4ο ΠΑΝΕΛΛΗΝΙΟ ΣΥΝΕΔΡΙΟ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕ ΔΙΕΘΝΗ ΣΥΜΜΕΤΟΧΗ

ΠΡΑΚΤΙΚΑ ΕΙΣΗΓΗΣΕΩΝ ΤΟΜΟΣ Ι

ΧΟΡΗΓΟΣ ΕΚΔΟΣΗΣ

COMPUTER ΓΙΑ ΟΛΟΥΣ

ΤΟ ΕΛΛΗΝΙΚΟ ΠΕΡΙΟΔΙΚΟ ΓΙΑ BUSINESS COMPUTING

Υπο την αιγίδα του Υπουργείου Προεδρείας της Κυβερνήσεως.

ΕΡΕΥΝΑ ΑΝΑΠΤΥΞΗ ΕΦΑΡΜΟΓΕΣ

ΔΙΟΡΓΑΝΩΣΗ:

ΕΛΛΗΝΙΚΗ ΕΤΑΙΡΙΑ ΕΠΙΣΤΗΜΟΝΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΥΝΔΙΟΡΓΑΝΩΣΗ:

- ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ ΠΑΝ/ΜΙΟΥ ΠΑΤΡΑΣ
- ΙΝΣΤΙΤΟΥΤΟ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ
- ΠΑΝΕΛΛΗΝΙΟΣ ΣΥΛΛΟΓΟΣ ΔΙΠΛΩΜΑΤΟΥΧΩΝ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΥΝΕΡΓΑΣΙΑ:

ΣΥΛΛΟΓΟΥ ΦΟΙΤΗΤΩΝ ΤΜΗΜΑΤΟΣ ΜΗΧΑΝΙΚΩΝ Η/Υ ΚΑΙ ΠΛΗΡΟΦΟΡΙΚΗΣ

Βελτίωση της απόδοσης λογικών προγραμμάτων με τη χρήση μετασχηματισμών

Μανόλης Γεργατσούλης Μαρία Κατζουράκη

Ινστιτούτο Πληροφορικής & Τηλεπικοινωνιών
ΕΚΕΦΕ 'Δημόκριτος', 153 10 Αγία Παρασκευή Αττικής
e_mail: manolis@iit.nrcps.ariadne-t.gr

Περίληψη

Η εργασία αυτή αναφέρεται στη βελτίωση της απόδοσης λογικών προγραμμάτων με τη χρήση μετασχηματισμών. Παρουσιάζεται ένα σύστημα μετασχηματισμών διπλώνω/ξεδιπλώνω και εξετάζονται μορφές νέων ορισμών και στρατηγικές εφαρμογής των μετασχηματισμών, εστιάζοντας στη μετατροπή προγραμμάτων που περιλαμβάνουν προτάσεις με μη γραμμική αναδρομή σε ισοδύναμα με γραμμική αναδρομή. Οι προτεινόμενοι νέοι ορισμοί οδηγούν στη γενίκευση λίστας και την εισαγωγή συσσωρευτών.

Λέξεις Κλειδιά: μετασχηματισμοί προγραμμάτων, γενίκευση, λογικός προγραμματισμός

Abstract

This paper concerns the use of program transformations to improve the efficiency of logic programs. We present a fold/unfold program transfromation system and discuss the forms of the new definitions and the transformation strategies, focussing on the transformation of a common class of logic programs which include clauses with nonlinear recursion into equivalent but linear recusrive programs. The proposed new definitions and strategies result in tupling generalization and in accumulator introduction.

Ι. ΕΙΣΑΓΩΓΗ

Οι στόχοι της ανάπτυξης προγραμμάτων τα οποία να είναι από τη μια μεριά σωστά, κατανοητά, και εύκολα τροποποιήσημα, και από την άλλη να έχουν μικρές απαιτήσεις σε μνήμη και να είναι αποδοτικά, έρχονται συχνά σε αντίθεση μεταξύ τους. Ετσι, ο προγραμματιστής αναγκάζεται πολλές φορές να θυσιάσει κάποιους από τους στόχους αυτούς για να πετύχει τους υπόλοιπους. Αυτό έχει σαν συνέπεια να αναπτύσσονται δυσνόητα και πολύπλοκα προγράμματα, να αυξάνεται η πιθανότητα λαθών, και να δυσκολεύεται η διόρθωση τους, προκειμένου αυτά τα προγράμματα να είναι αποδοτικά. Η αντίφαση αυτή παρατηρείται τόσο στην ανάπτυξη προγραμμάτων με τις γλώσσες του κλασικού αλγοριθμικού προγραμματισμού και τις συναρτησιακές γλώσσες όσο και στον *λογικό προγραμματισμό* (logic programming), [29,30] στον οποίο αναφέρεται αυτή η εργασία. Μια βασική μεθοδολογία που στοχεύει στην αντιμετώπιση του προβλήματος αποτελούν οι *μετασχηματισμοί προγραμμάτων* (program transformations). Με βάση τη μεθοδολογία αυτή, ο προγραμματιστής αναπτύσσει το πρόγραμμα, συγκεντρώνοντας την προσοχή του στην ορθότητα, και την ευκολία κατανόησης του και αδιαφορόντας για τις απαιτήσεις του σε μνήμη και χρόνο. Στη συνέχεια, ένα σύστημα *μετασχηματισμών που διατηρούν την ορθότητα* (correctness preserving program transformations) αναλάβάνει τη μετατροπή του προγράμματος, αν είναι δυνατόν με αυτόματο ή ημιαυτόματο τρόπο, σε ισοδύναμο πρόγραμμα το οποίο έχει καλύτερη υπολογιστική συμπεριφορά από το αρχικό. Οι μετασχηματισμοί μπορούν να εφαρμοστούν σε διάφορα επίπεδα. Στο επίπεδο του πηγαίου κώδικα (source level), στο οποίο αναφερόμαστε στην εργασία αυτή,

στο επίπεδο κάποιου ενδιάμεσου κώδικα, ή στο επίπεδο του τελικού κώδικα.

Αξιόλογη κατηγορία μετασχηματισμών είναι οι μετασχηματισμοί *διπλώνω/ξεδιπλώνω* (fold/unfold) οι οποίοι αναπτύχθηκαν αρχικά από τους R. Burstall και J. Darlington [7] για συναρτησιακά προγράμματα και στη συνέχεια διατυπώθηκαν από τους H. Tamaki και T. Sato [27,26] για λογικά προγράμματα, έτσι ώστε να διατηρούν τη *μοντελοθεωρητική σημασία των προγραμμάτων* (model theoretic semantics) [15]. Το σοβαρότερο πρόβλημα στη χρήση των μετασχηματισμών αυτών, οι οποίοι αποτελούνται από ένα πολύ μικρό αλλά δυνατό σύνολο κανόνων, βρίσκεται στο σημαντικό βαθμό "εφυίας" που απαιτείται για την εισαγωγή νέων ορισμών και την επιλογή της κατάλληλης στρατηγικής για την εφαρμογή τους, αφού η ανεξέλεγκτη χρήση τους οδηγεί σε συνδιαστική έκρηξη. Οι λόγοι αυτοί καθιστούν δύσκολη την πλήρη αυτοματοποίηση τέτοιων συστημάτων. Παρόλα αυτά, οι μετασχηματισμοί διπλώνω/ξεδιπλώνω αποτελούν βασικό συστατικό πολλών εργασιών που αναφέρονται σε βελτίωση της απόδοσης λογικών προγραμμάτων [17,4,10,5,9,24], μερικό υπολογισμό (partial evaluation) και εξειδίκευση προγραμμάτων (program specialization) [16,2], ενσωμάτωση πληροφορίας ελέγχου σε λογικά προγράμματα [6] και σύνθεση προγραμμάτων από προδιαγραφές σε λογική 1ης τάξης [22,13,14,10]. Ανάλογες εργασίες αναφέρονται και για συναρτησιακά και αλγοριθμικά προγράμματα [1,7,28].

Στην εργασία αυτή εξετάζουμε τα αίτια της μειωμένης αποδοτικότητας των λογικών προγραμμάτων (ενότητα II) και στη συνέχεια εισάγουμε ένα σύνολο μετασχηματισμών διπλώνω/ξεδιπλώνω (ενότητα III). Οι μετασχηματισμοί αυτοί χρησιμοποιούνται στη συνέχεια (ενότητες IV έως VII) για να μετασχηματισουμε προγράμματα που ανήκουν σε μια συγκεκριμένη κατηγορία. Στόχος μας είναι να περιορίσουμε μη-γραμμική αναδρομή που εμφανίζουν τα προγράμματα αυτά. Αυτό οδηγεί συχνά στην αύξηση της αποδοτικότητας των προγραμμάτων. Σημαντικές για τους μετασχηματισμούς αποδεικνύονται κάποιες ιδιότητες που έχουν πολλά κατηγορήματα όπως είναι η ύπαρξη *μοναδιαίων στοιχείων* (identity elements) και η *προσεταιριστικότητα* (associativity). Βασικό σημείο της προσέγγισης που ακολουθούμε αποτελεί η μεθοδολογία εισαγωγής νέων ορισμών που προτείνουμε, η οποία οδηγεί στη *γενίκευση* (generalization) των διαδικασιών, καθώς και οι στρατηγικές για την εφαρμογή των μετασχηματισμών. Τελειώνοντας, στην ενότητα 8 παρουσιάζονται τα συμπεράσματα και γίνεται αναφορά σε συναφείς εργασίες.

II. ΣΧΕΤΙΚΑ ΜΕ ΤΗΝ ΑΠΟΔΟΤΙΚΟΤΗΤΑ ΤΩΝ ΛΟΓΙΚΩΝ ΠΡΟΓΡΑΜΜΑΤΩΝ

Η σύνταξη λογικών προγραμμάτων με στόχο την ορθότητα και την ευκολία κατανόησης οδηγεί συχνά στη μειωμένη αποδοτικότητα των προγραμμάτων αυτών. Η μικρή αποδοτικότητα οφείλεται και σε άλλους λόγους οι οποίοι σχετίζονται με τα χαρακτηριστικά του λογικού προγραμματισμού. Ανάμεσα στα αίτια του προβλήματος σημαντικά είναι :

- Η διάσχιση δομών δεδομένων περισσότερες φορές από όσες αυτό είναι αναγκαίο, καθώς και η δόμηση μη αναγκαίων ενδιάμεσων τιμών.
- Η πολλαπλός υπολογισμός του ίδιου υποστόχου
- Η ύπαρξη πλεοναζώντων υπολογισμών (redundant computations) και γενικότερα υπολογισμών που θα μπορούσαν να γίνουν σε χρόνο μετάφρασης
- Η ύπαρξη διαδικασιών της μορφής: λύσε(..) ← γέννησε_πιθανή_λύση(..), έλεγξε_ορθότητα_λύσης(..)
- Η ουσιαστική έλλειψη τρόπων έκφρασης επανάληψης όπως οι *while* και *for* άλλων γλωσσών, με αποτέλεσμα η επανάληψη να εκφράζεται σχεδόν αποκλειστικά μέσω της αναδρομής
- Η πολύ μικρή δυνατότητα που παρέχεται στον προγραμματιστή να επιδρά στον έλεγχο, καθώς και η "τυφλή" οπισθοδρόμηση χωρίς εξέταση των αιτιών της αποτυχίας
- Η ιδιότητα των λογικών μεταβλητών να παίρνουν μια μόνο τιμή (single assignment)

Στη βελτίωση της απόδοσης των λογικών προγραμμάτων και τη μείωση των απαιτήσεων τους σε μνήμη μπορούν να συμβάλλουν οι μετασχηματισμοί περιορίζοντας αρκετές από τις αιτίες που προκαλούν τα προβλήματα. Ετσι, με τη χρήση μετασχηματισμών γίνεται προσπάθεια να αποφευχθεί η πολλαπλή διάσχιση δομών δεδομένων (συγχώνευση ανακυκλώσεων) [19,9], να περιοριστούν οι πλεονάζουσες και ενδιάμεσες μεταβλητές [20], να περιοριστεί ο περιτός ξαναϋπολογισμός του ίδιου υποστόχου [5], να ενσωματωθεί πληροφορία ελέγχου σε δηλωτικά προγράμματα [6], να υπολογιστούν τμήματα του προγράμματος για τα οποία υπάρχουν αρκετά δεδομένα σε χρόνο μετάφρασης [2,16], να εμφυτευτεί ο έλεγχος μέσα στη διαδικασία παραγωγής για προβλήματα του σχήματος δημιούργησε_λύση-έλεγξε_ορθότητα [24], να μετατραπεί η αναδρομή σε αναδρομή ουράς [9], να αντικατασταθούν δομές δεδομένων που χρησιμοποιεί το πρόγραμμα με άλλες οι οποίες μπορούν να χρησιμοποιηθούν πιο αποδοτικά [12].

Εναλλαχτική προσέγγιση στη χρήση μετασχηματισμών αποτελεί η εκτέλεση του προγράμματος με τη χρήση συστημάτων που ασκούν "πιό έξυπνο έλεγχο" από ότι η στάνταρτ Prolog, είτε αυτόματα είτε με βάση οδηγίες ελέγχου που διατυπώνονται από το χρήστη. Τέτοια συστήματα περιλαμβάνουν δυνατότητες όπως *ψευδοπαράλληλη εκτέλεση* (corroutines), [8], *έξυπνη οπισθοδρόμηση* (intelligent backtracking) χρήση μετα-κατηγορήματων και μετα-κανόνων για έλεγχο [11] κ.λ.π.

III. ΟΙ ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΔΙΠΛΩΝΩ/ΞΕΔΙΠΛΩΝΩ

Οι μετασχηματισμοί διπλώνω/ξεδιπλώνω (fold/unfold) αναπτύχθηκαν αρχικά από τους R. Burstall και J. Darlington [7] για προγράμματα γραμμένα σε μορφή αναδρομικών συναρτήσεων. Ανάλογους μετασχηματισμούς διαμόρφωσαν οι H. Tamaki και T. Sato [27,26] για λογικά προγράμματα, έτσι ώστε να διατηρούν τη μοντελοθεωρητική σημασία των προγραμμάτων (model theoretic semantics). Η χρήση των μετασχηματισμών στα λογικά προγράμματα είναι πιο δύσκολη από ότι στα συναρτησιακα αφού εδώ δεν έχουμε εφαρμογή και σύνθεση συναρτήσεων αλλά υπολογισμό σχέσεων μέσω της ταυτοποίησης, καθώς και λόγω του μη ντετερμινισμού. Στη συνέχεια παραθέτουμε τους μετασχηματισμούς διπλώνω/ξεδιπλώνω καθώς και άλλους χρήσιμους μετασχηματισμούς βασιζόμενοι στην εργασία [26].

1) *Νέοι ορισμοί* (Definitions): Ο ορισμός μιας νέας διαδικασίας εισάγεται στο πρόγραμμα. Ο ορισμός αυτός εισάγει μια σχεση με ένα νέο κατηγόρημα p το οποίο δεν υπάρχει στο αρχικό πρόγραμμα, με τη βοήθεια διαδικασιών που υπάρχουν ήδη στο πρόγραμμα. Ο νέος ορισμός μπορεί να είναι αναδρομικός.

2) *Ξεδίπλωμα* (Unfolding): Εστω η ακόλουθη πρόταση c

$c: \quad A \leftarrow A_1, ..., A_{i-1}, A_i, A_{i+1}, ..., A_k.$

και $c_1, ..., c_m$ όλες οι προτάσεις ενός προγράμματος P, των οποίων η κεφαλή ταυτοποιείται με τον ατομικό τύπο A_i με πιο γενικούς ταυτοποιητές τους $\theta_1, ..., \theta_m$ αντίστοιχα. Το *ξεδίπλωμα της c στο* A_i συνίσταται στην αντικατάσταση της c από τις προτάσεις $c'_1, ..., c'_m$ έτσι ώστε αν

$c_j: \quad B_j \leftarrow B_{j_1}, ..., B_{j_h}. \quad$ και $\quad B_j\theta_j = A_i\theta_j, \quad$ για όλα τα $j \ : \ 1 \leq j \leq m$

τότε

$c'_j: \quad (A \leftarrow A_1, ..., A_{i-1}, B_{j_1}, ..., B_{j_h}, A_{i+1}, ..., A_k)\theta_j.$

3) *Αντιστρέψιμο δίπλωμα* (reversible folding) : Εστω η ακόλουθη πρόταση c

$c: \quad B \leftarrow B_1, ..., B_{h_1}, ..., B_{h_k}, ..., B_n.$

Εστω ότι υπάρχει πρόταση c_1 στο πρόγραμμα P

$c_1: \quad A \leftarrow A_1, ..., A_k$

και αντικατάσταση θ για τις μεταβλητές του A τέτοια ώστε $A_i\theta = B_{h_i}$ για $i = 1, k$

τότε το *δίπλωμα των* $B_{h_1}, ..., B_{h_k}$ *στη c με τη* c_1 οδηγεί στην αντικατάσταση της c από τη c' όπου

$c': \quad B \leftarrow B_1, ..., (A)\theta, ..., B_n$

με τη προυπόθεση ότι η πρόταση c_1 είναι η μόνη πρόταση στο P με την οποία μπορεί να εκτελεστεί η κλήση $(A)\theta$, ξαναπαίρνοντας έτσι τη πρόταση c από τη πρόταση c'.

4) *Κανόνας αντικατάστασης* (replacement rule) είναι ένας κανόνας της μορφής

$\exists X_1, ..., X_n M_1, ..., M_k \Rightarrow \exists Y_1, ..., Y_m N_1, ..., N_h$

όπου όλες οι μεταβλητές που περιλαμβάνονται στα $M_1, ..., M_k$ εκτός από τις $X_1, ..., X_n$ περιλαμβάνονται και στα $N_1, ..., N_h$.

Ενας κανόνας αντικατάστασης αντικαταθιστά κάποιους στόχους στο σώμα μιας πρότασης από άλλους. Ενας τέτοιος κανόνας είναι *ορθός* (correct) ως προς ένα πρόγραμμα P αν για κάθε στιγμιότυπο χωρίς μεταβλητές του κανόνα ισχύει ότι : η σύζευξη των στόχων του δεξιού μέρους έπεται από το πρόγραμμα αν και μόνο αν η σύζευξη των στόχων του αριστερού του μέρους έπεται από το πρόγραμμα.

Ειδικές περιπτώσεις του κανόνα αντικατάστασης είναι η *διαγραφή στόχου* (goal deletion), η *συγχώνευση ταυτόσημων στόχων* (goal merging), και η *εισαγωγή στόχων* (goal addition). Ακόμη, εξαιρετικά χρήσιμοι κανόνες αντικατάστασης, όπως θα δούμε, είναι αυτοί που αναφέρονται σε ιδιότητες των κατηγορημάτων. Στη συνέχεια θα χρησιμοποιήσουμε τρείς τέτοιους κανόνες αντικατάστασης. Ο πρώτος αναφέρεται στην *προσεταιριστική ιδιότητα* (associativity) ενός κατηγορήματος και εκφράζεται από :

$\exists R1\ (qa(X, Y, R1), qa(R1, Z, R)) \Rightarrow \exists R2\ (qa(Y, Z, R2), qa(X, R2, R))$

Οι επόμενοι δύο αναφέρονται στην ύπαρξη *δεξιού μοναδιαίου στοιχείου* ι_r (right identity element) και *αριστερού μοναδιαίου στοιχείου* ι_l (left identity element) αντίστοιχα :

$qa(X, \iota_r, Y) \Rightarrow X = Y$

$qa(\iota_l, X, Y) \Rightarrow X = Y$

Η εφαρμογή των παραπάνω κανόνων μετασχηματισμού αποδεικνύεται [26] ότι διατηρεί τη *σημασία* (meaning) των προγραμμάτων, εφόσον τηρηθούν κάποιες προυποθέσεις κατά την εφαρμογής τους. Σαν σημασία M(P) ενός προγράμματος P ορίζεται το σύνολο M(P) = {G : G ατομικός τύπος χωρίς μεταβλητές (ground) που μπορεί να αποδειχθεί από το πρόγραμμα P}.

Η εφαρμογή των μετασχηματισμών που παρουσιάσαμε συναντά δύο σοβαρά προβλήματα.

- Τη συνδιαστική έκρηξη που προκαλείται από την ανεξέλεγκτη εφαρμογή των μετασχηματισμών (κύρια του ξεδιπλώματος).
- Τη δυσκολία εύρεσης των καταλλήλων νέων ορισμών, οι οποίοι γι'αυτό ονομάζονται και *Εύρηκα* (Eureka) [7], δουλειά που απαιτεί αρκετή εφυΐα και δυσκολεύει την αυτοματοποίηση.

Το πρόβλημα της εύρεσης νέων ορισμών καθώς και της εύρεσης αλγορίθμων για την εφαρμογή των μετασχηματισμών έχει αποδειχθεί *μη επιλύσιμο* (unsolvable) [18], ακόμα και για αρκετά απλές μορφές προγραμμάτων. Ετσι, πολλοί ερευνητές καταφεύγουν στη χρήση εμπειρικών κανόνων και στρατηγικών για την αντιμετώπιση των προβλημάτων αυτών [19,20,9,5]. Οι στρατηγικές που έχουν προταθεί, γενικά αφορούν την εισαγωγή μη αναδρομικών νέων ορισμών. Μια τέτοια περίπτωση θα δούμε στη συνέχεια.

IV. ΕΙΣΑΓΩΓΗ ΣΥΣΣΩΡΕΥΤΩΝ

Η περίπτωση της γραμμικής αναδρομής

Δίνεται η παρακάτω διαδικασία *e_sum* η οποία υπολογίζει το άθροισμα των στοιχείων μιας λίστας.

(1) $e\_sum([\,], 0)$.

(2) $e\_sum([X|Xs], S) \leftarrow e\_sum(Xs, S1),\ S\ is\ S1 + X$.

Στόχος μας είναι να μετατρέψουμε τη διαδικασία αυτή σε ισοδύναμη με αναδρομή ουράς. Εισάγουμε τον ακόλουθο νέο ορισμό (Eureka) :

(*D*) $e\_sum\_acc(L, Acc, S) \leftrightarrow e\_sum(L, S1), S\ is\ S1 + Acc$.

Δίνοντας τη τιμή 0 στη μεταβλητή *Acc* στην (D) παίρνουμε :

(3) $e\_sum\_acc(L, 0, S) \leftrightarrow e\_sum(L, S1), S\ is\ S1 + 0$.

Επειδή το 0 είναι μοναδιαίο στοιχείο της πράξης +, η (3) μετατρέπεται στην

(4) $e\_sum\_acc(L, 0, S) \leftrightarrow e\_sum(L, S)$.

Από την οποία παίρνουμε την (5)

(5) $e\_sum(L, S) \leftarrow e\_sum\_acc(L, 0, S)$.

Τώρα προσπαθούμε να βρούμε έναν αναδρομικό ορισμό για τη διαδικασία e_sum_acc. Ετσι, ξεδιπλώνουμε την (D) στο e_sum με τις προτάσεις (1) και (2). Παίρνουμε έτσι τις προτάσεις

(6) $e\_sum\_acc([\,], Acc, S) \leftarrow S \; is \; 0 + Acc$.

(7) $e\_sum\_acc([X|Xs], Acc, S) \leftarrow e\_sum(Xs, SXs), S1 \; is \; SXs + X, S \; is \; S1 + Acc$.

Η πρόταση (6) απλοποιείται δίνοντας την

(8) $e\_sum\_acc([\,], S, S)$.

ενώ η (7) μετατρέπεται στην (9) αξιοποιώντας την προσεταιριστικότητα της +, και αναδιατάσσοντας τις κλήσεις στο σώμα της πρότασης

(9) $e\_sum\_acc([X|Xs], Acc, S) \leftarrow Acc1 \; is \; X + Acc, e\_sum(Xs, SXs), S \; is \; SXs + Acc1$.

Διπλώνοντας την (9) με τη βοήθεια της (D) παίρνουμε

(10) $e\_sum\_acc([X|Xs], Acc, S) \leftarrow Acc1 \; is \; X + Acc, e\_sum\_acc(Xs, Acc1, S)$.

Ετσι, το νέο πρόγραμμα για τη διαδικασία e_sum απαρτίζεται από τις προτάσεις {5, 8, 10} και είναι:

(11) $e\_sum1(L, S) \leftarrow e\_sum\_acc(L, 0, S)$.

(12) $e\_sum\_acc([\,], S, S)$.

(13) $e\_sum\_acc([X|Xs], Acc, S) \leftarrow Acc1 \; is \; X + Acc, e\_sum\_acc(Xs, Acc1, S)$.

Το πρόγραμμα αυτό είναι ισοδύναμο με το {1, 2} και εμφανίζει αναδρομή ουράς. Η μεταβλητή Acc παίζει το ρόλο *συσσωρευτή* (accumulator) ο οποίος συσσωρεύει τα μερικά αποτελέσματα.

Εισαγωγή συσσωρευτών σε μή-γραμμικές αναδρομικές διαδικασίες

Εισαγωγή συσσωρευτών γίνεται με παρόμοιο τρόπο, και σε μη-γραμμικές αναδρομικές διαδικασίες, όπως προκύπτει από το παράδειγμα που ακολουθεί. Το πρόγραμμα *flatree* μαζεύει σε μια λίστα τις πληροφορίες των κόμβων ενός δυαδικού δέντρου με ετικέτες (labeled binary tree).

(1) $flattree(void, [\,])$.

(2) $flattree(t(E, LT, RT), [E|L]) \leftarrow flattree(LT, LL), flattree(RT, RL), append(LL, RL, L)$.

(3) $append([\,], X, X)$.

(4) $append([X|Xs], Y, [X|Z]) \leftarrow append(Xs, Y, Z)$.

Για το μετασχηματισμό του προγράμματος εισάγουμε το νέο ορισμό (D) ο οποίος είναι παρόμοιος με τον ορισμό της προηγούμενης παραγράφου.

(D) $flat\_acc(T, Acc, L) \leftrightarrow flattree(T, L1), append(L1, Acc, L)$.

Δίνοντας στη μεταβλητή Acc στην (D) τη τιμή $[\,]$ και απλοποιώντας (αξιοποιούμε το γεγονός ότι το $[\,]$ είναι δεξιό μοναδιαίο στοιχείο της σχέσης *append*) παίρνουμε

(5) $flat\_acc(T, [\,], L) \leftrightarrow flattree(T, L1)$.

Από τη (5) προκύπτει η

(6) $flattree(T, L1) \leftarrow flat\_acc(T, [\,], L)$.

Και εδώ αναζητούμε έναν αναδρομικό ορισμό για τη διαδικασία $flat\_acc$. Ετσι, ξεδιπλώνουμε τη (D) στην *flattree* με τις προτάσεις (1) και (2), παίρνοντας έτσι τις προτάσεις

(7) $flat\_acc(void, Acc, L) \leftarrow append([\,], Acc, L)$.

(8) $flat\_acc(t(E, LT, RT), Acc, L) \leftarrow flattree(LT, LL), flattree(RT, LR),$
$append(LL, LR, L1), append([E|L1], Acc, L)$.

Τώρα, ξεδιπλώνουμε την (7) στο *append* με τη πρόταση (3) και παίρνουμε

(9) $flat\_acc(void, L, L)$.

Ξεδιπλώνοντας την (8) στο $append([E|L1], Acc, L)$ με την (4) παίρνουμε

(10) $flat\_acc(t(E, LT, RT), Acc, [E|L]) \leftarrow flattree(LT, LL), flattree(RT, LR),$
$append(LL, LR, L1), append(L1, Acc, L)$.

Αξιοποιώντας την προσεταιριστικότητα της *append* και αναδιατάζοντας τους στόχους, η (10) γίνεται

(11) $flat\_acc(t(E, LT, RT), Acc, [E|L]) \leftarrow flattree(RT, LR), append(LR, Acc, Acc1),$
$flattree(LT, LL), append(LL, Acc1, L).$

Τέλος, διπλώνουμε (2 φορές) την (11) χρησιμοποιώντας τη (D). Ετσι παίρνουμε τη

(12) $flat\_acc(t(E, LT, RT), Acc, [E|L]) \leftarrow flat\_acc(RT, Acc, Acc1), flat\_acc(LT, Acc1, L).$

Οι προτάσεις {6,9,12} αποτελούν το νέο πρόγραμμα για το *flattree*. Το πρόγραμμα αυτό εξακολουθεί να έχει μη γραμμική αναδρομή, είναι όμως πιο αποδοτικό από ότι το αρχικό αφού η κλήση στη διαδικασία *append* δεν υπάρχει πια. Γενικεύοντας την παραπάνω συζήτηση μπορούμε να δούμε την ακόλουθη στρατηγική για περιπτώσεις αναδρομικών προγραμμάτων που ορίζουν ένα κατηγόρημα R με γραμμική ή μή γραμμική αναδρομή τα οποία στις κλήσεις των αναδρομικών τους προτάσεων περιλαμβάνουν την κλήση μιας διαδικασίας Q η οποια είναι προσεταιριστική και έχει μοναδιαίο στοιχείο.

1) Εισάγουμε ένα μη αναδρομικό νέο ορισμό της μορφής

$(Def) \quad R'(...) \leftrightarrow R(...), Q(...)$

Οι όροι των κατηγορημάτων είναι μεταβλητές. Οι κοινές μεταβλητές των R και Q αντανακλούν τη σύνδεση των κλήσεων στην αναδρομική διαδικασία του προγράμματος.

2) Αξιοποιώντας το μοναδιαίο στοιχείο της Q παίρνουμε από τη (Def) μια πρόταση της μορφής

$R(...) \leftarrow R'(...)$

3) Στη συνέχεια αναζητούμε αναδρομικό ορισμό για το R' εφαρμόζοντας τα βήματα

3.1) Ξεδίπλωσε τη πρόταση (Def) στο R. Διπλώσε κάθε πρόταση που προκύπτει με τη (Def) εφαρμόζοντας πρίν, αν χρειάστει, τον κανόνα για την προσεταιριστικότητα και κάνοντας απλοποιήσεις.

V. ΓΕΝΙΚΕΥΣΗ ΛΙΣΤΑΣ

Στόχος μας εδώ είναι να μετατρέψουμε το πρόγραμμα { 1, 2, 3, 4} της προηγούμενης παραγράφου σε άλλο ισοδύναμο με γραμμική αναδρομή. Εισάγουμε γι'αυτό τον ακόλουθο νέο (αναδρομικό) ορισμό.

$(D1) \quad f\_tpl([\,], [\,]).$

$(D2) \quad f\_tpl([T|Ts], L) \leftarrow flattree(T, FT), f\_tpl(Ts, LTs), append(FT, LTs, L).$

Η *συμπλήρωση του κατηγορήματος* (completion) [15] f_tpl είναι

$(C) \quad f\_tpl(TL, L) \leftrightarrow (TL = [\,], L = [\,])$ *or*
$(TL = [T|Ts], flattree(T, FT), f\_tpl(Ts, LTs), append(FT, LTs, L))$

Δίνοντας τη τιμή $[\,]$ στη μεταβλητή Ts στη (C) παίρνουμε

(5) $f\_tpl(TL, L) \leftrightarrow false$ *or* $(TL = [T], flattree(T, FT), f\_tpl([\,], LTs), append(FT, LTs, L))$

Η (5) απλοποιείται δίνοντας την (6)

(6) $f\_tpl([T], L) \leftrightarrow flattree(T, FT), append(FT, [\,], L)$

Χρησιμοποιούμε το μοναδιαίο στοιχείο ($[\,]$) της *append* και παίρνουμε το ένα μέρος της συνεπαγωγής:

(7) $flattree(T, L) \leftarrow f\_tpl([T], L).$

Προσπαθούμε τώρα να απαλείψουμε τη κλήση της *flattree* από τη (D2). Γιαυτό ξεδιπλώνουμε τη (D2) στη *flattree* χρησιμοποιώντας τις προτάσεις (1) και (2). Παίρνουμε έτσι τις προτάσεις

(8) $f\_tpl([void|Ts], L) \leftarrow f\_tpl(Ts, LTs), append([\,], LTs, L).$

(9) $f\_tpl([t(E, LT, RT)|Ts], L) \leftarrow flattree(LT, LL), flattree(RT, RL),$
$append(LL, RL, R), f\_tpl(Ts, LTs), append([E|R], LTs, L).$

Ξεδιπλώνοντας την (8) στο *append* με τη (4) παίρνοντας έτσι

(10) $f\_tpl([void|Ts], L) \leftarrow f\_tpl(Ts, L).$

Ακολούθως, ξεδιπλώνουμε την (9) στο $append([E|R], LTs, L)$ με την (4) παίρνοντας την (11)

(11) $f\_tpl([t(E, LT, RT)|Ts], [E|L]) \leftarrow flattree(LT, LL), flattree(RT, RL),$
$append(LL, RL, R), f\_tpl(Ts, LTs), append(R, LTs, L).$

Η αναδιάταξη των στόχων και η αξιοποίηση της προσεταιριστικότητας του *append* στην (11) δίνει τη

(12) $f\_tpl([t(E,LT,RT)|Ts],[E|L]) \leftarrow flattree(LT,LL), \underline{flattree(RT,RL)},$
$\underline{f\_tpl(Ts,LTs), append(RL,LTs,W)}, append(LL,W,L).$

Διπλώνοντας τη (12) με τη (D2) παίρνουμε τη (13)

(13) $f\_tpl([t(E,LT,RT)|Ts],[E|L]) \leftarrow flattree(LT,LL), f\_tpl([RT|Ts],W), append(LL,W,L).$

Ενα ακόμη δίπλωμα με τη (D2) δίνει

(14) $f\_tpl([t(E,LT,RT)|Ts],[E|L]) \leftarrow f\_tpl([LT,RT|Ts],L).$

Ετσι παίρνουμε τον τελικό ορισμό για την *f_tpl*, που είναι ο ακόλουθος :

(15) $flattree(T,L) \leftarrow f\_tpl([T],L).$
(16) $f\_tpl([\,],[\,]).$
(17) $f\_tpl([void|Ts],L) \leftarrow f\_tpl(Ts,L).$
(18) $f\_tpl([t(E,LT,RT)|Ts],[E|L]) \leftarrow f\_tpl([LT,RT|Ts],L).$

Το πρόγραμμα αυτό έχει γραμμική αναδρομή (και μάλιστα αναδρομή ουράς). Ακόμη, η απαλειφή της κλήσης στη διαδικασία *append* συμβάλει στην αύξηση της αποδοτικότητας του προγράμματος. Γενικεύοντας το παράδειγμα αυτό εξάγουμε την ακόλουθη στρατηγική.

1) Εισήγαγε ένα νέο αναδρομικό ορισμό της μορφής

$(D1)$ $R\_tpl([\,],\iota)$
$(D2)$ $R\_tpl([X|Y],Z) \leftarrow R(X,XX), R\_tpl(Y,Z1), Q(XX,Z1,Z)$

2) Βρες τη σχέση που συνδέει την R με την R_tpl χρησιμοποιώντας το μοναδιαίο στοιχείο της Q.

3) Προσπάθησε να απαλείψεις την R από τη (D2) ως εξής 3.1) Ξεδίπλωσε τη (D2) στην R . 3.2) Προσπάθησε να διπλωσεις τις προτάσεις που προκύπτουν στο 3.1 με τη (D2). Χρησιμοποίησε αν χρειαστεί ξεδίπλωμα και κανόνες αντικατάστασης.

VI. ΣΥΝΔΙΑΖΟΝΤΑΣ ΤΗ ΓΕΝΙΚΕΥΣΗ ΛΙΣΤΑΣ ΜΕ ΤΗΝ ΕΙΣΑΓΩΓΗ ΣΥΣΣΩΡΕΥΤΗ

Στο επόμενο παράδειγμα έχουμε τη διαδικασία *sumnodes* η οποία υπολογίζει το άθροισμα των τιμών των φύλλων ενός δυαδικού δέντρου χωρίς ετικέτες (unlabeled binary tree).

(1) $sumnodes(leaf(N),N).$
(2) $sumnodes(t(LT,RT),S) \leftarrow sumnodes(LT,SL), sumnodes(RT,SR), S\ is\ SL+SR.$

Οπως και προηγουμένως ορίζουμε

$(D1)$ $sum\_tpl([\,],0).$
$(D2)$ $sum\_tpl([T|Ts],S) \leftarrow sumnodes(T,ST), sum\_tpl(Ts,STs), S\ is\ ST+STs.$

Παίρνοντας και εδώ τη συμπλήρωση (completion) του κατηγορήματος *sum_tpl*, δίνοντας την τιμή [ ] στη μεταβλητή Ts, και απλοποιώντας όπως και προηγουμένως, παίρνουμε

(3) $sum\_tpl([T],S) \leftrightarrow sumnodes(T,S).$

Ξεδιπλώνοντας τη (D2) στη *sumnodes* με τις προτάσεις (1) και (2) παίρνουμε

(4) $sum\_tpl([leaf(N)|Ts],S) \leftarrow sum\_tpl(Ts,STs), S\ is\ N+STs.$
(5) $sum\_tpl([t(LT,RT)|Ts],S) \leftarrow sumnodes(LT,SL), sumnodes(RT,SR),$
$ST\ is\ SL+SR, sum\_tpl(Ts,STs), S\ is\ ST+STs.$

Λόγω της προσεταιριστικότητας της πρόσθεσης (+), παίρνουμε από την (6)

(6) $sum\_tpl([t(LT,RT)|Ts],S) \leftarrow sumnodes(LT,SL),$
$\underline{sumnodes(RT,SR), sum\_tpl(Ts,STs), W\ is\ SR+STs}, S\ is\ SL+W.$

διπλώνοντας τα υπογραμμισμένα άτομα με τη (D2) παίρνουμε

(7) $sum\_tpl([t(LT,RT)|Ts],S) \leftarrow sumnodes(LT,SL), sum\_tpl([RT|Ts],W), S\ is\ SL+W.$

Ενα ακόμη δίπλωμα δίνει

(8) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sum\_tpl([LT, RT|Ts], S)$.

Ετσι, το τελικό πρόγραμμα, το οποίο είναι ισοδύναμο με το αρχικό είναι

(9) $sumnodes(T, S) \leftrightarrow sum\_tpl([T], S)$.
(10) $sum\_tpl([\,], 0)$.
(11) $sum\_tpl([leaf(N)|Ts], S) \leftarrow sum\_tpl(Ts, STs), S\ is\ N + STs$.
(12) $sum\_tpl([t(LT, RT)|Ts], S) \leftarrow sum\_tpl([LT, RT|Ts], S)$.

Το πρόγραμμα αυτό έχει γραμμική αναδρομή, όχι όμως αναδρομή ουράς (λόγω της πρότασης (11)). Είναι όμως δυνατόν να μετασχηματιστεί σε ισοδύναμο πρόγραμμα με αναδρομή ουράς, αν εισάγουμε ένα συσσωρευτή με τη βοήθεια της στρατηγικής της ενότητας 4.2. Γιαυτό εισάγουμε τον ορισμό :

$(D3)$ $s\_tpl\_acc(L, Acc, S) \leftrightarrow sum\_tpl(L, S1), S\ is\ Acc + S1$.

Δίνοντας την τιμή 0 στη μεταβλητή Acc και την τιμή $[T]$ στη μεταβλητή L στη (D3), και απλοποιώντας παίρνουμε

(13) $s\_tpl\_acc([T], 0, S) \leftrightarrow sum\_tpl([T], S)$.

Η (13) σε συνδιασμό με την (9) δίνει

(14) $sumnodes(T, S) \leftarrow s\_tpl\_acc([T], 0, S)$.

Ας αναζητήσουμε τώρα έναν αναδρομικό ορισμό για την s_tpl_acc. Ξεκινώντας ξεδιπλώνουμε τη sum_tpl στη (D3) χρησιμοποιώντας τις προτάσεις (10), (11) και (12). Παίρνουμε έτσι

(15) $s\_tpl\_acc([\,], Acc, S) \leftarrow S\ is\ Acc + 0$.
(16) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow sum\_tpl(Ts, STs), S1\ is\ N + STs, S\ is\ Acc + S1$.
(17) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl([LT, RT|Ts], S1), S\ is\ Acc + S1$.

Η (15) απλοποιείται δίνοντας τη

(18) $s\_tpl\_acc([\,], S, S)$.

Λόγω της προσεταιριστικότητας της πρόσθεσης (+) και με αναδιάταξη των στόχων, η (16) δίνει την

(19) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1\ is\ Acc + N, sum\_tpl(Ts, STs), S\ is\ Acc1 + STs$.

Διπλώνοντας την (19) με τη βοήθεια της (D3) παίρνουμε

(20) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1\ is\ Acc + N, sum\_tpl\_acc(Ts, Acc1, S)$.

Τώρα διπλώνουμε τη (17) με τη (D3) παίρνοντας

(21) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl\_acc([LT, RT|Ts], Acc, S)$.

Ετσι παίρνουμε τον νέο πρόγραμμα για τη $sumnodes$ το οποίο είναι

(22) $sumnodes(T, S) \leftarrow s\_tpl\_acc([T], 0, S)$.
(23) $s\_tpl\_acc([\,], S, S)$.
(24) $s\_tpl\_acc([leaf(N)|Ts], Acc, S) \leftarrow Acc1\ is\ Acc + N, sum\_tpl\_acc(Ts, Acc1, S)$.
(25) $s\_tpl\_acc([t(LT, RT)|Ts], Acc, S) \leftarrow sum\_tpl\_acc([LT, RT|Ts], Acc, S)$.

VII. ΜΙΑ ΑΛΛΗ ΜΟΡΦΗ ΝΕΩΝ ΟΡΙΣΜΩΝ ΓΙΑ ΓΕΝΙΚΕΥΣΗ ΛΙΣΤΑΣ

Η διαδικασία $maxtree$ που ακολουθεί βρίσκει τη μεγαλύτερη από τις τιμές που υπάρχουν στα φύλλα ενός δυαδικού δέντρου χωρίς ετικέτες (unlabeled binary tree). Οι τιμές αυτές πραγματικοί αριθμοί.

(1) $maxtree(leaf(N), N)$.
(2) $maxtree(t(LT, RT), M) \leftarrow maxtree(LT, M1), maxtree(RT, M2), maxof2(M1, M2, M)$.
(3) $maxof2(M, M2, M) \leftarrow M >= M2$.
(4) $maxof2(M1, M, M) \leftarrow M > M1$.

Η $maxof$ είναι προσεταιριστική δεν έχει όμως μοναδιαίο στοιχείο. Αυτό μας εμποδίζει να εισάγουμε νέο ορισμό της μορφής της ενότητας 5. Ετσι θα δούμε μια άλλη μορφή νέων ορισμών. Εισάγουμε

$(D1)$ $max\_tpl(T, [\,], M) \leftarrow maxtree(T, M)$.
$(D2)$ $max\_tpl(T, [H|Hs], M) \leftarrow maxtree(T, M1), max\_tpl(H, Hs, M2), maxof2(M1, M2, L)$.

Η συμπλήρωση (completion) του max_tpl είναι

(C) $max\_tpl(T, L, M) \leftrightarrow maxtree(T, M1)$,
$(L = [\,], M = M1)$ *or* $(L = [H|Hs], max\_tpl(H, Hs, M2), maxof2(M1, M2, L))$.

Δίνοντας την τιμή $[\,]$ στη μεταβλητή L παίρνουμε από τη (C)

(5) $maxtree(T, M) \leftrightarrow max\_tpl(T, [\,], M)$.

Σε αναλογία με την ενότητα 5, προσπαθούμε να απαλείψουμε τη κλήση $maxtree$ από τον ορισμό {D1,D2}. Ξεδιπλώνουμε γιαυτό τις (D1) και (D2) στη $maxtree$ με τις προτάσεις (1) και (2) παίρνοντας

(6) $max\_tpl(leaf(N), [\,], N)$.
(7) $max\_tpl(t(LT, RT), [\,], M) \leftarrow maxtree(LT, ML), maxtree(RT, MR), maxof2(ML, MR, M)$.
(8) $max\_tpl(leaf(N), [H|Hs], M) \leftarrow max\_tpl(H, Hs, M2), maxof2(N, M2, M)$.
(9) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow maxtree(LT, ML), maxtree(RT, MR)$,
$maxof2(ML, MR, M1), max\_tpl(H, Hs, M2), maxof2(M1, M2, M)$.

Διπλώνοντας την κλήση $maxtree(RT, MR)$ στην (7) με τη (D1) παίρνουμε

(10) $max\_tpl(t(LT, RT), [\,], M) \leftarrow maxtree(LT, ML)$,
$max\_tpl(RT, [\,], MR), maxof2(ML, MR, M)$.

Τώρα διπλώνουμε την (10) με τη βοήθεια της (D2). Ετσι παίρνουμε

(11) $max\_tpl(t(LT, RT), [\,], M) \leftarrow max\_tpl(LT, [RT], M)$.

Εφαρμόζοντας στην (9) το κανόνα αντικατάστασης για τη προσεταιριστικότητα της $maxof2$ παίρνουμε

(12) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow maxtree(LT, ML), \underline{maxtree(RT, MR)}$,
$\underline{max\_tpl(H, Hs, M2), maxof2(MR, M2, L1)}, maxof2(ML, L1, M)$.

Διπλώνοντας τους υπογραμμισμένους στόχους στη (12) με τη (D2) παίρνουμε

(13) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow maxtree(LT, ML), max\_tpl(RT, [H|Hs], L1)$,
$maxof2(ML, L1, M)$.

Διπλώνοντας άλλη μια φορά με τη (D2) παίρνουμε

(14) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow max\_tpl(LT, [RT, H|Hs], M)$.

Το πρόγραμμα που προκύπτει συνοψίζεται στις προτάσεις:

(15) $maxtree(T, M) \leftarrow max\_tpl(T, [\,], M)$.
(16) $max\_tpl(leaf(N), [\,], N)$.
(17) $max\_tpl(leaf(N), [H|Hs], M) \leftarrow max\_tpl(H, Hs, M2), maxof2(N, M2, M)$.
(18) $max\_tpl(t(LT, RT), [\,], M) \leftarrow max\_tpl(LT, [RT], M)$.
(19) $max\_tpl(t(LT, RT), [H|Hs], M) \leftarrow max\_tpl(LT, [RT, H|Hs], M)$.

Το πρόγραμμα αυτό έχει γραμμική αναδρομή, όχι όμως αναδρομή ουράς (λόγω της πρότασης (17)). Προσπαθώντας να εισάγουμε ένα συσσωρευτή με τη στρατηγική της ενότητας 4.2 βρισκόμαστε πάλι αντιμέτωποι με την έλλειψη μοναδιαίου στοιχείου για τη $maxof$ πράγμα που μας αναγκάζει να διαφοροποιήσουμε τη στρατηγική μας. Ετσι, εισάγουμε τον επόμενο νέο ορισμό.

(D3) $max\_tpl\_acc(H, Hs, N, M) \leftarrow max\_tpl(H, Hs, M2), maxof2(N, M2, M)$.

Χρησιμοποιώντας τη (D3) διπλώνουμε τη (17) παίρνοντας

(20) $max\_tpl(leaf(N), [H|Hs], M) \leftarrow max\_tpl\_acc(H, Hs, N, M)$.

Αναζητούμε και εδώ έναν αναδρομικό ορισμό για τη max_tpl_acc. Ετσι, ξεδιπλώνουμε τη (D3) στη max_tpl με τις προτάσεις {16-19}. Παίρνουμε έτσι

(21) $max\_tpl\_acc(leaf(N1), [\,], N, M) \leftarrow maxof2(N, N1, M)$.
(22) $max\_tpl\_acc(leaf(N1), [H|Hs], N, M) \leftarrow max\_tpl(H, Hs, M22), maxof2(N1, M22, M2)$,
$maxof2(N, M2, M)$.
(23) $max\_tpl\_acc(t(LT, RT), [\,], N, M) \leftarrow max\_tpl(LT, [RT], M2), maxof2(N, M2, M)$.

(24) $max\_tpl\_acc(t(LT,RT),[H1|H1s],N,M) \leftarrow max\_tpl(LT,[RT,H1|H1s],M2),$
$maxof2(N,M2,M).$

Αξιοποιώντας την προσεταιριστικότητα της $maxof2$ και αναδιατάζοντας τους στόχους, η (22) γίνεται

(25) $max\_tpl\_acc(leaf(N1),[H|Hs],N,M) \leftarrow maxof2(N,N1,W),max\_tpl(H,Hs,M22),$
$maxof2(W,M22,M).$

Τώρα διπλώνουμε την (25) χρησιμοποιώντας τη (D3). Ετσι παίρνουμε (26)

(26) $max\_tpl\_acc(leaf(N1),[H|Hs],N,M) \leftarrow maxof2(N,N1,W),max\_tpl\_acc(H,Hs,W,M).$

Διπλώνοντας την (23) με τη (D3) παίρνουμε

(27) $max\_tpl\_acc(t(LT,RT),[\,],N,M) \leftarrow max\_tpl\_acc(LT,[RT],N,M).$

Διπλώνοντας την (24) με τη (D3) παίρνουμε

(28) $max\_tpl\_acc(t(LT,RT),[H1|H1s],N,M) \leftarrow max\_tpl\_acc(LT,[RT,H1|H1s],N,M).$

Μαζεύοντας όλες τις προτάσεις παίρνουμε το ακόλουθο πρόγραμμα για το $maxtree$.

(29) $maxtree(T,M) \leftarrow max\_tpl(T,[\,],M).$
(30) $max\_tpl(leaf(N),[\,],N).$
(31) $max\_tpl(leaf(N),[H|Hs],M) \leftarrow max\_tpl\_acc(H,Hs,N,M).$
(32) $max\_tpl(t(LT,RT),[\,],M) \leftarrow max\_tpl(LT,[RT],M).$
(33) $max\_tpl(t(LT,RT),[H|Hs],M) \leftarrow max\_tpl(LT,[RT,H|Hs],M).$
(34) $max\_tpl\_acc(leaf(N1),[\,],N,M) \leftarrow maxof2(N,N1,M).$
(35) $max\_tpl\_acc(leaf(N1),[H|Hs],N,M) \leftarrow maxof2(N,N1,W),max\_tpl\_acc(H,Hs,W,M).$
(36) $max\_tpl\_acc(t(LT,RT),[\,],N,M) \leftarrow max\_tpl\_acc(LT,[RT],N,M).$
(37) $max\_tpl\_acc(t(LT,RT),[H1|H1s],N,M) \leftarrow max\_tpl\_acc(LT,[RT,H1|H1s],N,M).$

Η διαδικασία μετασχηματισμού του παραδείγματος αυτού μπορεί να γενικεύτει και να διατυπωθεί σαν στρατηγική μετασχηματισμού. Αυτό όμως δεν θα μας απασχολήσει άλλο λόγω έλειψης χώρου.

VIII. ΣΥΜΠΕΡΑΣΜΑΤΑ

Στόχος της εργασίας αυτής είναι η διερεύνηση της χρησιμότητας της εισαγωγής αναδρομικών νέων ορισμών και η ανάπτυξη στρατηγικών εφαρμογής των μετασχηματισμών διπλώνω/ξεδιπλώνω. Ετσι, παρουσιάζουμε δύο διαφορετικές μορφές αναδρομικών νέων ορισμών κατάλληλες για μετασχηματισμό μη γραμμικά αναδρομικών διαδικασιών οι οποίες συνδιάζουν τα αποτελέσματα των αναδρομικών κλήσεων τους μέσω προσεταιριστικών διαδικασιών. Ακόμη παρουσιάσαμε στρατηγικές για την εφαρμογή των μετασχηματισμών καθώς και τη χρήση μη αναδρομικών νέων ορισμών για τη μετατροπή των διαδικασιών σε αναδρομικές ουράς. Οπως είναι γνωστό, τα προγράμματα που παρουσιάζουν αναδρομή ουράς εχουν μειωμένες απαιτήσεις σε μνήμη. Η μετατροπή των μη γραμμικών διαδικασιών σε ισοδύναμες γραμμικές καθιστά ευκολώτερη την παραπέρα διαδικασία μετασχηματισμών. Οι στρατηγικές εφαρμογής των μετασχηματισμών που προτείνουμε οδηγούν σε ισοδύναμα προγράμματα αφού όπως μπορεί να αποδειχθεί τηρούνται οι προυποθέσεις που τέθηκαν στην εργασία των Tamaki & Sato [26].

Το δίπλωμα με αναδρομικές προτάσεις αναφέρεται αρχικά στην εργασία [26], χωρίς όμως να εξετάζονται πιθανές μορφές νέων αναδρομικών ορισμών ή να δίνονται στρατηγικές για την εφαρμογή των μετασχηματισμών. Γενικά, η σημασία των αναδρομικών νέων ορισμών δεν έχει επαρκώς εκτιμηθεί γεγονός που αποδεικνύεται από το ότι στη συντριπτική πλειοψηφία των εργασιών [27,2,17,4,5,9,24,19,20,18] που αναφέρονται σε μετασχηματισμούς διπλώνω/ξεδιπλώνω επιτρέπονται μόνο μη αναδρομικοί νέοι ορισμοί που αποτελούνται από μια μόνο πρόταση για κάθε νέο κατηγόρημα. Η μοναδική εργασία που γνωρίζουμε η οποία (έμμεσα) εισάγει αναδρομικούς νέους ορισμούς είναι η [21] που αναφέρεται στη μετατροπή λογικών προγραμμάτων σε *μορφή περάσματος της συνέχειας* (Continuation Passing Style).

Για την εφαρμογή των μετασχηματισμών σημαντικές αποδειχθηκαν οι ιδιότητες των κατηγορημάτων όπως είναι η ύπαρξη *μοναδιαίων στοιχείων* και η *προσεταιριστικότητα*. Η χρήση της προσεταιριστικότητας σε μετασχηματισμούς προγραμμάτων αναφέρεται και σε άλλες εργασίες [4].

Κοινό χαρακτηριστικό των ορισμών που προτείνουμε αποτελεί το γεγονός ότι γενικέυουν διαδικασίες που ήδη ορίζονται στο προγραμμα. Εχουμε συζητήσει δύο είδη γενίκευσης. Το πρώτο αφορά την εισαγωγή επιπλέον ορισμάτων που παίζουν το ρόλο συσσωρευτή και αποτελεί ένα είδος *υπολογιστικής γενίκευσης* (computational generalization) [10], ενώ το δέυτερο τη γενίκευση μιας διαδικασίας που χειρίζεται όρους, έτσι ώστε να χειρίζεται λίστα όρων. Η γενίκευση αυτή ονομάζεται *γενίκευση λίστας* (tupling generalization) [31] και αποτελεί ένα είδος *δομικής γενίκευσης* (structural generalization) [10].

Η έννοια της γενίκευσης δεν είναι καινούρια. Μέθοδοι γενίκευσης χρησιμοποιούνται στα μαθηματικά για την απόδειξη θεωρημάτων. Στην επιστήμη των υπολογιστών μεθόδοι γενίκευσης αναφέρονται στη σύνθεση προγραμμάτων από παραδείγματα [25], στη μηχανική εκμάθηση (machine learning), την αυτόματη απόδειξη θεωρημάτων (automatic theorem proving) [3], τη σύνθεση προγραμμάτων από προδιαγραφές [10] και την επαλήθευση προγραμμάτων (program verification) [23].

IX. ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] J. Arsac and Y. Kordatoff. Some techniques for recursion removal from recursive functions. *ACM Transactions on Programming Languages and Systems*, 4(2):295–322, 1982.

[2] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[3] R. S. Boyer and J. S. Moore. Proving theorems about lisp functions. *J.ACM*, 22(1):129–144, Jan. 1975.

[4] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *J. Logic Programming*, 4(4):345–359, Dec. 1987.

[5] M. Bruynooghe, L. De Raedt, and D. De Screye. Explanation based program transformation. In *International Joint Conference on Artificial Intelligence*, pages 407–412, 1989.

[6] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *J. Logic Programming*, 135–162, 1989.

[7] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J.ACM*, 24(1):44–67, Jan. 1977.

[8] K. L. Clark and F. G. McCabe. The control facilities of ic-prolog. In D. Michie, editor, *Expert Systems in Microelectronic Age*, pages 122–149, Edinburg University Press, 1979.

[9] S. K. Debray. Unfold/fold transformations and loop optimization logic programs. In *SIGPLAN' 88 Conference on Programming Language Design and Implementation*, pages 297–307, June 1988.

[10] Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley Publishing Company,Inc, 1990.

[11] H. Gallaire and C. Lasserre. Metalevel control of logic programs. In K. L. Clark and S. -A. Tarnlund, editors, *Logic Programming*, pages 173–185, Academic Press, 1982.

[12] A. Hansson and S-A. Tarnlund. Program transformation by data structure mapping. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*, pages 117–122, Academic Press, 1982.

[13] T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Fourth International Conf. on Logic Programming*, pages 744–768, 1987.

[14] K. K. Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order specifications. In *7th International Conf. on Logic Programming*, pages 667–684, 1990.

[15] J. W. Lloyd. *Fountations of Logic Programming.* Springer-Verlag, 1984.

[16] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Logic Programming*, 11(3 & 4):217–242, Oct./Nov. 1991.

[17] H. Nakagawa. Prolog program transformations and tree manipulation algorithms. *J. Logic Programming*, 2:77–91, 1985.

[18] A. Pettorossi and M. Proietti. Decidability results and characterization of strategies for the development of logic programs. In *Sixth International Conf. on Logic Programming*, pages 539–553, 1989.

[19] M. Proietti and A. Pettorossi. *Techniques for the automatic improvement of logic programs.* Report R-231, Instituto di Analisi dei Sistemi ed Informatica, Rome, 1988.

[20] M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order for avoiding unnecessary variables in logic programs. In N. Jones, editor, *LNCS no. 528, Proc. PLILP' 91*, pages 347–358, Springer-Verlag, 1991.

[21] T. Sato and H. Tamaki. Existential continuation. *New Generation Computing*, 6:421–438, 1989.

[22] T. Sato and H. Tamaki. Transformational logic program synthesis. In *International Conference of Fifth Generation Computer Systems*, pages 195–201, 1984.

[23] H. Seki. Incorporating generalization heuristics into verification of prolog programs. In *International Joint Conference on Artificial Intelligence*, pages 737–741, 1985.

[24] H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *4th IEEE Symposium on Logic Programming*, pages 215–223, 1987.

[25] P. D. Summers. A methodology for lisp program construction from examples. *J.ACM*, 24(1):161–175, Jan. 1977.

[26] H. Tamaki and T. Sato. *A generalized correctness proof of the unfold/fold logic program transformation.* Technical Report No 86-4, Dept. of Information Science Ibaraki University, Japan, 1986.

[27] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Second International Conference on Logic Programming*, pages 127–138, 1984.

[28] M. Wand. Continuation-based program transformation strategies. *J.ACM*, 27(1):164–180, Jan. 1980.

[29] Μ. Κατζουράκη, Μ. Γεργατσούλης, Σ. Κόκκοτος "*PROγραμματίζοντας στη LOGική και εφαρμογές στην Τεχνητή Νοημοσύνη.*" Εκδόσεις ΕΠΥ 1991.

[30] Μ. Γεργατσούλης, Μ.Κατζουράκη "*Δ-Prolog: Μια υλοποίηση της Prolog.*" Πρακτικά 2ου Πανελλήνιου Συνεδρίου Πληροφορικής Τόμος 2 σελ. 418-426 Θεσσαλονίκη 1988.

[31] M. Gergatsoulis, M. Katzouraki "*A transformation for tupling generalization of Logic Programs*" Canadian Conference on Electrical and Computer Engineering (CCECE-92) 1992.