization ends when π releases ω. The next one loops, since ρ and τ keep calling themselves at intervals of 2 time periods. The third and fifth visualizations terminate in the same manner as the first. The fourth visualizations is infinite in the context where τ never releases ρ and π never releases ω. The last one contains an infinite loop that could terminate if ρ became true.

3.2

1)

The formal form is as follows:

$$\diamond(\neg\phi \land \neg\psi) \rightarrow \bigcirc^2(\tau \mathrm{R} \chi)$$

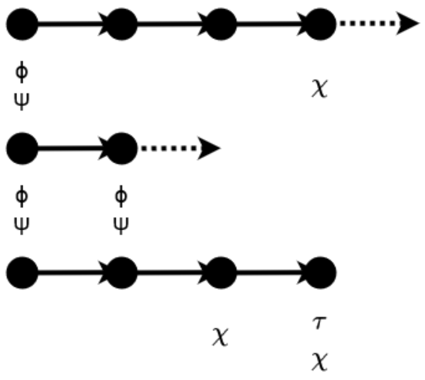

Figure 2: These are the temporal logic visualizations.

2)

If either α or β is false, then starting from time = i+1, γ will eventually be true up and until some future moment when δ becomes true.

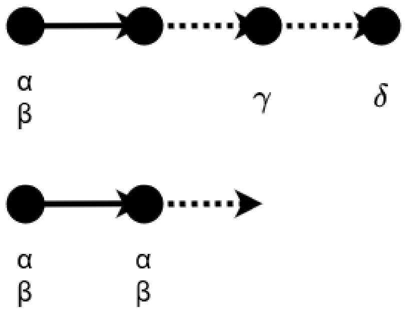

Figure 3: These are the temporal logic visualizations.

3)

If at time = i+1 τ becomes true and χ eventually becomes true, then at time = i+2, ϕ will be true unless ψ becomes true.

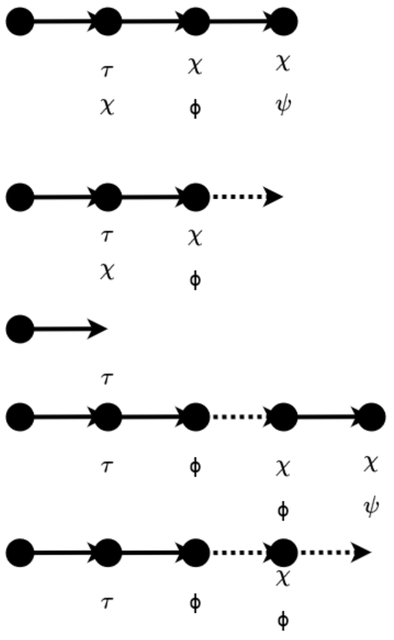

Figure 4: These are the temporal logic visualizations.

4

1)

In plain English, the P Languages is the set of different programming languages that are used in the domain of software development. It represents the powerset of the set Languages which means it includes all possible subsets that can be made from the set of Languages. The last element is a set itself which contains three languages.Therefore, the set contain both languages and sets of languages.

2)

a)

The expression is a variable declaration, in which the LHS represents the variable and the RHS represents the type. The expression signifies that there is a relationship between ”Favorites” and the set P Languages. Thus ”Favorites” can have any subset of the set P Languages since it is the powerset of the set Languages.

b)

Favorites : P Languages is interpreted as the variable Favorite can assume any value supported by the powerset of Languages. Thus Favorites can include any combination of programming languages listed in P languages.

c)

The legitimate values for the variable ”Favorites” could be Lua, Groovy, and C since in order for them to be legitimate, they must be a subset of P Languages. Additionally, the set itself and the empty set could also be legitimate since it is a subset of the set of languages.

3)

a)

The expression Favorites = P signifies that the variable ”Favorites” is assigned the entire powerset of the set Languages.

b)

It isn’t semantically equivalent to Favorites:P Languages since this expression Favorites = P assigns the whole powerset to the variable whereas this expression Favorites:P declares the variable to have the type powerset.

4)

No, since this expression is not an element of P Languages. P Languages contain the subsets of Languages but not distinct elements which are sets to begin with.

5)

No, since any subset that belongs to a power set has to be a set of sets. Therefore this structure is rather a set of atomic elements.

6)

a)

This is an non-atomic variable. The Languages declare the variable of the type Languages and that is the set of programming languages.

b)

This is an non-atomic variable. P Languages declare the variable of the type of P Languages that is the powerset of Languages.

7)

Library is a set that contains elements of programming languages and it can have any elements that are supported by the P Languages. Therefore, it is of type P Languages.

8)

Yes it is correct because this empty set that contains one element, which is the empty set, is a subset of any set and thus it is included as an element.

9)

```
(defun is-memberp (element set)
  (member element set))

(defun equal-setsp (set1 set2)
  (and (subsetp set1 set2) (subsetp set2 set1)))

; Define some sets
(defvar set1 '(1 2 3 4 5))
(defvar set2 '(3 4 5 6 7))
(defvar set3 '(1 2 3 4 5))
(defvar set4 '(3 4 5 2 1))

; Test is-memberp
(is-memberp 3 set1) ; Returns T
(is-memberp 6 set1) ; Returns NIL

; Test equal-setsp
(equal-setsp set1 set2) ; Returns NIL
(equal-setsp set1 set3) ; Returns T
(equal-setsp set2 set4) ; Returns NIL
```

5

1)

Qeueue

$\upharpoonright(Enqueue, Dequeue)$

$\Sigma_1 : \mathbb{P}\ STACK\_TYPE$
$count1 : \mathbb{N}$
$count2 : \mathbb{N}$
$element : \mathbb{T}$

$count1, count2 \geq 0$

Init

$Stack = \varnothing$
$Stack2 = \varnothing$
$count1 = 0$
$count2 = 0$

Enqueue1OK

$\Delta(\Sigma_1, count1, count2, element)$
$\Sigma_1? : STACK\_TYPE$

$count2 == 0$
$\Sigma_1\ ' \ = \Sigma_1?\ \cup \langle element? \rangle$
$count1' = count1 + 1$

Dequeue1OK

$\Delta(\Sigma_1, \Sigma_2, count1, count2)$
$\Sigma_1? : STACK\_TYPE$

$count1 == 0\ \&\&\ count2 \geq 1$
$\Sigma_2\ ' = \Sigma_1? \setminus \langle x \rangle$
$count2' = count2 - 1$
$topelement\ ! = x$

Enqueue2OK

$\Delta(\Sigma_1, \Sigma_2, count1, count2,$
$element)$
$\Sigma_1? : STACK\_TYPE$
$element? : \mathbb{T}$

$\Sigma_1\ ' \ = \Sigma_1\ \cup\ \Sigma_1?$
$\Sigma_1\ '' = \Sigma_1\ \cup \langle element? \rangle$
$count1 = count2 + 1$
$count2 = 0$

Dequeue2OK

$\Delta(\Sigma_1, \Sigma_2, count1, count2)$
$\Sigma_1? : STACK\_TYPE$

$count1 \geq 1\ \&\&\ count2 == 0$
$\Sigma_2\ ' = \Sigma_1?$
$\Sigma_2\ '' = \Sigma_2\ '\ \setminus \langle x \rangle$
$count2' = count1 - 1$
$count1 = 0$
$topelement\ ! = x$

$Enqueue \mathrel{\widehat=} (Enqueue1OK \mid\mid Enqueue2OK)$
$Dequeue \mathrel{\widehat=} (Dequeue1OK \mid\mid Dequeue2OK)$

2)

```
(setf stack1 '())
(setf stack2 '())
```

```
(defun length (lst)
(if (null lst)
0
(+ 1 (length (cdr lst)))))

(defun reverse (lst)
    (cond ((null lst) '())
        (t (append (reverse (cdr lst)) (list (car lst))))))

(defun enqueue ( queue element )
(let ((count1 length(stack1)) (count2 length(stack2))
) (if (zerop count2)
    (append '(element) queue)
    (append '(element) queue)
    )

;;  (if (and (not (zerop count1))       --->For dequeue
;;        (zerop count2))
;;    1
;;    0)

)
)
```

3)

```
head([H|_], H).

tail([_|T], T).

cons([], List, List).
cons(List, [], List).
cons(ATOM, List2, [ATOM|List2]).
cons([H|T], List2, [H|T3]) :- cons(T, List2, T3).


?- head([a, b, c, d], H).
H = a .

?- head([x], H).
H = x .

?- head([], H).
false.
```

```
?- tail([a, b, c, d], T).
T = [b, c, d] .

?- tail([x], T).
T = [] .

?- tail([], T).
false.

?- cons(a, [b, c], NewList).
NewList = [a, b, c] .

?- cons(a, [], NewList).
NewList = a ;
NewList = [a] .

?- cons([], [], NewList).
NewList = [] ;
NewList = [] ;
NewList = [[]] .
```

6

6.1

1) For R to be a partial order, it needs to be reflexive, anti-symmetric, transitive.

Reflexivity: All the elements x in the domain, xRx. The condition satisfies since for any x in the Java API, it reflects itself.

Anti-symmetry: For xRy and yRx, then we have x=y. This holds true since in Java all types are unique and can't be similar to one another simultaneously.

Transitivity: If we have xRy and yRz, then we have xRz. This holds true since Java allows for inheritance to be done in a single way, so if x is of type y and y is of type z, then x is of type z automatically.

We conclude that the relation R is reflexive, anti-symmetric and transitive, therefore is a partial order.

2) To prove that (V1,R) is a poset, it needs to be reflexive, anti-symmetric, transitive.

Reflexivity: All the values in V1 should be in R. This holds true since all the values are found in both vertices.

Anti-symmetry: If we have the values v and u, and they are related to each other then there is a direct relation between them,however, it is not the case here because of the directed edges.Thus there is anti-symmetry.

Transitivity: If we consider vRu and uRx, there is a direct link from v to u and from u to x. Thus, there is a direct link from v to x and for that reason the transitivity holds.

Therefore we have proved (V1,R) is a poset.

3) To create the Hasse Diagram, we will have to identify relationship between the vertices. For each edge (x,y) in E, we have xRy.

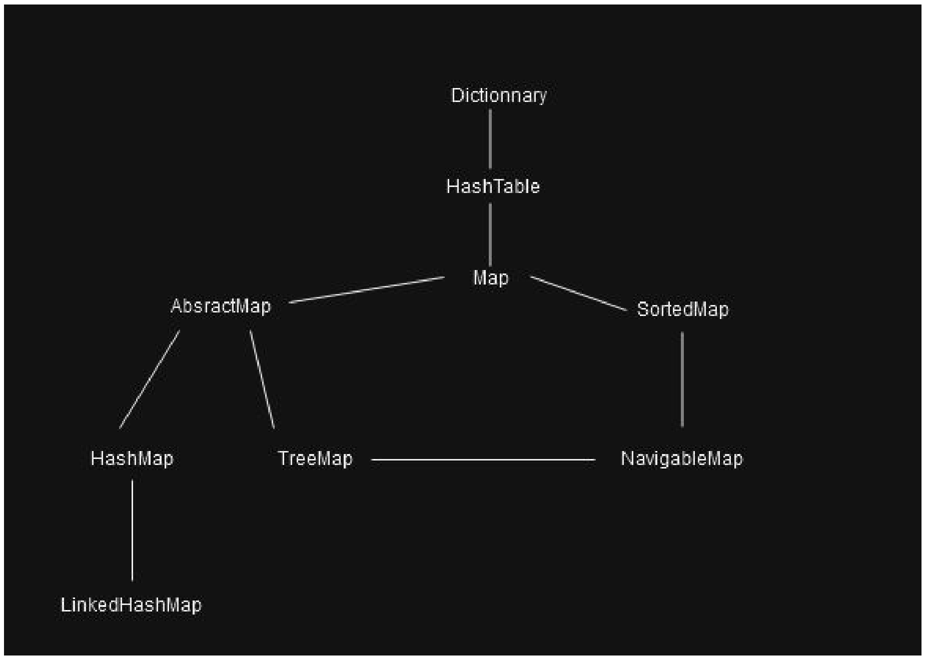

Figure 5: This is the Hasse Diagram.

Maximal element is Dictionary.
Minimal element is LinkedHashMap.

6.2

1) To prove that $\subseteq$ is a partial order, the binary relation must be reflexive, anti-symmetry, and transitive.

Reflexivity: Any set set x is a subset of itself since all elements in x are also elements in x. Thus, $x \subseteq x$.

Anti-symmetry: In case that x is a subset of y, and y is a subset of x, then x=y. Since $x \subseteq y$ and $y \subseteq x$, then every element in x is in y and vise-versa which implies that x=y as they have exactly the same elements.

Transitivity: In case x is a subset of y and y is a subset of z, then x is a subset of z. Thus, if $x \subseteq y$ and $y \subseteq z$, then $x \subseteq z$ and any value inside of x is inside of y and also inside of z.

Because the relation satisfies all three properties, we can say that the subset relation $\subseteq$ is a partial order over the domain of sets.

2) To prove that (P(V2),$\subseteq$) is a poset, it must be reflexive, anti-symmetry, and transitive.

Reflexivity: If we consider the set A in P(V2), then $A \subseteq A$ because a set is a subset of itself. So there is reflexivity.

Anti-symmetry: If we consider A and B $\in P(V2)$ as in $A \subseteq B$ and $B \subseteq A$, then any element in either set must also belong to the other one, so A=B.

Transitivity: If we let A, B, and C $\in P(V2)$ then we have $A \subseteq B \subseteq C$, which also means that $A \subseteq C$ and any element inside of A is inside of B and also inside of C.

Because all three properties hold, then (P(V2) ,$\subseteq$) is a poset.

3) To create a Hasse Diagram, we need to list all the elements of P(V2), thus the power set of V2.

Power set P(V2) is:

$$\{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

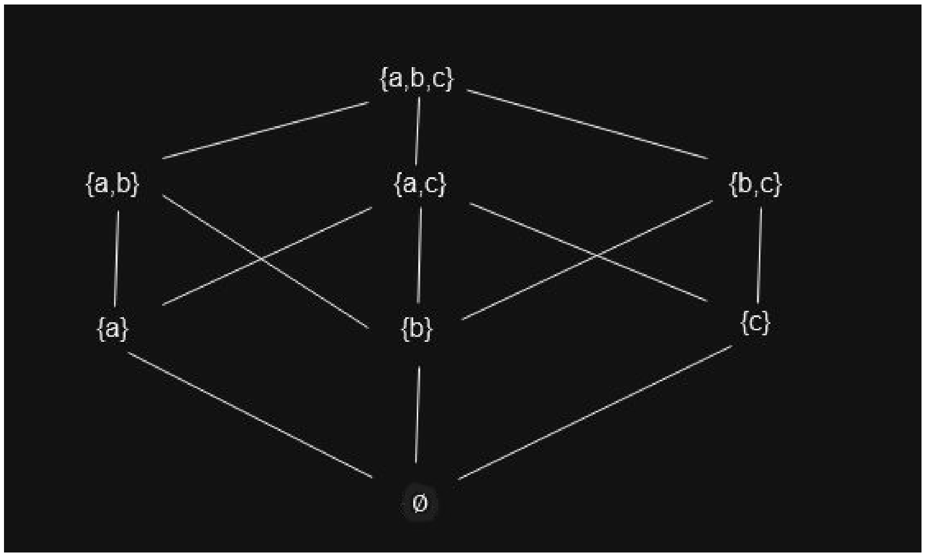

Figure 6: This is the Hasse Diagram.

The top element would be the maximal elements, thus Maximal elements:

$\{a,\ b,\ c\}$

The bottom elements would be the minimum elements, so Minimal element:

$\varnothing$

6.3

Map is a function since all elements in the domain map to exactly one element in the co-domain and it is perfectly fine to have two elements from the domain pointing to one element from the co-domain.
Map is a partial function since not all the elements of the co-domain are being mapped to.
Map is an injective function because each element of the domain map to one distinct element of the co-domain.It is not surjective since not each element of the co-domain is mapped to by at least one element of the domain. Thus the function is not bijective as bijection requires the function to be both injective and surjective.
The function is not order-preserving since the mapping does not involve sets that are ordered and the elements in the domain are not preserved by their images in the co-domain. The function is order-reflecting if the predecessor relationship between the elements in the co-domain is reflected by the pre-images in the domain. In this example, the order reflecting does not apply since the

sets are unordered collections. In order for a function to be order-embedding, it needs to be both order-preserving and order-reflecting, and since we said that the function is not order-reflecting, then it is not order-embedding. In order for a function to be isomorphic, it needs to be both order-embedding and surjective, therefore since this condition does not apply, then it is not isomorphic.

7

1)

```
(defun compress (lst)
  (if (null lst)
      '()
    (let ((compressed (list (first lst))))
      (dolist (elem (rest lst) compressed)
        (if (not (equal (first compressed) elem))
            (push elem compressed))))))
```

2)

```
(defun f (lst)
  (cond ((null lst) '())
        ((null (rest lst)) lst)
        ((equal (first lst) (second lst)) (f (rest lst)))
        (t (cons (first lst) (f (rest lst))))))
```

3)

We begin with the input list which is $\langle a, a, b, b, c, a \rangle$. Then we compress the function by adding "a" to the list. Because "a" is the same as the term before it, it is ignored, and thus "b" is added to the list that is compressed. So far we have $\langle a, b, c \rangle$. Then "a" is added to the list and finally we get the final compressed list which is $\langle a, b, c, a \rangle$.

4)

```
(defun compress (lst)
   (nreverse(if (null lst)
      '()
    (let ((compressed (list (first lst))))
      (dolist (elem (rest lst) compressed)
        (if (not (equal (first compressed) elem))
            (push elem compressed))))))
```