Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών

Στατιστικός Έλεγχος Υποθέσεων και Ερμηνεία με Χρήση Μη-Μονότονης Λογικής

Διπλωματική Εργασία

Νίκος Ασημάκης

Επιβλέπων Καθηγητής

Πέτρος Στεφανέας

2018

Περίληψη

Αντικείμενο αυτής της διπλωματικής εργασίας είναι η παρουσίαση μίας μεθοδολογίας για ανάπτυξη εφαρμογών που κάνουν χρήση της λογικής της επιχειρηματολογίας. Πιο συγκεκριμένα θα χρησιμοποιήσουμε το σύστημα Γοργίας που υλοποιεί τις ιδέες του Λογικού Προγραμματισμού χωρίς Άρνηση σαν Αποτυχία (Logic Programming without Negation as Failure, LPwNF) και τις συνδυάζει με την λογική απαγωγή. Σαν παράδειγμα αυτής της μεθοδολογίας θα αναπτύξουμε μία εφαρμογή που θα κάνει στατιστικό έλεγχο υποθέσεων αφού επιλέξει τον σωστό στατιστικό έλεγχο ανάλογα με τα χαρακτηριστικά του δείγματος και θα δίνει στον χρήστη μία απάντηση επιχειρηματολογώντας σαν μαθηματικός.

Λέξεις Κλειδιά: LPwNF, Gorgias, Prolog, R, Java, Στατιστικός Έλεγχος Υποθέσεων, Έλεγχος ανεξαρτησίας X^2, ακριβής έλεγχος Fisher

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της διπλωματικής, Πέτρο Στεφανέα, για την καθοδήγηση του κατά τη διάρκεια εκπόνησης της. Επίσης θα ήθελα να ευχαριστήσω τον υποψήφιο διδάκτορα του ΕΜΠ, Γιάννη Κιουβρέκη, για τις παρατηρήσεις του και την βοήθεια του. Ακόμα ευχαριστώ τον Αντώνη Κάκα, καθηγητή του Πανεπιστημίου Κύπρου, και τον Νίκο Σπανουδάκη, ερευνητή του Πολυτεχνείου Κρήτης για τις παρατηρήσεις τους και για την πλούσια εργασία τους χωρίς την οποία δεν θα μπορούσα να ολοκληρώσω αυτήν την διπλωματική. Τέλος ευχαριστώ την οικογένεια μου για την στήριξη τους στην διάρκεια των σπουδών μου.

Νίκος Ασημάκης

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Σκοπός αυτής της διπλωματικής είναι να παρουσιάσει μία μεθοδολογία ανάπτυξης εφαρμογών που εκμεταλλεύονται τα προτερήματα της λογικής της επιχειρηματολογίας. Πιο συγκεκριμένα θα χρησιμοποιήσουμε το σύστημα Γοργίας[12] που υλοποιεί ένα γενικό πλαίσιο επιχειρηματολογίας το οποίο συνδυάζει τις ιδέες της του συλλογισμού με προτιμήσεις και της λογικής απαγωγής με τρόπο που διατηρεί τα προτερήματα και των δύο.

Καθώς το σύστημα Γοργίας είναι υλοποιημένο στην γλώσσα λογικού προγραμματισμού Prolog[9] θα αναπτύξουμε μία διασύνδεση με την γλώσσα προγραμματισμού Java ώστε να μπορούμε να εκμεταλλευτούμε, μέσω των πολλών διαθέσιμων βιβλιοθηκών της Java, τα πλεονεκτήματα της λογικής της επιχειρηματολογίας σε διαφορετικά προβλήματα. Θα δούμε επίσης πώς μπορούμε να ξεχωρίζουμε τους κανόνες της λογικής που καθορίζουν την συμπεριφορά και παραμένουν σταθεροί σε κάθε πρόβλημα από τα δυναμικά δεδομένα που κάθε φορά μπορεί να είναι διαφορετικά και θα συζητήσουμε για τα προτερήματα μίας τέτοιας προσέγγισης.

Τα παραπάνω θα εφαρμοστούν στο πεδίο της της στατιστικής ανάλυσης δεδομένων και του έλεγχου υποθέσεων. Σήμερα υπάρχουν διαθέσιμα πολλά στατιστικά λογισμικά πακέτα (όπως για παράδειγμα η R[14]) που δίνουν νέες δυνατότητες στην ανάλυση δεδομένων. Επίσης η χρήση στατιστικών μεθόδων έχει αυξηθεί σε πολλούς τομείς, όπως για παράδειγμα στην ιατρική έρευνα, και χρησιμοποιείται από πολλούς επιστήμονες και όχι μόνο από μαθηματικούς. Πολλές φορές όμως, χωρίς την σωστή μαθηματική καθοδήγηση από εξειδικευμένους επιστήμονες, υπάρχει δυσκολία στην επιλογή των σωστών στατιστικών μεθόδων επειδή πολλές φορές παραλείπονται σημαντικές λεπτομέρειες, όπως ο τύπος των δεδομένων και της κατανομής, τα χαρακτηριστικά του στατιστικού δείγματος και οι προϋποθέσεις των θεωρημάτων. Αυτό έχει σαν αποτέλεσμα να γίνεται λάθος χρήση των εργαλείων της στατιστικής και να

εμφανίζονται σφάλματα.

Θα αναπτύξουμε μία σύνδεση της R με τον Γοργία μέσω της Java χρησιμοποιώντας την διασύνδεση που αναπτύξαμε ώστε κάνοντας χρήση των πλεονεκτημάτων της λογικής της επιχειρηματολογίας να αναπτύξουμε μία εφαρμογή που θα κάνει στατιστική ανάλυση και έλεγχο υποθέσεων και θα μειώνει τα λάθη που μπορεί να προκύψουν από λανθασμένη εφαρμογή των στατιστικών μεθόδων. Η εφαρμογή αυτή θα επιχειρηματολογεί σαν μαθηματικός και θα κάνει στατιστικό έλεγχο υποθέσεων δικαιολογώντας μάλιστα τις απαντήσεις της κάνοντας αναφορά σε συγκεκριμένους κανόνες και θεωρήματα. Επίσης μία τέτοια εφαρμογή είναι δυνατόν στο μέλλον να επεκταθεί εύκολα περαιτέρω, με την προσθήκη νέων κανόνων συλλογισμού που δεν θα διαταράξουν τους ήδη υπάρχοντες, ώστε να εξελιχθεί σε ένα διαδραστικό εργαλείο που θα δίνει απάντηση σε πολλά διαφορετικά στατιστικά ερωτήματα και θα επιλέγει αυτόματα κάθε φορά την κατάλληλη μεθοδολογία.

Κεφάλαιο 2

Λογικός Προγραμματισμός χωρίς Άρνηση σαν Αποτυχία

2.1 Πλαίσιο LPwNF

Στα πλαίσια του λογικού προγραμματισμού χωρίς άρνηση σαν αποτυχία[2] (LPwNF) τα λογικά προγράμματα είναι μη μονότονες θεωρίες όπου κάθε πρόγραμμα αντιμετωπίζεται σαν μία συλλογή από προτάσεις από τις οποίες πρέπει να επιλέξουμε ένα κατάλληλο υποσύνολο, το οποίο ονομάζεται επέκταση, ώστε να επιχειρηματολογήσουμε. Οι προτάσεις σε ένα λογικό πρόγραμμα γράφονται στην συνηθισμένη λογική γλώσσα με την διαφορά ότι γίνεται χρήση κλασσικής άρνησης και όχι άρνηση σαν αποτυχία. Για παράδειγμα έστω ότι έχουμε το παρακάτω πρόγραμμα

$$fly(x) \leftarrow bird(x)$$
$$\neg fly(x) \leftarrow penguin(x)$$
$$bird(x) \leftarrow penguin(x)$$
$$bird(Tweety)$$

με μία σχέση προτεραιότητας μεταξύ των κανόνων που ορίζει ότι ο δεύτερος κανόνας είναι ισχυρότερος από τον πρώτο. Από αυτό το πρόγραμμα μπορούμε να συμπεράνουμε ότι $fly(Tweety)$ γιατί μπορούμε να το εξάγουμε από τον πρώτο κανόνα και δεν υπάρχει τρόπος να εξάγουμε το $\neg fly(Tweety)$.

Αν προσθέσουμε την πρόταση $penguin(Tweety)$ τότε μπορούμε να εξάγουμε $fly(Tweety)$ και $\neg fly(Tweety)$ από τους κανόνες του προγράμματος. Όμως στην σχέση προτε-

ραιότητας που έχουμε ορίσει ο δεύτερος κανόνας είναι ισχυρότερος του πρώτου οπότε το συμπέρασμα που εξάγεται από αυτόν, το $\neg fly(Tweety)$, υπερισχύει.

Ακολουθούν οι βασικοί ορισμοί του πλαισίου λογικού προγραμματισμού χωρίς άρνηση σαν αποτυχία[1, 2]:

Ορισμός 2.1.1 (Πρόγραμμα)
Ένα πρόγραμμα $(K, <)$ είναι ένα σύνολο κανόνων K και μία σχέση προτεραιότητας $<$ που ορίζεται πάνω στους κανόνες K.

Ορισμός 2.1.2 (Επιτίθεται)
Έστω $(K, <)$ πρόγραμμα και $T, T' \subseteq K$. Το T' επιτίθεται στο T ανν υπάρχει $L, T_1 \subseteq T'$, και $T_2 \subseteq T$ τέτοια ώστε:

i $T_1 \vdash_{min} L$ και $T_2 \vdash_{min} \neg L$

ii $(\exists r' \in T_1, r \in T_2$ τ.ω. $r' < r) \Rightarrow (\exists r' \in T_1, r \in T_2$ τ.ω. $r < r')$

όπου $T \vdash_{min} L$ σημαίνει ότι ελάχιστο σύνολο για το οποίο ισχύει $T \vdash L$.

Για παράδειγμα, έστω ότι έχουμε το πρόγραμμα με κενή σχέση προτεραιότητας:

$$r_1 : p \leftarrow q$$
$$f_1 : q$$
$$r_2 : \neg p \leftarrow r$$
$$f_2 : r$$

Τότε το σύνολο $T_1 = r_1, f_1$ επιτίθεται στο σύνολο $T_2 = r_2, f_2$ αφού το T_1 εξάγει το συμπέρασμα p και το T_2 το $\neg p$. Αντίστοιχα το T_2 επιτίθεται στο T_1. Αν όμως προσθέσουμε την σχέση προτεραιότητας $r_1 > r_2$ τότε το T_1 εξακολουθεί να επιτίθεται στο T_2 αλλά το T_2 δεν επιτίθεται πια στο T_1. Αυτό συμβαίνει γιατί υπάρχει κάποιος κανόνας στο T_1 (ο r_1) που έχει μεγαλύτερη προτεραιότητα από κάποιον κανόνα του T_2 (τον r_2) ενώ το αντίθετο δεν συμβαίνει, δεν υπάρχει δηλαδή κανόνας του T_2 με προτεραιότητα μεγαλύτερη από κανόνες του T_1.

Ορισμός 2.1.3 (Συνεπές Σύνολο Κανόνων)
Έστω σύνολο κανόνων T. Το T είναι συνεπές σύνολο κανόνων αν για κάθε σταθερό άτομο (ground literal) k τέτοιο ώστε $T \vdash k$ δεν ισχύει $T \vdash \neg k$.

Για παράδειγμα το σύνολο $\{a, \neg a \leftarrow b\}$ είναι συνεπές ενώ το $\{a, \neg a \leftarrow b, b\}$ δεν είναι.

Ορισμός 2.1.4 (Αποδεκτό)
Έστω $(K, <)$ πρόγραμμα και T ένα κλειστό υποσύνολο του K. Τότε το T είναι αποδεκτό αν και μόνο αν:

i. το T είναι συνεπές σύνολο κανόνων, και

ii. για κάθε $T' \subseteq K$, εάν T' επιτίθεται στο T τότε T επιτίθεται στο T'.

Ορισμός 2.1.5 (Ασθενές Συμπέρασμα)
Έστω πρόγραμμα $(K, <)$ και k ένα σταθερό άτομο (ground literal). Τότε το k είναι ασθενές συμπέρασμα (credulous conclusion) του προγράμματος αν το k ισχύει σε κάποιο μέγιστο αποδεκτό υποσύνολο (maximal admissible set) του K.

Η έννοια του ασθενούς συμπεράσματος δεν είναι τόσο ισχυρή αφού για να ισχύει κάποιο άτομο ασθενώς αρκεί να ισχύει σε κάποιο αποδεκτό υποσύνολο του προγράμματος ενώ σε κάποιο άλλο μπορεί να μην ισχύει. Θα ορίσουμε την έννοια του ισχυρού συμπεράσματος

Ορισμός 2.1.6
Έστω πρόγραμμα $(K, <)$ και k ένα σταθερό άτομο. Τότε το k είναι ισχυρό συμπέρασμα (sceptical conclusion) του προγράμματος αν το k ισχύει σε κάθε μέγιστο αποδεκτό υποσύνολο (maximal admissible set) του K.

2.2 Διαδικασία Απόδειξης στον LPwNF

Αφού έχουμε δώσει τους βασικούς ορισμούς μπορούμε να περιγράψουμε μία διαδικασία απόδειξης στο Πλαίσιο του Λογικού Προγραμματισμού χωρίς άρνηση σαν αποτυχία [2]. Η διαδικασία αποτελείται από δύο είδη παραγωγής, που καλούνται τύπος Α και τύπος Β, οι οποίες εναλλάσσονται και τελικά κατασκευάζουν ένα αποδεκτό υποσύνολο κανόνων στο οποίο ένας δοσμένος στόχος ισχύει.

Αρχικά ξεκινάει μια παραγωγή τύπου Α η οποία προσπαθεί να αποδείξει τον αρχικό στόχο. Για να το πετύχουν αυτό οι παραγωγές τύπου Α κατασκευάζουν ένα μέρος της θεωρίας που είναι αρκετό για να παράγει κάποιον στόχο Q. Μέσα από κάθε παραγωγή τύπου Α μπορεί να ξεκινήσουν παραγωγές τύπου Β οι οποίες επιτίθενται στη παραγωγή τύπου Α. Οι παραγωγές τύπου Β ξεκινούν με ένα κανόνα r με κεφαλή k. Ο κανόνας r έχει χρησιμοποιηθεί προηγουμένως σε κάποια παραγωγή τύπου Α. Απαραίτητη προϋπόθεση για να ξεκινήσει η παραγωγή τύπου Β είναι να υπάρχει κάποιος κανόνας r' με κεφαλή $\neg k$ ο οποίος έχει μεγαλύτερη προτεραιότητα από τον r. Η παραγωγή τύπου Β προσπαθεί να αποδείξει τον κανόνα αυτό και να επιτεθεί κατά συνέπεια στην παραγωγή τύπου Α από την οποία ξεκίνησε.

Αντίστοιχα μέσα από μία παραγωγή τύπου Β μπορεί να ξεκινήσουν παραγωγές τύπου Α που στόχο έχουν να αμυνθούν κατά της επίθεσης από την Β παραγωγή. Αυτές οι τύπου Α παραγωγές ξεκινούν με κάποιο κανόνα s με κεφαλή l, ο οποίος έχει χρησιμοποιηθεί προηγουμένως σε κάποια παραγωγή τύπου Β, και προσπαθούν να αποδείξουν κάποιον κανόνα s' με κεφαλή $\neg l$. Όπως και στις παραγωγές τύπου Β έτσι και εδώ απαραίτητη προϋπόθεση είναι ο κανόνας s' να μην έχει χαμηλότερη προτεραιότητα από τον κανόνα s.

Αν η αρχική παραγωγή τύπου Α πετύχει να αποδείξει τον αρχικό στόχο ενώ καμία παραγωγή τύπου Β δεν πετύχει τότε το αρχικός στόχος έχει αποδειχθεί και επομένως αποτελεί ένα ασθενές συμπέρασμα της θεωρίας. Οι κανόνες που χρησιμοποιήθηκαν για την παραγωγή τύπου Α αποτελούν το αποδεκτό υποσύνολο κανόνων της θεωρίας που στηρίζουν τον αρχικό στόχο.

Με την διαδικασία που περιγράψαμε μπορούμε να αποδείξουμε αν κάποιος στόχος είναι ασθενές συμπέρασμα της θεωρίας. Για να είναι ισχυρό συμπέρασμα αρκεί να είναι ασθενές συμπέρασμα της θεωρίας, δηλαδή να πετύχει η παραπάνω διαδικασία για τον στόχο, αλλά συγχρόνως να αποτύχει ο υπολογισμός για την άρνηση του.

Σημειώνουμε εδώ ότι τους κανόνες που έχουν αποδειχθεί σε προηγούμενα στάδια αυτής της διαδικασίας μπορούν να χρησιμοποιηθούν σε επόμενα χωρίς να χρειάζεται να αποδειχθούν ξανά. Δηλαδή αν σε κάποια παραγωγή τύπου Α χρειάζεται να αποδείξουμε κάποιον κανόνα που είχε αποδειχθεί σε προηγούμενη παραγωγή τύπου Α δεν είναι ανάγκη να αποδειχθεί ξανά και θεωρούμε ότι ισχύει.

Ο παραπάνω υπολογισμός είναι ορθός, δηλαδή κάθε φορά που πετυχαίνει επιστρέφει ένα αποδεκτό υποσύνολο της θεωρίας και ο αρχικός στόχος αποτελεί ασθενές συμπέρασμα της. Αν αποτύχει τότε ο στόχος δεν είναι ασθενές συμπέρασμα της θεωρίας.

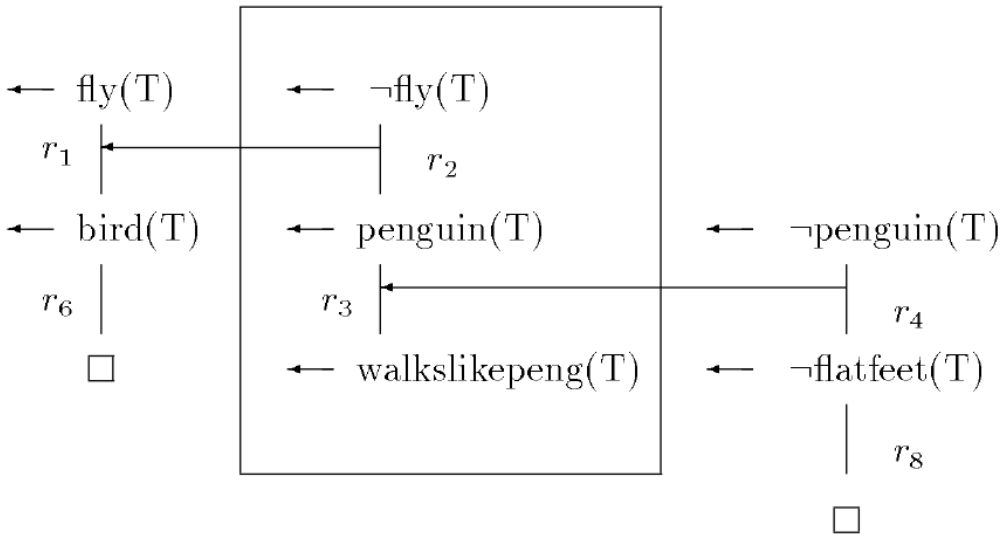
Παράδειγμα διαδικασίας απόδειξης

Έστω ότι έχουμε το πρόγραμμα με τους εξής λογικούς κανόνες:

$r_1 : fly(x) \leftarrow bird(x)$
$r_2 : \neg fly(x) \leftarrow penguin(x)$
$r_3 : penguin(x) \leftarrow walkslikepeng(x)$
$r_4 : \neg penguin(x) \leftarrow \neg flatfeet(x)$
$r_5 : bird(x) \leftarrow penguin(x)$
$r_6 : bird(t)$
$r_7 : walkslikepeng(t)$
$r_8 : \neg flatfeet(t)$

Οι προτεραιότητες ανάμεσα στους κανόνες είναι $r_2 > r_1$ και $r_4 > r_3$.

Έστω ότι ο αρχικός στόχος που θέλουμε να αποδείξουμε είναι $fly(t)$. Όλη η απόδειξη φαίνεται στο σχήμα 2.1 όπου η παραγωγή μέσα στο τετράγωνο είναι τύπου Β ενώ οι άλλες δύο είναι τύπου Α.



Σχήμα 2.1: Απόδειξη $fly(T)$, τα τόξα συμβολίζουν επίθεση

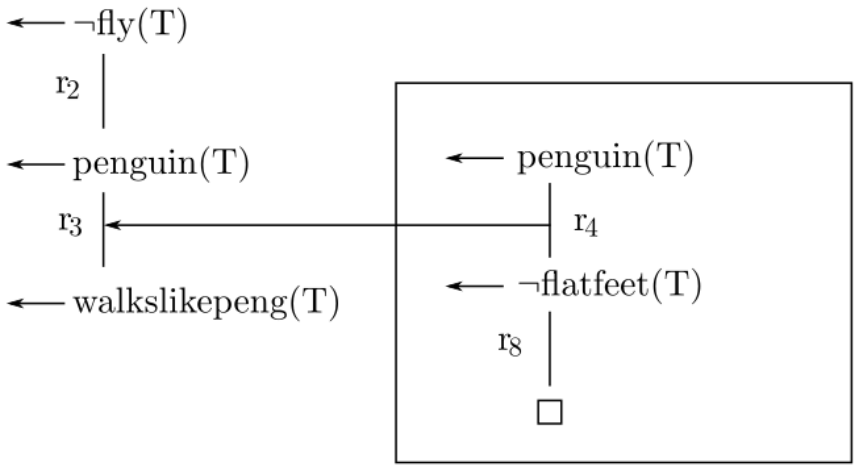
Με βάση την διαδικασία που έχουμε περιγράψει πρώτα θα ξεκινήσει μία παραγωγή τύπου Α η οποία θα προσπαθήσει να αποδείξει τον στόχο $fly(T)$. Για την απόδειξη του $fly(T)$ χρησιμοποιείται ο κανόνας r_1. Υπάρχει όμως ο κανόνας r_2 που επιτίθεται στον κανόνα r_1 καθώς η κεφαλή του r_2 είναι η άρνηση της κεφαλής του r_1 και ο r_2 έχει μεγαλύτερη προτεραιότητα από τον r_1. Έτσι ξεκινά μία παραγωγή τύπου Β που προσπαθεί να αποδείξει τον στόχο $\neg fly(t)$. Για να αποδειχθεί το $\neg fly(T)$ χρησιμοποιείται ο κανόνας r_2 και ο κανόνας r_3. Καθώς υπάρχει ο κανόνας r_4 ξεκινάει μία παραγωγή τύπου Α για να αμυνθεί κατά αυτής της επίθεσης. Αυτή η παραγωγή τύπου Α αποδεικνύει με επιτυχία τον στόχο $\neg penguin(T)$ οπότε βρέθηκε άμυνα στην επίθεση που έγινε στην αρχική παραγωγή τύπου Α και τελικά ο αρχικός στόχος $fly(T)$ αποδεικνύεται.

Ο στόχος $fly(T)$ είναι ασθενές συμπέρασμα της θεωρίας. Οι κανόνες που χρησιμοποιήθηκαν για να αποδείξουν το $fly(T)$ αποτελούν το αποδεκτό υποσύνολο κανόνων και είναι το $\{r_1, r_4, r_6, r_8\}$.

Αν η προτεραιότητα $r_4 > r_3$ δεν υπήρχε η παραπάνω διαδικασία θα πετύχαινε πάλι και θα είχε αποδειχθεί το $fly(T)$. Για να μπορέσει κάποιος κανόνας να χρησιμοποιηθεί για να αμυνθεί κατά κάποιας επίθεσης αρκεί να μην έχει χαμηλότερη προτε-

ραιότητα από τον κανόνα που χρησιμοποιήθηκε στην επίθεση. Άρα στο παράδειγμα χρειάζεται μόνο ο κανόνας r_4 να μην έχει χαμηλότερη προτεραιότητα από τον κανόνα r_3.

Για να αποδείξουμε ότι το $fly(T)$ αποτελεί ισχυρό συμπέρασμα της θεωρίας αρκεί να δείξουμε ότι η απόδειξη της άρνησης του στόχου, δηλαδή του $\neg fly(T)$ αποτυγχάνει. Στο παρακάτω σχήμα φαίνεται ότι η απόδειξη του $negfly(T)$ αποτυγχάνει καθώς δεν υπάρχει κάποια άμυνα στην επίθεση που ξεκινάει με τον κανόνα r_4. Άρα αφού έχουμε αποδείξει το $fly(T)$ και δεν μπορούμε να αποδείξουμε το $\neg fly(T)$ βγάζουμε το συμπέρασμα ότι το $fly(T)$ αποτελεί ισχυρό συμπέρασμα της θεωρίας.



Σχήμα 2.2: Απόδειξη $\neg fly(T)$

Οι τυπικοί ορισμοί των παραγωγών τύπου A και B και η απόδειξη της ορθότητας της διαδικασίας που περιγράψαμε υπάρχουν στο: Logic Programming without Negation as Failure - Yannis Dimopoulos and Antonis Kakas[2].

Κεφάλαιο 3

Γοργίας

3.1 Σύστημα Γοργίας

Το σύστημα Γοργίας[12] είναι ένα γενικό πλαίσιο επιχειρηματολογίας που συνδυάζει τις ιδέες της του συλλογισμού με προτιμήσεις (preference reasoning) και της λογικής απαγωγής (abduction) με τρόπο που διατηρεί τα προτερήματα και των δύο. Μπορεί να αποτελέσει την βάση για συλλογισμό με προσαρμόσιμες πολιτικές προτίμησης σε δυναμικά και εξελισσόμενα περιβάλλοντα, παρά τις ελλιπείς πληροφορίες[4]. Για παράδειγμα κατά την αγορά ενός αυτοκίνητο κάποιος προτιμάει κάποια χαρακτηριστικά και κάποιος άλλος διαφορετικά, στον χρονικό προγραμματισμό κάποιες προθεσμίες μπορεί να είναι πιο σημαντικές από άλλες, σε νομικούς συλλογισμούς[4] οι νόμοι υπόκεινται σε αρχές υψηλότερης τάξης όπως lex superior, lex posterior, κτλ.

3.2 Συντακτικό Γοργία

Το συντακτικό του Γοργία βασίζεται στην Prolog. Τα κατηγορήματα (predicates) του Γοργία χωρίζονται σε τρεις κατηγορίες:

- abducibles
- defeasible
- background (non-defeasible)

Τα literals αναπαρίστανται χρησιμοποιώντας όρους (terms) της Prolog. Ένα αρνητικό literal είναι ένας όρος της μορφής `neg(L)`.

Η γλώσσα για την αναπαράσταση των διαφόρων θεωριών (προβλημάτων) δίνεται από κανόνες της μορφής:

```
rule(Signature, Head, Body).
```

όπου:

- `Head` είναι ένα literal
- `Body` είναι μία λίστα από literals
- `Signature` είναι ένας σύνθετος όρος που αποτελείται από το όνομα του κανόνα μαζί με επιλεγμένες μεταβλητές από το `Head` και το `Body` του κανόνα.

Το ειδικό κατηγόρημα `prefer/2` χρησιμοποιείται για να κωδικοποιήσει τοπικά την σχετική δύναμη των κανόνων της θεωρίας. Για παράδειγμα το παρακάτω σημαίνει ότι ο κανόνας με signature `Sig1` έχει υψηλότερη προτεραιότητα από τον κανόνα με signature `Sig2` εφόσον οι συνθήκες στο `Body` ισχύουν:

```
rule(Signature, prefer(Sig1,Sig2), Body).
```

Τα abducible literals δηλώνονται χρησιμοποιώντας το ειδικό κατηγόρημα `abducible/2`, για παράδειγμα:

```
abducible(Literal, Preconditions).
```

Τέλος, η δήλωση `conflict(Sig1,Sig2)` δηλώνει ότι οι κανόνες με υπογραφές `Sig1` και `Sig2` συγκρούονται. Σε πολλές περιπτώσεις `conflict(Sig1,Sig2)` είναι αληθές ανν τα `Head` των κανόνων `Sig1` και `Sig2` είναι αντίθετα literals.

3.3 Χρήση Γοργία

Για την χρήση του Γοργία απαιτείται μία εγκατάσταση της SWI-Prolog. Με το συντακτικό που αναφέραμε μπορούμε να περιγράψουμε τα δικά μας προβλήματα προσθέτοντας τις παρακάτω δύο γραμμές στην κορυφή του αρχείου:

```
:- compile('../lib/gorgias.pl').
:- compile('../ext/lpwnf.pl').
```

Η πρώτη γραμμή φορτώνει το σύστημα ενώ η δεύτερη μία συλλογή από κανόνες που ορίζουν μία σχέση ιεράρχησης ανάμεσα στα επιχειρήματα η οποία χρησιμοποιείται από την σχέση «επιτίθεται» για να κωδικοποιήσει την σχετική δύναμη των επιχειρημάτων.

3.3.1 Απεικόνιση Γνώσης

Για να δηλώσουμε τους κανόνες, τις συγκρούσεις και τις προτιμήσεις μεταξύ τους θα χρησιμοποιήσουμε κανονικά όρους τις Prolog χρησιμοποιώντας τα κατηγορήματα/predicates του συστήματος Γοργία όπως τα ορίσαμε προηγουμένως. Έτσι για παράδειγμα για να δηλώσουμε ότι κάτι πετάει όταν είναι πουλί και ότι κάτι δεν πετάει όταν είναι πιγκουίνος γράφουμε:

```
rule(r1(X), fly(X), [bird(X)]).
rule(r2(X), neg(fly(X)), [penguin(X)]).
```

Σε αυτό το παράδειγμα φαίνεται ότι αυτοί οι δύο κανόνες έρχονται σε σύγκρουση όταν κάτι είναι πιγκουίνος και πουλί:

```
rule(f1, bird(tweety), []).
rule(f2, penguin(tweety), []).
```

Για να επιλύσουμε την σύγκρουση χρησιμοποιούμε το ειδικό κατηγόρημα `prefer/2`. Έτσι για αυτό το παράδειγμα έχουμε:

```
rule(pr1(X), prefer(r2(X), r1(X)), []).
```

3.3.2 Απάντηση Ερωτημάτων

Γενικά η κατασκευή μίας λύσης για ένα συγκεκριμένο ερώτημα μπορεί να γίνει σταδιακά, ξεκινώντας από ένα βασικό/αρχικό επιχείρημα, και προσθέτοντας στο αρχικό επιχείρημα κατάλληλες άμυνες για αυτό. Γενικά θα επικεντρωθούμε στον υπολογισμό μίας ειδικής κατηγορίας επιχειρημάτων, συγκεκριμένα στα αποδεκτά επιχειρήματα (admissible arguments). Διαισθητικά, ένα επιχείρημα είναι αποδεκτό αν αμύνεται ενάντια σε οποιαδήποτε επίθεση. Προφανώς ένα επιχείρημα που επιτίθεται στον εαυτό του δεν μπορεί να είναι αποδεκτό γιατί δεν υπάρχει επίθεση ενάντια στο κενό σύνολο.

Η διαδικασία υπολογισμού ενός αποδεκτού επιχειρήματος γίνεται σε δύο φάσεις: Στην πρώτη φάση ένας στόχος μειώνεται σε ένα κλειστό σύνολο που αποδεικνύει τον στόχο. Μετά, αυτό το αρχικό επιχείρημα επεκτείνεται με κατάλληλες άμυνες για κάθε επίθεση ενάντια στο αρχικό σύνολο. Μετά την επέκταση του αρχικού επιχειρήματος μπορεί να προκύψουν νέες συγκρούσεις και έτσι το σύστημα επαναλαμβάνει την διαδικασία μέχρι να μην υπάρχει άμυνα σε επίθεση ενάντια στο αρχικό επιχείρημα (δηλαδή δεν ήταν δυνατόν βρεθεί ένα αποδεκτό σύνολο επιχειρημάτων) ή δεν υπάρχουν άλλες επιθέσεις (δηλαδή προέκυψε ένα αποδεκτό επιχείρημα).

Πρακτικά τα ερωτήματα υποβάλλονται στο σύστημα ως εξής:

```
prove(Goals, Delta).
```

όπου `Goals` είναι μία λίστα θετικών (ή αρνητικών) literals και `Delta` είναι ένα αποδεκτό επιχείρημα για το συγκεκριμένο ερώτημα. Έτσι εκμεταλλευόμενοι την διαδικασία ενοποίησης όρων (term unification) που χρησιμοποιεί η Prolog όταν υποβάλλουμε το ερώτημα:

```
prove([neg(fly(tweety))],Delta).
```

το σύστημα Γοργίας απαντάει με το αποδεκτό επιχείρημα που στηρίζει το ερώτημα που θέσαμε:

```
Delta = [f2, r2(tweety)].
```

Αξίζει να σημειωθεί ότι επειδή υπάρχει ο κανόνα προτίμησης με όνομα `pr1` αν θέσουμε το ερώτημα `prove([fly(tweety)],Delta)` το σύστημα Γοργίας δεν θα μπορέσει να βρει λύση επειδή ο κανόνας `r2` επιτίθεται στον κανόνα `r1` και (καθώς είναι πιο ισχυρός) δεν υπάρχει άμυνα εναντίον του. Αν αφαιρέσουμε τον κανόνα προτίμησης `pr1` (προσθέτοντας ένα `%` στην αρχή της γραμμής) τότε το σύστημα μπορεί να βρει λύση (δηλαδή ένα αποδεκτό επιχείρημα) και για τα δύο ερωτήματα.

3.3.3 Δυναμικές Προτιμήσεις

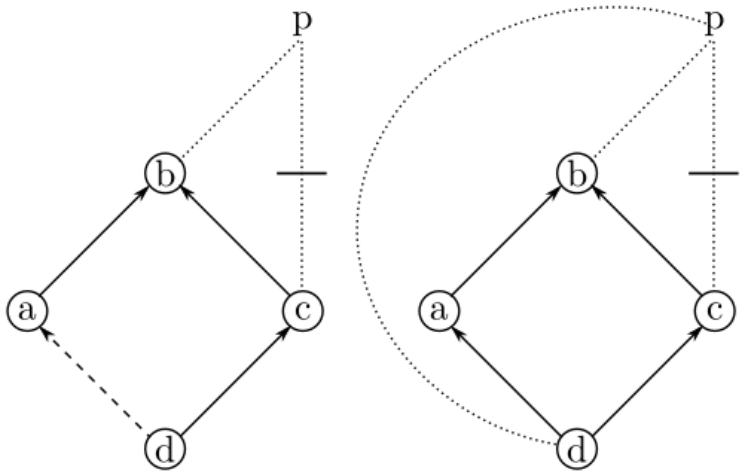
Κληρονομικότητα με εξαιρέσεις

Στο προηγούμενο παράδειγμα ο κανόνας προτίμησης ήταν στατικός (το `Body` του κανόνα ήταν η κενή λίστα). Θα μελετήσουμε ένα παράδειγμα με δυναμικούς κανόνες προτίμησης, δηλαδή προτιμήσεις που υπόκεινται σε συνθήκες.

Έστω ότι έχουμε την ιεραρχία κληρονομικότητας του παρακάτω σχήματος.

Τα βέλη αναπαριστούν τις σχέσεις μεταξύ κλάσεων και υποκλάσεων και οι γραμμές με τις τελείες τις ιδιότητες των αντικειμένων που ανήκουν σε αυτές τις κλάσεις. Από το σχήμα δηλαδή συμπεραίνουμε ότι τα αντικείμενα που ανήκουν στην κλάση `b` έχουν την ιδιότητα `p` ενώ αυτά που ανήκουν στην κλάση `c` δεν την έχουν. Το διακεκομμένο βέλος συμβολίζει την προσθήκη μίας σχέσης υποκλάσης μεταξύ `a` και `d` που θα συζητήσουμε παρακάτω όταν θα μιλήσουμε για προτιμήσεις υψηλότερης τάξης. Για να περιγράψουμε αυτή την ιεραρχία στον Γοργία θα χρησιμοποιήσουμε τους παρακάτω κανόνες/ορισμούς/δεδομένα:

```
rule(f1, def_subclass(a,b), []).
rule(f2, def_subclass(c,b), []).
rule(f3, def_subclass(d,c), []).
rule(f4, def_is_in(x1,a),   []).
```



Σχήμα 3.1: Ιεραρχία Κληρονομικότητας

```
rule(f5, def_is_in(x2,c),   []).
rule(f6, def_is_in(x3,d),   []).

rule(d1(X), has(X,p),      [is_in(X,b)]).
rule(d2(X), neg(has(X,p)), [is_in(X, c)]).

rule(pr1(X), prefer(d2(X), d1(X)), []).
```

Με βάση αυτούς τους κανόνες το αντικείμενο x1 ανήκει στην κλάση a, το x2 στην c και το x3 στην d. Το has(X,P) σημαίνει «το αντικείμενο X έχει την ιδιότητα P». Ακολουθούν οι κανόνες που αναπαριστούν τις γενικές ιδιότητες της σχέσης subclass και is_in (τα αξιώματα που είναι ανεξάρτητα από το συγκεκριμένο πρόβλημα) και επιτρέπουν στον Γοργία να επιχειρηματολογήσει:

```
% General properties of subclass and is_in
rule(r1(C1,C2), subclass(C1,C2), [def_subclass(C1,C2)]).
rule(r2(C0,C2), subclass(C0,C2), [def_subclass(C0,C1), subclass(C1,C2)]).
rule(r3(X,C),   is_in(X,C),      [def_is_in(X,C)]).
rule(r4(X,C),   is_in(X,C),      [subclass(S,C), is_in(X,S)]).
```

Είναι εύκολο να ελέγξουμε ότι η περιγραφή που έχουμε κάνει είναι συνεπής και το λογικό πρόγραμμα κατασκευάζει αποδεκτά επιχειρήματα για το has(x1,p), το neg(has(x2,p)) και το neg(has(x3,p)). Ενδεικτικά:

```
?- prove([has(x3,p)],Delta).
false.
```

```
?- prove([neg(has(x3,p))],Delta).
Delta = [f6, r3(x3, d), f3, r1(d, c), r4(x3, c), d2(x3)] .
```

Προτιμήσεις υψηλότερης τάξης

Θα επεκτείνουμε το παραπάνω πρόγραμμα ώστε να αναπαραστήσουμε το δεδομένο ότι το d είναι υποκλάση του a και έναν κανόνα προτίμησης για να εκφράσουμε ότι τα αντικείμενα του d έχουν την ιδιότητα p:

```
rule(f7, def_subclass(d,a), []).
rule(pr2(X), prefer(d1(X), d2(X)), [is_in(X,a)]).
```

Τώρα και το `d1(X) < d2(X)` και το `d2(X) < d1(X)` μπορούν να αποδειχθούν για το x3 οπότε υπάρχει ένα αποδεκτό υποσύνολο και για το `has(x3,p)` και για το `neg(has(x3,p))`:

```
?- prove([has(x3,p)],_).
true .

?- prove([neg(has(x3,p))],_).
true .
```

Ένας τρόπος για να επιλύσουμε αυτή την σύγκρουση είναι να αλλάξουμε τον προηγούμενο κανόνα προτίμησης `pr1(X)` σε:

```
rule(pr1(X), prefer(d2(X), d1(X)), [neg(is_in(X,a))]).
```

Κάτι τέτοιο όμως δεν περιγράφει το πρόβλημα μας με φυσικό τρόπο και θα οδηγούσε σε μία συνδυαστική έκρηξη στον αριθμό των όρων στο σώμα των κανόνων προτίμησης (ειδικά αν είχαμε μία μεγαλύτερη και πιο πολύπλοκη ιεραρχία) υποβαθμίζοντας την εκφραστικότητα της γλώσσας και αυξάνοντας την πιθανότητα λάθους. Ο Γοργίας όμως μπορεί να δεχθεί κανόνες προτίμησης που ορίζουν προτιμήσεις πάνω σε κανόνες προτίμησης, δηλαδή προτεραιότητες υψηλότερης τάξης. Για παράδειγμα ο παρακάτω κανόνας ορίζει ότι ένα αντικείμενο της κλάσης d έχει την ιδιότητα p παρόλο που η d είναι υποκλάση της c:

```
rule(pr3(X), prefer(pr2(X), pr1(X)), [is_in(X,a)]).
```

Η χρήση προτεραιοτήτων υψηλότερης τάξης μπορεί να βρει εφαρμογή σε νομικά επιχειρήματα. Για παράδειγμα όταν υπάρχουν αντίθετοι μεταξύ τους νόμοι όπου ο ένας επιτρέπει κάποια πράξη την οποία κάποιος άλλος νόμος την απαγορεύει προκύπτει αντινομία[11]. Για να γίνει άρση τέτοιου είδους αντινομιών χρησιμοποιούνται νομικές αρχές όπως ότι ο νεότερος νόμος υπερισχύει του παλαιότερου ή ότι ο ανώτερος

νόμος υπερισχύει του κατώτερου. Μία τέτοια σχέση μπορούμε να την εκφράσουμε στο Γοργία χρησιμοποιώντας προτεραιότητες υψηλότερης τάξης:

```
rule(lex_posterior(X,Y), prefer(X,Y), [newer(X,Y)]).
rule(lex_superior(X,Y),  prefer(Y,X), [state_law(X),federal_law(Y)]).

rule(prpr, prefer(lex_superior(X,Y), lex_posterior(X,Y)), []).
```

3.3.4 Απαγωγή στον Γοργία

Μέχρι στιγμής είδαμε πώς ο Γοργίας μπορεί να καταλήξει σε συμπέρασμα όταν υπάρχουν επιχειρήματα τα οποία είναι συγκρουόμενα και έχουν διαφορετική ισχύ μεταξύ τους. Αυτές οι προτιμήσεις μπορεί να είναι δυναμικές και να εξαρτώνται από άλλες συνθήκες οι οποίες μπορεί να αλλάζουν και να είναι και αυτές μέρος της επιχειρηματολογίας. Σε ένα τέτοιο δυναμικό περιβάλλον είναι δυνατό να υπάρχει ατελής γνώση που μας εμποδίζει να κατασκευάσουμε αποδεκτά επιχειρήματα καθώς αναγκαία πληροφορία μπορεί να λείπει. Σε τέτοιες περιπτώσεις ο Γοργίας μπορεί να χρησιμοποιήσει υποθετικό συλλογισμό μέσω απαγωγής (abduction).

Για να λειτουργήσει η απαγωγή θα ορίζουμε ένα ξεχωριστό σύνολο κατηγορημάτων ως υποθετικά χρησιμοποιώντας το `abducible/2` που εκφράζουν την ελλιπή ή άγνωστη πληροφορία στο συγκεκριμένο πρόβλημα. Αργότερα όταν θα υποβάλλουμε ένα ερώτημα στο σύστημα θα επεκτείνει την θεωρία με δεσμευμένα απαγόμενα κατηγορήματα και θα κατασκευάζει ένα αποδεκτό επιχείρημα για τον στόχο χρησιμοποιώντας τα.

Έστω για παράδειγμα ότι έχουμε μία βάση γνώσης που περιγράφει πότε κάποιος χρήστης σε ένα λειτουργικό σύστημα UNIX έχει πρόσβαση να διορθώσει ένα αρχείο:

```
rule(owner_changes_permissions(U,F),
     can_change_permissions(U,F),
     [is_file_owner(U,F)]).
rule(root_changes_permissions(U,F),
     can_change_permissions(U,F),
     [is_root(U)]).
rule(change_permissions(U,F),
     has_write_permission(U,F),
     [can_change_permissions(U,F)]).
rule(write_file(U,F),
     can_write_file(U,F),
     [has_write_permission(U,F)]).
```

Φυσικά θα μπορούσαμε να επεκτείνουμε το παράδειγμα με περισσότερους κανόνες. Ορίζουμε επίσης ποια επιχειρήματα θα προκύψουν μέσω απαγωγής:

```
abducible(is_root(_),                []).
abducible(is_file_owner(_,_),        []).
abducible(has_write_permission(_,_), []).
```

Όταν θέσουμε στον Γοργία το ερώτημα αν ο χρήστης έχει πρόσβαση στο αρχείο παίρνουμε την απάντηση:

```
?- prove([can_write_file(user,file)],Delta).
Delta = [ass(is_file_owner(user, file)),
         owner_changes_permissions(user, file),
         change_permissions(user, file),
         write_file(user, file)] ;
Delta = [ass(is_root(user)),
         root_changes_permissions(user, file),
         change_permissions(user, file),
         write_file(user, file)] ;
Delta = [ass(has_write_permission(user, file)),
         write_file(user, file)] ;
false.
```

Με την διαδικασία δηλαδή που περιγράψαμε πιο πριν η θεωρία επεκτείνεται με τα απαγόμενα επιχειρήματα και με αυτά κατασκευάζονται τα αποδεκτά επιχειρήματα που δείχνουν και τα βήματα που πρέπει να ακολουθήσει ο χρήστης για να αποκτήσει πρόσβαση στο αρχείο.

Φυσικά η απαγωγή μπορεί να συνδυαστεί με στατικές ή δυναμικές προτιμήσεις. Υπάρχει ένα παράδειγμα στο τέλος.

3.4 Εφαρμογή Γοργία: Βοηθός Επιχειρηματικότητας

Στα προηγούμενα περιγράψαμε τις δυνατότητες του Γοργία για επιχειρηματολογία με προτιμήσεις (πρώτης και ανώτερης τάξη) και λογική απαγωγή σε δυναμικά και εξελισσόμενα περιβάλλοντα όπου υπάρχει ελλιπής γνώση. Αναφέραμε επίσης στα παραδείγματα πώς μπορεί να χρησιμοποιηθεί για να μοντελοποιήσει με ακριβή τρόπος τις σχέσεις ανάμεσα στους νόμους και την δυνατότητα να απαντάει σε ερωτήματα που απορρέουν από αυτούς. Είναι προφανές ότι ένα τέτοιο σύστημα μπορεί να βρει εφαρμογές σε διάφορους τομείς της καθημερινότητας που εκμεταλλεύονται τα χαρακτηριστικά του.

Σαν εφαρμογή θα περιγράψουμε πώς μπορούμε να χρησιμοποιήσουμε τον Γοργία για να κατασκευάσουμε έναν επιχειρηματικό βοηθό/σύμβουλο που θα δίνει πληροφορίες και απαντήσεις σε συγκεκριμένα ερωτήματα σε επιχειρήσεις που δραστηριοποιούνται σε μία περιοχή. Ένας επιχειρηματικός βοηθός για να είναι χρήσιμος θα πρέπει να πληρεί τις παρακάτω προϋποθέσεις:

1. Να απαντάει σε ερωτήματα σχετικά με επιχειρηματικότητα.
2. Να αιτιολογεί τις απαντήσεις με αναφορές σε συγκεκριμένους νόμους.
3. Να λειτουργεί με μη ολοκληρωμένη γνώση.
4. Να παρέχει προτάσεις προς τον χρήστη.

Επιθυμητά χαρακτηριστικά ενός τέτοιου συστήματος:

6. Να είναι εύκολα επεκτάσιμο ώστε στο μέλλον να μπορεί να εμπλουτιστεί και να ενημερωθεί με νέους νόμους και σενάρια. Η προσθήκη νέων νόμων και σεναρίων πρέπει να γίνεται με τρόπο που δεν θα απαιτεί την αλλαγή των ήδη υπαρχόντων ώστε να μην υπάρχει κίνδυνος εισαγωγής σφαλμάτων στην ήδη δοκιμασμένη και ελεγμένη λειτουργία.
7. Να έχει την δυνατότητα να εξελιχθεί σε κεντρική πύλη πληροφοριών, ενημέρωσης και προτάσεων για όλα τα θέματα που έχουν να κάνουν με την επιχειρηματικότητα και την ανάπτυξη στην περιοχή.

Όπως φαίνεται και από τις προδιαγραφές που έχουμε θέσει ο χρήστης του συστήματος δεν έχει πλήρη γνώση των νόμων οπότε δεν γνωρίζει αν η εταιρεία του εμπίπτει στις διατάξεις τους. Το σύστημα πρέπει να μπορεί να δίνει μία γενική απάντηση ακόμα και δεν έχει άλλα δεδομένα για την εταιρία. Κατά την εγγραφή στο σύστημα ο χρήστης θα έχει την δυνατότητα να συμπληρώσει προαιρετικά μία φόρμα με στοιχεία για την εταιρία του. Αυτά τα στοιχεία θα χρησιμοποιούνται από τον Γοργία για να κατασκευάζονται αποδεκτά επιχειρήματα που θα δίνουν απάντηση στα ερωτήματα που θα υποβάλλει ο χρήστης.

Τα ερωτήματα που θα υποβάλλει ο χρήστης στο σύστημα έχουν σχέση με την επιχειρηματικότητα, πχ αν η εταιρεία του εμπίπτει στις διατάξεις κάποιου νόμου, αν δικαιούται να συμμετάσχει σε κάποιο διαγωνισμό του δημοσίου, αν δικαιούται επιδότηση κτλ. Όπως είπαμε ο χρήστης με την εισαγωγή του δεν έχει δώσει όλες τις πληροφορίες που αφορούν την εταιρεία (πχ γιατί δεν ξέρει ακόμα ποιες πληροφορίες χρειάζονται) πρέπει παρόλα να μπορεί να λάβει απαντήσεις στα ερωτήματα που υποβάλλει. Ο Γοργίας γενικά θα απαντάει αρνητικά καθώς δεν έχουν δωθεί αρκετές πληροφορίες/δεδομένα για να μπορέσει να κατασκευάσει ένα αποδεκτό επιχείρημα.

Όταν η απάντηση ενός ερωτήματος είναι αρνητική θα προσφέρει στον χρήστη μία επιλογή για να δει κάτω από ποιες προϋποθέσεις μπορεί το αρχικό ερώτημα να λάβει θετική απάντηση. Τότε χρησιμοποιώντας λογική απαγωγή (κατηγόρημα adbucible) θα επεκτείνει την θεωρία με αυτά τα απαγόμενα επιχειρήματα και θα κατασκευάζει ένα αποδεκτό επιχείρημα το οποίο και θα παρουσιάζει στον χρήστη με την μορφή ερωτημάτων. Για παράδειγμα, αν το ερώτημα ήταν:

- «Δικαιούμαι χρηματοδότηση από το ΕΣΠΑ;»

τότε τα απαγόμενα επιχειρήματα που θα παρουσιάζονται στον χρήστη με την μορφή ερωτημάτων θα μπορούσε να είναι:

- «Απασχολείται στην εταιρία σας νέους ερευνητές κάτω των 30 ετών;»,
- «Έχετε παράρτημα και απασχολείται προσωπικό σε ορεινές περιοχές;»,
- «Η δραστηριότητα της εταιρίας σας βοηθά άμεσα ασθενείς ομάδες του πληθυσμού;», κτλ

Σε περίπτωση που η εταιρία πληρεί αυτές τις προϋποθέσεις τότε ο χρήστης θα απαντήσει θετικά και θα έχει μάθει για κάποιον νόμο ή διάταξη που δεν γνώριζε πριν. Συγχρόνως, μαζί με την τελική απάντηση ο χρήστης θα έχει τις σχετικές παραπομπές σε νόμους. Δηλαδή το σύστημα θα δικαιολογεί τις απαντήσεις του καθώς θα παρουσιάζει το αποδεκτό επιχείρημα στον χρήστη. Επίσης με αυτόν τον τρόπο το σύστημα θα μάθει περισσότερες πληροφορίες για τον χρήστη, χωρίς να τον αναγκάζει να συμπληρώνει φόρμες, και θα χρησιμοποιήσει αυτήν την πληροφορία για να απαντήσει στο μέλλον σε άλλα ερωτήματα ή να ενημερώσει τις απαντήσεις σε παλαιότερα ερωτήματα που με τις νέες πληροφορίες μπορεί να έχουν θετική απάντηση.

Αν ο χρήστης δεν πληρεί αυτές τις προϋποθέσεις μπορεί να τις χρησιμοποιήσει σαν επιχειρηματικές προτάσεις για αλλαγές που μπορεί να κάνει στην εταιρία ή να ζητήσει από τον Γοργία ένα νέο σενάριο χρησιμοποιώντας ουσιαστικά την διαδικασία ενοποίησης της Prolog.

Τέλος ο χρήστης θα έχει την δυνατότητα να ζητάει από το σύστημα ερωτήματα που τον ενδιαφέρουν με τα οποία θα ενημερώνεται αυτόματα στην κεντρική σελίδα. Ενδεικτικά:

- Διαγωνισμοί σχετικοί με το πεδίο στο οποίο δραστηριοποιείται
- Ημερομηνίες και προθεσμίες υποβολής δικαιολογητικών και χρηματοδότησης

Ο διαχειριστής επίσης του συστήματος μπορεί να ενημερώνει τους χρήστες που πληρούν τις προϋποθέσεις για νέα προγράμματα και νόμους και έτσι να υπάρχει καλύτερη εκμετάλλευση των πόρων και μεγαλύτερη συμμετοχή σε νέες πρωτοβουλίες.

Πολύ σημαντική είναι η δυνατότητα για επέκταση του συστήματος που δίνεται στον διαχειριστή. Οι νέοι νόμοι μπορεί να έρχονται σε σύγκρουση με τους παλιούς. Επίσης υπάρχει περίπτωση νέοι νόμοι που εισάγονται να έρχονται σε σύγκρουση με τοπικές διατάξεις και εγκυκλίους. Είναι βασικό η εισαγωγή των νέων νόμων να γίνεται με τρόπο που δεν θα απαιτεί την αλλαγή των ήδη υπάρχοντων. Οι συγκρούσεις θα επιλύονται με γενικές και ειδικές προτεραιότητες που θα έχουν υλοποιηθεί μέσα στο σύστημα και είναι φυσιολογικοί στην νομολογία (πχ lex posterior ότι δηλαδή ο νεότερος νόμος έχει μεγαλύτερη ισχύ από τον παλαιότερο, lex superior ότι δηλαδή ο κρατικός νόμος ή το σύνταγμα έχει μεγαλύτερη ισχύ από μία τοπική διάταξη κτλ) και με προτεραιότητες ανώτερης τάξης (πχ το lex superior υπερισχύει του lex posterior, δηλαδή ο ανώτερος νόμος έχει προταιρεότητα από νεώτερους νόμους). Με αυτόν τον τρόπο εξασφαλίζεται η συνέπεια, η ορθότητα και η εύκολη επέκταση του συστήματος.

Κεφάλαιο 4

Σύνδεση Prolog και Γοργία με Java

4.1 Εισαγωγή

Το σύστημα Γοργίας[12] όπως το περιγράψαμε στην προηγούμενη ενότητα μπορεί να να αποτελέσει την βάση για συλλογισμό σε δυναμικά και εξελισσόμενα περιβάλλοντα συνδυάζοντας τις ιδέες του συλλογισμού με προτιμήσεις και της λογικής απαγωγής. Σε τέτοια περιβάλλοντα είναι πολύ πιθανό να υπάρχει ελλιπής πληροφορία και υποθέσεις που είχαμε κάνει προηγουμένως στο μέλλον να πάψουν να ισχύουν. Οπότε χρειαζόμαστε έναν τρόπο να ενημερώνουμε το σύστημα για αλλαγές που γίνονται στο περιβάλλον του προβλήματος, προσθέτοντας και αφαιρώντας συνθήκες ενώ οι κανόνες του κόσμου παραμένουν σταθεροί. Επίσης είναι επιθυμητό να μπορούμε εύκολα να χρησιμοποιήσουμε τον Γοργία και το πλαίσιο επιχειρηματολογίας που προσφέρει με διαφορετικά συστήματα που δεν είναι υλοποιημένα στην SWI-Prolog[9]. Αυτά θα τροφοδοτούν τον Γοργία με πληροφορίες για τον κόσμο και θα εκμεταλλεύονται την δυνατότητα του για δημιουργία αποδεκτών επιχειρημάτων όταν υπάρχει ελλιπής ή αντικρουόμενη γνώση με χρήση πολιτικών προτιμήσεων.

Για να τα πετύχουμε αυτά θα αναπτύξουμε μία σύνδεση του Γοργία με την Java. Η σύνδεση αυτή πρέπει να είναι αρκετά γενική ώστε να μπορεί να χρησιμοποιηθεί σε διαφορετικά προβλήματα. Επίσης μέσω αυτής της σύνδεσης θα μπορούμε να χρησιμοποιήσουμε τον Γοργία και την Prolog με τον ίδιο τρόπο ώστε να μπορούμε να αντιπαραβάλλουμε τα αποτελέσματα και τις διαφορές τους.

4.2 Υλοποίηση

Για να επιτευχθεί η επικοινωνία της Java με τον Γοργία και την SWI-Prolog θα χρησιμοποιήσουμε την βιβλιοθήκη διασύνδεσης JPL[10] που προμηθεύει η τελευταία. Αυτό θα επιτρέψει σε εφαρμογές γραμμένες σε Java να χρησιμοποιούν οποιαδήποτε από τις βιβλιοθήκες και τα κατηγορήματα της Prolog και κατά συνέπεια και τα κατηγορήματα του Γοργία. Αυτό θα μας δώσει την δυνατότητα να ξεχωρίσουμε το λογικό κομμάτι της εφαρμογής και μάλιστα να μπορούμε να το διορθώσουμε χωρίς να χρειάζεται να μεταγλωττίσουμε ξανά το υπόλοιπο πρόγραμμα.

Για να είναι η σύνδεση αρκετά γενική και να μπορούμε να χρησιμοποιούμε με τον ίδιο τρόπο τα κατηγορήματα του Γοργία αλλά και την απλή Prolog θα υλοποιήσουμε ένα interface της Java με όνομα Logic που θα ορίζει τις παρακάτω συναρτήσεις:

```
void load(String file);
void claim(String condition);
void claim(String condition, String label);

boolean test(String condition);
List<String> query(String variable, String condition);
List<List<String>> query(List<String> variables, String condition);
List<List<String>> why();

List<String> listPredicates();
void disclaimAll();
void disclaimLast();
void disclaim(String condition);
String negate(String condition);
```

Οι συναρτήσεις αυτές είναι αφηρημένες και υλοποιούνται σε ξεχωριστές κλάσεις για τον Γοργία και την Prolog. Ανάλογα με τις ανάγκες του προβλήματος αρχικοποιούμε το κατάλληλο αντικείμενο και καλούμε τις συναρτήσεις που ορίζει το `Logic` interface:

```
Logic prolog = new Prolog();
prolog.load("prolog_rules.pl");
Logic gorgias = new Gorgias();
gorgias.load("gorgias_rules.pl");
```

Με την `load()` φορτώνουμε ένα αρχείο κανόνων της Prolog ή του Γοργία που περιέχει τους κανόνες συμπερασματολογίας για το πρόβλημα. Για παράδειγμα αν θέλαμε να κανόνες που περιγράφουν ότι κάποιος είναι γονέας αν είναι μητέρα ή πατέρας και χρησιμοποιούσαμε την κλάση της Prolog τότε το αρχείο κανόνων θα περιείχε:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
```

Αντίστοιχα αν χρησιμοποιούσαμε την κλάση του Γοργία τότε σύμφωνα με την σύνταξη του Γοργία που περιγράψαμε το αρχείο θα περιείχε:

```
rule(r1(X,Y), parent(X,Y), [mother(X,Y)]).
rule(r2(X,Y), parent(X,Y), [father(X,Y)]).
```

Αυτό το αρχείο πρέπει να περιέχει μόνο τους κανόνες για το πρόβλημα μας καθώς τα δεδομένα είναι δυναμικά και δεν είναι γνωστά από πριν. Επίσης θεωρούμε ότι οι κανόνες του κόσμου παραμένουν σταθεροί και μόνο τα δεδομένα είναι δυναμικά και άγνωστα πριν αρχίσουμε να εκτελούμε το πρόγραμμα. Φυσικά αργότερα έχουμε την δυνατότητα να φορτώσουμε περισσότερους κανόνες χρησιμοποιώντας ξανά την συνάρτηση `load()`. Για να φορτώσουμε δεδομένα χρησιμοποιούμε την συνάρτηση `claim()`. Για παράδειγμα με τους προηγούμενους κανόνες μπορούμε να κάνουμε τους ισχυρισμούς:

```
l.claim("mother(alice,charlie)");
l.claim("father(bob,charlie)");
```

Παρατηρούμε ότι ανεξάρτητα από ποια υλοποίηση του interface `Logic` χρησιμοποιήσαμε υποβάλλουμε τους ισχυρισμούς στο σύστημα με τον ίδιο τρόπο. Ανάλογα με την υλοποίηση αυτοί μεταφράζονται εσωτερικά σε κανόνες Prolog ή σε κανόνες Γοργία. Οι παραπάνω ισχυρισμοί στην Prolog μετατρέπονται στα παρακάτω clauses:

```
mother(alice,charlie).
father(bob,charlie).
```

Αν χρησιμοποιούμε Γοργία θα μεταφραστούν σε κατάλληλα `rule/3` predicates του Γοργία. Αυτόματα θα δημιουργηθεί μία ετικέτα για αυτούς τους ισχυρισμούς ώστε να μπορεί ο Γοργίας να τους ξεχωρίσει. Υπάρχει επίσης και μία λίγο διαφορετική κλήση της `claim()` που μας επιτρέπει να προσδιορίσουμε το rule label που θέλουμε (το οποίο αγνοείται όταν χρησιμοποιούμε Prolog). Στο προηγούμενο παράδειγμα οι ισχυρισμοί θα μεταφραστούν στα παρακάτω κατηγορήματα του Γοργία:

```
rule(f1, mother(alice,charlie), []).
rule(f2, father(bob,charlie), []).
```

Εσωτερικά και οι δύο υλοποιήσεις κρατάνε την χρονική σειρά των ισχυρισμών καθώς και την αντιστοιχία των ισχυρισμών με τους κανόνες που προέκυψαν ώστε να μπορούμε να τους αποσύρουμε αργότερα με τις συναρτήσεις `disclaim()`, `disclaimLast()`, `disclaimAll()` ή να τους ανασύρουμε αργότερα με την συνάρτηση `listPredicates()`. Αυτό υλοποιείται με ένα πίνακα κατακερματισμού (HashMap) που διατηρεί την αντιστοιχία των ισχυρισμών που υποβάλλαμε στο σύστημα σε ισχυρισμούς στην Prolog

ή στον Γοργία με την χρονική σειρά που έγιναν αυτή. Αυτό επίσης μας μας ότι στο σύστημα μας δεν θα υπάρχουν διπλότυποι ισχυρισμοί (αν υποβάλλουμε έναν ισχυρισμό που ήδη υπάρχει το σύστημα θα το αναγνωρίσει και θα τον αγνοήσει).

Αφού έχουμε φορτώσει τους κανόνες και τα δεδομένα στην βάση μπορούμε να υποβάλλουμε ερωτήματα χρησιμοποιώντας τις συναρτήσεις `test()` και `query()`. Χρησιμοποιούμε την `test()` για να στείλουμε ένα απλό ερώτημα και να λάβουμε ως απάντηση αληθές/ψευδές. Πχ για το προηγούμενο παράδειγμα:

```
prolog.test("parent(alice,charlie)");
gorgias.test("parent(alice,charlie)");
```

Σε αυτό το ερώτημα και η Prolog και ο Γοργίας θα απαντήσουν με τον ίδιο τρόπο. Εσωτερικά το σύστημα Γοργίας για να μας απαντήσει με αυτόν τον τρόπο έχει κατασκευάσει ένα αποδεκτό επιχείρημα, δηλαδή ένα υποσύνολο από κανόνες και δεδομένα που στηρίζουν αυτήν την θέση. Με την συνάρτηση `why()` μπορούμε να ζητήσουμε από το σύστημα μία λίστα από τις ετικέτες των κανόνων και των δεδομένων που αποτελούν το αποδεκτό επιχείρημα:

```
gorgias.test("parent(alice,charlie)");
list = gorgias.why();
```

Με την συνάρτηση `query()` μπορούμε να υποβάλλουμε πιο σύνθετα ερωτήματα όπου εκμεταλλευόμαστε την διαδικασία ενοποίησης που προσφέρει η Prolog (και κατά συνέπεια και ο Γοργίας). Έτσι μπορούμε να βρούμε τις δυνατές τιμές των μεταβλητών που κάνουν το αληθές το ερώτημα που υποβάλλουμε. Για παράδειγμα για να δούμε ποιοι είναι οι γονείς κάποιου:

```
l.query("X", "parent(X,charlie)");
```

Η συνάρτηση αυτή επιστρέφει μία λίστα με όλες τις δυνατές τιμές που ικανοποιούν την συνθήκη που υποβάλλαμε. Η συνάρτηση `query()` είναι υπερφορτωμένη και μπορούμε να την καλέσουμε με ορίσματα διαφορετικού τύπου. Έτσι μπορούμε να ζητήσουμε από το σύστημα να ενοποιήσει περισσότερες μεταβλητές. Για παράδειγμα μπορούμε να βρούμε τις δυάδες όλων των δυνατών γονέων-παιδιών:

```
List<String> variables = new ArrayList<>();
variables.add("X");
variables.add("Y");

l.query(variables, "parent(X,Y)");
```

Όπως και με την συνάρτηση `test()` και εφόσον έχουμε χρησιμοποιήσει την κλάση του Γοργία μπορούμε να ζητήσουμε την λίστα των κανόνων και των δεδομένων που αποτελούν το αποδεκτό επιχείρημα με την συνάρτηση `why()`. Η λίστα που επιστρέ-

φει η συνάρτηση θα περιέχει τόσο στοιχεία όσες ήταν οι ενοποιήσεις που βρήκε το σύστημα. Κάθε ένα από τα στοιχεία της λίστας είναι μία λίστα από τις ετικέτες των κανόνων και των δεδομένων που αποτελούν το αποδεκτό επιχείρημα:

```
gorgias.query("X", "parent(X,charlie)");
List<List<String>> list = gorgias.why();
```

Τέλος με την συνάρτηση `negate()` μπορούμε να αντιστρέφουμε το ερώτημα που υποβάλλουμε με τις `test()` και query(). Για παράδειγμα:

```
l.test(l.negate("parent(alice,charlie)"));
```

Αν το `l` αντιπροσωπεύει ένα αντικείμενο κλάσης Prolog θα χρησιμοποιηθεί το κατηγόρημα `not()` της Prolog. Αν είναι κλάσης Gorgia θα χρησιμοποιηθεί το κατηγόρημα `neg()` που δέχεται ο Γοργίας.

Στο παράρτημα υπάρχει ο κώδικας αυτών των κλάσεων.

4.3 Σύγκριση Prolog - Γοργία

Εφόσον η κλάση του Γοργία και η κλάση της Prolog υλοποιούν την ίδια διασύνδεση Logic που περιγράψαμε μπορούν να χρησιμοποιηθούν με τον ίδιο ακριβώς τρόπο. Το μόνο που αλλάζει είναι η σύνταξη του αρχείου κανόνων. Ακόμα και αν το αρχείο κανόνων του Γοργία και της Prolog είναι ισοδύναμα (που δεν μπορεί να γίνει) πάντα ο Γοργίας έχει παραπάνω δυνατότητες που δεν τις έχει η Prolog, όπως για παράδειγμα να «αιτιολογεί» τις απαντήσεις παρουσιάζοντας το αποδεκτό σύνολο κανόνων που χρησιμοποιήθηκε για την απόδειξη.

Για παράδειγμα, με βάση τους κανόνες της Prolog και του Γοργία που περιγράψαμε στην προηγούμενη ενότητα μπορούμε να φτιάξουμε το παρακάτω πρόγραμμα Java:

```
import java.util.List;
import logic.*;

public class logic2 {

    public static void testLogic(Logic l) {
        l.claim("mother(alice,charlie)");
        l.claim("father(bob,charlie)");
        l.claim("mother(alice,david)");
        l.claim("father(bob,david)");
```

```
        System.out.println("Is alice mother of charlie?");
        System.out.println(l.test("mother(alice,charlie)"));
        System.out.println("Why?");
        List<List<String>> explanations;
        explanations = l.why();
        System.out.println(explanations.get(0));
    }

    public static void main(String[] args) {
        Logic prolog = new Prolog();
        prolog.load("prolog.pl");
        Logic gorgias = new Gorgias();
        gorgias.load("gorgias.pl");

        System.out.println("Prolog");
        testLogic(prolog);

        System.out.println();

        System.out.println("Gorgias");
        testLogic(gorgias);
    }

}
```

Η έξοδος του παραπάνω προγράμματος είναι

```
Prolog
Is alice mother of charlie?
true
Why?
[]

Gorgias
Is alice mother of charlie?
true
Why?
[f1]
```

Φαίνεται ότι μπορούμε να χρησιμοποιήσουμε ακριβώς με τον ίδιο τρόπο ένα αντικείμενο Γοργία και ένα αντικείμενο Prolog (καθώς υλοποιούν την ίδια διασύνδεση). Επίσης στο αρχείο κανόνων υπάρχουν μόνο οι κανόνες του κόσμου. Τα «δεδομένα»

του κόσμου, τα οποία μπορεί να είναι δυναμικά και να αλλάζουν κάθε φορά που εκτελούμε το πρόγραμμα (ή και να αλλάζουν κατά την εκτέλεση του) τα εισάγουμε αργότερα χρησιμοποιώντας τις κατάλληλες συναρτήσεις που έχουμε ορίσει.

Ο Γοργίας όμως μπορεί να μας δώσει το αποδεκτό επιχείρημα που κατασκεύασε κατά την απόδειξη. Έτσι μπορεί να παρέχει στον χρήστη τεκμηρίωση που ανάλογα με την εφαρμογή να είναι απαραίτητη (πχ αναφορές σε συγκεκριμένους νόμους αν ήταν νομική εφαρμογή) και συγχρόνως με αυτόν τον τρόπο μπορεί ο χρήστης να επιβεβαιώσει ότι οι κανόνες του κόσμου είναι σωστή και αν δεν είναι να τους διορθώσει. Επίσης ο Γοργίας σε σχέση με την απλή Prolog έχει προτεραιότητες ανάμεσα στους κανόνες. Αυτό κάνει πιο απλή την σύνταξη κανόνων και στο μέλλον πιο εύκολη την επέκταση τους με νέους χωρίς να δημιουργηθούν προβλήματα στους παλιούς.

Στο παράρτημα υπάρχουν παραδείγματα χρήσης των κλάσεων Logic, Prolog και Gorgias.

4.4 Προτερήματα αυτής της μεθοδολογίας

Χρησιμοποιώντας τις κλάσεις και τις μεθόδους που περιγράψαμε στην προηγούμενη ενότητα μπορούμε να κάνουμε πιο εύκολη την ανάπτυξη εφαρμογών που εκμεταλλεύονται τις δυνατότητες του λογικού προγραμματισμού της Prolog και τις δυνατότητες του Γοργία για λογικό προγραμματισμό χωρίς άρνηση σαν αποτυχία. Για παράδειγμα μπορούμε να διαχωρίσουμε τον κώδικα για την ερμηνεία των αποτελεσμάτων από τον κώδικα της υπόλοιπης εφαρμογής και κατά συνέπεια είναι δυνατόν να αναπτυχθεί παράλληλα από διαφορετικά ανθρώπους που είναι ειδικοί στο κάθε πεδίο. Αυτό μπορεί να μειώσει τον χρόνο ανάπτυξης της εφαρμογής και να μειώσει την πιθανότητα σφαλμάτων.

Αυτός ο διαχωρισμός μπορεί να κάνει πιο εύκολη την συντήρηση και την επέκταση της εφαρμογής στο μέλλον καθώς μπορούμε να αλλάζουμε εσωτερικά την υλοποίηση στα διαφορετικά τμήματα της εφαρμογής εφόσον διατηρούμε την ίδια διεπαφή. Επίσης είναι πιο εύκολο να επεκτείνουμε την ερμηνεία αν αυτή αποδειχθεί ελλιπής και να ελέγξουμε ότι είναι σωστή.

Τέλος, χάρη σε αυτόν τον διαχωρισμό, μπορούμε να χρησιμοποιήσουμε διαφορετικές γλώσσες προγραμματισμού για διαφορετικά τμήματα της εφαρμογής επιλέγοντας την κατάλληλη για κάθε πεδίο. Για παράδειγμα αν θέλουμε να κατασκευάσουμε μία εφαρμογή για ερμηνεία στατιστικών αποτελεσμάτων μπορούμε να χρησιμοποιήσουμε την γλώσσα προγραμματισμού R για την παραγωγή των αποτελεσμάτων (επειδή έχει πλούσια βιβλιοθήκη με συναρτήσεις για στατιστική) και να χρησιμοποιήσουμε τον Γοργία για την ερμηνεία αυτών των αποτελεσμάτων. Σε αυτήν την περίπτωση η Java

λειτουργεί σαν «γέφυρα» ανάμεσα στα διαφορετικά περιβάλλοντα και συγχρόνως αποκτάμε πρόσβαση στις δυνατότητες του Γοργία, όπως προτεραιότητες ανάμεσα στους κανόνες που κάνουν πιο φυσική την σύνταξη των κανόνων ή την δυνατότητα να «εξηγεί» τον λόγο που έδωσε κάποια απάντηση παρουσιάζοντας το αποδεκτό επιχείρημα που κατασκεύασε. Το τελευταίο μπορεί να είναι χρήσιμο για την ενημέρωση και «εκπαίδευση» του χρήστη καθώς στο αποδεκτό επιχείρημα θα υπάρχουν αναφορές σε συγκεκριμένους κανόνες που αναφέρονται στην βιβλιογραφία. Για παράδειγμα αν πρόκειται για εφαρμογή στατιστικού ελέγχου υποθέσεων το πρόγραμμα θα εξηγεί για πιο λόγω χρησιμοποίησε κάποιον συγκεκριμένο έλεγχο ανάλογα με τον τύπο των δεδομένων ή κάποιες ιδιότητες του δείγματος.

Κεφάλαιο 5

Αρχιτεκτονική εφαρμογής ελέγχου υποθέσεων με χρήση Γοργία

5.1 Χρήση Γοργία για έλεγχο ανεξαρτησίας

Στην προηγούμενη ενότητα περιγράψαμε την σύνδεση του Γοργία (και της Prolog) με την γλώσσα προγραμματισμού Java μέσω του interface `Logic` που ορίσαμε. Θα χρησιμοποιήσουμε τα παραπάνω για να περιγράψουμε μία μεθοδολογία για ανάπτυξη εφαρμογών ανάλυσης δεδομένων και στατιστική ανάλυσης που θα κάνουν χρήση της λογικής της επιχειρηματολογίας.

Για τον υπολογισμό των στατιστικών μεγεθών θα χρησιμοποιήσουμε την γλώσσα R[14, 6]. Η εφαρμογή θα επιλέγει τον σωστό στατιστικό έλεγχο, ανάλογα με τα χαρακτηριστικά του δείγματος και εφόσον ικανοποιούνται οι προϋποθέσεις των θεωρημάτων, και θα χρησιμοποιεί τον Γοργία για να ερμηνεύσει αυτά τα αποτελέσματα και να κάνει έλεγχο υποθέσεων. Αυτή η αρχιτεκτονική επιλέγεται γιατί:

- Η R παρέχει πολλές έτοιμες στατιστικές συναρτήσεις. Έτσι δεν χρειάζεται να τις υλοποιήσουμε και εκμεταλλευόμαστε το γεγονός ότι αυτές είναι δοκιμασμένες λόγω της μεγάλης διάδοσης και χρήσης της R στο πεδίο της στατιστικής ανάλυσης[6].
- Η λογική της ερμηνείας των αποτελεσμάτων βρίσκεται συγκεντρωμένη σε ένα αρχείο κανόνων του Γοργία και αντιστοιχεί πλήρως σε ό,τι αναφέρεται στην βιβλιογραφία της στατιστικής. Έτσι είμαστε σίγουροι για την ορθότητα της ερμηνείας και επίσης εξασφαλίζουμε καλύτερη ανάπτυξη της εφαρμογής καθώς ο μαθηματικός που γνωρίζει στατιστική μπορεί να ασχοληθεί μόνο με την σύνταξη αυτών των κανόνων και όχι με την υπόλοιπη εφαρμογή. Επίσης, αν στο

μέλλον αυτοί οι κανόνες πρέπει να διορθωθούν ή να επεκταθούν δεν χρειάζεται να αλλάξει το υπόλοιπο πρόγραμμα, αρκεί η διασύνδεση στα διαφορετικά τμήματα του να παραμείνει σταθερή.

- Η χρήση του Γοργία μας δίνει περισσότερες δυνατότητες σε σχέση με την Prolog. Για παράδειγμα μας επιτρέπει να μοντελοποιήσουμε πιο εύκολα και με μεγαλύτερη ακρίβεια τον έλεγχο υποθέσεων, όπως ακριβώς περιγράφεται στην βιβλιογραφία, κάνοντας χρήση των κανόνων προτεραιότητας όπως τους έχουμε περιγράψει σε προηγούμενες ενότητες.
- Επίσης μπορούμε να εκμεταλλευτούμε την δυνατότητα του Γοργία να μας παρουσιάζει το αποδεκτό επιχείρημα που κατασκευάστηκε κατά την απόδειξη και έτσι να «δικαιολογεί» τις απαντήσεις του. Για παράδειγμα αν ικανοποιούνται οι προϋποθέσεις του κεντρικού οριακού θεωρήματος θα χρησιμοποιείται ο έλεγχος X^2 ενώ σε αντίθετη περίπτωση θα χρησιμοποιεί κάποια άλλη μέθοδο και θα ενημερώνει τον χρήστη ανάλογα.
- Έτσι θα υπάρχει δυνατότητα ελέγχου από τον χρήστη ότι το αποτέλεσμα είναι ορθό γιατί η λογική του προγράμματος δεν θα είναι κρυμμένη μέσα στο πρόγραμμα αλλά θα φαίνεται κατά την χρήση του.
- Ο χρήστης (που μπορεί να μην είναι μαθηματικός) θα «μαθαίνει» στατιστική και θα χρησιμοποιεί τα εργαλεία της στατιστικής σωστά (πχ αν χρησιμοποιούσε μόνο την R μπορεί να έκανε έναν έλεγχο X^2 ενώ δεν ικανοποιούνται οι προϋποθέσεις του κεντρικού οριακού θεωρήματος).

5.2 Έλεγχος ανεξαρτησίας X^2, ακριβής έλεγχος Fisher

Για να παρουσιάσουμε αυτήν την μεθοδολογία θα κάνουμε στατιστικό έλεγχο για να ελέγξουμε αν υπάρχει εξάρτηση (συνάφεια) μεταξύ δύο χαρακτηριστικών, των A και B. Ένα τρόπος για να το κάνουμε αυτό είναι με τον **έλεγχο ανεξαρτησία** X^2.

Για την εφαρμογή αυτού του ελέγχου αναπαριστούμε συνήθως τα δεδομένα από ένα δείγμα μεγέθους n, υπό την μορφή ενός $2\chi 2$ **πίνακα συνάφεια (contingency table)**. Ο πίνακας αυτός είναι ένας πίνακας συχνοτήτων όπου στις γραμμές έχουμε τους δύο υποπληθυσμούς (κατηγορίες του χαρακτηριστικού B) και στις στήλες τις επιτυχίες και αποτυχίες (κατηγορίες του χαρακτηριστικού A)[6].

	Επιτυχία	**Αποτυχία**	Σύνολο
1ος υποπληθυσμός	n_{11}	n_{12}	n_1
2ος υποπληθυσμός	n_{21}	n_{22}	n_2
Συνολο	n_1	n_2	n

Πίνακας 5.1: 2×2 πίνακας συνάφειας

Έστω p_{ij} η (από κοινού) πιθανότητα να καταταγεί μία μονάδα του πληθυσμού στο κελί ij, όπου $i, j = 1, 2$. Έστω επίσης p_i η περιθώρια πιθανότητα να καταταγεί μία μονάδα του πληθυσμού στην σειρά i και p_j η περιθώρια πιθανότητα να καταταγεί μία μονάδα του πληθυσμού στην στήλη j.

Η υπόθεση της ανεξαρτησίας εκφράζεται από την μηδενική υπόθεση H_0 έναντι της εναλλακτικής H_1:

$$H_0 : p_{ij} = p_{i.}p_{.j}$$
$$H_1 : p_{ij} \neq p_{i.}p_{.j}$$

Οι «λογικοί» εκτιμητές των περιθωρίων πιθανοτήτων είναι:

$$\hat{p_{i.}} = \frac{n_{i.}}{n}$$
$$\hat{p_{.j}} = \frac{n_{.j}}{n}$$

Επομένως οι εκτιμητές των αναμενόμενων, κάτω από την μηδενική υπόθεση, συχνοτήτων είναι:

$$E_{ij} = n\hat{p_{i.}}\hat{p_{.j}} = \frac{n_{i.}n_{.j}}{n}$$

Για να ισχύουν τα παραπάνω χρειάζεται να ισχύει το κεντρικό οριακό θεώρημα (Κ.Ο.Θ.). Στην πράξη συνηθίζεται να έχουμε την προϋπόθεση: όλες οι αναμενόμενες συχνότητες να είναι ≥ 5.

Η γλώσσα προγραμματισμού R προμηθεύει την συνάρτηση `chisq.test()` που εκτελεί τον έλεγχο X^2 σε πίνακες συνάφειας με επίπεδο σημαντικότητας 5%. Για να έχει όμως νόημα το αποτέλεσμα της `chisq.test()` πρέπει, όπως ακριβώς περιγράφει η θεωρία, να ισχύει το Κ.Ο.Θ., δηλαδή οι αναμενόμενες συχνότητες που υπολογίζει η `chisq.test()` να είναι ≥ 5.

Όταν δεν ισχύουν οι προϋποθέσεις του Κ.Ο.Θ. μπορούμε να χρησιμοποιήσουμε τον ακριβή έλεγχο Fisher[6] για έλεγχο ανεξαρτησίας. Η R προμηθεύει την συνάρτηση `fisher.test()` η οποία ως βασικό όρισμα παίρνει τον πίνακα συνάφειας δεδομένων και εκτελείται με επίπεδο σημαντικότητας 5%.

Άρα όταν καλούμαστε πρακτικά να ελέγξουμε με την βοήθεια της R αν δύο χαρακτηριστικά του πληθυσμού είναι ανεξάρτητα και γνωρίζουμε τον πίνακα συνάφειας μπορούμε να χρησιμοποιήσουμε πρώτα την συνάρτηση `chisq.test()` και αν δούμε ότι δεν ικανοποιείται το Κ.Ο.Θ. (η R μας προειδοποιεί για αυτό) αγνοούμε το αποτέλεσμα της και κάνουμε έλεγχο ανεξαρτησίας με την `fisher.test()`.

Η παραπάνω διαδικασία μπορεί να μας απαντήσει σε ερωτήματα όπως αν το επίπεδο άσκησης των ερωτηθέντων σε μία στατιστική έρευνα είναι ανεξάρτητο του αν είναι καπνιστές, αν το ποσοστό ελαττωματικών αντικειμένων είναι ανεξάρτητο της γραμμής παραγωγής, κτλ. Στην επόμενη ενότητα θα χρησιμοποιήσουμε τον Γοργία και την λογική της επιχειρηματολογίας για να κατασκευάσουμε μία εφαρμογή που θα κάνει αυτόν τον έλεγχο και θα απαντάει στον χρήστη αν η μηδενική υπόθεση απορρίπτεται ή δεν απορρίπτεται. Συγχρόνως αυτήν την απάντηση θα την τεκμηριώνει στον χρήστη παραθέτοντας τις τιμές που υπολογίστηκαν από την R και τους λογικούς κανόνες του Γοργία (που αντιστοιχούν σε θεωρήματα της στατιστικής).

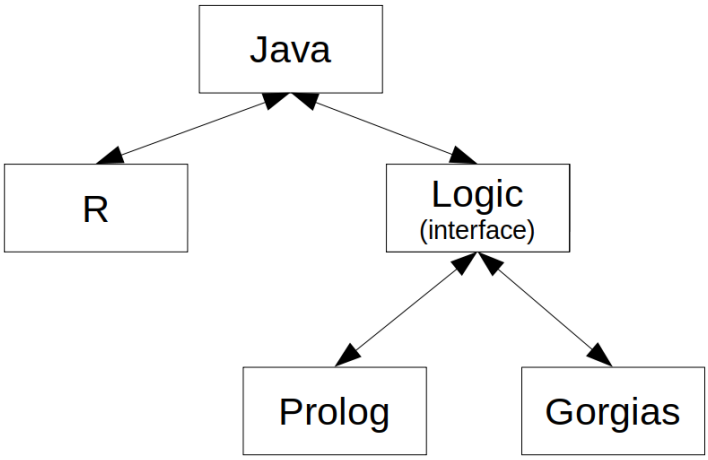
Αξίζει να σημειώσουμε ότι με μία τέτοια προσέγγιση ο χρήστης της εφαρμογής δεν είναι ανάγκη να γνωρίζει όλες τις λεπτομέρειες της μαθηματικής στατιστικής. Το πρόγραμμα θα λειτουργεί σαν «μαθηματικός» και θα δίνει την σωστή απάντηση. Αυτό μπορεί να βοηθήσει ώστε να αποφευχθούν σφάλματα που προκύπτουν από λάθος χρήση εργαλείων της στατιστικής από μη ειδικούς, όπως για παράδειγμα στην ιατρική.

Στο μέλλον μία τέτοια εφαρμογή μπορεί να επεκταθεί φυσικά χωρίς να διαταράσσεται η μέχρι τώρα λειτουργία της. Αρκεί να προστεθούν οι νέοι κανόνες ερμηνείας των στατιστικών αποτελεσμάτων και οι ερωτήσεις προς τον χρήστη για τον τύπο των δεδομένων και το είδος του προβλήματος.

5.3 Υλοποίηση διασύνδεσης Java με R

Για την επικοινωνία της Java με την R θα χρησιμοποιήσουμε το JRI[15] που προμηθεύει η βιβλιοθήκη διασύνδεσης της Java με την R, rJava. Αυτό μας επιτρέπει να καλέσουμε συναρτήσεις της R και να αποθηκεύσουμε το αποτέλεσμα σαν μία μεταβλητή στην Java.

Εφόσον, όπως είπαμε στην προηγούμενη ενότητα, θα επικεντρωθούμε στον έλεγχο ανεξαρτησίας μεταξύ 2 χαρακτηριστικών θα υλοποιήσουμε, με την βοήθεια του JRI, την κλάσης `Statistics.java` που θα περιέχεις τις σχετικές συναρτήσεις. Αυτές θα καλούν την R, θα φορτώνουν έναν πίνακα συνάφειας και στην συνέχεια θα καλούν τις συναρτήσεις `chisq.test()` και `fisher.test()` ώστε τα αποτελέσματα τους να περάσουν αργότερα στον Γοργία, μέσω του interface `Logic`, όπου θα γίνει η ερμηνεία τους όπως φαίνεται στο παρακάτω σχήμα.



Σχήμα 5.1: Δομή εφαρμογής

Στο παράρτημα φαίνεται η υλοποίηση της κλάσης `Statistics.java` που συγκεντρώνει όλη την λειτουργία της εφαρμογής που έχει να κάνει με την διασύνδεση με την R. Οι συναρτήσεις που περιέχει είναι:

```
void load(String rfile);
void closeR();
void chisq(String variable);
double getChisqStat(String variable);
int getChisqDf(String variable);
double getChisqPValue(String variable);
double getMinExpected(String variable);
void fisher(String variable);
double getFisherPValue(String variable);
```

Με την συνάρτηση `load()` φορτώνουμε μία δέσμη εντολών R από ένα αρχείο. Για παράδειγμα στο πρόβλημα ελέγχου σε επίπεδο 5% αν το ποσοστό ελαττωματικών προϊόντων για 2 παραγωγικές διαδικασίες είναι το ίδιο ($H_0 : p_1 = p_2$) έναντι της εναλλακτικής ότι δεν είναι το ίδιο ($H_1 : p_1 \neq p2$) αντιστοιχεί ο παρακάτω πίνακας συνάφειας:

	Αριθμός ελαττωματικών	**Αριθμός μη-ελαττωματικών**
1η παραγ. διαδικ.	12	288
2η παραγ. διαδικ.	20	380

Πίνακας 5.2: 2×2 πίνακας συνάφειας για το παράδειγμα

Για να φορτώσουμε αυτόν τον πίνακα συνάφειας στην εφαρμογή δημιουργούμε το αρχείο `data1.txt` με την παρακάτω εντολή της R:

```
data1<-rbind(c(12,288),c(20,380))
```

Στο κύριο πρόγραμμα δημιουργούμε ένα αντικείμενο της κλάσης `Statistics` και φορτώνουμε αυτό το αρχείο με την συνάρτηση `load()`:

```
Statistics s = new Statistics();
s.load("data1.txt");
```

Το αρχείο με τις εντολές της R μπορεί να είναι πολύ πιο πολύπλοκο, ανάλογα με το πρόβλημα, και εφόσον διαλέγουμε διαφορετική μεταβλητή για κάθε πίνακα συνάφειας μπορούμε να τους έχουμε όλους διαθέσιμους στης εφαρμογή χωρίς να υπάρχουν συγκρούσεις.

Στην συνέχεια με τις συναρτήσεις `chisq()` και `fisher()`, που ορίζει η κλάση, εφαρμόζουμε τον έλεγχο ανεξαρτησίας X^2 ή ακριβή έλεγχο Fisher στην μεταβλητή που

περιέχει τον πίνακα συνάφειας. Τα αποτελέσματα αυτών των ελέγχων αποθηκεύονται σε διαφορετική μεταβλητή για κάθε πίνακα συνάφειας και μπορούμε να τα ανασύρουμε με τις υπόλοιπες συναρτήσεις. Αυτά τα αποτελέσματα θα περάσουν στον Γοργία με την συνάρτηση `claim()` του interface `Logic` για να γίνει η ερμηνεία και ο έλεγχος υποθέσεων.

5.4 Έλεγχος υποθέσεων με Γοργία

Στην προηγούμενη ενότητα περιγράψαμε την κλάση `Statistics` που είναι υπεύθυνη για τον χειρισμό της R και την συλλογή των στατιστικών αποτελεσμάτων. Με βάση όσα είπαμε στην εισαγωγή η ερμηνεία αυτών των αποτελεσμάτων θα γίνει στον Γοργία χρησιμοποιώντας τις συναρτήσεις του interface `Logic`. Έτσι θα εκμεταλλευτούμε την λογική της επιχειρηματολογίας που υλοποιεί ο Γοργίας που θα μας αφήσει να εκφράσουμε με απολυτά φυσικό τρόπο την λογική του ελέγχου υποθέσεων, όπως την κάνει ένας μαθηματικός. Επίσης, μέσω της παραγωγής του αποδεκτού επιχειρήματος που πραγματοποιεί ο Γοργίας, θα μπορούμε να ενημερώνουμε τον χρήστη ποιοι κανόνες και δεδομένα οδήγησαν στο να δοθεί αυτή η απάντηση.

Ο τρόπος που θα κατασκευάσουμε το αρχείο κανόνων του Γοργία πρέπει να είναι συνεπής με τον τον τρόπο που γίνεται ο έλεγχος υποθέσεων στην κλασσική στατιστική.[6] Η μηδενική και η εναλλακτική υπόθεση δεν είναι απλά δύο υποθέσεις που θέλουμε να ελέγξουμε με βάση τα δεδομένα και να αποφασίσουμε ποια από τις δύο είναι ορθή. Αυτός είναι και ο λόγος που δεν μπορούμε να της εναλλάξουμε και αυτό πρέπει να αποτυπώνεται στο αρχείο κανόνων του Γοργία και στον τρόπο που γίνεται η απόδειξη.

Ως μηδενική απόδειξη θέτουμε αυτή για την οποία αμφιβάλουμε (πχ στο πρόβλημα μας αν δύο χαρακτηριστικά του δείγματος είναι ανεξάρτητα) και εξετάζουμε αν ένα τυχαίο δείγμα από τον πληθυσμό δίνει αρκετές αποδείξης υπέρ της απόρριψης της, ενάντια της εναλλακτικής. Αντίθετα η εναλλακτική υπόθεση (πχ υπάρχει συνάφεια στα χαρακτηριστικά του δείγματος) είναι η υπόθεση που θέλουμε να αποδείξουμε ότι είναι η αληθινή[6].

Επίσης, καθώς οι ενδείξεις για να απορρίψουμε ή να μην απορρίψουμε την H_0 προκύπτουν από το τυχαίο δείγμα που χρησιμοποιήσαμε και άρα έχει ληφθεί κάτω από συνθήκες αβεβαιότητας, δεν μπορούμε να είμαστε 100% σίγουροι για την ορθότητα της H_0 (ή της H_1). Οπότε δεν αποδεχόμαστε τη μηδενική υπόθεση, απλά δεν έχουμε σοβαρές ενδείξεις για να την απορρίψουμε. Δηλαδή αν με βάση τα δεδομένα απορρίψουμε τη μηδενική υπόθεση μπορούμε να συμπεράνουμε ότι η εναλλακτική πιθανώς να είναι αληθής, ενώ αν αποτύχουμε να την απορρίψουμε τότε αυτό δεν σημαίνει ότι

είναι οπωσδήποτε αληθής, απλά δεν υπάρχουν σοβαρές ενδείξεις που να συνηγορούν ότι αυτή είναι ψευδής.

Το παραπάνω το απεικονίζουμε στο αρχείο κανόνων του Γοργία μη απορρίπτοντας αρχικά την υπόθεση H_0 (τα δεδομένα είναι ανεξάρτητα) και την απορρίπτουμε όταν υπάρχουν στοιχεία που συνηγορούν για την απόρριψη της. Με κατάλληλους κανόνες προτεραιότητας κάνοντας χρήση του κατηγορήματος `prefer/2` του Γοργία λύνουμε την σύγκρουση μεταξύ αυτών των δύο κανόνων. Για παράδειγμα:

```
% H0 is not rejected
rule(h0_is_not_rejected(X),
     neg(rejecth0(X)),
     []).

% H0 is rejected
rule(chisq_rejects_h0(X),
     rejecth0(X),
     [chisq(X),
      chisq_valid(X),
      chisq_pvalue(X,Pvalue),
      significance(Significance),
      Pvalue < Significance]).

% prefer rule
rule(prefer_chisq_rejects_h0(X),
     prefer(chisq_rejects_h0(X),
            h0_is_not_rejected(X)),[]).
```

Η παραπάνω λογική των ελέγχων στατιστικής υπόθεσης είναι ανάλογη με την λογική των δικαστικών λειτουργιών. Στην απονομή δικαιοσύνης το δικαστήριο ξεκινάει από την υπόθεση ότι ο κατηγορούμενος είναι αθώος (αληθής η $_0$). Η εναλλακτική υπόθεση, ότι δηλαδή είναι ένοχος, πρέπει να το αποδείξει η κατηγορούσα αρχή προσκομίζοντας στο δικαστήριο επαρκή και έγκυρα στοιχεία που σχηματίζουν αρκετές ενδείξεις εναντίον του κατηγορημένου (η H_0 δεν είναι αληθής, υποστήριξη της H_1). Τότε μόνο ο δικαστής θα τον καταδικάσει. Για την αθωότητα του όμως, όπως και στην στατιστική, ο δικαστής δεν μπορεί ποτέ να είναι 100% σίγουρος. Αυτή η επιχειρηματολογία που χρησιμοποιείται στην στατιστική και στην νομική μπορεί να περιγραφεί φυσιολογικά από τον Γοργία και, χάρη στην αποδεικτική διαδικασία που ακολουθεί με την κατασκευή του αποδεκτού επιχειρήματος, το τελικό πόρισμα μαζί με όλους τους κανόνες και δεδομένα που το υποστηρίζουν μπορεί να παρουσιαστεί στον χρήστη της εφαρμογής. Αυτό είναι ένα μεγάλο πλεονέκτημα αυτής της μεθόδου έναντι

διαφορετικών προσεγγίσεων.

Με βάση τα παραπάνω κατασκευάζουμε το αρχείο κανόνων του Γοργία για τον έλεγχο ανεξαρτησίας με χρήση X^2 και Fisher. Η αντιστοιχία των κανόνων της στατιστικής με τους κανόνες του Γοργία φαίνεται στον πίνακα 5.4. Σημειώνουμε ότι αυτό το αρχείο κανόνων του Γοργία περιέχει μόνο κανόνες όπως αυτοί περιγράφονται στην στατιστική και που είναι ανεξάρτητοι από το στατιστικό δείγμα, τα δεδομένα (πχ τιμή P-Value, ελάχιστη αναμενόμενη συχνότητα, κτλ) που εξαρτώνται από το στατιστικό δείγμα (και θα υπολογιστούν από την R) θα περάσουν στον Γοργία μέσω της συνάρτησης `claim()` της διασύνδεση `Logic`.

Καταφέραμε με αυτόν τον τρόπο να χρησιμοποιήσουμε τον Γοργία σαν βάση γνώσης (agent) που συγκεντρώνει όλη την λογική για τον έλεγχο συνάφειας ανάμεσα σε δύο χαρακτηριστικά A και B ενός πληθυσμού. Μπορεί μελλοντικά αυτή η βάση γνώσης να επεκταθεί ώστε με πρόσθετα ερωτήματα (πχ είδος στατιστικών δεδομένων, χαρακτηριστικά δείγματος, κτλ) προς τον χρήστη να αρχικοποιεί κατάλληλα δεδομένα μέσω των απαντήσεων που θα καθορίσουν μελλοντικές ερωτήσεις και τελικά το σύνολο και την σειρά με την οποία θα εκτελεστούν οι στατιστικές συναρτήσεις. Επίσης η πολιτική που ορίζεται από τους κανόνες του Γοργία αποφεύγει κλήσεις συναρτήσεων που δεν θα χρειαστούν. Για παράδειγμα ο έλεγχος Fisher εκτελείται μόνο όταν ο έλεγχος X^2 δεν είναι έγκυρος (δεν ισχύει το Κ.Ο.Θ.). Τέλος χάρη στην δυνατότητα να θέτουμε προτιμήσεις μεταξύ των κανόνων αλλά και προτιμήσεις υψηλότερης τάξης, όπως περιγράψαμε στο κεφάλαιο του Γοργία) η επέκταση των κανόνων και η εισαγωγή νέων κανόνων γίνεται εύκολα χωρίς να χρειάζεται να αλλάξουμε τους παλιούς και μπορούμε να χρησιμοποιήσουμε εργαλεία όπως ο Gorgias-B[13] και την μεθοδολογία ανάπτυξης SoDA (Software Development for Argumentation)[16].

Όλος ο κώδικας της εφαρμογής που ενώνει την κλάση `Statistics` με το interface `Logic` και τον Γοργία βρίσκεται στο αρχείο `stats.java` και υπάρχει στο παράρτημα.

Κανόνες Γοργία	**Στατιστική[6]**
`rule(h0_is_not_rejected(X),` `  neg(rejecth0(X)),[]).`	Η H_0 δεν απορρίπτεται αρχικά. Θα απορριφθεί ή δεν θα απορριφθεί με βάση τι παρατηρείται στο τυχαίο δείγμα. (σελ 204)

`rule(chisq_is_valid(X),`
`  chisq_valid(X),[chisq(X)]).`
`rule(chisq_is_not_valid(X),`
`  neg(chisq_valid(X),`
`  [chisq(X),`
`  chisq_minexpected(X,MinExpected),`
`  MinExpected < 5]).`
`rule(prefer_chisq_is_not_valid(X),`
`  prefer(chisq_is_not_valid(X),`
`  chisq_is_valid(X)),[]).`
	Για να ισχύει ο έλεγχος X^2 πρέπει να ισχύει το Κ.Ο.Θ. Πρακτικά πρέπει όλες οι αναμενόμενες συχνότητες να είναι ≥ 5. (σελ 261)

`rule(chisq_rejects_h0(X),`
`  rejecth0(X),`
`  [chisq(X),chisq_valid(X),`
`  chisq_pvalue(X,Pvalue),`
`  significance(Significance),`
`  Pvalue < Significance]).`
`rule(prefer_chisq_rejects_h0(X),`
`  prefer(chisq_rejects_h0(X),`
`  h0_is_not_rejected(X)),[]).`
	Ο έλεγχος X^2 τρέχει με $\gamma = 95\%$ συντελεστή εμπιστοσύνης ή $\alpha = 5\%$ επίπεδο σημαντικότητας.
	Αν P-Value $< \alpha$ απορρίπτουμε την H_0. (σελ 211)

`rule(fisher_rejects_h0(X),`
`  rejecth0(X),`
`  [fisher(X),`
`  fisher_pvalue(X,Pvalue),`
`  significance(Significance),`
`  Pvalue < Significance]).`
`rule(prefer_fisher_rejects_h0(X),`
`  prefer(fisher_rejects_h0(X),`
`  h0_is_not_rejected(X)),[]).`
	Όταν δεν ισχύουν οι προϋποθέσεις του Κ.Ο.Θ. χρησιμοποιούμε τον έλεγχο Fischer (σελ 263)
	Ο έλεγχος Fischer στην R τρέχει με επίπεδο σημαντικότητας $\alpha = 5\%$.
	Αν P-value $< \alpha$ απορρίπτουμε την H_0 και δεχόμαστε την H_1. (σελ 211)

Πίνακας 5.3: Έλεγχος Υποθέσεων: Αντιστοιχία κανόνων Γοργία με Στατιστική

5.5 Παράδειγμα χρήσης

Για να δείξουμε την χρήση της εφαρμογής και την μορφή των αποτελεσμάτων της θα δοκιμάσουμε τρεις διαφορετικούς πίνακες συνάφειας για τα χαρακτηριστικά A και B από τα δείγματα 4 διαφορετικών πληθυσμών. Οι πίνακες συνάφειας είναι:

	Αριθμός ελαττωματικών	**Αριθμός μη-ελαττωματικών**
1η παραγ. διαδικ.	12	288
2η παραγ. διαδικ.	20	380

Πίνακας 5.4: Πίνακας Συνάφειας data1 [6, σελ264]

	Αριθμός ελαττωματικών	**Αριθμός μη-ελαττωματικών**
3η παραγ. διαδικ.	12	288
4η παραγ. διαδικ.	100	380

Πίνακας 5.5: Πίνακας Συνάφειας data2

	Μη Καπνιστής	**Περιστασιακός Καπνιστής**	**Καπνιστής**
Άντρας	28	8	22
Γυναίκα	26	2	14

Πίνακας 5.6: Πίνακας Συνάφειας data3 [6, σελ268]

	Αριθμός ελαττωματικών	**Αριθμός μη-ελαττωματικών**
5η παραγ. διαδικ.	1	10
6η παραγ. διαδικ.	14	15

Πίνακας 5.7: Πίνακας Συνάφειας data4

Αυτούς τους πίνακες συνάφειας τους φορτώνουμε στην εφαρμογή και τους αποθηκεύουμε σε διαφορετικές μεταβλητές με αρχεία που περιέχουν τις παρακάτω εντολές της R:

```
data1<-rbind(c(12,288),c(20,380))
data2<-rbind(c(12,288),c(100,380))
data3<-rbind(c(28,26),c(8,2),c(22,14))
data4<-rbind(c(1,10),c(14,15))
```

Όταν εκτελέσουμε την εφαρμογή αρχικά θα εφαρμοστεί η συνάρτηση `chisq.test()` της R επάνω στους πίνακες συνάφειας και αν με βάσει τους κανόνες του Γοργία ο έλεγχος X^2 δεν είναι έγκυρος (οι αναμενόμενες δεν είναι όλες ≥ 5) το πρόγραμμα θα

συνεχίσει εφαρμόζοντας την συνάρτηση `fisher.test()`. Η τελική απάντηση της εφαρμογής και ο έλεγχος υποθέσεων με την βοήθεια του Γοργία είναι:

```
Cannot reject null hypotheses, data1 are independent
Why? [[h0_is_not_rejected(data1)]]

Null hypotheses rejected, data2 are codependent.
Why? [[f1, f6, f5, chisq_is_valid(data2), f5, chisq_rejects_h0(data2)]]

Cannot reject null hypotheses, data3 are independent
Why? [[h0_is_not_rejected(data3)]]

Null hypotheses rejected, data4 are codependent.
Why? [[f1, f17, f16, fisher_rejects_h0(data4)]]

Gorgias state
-------------
consult(stats.pl).
rule(f1,significance(0.050000000000000044),[]).
rule(f2,chisq(data1),[]).
rule(f3,chisq_pvalue(data1,0.6570195719690067),[]).
rule(f4,chisq_minexpected(data1,13.714285714285714),[]).
rule(f5,chisq(data2),[]).
rule(f6,chisq_pvalue(data2,1.386470999319574E-10),[]).
rule(f7,chisq_minexpected(data2,43.07692307692308),[]).
rule(f9,chisq_pvalue(data3,0.22674842690343286),[]).
rule(f10,chisq_minexpected(data3,4.2),[]).
rule(f11,fisher(data3),[]).
rule(f12,fisher_pvalue(data3,0.24022012054762612),[]).
rule(f14,chisq_pvalue(data4,0.05485393990013243),[]).
rule(f15,chisq_minexpected(data4,4.125),[]).
rule(f16,fisher(data4),[]).
rule(f17,fisher_pvalue(data4,0.030221989999279542),[]).
```

Ερμηνεύοντας αυτά τα αποτελέσματα βλέπουμε ότι για τον πίνακα data1 ο έλεγχος X^2 ήταν έγκυρος αλλά δεν προέκυψαν επαρκή στοιχεία για να απορριφθεί η μηδενική υπόθεση H_0. Το αποδεκτό επιχείρημα που κατασκεύασε ο Γοργίας αποτελείται μόνο από την αρχική θέση ότι η μηδενική υπόθεση δεν απορρίπτεται εφόσον δεν υπάρχουν επαρκείς ενδείξεις (ανάλογα: ο κατηγορούμενος θεωρείται αθώος μέχρι να προσκομιστούν επαρκή στοιχεία). Για το data2 το αποτέλεσμα του έλεγχου X^2 έδωσε επαρκή

στοιχεία για την απόρριψη της μηδενικής υπόθεσης και αυτό φαίνεται στο αποδεκτό επιχείρημα. Στο data3 δεν προέκυψαν αρκετά στοιχεία για την απόρριψη της H_0. Αξίζει να παρατηρήσουμε ότι εδώ χρειάστηκε να εφαρμοστεί και ο έλεγχος Fisher γιατί οι αναμενόμενες συχνότητες που προέκυψαν από το `chisq.test()` κατέστησαν τον X^2 μη έγκυρο. Τέλος στο data4 επειδή το δείγμα ήταν μικρό το πρόγραμμα χρησιμοποίησε τον ακριβή έλεγχο Fisher και απέρριψε μέσω αυτού την μηδενική υπόθεση όπως φαίνεται στο αποδεκτό επιχείρημα της απάντησης.

Κεφάλαιο 6

Συμπεράσματα

Στα προηγούμενα κεφάλαια μελετήσαμε τον Λογικό Προγραμματισμό χωρίς Άρνηση σαν Αποτυχία και την υλοποίηση του, σε συνδυασμό με την λογική απαγωγή, στο σύστημα Γοργίας και είδαμε τα προτερήματα που προσφέρει αυτό το είδος λογικού προγραμματισμού. Συγκεκριμένα με την διαδικασία απόδειξης προτάσεων του Γοργία που βασίζεται στην κατασκευή του αποδεκτού επιχειρήματος μπορούμε να υλοποιήσουμε γνωσιακούς βοηθούς που λειτουργούν σε δυναμικά και εξελισσόμενα περιβάλλοντα παρά τις ελλιπείς πληροφορίες και τεκμηριώνουν τις απαντήσεις τους με αναφορές σε συγκεκριμένους κανόνες και δεδομένα.

Για να εκμεταλλευτούμε τις δυνατότητες του Γοργία σε περισσότερα προβλήματα και να τον τροφοδοτούμε με πληροφορίες από άλλα προγράμματα και βιβλιοθήκες που δεν είναι ανεπτυγμένα σε Prolog κατασκευάσαμε μία διασύνδεση της Prolog και του Γοργία με την Java μέσω ενός κοινού interface. Με αυτό το σχέδιο η Java λειτουργεί σαν γέφυρα επικοινωνίας του Γοργία με τον χρήστη και άλλες βιβλιοθήκες. Επίσης με αυτόν τον τρόπο οι κανόνες του προβλήματος παραμένουν σταθεροί ενώ τα δεδομένα, που μπορεί να είναι διαφορετικά κάθε φορά ή να αλλάζουν κατά την διάρκεια εκτέλεσης, περνάνε δυναμικά κατά την λειτουργία του συστήματος.

Με την μεθοδολογία αυτή αναπτύξαμε μία εφαρμογή που χρησιμοποιεί την γλώσσα R για να κάνει έλεγχο ανεξαρτησίας μεταξύ 2 χαρακτηριστικών ενός πληθυσμού μέσω του ελέγχου X^2 και του ακριβή ελέγχου Fisher. Η ερμηνεία των αποτελεσμάτων αυτών των ελέγχων γίνεται στον Γοργία ο οποίος επίσης διαλέγει και τον σωστό έλεγχο ανάλογα με τα χαρακτηριστικά του δείγματος. Ο χρήστης λαμβάνει σαν απάντηση αν απορρίφθηκε η μηδενική υπόθεση ή αν δεν προέκυψαν ενδείξεις για να απορριφθεί μαζί με το αποδεκτό επιχείρημα που κατασκευάστηκε που αναφέρει τους δεδομένα και τους κανόνες που οδήγησαν σε αυτήν την απάντηση. Το σύστημα αυτό βρίσκεται σε πρώιμο στάδιο και μπορεί να επεκταθεί με την προσθήκη περισσότερων στατιστι-

κών ελέγχων μαζί με ερωτήσεις προς τον χρήστη για το είδος της εργασίας και τα χαρακτηριστικά του δείγματος. Η επέκταση των κανόνων του Γοργία γίνεται εύκολα ώστε οι νέοι κανόνες να συμβαδίζουν με την βιβλιογραφία της στατιστικής και με την προσθήκη της κατάλληλης πολιτικής προτίμησης οι ήδη υπάρχοντες κανόνες δεν θα χρειαστεί να αλλάξουν.

Η μεθοδολογία που παρουσιάσαμε μπορεί να χρησιμοποιηθεί και σε άλλα πεδία, όπως για παράδειγμα νομικές εφαρμογές, επιχειρηματικούς βοηθούς κτλ, ώστε ο χρήστης να μπορεί να παίρνει τεκμηριωμένες απαντήσεις στα ερωτήματα του με τρόπο φυσικό ακόμα και αν δεν έχει προμηθεύσει στην εφαρμογή όλη την πληροφορία, χάρη στην δυνατότητα του Γοργία για συλλογισμό με ελλιπή γνώση. Επίσης χάρη στην δυνατότητα του Γοργία για προσαρμόσιμες πολιτικές προτίμησης αυτοί οι γνωσιακοί βοηθοί που κατασκευάζουμε μπορούν να επεκταθούν εύκολα με νέους κανόνες χωρίς να διαταράσσεται η λειτουργία τους.

Παράρτημα Α΄

Prolog

Α΄.1 Εισαγωγή

Το σύστημα Γοργιας[12] που υλοποιεί τις ιδέες του Λογικού Προγραμματισμού χωρίς Άρνηση σαν Αποτυχία (LPwNF[2]) και τις συνδυάζει με την λογική απαγωγή είναι υλοποιημένο στην γλώσσα λογικού προγραμματισμού Prolog και συγκεκριμένα στην διάλεκτο SWI-Prolog[9]. Θα περιγράψουμε τα βασικά χαρακτηριστικά της Prolog για να καταλάβουμε καλύτερα τον τρόπο που λειτουργεί ο Γοργίας και τις νέες δυνατότητες που προσθέτει στον λογικό προγραμματισμό.

Η Prolog[17] είναι μία γλώσσα λογικού προγραμματισμού[7]. Ένα πρόγραμμα σε Prolog είναι μία συλλογή από γεγονότα και κανόνες συνεπαγωγής που χρησιμοποιούνται για να αποδειχθούν προτάσεις. Τα προγράμματα στην Prolog δεν «εκτελούνται» με τον τρόπο που εννοούμε σε διαδικαστικές γλώσσες προγραμματισμού, όπως είναι η Java ή η C, αλλά ο χρήστης υποβάλλει ερωτήματα και το γλωσσικό σύστημα της Prolog χρησιμοποιεί τα γεγονότα και τους κανόνες του προγράμματος για να δώσει απαντήσεις. Αυτή η διαδικασία είναι πολύ πιο ευέλικτη από ό,τι φαίνεται αρχικά και η Prolog (μαζί με τις διαλέκτους της) έχεις τις ίδιες δυνατότητες με τις υπόλοιπες γλώσσες προγραμματισμού. Η ιδιαιτερότητα όμως αυτή κάνει την Prolog ιδιαίτερα χρήσιμη σε προβλήματα που εκφράζονται με την γλώσσα της λογικής με αποτέλεσμα η Prolog (μαζί με την Lisp) να είναι ιδιαίτερα δημοφιλής στην περιοχή των εφαρμογών τεχνητής νοημοσύνης. Αξίζει επίσης να σημειώσουμε ότι η Prolog είναι η μοναδική ευρέως χρησιμοποιούμενη γλώσσα λογικού προγραμματισμού. Υπάρχουν φυσικά διάφορες διάλεκτοι με μικρές διαφορές και διαφορετικές δυνατότητες (όπως η SWI-Prolog[9]) αλλά συνήθως η έννοια του λογικού προγραμματισμού είναι ταυτόσημη με την Prolog.

Α΄.2 Συντακτικό Prolog

Συγκριτικά με άλλες γλώσσες προγραμματισμού η Prolog συντακτικά είναι πολύ απλή γλώσσα. Τα πάντα σε ένα πρόγραμμα Prolog, τόσο το ίδιο το πρόγραμμα όσο και τα δεδομένα που χειρίζεται, αποτελούνται από όρους (terms). Υπάρχουν τρία είδη όρων:

- οι σταθερές (constants),
- οι μεταβλητές (variables) και
- οι σύνθετοι όροι (compound terms).

Το απλούστερο είδος όρου είναι οι σταθερές και αποτελούνται από αριθμούς (ακέραιους και πραγματικούς αριθμούς) και άτομα. Κάθε όνομα που αρχίζει με ένα πεζό γράμμα και συνεχίζεται με μηδέν η περισσότερα ψηφία, γράμματα ή κάτω παύλες είναι ένα άτομο. Άτομα επίσης θεωρούνται οι ακολουθίες των περισσότερων μη αλφαριθμητικών (όπως *, +, -, ., κτλ) και υπάρχουν και μερικά ιδιαίτερα άτομα όπως ο κενός κατάλογος `[]` και ο όρος-στόχος cut `!` (ο οποίος πάντα επιτυγχάνει και σταματάει την οπισθοχώρηση επιτρέποντας να χειριστούμε καλύτερα τον διαδικαστικό χαρακτήρα της Prolog).

Οι μεταβλητές της Prolog είναι κάθε όνομα που αρχίζει με κεφαλαίο γράμμα ή κάτω παύλα. Οι μεταβλητές που αρχίζουν με κάτω παύλα, συμπεριλαμβανομένης της `_` που είναι γνωστή ως ανώνυμη μεταβλητή, έχουν ειδική χρήση.

Το τελευταίο είδος όρων της Prolog είναι οι σύνθετοι όροι που αποτελούνται από ένα άτομο που ακολουθείται από έναν κατάλογο όρων που χωρίζονται με κόμματα και βρίσκονται μέσα σε παρενθέσεις. Το άτομο που ξεκινά έναν σύνθετο όρο καλείται κατηγόρημα (predicate). Για παράδειγμα για τον σύνθετο όρο `parent(alice, bob)` έχουμε το κατηγόρημα `parent` που δέχεται δύο παραμέτρους και συχνά συμβολίζουμε `parent/2`. Με το συντακτικό που περιγράψαμε μπορούμε να ορίσουμε τα κατηγορήματα που χρειαζόμαστε για κάθε πρόβλημα και έχουμε στην διάθεση μας και τα κατηγορήματα που προσφέρει κάθε γλωσσικό σύστημα Prolog.

Συνοψίζοντας, το συντακτικό για τους όρους της Prolog περιγράφεται από τους παρακάτω κανόνες σε απλοποιημένη BN μορφή:

```
<term> ::= <constant> | <variable> | <compound-term>
<constant> ::= <integer> | <real number> | <atom>
<compound-term> ::= <atom> ( <termlist> )
<termlist> ::= <term> | <term>, <termlist>
```

Με τους όρους που περιγράψαμε μπορούμε να εκφράσουμε τα δεδομένα σε ένα πρόγραμμα Prolog. Το συντακτικό των κανόνων (rules) της Prolog είναι επίσης απλό και

αποτελείται από έναν όρο που ονομάζεται κεφαλή του κανόνα και ακολουθείται από το λεξιμόριο `:-` και στην συνέχεια από έναν κατάλογο όρων, τις συνθήκες του κανόνα, και στο τέλος μία τελεία. Τελικά γεγονότα και κανόνες συγκροτούν φράσεις (clauses) και έτσι κάθε πρόγραμμα Prolog είναι μία σειρά από φράσεις (κανόνες ή γεγονότα):

```
<clause> ::= <fact> | <rule>
<fact> ::= <term> .
<rule> ::= <term> :- <termlist> .
```

Με βάση αυτό το συντακτικό μπορούμε να εκφράσουμε τα περισσότερα προγράμματα Prolog. Για παράδειγμα μία βάση γνώσης που ορίζει ένα γενεαλογικό δέντρο θα μπορούσε να εκφραστεί με τον παρακάτω τρόπο:

```
mother(alice, charlie).
father(bob, charlie).
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(Z, Y), ancestor(X, Z).
```

Αυτήν την βάση γνώσης μπορούμε να την φορτώσουμε σε κάποιο σύστημα Prolog με το ενσωματωμένο κατηγόρημα `consult/1` και χρησιμοποιώντας τους γραμματικούς κανόνες που περιγράψαμε μπορούμε να υποβάλλουμε ερωτήματα μετά το προτρεπτικό σύμβολο (`?-`) που εμφανίζεται, πχ:

```
?- consult(relations).
true.

?- parent(alice, charlie).
true .

?- parent(X, charlie).
X = alice ;
X = bob.
```

Τα γλωσσικά συστήματα Prolog υποστηρίζει επίσης μία εύχρηστη γραφή για λίστες (γράφουμε τους όρους ανάμεσα στις αγκύλες και τους χωρίζουμε με , πχ `[ term1, term2 ]`) μαζί με πολλά ενσωματωμένα κατηγορήματα και τελεστές (τελεστές είναι τα κατηγορήματα που μπορούν να γράφονται και ανάμεσα στους όρους, πχ `=(X, 1))` και `X = 1`).

Α΄.3 Σημασιολογία Prolog

Όπως αναφέραμε στην εισαγωγή του κεφαλαίου η Prolog είναι γλώσσα λογικού προγραμματισμού που σημαίνει ότι ένα πρόγραμμα είναι μία βάση γνώσης που αποτελείται από γεγονότα και κανόνες συνεπαγωγής τους οποίους το γλωσσικό σύστημα της Prolog χρησιμοποιεί για να απαντήσει σε ερωτήματα που υποβάλλουμε. Για παράδειγμα, στο προηγούμενο πρόγραμμα το κατηγόρημα `mother/2` ορίζει μία σχέση που συνδέει τις 2 παραμέτρους που αυτό δέχεται.

Οι κανόνες στην Prolog, που γράφονται με το συντακτικό που περιγράψαμε στην προηγούμενη ενότητα, καθορίζουν πώς αποδεικνύεται μία πρόταση: η κεφαλή του κανόνα είναι η πρόταση προς απόδειξη και ο κατάλογος όρων που ακολουθεί το `:-` είναι οι συνθήκες που αρκεί να αποδειχθούν από το γλωσσικό σύστημα Prolog για να ισχύει η κεφαλή του κανόνα. Με βάση την καθαρά λογική θεώρηση της prolog [7, σελ 418] το συντακτικό των κανόνων της Prolog αντιστοιχεί ακριβώς στην γλώσσα της μαθηματικής λογικής πρώτης τάξεως (κατηγορηματικός λογισμός). Για παράδειγμα οι κανόνες του προηγούμενου παραδείγματος:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(Z, Y), ancestor(X, Z).
```

αντιστοιχούν στους παρακάτω στην γλώσσα του κατηγορηματικού λογισμού:

$$\forall X, Y : parent(X, Y) \Rightarrow ancestor(X, Y)$$
$$\forall X, Y, Z : parent(Z, Y) \wedge ancestor(X, Z) \Rightarrow ancestor(X, Y)$$

Από αυτήν την οπτική γωνία κάθε πρόγραμμα Prolog αποτελείται από γεγονότα και κανόνες που αντιστοιχούν σε τύπους του κατηγορηματικού λογισμού. Επομένως η Prolog είναι μια δηλωτική γλώσσα καθώς σε μία βάση γνώσης της δεν περιγράφεται πώς πρέπει να εκτελεστεί κάποια διαδικασία αλλά περιέχονται μόνο κάποιοι ισχυρισμοί (γεγονότα και κανόνες συνεπαγωγής). Η ακριβής διαδικασία που θα χρησιμοποιήσει το γλωσσικό σύστημα για να απαντήσει στα ερωτήματα του χρήστη αποτελούν λεπτομέρειες της υλοποίησης. Αυτές οι ιδιότητες του λογικού προγραμματισμού κάνουν τα προγράμματα της Prolog πιο σύντομα από αντίστοιχα προγράμματα προστακτικών γλωσσών προγραμματισμού και μία ολόκληρη κατηγορία σφαλμάτων (πχ βήματα υπολογισμού, διαχείριση μνήμης, κτλ) απουσιάζουν.

Η Prolog όμως δεν είναι καθαρά δηλωτική γλώσσα και διαθέτει και διαδικαστική θεώρηση (πχ πολλές φορές η σειρά με την οποία γράφονται οι κανόνες και οι συνθήκες του κανόνα έχουν σημασία, υπάρχει ο τελεστής αποκοπής `!` που επιτυγχάνει πάντα αλλά μόνο μία φορά, δεν έχει δηλωτικό περιεχόμενο και ακυρώνει την οπισθοχώρηση και χρησιμοποιείται για να επιταχύνει τα προγράμματα Prolog, κτλ) την οποία

πρέπει να την λαμβάνουμε υπόψη γιατί διαφορετικά μπορούμε να κατασκευάσουμε προγράμματα που δεν τερματίζουν ή εμφανίζουν μεγάλες καθυστερήσεις. Επίσης για να καταλάβουμε την μέθοδο με την οποία η Prolog δίνει απαντήσεις στα ερωτήματα που της υποβάλλουμε πρέπει να κατανοήσουμε την διαδικαστική της λειτουργία. Για να μπορέσουμε να την κατανοήσουμε πρέπει πρώτα να περιγράψουμε τις τεχνικές που χρησιμοποιεί ο διερμηνευτής της Prolog για να δίνει απαντήσεις στα ερωτήματα που υποβάλλουμε.

Στον πυρήνα του μηχανισμού εκτέλεσης της Prolog βρίσκεται η τεχνική συμμόρφωσης προτύπων που λέγεται **ενοποίηση**[7]. Για παράδειγμα στην βάση γνώσης του προηγούμενου παραδείγματος αν θέσουμε το ερώτημα `mother(alice,charlie)` η Prolog θα μπορέσει να το αποδείξει και θα μας απαντήσει ότι ισχύει γιατί υπάρχει στην βάση γνώσης. Τα ερωτήματα όμως και οι κανόνες που ορίζουμε περιλαμβάνουν μεταβλητές όποτε χρειαζόμαστε μία συνάρτηση που θα αντιστοιχεί τις μεταβλητές σε όρους ώστε να μπορεί να εφαρμοστεί η ενοποίηση. Αυτή η συνάρτηση καλείται **αντικατάσταση**. Το αποτέλεσμα της εφαρμογής μίας αντικατάστασης σε έναν όρο καλείται στιγμιότυπο του όρου. Αν το στιγμιότυπο που προκύπτει μπορεί να ενοποιηθεί τότε ο διερμηνευτής της Prolog απαντάει θετικά και αναφέρει την αντικατάσταση που χρησιμοποιήθηκε. Για παράδειγμα, στο παρακάτω ερώτημα:

```
?- mother(X,Z),father(Y,Z).
X = alice,
Z = charlie,
Y = bob.
```

χρησιμοποιήθηκε η αντικατάσταση: $\sigma = \{X \rightarrow alice, Z \rightarrow charlie, Y \rightarrow bob\}$ και το στιγμιότυπο που προέκυψε μπόρεσε να ενοποιηθεί με τις φράσεις που αναφέρονται στην βάση γνώσης.

Εσωτερικά η Prolog αναζητά αντικαταστάσεις που μπορούν να ενοποιήσουν τους όρους. Αν το επιτύχει αυτό τότε απαντάει καταφατικά και αναφέρει στον χρήστη την αντικατάσταση που χρησιμοποιήθηκε. Αυτή η αντικατάσταση, που παίρνει δύο όρους της Prolog ώστε τα στιγμιότυπα που προκύπτουν να είναι ίδια, καλείται **ενοποιητής**. Είναι πιθανό για κάποιο ερώτημα να υπάρχουν πολλοί ενοποιητές που μπορούν να το απαντήσουν οπότε σε αυτήν την περίπτωση η Prolog χρησιμοποιεί την διαδικασία της **οπισθοχώρησης** για να τους αναζητήσει και να τους παρουσιάσει στον χρήστη, πχ:

```
?- parent(X,charlie).
X = alice ;
X = bob.
```

Αν το γλωσσικό σύστημα της Prolog δεν μπορέσει να βρει κάποιον ενοποιητή για τους

όρους του ερωτήματος τότε απαντάει αρνητικά. Για παράδειγμα οι όροι `f(X,b)` και `f(a,Y)` μπορούν να ενοποιηθούν με τον ενοποιητή $\{X \rightarrow a, Y \rightarrow b\}$. Σημειώνουμε εδώ ότι, όπως φαίνεται από τα γενικά σημεία της διαδικασίας απόδειξης που περιγράψαμε, η Prolog υλοποιεί την υπόθεση του κλειστού κόσμου (closed-world assumption, CWA[18]). Στην Prolog κάθε πρόταση που είναι αληθινή είναι γνωστό ότι είναι αληθινή, δηλαδή αναφέρεται στην βάση γνώσης που έχουμε φορτώσει ή προκύπτει από αυτήν (και τα εσωτερικά κατηγορήματα της Prolog) με την διαδικασία απόδειξης που αναφέραμε. Όλες οι υπόλοιπες προτάσεις θεωρούνται μη αληθείς.

Χρησιμοποιώντας τα παραπάνω μπορούμε να περιγράψουμε την διαδικαστική θεώρηση της Prolog[7]. Σε αυτήν κάθε κανόνας θεωρείται μια διαδικασία απόδειξης ενός στόχου. Έστω για παράδειγμα παρακάτω κανόνες:

```
p :- q, r.
q.
r.
```

Οι κανόνες και τα γεγονότα μπορούν να θεωρηθούν ως μια διαδικασία απόδειξης στόχων. Για να αποδείξει κάποιον στόχο ο διερμηνευτής της Prolog προσπαθεί αρχικά να τον ενοποιήσει με τον όρο p και αν τα καταφέρει προσπαθεί στην συνέχει να αποδείξει την πρόταση/στόχο q και την r. Εάν και τα τρία βήματα ολοκληρωθούν με επιτυχία ο αρχικός στόχος έχει αποδειχθεί ενώ αν κάποιο αποτύχει τότε ο δεν μπορεί να αποδειχθεί από αυτήν την διαδικασία και ο διερμηνευτής της Prolog θα αναζητήσει κάποιον άλλο κανόνα/διαδικασία για να δοκιμάσει το ίδιο. Φυσικά μία τέτοια διαδικασία απόδειξης θα ξεκινήσει και για τους επιμέρους στόχους της αποδεικτικής διαδικασίας. Με την διαδικασία της αντικατάστασης που περιγράψαμε η Prolog θα βρίσκει ενοποιητές για τους όρους και με την διαδικασία οπισθοχώρησης θα αναζητά όλους τους διαφορετικούς τρόπους που μπορεί να γίνει η απόδειξη.

Είδαμε λοιπόν τις δύο διαφορετικές θεωρήσεις της Prolog, την δηλωτική και την διαδικαστική, και περιγράψαμε σε γενικές γραμμές την αποδεικτική διαδικασία που ακολουθεί. Τα χαρακτηριστικά αυτά της Prolog την κάνουν πολύ χρήσιμη σε προβλήματα που ορίζουμε τις οντότητες του κόσμου μας και τις συσχετίσεις μεταξύ τους ώστε με την γενική διαδικασία απόδειξης να μπορεί να δίνει απαντήσεις και όλες τις δυνατές λύσεις ενός στόχου χωρίς να χρειάζεται να περιγράψουμε την διαδικασία κάθε φορά, αποφεύγοντας με αυτόν τον τρόπο συνηθισμένα προγραμματιστικά λάθη. Αυτά τα χαρακτηριστικά της Prolog τα εκμεταλλεύεται και τα επεκτείνει ο Γοργίας προσθέτοντας προσαρμόσιμες πολιτικές προτίμησης και λογική απαγωγή που μας επιτρέπει να φτιάξουμε γνωσιακούς βοηθούς που επεκτείνονται εύκολα και δίνουν απαντήσεις σε δυναμικά περιβάλλοντα με ελλιπή γνώση.

Παράρτημα Β΄

Κώδικας

Β΄.1 Ιεραρχίας κληρονομικότητας

```
% inheritance hierarchy, higher order preferences example

:- compile('./lib/gorgias.pl').
:- compile('./ext/lpwnf.pl').

rule(f1, def_subclass(a,b), []).
rule(f2, def_subclass(c,b), []).
rule(f3, def_subclass(d,c), []).
rule(f4, def_is_in(x1,a),   []).
rule(f5, def_is_in(x2,c),   []).
rule(f6, def_is_in(x3,d),   []).

rule(d1(X), has(X,p),      [is_in(X,b)]).
rule(d2(X), neg(has(X,p)), [is_in(X,c)]).

rule(pr1(X), prefer(d2(X), d1(X)), []).

% modification needed when not using higher order preferences
%rule(pr1(X), prefer(d2(X), d1(X)), [neg(is_in(X,a))]).

% General properties of subclass and is_in
rule(r1(C1,C2), subclass(C1,C2), [def_subclass(C1,C2)]).
rule(r2(C0,C2), subclass(C0,C2), [def_subclass(C0,C1), subclass(C1,C2)]).
rule(r3(X,C),   is_in(X,C),      [def_is_in(X,C)]).
rule(r4(X,C),   is_in(X,C),      [subclass(S,C), is_in(X,S)]).
```

```
% extension
rule(f7, def_subclass(d,a), []).
rule(pr2(X), prefer(d2(X), d2(X)), [is_in(X,a)]).

% higher order preference
rule(pr3(X), prefer(pr2(X), pr1(X)), [is_in(X,a)]).

% prove helper

prove_subclass(X,Y) :-
    prove([subclass(X,Y)],Delta),
    write(X), write(" is subclass of "), write(Y),
    write(" because "), write(Delta),
    nl, !. % use a cut to not go further

prove_subclass(X,Y) :-
    write(X), write(" is NOT a subclass of "), write(Y),
    nl. % goal did not succeed

prove_is_in(X,Y) :-
    prove([is_in(X,Y)],Delta),
    write(X), write(" is in "), write(Y),
    write(" because "), write(Delta),
    nl, !.

prove_is_in(X,Y) :-
    write(X), write(" is NOT in "), write(Y),
    nl.

prove_has_property(X) :-
    prove([has(X,p)],Delta),
    write(X), write(" has property p because "), write(Delta),
    nl, !.

prove_has_property(X) :-
    write("No admissible subset for "), write(X),
    write(" having property p"), nl.

prove_not_has_property(X) :-
    prove([neg(has(X,p))],Delta),
    write(X), write(" does NOT have property p because "), write(Delta),
    nl, !.
```

```
prove_not_has_property(X) :-
    write("No admissible subset for "), write(X),
    write(" NOT having property p"), nl.

start :-
    prove_subclass(a,b),
    prove_subclass(c,b),
    prove_subclass(d,c),
    prove_subclass(d,a),
    prove_subclass(d,b),
    prove_is_in(x1,a),
    prove_is_in(x2,c),
    prove_is_in(x1,b),
    prove_is_in(x2,b),
    prove_has_property(x1),
    prove_not_has_property(x2),
    prove_has_property(x3),
    prove_not_has_property(x3).
```

Β΄.2 Λογική απαγωγή στον Γοργία

```
% abducible example - file access permissions

:- compile('./lib/gorgias.pl').
:- compile('./ext/lpwnf.pl').

rule(owner_changes_permissions(U,F),
     can_change_permissions(U,F),
     [is_file_owner(U,F)]).
rule(root_changes_permissions(U,F),
     can_change_permissions(U,F),
     [is_root(U)]).
rule(change_permissions(U,F),
     has_write_permission(U,F),
     [can_change_permissions(U,F)]).
rule(write_file(U,F),
     can_write_file(U,F),
     [has_write_permission(U,F)]).

abducible(is_root(_),                []).
abducible(is_file_owner(_,_),        []).
abducible(has_write_permission(_,_), []).

prove_can_write_file(U,F) :-
    prove([can_write_file(U,F)],Delta),
    write(U), write(" can write file "), write(F), write(" when "),
    write(Delta), nl.

prove_can_write_file(U,F) :-
    write(U), write(" cannot write file "), write(F), nl.

start :-
    prove_can_write_file(user, foo).
```

Β΄.3 Λογική απαγωγή και κανόνες προτίμησης ανώτερης τάξης

```
% platypus example

:- compile('./lib/gorgias.pl').
:- compile('./ext/lpwnf.pl').

rule(monotreme_mammal(X), mammal(X),      [monotreme(X)]).
rule(fur_mammal(X),       mammal(X),      [hasFur(X)]).
rule(eggs_notmammal(X),   neg(mammal(X)), [laysEggs(X)]).
rule(bill_notmammal(X),   neg(mammal(X)), [hasBill(X)]).

rule(platypus_monotreme, monotreme(platypus), []).
rule(platypus_fur,       hasFur(platypus),    []).
rule(platypus_eggs,      laysEggs(platypus),  []).
rule(platypus_bill,      hasBill(platypus),   []).

rule(duck_has_bill,      hasBill(duck),       []).

rule(pr1(X), prefer(monotreme_mammal(X), eggs_notmammal(X)),   []).
rule(pr2(X), prefer(fur_mammal(X),       bill_notmammal(X)),   []).
rule(pr3(X), prefer(bill_notmammal(X),   monotreme_mammal(X)), []).
rule(pr4(X), prefer(eggs_notmammal(X),   fur_mammal(X)),       []).

rule(prpr1(X), prefer(pr1(X),pr4(X)), [monotreme(X)]).

% assert/retract abducible

do_assert :-
    asserta(abducible(monotreme(_),      [])),
    asserta(abducible(hasFur(_),         [])),
    asserta(abducible(laysEggs(_),       [])),
    asserta(abducible(hasBill(_),        [])),
    asserta(abducible(neg(monotreme(_)), [])),
    asserta(abducible(neg(hasFur(_)),    [])),
    asserta(abducible(neg(laysEggs(_)),  [])),
    asserta(abducible(neg(hasBill(_)),   [])).

do_retract :-
    retract(abducible(monotreme(_),      [])),
    retract(abducible(hasFur(_),         [])),
```

```
    retract(abducible(laysEggs(_),       [])),
    retract(abducible(hasBill(_),        [])),
    retract(abducible(neg(monotreme(_)), [])),
    retract(abducible(neg(hasFur(_)),    [])),
    retract(abducible(neg(laysEggs(_)),  [])),
    retract(abducible(neg(hasBill(_)),   [])).

% prove helper

prove_mammal(X) :-
    prove([(mammal(X))],Delta),
    write(X), write(" is a mammal because:"), nl,
    pretty(Delta), nl.

prove_mammal(X) :-
    write("no admissible subset for "), write(X), write(" is a mammal"), nl.

prove_not_mammal(X) :-
    prove([(neg(mammal(X)))],Delta),
    write(X), write(" is not a mammal because:"), nl,
    pretty(Delta), nl.

prove_not_mammal(X) :-
    write("no admissible subset for "), write(X), write(" is not a mammal"), nl.

platypus :- prove_mammal(platypus), prove_not_mammal(platypus).

duck :- prove_mammal(duck), prove_not_mammal(duck).

something :- prove_mammal(something).
otherthing :- prove_not_mammal(otherthing).
```

B′.4 Logic Interface

```
package logic;

import java.util.List;

public interface Logic {

    public abstract void load(String file);
    public abstract void claim(String condition);
    public abstract void claim(String condition, String label);
    public abstract boolean test(String condition);
    public abstract List<String> query(String variable, String condition);
    public abstract List<List<String>> query(List<String> variables, String condition);
    public abstract List<List<String>> why(); // explain the last test()/query()
    public abstract List<String> listPredicates();
    public abstract void disclaimAll();
    public abstract void disclaimLast();
    public abstract void disclaim(String condition);
    public abstract String negate(String condition);

}
```

B′.5 Prolog Class

```
package logic;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import org.jpl7.Query;
import org.jpl7.Term;

public class Prolog implements Logic {

    private Query q;
    private List<List<String>> explanation = new ArrayList<>();
    private final List<String> claims = new LinkedList<>();
    private final List<String> loadedFiles = new ArrayList<>();

    @Override
    public void load(String file) {
        String string = "consult('" + file + "').";
        q = new Query(string);
        q.hasSolution();
        loadedFiles.add(file);
    }

    @Override
    public void claim(String condition) {
        if (claims.contains(condition)) { // avoid duplicate claims
            disclaim(condition);
        }
        String string = "assert(" + condition + ").";
        q = new Query(string);
        q.hasSolution();
        claims.add(condition);
    }

    @Override
    public void claim(String condition, String label) {
        claim(condition);
    }

    @Override
```

```
 public void disclaimLast() {
     String string = "retract(" + claims.get(claims.size() - 1) + ").";
     q = new Query(string);
     q.hasSolution();
     claims.remove(claims.size() - 1);
 }

 @Override
 public boolean test(String condition) { //throws ↩
org.jpl7.PrologException {
     String string = condition + ".";
     boolean bool;
     q = new Query(string);
     try {
         bool = q.hasMoreSolutions();
         q.close(); // we don't care for more solutions
     } catch(org.jpl7.PrologException e) {
         bool = false;
     }
     explanation = new ArrayList<>();
     explanation.add(new ArrayList<>());
     return bool;
 }

 @Override
 public List<String> query(String variable, String condition) {
     List<String> listOfVariables = new ArrayList<>();
     listOfVariables.add(variable);
     List<List<String>> listOfLists = query(listOfVariables, condition);
     List<String> reply = new ArrayList<>();
     for (List list : listOfLists) {
         reply.add((String) list.get(0));
     }
     return reply;
 }

 @Override
 public List<List<String>> query(List<String> variables, String ↩
condition) {
     Map<String, Term> terms;
     List<List<String>> reply = new ArrayList();
     explanation = new ArrayList<>(); // reset explanation
     q = new Query(condition + ".");
```

```
    while (q.hasNext()) {
        terms = q.next();
        List<String> solution = new ArrayList<>();
        for (String variable : variables) {
            solution.add(terms.get(variable).toString());
        }
        reply.add(solution);
        explanation.add(new ArrayList<>());
    }
    return reply;
}

@Override
public List<List<String>> why() {
    return explanation;
}

@Override
public List<String> listPredicates() {
    List<String> predicates = new ArrayList<>();
    for (String claim : claims) {
        predicates.add(claim + ".");
    }
    return predicates;
}

@Override
public void disclaimAll() {
    for (int i = claims.size() - 1; i >= 0; i--) {
        q = new Query("retract(" + claims.get(i) + ").");
        q.hasSolution();
        claims.remove(i);
    }
}

@Override
public String toString() {
    String reply = "";
    for (String file : loadedFiles) {
        reply += "consult(" + file + ").\n";
    }
    for (String predicate : listPredicates()) {
        reply += predicate + "\n";
```

```
        }
        return reply;
    }

    @Override
    public void disclaim(String condition) {
        if (claims.indexOf(condition) != -1) {
            q = new Query("retract(" + condition + ").");
            q.hasSolution();
            claims.remove(condition);
        }
    }

    @Override
    public String negate(String condition) {
        return "not(" + condition + ")"; // not(foo). deprecated, use \+ foo.
    }

}
```

B′.6 Gorgias Class

```
package logic;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import org.jpl7.Query;
import org.jpl7.Term;

public class Gorgias implements Logic {

    private Query q;

    private final String factPrefix = "f";
    private int factCounter = 0;

    private List<List<String>> explanation = new ArrayList<>();
    private final Map<String, String> claimsToRules = new LinkedHashMap<>();
    private final List<String> loadedFiles = new ArrayList<>();

    @Override
    public void load(String file) {
        q = new Query("consult('" + file + "').");
        q.hasSolution();
        loadedFiles.add(file);
        //q.close(); // is this necessary?
    }

    @Override
    public void claim(String condition) {
        claim(condition, "");
    }

    @Override
    public void claim(String condition, String label) {
        if (claimsToRules.containsKey(condition)) { // avoid duplicate claims
            disclaim(condition);
        }
        if (label.equals("")) {
            label = factPrefix + ++factCounter;
        }
```

```
    q = new Query(wrapAssert(condition, label));
    q.hasSolution();
    //q.close(); // is this necessary?
    claimsToRules.put(condition, "rule(" + label + "," + condition + ↩
",[])");
 }

 @Override
 public void disclaimLast() {
    String lastClaim = (String) ↩
claimsToRules.keySet().toArray()[claimsToRules.size() - 1];
    String lastRule = claimsToRules.get(lastClaim);
    q = new Query("retract(" + lastRule + ").");
    q.hasSolution();
    claimsToRules.remove(lastClaim);
 }

 private String wrapAssert(String condition, String label) {
    String string;
    string = "assert(rule(";
    string += label;
    string += "," + condition;
    string += ",[])).";
    return string;
 }

 @Override
 public boolean test(String condition) {
    Map<String, Term> termmap;
    boolean bool;
    explanation = new ArrayList<>(); // reset explanation
    q = new Query(wrapProve(condition,"Delta"));
    bool = q.hasMoreSolutions();
    if (bool) {
        termmap = q.next();
        List<String> deltaList = new ArrayList<>();
        for (Term term : termmap.get("Delta").toTermArray()) {
            deltaList.add(term.toString());
        }
        explanation.add(deltaList);
    }
    q.close(); // we don't care for more solutions
    return bool;
```

```
}

private String wrapProve(String condition, String delta) {
    String string;
    string = "prove([";
    string += condition;
    string += "]," + delta + ").";
    return string;
}

@Override
public List<String> query(String variable, String condition) {
    List<String> listOfVariables = new ArrayList<>();
    listOfVariables.add(variable);
    List<List<String>> listOfLists = query(listOfVariables, condition);
    List<String> reply = new ArrayList<>();
    for (List list : listOfLists) {
        reply.add((String) list.get(0));
    }
    return reply;
}

@Override
public List<List<String>> query(List<String> variables, String ↩
condition) {
    Map<String, Term> terms;
    List<List<String>> reply = new ArrayList<>();
    String delta = getDelta(variables);
    explanation = new ArrayList<>(); // reset explanation
    q = new Query(wrapProve(condition, delta));
    while (q.hasNext()) {
        terms = q.next();
        List<String> solution = new ArrayList<>();
        for (String variable : variables) {
            solution.add(terms.get(variable).toString());
        }
        reply.add(solution);
        List<String> solutionDelta = new ArrayList<>();
        for (Term term : terms.get(delta).toTermArray()) {
            solutionDelta.add(term.toString());
        }
        explanation.add(solutionDelta);
    }
```

```
    return reply;
}

private String getDelta(List<String> variables) {
    String delta = "Delta";
    if (variables.indexOf(delta) != -1) {
        for (String variable : variables) {
            delta += variable;
        }
    }
    return delta;
}

@Override
public List<List<String>> why() {
    return explanation;
}

@Override
public List<String> listPredicates() {
    List<String> rules = new ArrayList<>();
    for (String claim : claimsToRules.keySet()) {
        rules.add(claimsToRules.get(claim) + ".");
    }
    return rules;
}

@Override
public void disclaimAll() {
    String[] claims = claimsToRules.keySet().toArray(
            new String[claimsToRules.keySet().size()]
    );
    for (int i = claimsToRules.size() - 1; i >= 0; i--) {
        q = new Query("retract(" + claims[i] + ").");
        q.hasSolution();
        claimsToRules.remove(claims[i]);
    }
    factCounter = 0;
}

@Override
public String toString() {
    String reply = "";
```

```
        for (String file : loadedFiles) {
            reply += "consult(" + file + ").\n";
        }
        for (String predicate : listPredicates()) {
            reply += predicate + "\n";
        }
        return reply;
    }

    @Override
    public void disclaim(String claim) {
        if (claimsToRules.containsKey(claim)) {
            q = new Query("retract(" + claimsToRules.get(claim) + ").");
            q.hasSolution();
            claimsToRules.remove(claim);
        }
    }

    @Override
    public String negate(String condition) {
        return "neg(" + condition + ")";
    }

}
```

Β΄.7 Παραδείγματα χρήσης κλάσεων Logic, Prolog, Gorgias

```
import java.util.ArrayList;
import java.util.List;
import logic.*;

public class logic {

    public static void testLogic(Logic l) {

        System.out.println("\n> Claim alice mother, bob father of charlie, ↩
   david...");
        l.claim("mother(alice,charlie)");
        l.claim("father(bob,charlie)");
        l.claim("mother(alice,david)");
        l.claim("father(bob,david)");

        System.out.println("\n> Is alice mother, bob father of charlie?");
        System.out.println(l.test("mother(alice,charlie)"));
        System.out.println(l.test("parent(bob,charlie)"));

        System.out.println("\n> Is bar foo of baz?");
        try {
            System.out.println(l.test("foo(bar,baz)"));
        } catch (org.jpl7.PrologException e) {
            System.out.println("a prolog error caught");
            //e.printStackTrace();
        }

        System.out.println("\n> Parents of charlie?");
        List<String> solution;
        solution = l.query("X", "parent(X,charlie)");
        for (String string : solution) {
            System.out.println(string);
        }

        System.out.println("\n> Why?");
        List<List<String>> explanations;
        explanations = l.why();
        for (int i = 0; i < explanations.size(); i++) {
            System.out.print(solution.get(i) + " because ");
            for (String argument : (List<String>) explanations.get(i) ) {
                System.out.print(argument + ", ");
```

```
        }
        System.out.println();
    }

    System.out.println("\n> Parent/child pairs?");
    List<List<String>> parents_children;
    List<String> variables = new ArrayList<>();
    variables.add("X");
    variables.add("Y");
    parents_children = l.query(variables, "parent(X,Y)");
    for (List parent_child : parents_children) {
        System.out.println(parent_child);
    }

    System.out.println("\n> Why?");
    for (List explanation : l.why()) {
        System.out.println(explanation);
    }

}

public static void banana(Logic l) {
    boolean b;
    System.out.println("\n> About bananas");
    bananaQuestions(l);
    System.out.println("\n> Claim banana is too ripe and ask again");
    l.claim("tooripe(b)", "banana_b_is_ripe");
    bananaQuestions(l);
}

public static void bananaQuestions(Logic l) {
    boolean b;
    b = l.test("color(b,yellow)");
    System.out.println("- Banana yellow? " + b);
    System.out.println("-- because " + l.why());
    b = l.test("color(b,black)");
    System.out.println("- Banana black? " + b);
    System.out.println("-- because " + l.why());
    b = l.test("taste(b,sweet)");
    System.out.println("- Banana sweet? " + b);
    System.out.println("-- because " + l.why());
    b = l.test("taste(b,mushy)");
    System.out.println("- Banana mushy? " + b);
```

```
        System.out.println("-- because " + l.why());
    }

    public static void multiple() { // doesn't work, there is always one ↩
instance of prolog
        Logic a, b;
        String time_a, time_b;
        a = new Gorgias();
        b = new Gorgias();
        a.claim("time(day)");
        b.claim("time(night)");
        time_a = a.query("X", "time(X)").get(0);
        time_b = b.query("X", "time(X)").get(0);
        System.out.println("a says " + time_a);
        System.out.println("b says " + time_b);
    }

    public static void listAndClean(Logic l) {
        System.out.println("\nPredicates before ↩
clean:\n------------------------");
        for (String predicate : l.listPredicates()) {
            System.out.println(predicate);
        }

        l.disclaimAll();

        System.out.println("\npredicates after ↩
clean:\n-----------------------");
        for (String predicate : l.listPredicates()) {
            System.out.println(predicate);
        }

        System.out.println("\nClaim something:\n----------------");
        l.claim("foo(bar,baz)");
        System.out.println(l);

        System.out.println("Test it:\n-------");
        System.out.println(l.test("foo(bar,baz)"));

        System.out.println("\nDisclaim last:\n--------------");
        l.disclaimLast();
        System.out.println(l);
```

```
        System.out.println("Test it:\n-------");
        System.out.println(l.test("foo(bar,baz)"));
    }

    public static void numbers(Logic l) {
        System.out.println("\nnumbers:\n--------");
        System.out.println("claim a is 4.0 and b is 6.0");
        l.claim("valueof(a,4.0)", "a_is_4");
        l.claim("valueof(b,6.0)", "b_is_6");
        System.out.println(
        "is a larger than 5? " + l.test("largerthan(a,5.0)") + ", because " + ↩
l.why() + "\n" +
        "is b larger than 5? " + l.test("largerthan(b,5.0)") + ", because " + ↩
l.why()
        );
    }

    public static void queryWithNoSolution(Logic l) {
        System.out.println("\nChildren of Charlie\n--------------------");
        System.out.println(l.query("X", "parent(charlie,X)"));
        System.out.println("Result is empty? " + l.query("X", ↩
"parent(charlie,X)").isEmpty());
    }

    public static void testNegation(Logic l) {
        System.out.println("\nTest Negation\n-------------");
        boolean b;
        String cond;

        cond = "increasing(1,2,3)";
        b = l.test(cond);
        System.out.println(cond + "?\t" + b + ", because " + l.why());
        b = l.test(l.negate(cond));
        System.out.println("not " + cond + "?\t" + b + ", because " + l.why());

        cond = "decreasing(1,2,3)";
        b = l.test(cond);
        System.out.println(cond + "?\t" + b + ", because " + l.why());
        b = l.test(l.negate(cond));
        System.out.println("not " + cond + "?\t" + b + ", because " + l.why());

        cond = "increasing(3,2,1)";
        b = l.test(cond);
```

```
        System.out.println(cond + "?\t" + b + ", because " + l.why());
        b = l.test(l.negate(cond));
        System.out.println("not " + cond + "?\t" + b + ", because " + l.why());

        cond = "decreasing(3,2,1)";
        b = l.test(cond);
        System.out.println(cond + "?\t" + b + ", because " + l.why());
        b = l.test(l.negate(cond));
        System.out.println("not " + cond + "?\t" + b + ", because " + l.why());

        System.out.println("\nHow negation looks: " + l.negate("foo(bar)"));
    }

    public static void main(String[] args) {

        System.out.println("\nProlog\n-------");
        Logic prolog = new Prolog();
        prolog.load("prolog.pl");
        testLogic(prolog);

        System.out.println("\nGorgias\n-------");
        Logic gorgias = new Gorgias();
        gorgias.load("gorgias.pl");
        testLogic(gorgias);

        gorgias.load("banana.pl");
        banana(gorgias);

        listAndClean(prolog);
        listAndClean(gorgias);

        prolog.load("numbers.pl");
        gorgias.load("numbers.pl"); // we don't have to load both because ↩
    there is always one prolog running
        numbers(prolog);
        numbers(gorgias);

        queryWithNoSolution(prolog);
        queryWithNoSolution(gorgias);

        prolog.load("increasing.pl");
        gorgias.load("increasing.pl");
        testNegation(prolog);
```

```
        testNegation(gorgias);

        System.out.println("\nTypes of objects\n----------------");
        System.out.println("prolog is " + prolog.getClass());
        System.out.println("gorgias is " + gorgias.getClass());

    }

}
```

Β΄.8 Διασύνδεση Java - R, Statistics.java

```
import org.rosuda.JRI.REXP;
import org.rosuda.JRI.Rengine;

public class Statistics {

    private final Rengine re;

    public Statistics() {
        String[] args = {"--vanilla"};
        re = new Rengine(args, false, null);
    }

    public void load(String rfile) {
        String string = "source('" + rfile + "')";
        re.eval(string);
    }

    public void closeR() {
        re.end();
    }

    public void printData(String variable) {
        REXP x;
        System.out.println(x = re.eval(variable));
    }

    /* ----------- */
    /* CHISQ STUFF */
    /* ----------- */

    public void chisq(String variable) {
        String string = variable + "_chisq = chisq.test(" + variable + ")";
        re.eval(string);
    }

    public double getChisqStat(String variable) {
        String string = variable + "_chisq$statistic";
        return re.eval(string).asDouble();
    }

    public int getChisqDf(String variable) {
```

```
        String string = variable + "_chisq$parameter";
        return re.eval(string).asInt();
    }

    public double getChisqPValue(String variable) {
        String string = variable + "_chisq$p.value";
        return re.eval(string).asDouble();
    }

    public double getMinExpected(String variable) {
        String string = "min(" + variable + "_chisq$expected)";
        return re.eval(string).asDouble();
    }

    /* ------------ */
    /* FISHER STUFF */
    /* ------------ */

    public void fisher(String variable) {
        String string = variable + "_fisher = fisher.test(" + variable + ")";
        re.eval(string);
    }

    public double getFisherPValue(String variable) {
        String string = variable + "_fisher$p.value";
        return re.eval(string).asDouble();
    }

}
```

Β΄.9 Διασύνδεση R - Γοργία, stats.java

```
import logic.*;

public class stats {

    public static void chisq(Logic l, Statistics s, String data) {
        s.chisq(data);
        double pvalue = s.getChisqPValue(data);
        double minExpected = s.getMinExpected(data);

        l.claim("chisq(" + data + ")"); // because many data/objects can be ↩
    loaded
        l.claim("chisq_pvalue(" + data + "," + pvalue + ")");
        l.claim("chisq_minexpected(" + data + "," + minExpected + ")");
    }

    public static void fisher(Logic l, Statistics s, String data) {
        s.fisher(data);
        double pvalue = s.getFisherPValue(data);

        l.claim("fisher(" + data + ")"); // because many data/objects can be ↩
    loaded
        l.claim("fisher_pvalue(" + data + "," + pvalue + ")");
    }

    public static void main(String[] args) {

        //String data = "data3";
        String[] allData = {"data1", "data2", "data3", "data4"};
        double confidence = 0.95;
        double significance = 1 - confidence;

        Statistics s = new Statistics();

        Logic l = new Gorgias();
        l.load("stats.pl");
        l.claim("significance(" + significance + ")");

        for (String data: allData) {

            s.load(data + ".txt");
            chisq(l, s, data);
```

```
            if (l.test("chisq_valid(" + data + ")")) {
            } else {
                l.disclaim("chisq(" + data + ")"); // not needed anymore but ↩
    nice to have - remind that chisq is not valid for this data
                fisher(l, s, data);
            }

            boolean rejecth0 = l.test("rejecth0(" + data + ")");
            if (rejecth0) {
                System.out.println("Null hypotheses rejected, " + data + " are ↩
    codependent.");
            } else {
                System.out.println("Cannot reject null hypotheses, " + data + ↩
    " are independent");
                l.test(l.negate("rejecth0(" + data + ")")); // just to fill ↩
    the explanation
            }
            System.out.println("Why? " + l.why());
            System.out.println();

        }

        System.out.println("Gorgias state");
        System.out.println("-------------");
        System.out.println(l);

        // close R
        s.closeR();

    }
}
```

Β΄.10 Έλεγχος υποθέσεων στον Γοργία, stats.pl

```
% stats.pl - chisq & fisher

% load gorgias
:- compile('lib/gorgias.pl').
:- compile('ext/lpwnf.pl').

rule(chisq_is_valid(X),
     chisq_valid(X),
     [chisq(X)]).

rule(chisq_is_not_valid(X),
     neg(chisq_valid(X)),
     [chisq(X),
      chisq_minexpected(X,MinExpected),
      MinExpected < 5]).

rule(prefer_chisq_is_not_valid(X),
     prefer(chisq_is_not_valid(X),
            chisq_is_valid(X)),
     []).

rule(h0_is_not_rejected(X),
     neg(rejecth0(X)),
     []).

rule(chisq_rejects_h0(X),
     rejecth0(X),
     [chisq(X),
      chisq_valid(X),
      chisq_pvalue(X,Pvalue),
      significance(Significance),
      Pvalue < Significance]).

rule(prefer_chisq_rejects_h0(X),
     prefer(chisq_rejects_h0(X),
            h0_is_not_rejected(X)),
     []).

rule(fisher_rejects_h0(X),
     rejecth0(X),
     [fisher(X),
```

```
      fisher_pvalue(X,Pvalue),
      significance(Significance),
      Pvalue < Significance]).

rule(prefer_fisher_rejects_h0(X),
     prefer(fisher_rejects_h0(X),
            h0_is_not_rejected(X)),
     []).
```

Βιβλιογραφία

[1] A. C. Kakas, P. Mancarella - The Acceptability Semantics for Logic Programs, 1994

[2] Y. Dimopoulos, A. Kakas - Logic Programming without Negation as Failure, 1995

[3] A. Kakas, F. Toni, P. Mancarella - Argumentation Logic, 2012

[4] N. I. Spanoudakis, E. Constantinou, A. Koumi, A. C. Kakas - Modeling Data Access Legislation with Gorgias, 2017

[5] Σ. Δημητριάδης - Χρήση Λογικής Επιχειρηματολογίας στη Δημιουργία Τειχών Προστασίας, 2008

[6] Δ. Φουσκάκης - Ανάλυση Δεδομένων με Χρήση της R, 2013, ISBN: 9786188074156

[7] Adam Brooks Webber - Σύγχρονες Γλώσσες Προγραμματισμού, 2009, ISBN: 9789605242824

[8] Rogers Cadenhead - Laura Lemay, Πλήρες Εγχειρίδιο της Java 6, 2009, ISBN: 9789605125387

[9] SWI-Prolog, http://www.swi-prolog.org

[10] JPL: A bidirectional Prolog/Java interface, http://www.swi-prolog.org/packages/jpl/

[11] (σελ 18,19) https://eclass.uoa.gr/modules/document/file.php/LAW169/H Ερμηνεία του Συντάγματος (3135) - Οικονομοπούλου Φωτεινή-Ελένη.pdf

[12] Gorgias, An Argumentation System with Abduction, http://www.cs.ucy.ac.cy/ nkd/gorgias/

[13] Gorgias-B Argumentation Tool, http://gorgiasb.tuc.gr/

[14] The R Project for Statistical Computing, https://www.r-project.org/

[15] JRI: Java/R Interface, https://www.rforge.net/JRI/

[16] Spanoudakis, N.I., Kakas, A.C., Moraitis, P.: Applications of argumentation: The SoDA methodology. In: 22nd European Conference on Artificial Intelligence, 29 Aug.-2 Sep., The Hague, The Netherlands (ECAI 2016)

[17] Alain Colmerauer, Philippe Roussel - The birth of Prolog (1996)

[18] Raymond Reiter - On closed world data bases (1978)