NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Extensions of Logic Programming for Preference Representation

Antonis K. Troumpoukis

ATHENS

JUNE 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Επεκτάσεις του λογικού προγραμματισμού για την αναπαράσταση προτιμήσεων

Αντώνιος Κ. Τρουμπούκης

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2019

PhD THESIS

Extensions of Logic Programming for Preference Representation

Antonis K. Troumpoukis

SUPERVISOR: Panagiotis Rondogiannis, Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Panagiotis Rondogiannis, Professor NKUA

Panagiotis Stamatopoulos, Assistant Professor NKUA

Christos Nomikos, Assistant Professor Univ. of Ioannina

SEVEN-MEMBER EXAMINATION COMMITTEE

Panagiotis Rondogiannis,
Professor NKUA

Panagiotis Stamatopoulos,
Assistant Professor NKUA

Christos Nomikos,
Assistant Professor Univ. of Ioannina

Manolis Gergatsoulis,
Professor Ionian University

Michael Dracopoulos,
Assistant Professor NKUA

Nikolaos Papaspyrou,
Professor NTUA

Andreas Stafylopatis,
Professor NTUA

Examination Date: June 25, 2019

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Επεκτάσεις του λογικού προγραμματισμού για την αναπαράσταση προτιμήσεων

Αντώνιος Κ. Τρουμπούκης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Παναγιώτης Ροντογιάννης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Παναγιώτης Ροντογιάννης, Καθηγητής ΕΚΠΑ

Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ

Χρήστος Νομικός, Επίκουρος Καθηγητής Παν. Ιωαννίνων

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Παναγιώτης Ροντογιάννης,
Καθηγητής ΕΚΠΑ

Παναγιώτης Σταματόπουλος,
Επίκουρος Καθηγητής ΕΚΠΑ

Χρήστος Νομικός,
Επίκουρος Καθηγητής Παν. Ιωαννίνων

Εμμανουήλ Γεργατσούλης,
Καθηγητής Ιονίου Πανεπ.

Μιχαήλ Δρακόπουλος,
Επίκουρος Καθηγητής ΕΚΠΑ

Νικόλαος Παπασπύρου,
Καθηγητής ΕΜΠ

Ανδρέας Σταφυλοπάτης,
Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης: 25 Ιουνίου 2019

ABSTRACT

We consider the problem of preference representation using extensions of logic programming. The effective representation of preferences is crucial in many scientific disciplines and it can be proven useful in many real-world applications. Preference representation formalisms in the literature usually fall into two basic categories: in the qualitative approach (where preferences are expressed with binary preference relations) and in the quantitative approach (where preferences are represented with the use of numerical values that express the degree of interest). In this dissertation, we propose two approaches for expressing preferences. The first approach uses an infinite-valued extension of logic programming for expressing quantitative preferences, while the second approach uses higher-order logic programming for expressing qualitative preferences.

We propose PrefLog, a logic programming language which uses an underlying infinite-valued truth domain in order to support quantitative preference operators. We introduce the syntax and the semantics of the language, and we study the properties of the PrefLog operators that are needed in order for programs to behave well from a semantic point of view. In addition, we introduce a terminating bottom-up evaluation method for a well-defined class of function-free PrefLog programs. Ensuring termination is not a straightforward task, because the underlying truth domain of PrefLog and the set of all possible interpretations of a function-free PrefLog program are both infinite.

We propose the use of higher-order logic programming as a framework for representing qualitative preferences. In this approach, relations, preferences between tuples, preferences between sets of tuples and operations on preferences are expressed in the same, higher-order language. The programs can be evaluated by standard higher-order programming systems, and their performance can be enhanced with generic and specialized optimization techniques. Among these techniques, we propose a novel program transformation technique for translating higher-order programs into first-order ones and we use this technique for optimizing the higher-order programs of our interest. Finally, we demonstrate the feasibility of our approach by presenting an implementation and an experimental evaluation of all the proposed concepts in the higher-order logic programming language HiLog.

SUBJECT AREA: Programming Languages

KEYWORDS: Preference Representation, Infinite-Valued Logic Programming, Higher-Order Logic Programming

ΠΕΡΙΛΗΨΗ

Εξετάζουμε το πρόβλημα της αναπαράστασης προτιμήσεων με τη χρήση επεκτάσεων του λογικού προγραμματισμού. Η αποτελεσματική αναπαράσταση προτιμήσεων είναι ζωτικής σημασίας σε πολλά επιστημονικά πεδία και μπορεί να αποδειχθεί χρήσιμη σε πολλές πραγματικές εφαρμογές. Οι φορμαλισμοί αναπαράστασης προτιμήσεων στη βιβλιογραφία συνήθως εμπίπτουν σε δύο βασικές κατηγορίες: στην ποιοτική προσέγγιση (όπου οι προτιμήσεις εκφράζονται με διμερείς σχέσεις προτίμησης) και στην ποσοτική προσέγγιση (όπου οι προτιμήσεις αναπαριστώνται με τη χρήση αριθμητικών τιμών που εκφράζουν το βαθμό ενδιαφέροντος). Σε αυτή τη διατριβή, προτείνουμε δύο προσεγγίσεις για την έκφραση προτιμήσεων. Η πρώτη προσέγγιση χρησιμοποιεί μια απειρότιμη επέκταση του λογικού προγραμματισμού για την έκφραση ποσοτικών προτιμήσεων, ενώ η δεύτερη προσέγγιση χρησιμοποιεί τον λογικό προγραμματισμό υψηλής τάξης για την έκφραση ποιοτικών προτιμήσεων.

Προτείνουμε τη γλώσσα προγραμματισμού PrefLog, μια επέκταση του λογικού προγραμματισμού που χρησιμοποιεί ένα άπειρο σύνολο τιμών αλήθειας για να υποστηρίξει τον ορισμό τελεστών ποσοτικής προτίμησης. Ορίζουμε το συντακτικό και τη σημασιολογία της γλώσσας και προσδιορίζουμε ένα σύνολο από ιδιότητες τις οποίες πρέπει να ικανοποιούν οι διαθέσιμοι τελεστές προτίμησης έτσι ώστε η γλώσσα να έχει καλώς ορισμένη σημασιολογία. Επιπλέον, προτείνουμε μία “από-κάτω-προς-τα-πάνω” τεχνική υλοποίησης για ένα καλώς ορισμένο υποσύνολο της PrefLog που αντιστοιχεί στο προτιμησιακό αντίστοιχο της γλώσσας Datalog. Η εξασφάλιση της ιδιότητας του τερματισμού μιας τέτοιας στρατηγικής δεν είναι προφανής γιατί το σύνολο των τιμών αληθείας και το σύνολο των πιθανών ερμηνειών για τέτοια προγράμματα είναι και τα δύο άπειρα.

Προτείνουμε τη χρήση του λογικού προγραμματισμού υψηλής τάξης για την αναπαράσταση ποιοτικών προτιμήσεων. Σε αυτήν την προσέγγιση, σχέσεις, προτιμήσεις μεταξύ πλειάδων, προτιμήσεις μεταξύ συνόλων από πλειάδες και υπολογισμοί σχετικά με προτιμήσεις εκφράζονται στην ίδια γλώσσα υψηλής τάξης. Τα προγράμματα αυτά μπορούν να εκτελεστούν σε πραγματικά συστήματα λογικού προγραμματισμού υψηλής τάξης και η απόδοσή τους μπορεί να ενισχυθεί είτε με γενικές είτε με εξειδικευμένες τεχνικές βελτιστοποίησης. Ανάμεσα σε αυτές, προτείνουμε μια νέα τεχνική μετατροπής λογικών προγραμμάτων υψηλής τάξης σε κλασικά λογικά προγράμματα (πρώτης τάξης) και την εφαρμόζουμε στα προγράμματα της προσέγγισής μας. Τέλος, αποδεικνύουμε την εφαρμοσιμότητα της προσέγγισής μας παρουσιάζοντας μια υλοποίηση και μια πειραματική αξιολόγηση στη γλώσσα λογικού προγραμματισμού υψηλής τάξης HiLog.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Αναπαράσταση προτιμήσεων, Απειρότιμος Λογικός Προγραμματισμός, Λογικός Προγραμματισμός Υψηλής Τάξης

Στους γονείς μου, Κώστα και Τασία
και στην αδερφή μου, Λένα

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my advisor, Prof. Panos Rondogiannis, for all his guidance, patience and support throughout my studies. From my undergraduate years until the present moment, he has been a valuable mentor for me. I would like to thank him for encouraging my research and for allowing me to grow as a research scientist. I will always feel proud to have been one of his students, and I look forward to collaborating with him in the future as well.

I would like to thank the members of the seven-member examination committee, for their kind comments and suggestions.

Special thanks to Angelos Charalambidis, a co-author and friend, for his invaluable insights and our exchange of ideas. His help had been crucial especially during the second half of the dissertation.

During the course of this Ph.D., I worked at the Software and Knowledge Engineering Lab (SKEL) at NCSR “Demokritos”. Studying for Ph.D. while working can be a difficult task, but it was made easier with the help and support of Stasinos Konstantopoulos, my supervisor in the Data Engineering Group (DEG) at SKEL. Thanks also to Giannis Mouchakis, for being a good office-mate.

Thanks also to all of my friends for their encouragement. Especially, Giorgos Papadimitriou, for the beers we drank together; Nikos Christou for the summers in Paros; and the bandmates from Lost N’ Found and Terra Gloria for the jams and the gigs.

Last but not least, I would like to express my deep gratitude to my family, for their tireless efforts, their understanding, and their unconditional love. I simply have no words to explain how I owe to them. Σας ευχαριστώ για όλα.

LIST OF PUBLICATIONS

[1] Angelos Charalambidis, Panos Rondogiannis, and Antonis Troumpoukis. Higher-order logic programming: An expressive language for representing qualitative preferences. *Sci. Comput. Program.*, 155:173–197, 2018.

[2] Antonis Troumpoukis and Angelos Charalambidis. Predicate specialization for definitional higher-order logic programs. In Fred Mesnard and Peter J. Stuckey, editors, *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers*, volume 11408 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2018.

[3] Antonis Troumpoukis, Stasinos Konstantopoulos, and Angelos Charalambidis. An extension of SPARQL for expressing qualitative preferences. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 711–727. Springer, 2017.

[4] Angelos Charalambidis, Panos Rondogiannis, and Antonis Troumpoukis. Higher-order logic programming: an expressive language for representing qualitative preferences. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 24–37. ACM, 2016.

[5] Panos Rondogiannis and Antonis Troumpoukis. Expressing preferences in logic programming using an infinite-valued logic. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 208–219. ACM, 2015.

[6] Panos Rondogiannis and Antonis Troumpoukis. The infinite-valued semantics: overview, recent results and future directions. *Journal of Applied Non-Classical Logics*, 23(1-2):213–228, 2013.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Οι προτιμήσεις παίζουν πολύ σημαντικό ρόλο στη λήψη αποφάσεων σε διάφορες καταστάσεις: από απλές προσωπικές επιλογές στην πρώιμη παιδική ηλικία (π.χ., «τι γεύση παγωτού προτιμάς;») μέχρι πολύπλοκα επαγγελματικά διλήμματα (π.χ., «να ακολουθήσω καριέρα στη Μουσική ή στην Πληροφορική;»). Ως εκ τούτου, δεν αποτελεί έκπληξη το ότι οι προτιμήσεις έχουν μελετηθεί σε πολλά επιστημονικά πεδία, όπως η Φιλοσοφία [37], τα Οικονομικά [28], και η Ψυχολογία [49]. Επιπλέον, η περιγραφή και επεξεργασία προτιμήσεων έχει πολλές εφαρμογές στην Πληροφορική κυρίως σε τομείς όπως η Τεχνητή Νοημοσύνη [24], οι Βάσεις Δεδομένων [69] και οι Γλώσσες Προγραμματισμού [22]. Βασικός στόχος της μελέτης των προτιμήσεων στην Πληροφορική είναι η δημιουργία γλωσσών και συστημάτων που θα δίνουν τη δυνατότητα τόσο σε προγραμματιστές όσο και σε απλούς χρήστες να αναπαριστούν με σαφήνεια τις προτιμήσεις τους ώστε να λαμβάνουν πιο σχετικές ή πιο περιεκτικές απαντήσεις.

Οι φορμαλισμοί που έχουν μέχρι στιγμής προταθεί για την αναπαράσταση προτιμήσεων μπορούν να χωριστούν σε δύο βασικές κατηγορίες: στους *ποσοτικούς* και τους *ποιοτικούς* φορμαλισμούς. Στους ποσοτικούς φορμαλισμούς [1, 2, 3, 45], οι προτιμήσεις αναπαριστώνται με την χρήση *αριθμητικών τιμών* που εκφράζουν τον βαθμό προτίμησης (π.χ., «η προτίμηση μου για μπύρα είναι 0.9 ενώ για κρασί είναι 0.2»). Αντίθετα, στους ποιοτικούς φορμαλισμούς [17, 19, 32, 36, 80], οι προτιμήσεις αναπαριστώνται με απευθείας συγκρίσεις π.χ., «προτιμώ μπύρα από κρασί») που οδηγούν στον ορισμό *διμερών σχέσεων προτίμησης*. Κάθε κατηγορία έχει τα δικά της πλεονεκτήματα και μειονεκτήματα. Αφενός, η ποιοτική προσέγγιση είναι πιο εκφραστική από την ποσοτική: όπως προκύπτει από τη βιβλιογραφία [17, 69], υπάρχουν διμερείς σχέσεις προτίμησης που δεν μπορούν να εκφραστούν στο ποσοτικό μοντέλο. Αφετέρου, στις ποσοτικές προσεγγίσεις μπορεί να διακριθεί το πόσο προτιμητέο είναι ένα αντικείμενο από ένα άλλο απλά συγκρίνοντας τις δυο τιμές προτίμησης (π.χ., ένα αντικείμενο που έχει βαθμό προτίμησης 0.9 είναι πολύ πιο προτιμητέο από κάποιο με βαθμό προτίμησης 0,001, αλλά είναι λίγο πιο προτιμητέο από ένα άλλο με βαθμό προτίμησης 0,899).

Η μελέτη της σχετικής βιβλιογραφίας οδήγησε στην παρατήρηση ότι τόσο οι ποιοτικές όσο και οι ποσοτικές τεχνικές που έχουν αναπτυχθεί, αφήνουν αρκετά περιθώρια για σημαντική βελτίωση, τόσο από άποψη εκφραστικότητας όσο και αποτελεσματικότητας. Οι ποσοτικές προσεγγίσεις συνήθως χρησιμοποιούν *συναρτήσεις προτιμήσεων* [3, 45] για να υπολογίζουν τους βαθμούς προτίμησης: ωστόσο, δεν είναι πάντοτε δυνατό να οριστεί μια τέτοια συνάρτηση προτίμησης (βλ. [32]). Επιπλέον, οι χρήστες στις περισσότερες περιπτώσεις δεν είναι ιδιαίτερα πρόθυμοι να ορίζουν τις προτιμήσεις τους με αριθμητικές τιμές, αλλά προτιμούν να τις εκφράζουν με δηλωτικό τρόπο (βλ. [24]). Οι ποιοτικές προτιμήσεις έχουν και αυτές τις αδυναμίες τους. Πρώτον, προσφέρουν συνήθως ένα αρκετά περιορισμένο σύνολο λειτουργιών για την επεξεργασία των προτιμήσεων [17, 80] (συνήθως προσφέρεται μόνο ένας τελεστής προτίμησης με το διαισθητικό νόημα «βρες τα πιο προτιμώμενα αντικείμενα σύμφωνα με την δοθείσα σχέση προτίμησης»). Δεύτερον, σχεδόν όλες οι προσεγγίσεις χρησιμοποιούν δύο ξεχωριστές γλώσσες, μία για την αναπαράσταση της βασικής γνώσης και μία για την αναπαράσταση των προτιμήσεων [17, 32, 36], κάνοντας τη δομή της αναπαράστασης στο σύνολό της σχετικά ανομοιόμορφη. Επιπλέον, οι ποιοτικές προσεγγίσεις βασίζονται σε δομές όπως οι *σχέσεις μερικής διάταξης*. Η διαδικασία χειρισμού μιας τέτοιας μοντελοποίησης είναι σαφώς πιο πολύπλοκη από εκείνη μιας ποσοτικής προτίμησης που βασίζεται σε αριθμητικές τιμές. Συνεπώς, η ανάπτυξη τεχνικών βελτιστοποίησης για την ενίσχυση της απόδοσης των φορμαλισμών ποιοτικών

προτιμήσεων μπορεί να είναι καθοριστικής σημασίας από πρακτική άποψη.

Σκοπός της διδακτορικής διατριβής αυτής είναι η διερεύνηση νέων φορμαλισμών για την αναπαράσταση και επεξεργασία προτιμήσεων. Πιο συγκεκριμένα, προτείνεται η χρήση δύο κατάλληλων *επεκτάσεων του λογικού προγραμματισμού* για την αναπαράσταση και επεξεργασία προτιμήσεων.

- Η πρώτη προσέγγιση χρησιμοποιεί μια επέκταση του λογικού προγραμματισμού για την έκφραση ποσοτικών προτιμήσεων. Αυτή η γλώσσα βασίζεται σε ένα *άπειρο σύνολο τιμών αληθείας* προκειμένου να υποστηρίξει ποσοτικούς τελεστές προτίμησης.

- Η δεύτερη προσέγγιση χρησιμοποιεί το *λογικό προγραμματισμό υψηλής τάξης* για την έκφραση ποιοτικών προτιμήσεων. Στο πλαίσιο αυτό, οι διμερείς σχέσεις προτίμησης και οι τελεστές προτιμήσεων εκφράζονται στην ίδια γλώσσα υψηλής τάξης.

Οι προσεγγίσεις μας προσπαθούν να ξεπεράσουν τα μειονεκτήματα που αναφέρθηκαν προηγουμένως. Στην ποσοτική μας προσέγγιση, ο χρήστης δεν ορίζει την προτίμηση του απευθείας με κάποια τιμή προτίμησης, αλλά την εκφράζει δηλωτικά, χρησιμοποιώντας κατάλληλους τελεστές προτίμησης. Στην ποιοτική μας προσέγγιση, η αναπαράσταση των προτιμήσεων είναι πιο ομοιόμορφη αφού η μοντελοποίηση της βασικής γνώσης, των προτιμήσεων και τον τελεστών που επεξεργάζονται προτιμήσεις γίνεται χρησιμοποιώντας την ίδια γλώσσα. Επιπλέον, δίνεται η δυνατότητα για τον ορισμό νέων τελεστών επεξεργασίας πάνω σε σχέσεις προτίμησης, εκτός από εκείνους που προσφέρονται στις περισσότερες ποιοτικές προσεγγίσεις στη βιβλιογραφία.

Στη συνέχεια θα περιγράψουμε τις βασικές αρχές των δύο παραπάνω τεχνικών για την αναπαράσταση προτιμήσεων, δίνοντας μια διαισθητική εισαγωγή και ένα παράδειγμα για καθεμία από τις προσεγγίσεις που αναπτύσσονται στην διατριβή.

Ποσοτικές Προτιμήσεις και Απειρότιμος Λογικός Προγραμματισμός

Η κεντρική ιδέα της προσέγγισης που χρησιμοποιεί τον απειρότιμο λογικό προγραμματισμό για την έκφραση ποσοτικών προτιμήσεων μπορεί να περιγραφεί συνοπτικά ως εξής:

«Μπορούμε να επιτύχουμε μια εκφραστική περιγραφή προτιμήσεων χρησιμοποιώντας μια λογική με πολλές τιμές αληθείας στην οποία τα διαφορετικά επίπεδα αλήθειας να αντιστοιχούν σε διαφορετικούς βαθμούς προτίμησης. Για να είναι πιο φυσική η περιγραφή, θα πρέπει ο χειρισμός των διαφορετικών επιπέδων αλήθειας να μη γίνεται με απευθείας επεξεργασία των αληθοτιμών, αλλά με τη χρήση τελεστών που προσομοιάζουν με τελεστές προτίμησης που υπάρχουν και στις φυσικές γλώσσες».

Η παραπάνω ιδέα μπορεί να εξηγηθεί καλύτερα με ένα παράδειγμα.

Παράδειγμα 1. *Ας υποθέσουμε ότι θέλουμε να χρησιμοποιήσουμε ένα ηλεκτρονικό σύστημα κράτησης πτήσεων για να πετάξουμε από την Αθήνα στη Βοστώνη. Ας υποθέσουμε επίσης ότι θα θέλαμε να πετάξουμε αν είναι δυνατόν με την αεροπορική εταιρία «Reliable Airlines». Η παραπάνω προτίμηση μπορεί να κωδικοποιηθεί ως εξής:*

```
desired_flight(F) :- from_to(athens,boston,F),
                     and-if-possible carrier(F,reliable_air).
```

Το παραπάνω πρόγραμμα μοιάζει με ένα κλασικό πρόγραμμα λογικού προγραμματισμού, με τη διαφορά ότι χρησιμοποιεί τον τελεστή `and-if-possible`*. Αν μπροστά από ένα άτομο δεν υπάρχει αυτός ο τελεστής, τότε το άτομο εκφράζει μια συνθήκη που πρέπει οπωσδήποτε να πραγματοποιηθεί. Διαφορετικά, το άτομο εκφράζει μια προτίμηση που «καλό θα ήταν να ικανοποιηθεί». Τώρα, ας υποθέσουμε την εκτέλεση ενός ερωτήματος της μορφής:*

```
?- desired_flight(F).
```

Αν το ερώτημα επιτύχει, τότε βρήκαμε μια πτήση από την Αθήνα στη Βοστώνη με την «Reliable Airlines». Εάν το ερώτημα αποτύχει, τότε δεν υπάρχει καμμία πτήση από Αθήνα προς Βοστώνη. Εάν το ερώτημα αποτύχει εν μέρει (αυτό σημαίνει ότι υπολογίζεται σε μια ενδιάμεση τιμή αληθείας), τότε βρέθηκε μια πτήση η οποία όμως δεν είναι με την «Reliable Airlines». Αυτό σημαίνει ότι μπορούμε μεν να πετάξουμε στον προορισμό μας, αλλά όχι με την αεροπορική εταιρία της προτίμησής μας.

Για τη θεμελίωση της παραπάνω προσέγγισης [62, 63], προτείνουμε τη γλώσσα PrefLog, η οποία είναι μια γλώσσα λογικού προγραμματισμού που χρησιμοποιεί τελεστές προτίμησης για την έκφραση προτιμήσεων (όπως π.χ. ο `and-if-possible`). Προσδιορίζουμε ένα σύνολο από ιδιότητες τις οποίες αν τις ικανοποιούν οι τελεστές προτίμησης, τότε η γλώσσα PrefLog θα έχει καλώς ορισμένη σημασιολογία. Εισάγουμε διάφορους τελεστές προτίμησης και δίνουμε πολλά παραδείγματα αναπαράστασης προτιμήσεων. Τέλος, προτείνουμε μια «από-κάτω-προς-τα-πάνω» μέθοδο υλοποίησης που τερματίζει για προγράμματα που ανήκουν ένα καλώς ορισμένο υποσύνολο της PrefLog και αποδεικνύουμε την ορθότητά της.

Ποιοτικές Προτιμήσεις και Λογικός Προγραμματισμός Υψηλής Τάξης

Η κεντρική ιδέα της προσέγγισης που χρησιμοποιεί το λογικό προγραμματισμό υψηλής τάξης για την έκφραση ποιοτικών προτιμήσεων μπορεί να περιγραφεί συνοπτικά ως εξής:

«Αφού οι προτιμήσεις μπορούν να εκφραστούν με τη χρήση σχέσεων και αφού η επεξεργασία των προτιμήσεων μπορεί να γίνει με τη χρήση τελεστών που παίρνουν ως παραμέτρους σχέσεις, θα μπορούσε κανείς να χρησιμοποιήσει μια γλώσσα λογικού προγραμματισμού υψηλής τάξης η οποία υποστηρίζει τόσο τον ορισμό σχέσεων όσο και τον ορισμό τελεστών πάνω σε αυτές».

Η παραπάνω ιδέα μπορεί να εξηγηθεί καλύτερα με ένα παράδειγμα.

Παράδειγμα 2. *Έστω ότι έχουμε μια σχέση με ταινίες* `movie((Name,Genre,Rating))`*. Ας υποθέσουμε ότι θέλουμε να εκφράσουμε την παρακάτω σχέση προτίμησης: «Προτιμώ μια ταινία από μια άλλη αν είναι του ίδιου είδους αλλά η πρώτη έχει υψηλότερη βαθμολογία». Η σχέση αυτή περιγράφεται πολύ εύκολα με το παρακάτω λογικό πρόγραμμα:*

```
c_pref((N1,G,R1), (N2,G,R2)) :-
        movie((N1,G,R1)),
        movie((N2,G,R2)),
        R1 > R2.
```

Το παραπάνω λογικό πρόγραμμα είναι πρώτης τάξης, δε χρησιμοποιεί δηλαδή κάποια χαρακτηριστικά υψηλής τάξης. Αν θέλουμε όμως να ορίσουμε ένα τελεστή που επεξεργάζεται σχέσεις προτίμησης όπως η παραπάνω, χρειαζόμαστε τις σχέσεις υψηλής τάξης. Για παράδειγμα, μπορούμε να ορίσουμε τον παρακάτω τελεστή προτίμησης `winnow(C,R,T)`·

ο τελεστής αυτός επιστρέφει τις καλύτερες πλειάδες `T` *από μία σχέση* `R` *σύμφωνα με μια διμερή σχέση προτίμησης* `C`*. Παρατηρούμε ότι το παρακάτω πρόγραμμα είναι υψηλής τάξης καθώς έχουμε μεταβλητές που εμφανίζονται στη θέση κατηγορημάτων και κατηγορήματα που παίρνουν ως παραμέτρους άλλα κατηγορήματα:*

```
winnow(C,R,T) :- R(T), not bypassed(C,R,T).
bypassed(C,R,T) :- R(Z), C(Z,T).
```

Συνεπώς, το παρακάτω ερώτημα:

```
?- winnow(c_pref,movie,T).
```

θα επιστρέψει τις προτιμότερες ταινίες από την σχέση `movie` *σύμφωνα με τη ζητούμενη σχέση προτίμησης* `c_pref`*.*

Η παραπάνω προσέγγιση [13, 14] ξεπερνάει τα περισσότερα μειονεκτήματα που έχουν οι υπάρχουσες ποιοτικές τεχνικές τόσο στις βάσεις δεδομένων όσο και στο λογικό προγραμματισμό. Η τεχνική αυτή παρέχει ένα ενιαίο πλαίσιο για την περιγραφή ποιοτικών προτιμήσεων, στο οποίο η μοντελοποίηση της βασικής γνώσης, των προτιμήσεων και τον τελεστών που επεξεργάζονται προτιμήσεις γίνεται στην ίδια γλώσσα λογικού προγραμματισμού υψηλής τάξης. Τα προγράμματα αυτά μπορούν να εκτελεστούν σε πραγματικά συστήματα λογικού προγραμματισμού υψηλής τάξης και η απόδοσή τους μπορεί να ενισχυθεί είτε με γενικές είτε με εξειδικευμένες τεχνικές βελτιστοποίησης. Ανάμεσα σε αυτές, προτείνουμε μια νέα τεχνική μετατροπής λογικών προγραμμάτων υψηλής τάξης σε κλασικά λογικά προγράμματα (πρώτης τάξης) [73] και την εφαρμόζουμε στα προγράμματα της προσέγγισής μας. Τέλος, αυτή η ποιοτική προσέγγιση μπορεί να βρει εφαρμογή σε πρακτικά συστήματα επερωτήσεων έξω από το πεδίο του λογικού προγραμματισμού και των σχεσιακών βάσεων δεδομένων [74].

Συνεισφορά στην Επιστημονική Γνώση

Η ερευνητική συνεισφορά της διατριβής συνοψίζεται στα παρακάτω σημεία:

- Προτείνεται η γλώσσα προγραμματισμού PrefLog, η οποία είναι μια ποσοτική επέκταση του λογικού προγραμματισμού για την έκφραση προτιμήσεων. Η γλώσσα αυτή βασίζεται σε ένα άπειρο σύνολο τιμών αληθείας και στοχεύει στην αναπαράσταση ποσοτικών προτιμήσεων με την χρήση κατάλληλων τελεστών προτίμησης. Προσδιορίζεται ένα σύνολο από ιδιότητες τις οποίες πρέπει να ικανοποιούν οι διαθέσιμοι τελεστές προτίμησης έτσι ώστε η γλώσσα να έχει καλώς ορισμένη σημασιολογία. Προτείνονται διάφοροι τελεστές προτίμησης και δίνονται πολλά παραδείγματα αναπαράστασης προτιμήσεων.

- Εισάγεται μια «από-κάτω-προς-τα-πάνω» τεχνική υλοποίησης για ένα καλώς ορισμένο υποσύνολο της PrefLog, που αντιστοιχεί στο προτιμησιακό αντίστοιχο της γλώσσας βάσεων δεδομένων Datalog. Αποδεικνύεται η ορθότητα της τεχνικής αυτής, καθώς και το γεγονός ότι η διαδικασία υπολογισμού τερματίζει για κάθε πρόγραμμα που ανήκει στο συγκεκριμένο υποσύνολο της γλώσσας. Αξίζει να σημειωθεί ότι (αντίθετα με ότι συμβαίνει στην Datalog) η εξασφάλιση της ιδιότητας του τερματισμού μιας τέτοιας στρατηγικής δεν είναι προφανής. Το σύνολο των τιμών αληθείας είναι άπειρο, οπότε μια αφελής στρατηγική θα χρειαζόταν άπειρα βήματα να λειτουργήσει ακόμα και για πεπερασμένα προτασιακά PrefLog προγράμματα.

- Προτείνεται η χρήση του λογικού προγραμματισμού υψηλής τάξης για την αναπαράσταση και την επεξεργασία ποιοτικών προτιμήσεων. Η κεντρική ιδέα της προσέγγισης αυτής βασίζεται στην παρατήρηση ότι εφόσον (όπως αναφέραμε νωρίτερα) οι ποιοτικές προτιμήσεις εκφράζονται ως διμερείς σχέσεις, άρα η επεξεργασία προτιμήσεων γίνεται με διαδικασίες που λαμβάνουν ως ορίσματα σχέσεις. Συνεπώς η χρήση μιας γλώσσας υψηλής τάξης (η οποία υποστηρίζει τόσο τον ορισμό σχέσεων όσο και τον ορισμό τελεστών πάνω σε αυτές) προσφέρει μια ενιαία και κομψή αναπαράσταση. Επιπλέον, η φαινομενικά πιο απαιτητική περίπτωση των προτιμήσεων ανάμεσα σε σύνολα, μπορεί να αναπαρασταθεί εξίσου εύκολα, καθώς οι προτιμήσεις μεταξύ συνόλων είναι ουσιαστικά σχέσεις δεύτερης τάξης και επομένως μπορούν να κωδικοποιηθούν εύκολα σε μια γλώσσα υψηλής τάξης.

- Η ιδέα της χρήσης κατηγορημάτων υψηλής τάξης για την αποτελεσματική αναπαράσταση προτιμήσεων, υλοποιήθηκε με τη χρήση της γλώσσας λογικού προγραμματισμού υψηλής τάξης HiLog στο σύστημα XSB. Επίσης, χρησιμοποιούνται εξειδικευμένες τεχνικές βελτιστοποίησης λογικών προγραμμάτων υψηλής τάξης που μοντελοποιούν ποιοτικές προτιμήσεις τόσο ανάμεσα σε απλά στοιχεία όσο και σε σύνολα. Ανάμεσα στις τεχνικές αυτές, προτείνεται μια νέα τεχνική μετατροπής λογικών προγραμμάτων υψηλής τάξης σε κλασικά λογικά προγράμματα πρώτης τάξης. Η τεχνική αυτή εφαρμόζεται σε ένα σαφώς ορισμένο υποσύνολο του λογικού προγραμματισμού υψηλής τάξης (που εμπεριέχει προγράμματα που εκφράζουν ποιοτικές προτιμήσεις). Η αποτελεσματικότητα των τεχνικών αυτών τεκμηριώνεται και πειραματικά. Ο κώδικας της υλοποίησης καθώς και τα πειράματα είναι διαθέσιμα σε δημόσια αποθετήρια.

CONTENTS

PREFACE

This dissertation is submitted for the degree of Doctor of Philosophy at the National and Kapodistrian University of Athens, Greece. The research described herein was conducted under the guidance and supervision of Professor Panos Rondogiannis in the Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece.

Part of work has been presented in a series of international conferences and journal articles, as detailed in the section titled “List of Publications”.

This work is at the best of my knowledge original, except where acknowledgments and references are made to previous works. Hereby I declare that this doctoral dissertation, my original investigation and achievement, has not been submitted for any other academic degree, diploma or other qualification at any other university.

Antonis Troumpoukis

June 2019

1. INTRODUCTION

The purpose of this dissertation is to use *extensions of logic programming* in order to express preferences. The effective representation of preferences is crucial in many scientific disciplines and it can be proven useful in many real-world applications. We propose two approaches for expressing preferences. The first approach uses *infinite-valued* logic programming for expressing *quantitative* preferences. This language is based on an infinite set of truth values in order to support operators for expressing preferences. The second approach uses *higher-order* logic programming for expressing *qualitative* preferences. In this approach, preference relations and operations on preferences are expressed in the same, higher-order language. We argue that infinite-valued logic programming and higher-order logic programming are two very expressive frameworks for representing and manipulating preferences.

1.1 Motivation

Preferences play a major role in human life. They can affect us in many situations; from simple personal choices in early childhood (e.g., “which ice-cream flavor do you prefer?”) up to complex professional decisions (e.g., “should I pursue a career in music or in computer science?”). Therefore, it comes as no surprise that preferences have been explored in many scientific disciplines (such as philosophy [37], economics [28], and psychology [49]). Research in preferences is very active in Computer Science, mostly in areas such as Artificial Intelligence [24], Database Systems [69], and Programming Languages [22]. One of the main objectives of the study of preferences in Computer Science is the design of languages and frameworks that can provide us with the ability to choose among alternatives in a declarative way, whether these alternatives are problem solutions, program answers, or query results. Effective user preference representation formalisms can be applied in information systems so that the responses presented to the users can be more compact and comprehensive because it can reflect their true interests.

Preference representation formalisms usually fall into two basic categories [69]. In the *quantitative* approach [1, 2, 3, 45], preferences are represented by a preference value function. Each object is associated with a *preference score*, which is a numerical value that expresses the degree of interest (e.g., “my preference in beer is 0.9 while in wine it is 0.2”). In the *qualitative* approach [17, 19, 32, 36, 80] preferences are expressed by direct comparisons between objects (e.g., “I prefer beer over wine”), thus resulting in a *binary preference relation*. Each category has its strengths and its weaknesses. The qualitative approach is more general than the quantitative approach (i.e., not all preference relations can be expressed by scoring functions or through degrees of interest [17, 69]). On the other hand, quantitative preferences can distinguish how much preferred one object is over another (e.g., a preference score of 0.9 is much more preferred than a score of 0.001, but is a little more preferred than a score of 0.899).

As a general observation, we could say that both qualitative and quantitative formalisms that have been developed leave much room for improvement both in terms of expressiveness and efficiency. Quantitative approaches usually rely on the definition of a preference function. However, a preference function cannot always be defined, and if it can, users in most cases are rarely willing to express their preferences directly in terms of such a function. Qualitative approaches have their weaknesses too; first, they usually offer a quite limited set of preference operations—in most cases, only one preference opera-

tor can be used (namely, "find the most preferred objects according to this preference"); second, almost all approaches use two distinct languages, one for representing the base knowledge and one for representing the preferences, making the structure of the representation non-uniform. Moreover, qualitative approaches rely on structures such as *partial order relations*, which are more complex than simple numerical values; therefore, the process of handling a qualitative preference is clearly a more complex task than that of a quantitative preference. As a result, the development of optimization techniques for enhancing the performance of qualitative preference frameworks can be of crucial importance from a practical point of view.

The purpose of this dissertation is to study new, more expressive formalisms for representing and manipulating preferences. In particular, we use two extensions of logic programming:

- The first approach uses infinite-valued logic programming for expressing quantitative preferences. This language is based on an infinite set of truth values in order to support operators for expressing preferences.
- The second approach uses higher-order logic programming for expressing qualitative preferences. In this approach, preference relations and operations on preferences are expressed in the same, higher-order language.

Our approaches attempt to overcome the shortcomings that were mentioned previously. In our quantitative approach, the preference values are not denoted directly but are expressed using appropriate preference operators. In our qualitative approach, base relations, preferences, and operations on preferences are represented using the same language making our approach more uniform. In addition, our framework allows the definition of many preference operators other than those that are offered in most qualitative approaches in the literature.

1.2 Our Approaches

In this section, we present a high-level description of our approaches for extending logic programming for expressing preferences. The central idea behind each one of our approaches is illustrated with a simple motivating example.

1.2.1 The infinite-valued approach

The central idea of the infinite-valued approach can be summarized as follows: *We can represent quantitative preferences with an infinite set of truth values, such that the different levels of truth correspond to different degrees of preference. In order for the description to be more natural though, the manipulation of the different levels of preference should not be done by processing the truth values directly, but with the use of operators that resemble preference operations that appear in natural languages*. The above idea can be illustrated in the following example:

Example 1.1. *Suppose that we are using an online flight reservation system in order to fly from Athens to Boston. Assume that we want to book a flight ticket from Athens to Boston, flying if possible with "Reliable Airlines". This fact can be encoded as follows:*

```
desired_flight(F) :- from_to(athens,boston,F),
                     and-if-possible carrier(F,reliable_air).
```

The above program looks like a classic logic program, with the difference that it uses the `and-if-possible` *operator. If this operator is not present then the atom expresses a necessary condition; otherwise, the atom expresses an optional condition that it "would be preferable but not necessary, to be satisfied". Now, suppose that we issue a query of the form:*

```
?- desired_flight(F).
```

If the query succeeds then we found a flight from Athens to Boston with Reliable Airlines. If the query completely fails then there does not exist any flights from Athens to Boston. If the query partially fails (meaning that it is evaluated into an intermediate truth value), then a flight has been found which however is not with Reliable Airlines. This means that we can fly to our destination, but not traveling with the carrier of our preference.

In order to formulate this approach [62, 63] we introduce the logic programming language PrefLog, its syntax, and its semantics; in particular, we study the properties of the PrefLog operators that are needed in order for the PrefLog programs to behave well from a semantic point of view. In addition, we introduce a bottom-up evaluation method for a well-defined class of function-free PrefLog programs.

1.2.2 The higher-order approach

The central idea of the higher-order approach can be summarized as follows: *Since qualitative preferences can be expressed using binary preference relations, and since operations on preferences involve operations that take preference relations as arguments, a higher-order language can offer increased representation capabilities*. The above idea can be illustrated in the following example:

Example 1.2. *Suppose that we have a relation of movies* `movie((Name,Genre,Rating))`*. Now, suppose that we want to express the following preference relation: "Prefer one movie over another iff their genres are the same and the rating of the first is higher". This preference relation can be encoded easily using the following logic program:*

```
c_pref((N1,G,R1), (N2,G,R2)) :-
        movie((N1,G,R1)),
        movie((N2,G,R2)),
        R1 > R2.
```

The above program is a first-order one, so it does not use any higher-order features. However, if we want to define an operator that processes preference relations such as the above, we need to define higher-order predicates. For example, the following operator `winnow(C,R,T)` *returns the best tuples* `T` *from a relation* `R` *according to a binary preference relation* `C`*:*

```
winnow(C,R,T) :- R(T), not bypassed(C,R,T).
bypassed(C,R,T) :- R(Z), C(Z,T).
```

As a result, the following query

```
?- winnow(c_pref,movie,T).
```

will return the most preferred movies from the relation `movie` *using the preference relation* `c_pref` *of our interest.*

The higher-order approach [13, 14], extends a seminal work by Chomicki [17, 80] and it goes beyond most disadvantages of existing qualitative techniques both in databases and in logic programming. The use of higher-order logic programming provides a uniform framework in which relations, preferences between tuples, preferences between sets of tuples and operations on preferences are expressed in the same, higher-order logic programming language. The programs can be evaluated by standard higher-order programming systems, and their performance can be enhanced with generic and specialized optimization techniques. Among these techniques, we propose a novel program transformation technique for translating higher-order programs into first-order ones and we used this technique for optimizing the higher-order programs of our interest [73]. Finally, as we have recently demonstrated [74] this qualitative approach can be used for implementing practical query systems outside the realm of logic programming and relational databases.

1.3 Contributions

Our contributions can be summarized as follows:

- We argue that the adoption of many-valued logics is a promising idea for developing expressive new preferential logic programming languages. We define the simple preferential logic programming language PrefLog which differs from other preferential logic programming approaches in that it uses an underlying infinite-valued truth domain in order to support quantitative preference operators. We show that the continuity of these operators over the infinite-valued underlying domain ensures that the resulting logic programming system retains all the standard and well-known properties of classical logic programming (and most notably the existence of a least Herbrand model).
- We demonstrate that terminating bottom-up evaluation can be performed for a large function-free fragment of PrefLog. This result is not obvious: despite the fact that the Herbrand Base of the programs we consider is finite, an atom may obtain an infinity of truth values during a bottom-up evaluation, resulting in possible non-termination.
- We argue that higher-order logic programming is a very expressive framework for representing and manipulating qualitative preferences. A significant advantage of our approach is that preference formulas as-well-as operators that are parameterized with such formulas can be expressed in the same language. Moreover, the seemingly more demanding case of preferences over sets can be handled without extra notational overhead, because preferences over sets are essentially second-order relations and can, therefore, be encoded easily in our higher-order language.
- We implement specialized techniques that can enhance higher-order logic programming so that it can better handle and manipulate preferences. We propose Predicate Specialization, a transformation technique based on the abstract framework of *Partial Evaluation*. This technique is used for optimizing higher-order logic programs that express preferences over tuples by transforming them into first-order ones. Moreover, we implement two custom-tailored implementation strategies for optimizing set-preference higher-order programs. Finally, we provide experimental results that suggest that the proposed techniques can enhance the performance of our higher-order framework.

1.4 Outline

The rest of the dissertation is structured as follows:

Chapter 2 introduces the PrefLog language, a logic programming language based on an infinite-valued domain in order to support operators for expressing preferences. In this chapter, we present the syntax and the semantics of the language and we demonstrate that if the operators used are monotonic and continuous over the infinite-valued underlying domain, then every PrefLog program has a minimum infinite-valued model.

Chapter 3 introduces a terminating bottom-up evaluation method for a well-defined class of function-free PrefLog programs. Ensuring termination is not a straightforward task, because the underlying truth domain of PrefLog and the set of all possible interpretations of a function-free PrefLog program are both infinite.

Chapter 4 proposes the use of higher-order logic programming as a logical framework for expressing qualitative preferences. This approach extends a seminal work by Chomicki [17, 80] and provides a uniform framework in which relations, preferences between tuples, preferences between sets of tuples and operations on preferences are expressed in the same, higher-order logic programming language.

Chapter 5 undertakes a HiLog implementation of our higher-order preferential framework. Apart from a basic, unoptimized implementation, we consider several optimization techniques for enhancing its performance. Among the techniques that we used, we propose Predicate Specialization, which is a transformation technique for optimizing HiLog programs that express preferences over tuples.

Chapter 6 presents experimental results that suggest the feasibility of the higher-order logic programming framework of Chapter 4 and the effectiveness of the optimization techniques presented in Chapter 5, especially when combined with standard logic programming optimizations, such as *tabling*.

Chapter 7 discusses related work regarding preference representation formalisms in the areas of databases and logic programming. We discuss both quantitative and qualitative approaches, and we compare them with our work.

Chapter 8 concludes the dissertation with a summary and a discussion of possible future research directions.

2. EXPRESSING PREFERENCES USING INFINITE-VALUED LOGIC PROGRAMMING

In this chapter, we introduce the logic programming language PrefLog. This language is based on an infinite-valued logic in order to support operators for expressing preferences. We demonstrate that if the operators used are monotonic and continuous over the infinite-valued underlying domain, then the resulting logic programming language retains the well-known properties of classical logic programming, such as the existence of a unique minimum Herbrand model.

2.1 Overview

In this section, we present the basic ideas of the PrefLog language. Our starting point is the *infinite-valued approach* [64]. In this work, Rondogiannis and Wadge proposed an infinite-valued logic, which is used in order to provide a purely model-theoretic semantics for logic programming with negation-as-failure. This logic, apart from the standard true value, denoted by T_0, also uses the truth values $T_1, T_2, \ldots$ that are less and less "true" than T_0; moreover, apart from the standard false value, denoted by F_0, it also uses the values $F_1, F_2, \ldots$ that are less and less "false" than F_0. In the middle between the false and the true values, there exists a "neutral" truth value denoted by 0. More formally:

$$F_0 < F_1 < F_2 < \cdots < 0 < \cdots < T_2 < T_1 < T_0$$

The logic that was developed is a propositional one and uses all standard logical connectives (namely conjunction, disjunction, implication, and negation).

As it was demonstrated by Rondogiannis and Wadge [64], *negation in logic programming can be considered as a preference operator*. In particular, if the truth value of an atom has been obtained with one use of the negation-as-failure rule, then its truth value should be considered weaker (i.e., having a *lower preference*) than the truth value of an atom which has been obtained without negation-as-failure. This preferential view of negation allowed Rondogiannis and Wadge to obtain a minimum-model result for logic programs with negation which extends the classical least-model theorem [50] of negation-less logic programs.

Of particular interest is the propositional query language by Agarwal and Wadge [1, 2] where this underlying infinite-valued logic is used in order to express preferential queries. In particular, Agarwal and Wadge considered a fragment of the aforementioned logic with only conjunction and disjunction but allowed the use of two extra operators that can be used to express preferences. The two operators, denoted throughout the chapter[1] by opt and alt, can express preferences of the form "A and optionally B" and "A or alternatively B".

The simple and elegant language of Agarwal and Wadge [1, 2] lacks in the following respects. First, it is a propositional one and does not allow first-order properties to be expressed. Second, it lacks recursion and therefore it can not express preferences regarding situations where no a-priori knowledge of the "depth" of the data is known (such as for example in the case of finding the best path between two vertices of a graph when various types of preferred connections are available in the graph). Finally, the set of

[1] In the query system of Agarwal and Wadge [1] the two operators are denoted by μ and ω respectively.

operators used, namely $\{\text{opt}, \text{alt}\}$, is quite restricted, and no indication is given of what constitutes a "proper" or "well-behaved" preference operator.

We remedy the above issues by proposing PrefLog, a logic programming language that supports various preference operators and has a clean and simple semantics. In the rest of this section, we motivate our proposal with a simple example that uses the original opt and alt operators [1, 2].

Example 2.1. *Assume we are using an online flight reservation system in order to fly from Athens to Boston. Suppose that we want to book a flight ticket from Athens to Boston, flying if possible with "Reliable Airlines". A query of this form will look like this:*

$$\leftarrow \texttt{from\_to}(\texttt{athens}, \texttt{boston}, \texttt{F}) \wedge \text{opt}\ \texttt{carrier}(\texttt{F}, \texttt{reliable\_air}).$$

A query of this form can get three possible answers: "yes", "no, because there is no flight available from Athens to Boston" and "no, because, although there exists a flight from Athens to Boston, this is not with "Reliable Airlines". Intuitively, the second "no" answer is a much less severe one than the first "no" answer (because it implies we can fly from Athens to Boston but not with the carrier of our preference, which is an optional requirement).

Notice that in the above example we have a situation in which we want to express a conjunction of a compulsory and an optional requirement. There exist situations where we want to express a disjunction involving preferences. These types of disjunctions will involve primary requirements and back-up requirements. Continuing the above example, assume that we also require to have a stopover in our flight (and this is also a compulsory requirement for us). Actually, we want to have a stopover in Rome or alternatively (but with smaller preference), a stopover in London. Hence, we have the query:

$$\leftarrow \texttt{stopover}(\texttt{F}, \texttt{rome}) \vee \text{opt}\ \texttt{stopover}(\texttt{F}, \texttt{london}).$$

A requirement of this type can have three possible answers: "no" (meaning that there is no flight that has a stopover), "yes" (meaning that there actually exists a flight with a stopover in Rome), and another weaker "yes" answer (meaning that there exists a flight with a stopover in London).

These observations suggest that we could use the aforementioned truth domain that contains various "true" and "false" values in order to express various levels of preferences. Moreover, we could allow preference operators to appear in the bodies of logic programming rules in order for the programmer to be able to specify preferences in a declarative manner.

The following program captures the above flight example:

$$\begin{array}{lcl} \texttt{desired\_flight}(\texttt{F}) & \leftarrow & \texttt{from\_to}(\texttt{athens}, \texttt{boston}, \texttt{F}) \wedge \\ & & \texttt{has\_stopover}(\texttt{F}) \wedge \\ & & \text{opt}\ \texttt{carrier}(\texttt{F}, \texttt{reliable\_air}). \\ \\ \texttt{has\_stopover}(\texttt{F}) & \leftarrow & \texttt{stopover}(\texttt{F}, \texttt{rome}) \vee \\ & & \text{alt}\ \texttt{stopover}(\texttt{F}, \texttt{london}). \end{array}$$

The informal meaning of the unary operators opt *and* alt *is "optionally" and "alternatively" respectively; their formal meaning will be described in the following section. Returning*

to the above example, if a query of the form $\leftarrow \texttt{desired\_flight}(\texttt{F})$ *completely fails then, either there does not exist a flight from Athens to Boston, or there does not exist such a flight that has stopovers; if the query partially fails, then a flight has been found which however is not with the desired carrier; if the query partially succeeds then there exists a flight which has a stopover in London (but not Rome); finally, the complete success of our query indicates that the flight found satisfies all our requirements (namely it flies from Athens to Boston and has a stopover in Rome).*

Notice that the above operators can be iterated, expressing in this way more than two levels of preference. For example, the clause:

$$\begin{array}{lll} \texttt{has\_stopover}(\texttt{F}) & \leftarrow & \texttt{stopover}(\texttt{F}, \texttt{rome}) \vee \\ & & \text{alt} \quad \texttt{stopover}(\texttt{F}, \texttt{london}) \vee \\ & & \text{alt}^2 \quad \texttt{stopover}(\texttt{F}, \texttt{frankfurt}). \end{array}$$

expresses that we prefer Rome over London and London over Frankfurt.

The rest of the chapter is organized as follows: in Section 2.2, we present the syntax and some basic semantic concepts behind the proposed language PrefLog; in Section 2.3 we present the fixed-point semantics of PrefLog; in Section 2.4 we describe a simple approach for building new continuous operators and we demonstrate their use in example programs; in Section 2.5 we discuss the expressive power of PrefLog programs; and finally, we close with a brief summary of the chapter.

2.2 The Logic Programming Language PrefLog

In this section, we introduce the syntax and the basic semantic notions regarding the new preferential logic programming language PrefLog. In a nutshell, our language extends classical logic programming with preference operators, whose semantic meanings are functions, whose domain and range is the infinite set of truth values. Throughout this chapter, we assume that the reader is familiar with the basic concepts and terminology of logic programming [50].

2.2.1 Syntax

In this subsection, we define the syntax of PrefLog. We begin by extending the usual set of symbols in logic programming with a set of preference operators. Then, we continue with the definition of a PrefLog program and we close with some remarks regarding the Herbrand universe and Herbrand base of PrefLog programs.

We assume the existence of a set *Op* of *preference operators*; programs of our language use operators of this set in the bodies of rules. We use the symbol ∇ to denote an arbitrary preference operator. Each operator ∇ has a fixed *arity* $n \in \mathbb{N}$; we denote this fact by writing ∇/n. For simplicity, we assume that the usual *conjunction* and *disjunction* operators of logic programs belong to *Op*; we denote conjunction by $\wedge$ and disjunction by $\vee$.

Since PrefLog is essentially an extension of first-order logic programming with preference operators in the bodies of the rules, we need to extend the usual logic programming syntax of the rule bodies:

Definition 2.1. *Body formulas are inductively defined as follows:*

- *The constant* true *is a body formula.*
- *If* A *is an atomic formula, then* A *is a body formula.*
- *If* ∇/n*, where* $n \geq 1$*, is a preference operator, and* $\mathbf{A}_1, \ldots, \mathbf{A}_n$ *are body formulas, then* $\nabla(\mathbf{A}_1, \ldots, \mathbf{A}_n)$ *is a body formula.*

When applying an operator to an expression, we often omit the parentheses when this creates no confusion (as we have done for opt and alt in Example 2.1). We continue with the definition of a PrefLog program:

Definition 2.2. *A PrefLog rule is the universal closure of a formula of the form* A $\leftarrow$ **B** *where* A *is an atom and* **B** *is a body formula. A PrefLog rule of the form* A $\leftarrow$ true *is called a PrefLog fact. A PrefLog program is a finite set of PrefLog rules.*

A PrefLog fact can be written in the form A $\leftarrow$ instead of the full form A $\leftarrow$ true. Notice that the program given in Example 2.1 is a valid PrefLog one (the conjunction and disjunction operators are used in their usual infix form instead of the prefix one suggested by Definition 2.1).

The *Herbrand universe* U_{P} and the *Herbrand base* B_{P} of a PrefLog program P are defined as in classical logic programming [50]. In the following, in order to avoid unnecessary technicalities in the specification of the semantics of PrefLog programs, we make a simplifying assumption that is common in the logic programming literature: instead of studying first-order PrefLog programs, we will study their ground instantiations. The *ground instantiation* of a program can be obtained by replacing the variables in rules with terms from the Herbrand universe in all possible ways. Essentially, the ground instantiation is a (possibly infinite) propositional program. In the rest of the chapter, when we refer to a *program* we will mean (unless otherwise stated) *the ground instantiation of a program*. However, in examples, we will use the more compact (and more human-readable) first-order form of a PrefLog program.

2.2.2 Infinite-Valued Models

In this subsection, we define infinite-valued interpretations and infinite-valued models of PrefLog programs. In order to do so, we must first provide the definitions of the set of truth values and the semantic meaning of the preference operators.

As mentioned earlier, the logic underlying PrefLog is infinite-valued.

Definition 2.3. *The underlying set of truth values of PrefLog is the set*

$$\mathbb{V} = \{F_i : i \in \mathbb{N}\} \cup \{0\} \cup \{T_i : i \in \mathbb{N}\},$$

with the following ordering:

$$F_0 < F_1 < F_2 < \cdots < 0 < \cdots < T_2 < T_1 < T_0.$$

Moreover, a notion that will prove useful in the following is that of the order of a given truth value:

Definition 2.4. *The order of a truth value is defined as:*

$$\mathsf{ord}(v) = \begin{cases} n, & \textit{if } v = T_n \textit{ or } v = F_n \\ +\infty, & \textit{if } v = 0 \end{cases}$$

We can now define Herbrand interpretations in the context of PrefLog (in the rest of the chapter, the term *interpretation* will mean an infinite-valued Herbrand interpretation):

Definition 2.5. *An infinite-valued Herbrand interpretation I of a PrefLog program* P *is a function $I : B_{\mathsf{P}} \to \mathbb{V}$.*

As a special case of interpretation, we write $\emptyset$ to denote the interpretation that assigns the F_0 value to all members of B_{P}. Notice that infinite-valued interpretations extend the classical Herbrand interpretations. This is because a classical Herbrand interpretation of a classical logic program P is defined as a subset of B_{P}, or equivalently as a function $B_{\mathsf{P}} \to \{\text{true}, \text{false}\}$.

The meaning of preference operators is specified as follows:

Definition 2.6. *Let ∇/n be a preference operator. The denotation of ∇ is a function $\|\nabla\| : \mathbb{V}^n \to \mathbb{V}$.*

We will provide examples of denotations of some preference operators in the next section. Now, using the above definition, we can extend the notion of interpretation in order to apply to body formulas:

Definition 2.7. *Let I be an interpretation of a PrefLog program* P*. Then, I can be extended to apply to ground body formulas as follows:*

- $I(\text{true}) = T_0$.
- *For every preference operator ∇/n and for all body formulas $\mathbf{A}_1, \ldots, \mathbf{A}_n$, we define $I(\nabla(\mathbf{A}_1, \ldots, \mathbf{A}_n)) = \|\nabla\|(I(\mathbf{A}_1), \ldots, I(\mathbf{A}_n))$.*

Finally, by using the above definitions, we can now define the notion of a model of a PrefLog program:

Definition 2.8. *Let I be an interpretation of a PrefLog program* P*. Then, I satisfies the rule* A $\leftarrow$ **B** *of* P *if $I(\mathrm{A}) \geq I(\mathbf{B})$. Moreover, I is a model of* P *if I satisfies all rules of* P*.*

2.2.3 Examples of PrefLog Operators

In this subsection, we present some examples regarding the notions that have been introduced in the previous two subsections. We introduce some simple preference operators and then we show the intuition behind their denotations using the example of Section 2.1.

We start by presenting the denotations of some basic operators on $\mathbb{V}$. That is logical conjunction and disjunction, as well as a basic operator that is used for changing the order of a truth value:

Definition 2.9. *The denotations of $\wedge$ and $\vee$ are $\|\wedge\| = \min$ and $\|\vee\| = \max$, respectively.*

Definition 2.10. *Let $v \in \mathbb{V}$ and $n \in \mathbb{N}$. Then, ϵ^n is a unary preference operator, whose denotation is the following:*

$$\|\epsilon^n\|(v) = \begin{cases} F_{k+n}, & \textit{if } v = F_k \\ 0, & \textit{if } v = 0 \\ T_{k+n}, & \textit{if } v = T_k. \end{cases}$$

Notice how the denotations of $\wedge$ and $\vee$ actually extend the usual semantic meaning in classical logic. In addition, notice that $\|\epsilon^0\|$ is the identity function on $\mathbb{V}$. Moreover, we usually write ϵ instead of ϵ^1. As we are going to see, the above operators can be used in order to define more interesting ones.

The two other families of preference operators that were introduced by Agarwal and Wadge in their preference query language [1, 2] are opt^n and alt^n, $n \in \mathbb{N}$. The semantics of these operators can be specified as follows:

Definition 2.11. *Let $v \in \mathbb{V}$ and $n \in \mathbb{N}$. Then,* opt^n *and* alt^n *are unary preference operators, whose denotations are the following:*

$$\|\mathsf{opt}^n\|(v) = \begin{cases} F_{k+n}, & \textit{if } v = F_k \\ 0, & \textit{if } v = 0 \\ T_k, & \textit{if } v = T_k \end{cases} \qquad \|\mathsf{alt}^n\|(v) = \begin{cases} F_k, & \textit{if } v = F_k \\ 0, & \textit{if } v = 0 \\ T_{k+n}, & \textit{if } v = T_k. \end{cases}$$

Notice that if v is true, then $\|\mathsf{opt}\|(v)$ is as true as v, but if v is false, then $\|\mathsf{opt}\|(v)$ is less false than v. This behavior is what we expect from an optional requirement, because the absence of an optional requirement does not annoy us as much as the absence of a compulsory requirement. Also, if v is true, then $\|\mathsf{alt}\|(v)$ is less true than v, but if v is false, then $\|\mathsf{alt}\|(v)$ is as false as v. Again, this is what we expect from an alternative (and not primary) option, because the presence of an alternative option is not as beneficial to us as the presence of a primary option. Notice that, as in the case of ϵ^1, we usually write opt and alt instead of opt^1 and alt^1, respectively.

To see the use of the above operators in the PrefLog framework, we refine Example 2.1 by adding some facts.

Example 2.2. *Consider again the flight example with added facts regarding three particular flights:*

$$\begin{array}{lcl}
\texttt{desired_flight(F)} & \leftarrow & \texttt{from_to(athens, boston, F)} \wedge \\
 & & \texttt{has_stopover(F)} \wedge \\
 & & \textsf{opt}\ \texttt{carrier(F, reliable_air)}. \\
 & & \\
\texttt{has_stopover(F)} & \leftarrow & \texttt{stopover(F, rome)} \vee \\
 & & \textsf{alt}\ \texttt{stopover(F, london)}. \\
 & & \\
\texttt{from_to(athens, boston, fl1)}. & & \\
\texttt{from_to(athens, boston, fl2)}. & & \\
\texttt{from_to(athens, boston, fl3)}. & & \\
 & & \\
\texttt{stopover(fl1, rome)}. & & \\
\texttt{stopover(fl2, london)}. & & \\
\texttt{stopover(fl3, rome)}. & & \\
 & & \\
\texttt{carrier(fl1, delay_air)}. & & \\
\texttt{carrier(fl2, reliable_air)}. & & \\
\texttt{carrier(fl3, reliable_air)}. & &
\end{array}$$

Notice that only flight `fl3` *satisfies fully all our requirements. The other two flights are not fully satisfactory. More specifically, flight* `fl1` *is not with "Reliable Airlines" and flight*

`fl2` has a stopover in London (and not Rome). One can easily verify that one interpretation that is a model of the above program is one in which `desired_flight(fl1)` *has the value* F_1*,* `desired_flight(fl2)` *has the value* T_1 *and* `desired_flight(fl3)` *has the value* T_0 *(we omit listing the truth values of all ground atoms for briefness). As a result, it is preferable to take flight* `fl3` *over* `fl2` *and* `fl1` *for this journey. In the next section, the semantics of PrefLog will be specified in detail and it will become apparent that this particular interpretation is the least model of the above program.*

2.3 The Fixed-Point Semantics of PrefLog

In this section, we show that if the preference operators we adopt obey certain simple properties, then the programs of our language are guaranteed to be well-behaved from a semantic point of view. In particular, we show that the usual fixed-point semantics of classical Logic Programming can be extended to PrefLog, provided that the preference operators that occur in the bodies of the clauses are monotonic and continuous.

We start by defining the usual (pointwise) ordering on n-tuples of elements of $\mathbb{V}$:

Definition 2.12. *Let* $\mathbf{x}, \mathbf{y} \in \mathbb{V}^n$*,* $n \geq 1$ *where* $\mathbf{x} = (x_1, \ldots, x_n)$ *and* $\mathbf{y} = (y_1, \ldots, y_n)$*. We write* $\mathbf{x} \leq \mathbf{y}$ *if* $x_i \leq y_i$*, for all* $1 \leq i \leq n$*.*

It is easy to verify that the set $\mathbb{V}^n$ with respect to the above pointwise ordering is a complete lattice for every $n \geq 1$. This fact is actually a straightforward extension of a similar property of the set $\mathbb{V}$ [64][Lemma 5.2]. In the following definitions, lub is the usual shorthand for "least upper bound".

Proposition 2.1. *Let* $n \geq 1$*. Then,* $(\mathbb{V}^n, \leq)$ *is a complete lattice. In particular, for every* $S \subseteq \mathbb{V}^n$ *it holds that* $\text{lub}(S) = (\text{lub}\{x_1 : (x_1, \ldots, x_n) \in S\}, \ldots, \text{lub}\{x_n : (x_1, \ldots, x_n) \in S\})$*. Moreover, the top (resp. bottom) element of the lattice is the element* $(x_1, \ldots, x_n)$*, where* $x_i = T_0$ *(resp.* $x_i = F_0$*) for every* $1 \leq i \leq n$*.*

Now, we can define the notions of monotonicity and continuity for preference operators:

Definition 2.13. *Let* ∇/n *be a preference operator. Then,* $\|\nabla\|$ *is called monotonic if for all* $\mathbf{x}, \mathbf{y} \in \mathbb{V}^n$ *it holds* $\mathbf{x} \leq \mathbf{y} \implies \|\nabla\|(\mathbf{x}) \leq \|\nabla\|(\mathbf{y})$*.*

Definition 2.14. *Let* ∇/n *be a preference operator. Then,* $\|\nabla\|$ *is called continuous if it is monotonic and for all sequences* $(\mathbf{x}_n)_{n \geq 0}$ *of elements of* $\mathbb{V}^n$ *such that for all* $n \geq 0$*,* $\mathbf{x}_n \leq \mathbf{x}_{n+1}$ *it holds* $\|\nabla\|(\text{lub}(\{\mathbf{x}_n : n \geq 0\})) = \text{lub}\{\|\nabla\|(\mathbf{x}_n) : \mathbf{x}_n \geq 0\}$*.*

By abuse of language, we will often say that "the operator ∇ is monotonic (resp. continuous)" instead of the more accurate "the denotation of the operator ∇ is monotonic (resp. continuous)".

It is easy to verify that every continuous operator is also monotonic. On the other hand, not every monotonic operator is continuous. This is illustrated in the example that follows:

Example 2.3. *Consider a preference operator whose denotation is the following function:*

$$\delta(v) = \begin{cases} F_0, & \textit{if } v < 0 \\ T_0, & \textit{if } v \geq 0. \end{cases}$$

The operator is monotonic, because if $x \leq y < 0$ *or* $0 \leq x \leq y$*, then* $\delta(x) = \delta(y)$*; moreover, if* $x < 0 \leq y$*, then* $\delta(x) = F_0$ *and* $\delta(y) = T_0$*. However, the operator is not continuous. Consider the set of all false values;* $\delta(\mathsf{lub}\{F_0, F_1, \dots\}) = \delta(0) = T_0$ *and* $\mathsf{lub}\{\delta(F_0), \delta(F_1), \dots\} = F_0$*.*

In order to give fixed-point semantics to PrefLog, we need to define a partial order relation on the set of interpretations:

Definition 2.15. *Let* I, J *be interpretations of a PrefLog program* P*. We write* $I \leq J$ *if* $I(\mathsf{A}) \leq J(\mathsf{A})$ *for all* $\mathsf{A} \in B_{\mathsf{P}}$*.*

It is not hard to see that the above relation $\leq$ on interpretations is a partial order (i.e., it is reflexive, transitive and antisymmetric). Given a program P, we denote by $\mathcal{I}_{\mathsf{P}}$ the set of all interpretations of P. It is easy to verify that $\mathcal{I}_{\mathsf{P}}$ is a complete lattice under the above pointwise relation $\leq$.

Proposition 2.2. *Let* P *be a PrefLog program. Then,* $(\mathcal{I}_{\mathsf{P}}, \leq)$ *is a complete lattice. In particular, for every* $\mathcal{X} \subseteq \mathcal{I}_{\mathsf{P}}$ *and* $\mathsf{A} \in B_{\mathsf{P}}$*,* $\mathsf{lub}(\mathcal{X})(\mathsf{A}) = \mathsf{lub}\{I(\mathsf{A}) : I \in \mathcal{X}\}$*. Moreover, the top (resp. bottom) element of the lattice is the interpretation that assigns the truth value* T_0 *(resp.* F_0*) to every* $\mathsf{A} \in B_{\mathsf{P}}$*.*

Now we can proceed to the definition of the immediate consequence operator for PrefLog programs. Recall that we have assumed that the programs we are studying are essentially propositional (they are the ground instances of first-order preferential programs). Therefore, in our definitions we do not need to refer to ground instances of program rules.

Definition 2.16. *Let* I *be an interpretation of a PrefLog program* P*. The operator* $T_P : \mathcal{I}_{\mathsf{P}} \to \mathcal{I}_{\mathsf{P}}$ *is defined as follows:*

$$T_{\mathsf{P}}(I)(\mathsf{A}) = \mathsf{lub}\{I(\mathbf{B}) : (\mathsf{A} \leftarrow \mathbf{B}) \in \mathsf{P}\}.$$

Notice that the T_{P} operator of our language generalizes the T_{P} operator that is used in the fixed-point semantics of classical logic programming. Notice also that we use lub in the definition of T_{P} (instead of max) because there may exist an infinite number of ground instances of rules of the form $\mathsf{A} \leftarrow \mathbf{B}$ in P.

The following two theorems generalize corresponding results that hold for classical logic programs [50]. The proofs of the two theorems can be obtained as special cases of abstract results obtained by Ésik and Rondogiannis [26]. More specifically, Ésik and Rondogiannis developed a general theory for obtaining fixed points of functions that may exhibit a controlled form of non-monotonicity. This class of potentially non-monotonic functions that they considered in their theorem [26], contains as special cases the class of monotonic and continuous functions considered in the semantics of PrefLog. Therefore, in the proofs of the following theorems we use some material from their article.

Theorem 2.1. *Let* P *be a PrefLog program. If all operators that are used in the bodies of the rules of* P *are monotonic (resp. continuous), then the* T_{P} *operator is also monotonic (resp. continuous).*

Proof. A special case of Lemma 7.12 of [26], in the case where the functions involved are monotonic (resp. continuous). □

Theorem 2.2. *Let* P *be a PrefLog program. If all operators used in the bodies of rules of* P *are continuous, then* T_{P} *has a least (with respect to* $\leq$*) fixed point* M_{P} *which is the least upper bound of the set* $\{T_{\mathsf{P}}^n(\emptyset) : n \in \mathbb{N}\}$*. Moreover,* M_{P} *is the least (with respect to* $\leq$*) among all models of* P*.*

Proof. As it is demonstrated in [26], Theorem 6.6 (and also Remark 6.5), since the operator T_{P} is continuous, it has a least (with respect to $\leq$) pre-fixed point which is also a least fixed point; this pre-fixed point is equal to $M_{\mathsf{P}} = \{T_{\mathsf{P}}^n(\emptyset) : n \in \mathbb{N}\}$. It is easy to verify that the set of pre-fixed points of T_{P} coincides with the set of models of program P. Therefore, M_{P} is the least (with respect to $\leq$) among all models of P. □

An example application of Theorem 2.2 in order to compute the minimum model of a given PrefLog program, will be given in Subsection 2.4.1. Actually, as we are going to see in the following chapter, we can use ideas that are based on Theorem 2.2 in order to compute the meaning of a significant class of PrefLog programs in a bottom-up manner.

2.4 Continuous Preference Operators

We have shown that the usual fixed-point semantics and the minimum model property of classical logic programming extends to PrefLog, provided that the preference operators that occur in the bodies of rules are monotonic and continuous. In this section, we reconsider the preference operators ϵ^n, alt^n and opt^n, we demonstrate that they are monotonic and continuous and we use them to define new preference operators. Moreover, we consider the use of these operators in the presence of recursion, which is the main difference of PrefLog from the query system of Agarwal and Wadge [1, 2]. Finally, we demonstrate that there exist simple continuous operators that cannot be defined using the aforementioned ones.

2.4.1 The ϵ operator

The ϵ operator will prove to be the main building block for almost all the operators that we will consider in the rest of the chapter. It is actually a direct task to verify that ϵ is monotonic and continuous:

Proposition 2.3. *The* ϵ *operator is monotonic and continuous.*

Proof. The monotonicity of ϵ is immediate. To demonstrate continuity, let $S = \{x_i : x_i \in \mathbb{V},\ i \in \mathbb{N}\}$ such that $x_i \leq x_{i+1}$ for every $i \in \mathbb{N}$. It suffices to show that $\|\epsilon\|(\text{lub}(S)) = \text{lub}\{\|\epsilon\|(x) : x \in S\}$. We distinguish two cases for lub(S).

- Let lub$(S) \in S$. In this case we immediately have $\|\epsilon\|(\text{lub}(S)) \leq \text{lub}\{\|\epsilon\|(x) : x \in S\}$. Moreover, since ϵ is monotonic, it holds that $\|\epsilon\|(\text{lub}(S)) \geq \|\epsilon\|(x)$, for all $x \in S$. Therefore, $\|\epsilon\|(\text{lub}(S))$ is an upper bound of $\{\|\epsilon\|(x) : x \in S\}$ which implies that $\|\epsilon\|(\text{lub}(S)) \geq \text{lub}\{\|\epsilon\|(x) : x \in S\}$.
- Let lub$(S) \notin S$. This implies that lub$(S) = 0$ and $\|\epsilon\|(\text{lub}(S)) = 0$. Moreover, for all $x \in S$ it holds $x < 0$ and there exists some $x' \in S$ such that $x < x'$. Since $\|\epsilon\|(F_k) = F_{k+1}$, for all $x \in S$ it holds $\|\epsilon\|(x) < 0$ and for each $x \in S$ there exists some $x' \in S$ such that $\|\epsilon\|(x) < \|\epsilon\|(x')$. Consequently, lub$\{\|\epsilon\|(x) : x \in S\} = 0$.

The above two cases imply that ϵ is continuous. □

Notice that it holds ϵ^n A $= \overbrace{\epsilon\ \epsilon\ \cdots\ \epsilon}^{n}$ A. Since the composition of monotonic (resp. continuous) operators is monotonic (resp. continuous), it is immediate that for every n, the operator ϵ^n is monotonic and continuous.

One noteworthy difference between the use of ϵ in PrefLog and in the framework of Agarwal and Wadge [1, 2] is that in the presence of recursion, the ϵ operator may lead to the intermediate truth value 0, as the following simple example illustrates.

Example 2.4. *Consider the following propositional program:*

$$\mathtt{p} \leftarrow \epsilon\,\mathtt{p}.$$

It is not hard to verify using the definition of the T_P operator and Theorem 2.2, that the approximations to the minimum model of the program are the following:

$$\begin{array}{c} \{(\mathrm{p}, F_0)\} \\ \{(\mathrm{p}, F_1)\} \\ \cdots \\ \{(\mathrm{p}, F_i)\} \\ \cdots \end{array}$$

The minimum model of the program is, therefore, the least upper bound of these approximations, namely the interpretation $\{(\mathrm{p}, 0)\}$. Notice that despite the simplicity of the given program, an infinite number of steps is required in order to obtain its minimum model.

Notice that in the context of the query system of Ararwal and Wadge [1, 2], the intermediate truth value 0 never arises (due to the lack of recursion).

2.4.2 The Operators opt and alt

It is also easy to see that the operators opt^n and alt^n are continuous (and therefore monotonic) for all $n \in \mathbb{N}$. If A is a formula then the following logical equivalences hold [1, 2]:

$$\text{opt}^n A \equiv (A \vee \epsilon^n A)$$

$$\text{alt}^n A \equiv (A \wedge \epsilon^n A)$$

The above observation together with the fact that the operators $\wedge$ and $\vee$ are easily seen to be monotonic and continuous, leads to the following proposition:

Proposition 2.4. *For every $n \in \mathbb{N}$, the operators* opt^n *and* alt^n *are monotonic and continuous.*

Table 2.1a and Table 2.1b describe the behavior of opt and alt for certain cases of inputs. Since these expressions are quite common when we express many different levels of preferences, we give the following proposition that captures the intuitive meaning of such expressions.

Proposition 2.5. *Let $n \geq 0$ and $v_0, \ldots, v_n \in \{F_0, T_0\}$. Then:*

1. $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) = T_0 \iff v_i = T_0$, *for all* $0 \leq i \leq n$.
2. $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) = F_k \iff v_i = T_0$, *for all* $0 \leq i < k$ *and* $v_k = F_0$.

Table 2.1: Truth tables of simple PrefLog queries.

(a) A simple PrefLog query that uses the operators opt^n **and** $\wedge$**.**

A	B	C	$A \wedge \text{opt}B \wedge \text{opt}^2C$
F_0	F_0, T_0	F_0, T_0	F_0
T_0	F_0	F_0, T_0	F_1
T_0	T_0	F_0	F_2
T_0	T_0	T_0	T_0

(b) A simple PrefLog query that uses the operators alt^n **and** $\vee$**.**

A	B	C	$A \vee \text{alt}B \vee \text{alt}^2C$
F_0	F_0	F_0	F_0
F_0	F_0	T_0	T_2
F_0	T_0	F_0, T_0	T_1
T_0	F_0, T_0	F_0, T_0	T_0

3. $\| \bigvee \|_{i=0}^{n} \|\text{alt}^i\|(v_i) = F_0 \iff v_i = F_0$, *for all* $0 \leq i \leq n$.
4. $\| \bigvee \|_{i=0}^{n} \|\text{alt}^i\|(v_i) = T_k \iff v_i = F_0$, *for all* $0 \leq i < k$ *and* $v_k = T_0$.

Proof. We study the case of $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i)$.

- To demonstrate (1), first notice that if $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) = T_0$, then $\|\text{opt}^i\|(v_i) = T_0$ for all $0 \leq i \leq n$ and therefore $v_i = T_0$ for all $0 \leq i \leq n$.
  On the other hand, if $v_i = T_0$ for all $0 \leq i \leq n$, we have that $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(T_0) = T_0$.
- To demonstrate (2), first assume that $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) = F_k$. Then, for every $0 \leq i < k$, it must be $v_i = T_0$ because otherwise it would be $\| \bigwedge \|_{i=0}^{k-1} \|\text{opt}^i\|(v_i) < F_k$ and therefore $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) < F_k$ (contradiction).
  Moreover, for every $k < i \leq n$ it holds that $\|\text{opt}^i\|(v_i) > F_k$. Therefore, $v_i = T_0$ for all $0 \leq i < k$ and $v_k = F_0$.
  On the other hand, assume that $v_i = T_0$ for all $0 \leq i < k$ and $v_k = F_0$. Then, it is $(\| \bigwedge \|_{i=0}^{k-1} \|\text{opt}^i\|(T_0)) \| \wedge \| (\|\text{opt}^k\|(F_0)) = F_k$.
  Moreover, $\| \bigwedge \|_{i=k+1}^{n} \|\text{opt}^i\|(v_i) > F_k$ and therefore $\| \bigwedge \|_{i=0}^{n} \|\text{opt}^i\|(v_i) = F_k$.

The proof for $\| \bigwedge \|_{i=0}^{n} \|\text{alt}^i\|(v_i)$ is symmetric and thus will be omitted. □

The above proposition has the following intuitive implications regarding a special form of body formulas of PrefLog. Consider a formula of the form $\bigwedge_{i=0}^{n} \text{opt}^i \text{A}_i$, where each A_i is an atom; this formula is essentially a conjunction of a primary (A_0) and some optional requirements ($\text{A}_1, \ldots, \text{A}_n$), with a requirement A_i being more preferred than A_j if $i < j$. Also, a formula of the form $\bigvee_{i=0}^{n} \text{alt}^i \text{A}_i$ corresponds to a disjunction of a primary (A_0) and some alternative options ($\text{A}_1, \ldots, \text{A}_n$), with an option A_i being more preferred than A_j if $i < j$. In the first case, if we know that the truth value of each A_i is either T_0 or F_0 and if the result is equal to F_k, we immediately know that the atom A_k is the first atom that fails; symmetrically, in the latter case, if the result is equal to T_k, the atom A_k is the first one that succeeds.

2.4.3 Preferences and Recursion

The combination of recursion with preferences gives interesting applications that could not be tackled with the non-recursive query language of Agarwal and Wadge [1, 2]. Three such examples are given below. In particular, the first two examples show a preferential version of the classical transitive closure logic program, while the third one offers a more real-world example where the 0 value can occur in a program.

Example 2.5. *Consider the graph of Figure 2.1, which represents a map of roads between villages in the countryside. Two villages can be connected by a two-lane road (denoted by a double line), by a one-lane road (denoted by a single line), or by a dirt road (denoted by a dashed line). This graph is denoted with the following set of facts:*

$\mathtt{e(a, b, two\_lane)}.$
$\mathtt{e(a, c, one\_lane)}.$
$\mathtt{e(b, c, two\_lane)}.$
$\mathtt{e(c, d, one\_lane)}.$
$\mathtt{e(a, d, dirt)}.$
$\mathtt{e(c, e, dirt)}.$

A driver who wants to travel in this area prefers a two-lane road over a one-lane road and a one-lane road over a dirt road. Given two villages in the map, we want to know if there exists a path between them, and if any, we would like to know if the driver is forced to drive through a road of smaller preference. The following program captures the above situation:

$$\begin{array}{lcl} \mathtt{ppath(X, Y)} & \leftarrow & \mathtt{p(X, Y)}. \\ \mathtt{ppath(X, Y)} & \leftarrow & \mathtt{p(X, Z)} \wedge \mathtt{ppath(Z, Y)}. \end{array}$$

$$\begin{array}{lcll} \mathtt{p(X, Y)} & \leftarrow & \mathtt{e(X, Y, two\_lane)} \vee & \\ & & \text{alt} & \mathtt{e(X, Y, one\_lane)} \vee \\ & & \text{alt}^2 & \mathtt{e(X, Y, dirt)}. \end{array}$$

In order to travel from a to b or c the driver will only drive through two-lane roads (by using the path through b), but in order to travel from a to d she has to pass through the single-track road cd (but still can ignore the dirt road ad). Therefore, the queries $\leftarrow \mathtt{ppath(a, b)}$, $\leftarrow \mathtt{ppath(a, c)}$ and $\leftarrow \mathtt{ppath(a, d)}$ will return T_0, T_0 and T_1, respectively. However, the driver can't avoid to use a dirt road in order to reach the village e, therefore the query $\leftarrow \mathtt{ppath(a, e)}$ will yield the truth value T_2.

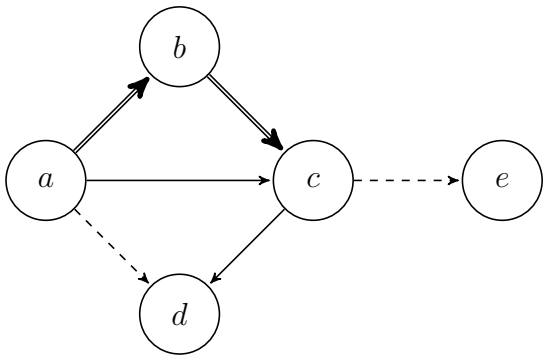

Figure 2.1: A graph that represents a map of roads.

Example 2.6. *Assume we would like to fly from a given city to another one, and we prefer direct flights from non-direct ones. Moreover, the more stopovers a flight has the less desirable it is to us. We can model such a situation as follows:*

$$\begin{array}{lcl} \texttt{flight}(\texttt{X},\texttt{Y}) & \leftarrow & \texttt{direct}(\texttt{X},\texttt{Y}) \vee \\ & & \mathsf{alt}(\texttt{direct}(\texttt{X},\texttt{Z}) \wedge \texttt{flight}(\texttt{Z},\texttt{Y})). \end{array}$$

Assume we are also given the facts:

$$\begin{array}{l} \texttt{direct}(\texttt{athens},\texttt{rome}). \\ \texttt{direct}(\texttt{rome},\texttt{london}). \\ \texttt{direct}(\texttt{london},\texttt{toronto}). \end{array}$$

Then, it can be easily checked that in the minimum Herbrand model of the program, the destinations that can be reached from Athens correspond to different truth values, depending on the number of stopovers that are needed in order to reach them; in particular, notice that the corresponding atoms $\texttt{flight}(\texttt{athens},\texttt{rome})$*,* $\texttt{flight}(\texttt{athens},\texttt{london})$ *and* $\texttt{flight}(\texttt{athens},\texttt{toronto})$ *will have in the least model the truth values* T_0*,* T_1 *and* T_2*, respectively.*

Example 2.7. *The following program expresses a preference of the form "I may not like this but it will be fine with me if you like it" is expressed by two persons:*

$$\begin{array}{lcl} \texttt{likes}(\texttt{john},\texttt{X}) & \leftarrow & \texttt{good\_quality}(\texttt{X}) \wedge \mathsf{opt}\ \texttt{likes}(\texttt{paul},\texttt{X}). \\ \texttt{likes}(\texttt{paul},\texttt{X}) & \leftarrow & \texttt{good\_quality}(\texttt{X}) \wedge \mathsf{opt}\ \texttt{likes}(\texttt{john},\texttt{X}). \end{array}$$

$$\texttt{good\_quality}(\texttt{object}).$$

Paul likes item $\texttt{item}$ *if it is of good quality and optionally if John likes it. Since John makes a symmetrical statement, the atoms* $\texttt{likes}(\texttt{john},\texttt{item})$ *and* $\texttt{likes}(\texttt{paul},\texttt{item})$ *have the value 0 in the minimum model. In other words, cycles that produce 0 can easily occur in innocent-looking programs.*

2.4.4 Defining New Operators

We now introduce an n-ary operator that counts the number of its arguments that succeed. The more arguments that succeed, the truer the output of the operator is. We denote this operator by howTrue.

Definition 2.17. *Let* $v_1, \ldots, v_n \in \mathbb{V}$*. We define the operator* howTrue *as follows:*

$$\|\mathsf{howTrue}\|(v_1, \ldots, v_n) = \left\| \bigwedge \right\|_{\pi \in \Pi_n} \left(\left\| \bigvee \right\|_{i=1}^{n} \|\mathsf{alt}^i\|(v_{\pi(i)}) \right)$$

where Π_n *is the set of permutations of the* n*-tuple* $(1, \ldots, n)$*.*

For example, given any two formulas A and B, the binary howTrue operator is equivalent to the formula:

$$\mathsf{howTrue}(A, B) = (A \vee \mathsf{alt}\, B) \wedge (B \vee \mathsf{alt}\, A)$$

Moreover, given any three formulas A, B and C, the ternary howTrue operator is equivalent to the formula:

$$\begin{array}{rcl} \mathsf{howTrue}(A, B, C) & = & (A \vee \mathsf{alt}\, B \vee \mathsf{alt}^2 C) \wedge \\ & & (A \vee \mathsf{alt}\, C \vee \mathsf{alt}^2 B) \wedge \\ & & (B \vee \mathsf{alt}\, A \vee \mathsf{alt}^2 C) \wedge \\ & & (B \vee \mathsf{alt}\, C \vee \mathsf{alt}^2 A) \wedge \\ & & (C \vee \mathsf{alt}\, A \vee \mathsf{alt}^2 B) \wedge \\ & & (C \vee \mathsf{alt}\, B \vee \mathsf{alt}^2 A) \end{array}$$

Example 2.8. *Consider the following program which motivates the use of* howTrue*. Three friends want to decide what movie to watch, based on the majority of their preferences.*

$$\begin{array}{lll} \texttt{watch(X)} & \leftarrow & \text{howTrue} \ (\\ & & \quad \texttt{likes(mary, X)}, \\ & & \quad \texttt{likes(bob, X)}, \\ & & \quad \texttt{likes(tom, X)} \\ & &). \end{array}$$

The genre preferences of the three friends are defined using the following facts:

$$\begin{array}{l} \texttt{likes(mary, drama)}. \\ \texttt{likes(bob, action)}. \\ \texttt{likes(tom, drama)}. \end{array}$$

It can be verified (see also the discussion below) that in the least model of the above program, the atom `watch(drama)` *will have the truth value* T_1 *while the atom* `watch(action)` *will have (the smaller) truth value* T_2*.*

Example 2.9. *Consider the following program which allows the user to express preferences regarding the paper she would like to review:*

$$\begin{array}{lll} \texttt{preferred\_paper(P)} & \leftarrow & \text{howTrue} \ (\\ & & \quad \texttt{databases(P)}, \\ & & \quad \texttt{logic\_programming(P)} \\ & &). \end{array}$$

If a paper $\texttt{p}$ *belongs to both databases and logic programming, then the corresponding atom* $\texttt{preferred\_paper(p)}$ *will have the value* T_0*, if it belongs to one of the two areas then it will have the value* T_1 *and if its topic is outside the two preferred areas, it will have the value* F_0*.*

Since the operators alt, $\wedge, \vee$ are continuous, the operator howTrue is also continuous, and as a result, we can add it to our language. Hence, the above definition allows us to show that howTrue is a continuous operator. However, it does not give the fastest way for calculating the value of howTrue. Alternatively, we can calculate the value of $\|\text{howTrue}\|(v_1, \ldots, v_n)$ by computing $\| \bigvee \|_{i=0}^{n-1} \|\text{alt}^i\| u_i$ where $(u_1, \ldots, u_n)$ is a *sorted* permutation in ascending order of the tuple $(v_1, \ldots, v_n)$.

Proposition 2.6. *Let* $v_1, \ldots, v_n \in \mathbb{V}$*. Then:*

$$\|\text{howTrue}\|(v_1, \ldots, v_n) = \left\| \bigvee \right\|_{i=1}^{n} \|\text{alt}^i\|(u_i)$$

where $(u_1, \ldots, u_n)$ *is a permutation of* $(v_1, \ldots, v_n)$ *such that* $u_i \leq u_{i+1}$ *for all* $1 \leq i < n$*.*

Proof. We use the notation $\delta(x_1, \ldots, x_n) = \| \bigvee \|_{i=1}^{n} \|\text{alt}^i\|(x_i)$.

Notice now that the value of $\|\text{howTrue}\|(v_1, \ldots, v_n)$ can be obtained by finding a permutation $(u_1, \ldots, u_n)$ of $(v_1, \ldots, v_n)$ such that $\delta(u_1, \ldots, u_n)$ is minimum. In order to show that if we sort $(v_1, \ldots, v_n)$ in ascending order we can have a sequence that minimizes δ, it

Table 2.2: Truth table of a PrefLog query that uses the operator howTrue**.**

A	B	C	howTrue(A, B, C)
F_0	F_0	F_0	F_0
F_0	F_0	T_0	T_2
F_0	T_0	F_0	T_2
T_0	F_0	F_0	T_2
F_0	T_0	T_0	T_1
T_0	T_0	F_0	T_1
T_0	F_0	T_0	T_1
T_0	T_0	T_0	T_0

suffices to show that if we swap two consecutive unsorted elements in the sequence the value of δ lowers.

Let $\mathbf{z} = (z_1, \ldots, z_n)$ be an unsorted permutation of the tuple $(v_1, \ldots, v_n)$. It must hold $z_k > z_{k+1}$ for some $1 \leq k < n$. Therefore, it must hold:

$$\begin{aligned} \|\text{alt}^k\|(z_k) \vee \|\text{alt}^{k+1}\|(z_{k+1}) &= \|\text{alt}^k\|(z_k) \\ \|\text{alt}^k\|(z_{k+1}) \vee \|\text{alt}^{k+1}\|(z_k) &= \|\text{alt}^{k+1}\|(z_k) \end{aligned}$$

and that should imply that:

$$\|\text{alt}^k\|(z_{k+1}) \vee \|\text{alt}^{k+1}\|(z_k) \;\leq\; \|\text{alt}^k\|(z_k) \vee \|\text{alt}^{k+1}\|(z_{k+1})$$

Let $\mathbf{z}'$ be a tuple obtained from $\mathbf{z}$ by swapping the z_k, z_{k+1} elements. By the above, $\delta(\mathbf{z}') \leq \delta(\mathbf{z})$. □

Suppose that we have a query where all arguments that are passed in howTrue are either true or false. In Table 2.2 we see such a truth table for the ternary version of howTrue. The following proposition captures this behavior of howTrue in a more general way.

Proposition 2.7. *Let* $n \geq 1$ *and let* $\mathbf{v} = (v_1, \ldots, v_n)$ *where* $v_i \in \{F_0, T_0\}$ *for all* $1 \leq i \leq n$. *Moreover, let* $\mathbf{v} \parallel v = \{i : v_i = v,\ 1 \leq i \leq n\}$ *Then:*

1. howTrue$(v_1, \ldots, v_n) = F_0 \iff \left|\mathbf{v} \parallel F_0\right| = n$ *and* $\left|\mathbf{v} \parallel T_0\right| = 0$.
2. howTrue$(v_1, \ldots, v_n) = T_k \iff \left|\mathbf{v} \parallel F_0\right| = k$ *and* $\left|\mathbf{v} \parallel T_0\right| = n - k$.

Proof. Let $(u_1, \ldots, u_n)$ be a sorted permutation of $v_1, \ldots, v_n$ in ascending order. Since $v_i \in \{F_0, T_0\}$ for all $1 \leq i \leq n$, it holds $\left|\mathbf{v} \parallel F_0\right| + \left|\mathbf{v} \parallel T_0\right| = n$.

- Regarding (1): $\left|\mathbf{v} \parallel F_0\right| = n$ and $\left|\mathbf{v} \parallel T_0\right| = 0 \iff u_i = F_0$ for all $1 \leq i \leq n \iff \| \bigvee \|_{i=0}^{n-1} \|\text{alt}^i\| u_i = F_0 \iff \|\text{howTrue}\|(v_1, \ldots, v_n) = F_0$.
- Regarding (2): $\left|\mathbf{v} \parallel F_0\right| = k$ and $\left|\mathbf{v} \parallel T_0\right| = n - k \iff u_i = F_0$ for all $1 \leq i \leq k$ and $u_i = T_0$ for all $k < i \leq n \iff \| \bigvee \|_{i=0}^{n-1} \|\text{alt}^i\| u_i = T_k \iff \|\text{howTrue}\|(v_1, \ldots, v_n) = T_k$. □

If we apply howTrue to a sequence of formulas, then we get a positive value if some of the formulas succeeded. The result is equal to F_0 if all formulas failed; the result is equal to T_0 if all formulas succeeded; the result is equal to some intermediate truth value if some formulas succeeded and others failed, and a greater truth value of the result corresponds to a greater number of true formulas.

2.4.5 Operators Non-Definable with $\wedge$, $\vee$ and ϵ

One can easily observe that there exist many natural monotonic and continuous operators that are not definable solely with the use of $\wedge, \vee$ and ϵ. The following proposition poses some straightforward restrictions on the operators that can be defined that way.

Proposition 2.8. *Let $\|\nabla\| : \mathbb{V}^n \to \mathbb{N}$ be an operator that has been defined solely with the use of $\wedge$, $\vee$ and ϵ. Then:*

1. *For all $v_1, \ldots, v_n$ such that $v_i \geq 0$ for all $1 \leq i \leq n$, it also holds $\|\nabla\|(v_1, \ldots, v_n) \geq 0$.*
2. *For all $v_1, \ldots, v_n$ such that $v_i < 0$ for all $1 \leq i \leq n$, it also holds $\|\nabla\|(v_1, \ldots, v_n) < 0$.*
3. *For all $v_1, \ldots, v_n \in \mathbb{V}$, $\mathsf{ord}(\|\nabla\|(v_1, \ldots, v_n)) \geq \min\{\mathsf{ord}(v_1), \ldots, \mathsf{ord}(v_n)\}$.*

Proof. The conjunction/disjunction of truth values that are greater than or equal to 0 returns a truth value that is greater than or equal to 0 (since the ϵ operator does not change the polarity of a truth value). Similarly for negative truth values. The third statement holds because ϵ increases the order of its argument. □

In the following example, we illustrate the existence of operators that are non-definable using $\wedge$, $\vee$ and ϵ.

Example 2.10. *Consider the following operator:*

$$\|\mathsf{isClassicalTrue}\|(v) = \begin{cases} F_0, & \textit{if } v < T_0 \\ T_0, & \textit{if } v = T_0 \end{cases}$$

The above function is monotonic and continuous. However, it is not definable solely with $\wedge$, $\vee$ and ϵ, since for all truth values that are greater than or equal to 0 (except for T_0), it returns a negative truth value. Consider now the following function:

$$\|\mathsf{prev}\|(v) = \begin{cases} F_0, & \textit{if } v = F_0 \\ F_{i-1}, & \textit{if } F_0 < v < 0 \\ 0, & \textit{if } v = 0 \\ T_{i-1}, & \textit{if } 0 < v < T_0 \\ T_0, & \textit{if } v = T_0 \end{cases}$$

Again, this function is also monotonic and continuous but it is not definable solely with $\wedge$, $\vee$ and ϵ, since the ϵ operator increases the order of its parameter (Case 3 of Proposition 2.8).

It is an interesting research direction to investigate formalisms that are easy to use and that guarantee the definition of monotonic and continuous preference operators that are beyond the class of those definable solely with $\wedge$, $\vee$ and ϵ. We will discuss this possibility in Chapter 8.

2.5 Expressiveness of PrefLog Programs

In this section, we discuss the expressive power of our language. In PrefLog preferences are expressed using preference operators, whose denotations are functions $\mathbb{V}^n \rightarrow \mathbb{V}$, where $\mathbb{V}$ is a totally ordered set such that a greater value corresponds to a greater degree of interest or to a greater preference. Regarding the expressiveness of PrefLog, a question that arises is whether there exist types of preferences that cannot be expressed using PrefLog operators. To begin with, consider the following example:

Example 2.11. *Consider a consumer that wants to rank hotels for summer vacations based on their price per night and the distance from the sea. In addition, assume that the consumer travels on a budget, therefore the former criterion is more important than the latter. That is, cheap hotels will always be preferred over expensive ones, and if two hotel rooms cost the same price, then the consumer will want the one which is closer to the sea.*

These types of preferences are known in the literature as *lexicographic preferences*. The term *lexicographic* refers to the fact that these types of comparisons resemble the comparison of two words in a dictionary; given two words, we compare their first letters, and only if they are the same we continue to compare the next and so on. In the following definition, we define the notion of lexicographic comparison. Without loss of generality, we will focus on pairs.

Definition 2.18. *Let V be a totally ordered set and $(x_1, x_2), (y_1, y_2) \in V \times V$. We write $(x_1, x_2) >_{\text{lex}} (y_1, y_2)$ iff either of the following propositions holds:*

- $x_1 > y_1$ *or*
- $x_1 = y_1$ *and* $x_2 > y_2$.

A function that represents a lexicographic preference should have the following property. Again, without loss of generality, we will focus on binary functions.

Definition 2.19. *Let V be a totally ordered set, $\mathbf{x}, \mathbf{y} \in V \times V$ and $f : V \times V \rightarrow V$. We say that f is a lexicographic function, if it holds $\mathbf{x} >_{\text{lex}} \mathbf{y} \implies f(\mathbf{x}) > f(\mathbf{y})$.*

A distinctive feature of lexicographic preferences is that they cannot be modeled using a real-valued preference function [67]. For instance, a function that has the form of a weighted summation cannot be used, because no matter how we try to increase the weight of the first argument, an appropriately chosen second argument can reverse the outcome. In the following, we show that this behavior is also transferred to the case of PrefLog.

Lemma 2.1. *Let V be a totally ordered set, and $f : V \times V \rightarrow V$ be a lexicographic function. Then, f is also a 1-1 function.*

Proof. Suppose that f is not a 1-1 function. Therefore, there exist some $\mathbf{x}, \mathbf{y} \in V \times V$ such that $\mathbf{x} \neq \mathbf{y}$ and $f(\mathbf{x}) = f(\mathbf{y})$. Notice that it must hold either $\mathbf{x} <_{\text{lex}} \mathbf{y}$ or $\mathbf{x} >_{\text{lex}} \mathbf{y}$. Let $\mathbf{x} <_{\text{lex}} \mathbf{y}$. Then, by Definition 2.19 it must be $f(\mathbf{x}) < f(\mathbf{y})$. Let $\mathbf{x} >_{\text{lex}} \mathbf{y}$. Again, using the same definition it must be $f(\mathbf{x}) > f(\mathbf{y})$. We reached a contradiction, therefore f must be a 1-1 function. □

Proposition 2.9. *There does not exist any lexicographic function* $f : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$*.*

Proof. Suppose that there exists such a function. Then, according to the definition of $\mathbb{V}$ and Definition 2.19 it holds

$$f(F_0, u) < f(F_1, F_0) < f(T_1, T_0) < f(T_0, v)$$

for all $u, v \in \mathbb{V}$. We can distinguish the following cases:

- Suppose that $f(F_1, F_0) < 0$. Then, $f(F_1, F_0) = F_k$ for some $k \in \mathbb{N}$. Then, $|\{f(F_0, u) : u \in \mathbb{V}\}| = k$. Since $S = \{(F_0, u) : u \in \mathbb{V}\}$ is infinite we deduce that there exists (at least) two elements $\mathbf{x}, \mathbf{y} \in S$ such that $f(\mathbf{x}) = f(\mathbf{y})$ (Pigeonhole Principle). Therefore, f is not a 1-1 function. Contradiction, due to Lemma 2.1.
- Suppose that $f(F_1, F_0) \geq 0$. Then, $f(T_1, T_0) = T_k$ for some $k \in \mathbb{N}$. Then, $|\{f(T_0, v) : v \in \mathbb{V}\}| = k$. Since $S = \{(T_0, v) : v \in \mathbb{V}\}$ is infinite we deduce that there exists (at least) two elements $\mathbf{x}, \mathbf{y} \in S$ such that $f(\mathbf{x}) = f(\mathbf{y})$ (Pigeonhole Principle). Therefore, f is not a 1-1 function. Contradiction, due to Lemma 2.1.

Therefore, there does not exist such a function f. □

The above proposition suggests that PrefLog programs cannot capture the full power of preference relations due to the form of the underlying set of truth values. More specifically, since there does not exist any lexicographic function $f : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{V}$, there does not exist any lexicographic preference PrefLog operator as well. A future extension of PrefLog that would support lexicographic preferences would possibly require an extension of the set $\mathbb{V}$. We will further discuss this prospect in Chapter 8.

2.6 Summary

In this chapter, we introduced PrefLog, an extension of classical logic programming that supports preference operators. The semantics of these operators are functions $\mathbb{V}^n \rightarrow \mathbb{V}$, where $n > 0$ and $\mathbb{V}$ is an infinite set of truth values. This set contains infinite truth values that are “less true” than standard truth, and infinite values that are “less false” than standard falsity. These different levels of truth values correspond to different degrees of preferences.

We demonstrated that if the denotations of the operators are continuous over this infinite-valued underlying domain $\mathbb{V}$, then the programs of the new language are guaranteed to retain the desirable properties of classical logic programming — most notably the existence of a minimum Herbrand model. In addition, we equipped PrefLog with a set of simple preference operators, including two special ones that have the intuitive meaning “optionally” and “alternatively” respectively. We also described a simple approach for building new continuous operators, and we illustrated the use of preferential operators in various example programs.

Finally, we closed this chapter by discussing briefly the expressiveness of our language. PrefLog programs can model quantitative preference functions over the set $\mathbb{V}$. However, since $\mathbb{V}$ does not support lexicographic functions, PrefLog cannot be used for expressing lexicographic preferences.

3. EVALUATION OF A FUNCTION-FREE CLASS OF PREFLOG PROGRAMS

In this chapter, we introduce a terminating bottom-up evaluation method for a well-defined class of function-free PrefLog programs. We demonstrate the correctness of this technique as well as the fact that the evaluation process terminates for any program that belongs to this class. Ensuring termination is not a straightforward task, because the underlying truth domain and the set of all possible interpretations of a function-free PrefLog program are both infinite.

3.1 Overview

In the previous chapter, we introduced PrefLog, a first-order logic programming language that uses an infinite set of truth values and a finite set of arbitrary continuous operators. PrefLog programs include two "levels" of infinity; first, a PrefLog program P has an infinite Herbrand base B_{P}; and second, each atom of B_{P} can receive a value taken from the infinite set $\mathbb{V}$. Due to this characteristic, the evaluation of PrefLog programs can cause termination problems, a phenomenon that will be discussed thoroughly throughout the chapter.

We reduce our focus in a class of PrefLog programs that do not contain any function symbols. Roughly speaking, this fragment of PrefLog is the preferential analogue of Datalog [75]. Unlike Prolog, the ground instantiation of a Datalog program is finite, therefore a *bottom-up* evaluation of a Datalog program is guaranteed to terminate. However, termination cannot be ensured in the case of naive bottom-up evaluation for function-free PrefLog programs; during the evaluation process, an atom may obtain an infinite set of truth values, and this may result in non-termination. In this chapter, we propose a bottom-up evaluation technique that exploits an important property of the PrefLog fragment of our interest and as a result, this technique is guaranteed to terminate for every program in this fragment.

The rest of the chapter is organized as follows: in Section 3.2, we define $\{\epsilon, \wedge\}$-programs, the function-free PrefLog fragment of our interest; in Section 3.3 we introduce our terminating bottom-up proof procedure for $\{\epsilon, \wedge\}$-programs; in Section 3.4 we make a brief discussion of the implementations of the above techniques, and finally, we close with a brief summary of the chapter.

3.2 The Class of $\{\epsilon, \wedge\}$-programs

In this section, we define the class of $\{\epsilon, \wedge\}$-programs, which is a subset of the class of general PrefLog programs. We then demonstrate that every such program has the property that its least model *does not contain a gap*. This property allows us to tackle termination problems. For instance, in the next section, we will demonstrate that the least fixed point of such *gapless programs* can be computed in a finite number of steps.

3.2.1 $\{\epsilon, \wedge\}$-programs

Throughout the previous chapter, we have been studying the ground instantiation of PrefLog programs. In this chapter, we additionally assume that the ground instantiation consists of a finite number of rules. Intuitively, this means that the initial (first-order) PrefLog program does not contain any function symbols (that would make the ground instantiation

infinite). Moreover, apart from focusing on function-free PrefLog programs, we impose yet another restriction; we reduce the set of preference operations to ϵ and $\wedge$ only. As we will discuss shortly, these restrictions are not quite severe, but on the contrary, this class of programs actually serves as a normal-form of almost all programs of the previous chapter. We formally define the exact class of programs in the following definition.

Definition 3.1. *A $\{\epsilon, \wedge\}$-program is a finite set of rules of one of the following forms:*

1. $\mathsf{A} \leftarrow \mathsf{true}$
2. $\mathsf{A} \leftarrow \mathsf{A}_1 \wedge \cdots \wedge \mathsf{A}_n \wedge \epsilon\,\mathsf{B}_1 \wedge \cdots \wedge \epsilon\,\mathsf{B}_m$

where the A, A_i *for all* $1 \leq i \leq n$ *and* B_j *for all* $1 \leq i \leq m$ *are ground atoms.*

In the previous chapter, we discussed that there exist valid preference operators that cannot be defined using $\wedge$, $\vee$ and ϵ. However, the programs that we have used in the examples of the previous chapter are all constructed using operators that have been built using the three basic operators $\wedge$, $\vee$ and ϵ. The following example illustrates that the ground instantiation of every PrefLog program that we have examined so far, can be transformed into an equivalent $\{\epsilon, \wedge\}$-program through a preprocessing that introduces new propositional atoms.

Example 3.1. *Consider the following program that consists of a single rule:*

$$\mathtt{p} \leftarrow \mathtt{q} \vee \mathsf{alt}\ \mathtt{r} \vee \mathsf{alt}^2\ \mathtt{s}$$

We first eliminate the alt *and* alt^2 *operators by using the equivalent formulas involving conjunction and* ϵ*:*

$$\mathtt{p} \leftarrow \mathtt{q} \vee (\mathtt{r} \wedge \epsilon\,\mathtt{r}) \vee (\mathtt{s} \wedge \epsilon^2\,\mathtt{s})$$

We then eliminate the ϵ^2 *operator by using the equivalence* $\epsilon^2\,\mathtt{s} \equiv \epsilon\,(\epsilon\,\mathtt{s})$*:*

$$\mathtt{p} \leftarrow \mathtt{q} \vee (\mathtt{r} \wedge \epsilon\,\mathtt{r}) \vee (\mathtt{s} \wedge \epsilon\,(\epsilon\,\mathtt{s}))$$

We now use multiple rules in order to eliminate disjunctions:

$$\begin{array}{lcl} \mathtt{p} & \leftarrow & \mathtt{q} \\ \mathtt{p} & \leftarrow & \mathtt{r} \wedge \epsilon\,\mathtt{r} \\ \mathtt{p} & \leftarrow & \mathtt{s} \wedge \epsilon\,(\epsilon\,\mathtt{s}) \end{array}$$

We can now eliminate the remaining pairs of parentheses by introducing extra propositional symbols:

$$\begin{array}{lcl} \mathtt{p} & \leftarrow & \mathtt{q} \\ \mathtt{p} & \leftarrow & \mathtt{r} \wedge \epsilon\,\mathtt{r} \\ \mathtt{p} & \leftarrow & \mathtt{s} \wedge \epsilon\,\mathtt{w} \\ \mathtt{w} & \leftarrow & \epsilon\,\mathtt{s} \end{array}$$

The above is a valid $\{\epsilon, \wedge\}$*-program.*

Provided that a PrefLog program can be transformed into a finite propositional program (this happens for instance in function-free PrefLog programs), and provided that every preference operator that is used in the program can be rewritten into a finite formula that uses only the operators $\wedge, \vee$ and ϵ, this program can be transformed into a finite $\{\epsilon, \wedge\}$-program using the procedure of Example 3.1.

3.2.2 The Gapless Property of $\{\epsilon, \wedge\}$-programs

In this subsection we will demonstrate a crucial property of $\{\epsilon, \wedge\}$-programs regarding their minimum model, namely that every $\{\epsilon, \wedge\}$-program is *gapless*. We begin by defining the notion of a *gap*.

Definition 3.2. *Let I be an interpretation of a $\{\epsilon, \wedge\}$-program* P*. We say that I contains a gap at order $\delta \in \mathbb{N}$ if:*

- *For every $0 \leq n < \delta$, there exists at least one atom* $\mathsf{A} \in B_P$ *such that* $\mathrm{ord}(I(\mathsf{A})) = n$.
- *There does not exist any atom* $\mathsf{A} \in B_P$ *such that* $\mathrm{ord}(I(\mathsf{A})) = \delta$.
- *There exists an atom* $\mathsf{A} \in B_{\mathsf{P}}$ *such that* $\delta < \mathrm{ord}(I(\mathsf{A})) < \infty$.

For example, notice that the interpretations $\{(\mathrm{p}, F_0), (\mathrm{q}, T_1)\}$, $\{(\mathrm{p}, 0)\}$ and $\{(\mathrm{p}, T_0), (\mathrm{q}, 0)\}$ do not contain a gap; on the other hand, notice that the interpretations $\{(\mathrm{p}, T_0), (\mathrm{q}, T_2)\}$, $\{(\mathrm{p}, T_1), (\mathrm{q}, F_1)\}$ and $\{(\mathrm{p}, F_2), (\mathrm{q}, 0)\}$ contain a gap (at orders 1, 0 and 0 respectively). We will refer to PrefLog programs that their minimum model does not contain a gap as *gapless* programs.

In the following proposition, we show that every $\{\epsilon, \wedge\}$-program is gapless.

Proposition 3.1. *Let* P *be a $\{\epsilon, \wedge\}$-program. Then, M_{P} does not contain a gap.*

Proof. Let $M = M_{\mathsf{P}}$ be the least model of P and assume that it contains a gap at order $\delta \in \mathbb{N}$. We establish a contradiction by constructing a model M^* of P such that $M^* < M$. We define the following interpretation:

$$M^*(\mathsf{A}) = \begin{cases} T_{r+1}, & \text{if } M(\mathsf{A}) = T_r,\ r > \delta \\ F_{r-1}, & \text{if } M(\mathsf{A}) = F_r,\ r > \delta \\ M(\mathsf{A}), & \text{otherwise.} \end{cases}$$

Obviously, $M^* < M$ (since in M^* all values with finite order greater than δ have been decreased). For the sake of contradiction, we have to show that M^* is also a model of P, i.e., that M^* satisfies every rule of P.

First, notice that M^* satisfies the rules of the form $(\mathsf{A} \leftarrow \text{true})$ because, since M satisfies such rules, it is $M(\mathsf{A}) = T_0$ and by the definition of M^* it is also $M^*(\mathsf{A}) = T_0$. Consider now a rule $\mathsf{A} \leftarrow \mathsf{A}_1, \ldots, \mathsf{A}_n, \epsilon\mathsf{B}_1, \ldots, \epsilon\mathsf{B}_k$. Since M is a model of P, $M(\mathsf{A}) \geq \min\{M(\mathsf{A}_1), \ldots, M(\mathsf{A}_n), \|\epsilon\|M(\mathsf{B}_1), \ldots, \|\epsilon\|M(\mathsf{B}_k)\}$. We show that M^* also satisfies the above rule. We distinguish cases based on the value of $M(\mathsf{A})$. We define:

$$\begin{aligned} v &= \min\{\ M(\mathsf{A}_1), \ldots, M(\mathsf{A}_n),\ \|\epsilon\|M(\mathsf{B}_1), \ldots, \|\epsilon\|M(\mathsf{B}_k)\ \}, \\ v^* &= \min\{\ M^*(\mathsf{A}_1), \ldots, M^*(\mathsf{A}_n),\ \|\epsilon\|M^*(\mathsf{B}_1), \ldots, \|\epsilon\|M^*(\mathsf{B}_k)\ \} \end{aligned}$$

The cases are as follows:

- $M(\mathsf{A}) = F_r$ with $r < \delta$. Since M is a model of P, we have $v \leq F_r$. By the definition of M^* we conclude that it also holds that $v^* \leq F_r$.

```
1: procedure NaiveEvaluation(P)
2:     I(A) := F_0, for all A ∈ B_P.
3:     repeat
4:         I' := I
5:         I(A) := max{I(B) : (A ← B) ∈ P}, for all A ∈ B_P.
6:     until I' = I
7:     return I
8: end procedure
```

Figure 3.1: Naive evaluation strategy for $\{\epsilon, \wedge\}$-programs.

- $M(\mathsf{A}) = T_r$ with $r < \delta$ or $M(\mathsf{A}) = 0$. The proofs for these two cases are similar to the one for the previous case.

- $M(\mathsf{A}) = F_r$ with $r > \delta$. By the definition of M^* we have that $M^*(\mathsf{A}) = F_{r-1}$. Since M is a model of P, we have $v \leq F_r$. Now, if $v = F_r$, we have $v^* = F_{r-1}$; if $v < F_r$ then $v^* < F_r$. In every case, it holds that $M^*(\mathsf{A}) \geq v^*$ and therefore M^* satisfies this rule.

- $M(\mathsf{A}) = T_r$ with $r > \delta$. By the definition of M^* we have that $M^*(\mathsf{A}) = T_{r+1}$. Since M is a model of P, we have $v \leq T_r$. Now, if $v = T_r$, we have $v^* \leq T_{r+1}$; if $v < T_r$ then $v^* < T_r$. In every case, it holds that $M^*(\mathsf{A}) \geq v^*$ and therefore M^* satisfies this rule.

Since M^* satisfies every rule of P, M^* is also a model of P (contradiction). Therefore, M_{P} can not contain a gap. □

The above proposition will play an important role in the evaluation of $\{\epsilon, \wedge\}$-programs. More specifically, the gapless property allows us to modify the usual naive bottom-up evaluation such that the minimum model of a $\{\epsilon, \wedge\}$-program can be computed in a finite number of steps.

3.3 Bottom-up Evaluation

In this section, we focus on the bottom-up evaluation of PrefLog programs. We start by discussing why the usual naive bottom-up evaluation is not adequate even for very simple PrefLog programs. We then reduce our focus into $\{\epsilon, \wedge\}$-programs and we derive a terminating bottom-up procedure for their evaluation.

3.3.1 Inadequacy of Naive Evaluation

Consider PrefLog programs that consist of a finite number of ground rules. The most direct way of evaluating the minimum model of a program P in a bottom-up manner is to mimic the computation of the least fixed point of the T_{P} operator of the program (cf. Figure 3.1). If the source program P was a classical Datalog one, the process of Figure 3.1 would terminate in a finite number of steps, since B_{P} is finite (and in a finite number of steps all the true atoms would be produced). In our case though, the Herbrand base is also finite, but in contrast, the number of possible interpretations of a program is infinite (since each atom can be assigned a truth value from the infinite set $\mathbb{V}$).

Example 3.2. *Consider again the following program:*

$$\mathtt{p} \leftarrow \epsilon\,\mathtt{p}.$$

As we saw in Example 2.4, an infinite number of steps is required in order to compute the least model of the above program. In addition, consider the program:

$$\begin{array}{lcl} \mathtt{q} & \leftarrow & \mathsf{opt}\,\mathtt{r} \\ \mathtt{r} & \leftarrow & \mathsf{opt}\,\mathtt{q} \end{array}$$

Again, the above program, which is derived from Example 2.7, behaves in a similar way.

The above example suggests that the naive bottom-up evaluation is not guaranteed to converge in a finite number of steps for every function-free PrefLog program, due to the infinity of the underlying truth domain. In order to devise a terminating bottom-up procedure, we have to make two basic assumptions. First, we have to consider a fixed set of preference operators with a known behavior; considering arbitrary continuous operators, is far too general since their behavior can be arbitrarily complex. Second, we have to simplify somehow the syntax of our source language. The class of $\{\epsilon, \wedge\}$-programs that was defined in the previous section satisfies these two requirements.

3.3.2 Terminating Bottom-up Evaluation of $\{\epsilon, \wedge\}$-programs

In this subsection, we define a terminating bottom-up evaluation procedure for the class of $\{\epsilon, \wedge\}$-programs. The fact that these programs are gapless offers us a termination criterion; this criterion is the appearance of a gap during the evaluation process.

An idea for computing the least fixed point of a $\{\epsilon, \wedge\}$-program P in a finite number of steps using the gapless property, is to iterate the T_{P} operator until an interpretation that contains a gap is produced. In such a case, all the atoms that have a value with an order that is above the gap are set to the 0 truth value. Unfortunately, this simple procedure does not work, as the following example illustrates.

Example 3.3. *Consider the following program:*

$$\begin{array}{lcl} \mathtt{p} & \leftarrow & \epsilon\,\mathtt{p}. \\ \mathtt{q} & \leftarrow & \mathtt{r}. \\ \mathtt{r} & \leftarrow & \mathtt{s}. \\ \mathtt{s} & \leftarrow & . \end{array}$$

The minimum model of this program is constructed by iterating the T_{P} operator as follows:

$$\begin{array}{c} \{(\mathtt{p}, F_0), (\mathtt{q}, F_0), (\mathtt{r}, F_0), (\mathtt{s}, F_0)\} \\ \{(\mathtt{p}, F_1), (\mathtt{q}, F_0), (\mathtt{r}, F_0), (\mathtt{s}, T_0)\} \\ \{(\mathtt{p}, F_2), (\mathtt{q}, F_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\} \end{array}$$

Notice that the last interpretation above has a gap at order 1. However, it would be wrong to stop the iterations at this point. Actually, one more iteration of the T_{P} operator gives the following interpretation:

$$\{(\mathtt{p}, F_3), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

```
procedure TerminatingEvaluation(P)
    n := 0
    S := B_P
    repeat
        I(A) := F_n, for all A ∈ S
        repeat
            I' := I
            I(A) := max{I(B) : (A ← B) ∈ P}, for all A ∈ B_P
        until I'(A) = I(A), for all A ∈ B_P such that ord(I(A)) = n
        S' := {A : A ∈ B_P, ord(I(A)) = n}
        if S' = {} then
            I(A) := 0, for all A ∈ S
            S := {}
        else
            S := S − S'
        end if
        n := n + 1
    until S = {}
    return I
end procedure
```

Figure 3.2: Terminating evaluation strategy for $\{\epsilon, \wedge\}$-programs.

Further iterations of the T_{P} will only affect the variable p which will eventually converge to 0. In other words, the correct model for the above program is:

$$\{(\mathtt{p}, 0), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

Looking at the above steps, one understands that it is not correct to stop immediately when a gap appears. Obviously, a more sophisticated approach needs to be followed.

Example 3.3 illustrates that the evaluation process should not be stopped unless the set of all atoms that contain values of order 0 (namely F_0, T_0 stabilizes. By generalizing this idea, we can come up with a proof procedure that guarantees the correct calculation of the minimum model M_{P} and at the same time termination in a finite number of steps. This procedure can informally be described as follows. As a first approximation to M_{P}, we start with the interpretation that assigns to every atom of B_{P} the value F_0 (as already mentioned, this interpretation is denoted by $\emptyset$). We start iterating the T_{P} on $\emptyset$ until both the set of atoms that have a F_0 value and the set of atoms having a T_0 value, stabilize (as we are going to discuss below, this is guaranteed to happen in a finite number of steps). We keep all these atoms whose values have stabilized and reset the values of all remaining atoms to the next false value (namely F_1). The procedure is repeated until the F_1 and T_1 values stabilize, and we reset the remaining atoms to a value equal to F_2, and so on. As we repeat this procedure, two possible outcomes can eventually happen: one possibility is that, since the Herbrand Base of P is finite, the procedure will terminate after all atoms have stabilized to some value different than 0; in this case, we have reached the desired fixed point. The second possible outcome that can happen is that at the k-th stage of the bottom-up evaluation, this process will not produce any new atoms having F_k or T_k values. At this point, we stop the iterations and reset the truth value of all the atoms that

have not yet received a value, to 0. We present this algorithm in a more compact way in Figure 3.2.

Example 3.4. *Continuing Example 3.3, after we have reached the interpretation:*

$$\{(\mathtt{p}, F_3), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

further iterations of the T_{P} will not affect the values of order 0. In other words, we have reached a fixed point with respect to the atoms that have values of order 0. We reset the value of the variable $\mathtt{p}$ to F_1 getting the interpretation:

$$\{(\mathtt{p}, F_1), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

One further iteration of the T_{P} operator gives:

$$\{(\mathtt{p}, F_2), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

The above interpretation contains a gap at order 1. We stop the iterations and reset the value of $\mathtt{p}$ to 0, getting:

$$\{(\mathtt{p}, 0), (\mathtt{q}, T_0), (\mathtt{r}, T_0), (\mathtt{s}, T_0)\}$$

It can be easily seen that this is the least Herbrand model of the program.

Example 3.5. *Consider the following more complex example:*

$$\mathtt{r} \leftarrow \epsilon\,\mathtt{r}.$$

$$\begin{array}{l}
\mathtt{q}_0 \leftarrow \mathtt{p}_{0,0} \wedge \mathtt{p}_{0,1} \wedge \cdots \wedge \mathtt{p}_{0,n}. \\
\mathtt{q}_1 \leftarrow \mathtt{p}_{1,0} \wedge \mathtt{p}_{1,1} \wedge \cdots \wedge \mathtt{p}_{1,n}. \\
\qquad\qquad \cdots \\
\mathtt{q}_n \leftarrow \mathtt{p}_{n,0} \wedge \mathtt{p}_{n,1} \wedge \cdots \wedge \mathtt{p}_{n,n}.
\end{array}$$

$$\begin{array}{lll}
\mathtt{p}_{0,1} \leftarrow \mathtt{p}_{0,0}. & \mathtt{p}_{0,2} \leftarrow \mathtt{p}_{0,1}. & \cdots \quad \mathtt{p}_{0,n} \leftarrow \mathtt{p}_{0,n-1}. \\
\mathtt{p}_{1,1} \leftarrow \mathtt{p}_{1,0}. & \mathtt{p}_{1,2} \leftarrow \mathtt{p}_{1,1}. & \cdots \quad \mathtt{p}_{1,n} \leftarrow \mathtt{p}_{1,n-1}. \\
\cdots & \cdots & \cdots \\
\mathtt{p}_{n,1} \leftarrow \mathtt{p}_{n,0}. & \mathtt{p}_{n,2} \leftarrow \mathtt{p}_{n,1}. & \cdots \quad \mathtt{p}_{n,n} \leftarrow \mathtt{p}_{n,n-1}.
\end{array}$$

$$\begin{array}{l}
\mathtt{p}_{0,0}. \\
\mathtt{p}_{1,0} \leftarrow \mathtt{q}_0 \wedge \epsilon\,\mathtt{q}_0. \\
\mathtt{p}_{2,0} \leftarrow \mathtt{q}_1 \wedge \epsilon\,\mathtt{q}_1. \\
\qquad\qquad \cdots \\
\mathtt{p}_{n,0} \leftarrow \mathtt{q}_{n-1} \wedge \epsilon\,\mathtt{q}_{n-1}.
\end{array}$$

The evaluation process proceeds as follows. First, we begin with the interpretation $\emptyset$, which assigns F_0 to all atoms:

$$\left\{ \begin{array}{c} (\mathtt{r}, F_0), \quad (\mathtt{q}_0, F_0), \ldots, (\mathtt{q}_n, F_0), \\ (\mathtt{p}_{0,0}, F_0), \ldots, (\mathtt{p}_{0,n}, F_0), \ldots, (\mathtt{p}_{n,0}, F_0), \ldots, (\mathtt{p}_{n,n}, F_0), \end{array} \right\}$$

In the first T_{P} iteration, the atom $\mathtt{p}_{0,0}$ receives the value T_0 (due to the existence of the fact $\mathtt{p}_{0,0}$.); the i-th T_{P} iteration (where $1 < i \leq n$) assigns T_0 to $\mathtt{p}_{0,\mathtt{i}}$ (due to the existence of the rule $\mathtt{p}_{0,\mathtt{i}} \leftarrow \mathtt{p}_{0,\mathtt{i}-1}$.); the $(n+1)$-th T_{P} iteration assigns T_0 to $\mathtt{q}_0$ as well (due to the

existence of the rule $\mathtt{q_0} \leftarrow \mathtt{p_{0,0}} \wedge \ldots \wedge \mathtt{p_{0,n}}$*.); and after that, the atoms that receive the values* F_0 *and* T_0 *stabilize, therefore we get the following interpretation:*

$$\left\{ \begin{array}{c} (\mathtt{r}, F_{n+2}), \quad (\mathtt{q_0}, T_0), (\mathtt{q_1}, F_0), \ldots, (\mathtt{q_n}, F_0), \\ (\mathtt{p_{0,0}}, T_0), \ldots, (\mathtt{p_{0,n}}, T_0), \quad (\mathtt{p_{1,0}}, T_1), (\mathtt{p_{1,1}}, F_0), \ldots, (\mathtt{p_{1,n}}, F_0), \\ (\mathtt{p_{2,0}}, F_0), \ldots, (\mathtt{p_{2,n}}, F_0), \ldots, (\mathtt{p_{n,0}}, F_0), \ldots, (\mathtt{p_{n,n}}, F_0), \end{array} \right\}$$

Now, we reset the atoms that contain values other than F_0 *and* T_0 *(namely the atoms* $\mathtt{p_{1,0}}$ *and* $\mathtt{r}$*) to the value* F_1*. We iterate the* T_{P} *as previously; at first, the atom* $\mathtt{p_{1,0}}$ *receives the value* T_1 *(due to the existence of the rule* $\mathtt{p_{1,0}} \leftarrow \mathtt{q_0} \wedge \epsilon\, \mathtt{q_0}$*); then the atoms* $\mathtt{p_{1,0}}, \ldots \mathtt{p_{1,n}}$ *and* $\mathtt{q_1}$ *receive the value* T_1 *as previously; finally the values* F_1 *and* T_1 *stabilize and therefore we result to the following interpretation:*

$$\left\{ \begin{array}{c} (\mathtt{r}, F_{n+3}), \quad (\mathtt{q_0}, T_0), (\mathtt{q_1}, T_0), (\mathtt{q_2}, F_0), \ldots, (\mathtt{q_n}, F_0), \\ (\mathtt{p_{0,0}}, T_0), \ldots, (\mathtt{p_{0,n}}, T_0), \quad (\mathtt{p_{1,0}}, T_1), \ldots, (\mathtt{p_{1,n}}, T_1), \\ (\mathtt{p_{2,0}}, T_2), (\mathtt{p_{2,1}}, F_0), \ldots, (\mathtt{p_{2,n}}, F_0), \ldots, (\mathtt{p_{n,0}}, F_0), \ldots, (\mathtt{p_{n,n}}, F_0), \end{array} \right\}$$

This process is repeated until all atoms of the form $\mathtt{p_{i,j}}$ *and* $\mathtt{q_i}$*, for all* $0 \leq i, j \leq n$ *receive a positive truth value. In particular, in the step where the values of order* i *stabilize, the atoms* $\mathtt{p_{i,0}}, \ldots \mathtt{p_{i,n}}$ *and* $\mathtt{q_i}$ *receive the value* T_1*. When the values* F_n, T_n *stabilize, we reach to this interpretation:*

$$\left\{ \begin{array}{c} (\mathtt{r}, F_{2n+2}), \quad (\mathtt{q_0}, T_0), (\mathtt{q_1}, T_1), \ldots, (\mathtt{q_n}, T_n), \\ (\mathtt{p_{0,0}}, T_0), \ldots, (\mathtt{p_{0,n}}, T_0), \ldots, (\mathtt{p_{n,0}}, T_n), \ldots, (\mathtt{p_{n,n}}, T_n), \end{array} \right\}$$

If we repeat the process one more time, the resulting interpretation will contain a gap at order $n+1$*. As a result, we can set the value* 0 *in* $\mathtt{r}$*:*

$$\left\{ \begin{array}{c} (\mathtt{r}, 0), \quad (\mathtt{q_0}, T_0), (\mathtt{q_1}, T_1), \ldots, (\mathtt{q_n}, T_n), \\ (\mathtt{p_{0,0}}, T_0), \ldots, (\mathtt{p_{0,n}}, T_0), \ldots, (\mathtt{p_{n,0}}, T_n), \ldots, (\mathtt{p_{n,n}}, T_n), \end{array} \right\}$$

Notice that the final interpretation is the minimum model of the program.

As we illustrate in Proposition 3.2, the Bottom-up Evaluation of Figure 3.2 is a terminating algorithm for $\{\epsilon, \wedge\}$-programs. Moreover, we will discuss the correctness of this algorithm in the next subsection.

Proposition 3.2. *The algorithm of Figure 3.2 terminates in a finite number of steps for any given* $\{\epsilon, \wedge\}$*-program* P*.*

Proof. Notice that the algorithm consists of two loops, that is the inner loop (lines 6-9) and the outer loop (lines 4-18). In order to prove that the algorithm terminates, we have to prove that both loops terminate. Suppose that the inner loop is infinite. Since B_{P} is finite, the set of atoms that will eventually receive truth values of order n is finite. Therefore, there must be some atom A $\in B_{\mathsf{P}}$ such that it receives T_n or F_n in an infinite number of steps. Since the only available operators are the operators ϵ and $\wedge$, we conclude P must contain an infinite number of clauses (contradiction). Therefore, the inner loop is finite. The outer loop is also finite because the inner loop is finite and the set S remains finite (it is initialized with the elements of B_{P} and it always decreases). □

3.3.3 Correctness of Terminating Bottom-up Evaluation

In this subsection, we provide a correctness proof for the terminating bottom-up evaluation algorithm of Figure 3.2.

Theorem 3.1. *The algorithm of Figure 3.2 correctly computes the least model M_{P} of any given $\{\epsilon, \wedge\}$-program* P*.*

The intuition behind the proof can be described as follows. It can be demonstrated that the T_{P} operator of a $\{\epsilon, \wedge\}$-program P has the property of being *k-monotonic for all $k \in \mathbb{N}$*. The notion of k-monotonicity was initially described in order to formalize the model-theoretic semantics of well-founded negation [64] and it was later extensively studied in the case of a general fixed-point theorem for non-monotonic functions [26]. Roughly speaking, the fact that T_{P} is k-monotonic means that when given two interpretations I and J that agree on the values of atoms of order less than k and I is "less than" J in the truth values of level k, then $T_{\mathsf{P}}(I)$ is "less than" $T_{\mathsf{P}}(J)$ in the truth values of level k. Based on this property we demonstrate that the above procedure produces at each inner step the same truth values as those that exist in M_{P}. Since the algorithm correctly computes each level of truth values and since we know that the minimum model of any $\{\epsilon, \wedge\}$-program does not contain a gap, we realize that as soon as a gap is found by the above procedure, this implies that the atoms that have a value above the gap, will have the value 0 in the minimum model M_{P}.

In order to establish Theorem 3.1 (i.e., in order to demonstrate the correctness of the bottom-up procedure), we need some background material from [64] and [26]. In the rest of this section, when we refer to "a program" we mean a "$\{\epsilon, \wedge\}$-program".

Definition 3.3. *[64] Let* P *be a program, I an interpretation of* P*, $v \in \mathbb{V}$ and $n \in \mathbb{N}$. We define: $I \parallel v = \{\mathsf{A} \in B_{\mathsf{P}} \mid I(\mathsf{A}) = v\}$.*

The following relations on interpretations are also needed:

Definition 3.4. *[64] Let I and J be interpretations of a program* P *and $n \in \mathbb{N}$. We write $I =_n J$, if for all $k \leq n$, $I \parallel T_k = J \parallel T_k$ and $I \parallel F_k = J \parallel F_k$.*

Definition 3.5. *[64] Let I and J be interpretations of a program* P *and $n \in \mathbb{N}$. We write $I \sqsubset_n J$, if for all $k < n$, $I =_k J$ and either $I \parallel T_n \subset J \parallel T_n$ and $I \parallel F_n \supseteq J \parallel F_n$, or $I \parallel T_n \subseteq J \parallel T_n$ and $I \parallel F_n \supset J \parallel F_n$. We write $I \sqsubseteq_n J$ if $I =_n J$ or $I \sqsubset_n J$. We write $I \sqsubset J$, if there exists $n \in \mathbb{N}$ such that $I \sqsubset_n J$. We write $I \sqsubseteq J$ if either $I = J$ or $I \sqsubset J$.*

Recall that by $\mathcal{I}_{\mathsf{P}}$ we denote the set of infinite-valued interpretations of a program P. It is easy to see [64] that the relation $\sqsubseteq$ on $\mathcal{I}_{\mathsf{P}}$ is a partial order (i.e., it is reflexive, transitive and antisymmetric).

Definition 3.6. *[26] Let* P *be a program and let $n \in \mathbb{N}$. Let $\mathcal{X} \subseteq \mathcal{I}_{\mathsf{P}}$ be a non-empty set of interpretations of* P *and assume that for all $I, J \in \mathcal{X}$ it holds that $I =_k J$ for all $k < n$. Let I be an arbitrary element of $\mathcal{X}$. We define:*

$$(\bigsqcup_n \mathcal{X})(\mathsf{A}) = \begin{cases} I(\mathsf{A}), & \textit{if } \mathrm{ord}(I(\mathsf{A})) < n \\ T_n, & \textit{if there exists } J \in \mathcal{X} \textit{ with } J(\mathsf{A}) = T_n \\ F_n, & \textit{if for all } J \in \mathcal{X},\ J(\mathsf{A}) = F_n \\ F_{n+1}, & \textit{otherwise.} \end{cases}$$

Notice that the first case in the above definition is well-defined because all the elements of $\mathcal{X}$ agree on the truth values of order less than n.

Definition 3.7. *[26] Let* P *be a program and let* $n \in \mathbb{N}$*. Then,* T_{P} *is called* n*-monotonic if for all* $I, J \in \mathcal{I}_{\mathsf{P}}$*, if* $I \sqsubseteq_n J$ *then* $T_{\mathsf{P}}(I) \sqsubseteq_n T_{\mathsf{P}}(J)$*. Moreover,* T_{P} *is called* n*-continuous if it is* n*-monotonic and for all sequences* $(I_k)_{k\geq 0}$ *of interpretations such that for all* $k \geq 0$*,* $I_k \sqsubseteq_n I_{k+1}$*, it holds that* $T_{\mathsf{P}}(\bigsqcup_n \{I_k : k \geq 0\}) =_n \bigsqcup_n \{T_{\mathsf{P}}(I_k) : k \geq 0\}$*.*

Based on the above material, we can now prove the following proposition that concerns $\{\epsilon, \wedge\}$-programs:

Proposition 3.3. *The* T_{P} *operator of a* $\{\epsilon, \wedge\}$*-program* P *is* n*-continuous for all* $n \in \mathbb{N}$*.*

Proof. By Corollary 7.8 and Lemma 7.12 of [26], if the T_{P} operator of a program uses in its body only operations that are n*-continuous for all* $n \in \mathbb{N}$, then T_{P} is also n-continuous for all $n \in \mathbb{N}$; the notion of n-continuity for operators is similar to that defined above for T_{P} (see Subsection 5.2 of [26] for the definition of the appropriate orderings on the set $\mathbb{V}$ and Definition 6.3 of the same article for the definition of n-continuity of functions over the set $\mathbb{V}$). The operator $\wedge$ is n-continuous for all $n \in \mathbb{N}$ (see [26][Lemma 7.10]) and it is a straightforward task to establish that ϵ also has the same property. Therefore, the T_{P} operator of $\{\epsilon, \wedge\}$-programs is n-continuous for all $n \in \mathbb{N}$. □

When the T_{P} operator is n-continuous for all $n \in \mathbb{N}$, we can find its least fixed point with respect to $\sqsubseteq$ with a construction that first stabilizes the order 0 values, then the order 1, and so on. This procedure is formally expressed by a construction described by the following theorem (which specializes to our setting the much more general Theorem 6.6 (see also Remark 6.5) of the article [26]). Actually, the limit of this construction is the least (with respect to the relation $\sqsubseteq$) among all models of P (see Theorem 4 of [25]).

Theorem 3.2. *[26] Let* P *be a* $\{\epsilon, \wedge\}$*-program. Consider the following doubly indexed sequence of interpretations of* P*:*

$$\begin{array}{lcl} I_{0,0} & = & \emptyset \\ I_{n,m+1} & = & T_{\mathsf{P}}(I_{n,m}) \\ I_{n,\omega} & = & \bigsqcup_n \{I_{n,m} : m \in \mathbb{N}\} \\ I_{n+1,0} & = & I_{n,\omega} \end{array}$$

Define the following infinite-valued interpretation N_{P} *of* P*:*

$$N_{\mathsf{P}}(\mathsf{A}) = \begin{cases} T_n, & \textit{if } I_{n,\omega}(\mathsf{A}) = T_n \\ F_n, & \textit{if } I_{n,\omega}(\mathsf{A}) = F_n \\ 0, & \textit{otherwise} \end{cases}$$

Then, N_{P} *is the least fixed point of* T_{P} *with respect to* $\sqsubseteq$*. Moreover,* N_{P} *is the least model of* P *with respect to* $\sqsubseteq$*.*

Notice that the construction of the interpretation N_{P} in the above theorem is actually identical to the construction performed by the bottom-up algorithm given in Figure 3.2. What remains to be shown is that N_{P} is exactly the same interpretation as M_{P}:

Lemma 3.1. *Let* P *be a* $\{\epsilon, \wedge\}$*-program and let* M_{P} *and* N_{P} *be the least fixed points of* T_{P} *with respect to* $\leq$ *and* $\sqsubseteq$ *respectively. Then,* $M_{\mathsf{P}} = N_{\mathsf{P}}$*.*

Proof. By Theorem 2.2, it holds that $M_{\mathsf{P}} \leq M$ for all models M of P. Since by Theorem 3.2 N_{P} is also a model of P, we get that $M_{\mathsf{P}} \leq N_{\mathsf{P}}$. However, this implies (see Lemma 4.13 of [64]) that $M_{\mathsf{P}} \sqsubseteq N_{\mathsf{P}}$. Since N_{P} is the least model of P with respect to $\sqsubseteq$, we also get that $N_{\mathsf{P}} \sqsubseteq M_{\mathsf{P}}$, and therefore $N_{\mathsf{P}} = M_{\mathsf{P}}$. □

The above discussion together with the above lemma implies that Theorem 3.1 holds.

3.4 Implementation

An implementation of the bottom-up technique that we proposed in this chapter is available [82]. This implementation uses as its basis the bottom-up Datalog system IRIS[2]. This implementation supports the preference operators $\wedge$, $\vee$, ϵ, opt, and alt and implements a Terminating Naive Bottom-up evaluation strategy (similar to that of Figure 3.2) and a Terminating Semi-Naive Bottom-up evaluation strategy, which extends the standard Semi-Naive evaluation of Datalog (cf. [82] for details).

Apart from the bottom-up implementation, we have implemented a transformation[3] of the PrefLog programs [72]. The basic idea of the transformation is to introduce an additional argument in every atom of the original program. This additional argument corresponds to the truth value that M_{P} assings in the original atom. For instance, in the context of $\{\epsilon, \wedge\}$-programs, for every propositional atom A we create a unary predicate A, such that the query $\leftarrow$ A($\mathtt{V}$) returns $\mathtt{V} = v$ if M_{P}(A) $= v$. Roughly speaking, such a Prolog translation consists of three main parts; first, a set of Prolog rules, obtained directly from the rules of the original program, that is used to compute the corresponding truth value; second, a set of Prolog rules that are used for obtaining the correct truth value for each atom among all values computed by the rules of the first part; and finally, a set of Prolog rules that correspond to the preference operators that are present in the original program. The implementation of the transformation is approximately 350 lines of Prolog code and is realized for the XSB system[4].

3.5 Summary

In this chapter, we focused on an evaluation technique for PrefLog programs. Since general PrefLog programs naturally introduce two levels of infinity (i.e., an infinite set of atoms can obtain an infinite set of values) we reduced our focus in a large and well-behaved subset of function-free fragment of PrefLog, denoted as "$\{\epsilon, \wedge\}$-programs", which is the preferential analogue of Datalog.

We demonstrated that terminating bottom-up evaluation can be performed for a $\{\epsilon, \wedge\}$-programs. This result is not obvious: despite the fact that the Herbrand Base of the programs we consider is finite, an atom may obtain an infinity of truth values during bottom-up evaluation, resulting in possible non-termination. For this reason, we devised a bottom-up execution strategy in which the atoms converge in levels until either the

[2] cf. `http://www.iris-reasoner.org/`
[3] cf. `http://bitbucket.org/antru/preflog`
[4] cf. `http://xsb.sourceforge.net/`

atoms are exhausted or a “gap” in the produced set of atoms is found (which signals that the iterations should stop). We demonstrated that the interpretation produced by this procedure coincides with the least model of the program.

4. EXPRESSING PREFERENCES USING HIGHER-ORDER LOGIC PROGRAMMING

In this chapter, we propose the use of higher-order logic programming as a logical framework for expressing qualitative preferences. Our approach extends a seminal work by Chomicki [17, 80] and provides a uniform framework in which relations, preferences between tuples, preferences between sets of tuples and operations on preferences are expressed in the same, higher-order logic programming language.

4.1 Overview

The starting point of our research is an influential proposal by J. Chomicki [17] for representing qualitative preferences in the context of relational database systems. Chomicki's approach is based on the following two ideas:

- Preferences between tuples of a database relation are specified using binary *preference relations*; these relations are defined using first-order formulas, called *preference formulas*.
- A new relational algebra operator is introduced. This operator is called *winnow* and takes two parameters; a database relation and a preference formula. The winnow operator selects from its input relation the most preferred tuples according to the given preference formula.

The approach advocated by Chomicki, despite groundbreaking, has certain limitations (some of which are recognized and discussed in his paper [17]). First of all, in this framework, only *intrinsic* preference formulas can be defined, namely formulas that establish the preference relation between two tuples solely on the basis of the values occurring in these tuples. Second, the preference relations and the preference queries are expressed in two different languages, namely, first-order logic and SQL extended with the winnow operator, which makes the approach less uniform. Finally, there is no way to define *directly* other operators apart from winnow (such as, for example, an operator that returns the second-best tuples from a given relation according to a preference formula).

In this chapter, we propose the use of higher-order logic programming as a logical framework that remedies all the above deficiencies. The key idea behind our proposal is that *since preferences are relations, and since we need to define operators over relations (such as winnow), and some times even preferences over sets, a higher-order language can offer increased representation capabilities*. In particular, we demonstrate that higher-order logic programming can be used to express both intrinsic and extrinsic preference formulas, it can represent both preference relations as-well-as queries, and it can be used to define a variety of interesting alternative operators beyond winnow.

We argue that higher-order logic programming is a very expressive framework for representing and manipulating qualitative preferences. A significant advantage of our approach is that preference formulas as-well-as operators that are parameterized with such formulas can be expressed in the same language. Moreover, the seemingly more demanding case of *preferences over sets* [80] can be handled without extra notational overhead because preferences over sets are essentially second-order relations and can, therefore, be encoded easily in our higher-order language. Finally, we identify a new and

significant (in our opinion) application area for higher-order logic programming. Higher-order logic programming has been around for many years [15, 52, 77] and it has recently been given a standard denotational and proof-theoretic semantics [10, 11, 12, 61]. We feel that higher-order logic programming deserves to be further developed and used since it extends in an elegant way the framework of classical logic programming.

The rest of the chapter is organized as follows: in Section 4.2 we present the main concepts behind the work of J. Chomicki [17] and its extension for preferences between sets [80]; in Section 4.3 we outline the basic notions regarding higher-order logic programming; in Section 4.4 we demonstrate how higher-order logic programming can be used to concisely represent and manipulate preferences and how it bypasses the main shortcomings of the framework of Chomicki [17]; in Section 4.5 we demonstrate that the proposed higher-order logic programming approach can also be used to elegantly represent preferences over sets of tuples; and finally, we close with a brief summary of the chapter.

4.2 Qualitative Preferences and Databases

In this section, we focus on preference representation in a relational (SQL) database context. We present the approach proposed in the seminal work of Chomicki [17]. Originally, this framework allows expression of preferences between tuples, but later was extended in order to express preferences over *sets of tuples* [80]. This brief presentation begins with the tuple preferences case, continues with examples of common qualitative preference compositions and closes with the set preferences extension of this framework.

4.2.1 Preferences over Tuples

In this subsection, we present the framework of Chomicki [17], which is used for expressing qualitative preferences between tuples in databases. This is done by extending relational algebra with *preference formulas* and the *winnow* operator.

Qualitative preferences in a relational database context are defined using a binary preference relation among database tuples:

Definition 4.1. *Given a relation schema $R(A_1, \ldots, A_n)$ such that U_i, $1 \leq i \leq n$, is the domain of the attribute A_i, a relation $\succ$ is a preference relation over R if it is a subset of $(U_1 \times \cdots \times U_n) \times (U_1 \times \cdots \times U_n)$. A tuple t_1 is said to be preferred from t_2, if it holds $t_2 \succ t_1$.*

In Chomicki's framework [17] preference relations are defined using first-order preference formulas. In the following definition, by "built-in predicates" we mean any standard SQL predicate such as equality, inequality, arithmetic comparison operations, and so on.

Definition 4.2. *Let t_1, t_2 denote tuples of a given database relation. A preference formula $C(t_1, t_2)$ is a first-order formula defining a preference relation $\succ_C$ in the standard sense, namely $t_1 \succ_C t_2 \iff C(t_1, t_2)$. An intrinsic preference formula (or ipf) is a preference formula that uses only built-in predicates.*

Intuitively, an *intrinsic* preference formula is one that establishes the preference relation between two tuples solely on the basis of the values occurring in these tuples. *Extrinsic* preference formulas may use not only built-in predicates, but also other constructs (such as for example database relations, properties of the relations from which the tuples have

Table 4.1: A simple `movie` relation.

ID	Name	Director	Genre	Runtime (min)	Rating
m_1	The Godfather	F. F. Coppola	drama	175	9.2
m_2	The Green Mile	F. Darabont	drama	189	8.5
m_3	Goodfellas	M. Scorsese	drama	146	8.7
m_4	The Big Lebowski	Coen Brothers	comedy	117	8.2
m_5	Forrest Gump	R. Zemeckis	comedy	142	8.8
m_6	Inception	C. Nolan	sci-fi	148	8.8

been selected, and so on). For example, if we prefer every tuple of a given relation r over every tuple of another relation s, then this is an extrinsic preference because it depends also on the origin of the tuples and not only on their attributes. The framework of Chomicki [17] focuses almost exclusively on intrinsic preference formulas.

Example 4.1. *Consider the* `movie(ID,Name,Director,Genre,Runtime,Rating)` *relation illustrated in Table 4.1. Now, suppose that we want to express the following preference:*

> *"Prefer one movie over another iff their genres are the same and the rating of the first is higher".*

This can be defined by the following ipf formula C_1:

$$(i, n, d, g, t, r) \succ_{C_1} (i', n', d', g', t', r') \equiv (g = g') \wedge (r > r')$$

The above preference means that, for example, "The Godfather" is preferred from "Goodfellas" because they have the same genre but the former has a higher rating than the latter. In addition, movies that are of different genre (e.g. "Forrest Gump" and "Inception") are incomparable. As another example of the same relation, consider the following preference:

> *"Prefer a movie from another one if the former lasts for less than or equal to 150 minutes while the latter does not".*

The corresponding preference relation $\succ_{C_2}$ can be defined by the following ipf formula C_2:

$$(i, n, d, g, t, r) \succ_{C_2} (i', n', d', g', t', r') \equiv (t \leq 150) \wedge (t' > 150)$$

According to the preference relation $\succ_{C_2}$, every movie with a runtime that is less than 150 minutes is preferred, e.g. "Forrest Gump" is preferred from "The Godfather". Moreover, all such movies are incomparable e.g. "The Big Lebowski" is equally preferred to "Inception". As a final example, consider the following preference:

> *"Prefer drama movies over movies of all other genres".*

The corresponding preference relation $\succ_{C_3}$ can be defined by the following ipf formula C_3:

$$(i, n, d, g, t, r) \succ_{C_3} (i', n', d', g', t', r') \equiv (g = \text{"drama"}) \wedge (g' \neq \text{"drama"})$$

According to $\succ_{C_3}$, the most preferred movies are "The Godfather", "The Green Mile" and "Goodfellas".

Table 4.2: Result sets of simple winnow queries.

(a) The result of the query $w_{C_1}(\texttt{movie})$ using the `movie` relation of Table 4.1.

ID	Name	Director	Genre	Runtime (min)	Rating
m_1	The Godfather	F. F. Coppola	drama	175	9.2
m_5	Forrest Gump	R. Zemeckis	comedy	142	8.8
m_6	Inception	C. Nolan	sci-fi	148	8.8

(b) The result of the query $w_{C_2}(\texttt{movie})$ using the `movie` relation of Table 4.1.

ID	Name	Director	Genre	Runtime (min)	Rating
m_3	Goodfellas	M. Scorsese	drama	146	8.7
m_4	The Big Lebowski	Coen Brothers	comedy	117	8.2
m_5	Forrest Gump	R. Zemeckis	comedy	142	8.8
m_6	Inception	C. Nolan	sci-fi	148	8.8

(c) The result of the query $w_{C_3}(\texttt{movie})$ using the `movie` relation of Table 4.1.

ID	Name	Director	Genre	Runtime (min)	Rating
m_1	The Godfather	F. F. Coppola	drama	175	9.2
m_2	The Green Mile	F. Darabont	drama	189	8.5
m_3	Goodfellas	M. Scorsese	drama	146	8.7

In the following, we discuss the *winnow* operator, the second basic component of the framework of Chomicki [17]. In order to select the best (i.e., most preferred) tuples from a given relation r based on a preference formula C, relational algebra can be enriched with a new operator called *winnow*. The formal definition of this new preference operator is the following:

Definition 4.3. *Let r be a relation and let C be a preference formula defining a preference relation $\succ_C$. The winnow operator is defined as*

$$w_C(r) = \{t \in r : \neg\exists t' \in r \textit{ such that } t' \succ_C t\}$$

A *preference query* is defined as a relational query that contains at least one occurrence of the winnow operator. In the following example, we illustrate three preference queries that are parameterized using preference relations that were defined previously.

Example 4.2. *Continuing Example 4.1, consider the simple preference queries $w_{C_1}(\texttt{movie})$, $w_{C_2}(\texttt{movie})$ and $w_{C_3}(\texttt{movie})$. These queries will return the best tuples according to the preference formulas that are being invoked.*

Regarding $w_{C_1}(\texttt{movie})$, the winnow query should return one movie for every available genre, and each genre should be "represented" with the movie with the highest rating. Regarding $w_{C_2}(\texttt{movie})$, the winnow query should return all movies that their duration is less than or equal to 150 minutes. Finally, regarding $w_{C_3}(\texttt{movie})$, the query should return all drama movies.

The result sets are illustrated in Table 4.2a, Table 4.2b, and Table 4.2c respectively.

In our framework, we are able to define alternative operators beyond winnow, and therefore our queries can be more general.

4.2.2 Composition of Preference Relations

In this subsection, we present some common preference compositions and we show how these compositions can be expressed using Chomicki's framework [17].

Preference relations can be combined in order to form more complex ones. Being binary relations, preference relations can be composed in many ways; examples include the use of standard set-theoretic operations. In the context of Chomicki [17], in order to compose two preference relations we compose the corresponding formulas, creating in this way more complicated ones. Three examples of such composition operations are the following:

- The *Boolean composition* of two preference relations (such as union, intersection and difference) can be captured by Boolean operations on the corresponding preference formulas. For example, the preference relation $\succ_{C_1 \wedge C_2} = \succ_{C_1} \cap \succ_{C_2}$ is captured by the formula:

$$t_1 \succ_{C_1 \wedge C_2} t_2 \equiv (t_1 \succ_{C_1} t_2) \wedge (t_1 \succ_{C_2} t_2).$$

- The *Prioritized composition* $\succ_{C_1 \triangleright C_2}$ of two preference relations C_1 and C_2 has the intuitive meaning of "prefer according to C_2 unless C_1 is applicable", and can be defined as follows:

$$t_1 \succ_{C_1 \triangleright C_2} t_2 \equiv (t_1 \succ_{C_1} t_2) \vee ((t_1 \sim_{C_1} t_2) \wedge (t_1 \succ_{C_2} t_2)),$$

  where $\sim_C$ is the *indifference relation* of a preference relation $\succ_C$ defined by the following formula:

$$t_1 \sim_C t_2 \equiv \neg(t_1 \succ_C t_2) \wedge \neg(t_2 \succ_C t_1)$$

- The *Pareto composition* $\succ_{C_1 \otimes C_2}$ of two preference relations C_1 and C_2 has the intuitive meaning of "prefer according to both C_1 and C_2 with equal importance", and can be defined as follows:

$$t_1 \succ_{C_1 \otimes C_2} t_2 \equiv ((t_1 \succ_{C_1} t_2) \wedge (t_2 \not\succ_{C_2} t_1)) \vee ((t_1 \succ_{C_2} t_2) \wedge (t_2 \not\succ_{C_1} t_1)),$$

  where $t_1 \not\succ_C t_2 \equiv \neg(t_1 \succ_C t_2)$.

The following examples illustrate the use of such operators.

Example 4.3. *Consider the* `movie` *relation of Example 4.1. In this example, we consider the Prioritized composition.*

Suppose that our preference for movies is the prioritized composition $C_2 \triangleright C_1$ of C_2 and C_1. This means that we have a primary preference for the movies that have a duration that is less than or equal to 150 minutes, and a secondary preference for the high rating (among the ones that have the same genre).

A process for discovering the best tuples according to $\succ_{C_2 \triangleright C_1}$ begins by finding the best tuples according to $\succ_{C_2}$, i.e., all movies that have a duration less than or equal to 150

minutes (cf. Table 4.2b). Now, among these incomparable tuples, we filter out all less preferred tuples according to $\succ_{C_1}$. That is, we remove m_4 since there exists a comedy movie (namely m_5) which has a higher rating. Thus, the preference query $w_{C_2 \triangleright C_1}(\texttt{movie})$ returns the result set shown in Table 4.3a.

Compare now this result set with that of $w_{C_1}(\texttt{movie})$ (cf. Table 4.2a). Notice that our primary preference for the duration of the movies has excluded some highly rated ones.

Example 4.4. *Consider the* `movie` *relation of Example 4.1. In this example we consider the Pareto composition.*

Suppose that our preference for movies is the Pareto composition $C_2 \otimes C_3$ of C_2 and C_3. This means that we prefer both movies that have a duration less than or equal to 150 minutes and dramas, but these preference criteria are now treated with equal importance.

Notice that the `movie` *relation of Table 4.1 contains one movie that satisfies both criteria; a drama movie with a runtime less than 150 minutes (i.e., the movie "Goodfellas"). Therefore, the result set for the preference query $w_{C_2 \otimes C_3}(\texttt{movie})$ returns only this movie (cf. Table 4.3b).*

Now, suppose that we remove the m_3 tuple (i.e., the movie "Goodfellas") from the original `movie` *relation. Now, there does not exist any movie that satisfies both preference criteria. Therefore, since we equally prefer drama movies and movies that have a duration less than or equal to 150 minutes, the preference query $w_{C_2 \otimes C_3}(\texttt{movie} \setminus \{m_3\})$ returns all movies that satisfy either C_2 or C_3 (cf. Table 4.3c).*

Through higher-order logic programming, we can directly define generic composition operators that can be applied to compose arbitrary preference relations (without the need to manipulate preference formulas).

4.2.3 Preferences over Sets

In this subsection, we discuss how the framework of Chomicki [17] can be extended so as to handle preferences between sets of tuples.

There exist many applications that require expressing preferences over sets of tuples. This case, which is more demanding and general than the case of preferences over tuples, was considered in a recent work by Zhang and Chomicki [80]. In this work, the sets that are involved in preference queries are assumed to be of fixed cardinality, an assumption which we will also adopt in our approach. The main ideas behind this proposal can be illustrated by a simple example.

Example 4.5. *Assume that I want to watch three movies, and I want to select them, based on some given preferences, out of our usual movie table. My preferences are the following:*

- *C_1: Prefer that the sum of the ratings of the movies is maximized.*
- *C_2: Prefer to watch at least one comedy.*
- *C_3: Prioritize C_2 over C_1.*

The above preferences are defined over three-element sets of movies and therefore a new method seems to be required for their representation.

Table 4.3: Result sets of composite winnow queries.

(a) The result of the query $w_{C_2 \triangleright C_1}(\texttt{movie})$ using the `movie` relation of Table 4.1.

ID	Name	Director	Genre	Runtime (min)	Rating
m_3	Goodfellas	M. Scorsese	drama	146	8.7
m_5	Forrest Gump	R. Zemeckis	comedy	142	8.8
m_6	Inception	C. Nolan	sci-fi	148	8.8

(b) The result of the query $w_{C_2 \otimes C_1}(\texttt{movie})$ using the `movie` relation of Table 4.1.

ID	Name	Director	Genre	Runtime (min)	Rating
m_3	Goodfellas	M. Scorsese	drama	146	8.7

(c) The result of the query $w_{C_2 \otimes C_1}(\texttt{movie} \setminus \{m_3\})$, i.e., using the `movie` relation of Table 4.1 without the "Goodfellas" movie tuple.

ID	Name	Director	Genre	Runtime (min)	Rating
m_1	The Godfather	F. F. Coppola	drama	175	9.2
m_2	The Green Mile	F. Darabont	drama	189	8.5
m_4	The Big Lebowski	Coen Brothers	comedy	117	8.2
m_5	Forrest Gump	R. Zemeckis	comedy	142	8.8
m_6	Inception	C. Nolan	sci-fi	148	8.8

Zhang and Chomicki [80] define a specialized approach for treating preferences over sets. First, they remark that for each preference there exist one or more "quantities of interest", which they call *features*. For example, for C_1 the relevant feature is the sum of the ratings of the movies, for C_2 the relevant feature is the number of comedies, and for C_3 both previous features are relevant. Given a set, we can construct a vector of all features based on the user preferences, i.e., the *profile* of the set. The "best sets" are those that have the "best profiles".

The representation of preferences in this approach [80] is performed as follows. For each feature $\mathcal{F}_i$, a function is defined that returns the result of an SQL query. Let `$S` be a variable denoting any three-element subset of the `movie` relation. Then, the following functions compute the total rating and the number of comedies in `$S`:

- $\mathcal{F}_1$(`$S`): `SELECT sum(rating) FROM $S;`
- $\mathcal{F}_2$(`$S`): `SELECT count(genre) FROM $S WHERE genre='comedy';`

Preferences over sets are defined by formulas over the above functions:

$$\begin{aligned} s_1 \succ_{C_1} s_2 &\equiv \mathcal{F}_1(s_1) > \mathcal{F}_1(s_2) \\ s_1 \succ_{C_2} s_2 &\equiv (\mathcal{F}_2(s_1) \geq 1) \wedge (\mathcal{F}_2(s_2) = 0) \\ s_1 \succ_{C_3} s_2 &\equiv s_1 \succ_{C_2 \triangleright C_1} s_2 \end{aligned}$$

The winnow operator is then used in order to select the "best" among all three-element subsets of the `movie` relation according to the above preference relations. In order for this to be done, the most direct approach is to enumerate all fixed-size subsets and check

them one by one. Notice that dealing with subsets of relations is, in general, an inherently inefficient task. For this reason, certain optimizations are provided that can potentially alleviate the inefficiency problem in specific cases.

The problem with the proposal of Zhang and Chomicki [80] is that the simple first-order logic language for expressing tuple preferences has to adapt significantly in order to accommodate set preferences, and the resulting formalism is more complicated. As a general comment on the above ideas, we could say that the features pf their approach are second-order predicates in disguise, and for this reason, they can be represented directly and very elegantly in our framework.

4.2.4 Discussion

The framework for expressing qualitative preferences in relational databases proposed by Chomicki [17] and its extension for fixed-cardinality sets [80], appears to have many advantages and it is relatively simple. However, it has certain shortcomings which, if resolved, could result in a more expressive formalism. In this subsection, we describe in detail the main shortcomings of these techniques [17, 80]. In subsequent sections, we demonstrate how these issues are resolved in our framework.

First of all, the framework of Chomicki [17] uses two different languages: the preference relations are expressed using first-order logic while preference queries are expressed using SQL extended with the winnow operator. Moreover, in the case of preferences over sets, yet another formalism (namely that of features and profiles) is introduced. As we argue in the coming sections, preference relations and queries can be expressed in a single language, simplifying in this way the representation of preferences. The differences in the conciseness of the representation are even more apparent in the case of set preferences, where we avoid the concepts of features and profiles [80], and our representation is a simple extension of the one used for tuple preferences.

A second characteristic of the framework of Chomicki [17] which we would like to avoid, is the restriction to *intrinsic* preference formulas. There exist many natural preference formulas that are *extrinsic* (see also the relevant discussion in [17][Section 7.3]). Some extrinsic preferences can be simulated using intrinsic formulas, but this is not always possible or convenient. Moreover, as we are going to see in Section 4.5, the use of extrinsic preferences is very important in the case of set preferences.

A final issue of the framework of Chomicki [17] is the restriction of having a single preference manipulation operator, namely winnow. To be fair, a few more operators similar to winnow are presented [17]. However, all these operators are defined in a language that is different from both first-order logic and SQL (additional set-theoretic and relational notations are used). In other words, new operators can only be “custom tailored” and they cannot be defined nor implemented with the available linguistic resources. In our framework, new operators can easily and naturally be defined as higher-order predicates.

4.3 Higher-Order Logic Programming

Higher-order logic programming [10, 15, 52] extends traditional first-order logic programming with higher-order constructs. In this section, we define some common higher-order logic programming concepts and some basic predicates that will be used throughout the chapter. A basic familiarity with some form of higher-order programming (either logic or functional) is assumed [10, 15, 27, 52].

The most popular higher-order extension is to allow the programmer to define predicates that accept other predicates as arguments, and variables to occur in places where predicates typically occur. For example, the following program defines the transitive closure of a given relation `R`:

```
tc(R,X,Y) :- R(X,Y).
tc(R,X,Y) :- R(X,Z), tc(R,Z,Y).
```

An advantage of predicates such as `tc` is that they can be used to achieve a *generic* style of programming: to compute the transitive closures of different relations, we invoke `tc` with different parameters (avoiding to write a transitive closure predicate for each different relation). As an example, the query:

```
?- tc(parent,john,Y).
```

will return all the ancestors of `john` while the query:

```
?- tc(graph,v1,V).
```

will return all vertices that are reachable from vertex `v1` of a given binary relation `graph`.

One interesting feature of higher-order languages is the use of partial applications, i.e., the ability to invoke a higher-order predicate with only some of its arguments. In order for this to be achieved, the single-tuple notation of classical logic programming is extended so as that a predicate can now have a sequence of tuples as successive parameters[5]. Then, we can invoke a predicate with only some of the tuples in the sequence. In this case we say that the predicate is *partially applied*. The partial application of a predicate yields a new predicate that expects the rest of the arguments and behaves exactly like a regular predicate. For example, we can write the `tc` predicate using a slightly different syntax:

```
tc(R)(X,Y) :- R(X,Y).
tc(R)(X,Y) :- R(X,Z), tc(R)(Z,Y).
```

Now, `tc(parent)` is an expression representing the transitive closure of the relation `parent` and can be used as an autonomous expression in the program and in queries. The idea of "partial application" can be further illustrated with the definition of two higher-order predicates that we will use in subsequent sections, namely the *conjunction* and *union* predicates over binary relations:

```
conj(R,Q)(X,Y) :- R(X,Y), Q(X,Y).
union(R,Q)(X,Y) :- R(X,Y).
union(R,Q)(X,Y) :- Q(X,Y).
```

Now the expression `union(tc(parent), tc(mother))` denotes the union of the two transitive closure relations while the expression `conj(tc(graph1), tc(graph2))` is the set of common edges that belong to the transitive closures of `graph1` and `graph2`.

The rest of this section contains some easy definitions of other higher-order predicates that we will use. Actually, some of them use a powerful feature of logic programming, namely negation-as-failure. A predicate that checks if a relation is empty can be implemented as follows:

[5] In the world of functional programming, this idea is called *currying*.

```
nonempty(R) :- R(X).
empty(R) :- not nonempty(R).
```

The nonempty predicate succeeds if there exists an object that satisfies its argument. The empty predicate succeeds if nonempty fails for the same parameter. Other useful operators are the following:

```
minus(R,X)(Y) :- R(Y), not (X=Y).
diff(R,Q)(X) :- R(X), not Q(X).
```

The minus operator takes as a parameter a relation R and an object X and returns a relation that contains all the objects in R except for X. In this sense minus removes X from relation R. Similar to the minus operator is the diff operator that creates the set difference of its given two relations. Given the above definitions, we can easily define the cardinality of a finite relation as:

```
size(R,0) :- empty(R).
size(R,N) :- R(X), size(minus(R,X),M), N is M+1.
```

In higher-order logic programming languages, queries of the following form are usually allowed:

```
?- tc(R)(john,mary).
```

The expected answer of such a query is not immediately obvious and this is actually a thorny subject among the higher-order logic programming approaches. The extensional approach [10] assumes that predicates denote sets and therefore two predicates that hold for the same elements are considered equal. An expected answer in the extensional approach is the substitution $\mathtt{R} = \{(\mathtt{john}, \mathtt{mary})\}$, namely the simplest set whose transitive closure contains the pair $\{(\mathtt{john}, \mathtt{mary})\}$. On the other hand, the intensional approach [15] assumes that each predicate is represented by its name and therefore two predicates are considered equal only if their names are the same. Under the intensional approach, the answer to the aforementioned query depends on whether there exists a predicate defined in the program that satisfies the goal. In the case that there is no such predicate in the program, the goal fails. As we are going to see, we will need to evaluate queries involving uninstantiated predicate variables (such as the above) in the case of preferences over sets. The way that we handle such queries in our implementation will be described in the following chapter.

4.4 Representing Preferences over Tuples in Higher-Order Logic Programming

In this section, we demonstrate how preference relations over tuples as-well-as operators over preference relations, can be defined in higher-order logic programming. The language that we will be using does not exploit the full power of higher-order logic programming, because most of our examples will be written in “higher-order Datalog”, namely higher-order logic programming without function symbols. There are only two exceptions to this issue:

- Our language supports tuples (in a restricted form) because preference relations are defined over tuples. In other words, we do not need arbitrary function symbols but we need our language to be able to handle tuples.

- In order to be able to define some operators over preference relations and some aggregate operations, we will use recursion and the usual arithmetic operations over the natural numbers. However, all the essential ideas from Chomicki [17] can be implemented in our framework without ever using natural numbers or any function symbols.

In conclusion, our framework is essentially a higher-order version of Datalog that supports tuples. In this respect, one can also view our proposal as a *higher-order deductive database framework for representing preferences*.

4.4.1 Representing Database Relations

In this short subsection, we describe how database relations can be represented in our framework. We follow the standard approach in deductive databases in which relations are represented by logic programming facts.

We adopt the following notational convention: we assume that every predicate that represents a database relation, does not have many different attributes but instead a single attribute that is a tuple. For example, in order to represent the `movie` relation of Table 4.1, we do not use a predicate with 6 arguments, but a predicate that takes a single argument that is a 6-ary tuple, namely `movie((ID,Name,Director,Genre,Runtime,Rating))`. The above convention is a merely technical one since it allows us to write a single generic version for every operator on preference relations. As we will see later, if we had not done this, we would have to write a specific n-ary version of each operator in order to support every possible n-ary database relation.

Apart from the above representation using logic programming facts, we can use logic programming rules to express a database relation. For instance, we could express preferences over database relations that are defined recursively:

Example 4.6. *Consider the relation* `edge((X,Y,C))`*, which defines a weighted directed graph, where* `C` *denotes the cost of edge* `(X,Y)`*. Then, the following relation represents the costs of all paths between every pair of nodes* `(X,Y)`*.*

```
path((X,Y,C)) :- edge((X,Y,C)).
path((X,Y,C)) :- edge((X,Z,C1)), path((Z,Y,C2)), C is C1+C2.
```

Then, if a path with a lower cost is preferred, then the most preferred path between two nodes is the shortest path between these nodes. The program that represents this formulation of the known shortest path problem will be presented in Subsection 4.4.5.

4.4.2 Representing Preference Relations

In this subsection, we describe how preference relations can be represented in our framework. In short, preference relations can be represented using binary logic programming predicates. This flexibility allows us to express both intrinsic and extrinsic preference relations.

We can represent a preference relation $\succ_C$ between tuples of an n-ary relation using a binary predicate `c_pref` with two arguments, each one of them being an n-ary tuple. That is, the atom `c_pref((X1,X2),(Y1,Y2))` corresponds to the formula $(\mathtt{X1}, \mathtt{X2}) \succ_{\mathtt{c}} (\mathtt{Y1}, \mathtt{Y2})$. The preference formula C is encoded by the body of the rule defining `c_pref`. Notice that for the representation of intrinsic preference relations, we do not use the higher-order characteristics of our source language.

Example 4.7. *Consider the* `movie(ID,Name,Director,Genre,Runtime,Rating)` *relation. We can represent the preference relations of Subsection 4.2.1 using predicates over tuples. Assume we have the preference relation C_1 of Example 4.1, namely: "prefer one movie over another iff their genres are the same and the rating of the first is higher". This can be represented as follows:*

```
c1_pref((I1,N1,D1,G,T1,R1),(I2,N2,D2,G,T2,R2)) :-
        movie((I1,N1,D1,G,T1,R1)),
        movie((I2,N2,D2,G,T2,R2)),
        R1 > R2.
```

Consider the preference relation C_2 from Example 4.1, namely "prefer a movie from another one if the former lasts for less than or equal to 150 minutes while the latter does not". This can be represented by:

```
c2_pref((I1,N1,D1,G1,T1,R1),(I2,N2,D2,G2,T2,R2)) :-
        movie((I1,N1,D1,G1,T1,R1)),
        movie((I2,N2,D2,G2,T2,R2)),
        T1 =< 150, T2 > 150.
```

Consider the preference relation C_3 from Example 4.1, namely "prefer drama movies over movies of all other genres". This can be represented as follows:

```
c3_pref((I1,N1,D1,drama,T1,R1),(I2,N2,D2,G,T2,R2)) :-
        movie((I1,N1,D1,drama,T1,R1)),
        movie((I2,N2,D2,G,T2,R2)),
        not (G=drama).
```

In the above examples, we explicitly check that each tuple belongs to the `movie` *relation, something that cannot be expressed in the preference formulas of Chomicki [17] (which only check the properties of individual elements of tuples).*

The fact that we can check whether each tuple belongs to a specific relation gives an advantage to our technique. In the above examples, we checked whether a specific tuple is a movie tuple. Going one step further, we could check whether a tuple (or some elements of a tuple) belongs to a relation other than the `movie` relation. As we are going to see below, this allows us to express extrinsic preferences. In the remaining part of the subsection, we present some examples of extrinsic preferences that are possible to be defined in an elegant way in our approach.

Example 4.8. *Assume we prefer any tuple from a relation* `r` *over any tuple from another relation* `s`*. In the framework of Chomicki [17] such a requirement can only be simulated, somewhat artificially, by adding an extra argument to each tuple that denotes the relation name to which the tuple belongs. In our case this can simply be written as:*

```
c_pref(T1,T2) :- r(T1), s(T2).
```

Notice that our assumption that database facts only take a single attribute that is a tuple, allows us to use above only the variables `T1` *and* `T2` *and to avoid listing all the attributes of relations* `r` *and* `s`*.*

Example 4.9. *Continuing Example 4.1, assume that we prefer those movies that have the most popular directors (namely directors who have filmed the maximum number of*

movies). This is an extrinsic preference because we cannot compare two movie tuples based only on the information contained in the tuples. Given two movie tuples, the only way to decide whether the one is preferred over the other is to access the entire `movie` *relation so as to discover the directors with the maximum number of movies. Such preferences cannot be defined in Chomicki's framework [17]. In order to solve this problem, Chomicki follows a specialized approach which is based on the construction of separate views through the use of SQL queries involving aggregate operators. In our case though, this preference relation can be expressed as follows:*

```
director_pref((I1,N1,D1,G1,T1,R1),(I2,N2,D2,G2,T2,R2)) :-
        movie((I1,N1,D1,G1,T1,R1)),
        movie((I2,N2,D2,G2,T2,R2)),
        director(D1,K1),
        director(D2,K2),
        K1 > K2.

movies_of_director(D)(I) :- movie((I,_,D,_,_,_)).

director(D,K) :- movie((_,_,D,_,_,_)),
                 size(movies_of_director(D),K).
```

Notice that we use an aggregate operator, namely the predicate `size` *which was defined in Section 4.3. Moreover, we use for the first time a higher-order characteristic of our language: the partially applied expression* `movies_of_director(D)` *is a relation that contains all the different movie IDs that a director has filmed. A difference from Chomicki [17] is that a unique language is used in order to express our preference relations.*

Example 4.10. *We give another example of a natural extrinsic preference relation which can easily be encoded in our framework. Continuing Example 4.1, assume we "prefer movies that have an above-average rating over those that have a below-average rating". This preference requires the calculation of the average rating of all movies, and therefore it is an extrinsic one. It can be expressed in our setting as follows:*

```
average_pref((I1,N1,D1,G1,T1,R1),(I2,N2,D2,G2,T2,R2)) :-
        movie((I1,N1,D1,G1,T1,R1)),
        movie((I2,N2,D2,G2,T2,R2)),
        size(movie,K),
        rating_sum(movie,S),
        R1 >= S/K, R2 < S/K.

rating_sum(Rel,0) :- empty(Rel).
rating_sum(Rel,S) :- Rel((I,N,D,G,M,R)),
                     rating_sum(minus(Rel,(I,N,D,G,M,R)),S1),
                     S is S1+R.
```

The `rating_sum` *predicate calculates the sum of ratings of all tuples in the* `movie` *relation; dividing this sum by the size of the* `movie` *relation gives us the average rating of all movies.*

4.4.3 Representing Composition Operators

In this subsection, we describe how preference compositions can be represented in our framework. In short, instead of simply creating new composite clauses that express

the composition of specific preference relations, we can benefit from the higher-order characteristics of our approach. This fact offers us the ability to define generic preference composition predicates.

A straightforward way to compose preference relations in our framework is to create new clauses that use in their bodies the predicates of the initial relations. For example, in order to obtain the logical conjunction of two preference relations `c1_pref` and `c2_pref`, we can simply define:

```
c_pref(T1,T2) :- c1_pref(T1,T2), c2_pref(T1,T2).
```

However, in higher-order logic programming, we can do better than this since we can have a generic `conj` operator, which is defined as follows:

```
conj(R,Q)(T1,T2) :- R(T1,T2), Q(T1,T2).
```

The advantage of the above approach is that in order to compose various preference relations we do not need to create new clauses (or new formulas as in the framework of Chomicki [17]), but we can specify the compositions in combinatory form. For example, `conj(c1_pref, union(c2_pref,c3_pref))` represents the conjunction of `c1_pref` with the union of `c2_pref` and `c3_pref`.

The *Prioritized* and *Pareto* compositions of preference relations can be easily defined in an analogous way using generic operators:

```
prioritized(C1,C2)(T1,T2) :- C1(T1,T2).
prioritized(C1,C2)(T1,T2) :- indifferent(C1)(T1,T2), C2(T1,T2).

indifferent(C)(T1,T2) :- not C(T1,T2), not C(T2,T1).

pareto(C1,C2)(T1,T2) :- C1(T1,T2), not C2(T2,T1).
pareto(C1,C2)(T1,T2) :- C2(T1,T2), not C1(T2,T1).
```

Another characteristic of our approach is that it allows us to define the transitive closure on preference relations by using the *tc* predicate given in Section 4.3. The following example motivates the need for this operator.

Example 4.11. *Assume we have a set of available items together with their color:*

```
item((a1,black)).
item((a4,green)).
```

Moreover, assume we prefer black over red, red over blue, blue over yellow and yellow over green items. This can be expressed using the following facts:

```
color_pref((_,black), (_,red)).
color_pref((_,red), (_,blue)).
color_pref((_,blue), (_,yellow)).
color_pref((_,yellow), (_,green)).
```

Notice that the relation `color_pref` *relation is not transitive, even though the intended meaning of* `color_pref` *should express that e.g. the color red is preferred over the color green. Therefore, instead of adding extra facts to the* `color_pref` *relation that would express its transitivity, we can instead use in our queries the relation* `tc(color_pref)`*.*

It should be noted that in the framework of Chomicki [17] the transitive closure of a relation cannot be directly specified (due to the inability of first-order logic to define transitive closure on finite structures). However, as shown in [17][Theorem 4.10], if a preference relation is defined through an ipf, then its transitive closure can also be defined through an ipf, which can be effectively constructed using a Datalog program. In our case though, no special construction is required apart from the application of the `tc` predicate to the given preference relation. Moreover, in our case, the transitive closure can also be applied to extrinsic preference relations.

Finally, we illustrate how *lexicographic preferences* can be expressed using higher-order logic programming. A *lexicographic ordering* (cf. Definition 2.18) can be expressed easily with a simple preference composition:

```
lexicographic(C1,C2)((X1,X2),(Y1,Y2)) :- C1(X1,Y1).
lexicographic(C1,C2)((X1,X2),(Y1,Y2)) :- X1 = X2, C2(Y1,Y2).
```

The following example uses the above composition for expressing a lexicographic preference:

Example 4.12. *Consider a* `room((ID,PricePerNight,DistanceFromTheSea))` *relation. Following Example 2.11, a lexicographic preference for hotel rooms that are cheap and closer to the sea, but the first criterion is more important than the second, can be defined as follows:*

```
less(X,Y) :- X < Y.

room_pref((I1,P1,D1),(I2,P2,D2)) :-
        room((I1,P1,D1)),
        room((I2,P2,D2)),
        lexicographic(less,less)((P1,D1),(P2,D2)).
```

Notice that the lexicographic preference composition is a *multi-dimensional* composition, and it differs from the preference compositions that we presented previously. This is due to the fact that the domain of the resulting preference relation is the Cartesian product of the domains of the given preference relations and not the same domain as that of the given preference relations. Moreover, in Proposition 2.9 we proved that lexicographic preferences cannot be expressed using the language PrefLog due to the form of its underlying set of truth values. The above example indicates that there exist (at least) one family of preferences that can be expressed using higher-order logic programming but not using PrefLog.

4.4.4 Representing Operators on Preference Relations

In this subsection, we describe how operations on preference relations can be represented in our framework. As previously, operations on preference relations are treated the same way as preference compositions, namely as higher-order predicates. Apart from the classical winnow operator, we can define additional preference operators.

An important characteristic of the proposed approach is its ability to define directly new operators on preference relations. These operators are in fact higher-order predicates that operate on database and preference relations. We start by recalling the winnow operator:

$$w_C(r) = \{t \in r : \neg\exists t' \in r \text{ s.t. } t' \succ_C t\}$$

The above definition can be directly transcribed in higher-order logic programming as follows:

```
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(T1), C(T1,T).
```

In the above definition observe that our assumption that database facts take a single attribute that is a tuple, allows us to use only the variables `T` and `T1` without caring about how many arguments the relation `R` has. If our facts did not use the single-tuple notation we would need to write a different winnow operator for each different tuple size. We can now use this operator to formulate queries such as:

```
?- winnow(c1_pref,movie)(T).
```

that will succeed for all tuples `T` that belong to the relation `movie` and are most preferred with respect to the relation `c1_pref`. Moreover, one can use the combinators defined in Section 4.4.3 to create more complex queries:

```
?- winnow(prioritized(c1_pref,c2_pref),movie)(T).
?- winnow(tc(color_pref),item)(T).
```

In a similar manner, we can define other interesting operators. A simple variation of winnow is the *"iterated-winnow"* which is defined in [17][Section 8] as follows:

$$\begin{array}{rcl} w_C^1(r) & = & w_C(r) \\ w_C^{n+1}(r) & = & w_C(r - \bigcup_{i=1}^{n} w_C^i(r)) \end{array}$$

The w_C^n operator selects the "n"-best tuples. For example, $w_C^2(r)$ returns the second-best tuples of r with respect to the preference relation C. In our framework w_C^n can be easily defined as a higher-order predicate:

```
w(C,R)(1)(T) :- winnow(C,R)(T).
w(C,R)(N)(T) :- N>1,M is N-1,
                winnow(C,diff(R,gen_union(w(C,R)(M))))(T).
```

where the `gen_union` operation is a generalized union operator (over an indexed family of sets) which can be defined as follows:

```
gen_union(R)(1)(X) :- R(1)(X).
gen_union(R)(N)(X) :- N>1,M is N-1,
                      union(gen_union(R)(M), R(N))(X).
```

Notice that the definition of `w` has a strong resemblance to the mathematical definition given previously. Given the definition of `w` one can retrieve the second-best tuples of a relation by posing the query:

```
?- w(c1_pref, movie)(2)(T).
```

Assume we want to return all the tuples up to a desired level. We can define the operator $wt_C^n(r) = \bigcup_{i=1}^{n} w_C^i(r)$ that uses w_C^n as follows:

```
wt(C,R)(N)(T) :- gen_union(w(C,R))(N)(T).
```

Finally, we can easily define the *ranking* operator [17], which ranks the elements of a relation r with respect to a preference C:

$$\eta_C(r) = \{(t, i) : t \in w_C^i(r)\}$$

The operator η can naturally be defined as follows:

```
eta(C,R)(T,I) :- size(R,N), between(1,N,I), w(C,R)(I)(T).
```

where `between` is a (simple to define) predicate which is true if `I` is an integer between `1` and `N`.

4.4.5 Additional Complex Representations

In this subsection, we present additional, more complex preference representations that are generally not is not generally possible in the approach of Chomicki [17]. Since base relations, preference relations and preference operations are expressed in the same language, we can have complex preference representations that mix all these elements in a uniform way. In the following, we illustrate this using the *shortest path* problem as an example.

As we have seen earlier, an important characteristic of our approach is that we can define preference relations over database relations that are defined recursively (known as IDBs in the deductive database literature); this is not generally possible in the approach of Chomicki [17] (see the discussion in [17][Section 4.3, pages 439-440]).

Example 4.13. *Consider the relation* `edge((X,Y,C))`*, which defines a weighted directed graph, where* `C` *denotes the cost of edge* `(X,Y)`*. We can formulate the shortest path problem, in a somewhat naive way, as follows:*

```
naive_shortest(X,Y,C) :- winnow(path_pref,path)((X,Y,C)).

path_pref((X,Y,C1),(X,Y,C2)) :- path((X,Y,C1)),
                                path((X,Y,C2)), C1 < C2.

path((X,Y,C)) :- edge((X,Y,C)).
path((X,Y,C)) :- edge((X,Z,C1)), path((Z,Y,C2)), C is C1+C2.
```

In other words, the most preferred path is the one that has the smallest cost.

The above program enumerates all paths from `X` to `Y`, and then uses winnow to select the most preferable one(s). We can write a more efficient version by embedding winnow inside the recursive definition of the `path` predicate. We believe that this is a nice consequence of a single language for representing preferences and operators on them, and gives another interesting application beyond the system described by Chomicki [17]. This enhanced path idea was motivated by the "optimal subproblem property" for shortest distance, discussed by Govindarajan et al. [33][page 94].

Example 4.14. *The following program finds the shortest path from* `X` *to* `Y` *by first finding the optimal paths from every neighbor* `Z` *of* `X` *to* `Y`*.*

```
enhanced_shortest(X,Y,C) :- winnow(path_pref,opt_path)((X,Y,C)).

path_pref((X,Y,C1),(X,Y,C2)) :- opt_path((X,Y,C1)),
                                opt_path((X,Y,C2)), C1 < C2.

opt_path((X,Y,C)) :- edge((X,Y,C)).
opt_path((X,Y,C)) :- edge((X,Z,C1)),
```

```
                    winnow(path_pref,opt_path)((Z,Y,C2)),
                    C is C1+C2.
```

Notice the use of winnow inside the recursive definition of `opt_path`.

Both the naive and the optimized `path` predicates work correctly when the graph is acyclic. In a graph that contains cycles, there may exist an infinite number of paths that go from `X` to `Y`, because some paths can go around a cycle for an arbitrary number of times. One can easily extend the above program to take an extra parameter that restricts the length of the desired path to be less than or equal to the number of vertices in the graph. This modified path predicate is shown in the following example.

Example 4.15. *The following program finds the shortest path from* `X` *to* `Y` *that uses at most* `N` *edges by first finding the optimal paths from every neighbor* `Z` *of* `X` *to* `Y` *that use at most* `N-1` *edges.*

```
enhanced_shortest(X,Y,C,N) :- winnow(path_pref,opt_path)((X,Y,C,N)).

path_pref((X,Y,C1,N),(X,Y,C2,N)) :- opt_path((X,Y,C1,N)),
                                    opt_path((X,Y,C2,N)), C1 < C2.

opt_path((X,Y,C,N)) :- N>0, edge((X,Y,C)).
opt_path((X,Y,C,N)) :- N>1, edge((X,Z,C1)), N1 is N-1,
                       winnow(path_pref,opt_path)((Z,Y,C2,N1)),
                       C is C1+C2.
```

Again, notice the use of winnow inside the recursive definition of `opt_path`.

4.5 Representing Preferences over Sets in Higher-Order Logic Programming

In this section, we illustrate how set preferences are expressed in our framework. The approach we follow is a generalization of the techniques we have used in order to represent tuple preferences. More specifically, we now have to define *preference relations over sets*. This does not require any changes in our notation: in higher-order logic programming, predicates denote sets and therefore sets can be the parameters of other predicates. As a result, the preference relations here are second-order since their arguments are predicates themselves.

Example 4.16. *Consider the* `movie` *relation that we have been using as our running example. In the following, we demonstrate how the example preference relations discussed in Subsection 4.2.3 can be represented in our framework.*

1. *I prefer the sum of the ratings of the movies to be the highest possible. This means that a set* `S1` *is preferable over a set* `S2` *if the sum of the ratings of the elements of* `S1` *is greater than that of the set* `S2`*. The* `rating_sum` *predicate, defined in Example 4.10, recursively calculates the cumulative rating of a given set by selecting an element from the set, calculating the sum of the ratings of the remaining elements and then adding the rating of the selected element.*

   ```
   rating_pref(S1,S2) :- rating_sum(S1,R1),
                         rating_sum(S2,R2),
                         R1 > R2.
   ```

2. *I prefer to watch at least one comedy. In other words, a set* `S1` *is more preferred than another set* `S2` *if the former contains at least one comedy, while the latter does not contain any comedies.*

```
comedy_pref(S1,S2) :- S1((_,_,_,comedy,_,_)),
                      not S2((_,_,_,comedy,_,_)).
```

3. *Prioritize (1) to (2). In other words, choose the set that has the least cumulative rating, but if two sets have the same total rating, prefer the one that contains at least one comedy movie.*

```
ratingcomedy_pref(S1,S2) :- prioritized(rating_pref,comedy_pref)(S1,S2).
```

Notice that we can directly use the `prioritized` *operator we defined in Subsection 4.4.3 for preference relations over tuples. More generally, defining preference relations over sets in higher-order logic programming is an analogous task to that of defining preference relations over tuples.*

The use of the winnow operator is very similar to that for the tuple case. For example, in order to find the "best" 3-element subsets of the `movie` relation with respect to the preference relation `rating_pref`, we simply need to ask the query:

```
?- winnow(rating_pref, subsetof(movie,3))(S).
```

The execution of the above query returns as bindings for the variable `S` all the "best" 3-element subsets with respect to `rating_pref`. The variable `S` in the above query is actually an uninstantiated predicate variable, i.e., it represents a set. The above query assumes the existence of a predicate `subsetof(R,K)(S)`, which qiven a relation `R` and a natural number `K`, returns as bindings in the variable `S` all the subsets of `R` that are of size `K`. As we have already mentioned, the treatment of such queries depends on the higher-order language under consideration. This means that the `subsetof` predicate can be implemented in various ways depending on the higher-order language adopted and on the applications aimed at. The approach we have followed in order to treat such variables in our actual implementation will be described in the next chapter.

An advantage of the higher-order logic programming approach to preference representation is the ability to express non-intrinsic preferences over sets:

Example 4.17. *Consider a* `movie` *relation (different than that of our running example), defined by the predicate* `movie((ID, Name, PID))`*, where* `ID` *uniquely identifies the movie and* `PID` *refers to the id of the movie that this movie is a sequel of. We also have a* `prequelOf` *predicate that identifies a movie tuple that is a prequel of another one:*

```
movie((m01, the_Godfather, null)).
movie((m02, the_Godfather_Part_II, m01)).
movie((m03, the_Godfather_Part_III, m02)).
movie((m04, kill_Bill_Vol_1, null)).
movie((m05, kill_Bill_Vol_2, m04)).
movie((m06, the_Sting, null)).

prequelOf((Z,_,_),(_,_,Z)).
```

Suppose that we prefer collections of movies that are complete, i.e., collections that contain all parts of a movie. This can be expressed by the following preference relation over sets of movies:

```
movie_pref(S1,S2):- not seriesMissing(S1), seriesMissing(S2).

seriesMissing(S) :- S(X), movie(Y), inseries(X,Y), not S(Y).

inseries(X,Y) :- prequelOf(X,Y) ; prequelOf(Y,X).

preferred_sets(N)(S) :- winnow(movie_pref,subsetof(movie,N))(S).
```

Consider the following five queries:

```
?- preferred_sets(1)(S), S(T).
?- preferred_sets(2)(S), S(T).
?- preferred_sets(3)(S), S(T).
?- preferred_sets(4)(S), S(T).
?- preferred_sets(5)(S), S(T).
```

The first query will return a singleton set `S` *that contains as the only tuple* `T` *the one that corresponds to "The Sting" movie. The second one will return the set* `S` *that contains the two tuples of the "Kill Bill" series. The third one will return two solutions: one that contains the three tuples concerning the "Godfather" series, and one that contains the "Kill Bill" series together with "The Sting". The fourth query will return a set containing the "Godfather" series together with "The Sting". Finally, the fifth query will return a set containing the "Godfather" series together with the "Kill Bill" series.*

Notice also that the relation `movie_pref` *is extrinsic since it clearly depends on other tuples that are outside the subsets we are comparing.*

4.6 Summary

In this chapter, we proposed the use of higher-order logic programming for expressing and manipulating preferences. The proposed framework can express both intrinsic and extrinsic preference relations, it can be used to define a variety of interesting alternative operators beyond winnow and it can also represent set preferences in a natural way.

Our framework extends the seminal work of Chomicki [17] which advocates the embedding of first-order preference formulas into relational algebra through a winnow operator that is parameterized by a database relation and a preference formula. Despite its elegance, Chomicki's proposal has certain shortcomings: only intrinsic preference formulas are supported, the preference relations and the preference queries are expressed in two different languages, and there is no direct way to define alternative operators beyond winnow. Moreover, another formalism (namely that of features and profiles) needs to be introduced for expressing preferences over sets [80].

We demonstrated that the use of higher-order logic programming as a logical framework remedies the above deficiencies. We introduced examples of extrinsic preferences, examples of preference relation compositions that are defined using generic predicates, and definitions of alternative preference operations beyond winnow. We showed that the conciseness of the representation is also evident in the case of set preferences, where we avoided the concepts of features and profiles, by providing a representation which is

a simple extension of the one used for tuple preferences. By doing so, we proposed a uniform framework in which relations, preferences between tuples, preferences between sets of tuples and operations on preferences are expressed in the same language.

5. OPTIMIZING PREFERENTIAL HIGHER-ORDER LOGIC PROGRAMS

In this chapter, we undertake a HiLog implementation of our higher-order preferential framework. Apart from a basic, unoptimized implementation, we consider optimization techniques for enhancing its performance. The first family of optimizations targets HiLog programs that express preferences over tuples, while the second targets HiLog programs that express preferences over sets. Amongst the techniques that we use, we propose Predicate Specialization, a novel transformation technique used for optimizing higher-order logic programs that express preferences over tuples.

5.1 Overview

In order to assess the potential of higher-order logic programming for expressing qualitative preferences, we undertook an implementation of our higher-order preferential framework that we proposed in Chapter 4. The language of the implementation is the language HiLog [15], a stable and well-known higher-order logic programming language. Our implementation has been realized in the XSB system[6] [70], a mature, goal-oriented, top-down Prolog system that supports HiLog natively. We argue that higher-order logic programming is not only a very expressive language for representing qualitative preferences, but it can offer an effective way of implementing them from a practical point of view, as well.

Apart from providing a straightforward implementation of our ideas in HiLog, we focus on techniques that can enhance higher-order logic programming so that it can better handle and manipulate preferences. We propose *Predicate Specialization*, a novel technique that transforms a higher-order program into a faster equivalent first-order program according to a specific input query. Tuple-preference HiLog programs are optimized using Predicate Specialization combined with classical Prolog optimizations such as *tabling*. Set-preference HiLog programs are optimized using pruning techniques obtained from Zhang and Chomicki [80] and adapted to our setting. These optimizations reduce the program execution time by reducing the number of the generated candidate sets.

The rest of the chapter is organized as follows: in Section 5.2 we present a basic, unoptimized implementation of our higher-order preferential framework in HiLog; in Section 5.3 we present Predicate Specialization, a program transformation technique for optimizing definitional higher-order logic programs by transforming them into equivalent first-order programs; in Section 5.4 we optimize HiLog programs that express preferences over tuples by using Predicate Specialization; in Section 5.5 we optimize HiLog programs that express preferences over sets by using two simple pruning techniques adopted from [80]; and finally we close with a brief summary of the chapter.

5.2 A Naive Implementation

In this section, we propose a basic implementation[7] of our proposed higher-order logic programming framework in the logic programming language HiLog. A naive, unoptimized implementation is relatively straightforward.

[6] cf. `http://xsb.sourceforge.net/`
[7] cf. `http://bitbucket.org/antru/holppref`

Instead of developing a tailor-made higher-order language for executing the preferential programs of Chapter 4, we thought that it would be more convenient to build upon an existing language. As we stated in Section 4.3, existing higher-order logic programming languages fall in two major categories; the *itensional* and the *extensional* ones. The main difference between these two families of languages is the way how they handle the free (or uninstantiated) predicate variables. Implementations of extensional higher-order languages have not yet reached the same level of maturity as that of intensional ones. As a result, we chose an intensional language (i.e., HiLog) for implementing our higher-order preferential framework. HiLog [15] is a stable and well-known logic programming language. Its syntax is embedded in the XSB system [70], a mature, goal-oriented, top-down Prolog engine.

Programs for tuple preferences do not use uninstantiated predicate variables. Therefore, regarding the choice of the language of the implementation, for the tuple-preference case either an intensional or an extensional language would do. Indeed, the higher-order logic programs that express preferences over tuples that we have considered in Section 4.4 are valid HiLog programs and can be compiled and executed by XSB directly. Even though we could use various techiques for optimizing the program execution times, a basic, unoptimized implementation of our higher-order framework in HiLog is pretty straightforward.

The case of higher-order logic programs that express preferences over sets is more demanding than those that express preferences over tuples, and even their unoptimized execution requires some nontrivial interventions from our side. Programs for set preferences make use of uninstantiated variables, but only in the higher-order predicate that generates the candidate sets. Regarding the choice of the language of the implementation, it seems that an extensional language would suit better to our needs. This is because in an extensional language, a query with an uninstantiated predicate variable can produce a suitable relation that satisfies the query, while in an intensional language, this query cannot produce a new relation, but it returns the name of the relation that satisfies the query only if this query is already defined in the program. In other words, an extensional language can produce all subset relations "on the fly"[8], but in an intensional language such as HiLog the subset relations must have been generated beforehand. More specifically, when a HiLog query contains a variable denoting a set, the implementation searches to find whether there exists *a predicate defined in the program* that could satisfy the given query. If there exists such a predicate, then its name is returned; otherwise, the query fails.

Example 5.1. *Consider the following HiLog program:*

```
p(Q) :- Q(a),Q(b).
q(a).
q(b).
```

The query:

```
?- p(Q).
```

[8] In Chapter 8, we give a hint of how `subsetof` could have been implemented in an extensional higher-order logic programming language. Such a system is the logic programming language Hopes [10], but the corresponding implementation is not as mature as that of HiLog. In Hopes, a query involving uninstantiated predicate variables directly produces all the potential subsets.

will return the answer `Q = q`*. However, if there is no predicate* `q` *defined in the program, the above query will fail in HiLog.*

The reasons why the above handling of queries with uninstantiated predicate variables is inconvenient, is clarified by the following example.

Example 5.2. *Recall Example 4.17 and consider the following query:*

```
?- preferred_sets(2)(S).
```

which amounts to the equivalent query:

```
?- winnow(movie_pref,subsetof(movie,2))(S).
```

In order for HiLog to answer properly the above query, there must exist explicitly defined in the program predicates that denote three-element subsets of the `movie` *relation (because otherwise, the query will fail).*

The above example suggests that in order for a query `?- winnow(c,subsetof(r,2))(S)` to function properly in an unoptimized implementation, all 2-element subsets of `r` must have been generated beforehand as *named* predicates. This means that each subset is associated with a predicate with a unique *name* and every such predicate is defined with exactly 2 facts, which are the elements of the set. There are mainly two approaches for solving this problem of set generation; in a static approach, the facts that correspond to the candidate 2-element sets are generated using a preprocessing at compile time; in a dynamic approach, the corresponding facts are asserted in the dynamic database during the first time `subsetof(r,2)(S)` is called. In our implementation, we followed the dynamic approach. This behavior of the `subsetof` predicate is illustrated in the following example:

Example 5.3. *Assume that we have defined a predicate* `r` *with these facts:*

```
r(a).
r(b).
r(c).
```

and we have the goal clause:

```
?- subsetof(r,2)(S).
```

Then, the implementation proceeds as follows:

- *Three new predicate names* `s1`*,* `s2` *and* `s3` *are introduced and corresponding facts are asserted in the program. More specifically, predicate* `s1` *consists of the facts* `s1(a)` *and* `s1(b)`*, predicate* `s2` *consists of the facts* `s2(a)` *and* `s2(c)` *and predicate* `s3` *consists of the facts* `s3(b)` *and* `s3(c)`*.*
- *The* `S` *variable gets the new predicate names* `s1`*,* `s2` *and* `s3` *as bindings.*

In order to achieve the above behavior, unique symbol names for the dynamically generated predicates are first produced. Then, a systematic procedure is followed in order to enumerate the elements of each subset, create the appropriate facts and assert them into the knowledge-base. The procedure is implemented by a `generate_relations` predicate

which produces a list of tuples and asserts each tuple as a fact under the newly generated predicate name. The `generate_relations` has been defined in an extensible way that allows the incorporation of specialized pruning strategies and more sophisticated optimizations, some of which will be discussed in Section 5.5.

5.3 Predicate Specialization: A technique for optimizing Definitional Higher-order Logic Programs

In this section, we present Predicate Specialization, a novel program transformation technique, based on partial evaluation, that can be applied to a modest but useful class of higher-order logic programs that includes tuple-preference higher-order programs. This specialization technique transforms a definitional higher-order program into an equivalent first-order program without introducing additional data structures. The resulting first-order programs can be executed efficiently by conventional logic programming interpreters and benefit from other optimizations that might be available. In our context, we can use this technique for optimizing the program execution time of HiLog programs that express tuple preferences.

5.3.1 Overview of the Technique

Higher-order logic programming has been long studied as an interesting extension of traditional first-order logic programming and various approaches exist with different features and semantics [10, 15, 52]. The support of higher-order features, however, usually comes with a price, and the efficient implementation in either logic or functional programming is a non-straightforward task.

The use of higher-order constructs is a standard feature in every functional language in contrast to the logic programming languages. As a result, there exists a plethora of optimizations that target specifically the efficient implementation of such features. A popular direction is to remove higher-order structures altogether by transforming higher-order programs into equivalent first-order ones, with the hope that the execution of the latter will be much more efficient. Reynolds, in his seminal paper [60], proposed a defunctionalization algorithm that is complete, i.e., it succeeds to remove all higher-order parameters from an arbitrary functional program. There is however a tradeoff; his algorithm requires the introduction of data structures in order to compensate for the inherent loss of expressivity [39]. Other approaches [16, 53, 55] have been proposed that do not use data structures but share the limitation that are not complete.

In the logic programming context there exist many transformation algorithms with the purpose of creating more efficient programs. Partial evaluation algorithms [29, 51, 47], for example, can be used to obtain a more efficient program by iteratively unfolding logic clauses. Most of the proposals, however, focus on first-order logic programs. Proposals that can be applied to higher-order programs are limited. The prominent technique that targets higher-order logic programs [15, 79] is adopted from HiLog. It employs Reynolds' defunctionalization adapted for logic programs. As a consequence it naturally suffers from the same shortcomings as the original technique: the resulting programs are not natural and the conventional logic programming interpreters fail to identify potential optimizations without specialized tuning [65].

In this section, we propose a partial evaluation technique that can be applied to higher-order logic programs. The technique propagates only higher-order arguments and avoids to change the structure of the original program. Moreover, it differs from Reynolds' style

defunctionalization approaches as it does not rely on any type of data structures. As a result, the technique will only guarantee to remove the higher-order arguments in a well-defined subset of higher-order logic programs. We have identified a well-defined fragment of higher-order logic programming for which the technique terminates and produces a logic program without higher-order arguments. This technique can be used in optimizing the program execution time for the case of higher-order logic programs that express preferences over tuples.

In the rest of this subsection, we present an introductory example so as to give an informal description of our technique. We borrow an example of a simple winnow operation from the previous chapter (ref. Subsection 4.4.4).

Example 5.4. *Recall the definition of the winnow operator, which selects the most preferred tuples* `T` *out of a given unary relation* `R`*, based on a binary preference predicate* `C`*. The preference predicate, given two tuples, succeeds if the first tuple is more preferred than the second.*

```
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(Z), C(Z,T).
```

The program contains predicate variables (for example, the variable `C` *and the variable* `R` *on both rules of the program), that is variables that can occur in places where predicates typically occur.*

Assume that we have a unary predicate `movie` *which corresponds to a relation of movies and a binary predicate* `pref` *which given two movies succeeds if the first argument has a higher rating than the second one. Now, suppose that we issue the query:*

```
?- winnow(pref,movie)(T).
```

We expect as answers the most "preferred" movies, that is all movies with the highest rating.

In the following, we will show how we can create a first-order version of the original program specialized for this specific query. Notice that the atom `winnow(pref,movie)(T)`*, that makes up our given query, does not contain any free predicate variables, but on the contrary, all of its predicate variables are substituted with predicate names. Therefore, we can specialize every program clause that defines* `winnow` *by substituting its predicate variables with the corresponding predicate names. By doing so, we get a program where our query yields the same results as to those in the original program:*

```
winnow(pref,movie)(T) :- movie(T), not bypassed(pref,movie)(T).
bypassed(P,R)(T) :- R(Z), P(Z,T).
```

We can continue this specialization process by observing that in the body of this newly constructed clause there exists the atom `bypassed(pref,movie,T)`*, in which all predicate variables are again substituted with predicate names. Therefore, we can specialize the second clause of the program accordingly:*

```
winnow(pref,movie)(T) :- movie(T), not bypassed(pref,movie)(T).
bypassed(pref,movie)(T) :- movie(Z), pref(Z,T).
```

There are no more predicate specializations to be performed and the transformation stops. Notice that the resulting program does not contain any predicate variables, but it

is not a valid first-order one. Therefore, we have to perform a simple rewriting in order to remove all unnecessary predicate names that appear as arguments.

```
winnow1(T) :- movie(T), not bypassed2(T).
bypassed2(T) :- movie(Z), pref(Z,T).
```

Due to this renaming process, instead of the initial query, the user now has to issue the query `?- winnow1(T)`. *Comparing the final first-order program with the original one it is easy to observe that no additional data structures were introduced during the first-order transformation, a characteristic that leads to significant performance improvement comparing to other transformation techniques that introduce data structures on the resulting programs (cf. Chapter 6).*

This technique, however, cannot be applied to every higher-order logic program. Notice that the resulting program of the previous example does not contain any predicate variables. This holds due to the fact that in the original program, every predicate variable that appears in the body of a clause also appears in the head of this clause. By restricting ourselves to programs that have this property we ensure that the transformation outputs a first-order program. Moreover, the transformation in this example terminates because the set of the *specialization atoms* (i.e., `winnow(pref,movie)(T)` and `bypassed(pref,movie)(T)`) is finite, which is not the case in every higher-order logic program. To solve this, we need to keep the set of specialization atoms finite. This is achieved in two ways. Firstly, we ignore all first-order arguments in every specialization atom, meaning that in a query of the form `?- winnow(pref,movie)(m_001)`, we will specialize the program with respect to the atom `winnow(pref,movie)(T)`. Secondly, we impose one more program restriction; we focus on programs where the higher-order arguments are either variables or predicate names. Since the set of all predicate names is finite and since we ignore all first-order values, the set of specialization atoms is also finite and as a result, the algorithm is ensured to terminate.

The rest of the section is organized as follows; in Subsection 5.3.2 we formally define the fragment of the higher-order logic programs we will use as an input; in Subsection 5.3.3 we describe the abstract framework of partial evaluation; in Subsection 5.3.4 we introduce the details of our method; and finally, in Subsection 5.3.5 we discuss some details of our implementation.

5.3.2 Definitional Higher-order Logic Programs

In this subsection, we define the higher-order language of our interest. We begin with the syntax of the language $\mathcal{H}$ we use throughout the section. $\mathcal{H}$ is based on a simple type system with two base types: o, the boolean domain, and ι, the domain of data objects. The composite types are partitioned into three classes: *functional* (assigned to function symbols), *predicate* (assigned to predicate symbols) and *argument* (assigned to parameters of predicates).

Definition 5.1. *A type can either be functional, argument, or predicate, denoted by σ, ρ and π respectively and defined as:*

$$\begin{array}{rcl|l} \sigma & := & \iota & (\iota \rightarrow \sigma) \\ \pi & := & o & (\rho \rightarrow \pi) \\ \rho & := & \iota & \pi \end{array}$$

Definition 5.2. *The alphabet of the language $\mathcal{H}$ consists of the following:*

1. *Predicate variables of every predicate type π (denoted by capital letters such as* $\mathsf{P}, \mathsf{Q}, \mathsf{R}, \ldots$*).*
2. *Individual variables of type ι (denoted by capital letters such as* $\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots$*).*
3. *Predicate constants of every predicate type π (denoted by lowercase letters such as* $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$*).*
4. *Individual constants of type ι (denoted by lowercase letters such as* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$*).*
5. *Function symbols of every functional type $\sigma \neq \iota$ (denoted by lowercase letters such as* $\mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots$*).*
6. *The inverse implication constant $\leftarrow$, the negation constant $\sim$, the comma, the left and right parentheses, and the equality constant $\approx$ for comparing terms of type ι.*

The set consisting of the predicate variables and the individual variables of $\mathcal{H}$ will be called the set of *argument variables* of $\mathcal{H}$. Argument variables will be usually denoted by V and its subscripted versions.

Definition 5.3. *The set of expressions of $\mathcal{H}$ is defined as follows:*

- *Every predicate variable (resp. predicate constant) of type π is an expression of type π; every individual variable (resp. individual constant) of type ι is an expression of type ι;*
- *if* f *is an n-ary function symbol and* $\mathsf{E}_1, \ldots, \mathsf{E}_n$ *are expressions of type ι then* $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$ *is an expression of type ι;*
- *if* P *is a predicate variable or a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o$ and* E_i *an expression of type ρ_i, $1 \leq i \leq n$, then* $(\mathsf{P}\ \mathsf{E}_1\ \cdots\ \mathsf{E}_n)$ *is an expression of type o.*
- *if* $\mathsf{E}_1, \mathsf{E}_2$ *are expressions of type ι, then* $(\mathsf{E}_1 \approx \mathsf{E}_2)$ *is an expression of type o.*

We will omit parentheses when no confusion arises. Expressions of type o will often be referred to as *atoms*. We write *vars*(E) to denote the set of all variables in E. We say that E_i is the i-th *argument* of an atom $\mathsf{P}\ \mathsf{E}_1\ \cdots\ \mathsf{E}_n$. A *ground expression* E is an expression where *vars*(E) is the empty set.

Definition 5.4. *A clause is a formula*

$$\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n\ \leftarrow\ \mathsf{L}_1, \ldots, \mathsf{L}_m, \sim \mathsf{L}_{m+1}, \ldots, \sim \mathsf{L}_{m+k}$$

where p *is a predicate constant of type $\rho_1 \to \cdots \to \rho_n \to o$,* $\mathsf{V}_1, \ldots, \mathsf{V}_n$ *are distinct variables of types $\rho_1, \ldots, \rho_n$ respectively, and* $\mathsf{L}_1, \ldots, \mathsf{L}_{m+k}$ *are expressions of type o, such that every predicate argument of* L_i*, $1 \leq i \leq m+k$, is either variable or ground. A program* P *of the higher-order language $\mathcal{H}$ is a finite set of program clauses.*

The syntax of programs given in Definition 5.4 differs slightly from the usual HiLog syntax that we have used in Chapter 4 and in Example 5.4. However, one can easily verify that we can rewrite every program from the former syntax to the latter. For instance, we could use the constant $\approx$ in order to eliminate individual constants that appear in the head of a clause that uses the HiLog syntax.

Example 5.5. *Consider the following program in HiLog syntax, in which we have three predicate definitions, namely* $\mathtt{p} : \iota \to o$, $\mathtt{q} : \iota \to \iota \to o$, *and* $\mathtt{r} : (\iota \to o) \to (\iota \to o) \to (\iota \to \iota) \to o$.

```
p(a).
q(X,X).
r(P,Q,f(X)) :- P(X),Q(Y).
```

In our more formal notation, these clauses can be rewritten as:

$\mathtt{p}\ \mathtt{X} \leftarrow (\mathtt{X} \approx \mathtt{a}).$
$\mathtt{q}\ \mathtt{X}\ \mathtt{Y} \leftarrow (\mathtt{X} \approx \mathtt{Y}).$
$\mathtt{r}\ \mathtt{P}\ \mathtt{Q}\ \mathtt{Z} \leftarrow (\mathtt{Z} \approx \mathtt{f}(\mathtt{X})),\ (\mathtt{P}\ \mathtt{X}),\ (\mathtt{Q}\ \mathtt{Y}).$

Notice that all clauses are now valid $\mathcal{H}$ *clauses.*

Notice that in a $\mathcal{H}$ program, all arguments of predicate type are either variables or predicate names, which as discussed in Subsection 5.3.1 leads to the termination of our technique. However, in a $\mathcal{H}$ program, all head predicate variables are required to be distinct. That implies that checking for equality between predicates (higher-order unification) is forbidden. In other words, the higher-order parameters can be used in ways similar to functional programming, namely either invoked or passed as arguments. We decided to impose this restriction because equality between predicates is treated differently in various higher-order languages [10, 15, 52]. Moreover, in Subsection 5.3.1, we briefly discussed that the reason why our technique can produce a first-order program is due to the following property:

Definition 5.5. *A clause is called definitional if every predicate variable that appears in the body appears also as a formal parameter of the clause. A definitional program is a finite set of definitional clauses.*

Example 5.6. *Consider the following program in HiLog syntax:*

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

This program does not belong to our fragment, because the first clause is a non-$\mathcal{H}$ *clause and the second clause is a non-definitional clause. Regarding the first clause, the predicate variable* `Q` *appears twice in the head, therefore the formal parameters are not distinct. Regarding the second clause, the predicate variable* `R` *that appears in the body, does not appear in the head of the clause.*

We extend the notion of substitution from classical logic programming to apply to $\mathcal{H}$ programs.

Definition 5.6. *A substitution* θ *is a finite set* $\{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_n/\mathsf{E}_n\}$ *where the* V_i*'s are different argument variables and each* E_i *is a term having the same type as* V_i*. We write dom*$(\theta) = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ *to denote the domain of* θ*.*

Definition 5.7. *Let θ be a substitution and* E *be an expression of $\mathcal{H}$. Then,* Eθ *is an expression obtained from* E *as follows:*

- $\mathsf{E}\theta = \mathsf{E}$ *if* E *is a predicate constant or individual constant;*
- $\mathsf{V}\theta = \theta(\mathsf{V})$ *if* $\mathsf{V} \in$ *dom*(θ)*; otherwise,* $\mathsf{V}\theta = \mathsf{V}$*;*
- $(\mathsf{f}\ \mathsf{E}_1 \cdots \mathsf{E}_n)\theta = (\mathsf{f}\ \mathsf{E}_1\theta \cdots \mathsf{E}_n\theta)$*;*
- $(\mathsf{P}\ \mathsf{E}_1\ \cdots\ \mathsf{E}_n)\theta = (\mathsf{P}\theta\ \mathsf{E}_1\theta\ \cdots\ \mathsf{E}_n\theta)$*;*
- $(\mathsf{E}_1 \approx \mathsf{E}_2)\theta = (\mathsf{E}_1\theta \approx \mathsf{E}_2\theta)$*;*
- $(\mathsf{L}_1, \ldots, \mathsf{L}_m, \sim \mathsf{L}_{m+1}, \ldots, \sim \mathsf{L}_n)\theta = (\mathsf{L}_1\theta, \ldots, \mathsf{L}_m\theta, \sim (\mathsf{L}_{m+1}\theta), \ldots, \sim (\mathsf{L}_n\theta))$.

Definition 5.8. *Let θ be a substitution and* E *be an expression. Then,* Eθ *is called an instance of* E.

5.3.3 Partial Evaluation of Logic Programs

In this subsection, we describe the abstract framework of partial evaluation. Our program optimization technique makes heavy use of this framework, since predicate specialization is an instance of partial evaluation for a specific type of higher-order logic programs.

Partial evaluation [40] is a program optimization that specializes a given program according to a specific set of input data, such that the new program is more efficient than the original and both programs behave in the same way according to the given data. In the context of logic programming [29, 47, 51], a partial evaluation algorithm takes a program P and a goal G and produces a new program $\mathsf{P}_{\mathsf{spec}}$ such that $\mathsf{P} \cup \{\mathsf{G}\}$ and $\mathsf{P}_{\mathsf{spec}} \cup \{\mathsf{G}\}$ are semantically equivalent. In Figure 5.1 we illustrate a basic scheme that aims to describe every partial evaluation algorithm in logic programming, which is based on similar ones in the literature [29, 47]. This general algorithm depends on two operations, namely Unfold and Abstract, which can be implemented differently in several partial evaluation systems.

Firstly, the algorithm uses an *unfolding* rule [68, 71] in order to construct a finite and possibly incomplete proof tree for every atom in the set S and then creates a program $\mathsf{P}_{\mathsf{spec}}$ such that every clause of it is constructed from all root-to-leaf derivations of these proof trees. This part of the process is referred to as the *local control* of partial evaluation. There are many possible unfolding rules, some of which being more useful for a particular

1: **Input:** a program P and a goal G
2: **Output:** a specialized program $\mathsf{P}_{\mathsf{spec}}$
3: $\mathsf{S} := \{\mathsf{A} : \mathsf{A}$ is an atom of $\mathsf{G}\}$
4: **repeat**
5: $\quad \mathsf{S}_{\mathsf{old}} := \mathsf{S}$
6: $\quad \mathsf{P}_{\mathsf{spec}} := \mathsf{Unfold}(\mathsf{P}, \mathsf{S})$
7: $\quad \mathsf{S} := \mathsf{S} \cup \{\mathsf{A} : \mathsf{A}$ is an atom that appears in a body of a clause in $\mathsf{P}_{\mathsf{spec}}\}$
8: $\quad \mathsf{S} := \mathsf{Abstract}(\mathsf{S})$
9: **until** $\mathsf{S}_{\mathsf{old}} = \mathsf{S}$ (modulo variable renaming)
10: **return** $\mathsf{P}_{\mathsf{spec}}$

Figure 5.1: Basic algorithm for Partial Evaluation.

application than others. There exist many types of unfolding strategies [29, 47]. In some cases though, taking a simple approach which performs no unfolding at all, or in other words by using *one-step unfolding strategy*, may result in useful program optimizations. In such a case, Unfold exports a program that is constructed by finding the clauses that unify with each atom in S and then by specializing these clauses accordingly, using simple variable substitutions.

Secondly, the algorithm uses an Abstract operation, which calculates a finite *abstraction* of the set S. We say that S′ is an *abstraction* of S if every atom of S is an instance of some atom in S′, and there does not exist two atoms in S′ that have a common instance in S′. This operation is used to keep the size of the set of atoms S finite, which will ensure the termination of the algorithm. This part of the process is referred to as the *global control* of partial evaluation. There exist many types of abstraction operators [47, 48]. An example is to distinguish between static and dynamic arguments and then to generalize away all dynamic arguments for every atom in S [48].

A partial evaluation algorithm should ensure termination in both levels of control. Firstly, we have the *local termination problem*, which is the problem of the non-termination of the unfolding rule, and the *global termination problem* which is the problem of the non-termination of the iteration process (i.e., the repeat loop in the algorithm). As we stated earlier, the global termination problem is solved by keeping the set S finite through a finite abstraction operation. Regarding the local termination problem, one possible solution is ensuring that all constructed proof trees are finite. The one-step unfolding rule is by definition a strategy that can ensure local termination.

5.3.4 Predicate Specialization

In this subsection, we define our technique using the standard framework of partial evaluation (ref. Subsection 5.3.3), by specifying its local and global control strategies (namely Unfold and Abstract operations). In particular, we will use a one-step unfolding rule and an abstraction operation which generalizes all individual (i.e., non-predicate) arguments from all atoms of the partial evaluation.

Definition 5.9. *Let* P *be a* $\mathcal{H}$ *program and* S *be a set of atoms. Then,*

$$\mathsf{Unfold}(\mathsf{P},\mathsf{S}) = \left\{ \mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n \leftarrow \mathsf{B}\theta : \begin{array}{l} (\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n) \in \mathsf{S}, \\ (\mathsf{p}\ \mathsf{V}_1 \cdots \mathsf{V}_n \leftarrow \mathsf{B}) \in \mathsf{P}, \\ \theta = \{\mathsf{V}_1/\mathsf{E}_1, \ldots, \mathsf{V}_n/\mathsf{E}_n\} \end{array} \right\}$$

Definition 5.10. *Let* S *be a set of atoms. Then,*

$$\mathsf{Abstract}(\mathsf{S}) = \left\{ \mathsf{p}\ \mathsf{E}'_1 \cdots \mathsf{E}'_n : (\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n) \in \mathsf{S} \right\}$$

where $\mathsf{E}'_i = \mathsf{E}_i$ *if* E_i *is of predicate type, otherwise* $\mathsf{E}'_i = \mathsf{V}_{\mathsf{p},i}$*, where* $\mathsf{V}_{\mathsf{p},i}$ *is a variable of the same type as of* E_i*.*

Example 5.7. *Consider a* $\mathcal{H}$ *program* P*, encoded in the HiLog syntax, that contains the predicate* `union` $: (\iota \to o) \to (\iota \to o) \to \iota \to o$ *with the following definition:*

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

together with the definitions of three first-order binary predicates $\mathtt{p} : \iota \to o$, $\mathtt{q} : \iota \to o$ *and* $\mathtt{r} : \iota \to o$. *Then,*

$$\text{Unfold}\left(\mathsf{P}, \left\{ \begin{array}{l} \mathtt{union(p,q,X)}, \\ \mathtt{union(q,r,X)} \end{array} \right\}\right) = \left\{ \begin{array}{l} \mathtt{union(p,q,X)\ :\text{-}\ p(X).} \\ \mathtt{union(p,q,X)\ :\text{-}\ q(X).} \\ \mathtt{union(q,r,X)\ :\text{-}\ q(X).} \\ \mathtt{union(q,r,X)\ :\text{-}\ r(X).} \end{array} \right\}$$

$$\text{Abstract}\left(\left\{ \begin{array}{c} \mathtt{union(p,r,0)}, \\ \mathtt{union(p,r,s(0))}, \\ \mathtt{union(p,r,s(s(0)))}, \\ \mathtt{union(p,r,s(s(s(0))))}, \\ \ldots \end{array} \right\}\right) = \left\{ \mathtt{union(p,r,V)} \right\}.$$

Notice that we use the variable name $\mathtt{V}$ *instead of* $\mathsf{V}_{\mathtt{union},3}$ *for simplicity.*

In the following lemma, we make the assumption that all atoms of S are constructed using predicate constants from a finite set C of predicate constants. We argue that it is safe to make this assumption because during every stage of the algorithm of Figure 5.1 all atoms of S are obtained either from initial goal G or from the rules of the input program P, which both contain a finite set of predicate constants.

Lemma 5.1. *Let* P *be a definitional program,* C *be a finite set of predicate constants, and* S *be a (possibly infinite) set of atoms such that every predicate argument of every atom in* S *is predicate constant from* C. *Then:*

1. *If* S *is finite, then* $\text{Unfold}(\mathsf{P}, \mathsf{S})$ *is finite.*
2. $\text{Abstract}(\mathsf{S})$ *is a finite abstraction of* S.
3. *Every atom of* $\text{Unfold}(\mathsf{P}, \mathsf{S})$ *does not contain any predicate variables.*

Proof. 1. Obvious from the construction of $\text{Unfold}(\mathsf{P}, \mathsf{S})$.

2. From the construction of $\text{Abstract}(\mathsf{S})$ it is obvious that $\text{Abstract}(\mathsf{S})$ is an abstraction of S. Let $\mathsf{A} \in \text{Abstract}(\mathsf{S})$. Then, every predicate argument of A is a predicate constant (taken from a finite set of prediate constants C) and every individual argument of A is a unique individual variable. Therefore, $\text{Abstract}(\mathsf{S})$ is finite.

3. Suppose that $\text{Unfold}(\mathsf{P}, \mathsf{S})$ contains an atom A that contains a predicate variable V. If A appears in the head of a clause, then from the construction of $\text{Unfold}(\mathsf{P}, \mathsf{S})$, S must contain A. If A appears in the body of a clause, then since P is definitional, V also appears in the head of this clause. In any case, S must contain an atom that contains the predicate variable V. □

The first part of the lemma ensures local termination and the second part of the lemma ensures global termination. The third part identifies that the transformation to first-order succeeds, provided that the program belongs to our fragment and the initial goal does not contain higher-order variables. In the following corollaries, by Φ we denote the algorithm of Figure 5.1 combined with the operations in Definition 5.9 and Definition 5.10.

Corollary 5.1. *Let* P *be a definitional program and* G *a goal. Then, the computation of* $\Phi(\mathsf{P}, \mathsf{G})$ *terminates in a finite number of steps.*

Corollary 5.2. *Let* P *be a definitional program and* G *a goal that does not contain any predicate variables. Then, the output of* $\Phi(\mathsf{P}, \mathsf{G})$ *does not contain any predicate variables.*

The result of Φ is neither a valid $\mathcal{H}$ program since it contains predicate names as arguments in the heads, nor a valid first-order program since some symbols appear both as arguments and as predicate symbols. Therefore, we must apply a simple *renaming* [29, Section 3] in order to construct a valid first-order output. In our case, at the end of the partial evaluation algorithm, every atom $\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n$ of S is renamed into $\mathsf{p}'\ \mathsf{V}_1 \cdots \mathsf{V}_m$, where p' is a fresh predicate symbol and $\{\mathsf{V}_1 \cdots \mathsf{V}_m\} = \mathit{vars}(\mathsf{p}\ \mathsf{E}_1 \cdots \mathsf{E}_n)$. Moreover, all instances of every atom of S in the resulting program are renamed accordingly.

5.3.5 Implementation

We have developed a prototype implementation[9] of our predicate specialization technique. The source language is HiLog and the target language is Prolog.

A feature that we need and is not supported in HiLog though, is the use of types. Our algorithm needs types not only for deciding whether the input program belongs to our fragment, but also for the abstraction operation in Definition 5.10. Since the process of extending HiLog with types is outside the scope of this dissertation, we assume that the input programs are well-typed and accompanied with type annotations for all predicates that contain predicate arguments.

The implementation of our predicate specialization technique is approximately 700 lines of Prolog code and is realized for the XSB system.

5.4 Predicate Specialization and Preferential Higher-order Logic Programs

In this section, we consider higher-order logic programs that express preferences over tuples. Since these programs do not contain free predicate variables in the bodies of the clauses, we can use Predicate Specialization in order to optimize their program execution time. This technique can be used also for preference operators other than `winnow`, even though in some cases (e.g., the operator `w`) the programs are not contained in the fragment that is considered by Predicate Specialization. Transforming programs and queries into first-order ones has important benefits from a practical point of view, as we will see in the experiments of Chapter 6.

Most higher-order logic programs that express tuple preferences have a relatively simple structure; they consist of a preference relation definition part (which usually is a first-order program) and a query using `winnow`. Such programs can be successfully transformed into first-order using Predicate Specialization. For instance, Example 5.4 produces a first-order transformation of `winnow` according to the following query:

```
?- winnow(c1_pref,movie)(T).
```

However, not all preferential programs that we have considered in Section 4.4 belong to the fragment that was discussed in Subsection 5.3.2, because in this fragment the only

[9] cf. `http://bitbucket.org/antru/firstify`

elements that can appear as predicate arguments are variables and predicate constants. Consider for example the following queries:

```
?- winnow(prioritized(c1_pref,c2_pref),movie)(T).
?- w(c1_pref,movie)(2)(T).
```

These examples use *partial applications*, which is a useful feature of most higher-order languages (and HiLog among them). This feature is the ability to invoke a higher-order predicate with only some of its arguments. In the first goal, notice that the first argument (which is a predicate argument) is not a predicate constant, but the partial application `prioritized(c1_pref,c2_pref)`. In the second goal, even though the predicate arguments are predicate constants, several partial applications occur in the clauses that define the `w` operator. Predicate Specialization can produce a first-order translation in these cases, even though these examples do not belong in the class of definitional $\mathcal{H}$ programs.

In the following, we present three examples of programs or queries that do not belong to the fragment of Subsection 5.3.2. In the first two cases the process of the transformation terminates and provides a first-order translation, while in the third case the transformation does not terminate. Moreover, in the following examples, by writing $\{\mathsf{A}_1 \rightsquigarrow \mathsf{B}_1, \ldots, \mathsf{A}_n \rightsquigarrow \mathsf{B}_n\}$ we mean that the input program was specialized according to the set of specialization atoms $\{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$ and every higher-order atom A_i is renamed into B_i in the output program.

Example 5.8. *Consider the* `c1_pref`*,* `c2_pref` *and* `movie` *relations from Section 4.4 and the following clauses:*

```
prioritized(C1,C2)(T1,T2) :- C1(T1,T2).
prioritized(C1,C2)(T1,T2) :- indifferent(C1)(T1,T2), C2(T1,T2).
indifferent(C)(T1,T2) :- not C(T1,T2), not C(T2,T1).
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(Z), C(Z,T).
```

and the query:

```
?- winnow(prioritized(c1_pref,c2_pref),movie)(T).
```

By applying Predicate Specialization to the above program, we get the following program:

```
winnow1(T) :- movie(T), not bypassed2(T).
bypassed2(T) :- movie(Z), prioritized3(Z,T).
prioritized3(T1,T2) :- c1_pref(T1,T2).
prioritized3(T1,T2) :- indifferent4(T1,T2), c2_pref(T1,T2).
indifferent4(T1,T2) :- not c1_pref(T1,T2), not c1_pref(T2,T1).
```

with the following renamings:

$$\left\{\begin{array}{rcl}
\texttt{winnow(prioritized(c1_pref,c2_pref),movie)(T)} & \rightsquigarrow & \texttt{winnow1(T)} \\
\texttt{bypassed(prioritized(c1_pref,c2_pref),movie)(T)} & \rightsquigarrow & \texttt{bypassed2(T)} \\
\texttt{prioritized(c1_pref,c2_pref)(T1,T2)} & \rightsquigarrow & \texttt{prioritized3(T1,T2)} \\
\texttt{indifferent(c1_pref)(T1,T2)} & \rightsquigarrow & \texttt{indifferent4(T1,T2)}
\end{array}\right\}$$

The above query can now be stated as:

```
?- winnow1(T).
```

Example 5.9. *Consider a binary preference relation* `c` *and a unary base relation* `m` *and the following clauses:*

```
w(C,R)(1)(T) :- winnow(C,R)(T).
w(C,R)(N)(T) :- N>1,M is N-1,
                winnow(C,diff(R,gen_union(w(C,R))(M)))(T).
winnow(C,R)(T) :- R(T), not bypassed(C,R)(T).
bypassed(C,R)(T) :- R(Z), C(Z,T).
gen_union(R)(1)(T) :- R(1)(T).
gen_union(R)(M)(T) :- M>1, K is M-1,
                      union(gen_union(R)(K), R(M))(T).
union(R,Q)(T) :- R(T).
union(R,Q)(T) :- Q(T).
diff(R,Q)(T) :- R(T), not Q(T).
```

and the query:

```
?- w(c,m)(2)(T).
```

By applying Predicate Specialization to the above program, we get the following program:

```
w1(1,T) :- winnow2(T).
w1(N,T) :- N > 1, M is N-1, winnow3(M,T).
winnow2(T) :- m(T), not bypassed4(T).
bypassed4(T) :- m(Z), c(Z,T).
winnow3(M,T) :- diff5(M,T), not bypassed6(M,T).
bypassed6(M,T) :- diff5(M,Z), c(Z,T).
gen_union7(1,T) :- w1(1,T).
gen_union7(M,T) :- M > 1, K is M-1, union8(K,M,T).
union8(K,M,T) :- gen_union7(K,T).
union8(K,M,T) :- w1(M,T).
diff5(M,T) :- m(T), not gen_union7(M,T).
```

with the following renamings:

$$\left\{\begin{array}{rcl}
\texttt{w(c,m)(N)(T)} & \rightsquigarrow & \texttt{w1(N,T)} \\
\texttt{winnow(c,m)(T)} & \rightsquigarrow & \texttt{winnow2(T)} \\
\texttt{bypassed(c,m)(T)} & \rightsquigarrow & \texttt{bypassed4(T)} \\
\texttt{winnow(c,diff(m,gen_union((w(c,m)),M)))(T)} & \rightsquigarrow & \texttt{winnow3(M,T)} \\
\texttt{bypassed(c,diff(m,gen_union(w(c,m))(M)))(T)} & \rightsquigarrow & \texttt{bypassed6(M,T)} \\
\texttt{gen_union(w(c,m))(M)(T)} & \rightsquigarrow & \texttt{gen_union7(M,T)} \\
\texttt{union(gen_union(w(c,m))(K),w(c,m)(M))(T)} & \rightsquigarrow & \texttt{union8(K,M,T)} \\
\texttt{diff(m,gen_union(w(c,m))(M)))(T)} & \rightsquigarrow & \texttt{diff5(M,T)}
\end{array}\right\}$$

The above query can now be stated as:

```
?- w1(2,T).
```

Example 5.10. *Consider the* `director_pref`, `movie`, `winnow` *and* `bypassed` *predicates from Section 4.4 and the query:*

```
?- winnow(director_pref,movie)(T).
```

Predicate Specialization in this case does not terminate. The non-termination occurs during the attempt of specializing the predicate `size`*. This predicate is defined using the following clauses:*

```
size(R,0) :- empty(R).
size(R,N) :- R(X), size(minus(R,X),M), N is M+1.
```

This behavior is happening due to the fact that the specialization of `size` *according to* `size(r,N)` *causes the specialization of* `size` *according to* `size(minus(r,X1),N1)` *which causes the specialization of* `size` *according to* `size(minus(minus(r,X2),X1),N2)` *and so on, which leads to non-termination.*

The above examples illustrate that Predicate Specialization can be effective in the case of programs that make limited use of complex predicate expressions. In particular, it seems that the specialization of non-recursive programs (cf. Example 5.8) does not have termination problems, but the specialization of programs that combine higher-order features together with recursion may terminate (cf. Example 5.9) or not (cf. Example 5.10). In order to spot the crucial difference between the terminating and the non-terminating examples, consider the clauses that define the recursive predicates `w`, `gen_union` and `size`. Notice that in the terminating example, the recursive calls `w(C,R)` and `gen_union(R)(K)` that appear in the bodies of the clauses do not contain complex predicate expressions as arguments. This is not the case for the recursive call `size(minus(R,X),M)`, which is the reason why the specialization process does not terminate. Intuitively speaking, Predicate Specialization can be effective in HiLog programs that contain clauses of the form:

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

where a non-variable and non-constant predicate argument `r(...)` appears in the expression `p(..., r(...), ...)`, *only if* the predicates `p` and `q` do not belong in the same cycle in the *predicate dependency graph*[10]. The transformation, in this case, is ensured to terminate (because due to the form of the program all predicate variables of a predicate that depends on itself have to be specialized only with predicate names and therefore the set of all possible specialization atoms will remain finite). It is worth mentioning though, that this class of programs has the same expressive power as that of the fragment of Subsection 5.3.2, and this extension can be viewed as a "syntactic sugar" of definitional $\mathcal{H}$ programs[11].

[10] An edge from the predicate p to predicate q in the predicate dependency graph means that there exists a clause that p appears in the head and q appears in the body of the same clause. Recursive definitions create cycles in the predicate dependency graph of the program.

[11] It is not hard to show that every program of this extended fragment can be translated into an equivalent program that does not use partial applications. As a simple example, consider the following program:

```
conj3(P,Q,R)(X) :- conj2(P,conj2(Q,R))(X).
conj2(P,Q)(X) :- P(X), Q(X).
```

This program is equivalent to the following that does not use any partial applications:

```
conj3(P,Q,R,X) :- P(X), conj2(Q,R,X).
conj2(P,Q,X) :- P(X), Q(X).
```

Interestingly, we can use Predicate Specialization to convert a program of the extended fragment into its equivalent HiLog program without partial applications. This can be done by initializing the transformation process with a query that consists of the top predicate (here `?- conj3(P,Q,R)(X)`).

Regarding the optimization of the `size` predicate, which is the only predicate that cannot be transformed into first-order by Predicate Specialization, we can use an alternative implementation using second-order Prolog built-ins[12]. Even though this approach is not technically a first-order one, it can be useful since it can be executed in almost every practical Prolog system.

5.5 Optimization Strategies for Set Preferences

In this section, we present two optimization techniques for higher-order logic programs that express preferences over sets. In particular, we give a relatively high-level description of two optimizations and of their implementation in our framework. For more details, the interested reader should consult the detailed exposition of these optimizations in the work of Zhang and Chomicki [80].

5.5.1 Overview of the Optimizations

As discussed in Section 5.2, HiLog's approach for treating uninstantiated predicate variables in queries is to instantiate them with predicate names that are defined in the program. By trying all possible instantiations for such variables we have a guarantee that if there exists a predicate that satisfies the query, it will be eventually found by the system. Therefore, it is evident that the performance of query execution in the case of preferences over sets heavily depends on the number of the predicate names that must be substituted and checked, or equivalently the number of the sets that are generated by the `subsetof` predicate (which was described in Section 5.2). This process is inherently slow in the worst case because there exist applications where all the subsets of a specific size must be tested. Therefore, it would be beneficial to devise techniques that will reduce the number of the subsets generated and at the same time will not compromise the soundness of the implementation.

There exist many applications where optimizations may help to get a performance that is acceptable in practice. In our HiLog implementation, we have experimented with the two main optimization techniques that have been proposed by Zhang and Chomicki [80] for speeding-up the subset generation process during query processing, namely *superpreference* and *M-relation*. Each technique uses a mechanism for pruning the set of the candidate k-subsets. Intuitively, superpreference removes tuples that will not contribute to the production of any best k-subset, while the M-relation "groups together" tuples that are exchangeable with respect to a given set preference. It should be noted that these two optimizations will not work for every preference relation between sets, but it can be applied only on preferences that are *additive*, a class that is quite common in practice.

In the following, we present the notion of additive preferences and we give a simple example. Recall the definitions of *profiles* and *features* from Subsection 4.2.3. Suppose that we are given a relation r. A feature $\mathcal{F}$ is a function that maps the subsets of r to a numerical value, and the profile of a given set is a vector of all features of this set. We say that a feature $\mathcal{F}$ is *additive* if there exists a function f such that (1) for every tuple $t \in r$ it holds $\mathcal{F}(\{t\}) = f(t)$ and (2) for every subset $s \subseteq r$ and for every tuple $t \in r - s$ it holds $\mathcal{F}(s \cup \{t\}) = \mathcal{F}(s) \cup f(t)$. Finally, we say that a set preference is additive if all

[12] A possible implementation would be the `size(R,N) :- findall(X,R(X),L), length(L,N).` which is the implementation that we used in our experiments.

Table 5.1: A simple, short `movie` relation.

ID	Name	Year	Runtime	Rating
m_1	Raiders of the Lost Ark	1981	115	8.5
m_2	The Terminator	1984	107	8.0
m_3	Lethal Weapon	1987	109	7.5
m_4	Die Hard	1988	132	8.0
m_5	Terminator 2: Judgment Day	1991	137	8.5

Table 5.2: The profile relation of all 3-element subsets of the `movie` relation of Table 5.1.

Set	$\mathcal{F}(\cdot)$
$s_1 = \{m_1, m_2, m_3\}$	$f(m_1) + f(m_2) + f(m_3) = 8.5 + 8.0 + 7.5 = 24.0$
$s_2 = \{m_1, m_2, m_4\}$	$f(m_1) + f(m_2) + f(m_4) = 8.5 + 8.0 + 8.0 = 24.5$
$s_3 = \{m_1, m_2, m_5\}$	$f(m_1) + f(m_2) + f(m_5) = 8.5 + 8.0 + 8.5 = 25.0$
$s_4 = \{m_1, m_3, m_4\}$	$f(m_1) + f(m_3) + f(m_4) = 8.5 + 7.5 + 8.0 = 24.0$
$s_5 = \{m_1, m_3, m_5\}$	$f(m_1) + f(m_3) + f(m_5) = 8.5 + 7.5 + 8.5 = 24.5$
$s_6 = \{m_1, m_4, m_5\}$	$f(m_1) + f(m_4) + f(m_5) = 8.5 + 8.0 + 8.5 = 25.0$
$s_7 = \{m_2, m_3, m_4\}$	$f(m_2) + f(m_3) + f(m_4) = 8.0 + 7.5 + 8.0 = 23.5$
$s_8 = \{m_2, m_3, m_5\}$	$f(m_2) + f(m_3) + f(m_5) = 8.0 + 7.5 + 8.5 = 24.0$
$s_9 = \{m_2, m_4, m_5\}$	$f(m_2) + f(m_4) + f(m_5) = 8.0 + 8.0 + 8.5 = 24.5$
$s_{10} = \{m_3, m_4, m_5\}$	$f(m_3) + f(m_4) + f(m_5) = 7.5 + 8.0 + 8.5 = 24.0$

features of the profile of the set is an additive feature. The following example illustrates a simple additive preference relation over sets of movies.

Example 5.11. *Consider the* `movie((ID,Name,Runtime,Rating))` *relation illustrated in Table 5.1. Now, suppose that we want to watch three movies and we prefer that the sum of the ratings of these movies to be as high as possible (he have already seen this set preference relation in Example 4.5 and Example 4.16).*

This set preference is additive since the profile of each set consists of of one feature $\mathcal{F}$, and this feature is additive, since it holds $\mathcal{F}(\{t_1, t_2, t_3\}) = f(t_1) + f(t_2) + f(t_3)$, where $f(t)$ is the rating of the movie t. A set s is preferred to s' if it holds $\mathcal{F}(s) > \mathcal{F}(s')$. In Table 5.2 we illustrate the profile relation of all 3-element subsets of `movie`*. As we have seen in Section 4.5, the best sets are found using a winnow query on this profile relation, but this table has to be constructed beforehand.*

The higher-order predicate `rating_pref` *that expresses this preference in our higher-order framework can be found in Example 4.16.*

5.5.2 Pruning Sets by Removing Unnecessary Tuples

In this subsection, we give a description of the superpreference optimization. This optimization reduces the input tuples and the generated sets by filtering out tuples that do not belong to any "best" k-subset.

The key idea behind the superpreference optimization is that in some cases, instead of examining a preference relation between sets, we can examine a (so-called) superpreference relation *between tuples*. Given a relation r and a preference relation $\succ_C$ between

Table 5.3: Superpreference optimization for Example 5.11.

(a) Remaining tuples from the `movie` relation of Table 5.1 after the 1st Pruning condition.

ID	Name	Year	Runtime	Rating
m_1	Raiders of the Lost Ark	1981	115	8.5
m_2	The Terminator	1984	107	8.0
m_4	Die Hard	1988	132	8.0
m_5	Terminator 2: Judgment Day	1991	137	8.5

(b) The profile relation of all 3-element subsets of the relation of Table 5.4a. The third column shows if the specific set was pruned according to the 2nd Pruning condition.

Set	$\mathcal{F}(\cdot)$	pruned?
$\{m_1, m_2, m_4\}$	24.5	yes*
$\{m_2, m_4, m_5\}$	24.5	yes*
$\{m_1, m_2, m_5\}$	25.0	no†
$\{m_1, m_4, m_5\}$	25.0	no†

* This set s is pruned because m_5 (resp. m_1) $\succ^+ m_2$ and m_5 (resp. m_1) $\notin s$.
† This set s is not pruned because for every $m_i \in s$ it holds $\{m : m \succ^+ m_i\} \supseteq s$.

k-subsets of r, we say that $\succ_C^+$ is the corresponding superpreference relation between tuples of r iff it holds $t_1 \succ_C^+ t_2 \iff \{t_1\} \cup s \succ_C \{t_2\} \cup s$, for every $(k-1)$-subset s or $r - \{t_1, t_2\}$.

Consider again the preference relation from Example 5.11 which is defined over 3-element sets. It is not hard to see that this preference relation is closely connected to the following superpreference relation among tuples. A movie tuple is "super-preferred" over another if the rating of the former is higher than the rating of the latter:

```
rating_superpref((I1,N1,Y1,T1,R1), (I2,N2,Y2,T2,R2)) :-
        movie((I1,N1,Y1,T1,R1)),
        movie((I2,N2,Y2,T2,R2)),
        R1 > R2.
```

Given a superpreference relation between tuples, the superpreference optimization applies two pruning conditions, which we outline below using our running `movie` example. First, notice that the movie m_3 has at least 3 movies that are "super-preferred" from it (i.e., at least 3 movies with a higher rating). Therefore, m_3 and can be filtered out before the start of the subset generation process, because it is certain that it will not contribute to a best set (1st Pruning condition). Secondly, notice that any subset that contains the movie m_2 (or m_4) and does not contain both the movies m_1 and m_5 (i.e., it does not contain all movies that are "super-preferred" from it) is not a best subset and can be pruned at the subset generation step (2nd Pruning condition). Table 5.3 illustrates the superpreference optimization for Example 5.11.

In our implementation we provide an optimized winnow operator, called `winnowsuper`, which is enhanced according to the above two pruning conditions of the superpreference optimization. In order to get the most preferred 3-element movie sets according to the preference relation of Example 5.11, we can issue the following query:

```
?- winnowsuper(rating_pref, rating_superpref, movie, 3)(S).
```

Omitting the details, this call begins by pruning the base relation `movie` according to the 1st pruning condition, using the superpreference `rating_superpref`. Then, it generates all 3-subsets of the pruned `movie` relation, but it keeps only the appropriate sets according to the 2nd pruning condition, again using `rating_superpref`. Finally, it applies a winnow operator over this pruned set of 3-element sets using the preference relation `rating_pref` between sets (which was defined in Example 4.16) in order to get the most preferred sets. Notice that, in order for the above query to function properly, the `rating_superpref` must be the corresponding superpreference relation for the preference relation of `rating_pref`.

5.5.3 Pruning Sets by Grouping Exchangeable Tuples

In this subsection, we give a description of the M-relation optimization. This optimization reduces the generated sets by grouping tuples that are exchangeable with respect to the given set preference.

The key idea behind the M-relation optimization is that in some cases, an exchangeability relation between tuples can lead us to indifference relation between sets, which can be used as a pruning mechanism. Given a relation r and a preference relation $\succ_C$ between k-subsets of r, we say that $\approx_C$ is the corresponding exchangeability relation between tuples of r iff it holds $t_1 \approx_C t_2 \iff \{t_1\} \cup s \sim_C \{t_2\} \cup s$ for every $(k-1)$-subset s or $r - \{t_1, t_2\}$. By $s_1 \sim_C s_2$ we denote that the sets s_1, s_2 are indifferent or equally preferred (for the definition indifference relation of a given preference relation refer to Subsection 4.2.2). Intuitively, two tuples are exchangeable if they contribute equally to the profile value of a set.

Consider again the preference relation from Example 5.11 which is defined over 3-element sets. The movies m_2 and m_4 have the same rating, so the sets $\{m_1, m_2, m_5\}$ and $\{m_1, m_4, m_5\}$ have the same sum of ratings and are equally preferred. So, if we could "group" the tuples m_2 and m_4 into one meta-tuple $m_{2,4}$, we would be able to "group" these two sets into the meta-set $\{m_1, m_{2,4}, m_5\}$, thus reducing the number of candidate sets that are of equal importance, and avoiding unnecessary repetitions of sets that are equally preferred. At the end, the most preferred sets are the sets that correspond to the most preferred meta-set. The set of meta-tuples (called M-tuples) is constructed from the original tuples, by keeping only the fields that are needed for the computation of the best subsets, leaving out the fields that do not contribute in the computation of the preference relation. For instance, in the preference relation of Example 5.11, all M-tuples (i.e., the tuples of the M-relation) contain only one field, which is the rating of the specific movie. The transformation of a movie tuple into the corresponding single-field M-tuple can be encoded using this `rating_mrel` predicate:

```
rating_mrel((I,N,Y,T,Rating), (Rating)) :- movie((I,N,D,G,M,Rating)).
```

The k-subsets are generated from the M-tuples and not from the original tuples (because the number of M-tuples is smaller than the number of the original tuples). However, the set generation process here differs from its unoptimized counterpart. Since one M-tuple t corresponds to $n \geq 1$ original tuples, we must allow up to n duplicates of t in the subset generation step. This is needed because if we did not allow duplicates of the M-tuples, the set $\{m_1, m_2, m_5\}$ which is one of the most preferred sets of Example 5.11 (which is an instance of the multiset $\{m_{1,5}, m_{1,5}, m_{2,4}\}$) could not be produced. On the other hand, if

Table 5.5: M-relation optimization for Example 5.11.

(a) M-relation obtained from the `movie` relation of Table 5.1.

ID	Count	Rating
$m_{1,5}$	2	8.5
$m_{2,4}$	2	8.0
m_3	1	7.5

(b) The profile relation of all 3-multisets of the M-relation of Table 5.5a.

MultiSet	$\mathcal{F}(\cdot)$
$\{m_{1,5}, m_{1,5}, m_{2,4}\}$	25.0
$\{m_{1,5}, m_{1,5}, m_3\}$	24.5
$\{m_{1,5}, m_{2,4}, m_{2,4}\}$	24.5
$\{m_{1,5}, m_{2,4}, m_3\}$	24.0
$\{m_{2,4}, m_{2,4}, m_3\}$	23.5

(c) The 3-subsets of the `movie` relation of Table 5.1 that correspond to all 3-multisets of Table 5.5b.

MultiSet	$\mathcal{F}(\cdot)$	Corresponding Sets
$\{m_{1,5}, m_{1,5}, m_{2,4}\}$	25.0	$\{m_1, m_2, m_5\}$, $\{m_1, m_4, m_5\}$
$\{m_{1,5}, m_{1,5}, m_3\}$	24.5	$\{m_1, m_3, m_5\}$
$\{m_{1,5}, m_{2,4}, m_{2,4}\}$	24.5	$\{m_1, m_2, m_4\}$, $\{m_2, m_4, m_5\}$
$\{m_{1,5}, m_{2,4}, m_3\}$	24.0	$\{m_1, m_2, m_3\}$, $\{m_1, m_3, m_4\}$, $\{m_2, m_3, m_5\}$, $\{m_3, m_4, m_5\}$
$\{m_{2,4}, m_{2,4}, m_3\}$	23.5	$\{m_2, m_3, m_4\}$

we allowed any arbitrary number of duplicates of M-tuples, the multiset $\{m_{1,5}, m_{1,5}, m_{1,5}\}$ would also be produced, which does not correspond to a valid solution of the problem. Table 5.5 illustrates the M-relation optimization for Example 5.11.

Moreover, in our implementation, we also provide another optimized winnow operator, called `winnowmrelation`, which is enhanced with this optimization. In order to get the most preferred 3-element movie sets according to the preference relation of Example 5.11, we issue the following query:

```
?- winnowmrelation(rating_pref, rating_mrel, movie, 3)(S).
```

Omitting the details, this call begins by converting the original tuples into M-tuples using the `rating_mrel` predicate and tags every M-tuple with the number of the corresponding original tuples. Then, it generates all 3-multisubsets of the M-tuples, by allowing up to the correct number of duplicates, and after the generation step, a winnow operator is applied on these multisets, using the preference relation `rating_pref`. Notice though that since this winnow operator operates on multisets of M-tuples, the `rating_pref` predicate should be redefined in order to operate over multisets with M-tuples rather than sets of original tuples. Finally, since the resulting sets are sets of M-tuples, it concludes with a final procedure that transforms the 3-multisets with M-tuples back to 3-sets with original ones, again using the `rating_mrel` predicate. Notice that, in order for the above query to function properly, if any tuples from the `movie` relation are mapped by `rating_mrel` to the same M-tuple, then these tuples must be exchangeable.

5.5.4 Implementation

We have developed an implementation[13] of the superpreference and M-relation optimizations in the higher-order logic programming language HiLog. As we mentioned in the previous subsections, for every optimization we have implemented an enhanced version of `winnow`, namely `winnowsuper` and `winnowmrelation`. The implementation of these predicates, together with the implementation of the unoptimized `subsetof` predicate) is approximately 300 lines of HiLog code and is realized for the XSB system.

Notice that both enhanced versions of winnow require from the programmer to provide an additional helper relation that is needed for the optimization to operate; that is the superpreference relation for `winnowsuper` and a mapping from original tuples into M-tuples for `winnowmrelation`. In order to help the programmer on the task of discovering each helper relation, we implemented two predicates (namely `valid_superpreference` and `valid_mrelation`) such that if the two queries of the following form

```
?- valid_superpreference(superpref,pref,r,k).
?- valid_mrelation(mrel,pref,r,k).
```

succeed, then it means that `superpref` is the superpreference relation of the preference relation `pref` among `k`-element subsets of `r` and if any tuples from `r` are mapped by `mrel` to the same M-tuple, then these tuples are exchangeable according to the preference relation `pref` over `k`-element subsets of `r`[14]. Therefore, in such a case the `winnowsuper` and `winnowmrelation` operators are ensured to function properly.

To sum up, the fact that these optimizations can be expressed concisely and implemented efficiently in higher-order logic programming, highlights, one more time, the important advantage of the higher-order logic programming approach; that is, the ability to express preference relations, operators for processing preference relations, preference queries, and even optimizations on preference operators in the same language.

5.6 Summary

In this chapter, we undertook an implementation of our higher-order preferential framework. Instead of developing a tailor-made language only for preferential higher-order logic programs, we chose instead to use the programming language HiLog, a stable and mature logic programming language, which is integrated in the XSB system. Apart from a basic, unoptimized implementation, we considered optimization techniques for enhancing its performance.

A simple, unoptimized implementation was almost straightforward, since almost all preferential higher-order programs run directly into XSB. The only demanding part of the implementation was the process of generating the candidate sets for set-preferences; the reason is that the behavior of queries with uninstantiated predicate variables in HiLog (which is an intensional higher-order language) is not convenient for our case. As a result, we provide a suitable implementation of a predicate that generates all subsets of a given relation, that mimics the behavior of an extensional higher-order logic programming language.

[13] cf. `http://bitbucket.org/antru/holppref`

[14] These predicates operate in a naive way, meaning that they check for the superpreference and exchangeability properties for the full relation `r`. In practice, instead of using the full relation `r`, the programmer could use a carefully chosen or random subset of `r` for the test.

For optimizing HiLog programs that express preferences over tuples, we proposed Predicate Specialization, a program transformation technique based on partial evaluation. This specialization technique transforms a definitional higher-order program into an equivalent first-order program, which can be executed efficiently in conventional Prolog systems. Predicate Specialization cannot operate on every higher-order logic program, but in a modest but well defined fragment, which includes HiLog programs that express preferences over tuples that use various preference operators other than winnow.

Predicate Specialization cannot be used for programs that express preferences over sets since these sets involve queries with uninstantiated predicate variables and such programs are not definitional. Hence, for optimizing programs that express set preferences, we experimented with two optimization techniques by Zhang and Chomicki [80]. These techniques optimize the program execution time by reducing the set of candidate subsets, and can be used for additive set preferences, a class that is quite common in practice.

6. EXPERIMENTS AND EVALUATION

In this chapter, we present experimental results that suggest the feasibility of the higher-order logic programming framework for expressing preferences of Chapter 4. The framework is implemented and optimized according to the techniques presented in Chapter 5, combined with standard logic programming optimizations, such as tabling. For the evaluation of the experiments, we used the XSB system.

6.1 Overview

In this chapter, we conduct a series of experiments that highlight the feasibility of the higher-order logic programming framework for expressing preferences. The implementation of this framework was enhanced with the techniques and the optimizations presented in Chapter 5. We have built a test suite[15] to measure the query running time for several preference queries, and we carried out the following four different types of experiments:

- Winnow queries on tuple preferences.
- Queries of other preferential operators (besides winnow).
- Queries on recursively defined preference relations.
- Queries on set preferences.

The experimental evaluation of this chapter has two main purposes. The first purpose is to identify the strengths and weaknesses of the logic programming approach to preference representation. Since (to our knowledge) there do not exist any other available systems or implementations for representing preferences in a logic programming setting, we can not provide a comparative assessment of our approach. For this reason, we focus on providing experimental results regarding the performance of our approach based on increasing base relation sizes. The second purpose of this experimental evaluation is to measure the effectiveness of the proposed optimizations of preferential higher-order programs, especially when these techniques are combined with generic optimizations for classical logic programs. For this reason, we are comparing the performance between the unoptimized and optimized versions of every input program. In Table 6.1 we illustrate the techniques that were used in the optimized version of each one of the experiments.

The most important generic optimization that we use in our experimental evaluation is *memoization*. Memoization [27] is a well-known optimization for declarative programming languages, which is based on the idea that we can store already computed results of calls, which can later be retrieved and used directly when the same calls occur again. In logic programming, memoization is usually referred as *tabling*. A re-evaluation of a *tabled predicate* is avoided by memoizing (i.e., remembering) its answers. The XSB system is known for its elaborate and efficient implementation of tabling for first-order logic programs. For higher-order HiLog programs, however, XSB's tabling mechanism may not be as effective as it is for first-order ones. The reason is that in order to table any HiLog predicate one has to table all HiLog code. This may lead to high memory consumption and can be problematic for large-scale program development. As we are going to

[15] cf. `http://bitbucket.org/antru/holppref`

Table 6.1: Optimizations used in the experiments.

Section	Pref. Type	Experiment	Optimizations Used
Section 6.2	tuple	winnow	Predicate Specialization, Prolog built-ins*
Section 6.3	tuple	w(n)	Predicate Specialization, Tabled Execution
		wt(n)	Predicate Specialization, Tabled Execution
Section 6.4	tuple	naive_shortest	Predicate Specialization, Tabled Execution
		enhanced_shortest	Predicate Specialization, Tabled Execution
Section 6.5	set	winnowopt	Prolog builtins*
		winnowsuper	Superpreference Optimization
		winnowmrel	M-relation Optimization

* Implementation of `size/2` and `rating_sum/2` using second-order Prolog built-ins.

see, memoization offers significant benefits in the case of programs that use preference operators beyond winnow and in programs that use extrinsic preference relations.

All experiments were performed on a Linux Desktop PC (Ubuntu 14.04 LTS) with Intel(R) Core(TM) i7-4790 CPU, 8 GB RAM. The experiments were executed in XSB[16]. The execution time is measured using the standard `time/1` predicate. All data has been artificially generated.

The rest of the chapter is organized as follows: in Section 6.2 we experiment with winnow queries that involve tuple preferences; in Section 6.3 we experiment with preference queries that make use of preference operators other than winnow; in Section 6.4 we experiment with recursively defined preference relations; in Section 6.5 we experiment with preference queries that involve preferences over sets; and finally, in Section 6.6 we provide an additional experiment[17] in which evaluate the performance of higher-order logic programs that are optimized only using Predicate Specialization (an optimization tehcinque that we proposed in Section 5.3).

6.2 Experiments on Tuple Preferences

In the first series of experiments, we evaluate the performance of the `winnow` operator over a randomly generated `movie` relation, using the preference relations defined in Section 4.4. We compare the execution of the HiLog program of in Section 4.4 with the execution of the equivalent Prolog program that is obtained using Predicate Specialization. In the case of `director_pref`, we implement the `size` predicate with second-order Prolog built-ins.

[16]version 3.7, cf. `http://xsb.sourceforge.net/`
[17] cf. `http://bitbucket.org/antru/firstify`

We begin by generating a random relation of movies that contains n facts of the form `movie((ID,Name,Director,Genre,Runtime,Rating))`, where the first two fields are unique, the `Director` field is randomly selected from a set that contains $n/5$ director ids, the `Genre` field is randomly selected from a set that contains 22 genres, and the remaining fields are random integers, with `Runtime` ranging in $[100, 260]$ and `Rating` ranging in $[5, 95]$. Then, for every preference relation `pref` we issue in XSB the following queries:

```
?- winnow(pref,movie)(X), fail.
?- winnow_pref(X), fail.
```

which return all best tuples from `movie` with respect to the `pref` relation. The former goal corresponds to the unoptimized higher-order `winnow` query, while the latter corresponds to the optimized version. We use the intrinsic preference relations `c1_pref`, `c2_pref`, `c3_pref`, the conjunction composition `conj(c1_pref,c2_pref)`, the prioritized composition `prioritized(c2_pref,c1_pref)` and the extrinsic preference relation `director_pref`. We evaluate the system for $n = 100, 500, 1000, 2000, 4000, 8000, 10000$ facts.

The results of this experiment are shown in Table 6.2. In the first column, we illustrate the size of the `movie` relation, while in the remaining columns we illustrate the `winnow` query execution times in seconds. For each preference, we show the query execution time for both the naive and the optimized version of the query. Regarding the first five intrinsic preferences, the program execution time of the optimized Prolog version is equal to the 80% of that of the naive HiLog version. This happens due to the runtime overhead that is introduced by XSB in order to execute HiLog code. However, a more extreme time difference occurs in the execution times for the extrinsic preference `director_pref`. This behavior is expected because in order to compare two movie tuples in this case, one has to compute the size of the relation of movies that each of the two directors has directed. The transformation to first-order and the more efficient implementation of the `size` predicate with standard, second-order Prolog built-ins results to a significant speedup, and therefore, the query execution times differ by an order of magnitude.

The query execution time obviously increases as the size of the relation increases, but for similar base-relation sizes, the execution time clearly depends on the preference relation being evaluated. Since these queries return result sets of varying size, in Table 6.2c we divide the query execution time of the `winnow` query for each preference relation with the number of the returned results. This measurement shows how time-consuming is to obtain a single result and this is clearly analogous to the difficulty of each preference computation. For the extrinsic preference `director_pref` we show the different measurements for each of the optimized and unoptimized cases, while in the intrinsic ones, we display a mean value because the execution times are similar. For the simple intrinsic preferences `c1_pref`, `c2_pref`, `c3_pref` the execution times per result are quite similar and relatively low. The use of compositions of preferences though, leads to slower execution times per result, with the prioritized composition being the slowest of the two. This is due to the fact that the prioritized composition of two preference relations requires a more difficult computation than a simple conjunction. Finally, as we saw previously, the most difficult preference computation is the extrinsic preference relation `director_pref`. This fact is also highlighted in this table.

Table 6.2: Program execution times for the winnow operator.

(a) Winnow execution times for tuple preferences.

	winnow execution time (sec)					
	c1_pref		c2_pref		c3_pref	
#facts	*unoptimized*	*optimized*	*unoptimized*	*optimized*	*unoptimized*	*optimized*
100	0.07	0.08	0.04	0.05	0.02	0.05
500	0.15	0.13	0.12	0.10	0.06	0.04
1,000	0.32	0.30	0.37	0.31	0.08	0.07
2,000	1.00	0.82	1.66	1.36	0.46	0.38
4,000	3.21	2.74	10.51	8.48	2.72	2.20
8,000	13.18	10.94	58.80	45.70	18.86	15.35
10,000	22.51	21.02	96.44	78.09	49.23	39.63

(b) Winnow execution times for tuple preferences (cont.).

	winnow execution time (sec)					
	conj		prioritized		director_pref	
#facts	*unoptimized*	*optimized*	*unoptimized*	*optimized*	*unoptimized*	*optimized*
100	0.05	0.05	0.05	0.05	0.16	0.07
500	0.21	0.16	0.15	0.10	4.72	0.58
1,000	0.91	0.68	0.40	0.31	12.75	1.36
2,000	5.39	4.10	1.57	1.27	99.89	10.84
4,000	30.48	24.27	6.96	5.50	547.88	61.13
8,000	154.49	123.18	29.22	23.37	923.22	94.40
10,000	254.49	204.69	44.23	35.68	1,805.95	165.85

(c) Winnow execution times per result for tuple preferences.

	winnow execution time (sec) / # results						
						director_pref	
#facts	c1_pref	c2_pref	c3_pref	conj	prioritized	*unoptimized*	*optimized*
100	0.004	0.001	0.004	0.001	0.002	0.015	0.006
500	0.006	0.001	0.002	0.001	0.005	0.236	0.029
1,000	0.011	0.004	0.002	0.002	0.015	1.063	0.113
2,000	0.025	0.010	0.005	0.011	0.065	4.162	0.452
4,000	0.056	0.035	0.015	0.046	0.260	11.414	1.273
8,000	0.140	0.097	0.053	0.164	1.052	65.944	6.743
10,000	0.198	0.124	0.096	0.224	1.480	120.397	11.057

6.3 Experiments on Preference Operators

In the second series of experiments, we evaluate the performance of the operators w_C^n and wt_C^n that were introduced in Subsection 4.4.4. For each of the two operators, we have also implemented an optimized version, as described in Section 5.4. The optimized version is executed using tabled execution. The datasets we use are generated randomly with the same manner as in the previous class of experiments. For the evaluation we use the queries:

```
?- w(c1_pref,movie)(n)(X), fail.
?- wt(c1_pref,movie)(n)(X), fail.
?- wo(n,X), fail.
?- wto(n,X), fail.
```

where `w` and `wt` correspond to the unoptimized versions of w_C^n and wt_C^n respectively, and `wo` and `wto` are their optimized versions. We use only the `c1_pref` preference over the `movie` relation, and `n` is the desired level n at which each operator is evaluated. The results of this experiment are shown in Table 6.3. The empty entries in the table correspond to query execution times over 2 hours (7200 seconds).

Obviously, the unoptimized implementations of the w_C^n and wt_C^n operators are completely impractical. Using the optimized versions of these operators makes the execution times much more reasonable. Observe that while in the unoptimized versions the execution times increase dramatically as n increases, the corresponding execution times for the optimized versions exhibit a graceful increase. This behavior is due to the fact that for each n the computations of $w_C^n(r)$ and $wt_C^n(r)$ both require the recursive computations $w_C^1(r)$, $w_C^2(r)$, $\ldots$, $w_C^{n-1}(r)$, therefore the complexity of the overall computation is exponential. However, by using the memoization of the individual calls we avoid redundant computations. This explains both the huge time differences between the unoptimized and the optimized implementations of w_C^n and wt_C^n and the fact that the query execution time increases smoothly with every increase in n and the size of the base relation.

6.4 Experiments on Path Preferences

In the third series of experiments, we evaluate the performance of our approach for the path preferences as defined in Subsection 4.4.5. For each of the two operators, we have also implemented an optimized version, as described in Section 5.4. As in the previous experiments, the optimized version of every predicate is executed using tabled execution.

For each experiment we generate a directed acyclic graph of the form of Figure 6.1. Every directed edge of the graph has a random weight ranging in $[(k+m),\ 50 \cdot (k+m)]$. We generate several graphs with varying k and m parameters, and then we measure the execution times of the following queries, which calculate the cost of the shortest path from node a to node z:

```
?- naive_shortest(a,z,C),!.
?- enhanced_shortest(a,z,C),!.
?- naive_shortest_opt(a,z,C),!.
?- enhanced_shortest_opt(a,z,C),!.
```

As in the previous examples, the first two calls correspond exactly to the first two programs shown in Subsection 4.4.5. These programs are first, the naive implementation of

Table 6.3: Program execution times for w_C^n, wt_C^n operators.

		w(n) execution time (sec)		wt(n) execution time (sec)	
#facts	level	*unoptimized*	*optimized*	*unoptimized*	*optimized*
100	$n=1$	0.08	0.06	0.03	0.03
	$n=2$	0.08	0.03	0.09	0.03
	$n=3$	1.69	0.04	1.74	0.04
	$n=4$	42.90	0.04	44.54	0.04
500	$n=1$	0.13	0.16	0.05	0.08
	$n=2$	2.62	0.24	2.71	0.13
	$n=3$	219.96	0.18	217.63	0.18
	$n=4$	–	0.20	–	0.23
1,000	$n=1$	0.33	0.36	0.16	0.22
	$n=2$	12.50	0.39	12.60	0.41
	$n=3$	1,367.27	0.54	1,373.70	0.54
	$n=4$	–	0.64	–	0.63
2,000	$n=1$	0.95	1.02	0.54	0.77
	$n=2$	73.62	1.52	74.13	1.50
	$n=3$	–	1.93	–	1.78
	$n=4$	–	2.32	–	2.29

shortest path, in which a winnow operator is applied over a recursively defined `path` relation and the enhanced implementation, which embeds the winnow operator inside the definition of the shortest path relation itself (according to the optimal subproblem property defined by Govindarajan et al. [33]). The remaining two calls correspond to the optimized versions of the above predicates, which are derived using Predicate Specialization and Memoization. The results of this experiment are displayed in Table 6.4. As in the previous experiment, the empty entries correspond to query execution times over 2 hours (7200 seconds).

The results in Table 6.4 lead to the following two observations. First, as in the previous experiments, we notice that the non-optimized versions of the path predicates are becoming impractical for larger inputs and that this situation can be remedied by transforming the corresponding queries to first-order and using memoization. Second, we notice that in the non-optimized case, the version that applies a `winnow` over the recursive definition of paths is faster than the version that uses `winnow` inside the definition of the shortest path. In the optimized versions of these predicates though, we have the opposite behavior; the

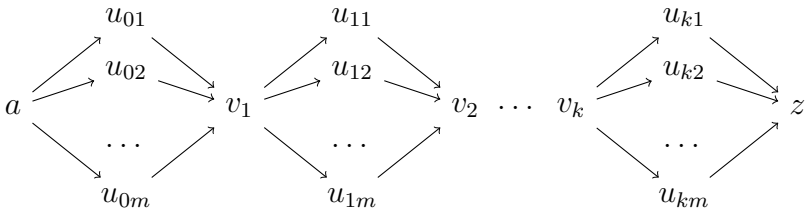

Figure 6.1: Input graph for the Shortest Path experiment.

Table 6.4: Program execution times for path preferences.

			shortest execution time (sec)			
			unoptimized		*optimized*	
# edges	graph parameters		naive	enhanced	naive	enhanced
10	$k = 1$	$m = 5$	0.06	0.03	0.07	0.04
20	$k = 1$	$m = 10$	0.05	0.03	0.03	0.03
40	$k = 1$	$m = 20$	0.05	0.05	0.04	0.04
20	$k = 2$	$m = 5$	0.04	0.07	0.04	0.04
40	$k = 2$	$m = 10$	0.27	1.84	0.06	0.03
30	$k = 3$	$m = 5$	0.14	22.54	0.03	0.03
80	$k = 2$	$m = 20$	1.95	32.84	0.06	0.04
40	$k = 4$	$m = 5$	10.47	$3,242.15$	0.08	0.05
60	$k = 3$	$m = 10$	13.22	$2,264.04$	0.12	0.03
50	$k = 5$	$m = 5$	179.74	–	0.45	0.04
80	$k = 4$	$m = 10$	–	–	1.08	0.03
100	$k = 5$	$m = 10$	–	–	1.83	0.02
200	$k = 10$	$m = 10$	–	–	20.64	0.07
400	$k = 10$	$m = 20$	–	–	98.86	0.05

former method becomes much slower and uses a very large amount of memory (this fact is not depicted in Table 6.4). Therefore, we argue that in order for an implementation that exploits the minimum subproblem property (i.e., the latter in this case) to offer an improvement in the program execution time, the solutions of the individual subproblems must be memoized. Otherwise, the execution is much slower since, as in the case of the operators `w` and `wt`, we have redundant computations of the individual subproblems. To sum up, we see that optimization techniques such that memoization and the techniques that we have developed in Section 5.4 combined with the flexibility of embedding the `winnow` operator inside the base relations, as we have shown in Subsection 4.4.5, can result to much better performance.

6.5 Experiments on Set Preferences

In this series of experiments, we evaluate the performance of our approach for set preferences. We compare the unoptimized implementation for set preferences that we described in Section 5.2 with the two optimizations for set preferences described in Section 5.5. We use the `rating_pref` set preference of Section 4.5. Since the queries of set preferences involve uninstantiated predicate variables, we could not use Predicate Specialization here.

For each experiment, we first generate n facts of the `movie` relation as described in the previous experiments. For each run of the experiments, we measure the execution times of the following four queries:

```
?- winnow(rating_pref,subsetof(movie,k)(S), fail.
?- winnow(opt_rating_pref,subsetof(movie,k)(S), fail.
?- winnowsuper(rating_pref,rating_superpref,movie,k)(S), fail.
?- winnowmrelation(rating_pref,rating_mrel,movie,k)(S), fail.
```

Table 6.5: Program execution times for set preferences.

(a) Program execution times for set preferences.

		query execution time (sec)			
#facts	set size	winnow	winnowopt	winnowsuper	winnowmrel
10	$k = 3$	0.10	0.10	0.08	0.06
25	$k = 3$	0.93	0.28	0.15	0.21
50	$k = 3$	83.27	11.41	1.16	5.14
75	$k = 3$	217.62	41.47	3.59	7.90
100	$k = 3$	2,089.67	350.43	33.02	34.53
25	$k = 2$	0.12	0.11	0.15	0.10
25	$k = 3$	0.93	0.28	0.15	0.21
25	$k = 4$	15.13	2.35	0.19	1.22
25	$k = 5$	179.18	32.51	0.22	9.55
25	$k = 6$	1,434.29	274.41	0.23	54.15

(b) Number of generated sets for set preferences.

		# sets			
#facts	set size	winnow	winnowopt	winnowsuper	winnowmrel
10	$k = 3$	120	120	1	92
25	$k = 3$	2,300	2,300	1	1,603
50	$k = 3$	19,600	19,600	1	8,808
75	$k = 3$	67,525	67,525	1	18,054
100	$k = 3$	161,700	161,700	3	37,558
25	$k = 2$	300	300	1	234
25	$k = 3$	2,300	2,300	1	1,603
25	$k = 4$	12,650	12,650	1	7,948
25	$k = 5$	53,130	53,130	1	30,384
25	$k = 6$	177,100	177,100	1	93,139

The first two calls select the best k-sets of movies according to `rating_pref` in a naive way, the third one is optimized using the "superpreference" technique, and the fourth one is optimized using the M-relation technique. The difference between the first two calls is that in the second one we use standard Prolog built-ins in order to construct an efficient version of the `rating_sum/2` predicate (which counts the sum of the ratings in a set). We evaluate the queries for fixed subset size $k = 3$ and varying base-relation sizes $n = 10, 25, 50, 75, 100$. Then, we perform again the evaluation for fixed base-relation size $n = 25$ and varying subset size $k = 2, 3, 4, 5, 6$. For each run of the experiments, we also measure the total number of sets that are generated for each approach. The query execution times are displayed in Table 6.5a and the number of the generated sets for each query are displayed in Table 6.5b.

As we can see from the results of Tables 6.5a and 6.5b, the superpreference and M-relation optimizations are both performing very well in our case, due to the effective pruning of the total number of subsets that need to be generated. Considering the naive

evaluation, all $\binom{n}{k}$ subsets have to be generated. However, since the second version of the naive evaluation (`winnowopt`) uses a more efficient implementation of the preference relation predicate we have a significant drop in the execution times which is roughly one order of magnitude. Despite this decrease, the other two optimizations that focus on the reduction of the number of the generated sets are performing even better. Considering the superpreference optimization, notice that in our case only the movies that have the highest ratings contribute to the best sets. Therefore, at the first step of the superpreference optimization, all movie tuples from the base relation are removed, except for the ones whose ratings belong to the k highest ones. This results in a radical pruning of the set of the generated subsets, which results in a significant decrease of the query execution time. Considering the M-relation optimization, notice that we do not have as much pruning here as in the superpreference optimization. However, the M-tuples contain only the required information for the preference comparison, namely the rating of the movie, and since these tuples are more compact than the original movie tuples, the generation of all subsets is much faster, resulting to better performance.

6.6 Experiments on Predicate Specialization

Among the techniques that we used for optimizing preferential HiLog programs, we proposed Predicate Specialization, a novel program transformation technique for translating higher-order programs into first-order ones. In this series of experiments, we evaluate the performance of this technique in general HiLog programs. The purpose of this set of experiments is twofold; first, to measure the effectiveness of this transformation without using any other optimization techniques; and second, to illustrate that Predicate Specialization can be used for improving the execution runtime not only for preferential higher-order programs but for other types of higher-order logic programs as well.

We have tested our method with a set of benchmarks that include the computation of the transitive closure of a chain of elements, a k-ary disjunction and k-ary conjunction of k relations (for $k = 5, 10$). Regarding the k-ary operators, we used two programs for the same computation. For instance, the first program for k-ary disjunction uses a non recursive computation of the form $c_k = (\ldots((r(1) \cup r(2)) \cup r(3)) \cdots \cup r(k))$ while the second one uses a recursive definition of the form $c_1 = r(1)\ ;\ c_k = r(k) \cup c_{k-1}$. The remaining benchmarks are the winnow, w, wt, path_naive and path_enhanced that were presented in the previous sections. As previously, the higher-order programs are expressed in HiLog and executed using the HiLog module of XSB. We compare their execution with the execution of the Prolog programs produced by Predicate Specialization. Apart from XSB, we also consider the execution of the specialized program in other Prolog engines. The Prolog engines that we use are SWI-Prolog[18], and YAP[19]. Every program is executed several times, each time with a predefined set of facts.

Table 6.6 summarizes the experimental results. In the first column, we show the average execution time of the original, HiLog program and in the following columns, we show the corresponding execution times for the Prolog programs for each engine. The execution times are depicted in seconds. Table 6.6 also illustrates the number of the (non-fact) clauses of the original higher-order program, the number of the (non-fact) clauses of the resulting first-order program after the transformation, and the ranges of the number of the

[18]version 7.2.3, cf. `http://www.swi-prolog.org/`
[19]version 6.2.2, cf. `http://www.dcc.fc.up.pt/~vsc/Yap/`

Table 6.6: Program execution times for Predicate Specialization.

	Program execution time (sec)				Program size		
	HiLog	Prolog			#rules		
Program	xsb	xsb	swi	yap	h.o.	f.o.	#facts
closure	1744.829	17.426	15.813	8.782	3	3	1000-8000
closure_1000	12.132	0.801	0.609	0.372	3	3	1000
closure_2000	91.284	2.884	2.644	1.332	3	3	2000
closure_4000	709.356	11.336	10.918	5.464	3	3	4000
closure_6000	2365.728	25.536	23.459	13.532	3	3	6000
closure_8000	5545.644	46.576	41.433	23.208	3	3	8000
conj5	9.887	1.090	0.026	0.010	3	6	1000-8000
genconj(5)	9.921	1.101	0.028	0.011	4	4	1000-8000
conj10	21.676	2.414	0.023	0.015	3	11	1000-8000
genconj(10)	21.580	2.415	0.039	0.013	4	4	1000-8000
union5	0.035	0.028	0.030	0.023	4	10	1000-8000
genunion(5)	0.034	0.030	0.025	0.021	5	5	1000-8000
union10	0.063	0.062	0.046	0.036	4	20	1000-8000
genunion(10)	0.062	0.079	0.054	0.035	5	5	1000-8000
path_enhanced	971.326	679.557	975.027	54.156	6	6	10-80
path_naive	5.725	4.248	6.661	0.407	6	6	10-80
winnow	0.147	0.130	0.117	0.039	3	3	1000-10000
w(2)	3.920	3.257	3.844	0.527	10	12	100-2000
w(3)	129.457	107.183	122.556	21.103	10	12	100-2000
wt(2)	4.146	3.288	3.857	0.530	11	13	100-2000
wt(3)	130.540	108.048	126.876	21.360	11	13	100-2000

corresponding facts. We do not show the runtime of each transformation from the higher-order to first-order since the execution of the transformation process was negligible (e.g. less than 0.01 seconds in all cases).

Firstly, we observe that the first-order programs are in general much faster than the higher-order ones. Even in the context of XSB which offers native support of HiLog, the Prolog code is in almost all cases faster than the HiLog code. Especially in the transitive closure and the k-ary conjunction, we have an improvement by one or more of orders of magnitude. In most programs in our experiment, we noticed that the ratio between the execution time of Prolog code and the execution time of HiLog code does not change much if we increase the number of facts, with the exception of the transitive closure benchmark, in which the more we increase the number of facts, the more performant is the equivalent Prolog program. Secondly, one of the important advantages of executing standard Prolog code is that it allows us to choose from a wide range of available Prolog engines. Notice that from the three Prolog engines that we used, YAP is the most performant one. Therefore, as a second observation, we notice that we can get a further decrease in execution times by simply choosing a different Prolog engine, a fact that is not possible if we want to execute HiLog code directly.

Finally, regarding the size of the transformed first-order transformations, consider the programs that deal with the k-ary conjunction and disjunction, i.e. the pairs conj5 – genconj(5), conj10 – genconj(10), union5 – genunion(5), and union10 – genunion(10). Both programs of each of these pairs are making the same computation, with the former expressed in a non-recursive way and the latter in a recursive way. These programs differ also in the size of their first order counterparts. The first-order form of the non-recursive version has more clauses than the first-order form of the recursive version. We observe that both the higher-order and the first-order versions of the same computation have similar execution times, even though the first-order versions have different numbers of clauses. As a result, this increase in the size of the resulting first-order output did not produce any overhead in the overall program execution time.

7. RELATED WORK

In this chapter, we discuss related work regarding preference representation formalisms in the area of databases and logic programming. We discuss both quantitative and qualitative approaches and we compare them with our work; i.e., the infinite-valued logic programming language PrefLog defined in Chapter 4 and the preferential higher-order logic programming framework of Chapter 4. Moreover, we discuss related work regarding Predicate Specialization, which is a proposed technique that we have used for optimizing tuple-preference HiLog programs.

7.1 Overview

Preferences play an important role in knowledge representation and have many applications in diverse domains. In this chapter, we discuss related work regarding preference representation formalisms that have been proposed in the areas of databases and logic programming. As a general comment, we could say that our approaches are not very directly related to most of the existing ones, since, to the best of our knowledge, it is the first time that the uses of an infinite-valued or a higher-order language are proposed as logic programming frameworks for expressing preferences. However, in some cases there exist some underlying connections between our approaches and certain of the already proposed techniques, which we highlight in the rest of the chapter.

As already mentioned, formalisms for representing preferences can be divided [69] into the *qualitative* and the *quantitative* ones. It has been argued that the quantitative approaches are less general than the qualitative ones: there exists a preference relation that can not be expressed using a scoring function (see Example 1.2 in [17][page 428] and the discussion in Subsection 7.1.2 of the same paper). This happens in our case as well; as we have discussed in Subsection 4.4.3 there exist (at least) one type of preference relation that can be expressed using our higher-order framework but not with PrefLog. This is an expected phenomenon because PrefLog is a quantitative programming language and the higher-order framework is designed to express qualitative preferences. In order to draw a fair comparison between works in the literature and our approaches, in the following sections, we will compare the quantitative approaches with PrefLog and the qualitative approaches with our higher-order framework.

The process of executing and optimizing preferential programs opened up for us the possibility of devising specialized transformations and other optimizations for enhancing their performance. Promising such techniques are given in Chapter 3 and Chapter 5 and their efficiency is verified in Chapter 6. Among all the techniques discussed, the most interesting one is Predicate Specialization, which was designed for optimizing tuple-preference higher-order logic programs. This technique is based on partial evaluation [40] and is related to several approaches in the program transformation literature, especially in techniques that transform higher-order programs into first-order ones. We include a discussion on how these approaches compare with Predicate Specialization towards the end of this chapter.

The rest of the chapter is organized as follows: in Section 7.2, we start with a discussion about quantitative and qualitative systems in the area of databases; in Section 7.3 we continue with a discussion about approaches that are used for expressing qualitative preferences in logic programming; in Section 7.4 we continue with a discussion about quantitative extensions of logic programming; in Section 7.5 we continue with a discus-

sion about related work regarding Predicate Specialization; and finally, we close with a brief summary of the chapter.

7.2 Preferences in Databases

Each one of our logic programming approaches is based on a relevant approach in the database domain. In particular, our infinite-valued approach is based on the query system of Agarwal and Wadge [1, 2], and our higher-order approach is based on the framework of Chomicki [17, 80]. In this section, we present some related work in the field of relational databases. We start with a discussion about quantitative approaches (which are mostly related to our infinite-valued approach) and then we continue with a discussion about qualitative approaches (which are mostly related to our higher-order approach). For a detailed comparison between quantitative and qualitative formalisms in database systems, refer to a detailed survey by Stefanidis et al [69].

7.2.1 Quantitative Preferences in Databases

In the quantitative approach, preferences are expressed by assigning numerical values on tuples, such that one tuple is preferred over another if its preference score is higher. Among the many approaches in the literature, two interesting quantitative approaches are developed by Agrawal and Whimmers [3] and by Koutrika and Ioannidis [45]. In the first approach [3], the preference score of a tuple can be obtained by a function which takes a tuple and returns a numerical value (e.g., the preference function $f(t) = 0.1 \cdot t$.rating calculates a preference score for every movie according to its rating). In the second approach [45], the score is obtained by specifying appropriate selection conditions (e.g., the preference $(\text{movie.genre} = \text{'drama'}, 0.9)$ expresses a preference score of 0.9 on drama movies). Both approaches offer preference combination mechanisms, in which two (or more) preference scores are combined, by applying a combining function such as weighted summation, maximum, minimum, average and so on.

We believe that the above quantitative systems [3, 45] are actually quite sophisticated and they may prove to have interesting applications. However, when it comes to expressing preferences at a high declarative level, we feel that expressing preferences directly using preference scores has a disadvantage. As remarked by Domshlak et al. [24], "*humans are rarely willing to express their preferences directly in terms of a value function. [...] Instead of rating complete alternatives immediately, it is normally much easier and arguably more natural to provide information about preferences in separate pieces, preferably in a qualitative way*". In contrast, the preference values in PrefLog are not explicit but are created using appropriate preference operators. This means that even though PrefLog is a quantitative preference framework, it allows the user to express preferences in a qualitative way and pushes the quantitative burden to the underlying semantics.

7.2.2 Qualitative Preferences in Databases

One of the earliest works in qualitative preference queries in databases is the approach of Lacroix and Lavency [46]. The authors propose an extension of relational calculus in which preferences for tuples satisfying given logical conditions can be expressed. For example, one could say: pick the tuples of R satisfying $Q \wedge P_1 \wedge P_2$; if the result is empty, pick the tuples of R satisfying $Q \wedge P_1 \wedge \neg P_2$; if the result is again empty, pick the tuples of R satisfying $Q \wedge \neg P_1 \wedge P_2$. As discussed by Chomicki [17], this approach can be simulated

by the techniques in his framework [17], and therefore also by our higher-order logic programming framework.

The framework of Chomicki [17, 80] is one of the most simple and expressive ones in the database literature regarding preference representation. Since (to our knowledge) there does not exist any widely available implementation of this framework, it is not possible to directly compare this approach with our own in terms of efficiency (even though, we presume that an implementation of this framework over a SQL system will probably be more efficient and can have a broader impact and real-world applications than our logic programming approach). A more detailed comparison between the framework of Chomicki [17, 80] and our higher-order logic programming framework has been given throughout Chapter 4.

Another influential work in the area of qualitative preferences in the database domain is that of Kießling [42]. Contrary to the logical approach advocated by Chomicki[17] Kießling takes an algebraic approach by using a language that offers preference constructors. Two basic preference constructors are, for example, the POS and NEG ones, which are used to provide preferred and non-preferred values respectively. For instance, a $\mathsf{POS}(\mathsf{genre}, \{\mathsf{comedy}\})$ preference states that a comedy movie is preferred, while a $\mathsf{NEG}(\mathsf{genre}, \{\mathsf{drama}, \mathsf{scifi}\})$ preference states that a movie is not preferred if it is either a drama or a science fiction one. These preference constructors can be further combined in order to express more complex preferences. Several preference combinators are supported, such as Pareto, lexicographic, and so on. In addition, two versions of this approach are provided, namely *Preference XPATH* [43] and *Preference SQL* [44]. Kießling's framework has some common characteristics with that of Chomicki [17] (see [17][Section 10.1] for a detailed comparison) and many arguments we have used throughout Chapter 4 in order to compare our higher-order approach with that of Chomicki [17], can also be used in order to compare our framework against the work of Kießling [42]. For example, the framework of Kießling [42] does not allow having arbitrary constraints in preference formulas and it allows only restricted use of extrinsic preference relations and of the transitive closure operation.

7.3 Qualitative Preferences in Logic Programming

A number of different approaches have been proposed in the logic programming domain with the purpose of supporting qualitative preferences. These approaches can be categorized into two main streams:

- those that use preferences in order to select the best solutions to a given problem that has been expressed as a logic program.
- those that use preferences in order to resolve conflicts that appear in non-monotonic extensions of logic programming (such as multiple minimal models).

To the best of our knowledge, there does not exist any works for expressing quantitative preferences in the logic programming domain.

7.3.1 Preferences over Program Solutions

Closer to our qualitative higher-order approach are the approaches that fall in the first category [19, 32, 33, 34, 36, 38]. In order to support preferences, it is common to use

syntactically-extended logic programs. For example, Govindarajan et al. [32, 33, 34, 38] study an extension of classical logic programs, called *Preference Logic Programming* (in short PLP). These programs consist of two basic parts; the first part is a set of first-order definite logic clauses while the second part is a set of optimization definitions expressed as constraints on the atoms of the first part. The proposed semantics selects a preference model of the program among the possible models, such that it optimizes the second part of the program. The formalism of PLP programs does not support negation-as-failure.

Cui and Swift [19] in a later extension of PLP, identify this lack of negation and propose a different technique in order to support logic programs with negation-as-failure. Their proposed approach is to transform an extended PLP program into an equivalent logic program with negation. They suggest that the well-founded model of the transformed program is the intended model of the program with preferences. Moreover, if this program is a standard PLP program, they prove that this well-founded model coincides with that of Govidarajan et al. [32]. The program transformation encodes a behavior that is similar to that of the winnow operator. Guo and Jayaraman [36] propose a logic programming language, which is also closely related to PLP [32]. The proposed semantics uses two meta-operators over the minimum model of the non-optimization part of the program in order to remove the atoms that are not the most desired ones with respect to the preference definition. These operators, again, resemble the winnow operator that we are using, but in this case the winnow operator is not hard-coded in the transformed program (as in the case of Cui and Swift [19]) but in the evaluation of the preference model. PLP and its extensions [19, 32, 33, 34, 36, 38] have been demonstrated to have many applications in resolving ambiguity in programming languages and natural language grammars, in scheduling and optimization, as-well-as in database querying.

In the approaches in this category [19, 32, 36] the preference relations and the winnow operator cannot appear as building blocks of queries (not can one define alternative operators beyond winnow). The main difference between this line of research and our own, is that the former uses a specialized formalism that requires the development of novel model-theoretic techniques in order to express its semantics and also of specialized techniques in order to implement it. On the other hand, our approach does not use any specialized technique outside the realm of higher-order logic programming, and can be implemented using standard higher-order logic programming languages. However, as an overall comment, the work reported in these papers [19, 32, 33, 34, 36, 38] is based on quite interesting concepts, and the idea of the “optimal subproblem property” discussed by Govindarajan et al. [33] has motivated our optimized path program given in Subsection 4.4.5.

7.3.2 Preferences over Program Models

The second category of qualitative approaches in logic programming is applicable to logic programming languages that are extended with non-monotonic features; such features are disjunctions in the head, default and explicit negation, and so on. In short, non-monotonicity plays a vital role in the approaches of this category. Many research works fall into this category [6, 7, 8, 21, 23, 66, 78, 81] (see also the excellent literature review given by Sakama and Inoue [66]). The key idea behind all these approaches can be described as follows. A logic program that is extended with non-monotonic characteristics usually has many minimal models. In order to choose the most appropriate models, we add to the program preference information. Usually, this preference information is either

given as an ordering of atoms, literals, or even rules of the program. We will concentrate on the work of Sakama and Inoue [66] (and similar arguments can be given for the rest of the cited works). Here, a priority relation, which is reflexive and transitive, is given over the set of literals of the program. Then, the *preferred answer-set* semantics of the program is defined, an approach that generalizes the classical stable model semantics of Gelfond and Lifschitz [31], and aims to select a subset of preferred minimal models.

There are many important differences between our higher-order framework and the works in this category [6, 8, 66, 81]. First, our work can be applied both to positive programs as-well-as to programs that use negation. On the other hand, the work of Sakama and Inoue [66] (as-well-as other works in this category) is only meaningful when non-monotonicity is present in the program (if a program is positive then it always has the same model independently of the preference relation given [66]). Therefore it is not obvious how the technique of Sakama and Inoue [66] (and all other related approaches) can be applied to express preferences over simple database relations. The task of defining preferences over sets of tuples appears to be even more difficult. Another important difference between our higher-order logic programming approach and many other existing approaches in the logic programming domain is the fact that it is not very hard to implement and it can be used to directly run some interesting and non-trivial applications.

7.4 Quantitative Extensions of Logic Programming

As we mentioned in the previous section, there does not exist any works for expressing quantitative preferences in the logic programming domain (to the best of our knowledge). However, the quantitative language PrefLog is related to various quantitative extensions of logic programming. In this section, we compare PrefLog with other infinite-valued approaches in logic programming and with probabilistic extensions of logic programming.

7.4.1 Infinite-Valued Logic Programming

As we mentioned in Subsection 2.1, the programming language PrefLog is based on infinite-valued logic, which was previously used in order to provide a purely model-theoretic semantics for logic programming with negation-as-failure [64]. Consider the following program that uses the negation-as-failure operator $\{\mathtt{p},\ \mathtt{q} \leftarrow\sim \mathtt{r}\}$. Under the well-founded semantics [30], both $\mathtt{p}$ and $\mathtt{q}$ receive the value true. However, Rondogiannis and Wadge in their article [64] argue that in some sense $\mathtt{p}$ is “truer” than $\mathtt{q}$; $\mathtt{p}$ is true because there is a rule which says so, whereas $\mathtt{q}$ is true only because there is no evidence that $\mathtt{r}$ is true; the same happens for false values. Therefore, their approach is to understand negation-as-failure as combining ordinary negation with a weakening (which corresponds to the operator ϵ). Since negations can be iterated, an infinite set of truth values is produced, similar to the set $\mathbb{V}$ of our approach. A similar approach is followed to provide a purely model-theoretic semantics for disjunctive logic programming with negation-as-failure [9].

From a syntactic point of view, logic programs with negation-as-failure can be viewed as PrefLog programs with the following set of operators $\{\wedge, \vee, \sim\}$. Contrary to PrefLog operators though, the $\sim$ operator is not monotonic, therefore cannot be used by PrefLog. Due to the fact that $\sim$ is not monotonic, the minimum infinite-valued model for logic programs with negation-as-failure [64] cannot be computed simply by iterating the $T_{\mathtt{P}}$ operator until a fixed point is reached (as this holds in our approach) but this involves a more complex

computation that evaluates the minimum model of the program in stages[20]. The continuity property of T_{P} for PrefLog programs guarantees that the fixed point is reachable through this process in at most ω steps, a property that does not hold for logic programs with negation-as-failure. The final difference from our approach is the size of the underlying set of truth values. Logic programs with negation-as-failure [64] require truth values T_α and F_α for every countable ordinal α, while PrefLog programs require truth values T_n and F_n for every $n \in \mathbb{N}$.

7.4.2 Probabilistic Logic Programming

Quantitative approaches in logic programming can be found in certain many-valued or probabilistic extensions of traditional logic programming [20, 54, 58, 76]. In probabilistic extensions, the programmer is usually required to rank rules (or facts) with a certainty factor or to use a special form of implication in rules that have attached a numerical attenuation factor. These extensions of logic programming are usually not designed to express preferences but have a flavor of preferential logic programming, since, for example, facts that have different numerical factors can be considered to be of different preferences. As in the quantitative approaches in the database domain, the probabilistic approaches express the quantitative (propabilistic) scores directly with numerical values; as we stated previously, expressing preferences directly using numerical values is not as intuitive and desirable in most preference situations from a user's point of view. In contrast, preferences in PrefLog are expressed quantitatively with the use of preference operators, and the quantitative preference score is inferred. Moreover, the operators that are offered in these probabilistic extensions [20, 54, 58, 76] are perfectly suited to the probabilistic domain but are not always suitable for expressing preferences. For instance, the conjunction operator between atoms in ProbLog [58] multiplies the probability values of these atoms, which is an expected behavior for probabilistic scenarios (since the probability of a conjunction of events is equal to the product of the probabilities of these events). However, for preference applications, it is not always the case that the intended preference score of a conjunction of atoms must be equal to the product of the individual preference scores. In PrefLog, the standard behavior for the conjunction operator is the minimum operator. However, PrefLog gives the freedom to define new preference operators, depending on the application at hand.

7.5 Related work on Predicate Specialization

In this section, we present some related work on Predicate Specialization, which is used for optimizing preferential higher-order logic programs. This technique is proposed in Section 5.3. In short, given a higher-order logic program and query that does not contain any free predicate variables, it transforms the program into a first-order one, specialized for this specific query.

7.5.1 Partial Evaluation

Predicate Specialization is closely connected with related work on partial evaluation of logic programs [51, 29, 47]. More specifically, the proposed technique is a special form of partial evaluation which targets higher-order arguments and uses a simple one-step

[20] However, we have used a similar procedure to provide a terminating bottom-up strategy for function-free PrefLog programs (c.f. Section 3.3).

unfolding rule to propagate the constant higher-order arguments without changing the structure of the original program. Consequently, first-order programs remain unchanged. To the best of our knowledge, partial evaluation techniques have not been previously applied directly to higher-order logic programming with the purpose to produce a simpler first-order program.

7.5.2 Defunctionalization and its Extensions

Other techniques, however, have been proposed that focus on the removal of higher-order parameters in logic programs. D. H. D. Warren, in one of the early papers that tackle similar issues [79], proposed that simple higher-order structures are non-essential and can be easily encoded as first-order logic programs. The key idea is that every higher-order argument in the program can be encoded as a symbol utilizing its name and a special apply predicate should be introduced to distinguish between different higher-order calls. A very similar approach has been employed in HiLog [15]; a language that offers a *higher-order syntax* with first-order semantics. A HiLog program is transformed into an equivalent first-order one using a transformation similar to Warren's technique [79]. Actually, these techniques are closely related to Reynolds' *defunctionalization* [60] that has been originally proposed to remove higher-order arguments in functional programs. These techniques are designed to be applied to arbitrary programs in comparison to our approach. In order to achieve this, they require data structures in the resulting program. However, on a theoretical view, this imposes the requirement that the target language should support data structures even if the source language does not support that. This is apparent when considering Datalog; transforming a higher-order Datalog program will result in a first-order Prolog program. On a more practical point, the generic data structures introduced during the defunctionalization render the efficient implementation of these programs challenging. The wrapping of the higher-order calls with the generic apply predicate makes it cumbersome to utilize the optimizations in first-order programs such as indexing and tabling. In comparison, our technique produces more natural programs that do not suffer from this phenomenon. Moreover, it does not introduce any data structures and as a result, a higher-order Datalog program will be transformed into a first-order one amenable to more efficient implementation.

In order to remedy the shortcomings of defunctionalization, there have been proposed some techniques to improve the performance of the transformed programs. Sagonas and Warren [65] proposed a compile-time optimization of the classical HiLog encoding that eliminates some partial applications using a family of apply predicates thus increasing the number of predicates in the encoded program, which leads to more efficient execution. The original first-order encoding of HiLog, as well as this optimization are included in the XSB system [70]. In the context of functional-logic programming, there exist some mixed approaches that consider defunctionalization together with partial evaluation for functional-logic programs [4, 59], where a partial evaluation process is applied in a defunctionalized functional-logic program. Even though these approaches can usually offer a substantial performance improvement, the resulting programs still use a Reynolds-style encoding; for instance, the performance gain of the optimizations offered by XSB is not sufficient when compared to Predicate Specialization, as presented in Section 6.6.

7.5.3 Other Higher-order Removal Methods

The process of eliminating higher-order functions is being studied extensively in the functional programming domain. Apart from defunctionalization, there exist some approaches

that do not introduce additional data structures while removing higher-order functions. These techniques include the *higher-order removal* method [16], the *firstification* technique [55] and the *firstify* algorithm [53]. The removal of higher-order values here is achieved without introducing additional data structures, so the practical outcome is that the resulting programs can be executed in a more efficient way than the original ones. The basic operation of these transformation methods is *function specialization*, which involves generating a new function in which the function-type arguments of the original definition are eliminated.

A predicate specialization operation is also the core operation in our approach, so at this point, these approaches are similar to ours. The remaining operations that can be found in those approaches (e.g. simplification rules, inlining, eta-abstractions etc.), are either inapplicable to our domain or not needed for our program transformation. Contrary to Reynolds' defunctionalization, these higher-order removal techniques [16, 53, 55] are not complete, meaning that they do not remove all higher-order values from a functional program, and therefore the resulting programs are not always first order. This phenomenon would happen in our case as well if we considered the full power of higher-order programming. However, because of the fact that we focus on a smaller but still useful class of higher-order logic programs, we are sure that the output of our transformation technique will produce a valid first-order program for every program that belongs to our fragment.

7.6 Summary

In this chapter, we compared our approaches with several works in the literature. To the best of our knowledge, our approaches are not very directly related to most of the existing ones, since it is the first time that the use of an infinite-valued or a higher-order language are used for expressing preferences in the logic programming domain. The key advantages of our approaches are the following: Regarding PrefLog, the main advantage is that even though PrefLog is a quantitative extension of logic programming, the preference scores are not denoted explicitly with numerical values but are expressed indirectly using preference operators, making the extension more intuitive from a user point of view. Regarding higher-order logic programming, the main advantages are the following; firstly, the fact that relations, preference relations, and operations on preferences are encoded in the same language (an important benefit that we have heavily discussed in Chapter 4); and secondly, the fact that the expressive power of higher-order logic programming allows us to emulate all qualitative preference frameworks that we have discussed in this chapter. Moreover, the use of higher-order logic programming opened up for us the possibility of designing specialized techniques for optimizing higher-order logic programs, such as Predicate Specialization.

8. CONCLUSIONS AND FUTURE WORK

In this chapter, we conclude the dissertation, beginning with a summary of our contributions and continuing with a discussion of possible future research directions.

8.1 Conclusions

The dissertation contributes to the area of preference representation and our results can be perceived as logical frameworks for expressing and manipulating preferences. More specifically, we propose two approaches for expressing preference using extensions of logic programming:

- The first approach uses infinite-valued logic programming for expressing quantitative preferences. This language is based on an infinite set of truth values in order to support operators for expressing preferences.
- The second approach uses higher-order logic programming for expressing qualitative preferences. In this approach, preference relations and operations on preferences are expressed in the same, higher-order language.

Our approaches attempt to overcome some shortcomings of existing approaches in the domain of representation of quantitative and qualitative preferences.

We proposed the logic programming language PrefLog (cf. Chapter 2), which uses an infinite-value domain for expressing quantitative preferences (cf. Section 2.2). The preference value is not defined directly but is expressed using preference operators. If all preference operators used are monotonic and continuous over the set of truth values $\mathbb{V}$, then every such PrefLog program has a minimum infinite-valued model (cf. Section 2.3). We defined some basic preference operators (such as opt and alt), we described a simple approach for building new operators and we demonstrated their use in example programs (cf. Section 2.4). Moreover, we proposed a method for evaluating PrefLog programs (cf. Chapter 3), which is a terminating bottom-up evaluation (cf. Section 3.3) for a well defined function-free fragment of PrefLog, namely $\{\epsilon, \wedge\}$-programs (cf. Section 3.2). Ensuring termination in this case is not a straightforward task because the underlying truth domain and the set of all possible interpretations of a function-free PrefLog program are both infinite.

We proposed the use of higher-order logic programming as a framework for representing and manipulating qualitative preferences (cf. Chapter 4). In this approach, base and preference relations, operations on preference relations (cf. Section 4.4) and preferences over sets (cf. Section 4.5) are expressed in the same, higher-order language (cf. Section 4.3). Moreover, we developed specialized optimizations (cf. Chapter 5) for enhancing the performance of this framework, since a basic, unoptimized implementation is almost straightforward (cf. Section 5.2). We proposed Predicate Specialization (cf. Section 5.3), a technique that reduces the program execution time of higher-order logic programs by transforming them into first-order ones. This optimization targets higher-order logic programs that express preferences over tuples (cf. Section 5.4). Finally, for optimizing programs that express preferences over sets, we used two pruning techniques, namely superpreference and M-relation. These optimizations reduce the program execution time by reducing the number of the candidate sets (cf. Section 5.5).

We undertook an implementation of our higher-order approach for the XSB system (cf. Section 5.2, Subsection 5.3.5, and Subsection 5.5.4). Moreover, we provided experimental results that suggest the feasibility of our approach and the effectiveness of the proposed optimization techniques (cf. Chapter 6). The source code of the implementations and the experiments is publicly available[21].

Finally, regarding the comparison between the two proposed approaches in terms of expressivity, there exists a type of preferences (i.e., lexicographic preferences) that can be expressed using our higher-order logic programming framework (cf. Subsection 4.4.3) but not with PrefLog (cf. Section 2.5). This comes as no surprise, because PrefLog is a quantitative programming language and our higher-order framework is designed to express qualitative preferences and it is argued that the quantitative approaches are more general than the quantitative ones [17][page 428].

8.2 Future Work

In this last section, we propose some potential directions for future research. We discuss some interesting aspects in which the infinite-valued approach and the higher-order approach can both be extended in terms of expressivity and be enhanced in terms of usability and performance.

8.2.1 Future Work on the Infinite-Valued approach

In this subsection, we reduce our focus on possible extensions of PrefLog. We discuss some aspects for future investigation in terms of extending the representation capabilities of our infinite-valued approach.

Possibly the most interesting such topic is the addition of negation-as-failure to PrefLog. It has been demonstrated [64] that the meaning of negation can also be captured using the infinite-valued domain $\mathbb{V}$ that we adopted for defining the semantics of PrefLog. It would be interesting to investigate how the preference operators of PrefLog could coexist with negation in a unified framework. The following example motivates such an investigation.

Example 8.1. *Continuing Example 2.1, assume that we would now like to fly from Athens to Boston, but (if possible) avoiding the "Delay Air" carrier. This suggests the use of some form of negation:*

$$\begin{array}{lll} \texttt{desired\_flight(F)} & \leftarrow & \texttt{from\_to(athens, boston, F)} \wedge \\ & & \text{opt} \sim \texttt{carrier(F, delay\_air)}. \end{array}$$

In the above program, a flight from Athens to Boston with a carrier that is different than "Delay Air" is expected to return a true result, a flight that does not go from Athens to Boston will return a false result and a flight from Athens to Boston with "Delay Air" will return some false truth value (different than absolute falsity).

It is unclear at the moment whether negation should operate on the same "dimension" of truth values as the other preferential operators (like opt and alt) or whether a separate dimension should be used.

[21] cf. http://bitbucket.org/antru/holppref
http://bitbucket.org/antru/preflog
http://bitbucket.org/antru/firstify

The PrefLog language depends on a set of preference operators that must be monotonic and continuous over the infinite-valued domain $\mathbb{V}$. In Section 2.4 we described a simple approach for building new operators; we used a set of three continuous operators (i.e., $\wedge$, $\vee$ and ϵ), and we built new operators by combining them. The resulting operators are continuous since the composition of continuous operators is also continuous. However, we showed that not all continuous operators over $\mathbb{V}$ can be defined using $\wedge$, $\vee$ and ϵ. An interesting aspect for future investigation would be to devise a metalanguage in which to define new preference operators that are guaranteed to be monotonic and continuous. One possible candidate for such a language would probably be a fragment of the language Hopes [10], which allows the definition of predicates of various types (and even predicates over boolean domains) that are guaranteed to be monotonic and continuous.

In Section 2.5, we showed that lexicographic preferences cannot be expressed using PrefLog due to the form of the infinite set of truth values $\mathbb{V}$. One interesting research direction would be the extension of the underlying set of truth values such that PrefLog to be able to support lexicographic preferences. Following similar preferential extensions of the set of real numbers $\mathbb{R}$ [18, 35, 41], Papadimitriou [56] proposed *lxpQL*, a query language which extends the query language of Agarwal and Wadge [1, 2] with a lexicographic preference operator. Obviously, the underlying set of truth values in lxpQL is not the set $\mathbb{V}$, but it consists of *finite sequences of elements from* $\mathbb{V}$. Two such sequences are compared in a lexicographic way, e.g., the sequence (T_2, T_1, F_0) is preferred to (T_2, T_2, T_0). At this moment, it is an open question whether the extended domain and the operators of lxpQL can be used in a logic programming setting.

8.2.2 Future Work on the Higher-Order approach

In this subsection, we reduce our focus on possible future directions regarding evaluation techniques and real-world applications of the qualitative preference higher-order logic programming framework.

Chapter 4 suggests that a fragment of higher-order logic programming can be used as a purely logical framework for expressing preferences. This fragment is essentially a higher-order extension of Datalog that supports negation and tuples in a restricted way. We have used natural numbers only in order to define operators over preference relations (cf. Subsection 4.4.4) and aggregate operators in the case of extrinsic (cf. Examples 4.9 and 4.10) and set-based preference relations (cf. Example 4.16). It seems that this framework, apart from its preference representation capabilities, can benefit from more effective evaluation techniques. A sign that would support this claim is, for instance, the situation on first-order logic programming, in which bottom-up techniques are more efficient than top-down ones when it comes to first-order Datalog programs. As a result, we believe that it would be very interesting to study the properties of this higher-order Datalog with tuples and negation, a language that generalizes classical first-order Datalog. For example, it would be interesting to investigate bottom-up proof procedures or other optimizations (such as a higher-order extension of magic-sets [5]).

For the implementation of our higher-order preferential framework, we have used the XSB system which provides a mature and stable implementation of HiLog. Despite the fact that we faced no problems in implementing preferences over tuples, the implementation of queries over set preferences was not straightforward. The reason for the difficulties we faced was the way that HiLog treats uninstantiated predicate variables: it searches the program to find appropriate predicates that can be substituted; if no predicates are

found, the query fails. This state of affairs is due to a well-known distinction between extensional and intensional semantics for higher-order logic programming [10]. We believe that an extensional higher-order logic programming language would offer advantages in the implementation of set preferences because the interpreter of the language would automatically produce all the possible sets that should be substituted for the uninstantiated predicate variables. One such language is the extensional higher-order language Hopes [10]. Set preferences can be modeled in Hopes as follows:

Example 8.2. *Recall Example 4.16, and the preference relation* `rating_pref` *in which a set of movies is more preferred to another if the sum of the ratings of its movies is greater. Moreover, consider the following clauses:*

```
subsetof(R,N)(S) :- subset(S,R), size(S,N).
subset(S,R) :- not nonsubset(S,R).
nonsubset(S,R) :- S(X), not R(X).
```

Then, the most preferred 3-element sets of movies can be found using the following query:

```
?- winnow(rating_pref, subsetof(movie,3))(S).
```

The definitions of the `winnow`, `bypassed` *and* `size` *can be found in Chapter 4.*

As we mentioned earlier though, implementations of extensional higher-order languages (and Hopes among them) have not yet reached the same level of maturity as that of intensional ones, especially when negation is considered. As a result, an important future direction is to work on producing a stable and efficient version of Hopes that will support negation [11], a concept that is very useful in the processing of set preferences.

In this last paragraph, we discuss future work regarding the proposed technique of Predicate Specialization. In Subsection 5.4, we saw that Predicate Specialization can be effective for a class of higher-order logic programs that make limited use of partial applications, in the case of predicates that do not belong in the same cycle in the predicate dependency graph. From a syntactic point of view, this class is a broader class than definitional programs; however, it seems that both fragments have the same expressive power. An interesting open question that arises is whether Predicate Specialization can be used as a first-order reduction method only for the fragment of definitional programs (or a fragment that has the same expressive power) or if it can be used for a wider and more expressive class of higher-order programs. Until now, we have used and evaluated Predicate Specialization only as an optimization method for performance improvement. However, in the functional programming domain, such techniques have been used in additional applications, such as program analysis [53] and implementation of debuggers [57]. Therefore, an interesting aspect for future investigation would be the search of similar or completely new applications of Predicate Specialization in the logic programming domain.

REFERENCES

[1] Ruchi Agarwal. A framework for expressing prioritized constraints using infinitesimal logic. Master's thesis, University of Victoria, Canada, 2005.

[2] Ruchi Agarwal and William W. Wadge. The lazy evaluation of infinitesimal logic expressions. In Hamid R. Arabnia, editor, *Proceedings of The 2005 International Conference on Programming Languages and Compilers, PLC 2005, Las Vegas, Nevada, USA, June 27-30, 2005*, pages 3–7. CSREA Press, 2005.

[3] Rakesh Agrawal and Edward L. Wimmers. A framework for expressing and combining preferences. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 297–306. ACM, 2000.

[4] Elvira Albert, Michael Hanus, and Germán Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002, 2002.

[5] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In Avi Silberschatz, editor, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 1–15. ACM, 1986.

[6] Gerhard Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *J. Artif. Intell. Res.*, 4:19–36, 1996.

[7] Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1467–1474. AAAI Press, 2015.

[8] Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artif. Intell.*, 109(1-2):297–356, 1999.

[9] Pedro Cabalar, David Pearce, Panos Rondogiannis, and William W. Wadge. A purely model-theoretic semantics for disjunctive logic programs with negation. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2007.

[10] Angelos Charalambidis, Konstantinos Handjopoulos, Panagiotis Rondogiannis, and William W. Wadge. Extensional higher-order logic programming. *ACM Trans. Comput. Log.*, 14(3):21:1–21:40, 2013.

[11] Angelos Charalambidis and Panos Rondogiannis. Constructive negation in extensional higher-order logic programming. In Chitta Baral, Giuseppe De Giacomo, and Thomas Eiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 2014, Vienna, Austria, July 20-24, 2014*. AAAI Press, 2014.

[12] Angelos Charalambidis, Panos Rondogiannis, and Ioanna Symeonidou. Approximation fixpoint theory and the well-founded semantics of higher-order logic programs. *TPLP*, 18(3-4):421–437, 2018.

[13] Angelos Charalambidis, Panos Rondogiannis, and Antonis Troumpoukis. Higher-order logic programming: an expressive language for representing qualitative preferences. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 24–37. ACM, 2016.

[14] Angelos Charalambidis, Panos Rondogiannis, and Antonis Troumpoukis. Higher-order logic programming: An expressive language for representing qualitative preferences. *Sci. Comput. Program.*, 155:173–197, 2018.

[15] Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A foundation for higher-order logic programming. *J. Log. Program.*, 15(3):187–230, 1993.

[16] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Lisp and Symbolic Computation*, 9(4):287–322, 1996.

[17] Jan Chomicki. Preference formulas in relational queries. *ACM Trans. Database Syst.*, 28(4):427–466, 2003.

[18] Maria Chowdhury, Alex Thomo, and William W. Wadge. Preferential infinitesimals for information retrieval. In Lazaros S. Iliadis, Ilias Maglogiannis, Grigorios Tsoumakas, Ioannis P. Vlahavas, and Max Bramer, editors, *Artificial Intelligence Applications and Innovations III, Proceedings of the 5TH IFIP Conference on Artificial Intelligence Applications and Innovations (AIAI'2009), April 23-25, 2009,*

Thessaloniki, Greece, volume 296 of *IFIP Advances in Information and Communication Technology*, pages 113–125. Springer, 2009.

[19] Baoqiu Cui and Terrance Swift. Preference logic grammars: Fixed point semantics and application to data standardization. *Artif. Intell.*, 138(1-2):117–147, 2002.

[20] Evgeny Dantsin. Probabilistic logic programs and their semantics. In Andrei Voronkov, editor, *Logic Programming, First Russian Conference on Logic Programming, Irkutsk, Russia, September 14-18, 1990 - Second Russian Conference on Logic Programming, St. Petersburg, Russia, September 11-16, 1991, Proceedings*, volume 592 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 1991.

[21] James P. Delgrande, Torsten Schaub, and Hans Tompits. Logic programs with compiled preferences. In Werner Horn, editor, *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany, August 20-25, 2000*, pages 464–468. IOS Press, 2000.

[22] James P. Delgrande, Torsten Schaub, Hans Tompits, and Kewen Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.

[23] Pierangelo Dell'Acqua and Luís Moniz Pereira. Preferential theory revision. *J. Applied Logic*, 5(4):586–601, 2007.

[24] Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in AI: an overview. *Artif. Intell.*, 175(7-8):1037–1052, 2011.

[25] Zoltán Ésik and Panos Rondogiannis. Theorems on pre-fixed points of non-monotonic functions with applications in logic programming and formal grammars. In Ulrich Kohlenbach, Pablo Barceló, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation - 21st International Workshop, WoLLIC 2014, Valparaíso, Chile, September 1-4, 2014. Proceedings*, volume 8652 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2014.

[26] Zoltán Ésik and Panos Rondogiannis. A fixed point theorem for non-monotonic functions. *Theor. Comput. Sci.*, 574:18–38, 2015.

[27] A. J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[28] Peter Fishburn. Preference structures and their numerical representations. *Theoretical Computer Science*, 217(2):359–383, 1999.

[29] John P. Gallagher. Tutorial on specialisation of logic programs. In David A. Schmidt, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93, Copenhagen, Denmark, June 14-16, 1993*, pages 88–98. ACM, 1993.

[30] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[31] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[32] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference logic programming. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 731–745. MIT Press, 1995.

[33] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Optimization and relaxation in constraint logic languages. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 91–103. ACM Press, 1996.

[34] Kannan Govindarajan, Bharat Jayaraman, and Surya Mantha. Preference queries in deductive databases. *New Generation Comput.*, 19(1):57–86, 2000.

[35] Gösta Grahne, Alex Thomo, and William W. Wadge. Preferentially annotated regular path queries. In Thomas Schwentick and Dan Suciu, editors, *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*, volume 4353 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2007.

[36] Hai-Feng Guo and Bharat Jayaraman. Logic programming with solution preferences. *J. Log. Algebr. Program.*, 78(1):1–21, 2008.

[37] Sven Ove Hansson. Preference logic. In Dov M. Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic, vol 4*, pages 319–393. Springer, Dordrecht, 2001.

[38] Bharat Jayaraman, Kannan Govindarajan, and Surya Mantha. Preference logic grammars. *Comput. Lang.*, 24(3):179–196, 1998.

[39] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.*, 11(1):5–94, 2001.

[40] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[41] Maryam Khezrzadeh, Alex Thomo, and William W. Wadge. Harnessing the power of "favorites" lists for recommendation systems. In Lawrence D. Bergman, Alexander Tuzhilin, Robin D. Burke, Alexander Felfernig, and Lars Schmidt-Thieme, editors, *Proceedings of the 2009 ACM Conference on Recommender Systems, RecSys 2009, New York, NY, USA, October 23-25, 2009*, pages 289–292. ACM, 2009.

[42] Werner Kießling. Foundations of preferences in database systems. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 311–322. Morgan Kaufmann, 2002.

[43] Werner Kießling, Bernd Hafenrichter, Stefan Fischer, and Stefan Holland. Preference XPATH: A query language for e-commerce. In Hans Ulrich Buhl, Andreas Huther, and Bernd Reitwiesner, editors, *Information Age Economy: 5. Internationale Tagung Wirtschaftsinformatik 2001, Augsburg, Germany*, page 32. Physica Verlag / Springer, 2001.

[44] Werner Kießling and Gerhard Köstler. Preference SQL - design, implementation, experiences. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 990–1001. Morgan Kaufmann, 2002.

[45] Georgia Koutrika and Yannis E. Ioannidis. Personalization of queries in database systems. In Z. Meral Özsoyoglu and Stanley B. Zdonik, editors, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 597–608. IEEE Computer Society, 2004.

[46] M. Lacroix and Pierre Lavency. Preferences; putting more knowledge into queries. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 217–225. Morgan Kaufmann, 1987.

[47] Michael Leuschel. Logic program specialisation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*, volume 1706 of *Lecture Notes in Computer Science*, pages 155–188. Springer, 1998.

[48] Michael Leuschel and Germán Vidal. Fast offline partial evaluation of logic programs. *Inf. Comput.*, 235:70–97, 2014.

[49] Sarah Lichtenstein and Paul Slovic. *The Construction of Preference*. Cambridge University Press, 2006.

[50] John W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2nd extended edition, 1993.

[51] John W. Lloyd and John C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(3&4):217–242, 1991.

[52] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[53] Neil Mitchell and Colin Runciman. Losing functions without gaining data: another look at defunctionalisation. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 13–24. ACM, 2009.

[54] Stephen Muggleton. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.

[55] George Nelan. *Firstification*. PhD thesis, Arizona State University, USA, 1991.

[56] Giorgos Papadimitriou. A query language for lexicographic preferences. Master's thesis, University of Athens, Athens, 2017.

[57] Bernard J. Pope and Lee Naish. Specialisation of higher-order functions for debugging. *Electr. Notes Theor. Comput. Sci.*, 64:277–291, 2002.

[58] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

[59] J. Guadalupe Ramos, Josep Silva, and Germán Vidal. Fast narrowing-driven partial evaluation for inductively sequential programs. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 228–239. ACM, 2005.

[60] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[61] Panos Rondogiannis and Ioanna Symeonidou. Extensional semantics for higher-order logic programs with negation. *Logical Methods in Computer Science*, 14(2), 2018.

[62] Panos Rondogiannis and Antonis Troumpoukis. The infinite-valued semantics: overview, recent results and future directions. *Journal of Applied Non-Classical Logics*, 23(1-2):213–228, 2013.

[63] Panos Rondogiannis and Antonis Troumpoukis. Expressing preferences in logic programming using an infinite-valued logic. In Moreno Falaschi and Elvira Albert, editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 208–219. ACM, 2015.

[64] Panos Rondogiannis and William W. Wadge. Minimum model semantics for logic programs with negation-as-failure. *ACM Trans. Comput. Log.*, 6(2):441–467, 2005.

[65] Konstantinos Sagonas and David Scott Warren. Efficient execution of hilog in wam-based prolog implementations. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 349–363. MIT Press, 1995.

[66] Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artif. Intell.*, 123(1-2):185–222, 2000.

[67] Amartya Sen. *Collective choice and social welfare*. Harvard University Press, 1970.

[68] John C. Shepherdson. Unfold/fold transformations of logic programs. *Mathematical Structures in Computer Science*, 2(2):143–157, 1992.

[69] Kostas Stefanidis, Georgia Koutrika, and Evaggelia Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(3):19:1–19:45, 2011.

[70] Terrance Swift and David Scott Warren. XSB: extending prolog with tabled logic programming. *TPLP*, 12(1-2):157–187, 2012.

[71] Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In Sten-Åke Tärnlund, editor, *Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984*, pages 127–138. Uppsala University, 1984.

[72] Antonis Troumpoukis. A Prolog transformation of PrefLog programs. Unpublished Manuscript.

[73] Antonis Troumpoukis and Angelos Charalambidis. Predicate specialization for definitional higher-order logic programs. In Fred Mesnard and Peter J. Stuckey, editors, *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers*, volume 11408 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2018.

[74] Antonis Troumpoukis, Stasinos Konstantopoulos, and Angelos Charalambidis. An extension of SPARQL for expressing qualitative preferences. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 711–727. Springer, 2017.

[75] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*, volume 14 of *Principles of computer science series*. Computer Science Press, 1988.

[76] Maarten H. van Emden. Quantitative deduction and its fixpoint theory. *J. Log. Program.*, 3(1):37–53, 1986.

[77] William W. Wadge. Higher-order horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 289–303. MIT Press, 1991.

[78] Kewen Wang, Lizhu Zhou, and Fangzhen Lin. Alternating fixpoint theory for logic programs with priority. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2000.

[79] David H. D. Warren. Higher-order extensions to Prolog: Are they needed? In John E. Hayes, Donald Michie, and Yih-Hsing Pao, editors, *Machine Intelligence*, volume 10, pages 441–454. Ellis Horwood, 1982.

[80] Xi Zhang and Jan Chomicki. Preference queries over sets. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data*

Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, pages 1019–1030. IEEE Computer Society, 2011.

[81] Yan Zhang and Norman Y. Foo. Answer sets for prioritized logic programs. In Jan Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 69–83. MIT Press, 1997.

[82] Lydia Zogmpi. Implementation of the logic programming language PrefLog. Bachelor's thesis, University of Athens, Athens, 2016.