Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Μερική Αποτίμηση Προγραμμάτων και Εφαρμογή στην Κατασκευή Μεταγλωττιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ Κ. ΠΡΟΚΟΠΙΟΥ

Επιβλέπων : Νικόλαος Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Δεκέμβριος 2006

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Μερική Αποτίμηση Προγραμμάτων και Εφαρμογή στην Κατασκευή Μεταγλωττιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ Κ. ΠΡΟΚΟΠΙΟΥ

Επιβλέπων : Νικόλαος Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Δεκεμβρίου 2006.

Νικόλαος Παπασπύρου	Ευστάθιος Ζάχος	Κωνσταντίνος Σαγώνας
Επίκ. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Δεκέμβριος 2006

Παναγιώτης Κ. Προκοπίου

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η μερική αποτίμηση προσφέρει έναν κοινό τρόπο αντιμετώπισης και επίλυσης σε ένα ευρύ φάσμα προβλημάτων, που περιλαμβάνει την βελτιστοποίηση, μεταγλώττιση και διερμήνευση των προγραμμάτων και την αυτόματη παραγωγή γεννητόρων προγραμμάτων. Είναι μια τεχνική βελτιστοποίησης προγραμμάτων, που μπορεί να χαρακτηριστεί και ως εξειδίκευση και δίνει έμφαση στην πλήρως αυτοματοποιημένη παραγωγή αποτελεσμάτων. Μέσω αυτής της τεχνικής μπορούμε να απαλείφουμε περιττούς υπολογισμούς από τα προγράμματά μας και να επιτυγχάνουμε μείωση της πολυπλοκότητας.

Ιδιαίτερη έμφαση στην έρευνα της μερικής αποτίμησης έχει δοθεί στην αυτόματη παραγωγή μεταγλωττιστών. Η τεχνική αυτή προβάλλεται ως εναλλακτική της κλασσικής αντιμετώπισης για την κατασκευή μεταγλωττιστών που ξεχωρίζει σε μεγάλο βαθμό την υλοποίηση του μεταγλωττιστή από την σημασιολογία της γλώσσας που υλοποιεί. Έχοντας στη διάθεσή μας μόνο έναν διερμηνέα που τον χρησιμοποιούμε προκειμένου να ορίσουμε μια γλώσσα προγραμματισμού, μέσω της μερικής αποτίμησης μπορούμε να τον μετατρέψουμε σε μεταγλωττιστή ανάλογων επιδόσεων με αυτούς που κατασκευάζονται κλασσικά.

Δεν είναι όμως μόνο η τεχνολογία των μεταγλωττιστών που έχει οφέλη από την χρήση των τεχνικών της μερικής αποτίμησης. Πολλοί τομείς της επιστήμης των υπολογιστών, ειδικότερα όσοι ασχολούνται με θέματα μεγάλης υπολογιστής πολυπλοκότητας ευνοούνται από την εξειδίκευση γενικά ορισμένων αλγορίθμων σε σχέση με τα συγκεκριμένα χαρακτηριστικά και δεδομένα του εκάστοτε προβλήματος. Σαν παράδειγμα αναφέρουμε το λογικό προγραμματισμό, το μεταπρογραμματισμό, τα έμπειρα συστήματα και τα εφαρμοσμένα μαθηματικά.

Λέξεις κλειδιά

Μερική αποτίμηση, εξειδίκευση, μεταγλωττιστής, διερμηνέας, αυτοεφαρμογή, αυτοδιερμηνέας, προβολές Futamura.

Abstract

Partial evaluation provides a common way to deal with and solve a wide spectrum of problems, including optimization, compilation and interpretation of computer programs and the automatic production of program generators. It is program optimization technique, also known as specialization, emphasizing on the totally automatic production of results. Through this technique we can remove the overhead from unnecessary computations in our programs and achieve a reduction in complexity.

Researchers have applied partial evaluation for automatic compiler generation. This technique can be used as an alternative to classic compiler generation, which typically separates the implementation of the compiler from the semantics of the programming language that it implements. Partial evaluation can transform an interpreter for a programming language, used to define the semantics of the language, to a compiler whose performance is similar to those built with the classical approach.

However, it is not only compiler technology that has to gain from partial evaluation. Many fields of computer science, especially those involving computations of significant complexity, take advantage of specializing algorithms with respect to the specific characteristics and data of a given problem. Examples of such scientific fields are logic programming, metaprogramming, expert systems and applied mathematics.

Key words

Partial Evaluation, specialization, compiler, interpreter, self application, self interpreter, Futamura projections.

Ευχαριστίες

Κατ' αρχήν θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου, κ. Νίκο Παπασπύρου για το ενδιαφέρον που έδειξε και για τον ατέλειωτο χρόνο που αφιέρωσε σε αυτή την εργασία. Θέλω επίσης να ευχαριστήσω τους γονείς που μου εξασφάλισαν τη δυνατότητα να διεκπεραιώσω απερίσπαστος τις σπουδές μου. Θέλω να ευχαριστήσω τα μέλη του Εργαστηρίου Τεχνολογίας Λογισμικού για τη συνεργασία που είχαμε κατά διάρκεια αυτού του χρόνου. Τέλος οφείλω ένα μεγάλο ευχαριστώ στου φίλους μου που με στήριξαν και με ανέχθηκαν όσο καιρό δούλευα την διπλωματική μου.

Παναγιώτης Κ. Προκοπίου,

Αθήνα, 18η Δεκεμβρίου 2006.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-06, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Δεκέμβριος 2006.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Σχήματα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός αυτής της διπλωματικής εργασίας είναι η μελέτη της μερικής αποτίμησης και των εφαρμογών της, ιδιαίτερα στην κατασκευή μεταγλωττιστών. Ο όρος μερική αποτίμηση αναφέρεται στην εκτέλεση υπολογισμών χωρίς να είναι γνωστά όλα τα δεδομένα που χρειαζόμαστε για να φτάσουμε σε κάποιο αποτέλεσμα. Στον τομέα της πληροφορικής αυτό μπορεί να σημαίνει ότι παίρνουμε ένα πρόγραμμα και κάποιες παραμέτρους του και κατασκευάζουμε ένα άλλο πρόγραμμα που δεν χρειάζεται πλέον αυτές τις παραμέτρους γιατί οι υπολογισμοί που εξαρτώνται από αυτές έχουν εξαλειφθεί

Όπως θα δούμε και στην συνέχεια η μερική αποτίμηση παρουσιάζει μεγάλο ενδιαφέρον όταν το πρόγραμμα αυτό είναι ένας διερμηνέας και οι παράμετροι που του δίνουμε είναι ένα πρόγραμμα προς εκτέλεση. Αυτό που θα πάρουμε στο τέλος θα είναι ο αρχικός διερμηνέας μόνο που τώρα δεν θα χρειάζεται να του δίνουμε και το πρόγραμμα σαν παράμετρο, παρά μόνο τις μεταβλητές του προγράμματος. Μέσω της μερικής αποτίμησης δηλαδή καταφέρνουμε να μετατρέψουμε ένα διερμηνέα σε μεταγλωττιστή και στόχος μας είναι να κάνουμε αυτή τη μεταβολή όσο πιο σωστά γίνεται.

Παρ' όλα αυτά η διαδικασία της μερικής αποτίμησης δεν μπορεί να εφαρμόζεται πάντα ούτε πρέπει να εφαρμόζεται όπου μπορεί. Χρειαζόμαστε κάποια κριτήρια για να πάρουμε τις αποφάσεις μας κι αυτά κυρίως έχουν να κάνουν με το αν τελικά το αποτέλεσμα της μερικής αποτίμησης είναι ένα πρόγραμμα που εκτελείται πιο γρήγορα. Επειδή όπως είναι λογικό αυτό δεν μπορούμε να το ξέρουμε εκ των προτέρων, αναπτύσσουμε κάποια προσεγγιστική διαδικασία βάση των αποτελεσμάτων της οποίας παίρνουμε τις αποφάσεις μας.

Οι εφαρμογές της μερικής αποτίμησης όμως δεν αναλώνονται στην παραγωγή μεταγλωττιστών. Κάθε πρόγραμμα είναι εν δυνάμει υποψήφιο για μερική αποτίμηση αρκεί να έχουμε τα κατάλληλα εργαλεία. Πολλοί τομείς των επιστημών έχουν ωφεληθεί από την χρήση της μερικής αποτίμησης, αλλά τα αποτελέσματα είναι πιο εμφανή σε όσους ασχολούνται με μεγάλους και χρονοβόρους υπολογισμούς.

1.2 Σύνοψη

Στο κεφάλαιο 2 θα ασχοληθούμε με το θεωρητικό υπόβαθρο της μερικής αποτίμησης. Θα παρουσιάσουμε βασικές έννοιες της επιστήμης των υπολογιστών και σταδιακά θα φτάσουμε στον ορισμό της μερικής αποτίμησης. Σημαντικό κομμάτι της μερικής αποτίμησης είναι και οι προβολές Futamura, με τις οποίες κάνουμε και την πρώτη αναφορά στην μετατροπή ενός διερμηνέα σε μεταγλωττιστή. Θα ολοκληρώσουμε το κεφάλαιο θέτοντας τις απαιτήσεις μας από την μερική αποτίμηση και αναζητώντας τα γιατί, πως και πότε αυτής της διαδικασίας.

Στο κεφάλαιο 3 αναλύουμε την εφαρμογή της μερικής αποτίμησης στον κώδικα ενός διερμηνέα με απώτερο στόχο να παράγουμε ένα μεταγλωττιστή. Αντιμετωπίζουμε τα προβλήματα που προκύπτουν και εισάγουμε απαραίτητες συνοδευτικές έννοιες όπως είναι η διαίρεση και η ανάλυση της στιγμής δέσμευσης. Ασχολούμαστε επίσης και με διάφορες τεχνικές βελτίωσης του

παραγόμενου αποτελέσματος. Στο τέλος του κεφαλαίου κάνουμε μια επίδειξη των δυνατοτήτων της μερικής αποτίμησης με τη βοήθεια ενός αλγόριθμου ταιριάσματος συμβολοακολουθιών.

Στο κεφάλαιο 4 θέτουμε τις βάσεις για την ανάλυση στο πεδίο του χρόνου της μερικής αποτίμησης. Παρουσιάζουμε τον ορισμό της επιτάχυνσης και το πως αυτή μετράται. Η μέτρηση όμως δεν είναι εφικτή πριν την εξειδίκευση και για αυτό χρειαζόμαστε και κάποιο τρόπο πρόβλεψης της επιτάχυνσης. Αυτή εκφράζεται μέσω των ασφαλών διαστημάτων επιτάχυνσης. Κλείνουμε το κεφάλαιο αφού πρώτα καλύψουμε όλες τις διαφορετικές περιπτώσεις επιτάχυνσης.

Στο κεφάλαιο 5 κάνουμε για πρώτη φορά την διάκριση ανάμεσα σε online και offline μερική αποτίμηση. Παρουσιάζουμε τις διαφορές και τα πλεονεκτήματα της κάθε αντιμετώπισης και δείχνουμε πως αυτές συγκλίνουν σε υβριδικές τεχνικές. Έχοντας σαν στόχο μας την μεταγλώττιση μέσω μερικής αποτίμησης, τροποποιούμε κατάλληλα την online τεχνική για να μας δίνει αποδεκτά αποτελέσματα. Τέλος παρουσιάζουμε μια δοκιμασμένη συνταγή για να γράψουμε τον δικό μας μερικό αποτιμητή

Στο κεφάλαιο 6 παρουσιάζουμε εφαρμογές τις μερικής αποτίμησης σε διάφορες κατηγορίες προβλημάτων. Ιδιαίτερη έμφαση δίνουμε στους υποψιασμένους, τους ανυποψίαστους και τους ασθενώς ανυποψίαστους αλγόριθμους.

Κεφάλαιο 2

Μερική αποτίμηση

Σε αυτό το κεφάλαιο θα ασχοληθούμε γενικά με την θεωρία της μερικής αποτίμησης. Θα ξεκινήσουμε ορίζοντας βασικές έννοιες της επιστήμης των υπολογιστών και χρησιμοποιόντας αυτές θα παρουσιάσουμε γιατί και πότε είναι συμαντική η τεχνική της μερικής αποτίμησης.

2.1 Ορισμοί

2.1.1 Γλώσσα

Με τον όρο γλώσσα προγραμματισμού εννοούμε μια τεχνητή γλώσσα που χρησιμοποιείτε για να ελέγχει την συμπεριφορά μιας μηχανή, και πιο συγκεκριμένα συνήθως ενός ηλεκτρονικού υπολογιστή. Όπως και οι φυσικές γλώσσες, κάθε γλώσσα προγραμματισμού έχει κάποιους συντακτικούς, αυστηρά ορισμένους, και κάποιους σημασιολογικούς κανόνες για να καθορίζουν την δομή και την λειτουργία της αντίστοιχα.

Οι γλώσσες προγραμματισμού χρησιμοποιούνται να να οργανώσουν και να χειριστούν κάποια πληροφορία καθώς και για να περιγράψουν με ακρίβεια κάποιον αλγόριθμο. Υπάρχουν γλώσσες που είναι σε θέση να περιγράψουν κάθε πιθανό αλγόριθμο, ενώ άλλες δεν παρέχουν τόση μεγάλη ευελιξία. Με τον όρο γλώσσες προγραμματισμού συνήθως αναφερόμαστε στην πρώτη κατηγορία.

2.1.2 Πρόγραμμα

Ως πρόγραμμα ορίζουμε ένα κείμενο γραμμένο βάση των συντακτικών κανόνων που ορίζει μια γλώσσα προγραμματισμού X. Λέμε μάλιστα ότι αυτό είναι ένα πρόγραμμα γραμμένο στη X γλώσσα. Ένα πρόγραμμα είναι μια συλλογή από εντολές που περιγράφουν μια διαδικασία ή ένα σύνολο διαδικασιών που θα πρέπει να επιτελέσει ο υπολογιστής. Όταν λέμε ότι εκτελούμε το πρόγραμμα εννοούμε ότι ο υπολογιστής εκτελεί τις εντολές που αναφέρονται στο πρόγραμμα βάση της σημασιολογίας της γλώσσας προγραμματισμού για την οποία έχει γραφτεί αυτό.

Θα χρησιμοποιούμε των όρο κώδικας του προγράμματος για να αναφερόμαστε στο κείμενο του προγράμματος, είσοδος του προγράμματος για να αναφερόμαστε σε παραμέτρους που δίνουμε στο πρόγραμμα τη στιγμή που πάμε να το εκτελέσουμε, και έξοδος για να αναφερόμαστε στο αποτέλεσμα της εκτέλεσης του προγράμματος.

Τα προγράμματα συνήθως τα γράφουν άνθρωποι, υπάρχει όμως και μια ειδική κατηγορία προγραμμάτων που η έξοδός τους είναι άλλα προγράμματα. Λέμε ότι τέτοια προγράμματα παράγουν προγράμματα.

Έστω το πρόγραμμα p γραμμένο στην γλώσσα S. Λέμε τότε ότι το p είναι ένα S-πρόγραμμα και ότι η είσοδος και η έξοδος του p είναι τύπου $S-$δεδομένα. Με τον συμβολισμό $[\![p]\!]_S$ εννοούμε τη σημασιολογία του p με βάση τον ορισμό της γλώσσας S. Μπορούμε να θεωρήσουμε ότι το $[\![p]\!]_S$ είναι μια συνάρτηση τύπου $S-data \rightarrow S-data$. Συνολικά:

$$[\![p]\!]_S : S-data \rightarrow S-data$$
και
$$output = [\![p]\!]_S(input)$$

Να επισημάνουμε ότι έχουμε κάνει την παραδοχή ότι $S{-}data \times S{-}data \subseteq S{-}data$ κάτι που γενικά ισχύει, οπότε μπορούμε να θεωρούμε ότι η παράμετρος εισόδου είναι μόνο μία και να γράφουμε τις σχέσεις μας κατά αυτό τον τρόπο, αν δεν μας απασχολεί να φαίνεται το πλήθος τους. Επίσης τυπικά το $[\![p]\!]_S$ δεν είναι συνάρτηση, αλλά μερική συνάρτηση τύπου $S{-}data \to S{-}data_\perp$ γιατί είναι δυνατό κάποιο πρόγραμμα να μην τερματίζει την εκτέλεσή του, αλλά στη συνέχεια θα το αγνοήσουμε καθώς πάντοτε θα αναφερόμαστε σε προγράμματα που τερματίζουν.

2.1.3 Διερμηνέας (Interpreter)

Ο διερμηνέας είναι ένα πρόγραμμα που εκτελεί άλλα προγράμματα. Αυτό σημαίνει πως σαν παραμέτρους εισόδου έχει ένα πρόγραμμα p γραμμένο στην γλώσσα S και τις παραμέτρους του προγράμματος p, και στην έξοδό του δίνει το αποτέλεσμα της εκτέλεσης του προγράμματος p με τις ίδιες παραμέτρους εισόδου υπό τη σημασιολογία της γλώσσας S. Ο διερμηνέας int ως πρόγραμμα είναι κι αυτός γραμμένος σε κάποια γλώσσα L. Αντιμετωπίζοντας τον διερμηνέα σαν ένα $L{-}$πρόγραμμα πρέπει να ισχύει ότι

$$[\![int]\!]_L : L{-}data \to L{-}data$$

και λαμβάνοντας υπόψιν μας τις ιδιαιτερότητές του σημασιολογικά, πρέπει να ισχύουν και τα παρακάτω.

$$S{-}prog \times S{-}data \subseteq L{-}data$$
$$\text{και } S{-}prog \subseteq L{-}data$$

Οπότε με ένα πιο αυστηρό ορισμό προκύπτει

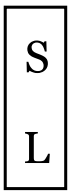
$$[\![int]\!]_L : (S{-}prog \times S{-}data) \to S{-}prog$$

Μια γλώσσα για την οποία ισχύει ότι $S{-}prog \times S{-}data \subseteq S{-}data$ θα λέμε πως έχει συμπαγή (concrete) σύνταξη, ενώ αν $S{-}data \times S{-}data \subseteq S{-}data$ θα λέμε πως η S έχει ζεύγη (pairing).

Για να δηλώσουμε το σύνολο των διερμηνέων int για την γλώσσα S που είναι γραμμένοι στην γλώσσα L, δηλαδή το

$$\{int \mid \forall p, d. [\![p]\!]_S(d) = [\![int]\!]_L(p, d)\}$$

χρησιμοποιούμε τον εξής συμβολισμό:



Σχήμα 2.1: Διερμηνέας

2.1.4 Μεταγλωττιστής (Compiler)

Ο μεταγλωττιστής είναι ένα πρόγραμμα που διαβάζει από την είσοδο ένα αρχικό πρόγραμμα p γραμμένο σε μια αρχική γλώσσα S, και το μεταφράζει ώστε να παράγει στην έξοδό του ένα τελικό πρόγραμμα p' γραμμένο σε μια γλώσσα T σημασιολογικά ταυτόσημο με το αρχικό. Η βασική χρησιμότητα των μεταγλωττιστών είναι να μεταφράζουν ένα πρόγραμμα που έχει γράψει και αντιλαμβάνεται τη σημασιολογία του κάποιος άνθρωπος, σε ένα ισοδύναμο πρόγραμμα

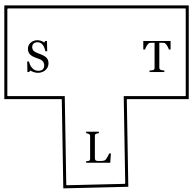
γραμμένο σε γλώσσα μηχανής, την μόνη γλώσσα που καταλαβαίνουν οι υπολογιστές η οποία όμως είναι δύσχρηστη για να χρησιμοποιηθεί ως αρχική γλώσσα. Ο μεταγλωττιστής είναι κι ο ίδιος γραμμένος σε κάποια γλώσσα L, είναι δηλαδή ένα $L-$πρόγραμμα, που στην είσοδό του δέχεται ένα $S-$πρόγραμμα και η έξοδός του είναι ένα $T-$πρόγραμμα. Έτσι ισχύουν τα παρακάτω:

$$\begin{aligned} [\![comp]\!]_L : L{-}data \to L{-}data & \quad comp \text{ as } L{-}prog \\ S{-}prog \subseteq L{-}data & \quad S{-}prog \text{ as } comp\text{'s input} \\ T{-}prog \subseteq L{-}data & \quad T{-}prog \text{ as } comp\text{'s output} \\ [\![comp]\!]_L : S{-}prog \to T{-}prog & \quad \text{more strict definition} \\ output = [\![[\![comp]\!]_L(p)]\!]_T(input) = [\![p]\!]_S(input) & \quad \text{συμπέρασμα} \end{aligned}$$

Για να δηλώσουμε το σύνολο των μεταγλωττιστών comp από την γλώσσα S στη γλώσσα T που είναι γραμμένοι στην γλώσσα L, δηλαδή το

$$\{comp \mid \forall p \in S{-}prog, \forall d \in S{-}data. [\![p]\!]_S(d) = [\![[\![comp]\!]_L(p)]\!]_L(d)\}$$

χρησιμοποιούμε τον εξής συμβολισμό:



Σχήμα 2.2: Μεταγλωττιστής

2.1.5 Γεννήτορας Μεταγλωττιστών

Ο γεννήτορας μεταγλωττιστών είναι ένα πρόγραμμα που στην είσοδό του τού δίνουμε την περιγραφή μιας γλώσσας προγραμματισμού και στην έξοδό του μας δίνει έναν μεταγλωττιστή για αυτή τη γλώσσα. Στην ιδανική περίπτωση, ένας γεννήτορας μεταγλωττιστών θα έπαιρνε στην είσοδό του το συντακτικό και την σημασιολογία της αρχικής γλώσσας S και το συντακτικό και της σημασιολογία της τελικής γλώσσας T, και θα παρήγαγε με τελείως αυτοματοποιημένο τρόπο έναν μεταγλωττιστή από τη γλώσσα S στη γλώσσα T γραμμένο στη γλώσσα L. Δεν έχουμε φτάσει όμως πολύ κοντά σε αυτό το επίπεδο, έτσι στην πράξη αυτό που λέμε γεννήτορα μεταγλωττιστών τις περισσότερες φορές είναι ένα πρόγραμμα που διαβάζει τον ορισμό της σύνταξης της αρχικής γλώσσας σε μια τυποποιημένη μορφή και παράγει το τμήμα του μεταγλωττιστή σε γλώσσα L που ασχολείται με την συντακτική ανάλυση. Η σημασιολογική ανάλυση και η αντιστοίχηση της στη σημασιολογία και τη σύνταξη της τελικής γλώσσας εξακολουθεί να γίνεται χειροκίνητα.

2.1.6 Αυτοεφαρμογή (Self application)

Με τον όρο αυτοεφαρμογή αναφερόμαστε στην διαδικασία κατά την οποία ένα πρόγραμμα εκτελείται λαμβάνοντας σαν είσοδο το ίδιο το πρόγραμμα. Είναι κατανοητό ότι η γλώσσα στην οποία γράφεται ένα πρόγραμμα που προορίζεται για αυτοεφαρμογή πρέπει να είναι ίδια με την γλώσσα εισόδου.

Χαρακτηριστικό παράδειγμα αυτοεφαρμογής είναι ο αυτοδιερμηνέας (self interpreter) που είναι ένας διερμηνέας γραμμένος στην γλώσσα που ορίζει. Σε αυτή την περίπτωση έχουμε ότι $sint' = [\![sint]\!]_S(sint)$ κάτι που σίγουρα μας κάνει εντύπωση την πρώτη φορά που το συναντάμε. Να σημειώσουμε ότι λειτουργικά τα $sint$ και $sint'$ είναι ισότιμα. Με τη χρήση ενός αυτοδιερμηνέα είναι εξαιρετικά απλό να ορίσουμε τη σημασιολογία μιας γλώσσας, ενώ ακολουθώντας

μια συγκεκριμένη μεθοδολογία μπορούμε σταδιακά να επεκτείνουμε τη λειτουργικότητα του αρχικού διερμηνέα, και άρα τις δυνατότητες της γλώσσας μας.

Το τελευταίο που αναφέραμε βρίσκει εφαρμογή κυρίως στην κατασκευή των μεταγλωττιστών. Υποθέτοντας πως έχουμε τον αρχικό κώδικα ενός μεταγλωττιστή γραμμένου στην γλώσσα που ορίζει και του εκτελέσιμου προγράμματος που προκύπτει από τον κώδικα, μπορούμε σταδιακά να προσθέτουμε λειτουργικότητα στον μεταγλωττιστή και καινούρια χαρακτηριστικά στη γλώσσα μας.

2.2 Ορισμός της μερικής αποτίμησης

Στο προηγούμενο κεφάλαιο είδαμε ότι τα προγράμματα και τους διερμηνείς μπορούμε να αντιληφθούμε σαν συναρτήσεις πολλών μεταβλητών. Για να μπορέσουμε να υπολογίσουμε την τιμή που μας επιστρέφουν λοιπόν θεωρούμε λογική απαίτηση να ξέρουμε τις τιμές αυτών των μεταβλητών πριν αρχίσουμε την αποτίμησή τους. Για παράδειγμα ένα πρόγραμμα με δύο μεταβλητές εισόδου πρέπει να γνωρίζει τις τιμές και των δύο πριν εκτελεστεί. Το ερώτημα που προκύπτει είναι στην περίπτωση που δεν έχουμε τιμές για όλες τις μεταβλητές εισόδου, τί μπορούμε να κάνουμε. Θα παρουσιάσουμε ότι ακόμα και στην περίπτωση που δεν έχουμε τα πλήρη αρχικά δεδομένα, για παράδειγμα ξέρουμε μόνο μία από τις δύο αρχικές τιμές των μεταβλητών ενός προγράμματος, μπορούμε να εκτελέσουμε κάποιους υπολογισμούς και να είμαστε σε θέση να ολοκληρώσουμε την υπολογιστική διαδικασία όταν μάθουμε και τα υπόλοιπα δεδομένα.

Η διαδικασία που σκιαγραφήσαμε παραπάνω λέγεται μερική αποτίμηση και με πιο αυστηρό τρόπο ορίζεται ως η μετατροπή του αρχικού προγράμματος σε ένα τελικό, το εναπομένον πρόγραμμα, λαμβάνοντας υπόψιν μας μέρος των μεταβλητών εισόδου του αρχικού προβλήματος. Αναγκαία συνθήκη στην οποία οφείλεται και η αξία της μερικής αποτίμησης είναι ότι όταν το εναπομένον πρόγραμμα πάρει στην είσοδό του τα υπόλοιπα δεδομένα θα δώσει στην έξοδό του ότι και το αρχικό αν εξαρχής του δίναμε τα πλήρη δεδομένα. Δηλαδή

$$output = [\![p]\!](in_1, in_2) = [\![[\![spec]\!](p, in_1)]\!](in_2) = [\![p_{in_1}]\!](in_2)$$

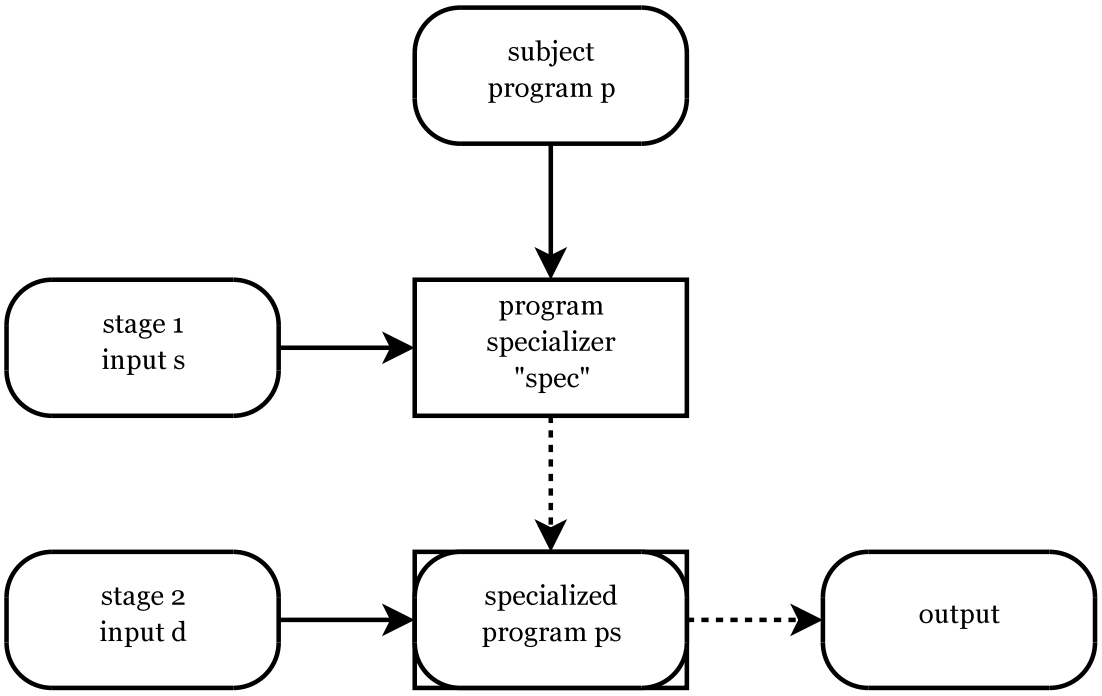
Ορίζουμε τον μερικό αποτιμητή spec, ως ένα πρόγραμμα γραμμένο στη γλώσσα υλοποίησης L που στην είσοδό του παίρνει ένα πρόγραμμα p γραμμένο στην αρχική γλώσσα S με μέρος από τις μεταβλητές εισόδου, και στην έξοδό του δίνει ένα πρόγραμμα γραμμένο στη γλώσσα T. Τότε ισχύουν

$$\begin{array}{rl} [\![spec]\!]_L : L-data \rightarrow L-data & spec \text{ as } L{-}prog \\ S-prog \subseteq L-data & S{-}prog \text{ as } spec\text{'s input} \\ T-prog \subseteq L-data & T{-}prog \text{ as } spec\text{'s output} \\ [\![spec]\!]_L : S-data \rightarrow T-data & \text{more strict definition} \end{array}$$

Στην συνέχεια θα χρησιμοποιούμε ισοδύναμα με τον όρο μερική αποτίμηση τον όρο εξειδίκευση, θα λέμε δηλαδή ότι εξειδικεύουμε το p σε σχέση με το in_1. Η θεωρητική δυνατότητα για μερική αποτίμηση προγραμμάτων για πρώτη φορά έγινε γνωστή από τις μελέτες του Kleene. Ο Kleene κατέληξε σε μια μορφή μερικής αποτίμησης αναζητώντας μεθοδολογίες για να δείξει ποια προβλήματα είναι εφικτό να λυθούν και ποιες υπολογιστικές διεργασίες τερματίζουν σε πεπερασμένο χρόνο.

Σε αυτό το σημείο να τονίσουμε ότι όταν αναφερόμαστε σε μερική αποτίμηση εννοούμε μια τελείως αυτοματοποιημένη διαδικασία μετατροπής του αρχικού προγράμματος στο εναπομένον, χωρίς να μεσολαβεί κανένα στάδιο που να χρειάζεται αλληλεπίδραση με το χρήστη.

Στο παρακάτω σχήμα φαίνεται παραστατικά η ροή δεδομένων κατά τη διαδικασία της μερικής αποτίμησης.



Σχήμα 2.3: Μερική Αποτίμηση

2.3 Προβολές Futamura

2.3.1 Πρώτη προβολή Futamura

Θεωρούμε πως έχουμε στην διάθεσή μας έναν μερικό αποτιμητή *spec* από τη γλώσσα L στη γλώσσα T, έναν διερμηνέα *int* για τη γλώσσα S γραμμένο στη L και ένα αρχικό πρόγραμμα *source* γραμμένο στην S. Μπορούμε τότε με την ακόλουθη διαδικασία να πάρουμε ένα πρόγραμμα target γραμμένο στην T ισοδύναμο με το *source*.

$$target = [\![spec]\!]_{Impl}(int, source)$$

Η απόδειξη για την ορθότητα των υπολογισμών φαίνεται στις ακόλουθες εξισώσεις

$$\begin{array}{rcll} out & = & [\![source]\!]_S(in) & \text{from the definition of program} \\ & = & [\![int]\!]_L(source, in) & \text{from the definition of interpreter} \\ & = & [\![[\![spec]\!]_{Impl}(int, source)]\!]_T(in) & \text{from definition of specializer} \\ & = & [\![target]\!](input) & \text{from definition of target} \end{array}$$

Με άλλα λόγια η πρώτη προβολή του Futamura μας λέει ότι μπορούμε να μεταγλωττίσουμε τα προγράμματα της γλώσσας S στην γλώσσα εξόδου του μερικού αποτιμητή T, αν κατασκευάσουμε ένα διερμηνέα για την S γραμμένο στην γλώσσα εισόδου του μερικού αποτιμητή L.

2.3.2 Δεύτερη προβολή Futamura

Θεωρούμε πως έχουμε στην διάθεσή μας έναν μερικό αποτιμητή *spec* από τη γλώσσα L στη γλώσσα T, γραμμένο στην γλώσσα εισόδου του L και έναν διερμηνέα *int* για τη γλώσσα S γραμμένο στη L. Μπορούμε τότε με την ακόλουθη διαδικασία να πάρουμε έναν μεταγλωττιστή από τη γλώσσα S στη γλώσσα T γραμμένο στη γλώσσα L.

$$compiler = [\![spec]\!]_L(spec, int)$$

Η απόδειξη για την ορθότητα των υπολογισμών φαίνεται στις ακόλουθες εξισώσεις

$$
\begin{array}{rcll}
\text{target} & = & [\![spec]\!]_L(int, source) & \text{from the first Futamura projection} \\
& = & [\![[\![spec]\!]_L(spec, int)]\!](source) & \text{from the definit ion of specializer} \\
& = & [\![compiler]\!]_T(source) & \text{from the definition of compiler}
\end{array}
$$

Αυτό που κάνουμε είναι να εξειδικεύουμε τον μερικό αποτιμητή σε σχέση με τον διερμηνέα, κάτι που λειτουργικά είναι δύσκολο να το αντιληφθούμε γιατί περιέχει την έννοια της αυτοεφαρμογής. Όμως στην πράξη οι μεταγλωττιστές που παράγονται από την δεύτερη προβολή του Futamura έχουν πολύ καλά χαρακτηριστικά

2.3.3 Τρίτη προβολή Futamura

Θεωρούμε πως έχουμε στην διάθεσή μας έναν μερικό αποτιμητή $spec$ από τη γλώσσα L στη γλώσσα T, γραμμένο στην γλώσσα L. Μπορούμε τότε με την ακόλουθη διαδικασία να πάρουμε έναν γεννήτορα μεταγλωττιστών που δεδομένου ενός διερμηνέα της γλώσσας S γραμμένο στη γλώσσα L μας δίνει ένα μεταγλωττιστή από τη γλώσσα S στη γλώσσα L γραμμένο στη γλώσσα L.

$$cogen = [\![spec]\!]_L(spec, spec)$$

Η απόδειξη για την ορθότητα των υπολογισμών φαίνεται στις ακόλουθες εξισώσεις

$$
\begin{array}{rcll}
\text{compiler} & = & [\![spec]\!]_L(spec, int) & \text{from the second Futamura projection} \\
& = & [\![[\![spec]\!]_L(spec, spec)]\!]_T(int) & \text{from the definition of specializer} \\
& = & [\![cogen]\!]_T(int) & \text{from the definition of compiler}
\end{array}
$$

Με βάση τη τρίτη προβολή του Futamura μπορούμε να μετατρέψουμε ένα διερμηνέα της γλώσσας S γραμμένο στην L σε ένα μεταγλωττιστή από τη γλώσσα S στην L γραμμένο στην L.

2.4 Ιδιότητες της μερικής αποτίμησης

2.4.1 Επιθυμητές ιδιότητες

Τρεις είναι ο επιθυμητές ιδιότητες της διαδικασίας της μερικής αποτίμησης, η καθολικότητα, η υπολογιστική πληρότητα, και η βελτιστότητα.

Καθολικότητα

Θέλουμε για κάθε πρόγραμμα p και μέρος των δεδομένων εισόδου του s να ορίζεται το αποτέλεσμα της εξειδίκευσης:

$$p_s = [\![spec]\!](p, s)$$

Υπολογιστική πληρότητα

Θέλουμε κατά την εξειδίκευση του p με βάση το s να γίνονται όσοι υπολογισμοί εξαρτώνται από το s. Στην πράξη όπως θα δούμε σε επόμενα κεφάλαια, θέλουμε να γίνονται όσο το δυνατό περισσότεροι υπολογισμοί εξαρτώνται από το s αλλά ταυτόχρονα να εξασφαλίζεται ότι το στάδιο της μερικής αποτίμησης θα τερματίσει. Δυστυχώς το πρόβλημα της εύρεσης του βέλτιστου συνόλου υπολογισμών που μπορούν να γίνουν είναι άλυτο, έτσι περιοριζόμαστε στο να το προσεγγίζουμε από την ασφαλή πλευρά, δηλαδή από την πλευρά που μας εξασφαλίζει τερματισμό.

Βελτιστότητα

Θέλουμε κατά την μερική αποτίμηση να εξαφανίζουμε όλους του πλεονάζοντες υπολογισμούς που προκύπτουν στο αρχικό πρόγραμμα λόγω διερμηνείας (interpretation overhead). Την απαίτηση αυτή μπορούμε να την αντιληφθούμε πληρέστερα αν πάρουμε για παράδειγμα τον αυτοδιερμηνέα *sint*. Απαιτούμε το αποτέλεσμα της εξειδίκευσης του αυτοδιερμηνέα σε σχέση με τον εαυτό του, να είναι τουλάχιστον το ίδιο αποδοτικό όπως ο αυτοδιερμηνέας. Το πώς ακριβώς μετράμε την βελτίωση των επιδόσεων θα το δούμε σε επόμενο κεφάλαιο.

2.4.2 Γιατί κάνουμε την μερική αποτίμηση

Η μερική αποτίμηση δεν είναι μια απλή διαδικασία οπότε θα πρέπει να έχουμε κάποιο όφελος για να ασχολούμαστε μαζί της. Όπως ήταν λογικό και αναμενόμενο ο λόγος είναι η βελτίωση των επιδόσεων. Τα εξειδικευμένα προγράμματα σίγουρα είναι ταχύτερα από τα αρχικά ενώ δεν είναι λίγες οι φορές που ακόμα και ο συνολικός χρόνο που χρειάζεται για την εξειδίκευση του αρχικού προγράμματος και την εκτέλεση του εναπομείναντος είναι μικρότερος από τον χρόνο που δαπανάται κατά την εκτέλεση του αρχικού προγράμματος.

2.4.3 Πώς κάνουμε την μερική αποτίμηση

Στα επόμενα κεφάλαια θα περιγράψουμε την μεθοδολογία που ακολουθούμε για να ολοκληρώσουμε την μερική αποτίμηση. Είπαμε πως η μερική αποτίμηση είναι εφικτή, αλλά για να καταφέρουμε να έχουμε αποτελέσματα πέρα από τα τετριμμένα πρέπει να δώσουμε ιδιαίτερη προσοχή σε μερικά θέματα που αφορούν τις λειτουργίες του μερικού αποτιμητή ή ακόμα και να φροντίσουμε ώστε το αρχικό μας πρόγραμμα να έχει μια συγκεκριμένη μορφή που ευνοεί την εξειδίκευση.

2.4.4 Πότε κάνουμε την μερική αποτίμηση

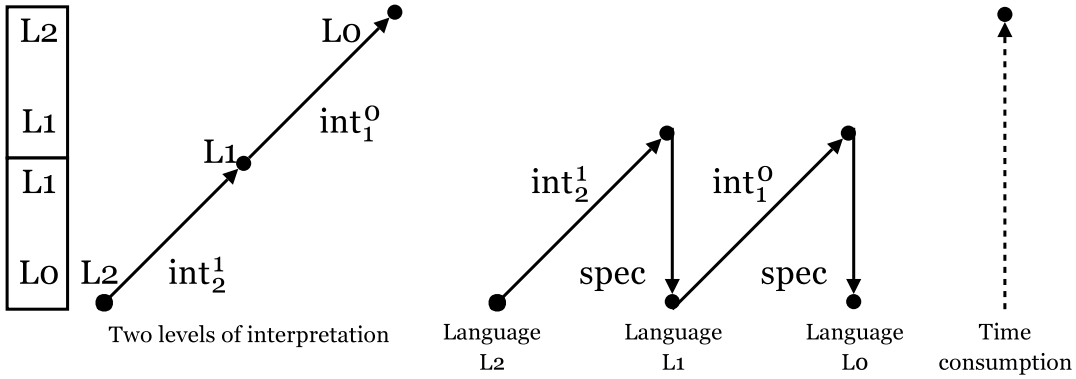
Η επιλογή του πότε θα κάνουμε μερική αποτίμηση και πότε όχι είναι προφανής. Ένα πρόγραμμα προσφέρεται για μερική αποτίμηση όταν με αυτό τον τρόπο εκτελείτε και παράγει τα δεδομένα εξόδου ταχύτερα. Όμως δεν είναι εύκολο να ξέρουμε ότι αυτό θα συμβεί, ούτε και να ορίσουμε πόσο πιο γρήγορα θα εκτελείται. Επιπλέον θα πρέπει να λάβουμε υπόψιν μας και το γενικότερο πρόβλημα που καλούμαστε να λύσουμε. Για παράδειγμα μπορεί όλη η διαδικασία της εξειδίκευσης και εκτέλεσης του εναπομείναντος προγράμματος να είναι χρονοβόρα σε σχέση με την εκτέλεση του αρχικού προγράμματος, αλλά αν το εναπομένον πρόγραμμα σκοπεύουμε να το χρησιμοποιήσουμε περισσότερες από μια φορές αλλάζοντας απλά τα δεδομένα εισόδου του, τότε η μερική αποτίμηση μπορεί να βρει εφαρμογή.

2.4.5 Στρώματα διερμήνευσης

Ας σκεφτούμε τώρα το ακόλουθο σενάριο. Θέλουμε να εκτελέσουμε ένα πρόγραμμα p γραμμένο στη γλώσσα L_2. Έχουμε στην διάθεσή μας ένα διερμηνέα int_2^1 για τη γλώσσα L_2 γραμμένο στην γλώσσα L_1 κι έναν int_1^0 για την γλώσσα L_1 γραμμένο στην L_0. Στην πράξη αυτό μπορεί να σημαίνει ότι η γλώσσα L_0 είναι η γλώσσα μηχανής, η μοναδική γλώσσα που μπορεί να καταλάβει το υλικό των υπολογιστών και άρα η μόνη γλώσσα για την οποία δεν χρειαζόμαστε διερμηνέα. Συνολικά η εκτέλεση του προγράμματος p γίνεται κάπως έτσι:

$$out = [\![p]\!]_{L_2}(in) = [\![int_2^1]\!]_{L_1}(p, in) = [\![int_1^0]\!]_{L_0}(int_2^1, p, in)$$

Όπως βλέπουμε κι από το σχήμα που ακολουθεί, κάθε διερμηνέας που παρεμβάλλεται αυξάνει την υπολογιστική πολυπλοκότητα της εκτέλεσης του προγράμματος p και συνεπώς και τον χρόνο που απαιτείται για την ολοκλήρωσή της.



Σχήμα 2.4: Μεταπρογραμματισμός

Με την εφαρμογή της εξειδίκευσης μπορούμε να μειώσουμε το επιπλέον υπολογιστικό κόστος με δύο τρόπους. Ο πρώτος είναι να κατασκευάσουμε ένα διερμηνέα της γλώσσας L_2 γραμμένο στην L_0 ως εξής:

$$int_2^0 = [\![spec]\!]_{L_0}(int_1^0, int_2^1) \text{ όπου } out = [\![int_2^0]\!]_{L_0}(p, in)$$

Εναλλακτικά μπορούμε να πάμε ένα βήμα πιο πέρα και με βάση τον cogen που ορίστηκε στην τρίτη προβολή του Futamura να μεταγλωττίσουμε μέσω εξειδίκευσης τον διερμηνέα int_2^0:

$$comp_2^0 = [\![cogen]\!]_{L_0}(int_2^0)$$

Το συμπέρασμα από αυτή τη διαδικασία είναι ότι είμαστε σε θέση να εφαρμόσουμε την τεχνική του μεταπρογραμματισμού, δηλαδή να χτίσουμε μια γλώσσα προγραμματισμού πατώντας πάνω σε μία άλλη χωρίς αυτό να συνεπάγεται την αύξηση της πολυπλοκότητας, και τα συνέπεια την μείωση των επιδόσεων.

2.5 Το mix ως μερικός αποτιμητής

Στα επόμενα κεφάλαια όποτε αναφερόμαστε στο mix θα εννοούμε έναν μερικό αποτιμητή, όπως το $spec$. Αυτό που τον διαφοροποιεί από το $spec$ είναι ότι είναι ειδικής χρήσης, αποκλειστικά για την κατασκευή μεταγλωττιστών. Ως εκ τούτου πρέπει να ισχύουν οι προβολές Futamura, από όπου έπεται ότι ο mix είναι γραμμένος στην γλώσσα εισόδου του. Μερικές εξισώσεις που ισχύουν για τον mix, συνοπτικά είναι οι:

$$\begin{array}{ccccc} \text{out} & = & [\![int]\!](source, in) & = & [\![target]\!](in) \\ \text{target} & = & [\![mix]\!](int, source) & = & [\![compiler]\!](source) \\ \text{compiler} & = & [\![mix]\!](mix, int) & = & [\![cogen]\!](int) \\ \text{cogen} & = & [\![mix]\!](mix, mix) & = & [\![cogen]\!](mix) \end{array}$$

Κεφάλαιο 3

Μερική αποτίμηση στην πράξη

Σε αυτό το κεφάλαιο θα προσπαθήσουμε να δείξουμε πώς γίνεται η μερική αποτίμηση στην πράξη. Αρχικά θα παρουσιάσουμε την σύνταξη μιας απλή γλώσσας που θα μας βοηθήσει στα παραδείγματά μας. Στην συνέχεια θα ορίσουμε έννοιες όπως η κατάσταση (state), το σημείο του προγράμματος (program point) και της διαίρεσης (division). Θα υλοποιήσουμε ένα διερμηνέα για την γλώσσα μας και σταδιακά θα αναφερθούμε σε όλες τις μεθόδους μου μεχρι το τελος θα μας βοηθήσουν να παράγουμε ένα αποδοτικό μεταγλωττιστή.

3.1 Μια απλή γλώσσα

Η σύνταξη της γλώσσας μας είναι αυτή που ακολουθεί. Δεν έχουμε μπει σε μεγάλη λεπτομέρεια να ορίσουμε ούτε το τι είναι μια τιμή (value), ούτε ποιοι είναι οι τελεστές (operators), ούτε την ακριβή μορφή των ετικετών (labels). Αυτά θα προκύψουν στην πορεία με βάση τις ανάγκες μας. Αυτό που πρέπει να προσέξουμε είναι ότι στην γλώσσα αυτή το μόνο που μπορούμε να κάνουμε είναι να αναθέσουμε μια τιμή σε κάποια μεταβλητή ή μετά από ένα πεπερασμένο πλήθος αναθέσεων είτε να εκτελέσουμε ένα άλμα, υπό συνθήκη ή μη, προς κάποιο άλλο σημείο του προγράμματος, με αναφορά σε κάποια ετικέτα ή να τερματίσουμε την εκτέλεση του προγράμματος.

```
<Program> ::== read <Var>, ..., <Var>; <BasicBlock>+
<BasicBlock> ::== <Label>: <Assignment>* <Jump>
<Assignment> ::== <Var> := <Expr>;
<Jump> ::== goto <Label>;
         |  if <Expr> goto <Label> else <Label>;
         |  return <Expr>;
<Expr> ::== <Constant>
         |  <Var>
         |  <Op> <Expr> ... <Expr>
<Constant> ::== quote <Val>
<Op> ::== hd | tl | cons | ...
          plus any others needed for writing interpreters or program
specializers
<Label> ::== any identifier or number
```

Είναι προφανές ότι η γλώσσα που ορίζουμε είναι ικανή να υλοποιήσει κάθε αλγόριθμο που χρειαζόμαστε, με μεγαλύτερη δυσκολία προφανώς από ότι αν χρησιμοποιούσαμε μια πιο εκφραστική γλώσσα. Όμως αυτός ήταν ο στόχος μας εξ αρχής, να παρουσιάσουμε δηλαδή μια απλή γλώσσα και με αυτή να εξετάσουμε τις τεχνικές της μερικής αποτίμησης. Καταφέρνουμε έτσι να επικεντρώσουμε την προσοχή μας στο πρόβλημα που μας απασχολεί και όχι στις λεπτομέρειες της γλώσσας.

Ένα παράδειγμα προγράμματος σε αυτή τη γλώσσα είναι το παρακάτω. Αν μάλιστα δώσουμε και λίγη προσοχή διαπιστώσουμε ότι υλοποιεί τον αλγόριθμο του Ευκλείδη για την εύρεση του μέγιστου κοινού διαιρέτη δύο αριθμών.

```
read x, y;
1: if x = y goto 7 else 2
2: if x < y goto 5 else 3
3: x := x - y;
   goto 1
5: y := y - x;
   goto 1
7: return x
```

3.2 Ο διερμηνέας της γλώσσας

Ακολούθως παρουσιάζουμε μια απλή υλοποίηση ενός διερμηνέα για τη γλώσσα μας, γραμμένο σε ML. Η σημασιολογία της γλώσσας μπορεί άμεσα να εξαχθεί κοιτάζοντας τον κώδικα αυτό του διερμηνέα. Μοναδική επισήμανση είναι η χρήση μιας αποθήκης (store), η οποία στην πράξη είναι ένα ζεύγος από δύο λίστες ίδιου μήκους, όπου στην πρώτη συναντάμε τα ονόματα των μεταβλητών και στην δεύτερη τις τιμές τους, στην ίδια θέση. Η αποθήκη χρησιμοποιείτε για τρεις λειτουργείς, αναζήτη της τιμής μιας μεταβλητής με βάση το όνομά της, καταχώρηση ενός ζεύγους μεταβλητής και τιμής, και σαν υποπερίπτωση της καταχώρησης, την ενημέρωση της τιμής μια μεταβλητής, εφόσον αυτή υπάρχει ήδη στην αποθήκη.

```
datatype expr =
  Int of int                  (* Integer Constant *)
  | Var of string             (* Variable *)
  | Op of string * expr list  (* Base application *)
and command =
    Goto of int               (* Goto command *)
  | Assing of string * expr   (* Assingment command *)
  | If of expr * int * int    (* If-then-else command *)
  | Return of expr            (* Return command *)
and program = Read of string list * command list

fun nth (c::cs, 1) = c
  | nth (c::cs, n) = nth(cs, n-1)

fun lookup (x, ([], [])) = 0 (* Initial values *)
  | lookup (x, (n::ns, v::vs)) =
      if x = n then v else lookup(x, (ns, vs))

fun update (([], []), x, w) = ([x], [w]) (* First assignment)
  | update ((n::ns, v::vs), x, w) =
      if x = n then (n::ns, w::vs) (* Update value *)
      else let val (ns1, vs1) = update((ns, vs), x, w) (* Insert variable *)
        in (n::ns1, v::vs1) end

fun eval (Int n, s) = n
  | eval (Var x, s) = lookup(x, s)
  | eval (Op("+", [e1, e2]), s) = eval(e1, s) + eval(e2, s)
(* do the same for the rest of the operators *)

fun run (l, Goto n, s, p) = run(n, nth(p, n), s, p)
  | run (l, Assign(x, e), s, p) =
```

```
      let val s1 = update(s, x, eval(e, s))
      in run(l+1, nth(p, l+1), s1, p) end
  | run (l, If(e, m, n), s, p) =
      if eval(e, s) = 0
      then run(m, nth(p, m), s, p)
      else run(n, nth(p, n), s, p)
  | run (l, Return e, s, p) = eval(e, s)

fun interpret(pgm, args) =
  let val Read(vars, cmds) = pgm
    val (c1::_) = cmds
    val store = (vars, args)
  in run(1, c1, store, cmds) end
```

3.3 Υπολογιστική κατάσταση

Μια υπολογιστική κατάσταση είναι ένα στιγμιότυπο της εκτέλεσης ενός προγράμματος. Για τα προγράμματα που είναι γραμμένα στην γλώσσα που περιγράψαμε, κάθε στιγμιότυπο μπορεί να χαρακτηριστεί μονοσήμαντα από ένα ζεύγος τιμών (pp, store). Το pp αντιπροσωπεύει το σημείο του προγράμματος στο οποίο βρίσκετε ο έλεγχος και το store είναι η αποθήκη που περιέχει όλες τις τιμές των μεταβλητών του προγράμματος. Για παράδειγμα, έστω ότι βρισκόμαστε στην κατάσταση (pp, store), και στο σημείο pp η εντολή είναι η `pp: x := 3;` τότε μετά την εκτέλεση αυτής της εντολής η αναμενόμενη κατάσταση θα είναι η $(pp + 1, store[x \mapsto 3])$. Αν η εντολή ήταν η `pp: goto pp1;` τότε μετά την εκτέλεση η αναμενόμενη κατάσταση θα ήταν η (pp1, store), όπου θα έχει αλλάξει σε σχέση με την προηγούμενη κατάσταση μόνο το σημείο του προγράμματος αφού η εντολή goto δεν επιδρά με την αποθήκη. Με ανάλογο τρόπο ορίζονται οι επόμενες καταστάσεις δεδομένης την τρέχουσας και της προς εκτέλεσης εντολής, για κάθε εντολή. Είναι δυνατό μία κατάσταση να έχει ως επόμενη κατάσταση όχι μόνο μία, αλλά περισσότερες καταστάσεις. Αυτό για την γλώσσα μας γίνεται όταν ο έλεγχος φτάσει σε μια εντολή If, όπου η επόμενη κατάσταση είναι ένα σύνολο δύο καταστάσεων όπου κάθε μια προκύπτει από διαφορετικό κλάδο της εντολής διακλάδωσης.

3.4 Εναπομένον πρόγραμμα

Υποθέτουμε τώρα ότι έχουμε δεδομένο μόνο ένα υποσύνολο των μεταβλητών εισόδου. Αυτές οι μεταβλητές δεν επαρκούν για να γεμίσουμε με τις απαραίτητες πληροφορίες την αποθήκη. Έτσι αντί να πάρουμε ένα τελικό πρόγραμμα, μπορούμε να εκμεταλλευτούμε την μερική γνώση για τα δεδομένα εισόδου και να κατασκευάσουμε ένα εναπομένον πρόγραμμα (residual program). Σε αυτό το πρόγραμμα η αποθήκη στην αρχική και κάθε επόμενη κατάσταση θα είναι ημιτελής οπότε κατά την στιγμή της εξειδίκευσης δεν θα μπορούν να αποτιμηθούν όλες οι εκφράσεις του αρχικού προγράμματος.

3.5 Διαίρεση

Για να μπορούμε κάθε στιγμή να ξέρουμε ποιες μεταβλητές είναι γνωστές και ποιες όχι κάνουμε μια κατηγοριοποίηση και θεωρούμε πως κάθε μεταβλητή είναι είτε στατική είτε δυναμική. Στατική θα πει πως μπορούμε να ανακτήσουμε την τιμή της μεταβλητής ανά πάσα στιγμή κατά την διάρκεια της εξειδίκευσης. Δυναμικές είναι όσες μεταβλητές δεν είναι στατικές. Με βάση αυτή τη κατηγοριοποίηση την οποία στο εξής θα ονομάζουμε διαίρεση ορίζουμε την μερική υπολογιστική κατάσταση ως το ζεύγος (pp, vs) όπου το pp αντιστοιχεί σε κάποιο σημείο του

προγράμματος, και το vs αντιπροσωπεύει το σύνολο των τιμών των στατικών μεταβλητών της αποθήκης σε εκείνο το σημείο.

Η διαίρεση που περιγράψαμε πιο πάνω δεν είναι η μοναδική δυνατή. Στη συνέχεια θα δούμε έναν αλγόριθμό για την κατασκευή της και διάφορες εναλλακτικές στρατηγικές υπολογισμού μιας διαίρεσης Όπου δεν αναφέρεται διαφορετικά θα θεωρούμε ότι η μία αρχική διαίρεση ισχύει για όλα τα σημεία του προγράμματος, και θα την ονομάζουμε ενιαία Μια βασική προϋπόθεση ώστε η διαίρεση να έχει αξία στην μερική αποτίμηση είναι ότι θα πρέπει να είναι ταιριαστή (congruent). Αυτό θα πει, πως κατά τη μετάβαση από μια υπολογιστική κατάσταση (pp, s) σε μια δεύτερη (pp', s'), οι τιμές των στατικών μεταβλητών του συνόλου s' υπολογίζονται αποκλειστικά και μόνο από τις τιμές του συνόλου s. Ως επακόλουθο αυτής της απαίτησης, οποιαδήποτε μεταβλητή εξαρτάται σε κάποιο σημείο του προγράμματός μας από κάποια δυναμική μεταβλητή πρέπει να είναι κι η ίδια δυναμική. Με όμοιο τρόπο χαρακτηρίζουμε ως στατικές 'η δυναμικές και τις εκφράσεις, δηλαδή αν μια έκφραση αποτελείτε μόνο από σταθερές και στατικές μεταβλητές τότε είναι στατική. Σε αντίθετη περίπτωση θεωρείται δυναμική.

Στο παρακάτω αν θεωρήσουμε ότι οι μεταβλητές name και namelist είναι στατικές τότε η μεταβλητές value και valuelist προκύπτει ότι είναι δυναμικές:

```
search: if name = hd (namelist) goto found else cont;
cont:   valuelist := tl (valuelist);
        namelist := tl (namelist);
        goto search;
found:  value := hd (namelist);
```

3.6 Εξειδίκευση σημείου προγράμματος

Η βασική ιδέα της εξειδίκευσης σε επίπεδο σημείου προγράμματος είναι να εισάγουμε τις τιμές των στατικών μεταβλητών στα σημεία που εμφανίζεται αναφορά προς αυτές. Ας υποθέσουμε πως κατά την κανονική εκτέλεση του προγράμματος ο έλεγχος φτάνει στο σημείο pp και ταυτόχρονα το vs είναι το σύνολο των στατικών μεταβλητών. Τότε θέτουμε το (pp, vs) να είναι ένα σημείο στο εναπομένον πρόγραμμα και ο κώδικας σε αυτό το σημείο είναι μια βελτιστοποιημένη εκδοχή του κώδικα στο σημείο pp του αρχικού προγράμματος. Την δυνατότητα για βελτιστοποίηση την έχουμε λόγω της γνώσης των τιμών των στατικών μεταβλητών. Όπως γίνεται φανερό από τα προηγούμενα, ένα σημείο του αρχικού προγράμματος μπορεί να εμφανίζεται σε περισσότερα από ένα σημεία του εναπομείναντος προγράμματος.

Θεωρούμε και πάλι το πρόγραμμα που παρουσιάσαμε πιο πάνω. Στόχος μας είναι να εξειδικεύσουμε τον κομμάτι με ετικέτα search ενώ ξέρουμε ότι αρχικά `name := z` και `namelist := (x y z)`, με άλλα λόγια `vs = (z, (x y z))`. Εκτελώντας με το χέρι το πρόγραμμα βλέπουμε ότι στο σημείο search τα στοιχεία του συνόλου vs αποκτάνε διαδοχικά τις τιμές `(z, (x y z))`, `(z, (y z))` και `(z, (z))`. Αντιστοίχως στο σημείο cont το vs γίνεται `(z, (x y z))` και `(z, (y z))` και στο σημείο found, `vs = (z, (z))`.

Ορίζουμε ως σημείο του εξειδικευμένου προγράμματος το ζευγάρι (pp, vs) όπου το pp αντιστοιχεί στο σημείο του αρχικού προγράμματος που εξειδικεύεται και η λίστα vs αντιπροσωπεύει τις τιμές των στατικών μεταβλητών βάση των οποίον γίνεται η εξειδίκευση. Στην ουσία ένα σημείο του εξειδικευμένου προγράμματος αντιστοιχίζεται σε ένα σύνολο από καταστάσεις, από τις οποίες μπορεί να περάσει η εκτέλεση του αρχικού προγράμματος, όπου το σημείο είναι το pp και η αποθήκη περιέχει τις τιμές του vs στην θέση των τιμών των στατικών μεταβλητών, αφήνοντας ελευθερία στην επιλογή των τιμών των δυναμικών παραμέτρων.

Ορίζουμε ως poly το σύνολο των σημείων του εξειδικευμένου προγράμματος από τα οποία είναι περνάει η εκτέλεση του αρχικού προγράμματος. Το σύνολο poly μπορούμε να το αντιληφθούμε και ως το κλείσιμο της αρχικής κατάστασης (pp_0, vs_0) υπό τη σχέση "η επόμενη κατάσταση μπορεί να είναι". Για παράδειγμα για το πρόβλημα που αναφέραμε πριν έχουμε ότι

```
poly = { (search, (z, (x y z))), (search, (z, (y z))), (search, (z, (z))),
         (cont,  (z, (x y z))), (cont,  (z, (y z))),
         (found, (z, (z))) }
```

Για τον υπολογισμό του συνόλου poly χρησιμοποιούμε την συνάρτηση successors η οποία δέχεται σαν όρισμα ένα σημείο του εξειδικευμένου προγράμματος (pp, vs) και δίνει σαν έξοδο ένα σύνολο successors(pp, vs) με τα σημεία του εξειδικευμένου προγράμματος στα οποία είναι πιθανό να μεταβεί ο έλεγχος μετά της εκτέλεση των εντολών σε αυτό το σημείο. Είναι προφανές ότι αν η ετικέτα (pp, vs) αντιστοιχεί σε εντολή διακλάδωσης, τότε το σύνολο successors θα περιέχει τόσα στοιχεία όσες είναι και διαφορετικές επιλογές (στην περίπτωση του if δύο), αν αντιστοιχεί σε εντολή επιστροφής θα είναι κενό και σε κάθε άλλη περίπτωση θα είναι μονοσύνολο.

3.7 Ελάττωση εκφράσεων

Δεδομένου πλέον του αρχικού προγράμματος και του συνόλου poly μπορούμε να κάνουμε τα πρώτα μας βήματα στην εξειδίκευσή του. Πέρα από την κατάταξη των μεταβλητών σε στατικές και δυναμικές κατά το χρόνο εξειδίκευσης διαχωρίζουμε και τις ενέργειες που πρέπει να γίνουν, σε αυτές που μπορούν να γίνουν κατά την εξειδίκευση και σε αυτές που πρέπει να γίνουν κατά την εκτέλεση του εναπομείναντος προγράμματος οπότε πρέπει να παραχθεί ο ανάλογος κώδικας. Η πιο χαρακτηριστική ενέργεια κατά την εξειδίκευση είναι η αντικατάσταση της αναφοράς σε μια στατική μεταβλητή με την τιμή αυτής ή η αποτίμηση ενός στατικού κομματιού (με την έννοια ότι εξαρτάται αποκλειστικά από τιμές στατικών μεταβλητών) μιας κατά τα άλλα δυναμικής έκφρασης. Αυτή η διαδικασία ονομάζεται ελάττωση (reduc tion) και εκτελείτε από την συνάρτηση reduce, που καλείται πάντα με πρώτο όρισμα την μεταβλητή ή έκφραση προς μείωση και δεύτερο το σύνολο vs. Μπορούμε να συνοψίσουμε τις περιπτώσεις της εξειδίκευσης στον παρακάτω πίνακα.

Command	**Done at specialization time**	**Generated code**
`X := exp` (X is dynamic)	`reduced_exp := reduce(exp, vs)`	`X := reduced_exp`
`X := exp` (X is static)	`val := eval(exp, vs);` `vs := vs[X->val]`	
`return exp`	`reduced_exp := reduce(exp, vs)`	`return reduced_exp`
`goto pp'`	`goto (pp', vs)`	

`if exp goto pp' else pp''`
(exp is dynamic)
	`reduced_exp := reduce(exp, vs)`	`if reduced_exp`
		`goto (pp', vs)`
		`else (pp'', vs)`

`if exp goto pp' else pp''` (exp is static, reduced_exp = true)	`reduced_exp := reduce(exp, vs)`	`goto(pp', vs)`
`if exp goto pp' else pp''` (exp is static, reduced_exp = false)	`reduced_exp := reduce(exp, vs)`	`goto(pp'', vs)`

3.8 Τύλιγμα καταστάσεων

Μία ακόμα τεχνική που εφαρμόζουμε κατά την εξειδίκευση των προγραμμάτων είναι το τύλιγμα των μεταβάσεων (transition folding), όπου με τον όρο μετάβαση αναφερόμαστε στην μετάβαση από την μία υπολογιστική κατάσταση στην επόμενη. Εφαρμόζοντας πιστά τις οδηγίες του παραπάνω πίνακα, είναι δυνατό να παράγουμε κώδικα ώστε ο έλεγχος να περάσει στο επόμενο σημείο ελέγχου, όμως σε αυτό το σημείο να υπάρχει μια στατική έκφραση η οποία ελαττώνεται και κατά συνέπεια δεν παράγεται κώδικας για αυτή. Αν ακολουθεί ακόμα μία εντολή άλματος, τότε θα δημιουργηθεί και πάλι κώδικας για να γίνει η μετάβαση στην επόμενη κατάσταση όπου κι εκεί αντιμετωπίζουμε το ίδιο πρόβλημα. Ας δούμε για παράδειγμα τι συμβαίνει όταν

εξειδικεύουμε το πρόγραμμα που παρουσιάσαμε στην αρχή. Το εναπομένον πρόγραμμα είναι το εξής

```
(search, (z, (x y z))): goto (const, (z, (x y z)));
(const,  (z, (x y z))): valuelist := tl (valuelist);
                        goto (search, (z, (y z)));
(search,   (z, (y z))): goto (const, (z, (y z)));
(const,    (z, (y z))): valuelist := tl (valuelist);
                        goto (searchl, (z, (z)));
(search,     (z, (z))): goto (found, (z, (z)));
(found,      (z, (z))): value := hd (valuelist)));
```

Το αποτέλεσμα σίγουρα είναι ορθό, αλλά παρατηρούμε ότι γίνονται αρκετές άσκοπες μεταβάσεις. Η τεχνική που περιγράφουμε, το τύλιγμα των μεταβάσεων, αποσκοπεί στο να εξαφανίσει τα περιττά άλματα. Αν θεωρήσουμε πως στο αρχικό πρόγραμμα συναντάμε την ετικέτα pp και έστω ότι σε κάποιο σημείο του υπάρχει μια εντολή άλματος προς αυτή την ετικέτα Η αντικατάσταση του άλματος με τον κώδικα στον οποίο δείχνει η ετικέτα καθεαυτό ονομάζεται τύλιγμα μετάβασης.

Όταν τυλίγουμε το προηγούμενο πρόγραμμα για να εξαλείψουμε τα πλεονάζοντα άλματα παίρνουμε το εναπομένον πρόγραμμα που επιθυμούμε, με μόνη ίσως παραφωνία το περίεργο όνομα των ετικετών, κάτι που μπορούμε εύκολα να αλλάξουμε με συνεπή τρόπο σε όλο το πρόγραμμα.

```
(search, (z, (x y z))): valuelist := tl (valuelist);
                        valuelist := tl (valuelist);
                        value := hd (valuelist);
```

Τα κέρδη της τεχνικής του τυλίγματος των μεταβάσεων είναι εμφανή. Πρώτα απ' όλα το εναπομένον πρόγραμμα είναι σημαντικά μικρότερο σε μέγεθος. Επιπλέον είναι αρκετά πιο αποδοτικό, γιατί οι εντολές άλματος αν και χρησιμοποιούνται συνεχώς είναι ακριβές σε υπολογιστικό χρόνο. Παρόλα αυτά η αδιάκριτη χρήση του τυλίγματος κρύβει δύο παγίδες. Μπορεί να εισάγουμε στο τελικό πρόγραμμα τον ίδιο κώδικα πολλές φορές, όταν τυλίγουμε δύο ή περισσότερες μεταβάσεις προς το ίδιο σημείο του προγράμματος. Ακόμα χειρότερα μπορούμε να οδηγηθούμε σε άπειρο τύλιγμα, δηλαδή αν το εναπομένον πρόγραμμα περιέχει κάποιο βρόχο είναι πιθανό να εκτελούμε συνεχώς διαδοχικές αντικαταστάσεις της μιας εντολής άλματος με την άλλη και η διαδικασία της εξειδίκευσης να μην τερματίσει ποτέ.

Προκύπτει λοιπόν το θέμα πότε μπορούμε να τυλίξουμε μια μετάβαση. Μια ασφαλής αντιμετώπιση είναι να εφαρμόσουμε την τεχνική του τυλίγματος των καταστάσεων σαν ένα στάδιο μετά την εξειδίκευση. Έτσι μπορούμε να αναλύσουμε το πρόγραμμα και να αποφύγουμε τα προβλήματα που αναφέρθηκαν. Στην πράξη όμως αυτό δεν είναι αποδεκτό σαν λύση, καθώς επιθυμούμε το τύλιγμα να γίνεται κατά τη διάρκεια της εξειδίκευσης γιατί έτσι όλη η διαδικασία θα είναι πιο αποδοτική, δεν θα χρειαστεί να παράγουμε κώδικα που θα διαγράψουμε στη συνέχεια. Η ασφαλής λύση που έχει προταθεί λοιπόν είναι να τυλίξουμε όλες τις εντολές άλματος που γίνονται προς ετικέτες που δεν παρουσιάζονται σε κάποια εντολή διακλάδωσης του εναπομείναντος προγράμματος. Το τελικό πρόγραμμα με αυτή τη διαδικασία μπορεί να περιέχει διπλά αντίτυπα του ίδιου κώδικα είναι όμως εξασφαλισμένο ότι η διαδικασία της εξειδίκευσης δεν θα εισέλθει σε κάποιο άπειρο βρόχο, το οποίο είναι και το σημαντικότερο

3.8.1 Η σημασία της σωστής διαίρεσης

Μέχρι εδώ δε δώσαμε ιδιαίτερη σημασία στην επιλογή της σωστής διαίρεσης, της κατηγοριοποίησης των μεταβλητών σε στατικές και δυναμικές. Θεωρήσαμε ότι στατικές είναι όσες

μεταβλητές προκύπτουν από υπολογισμούς με μεταβλητές που είναι ήδη γνωστό ότι είναι στατικές και με σταθερές. Το παρακάτω πρόγραμμα όμως μπορεί να αποδείξει ότι η μέθοδος αυτή δεν είναι πάντοτε σωστή.

```
iterate: if Y != 0 then begin
           X := X + 1;
           Y := Y - 1;
           goto iterate;
```

Το πρόγραμμα χρησιμοποιεί δύο μεταβλητές, την X και την Y, κι έστω ότι γνωρίζουμε ότι η τιμή της X είναι 0, οπότε αφού η τιμή της εξαρτάται μόνο από τον εαυτό της και την σταθερά 1, την χαρακτηρίζουμε στατική. Αυτό σημαίνει ότι για την εντολή `X := X + 1;` δεν παράγουμε κώδικα, απλά κατά στην εξειδίκευση υπολογίζουμε την ανανεωμένη τιμή της X. Ας προσπαθήσουμε τώρα να υπολογίσουμε το σύνολο poly. Αρχικά στο σύνολο εισάγουμε το σημείο του εξειδικευμένου προγράμματος (iterate, 0). Εφαρμόζοντας τους κανόνες που περιγράψαμε στον παραπάνω πίνακα παίρνουμε το πρόγραμμα

```
(iterate, 0): if Y != 0 then
                Y := Y - 1;
                goto (iterate, 1);
```

και προσθέτουμε στο poly το (iterate, 1). Για αυτό το λόγο παράγεται και ο επόμενος κώδικας

```
(iterate, 1): if Y != 0 then
                Y := Y - 1;
                goto (iterate, 2);
```

Σε αυτό το σημείο μπορούμε και είμαστε σίγουροι ότι η διαδικασία της εξειδίκευσης δεν πρόκειται να τελειώσει ποτέ, και το σύνολο poly περιέχει μη πεπερασμένο αριθμό στοιχείων

```
poly = { (iterate, 0), (iterate, 1), (iterate, 2), ... }
```

Το πρόβλημα εντοπίζεται στο ότι η τιμή της μεταβλητής X, αν και γνωστή αρχικά, δεν είναι φραγμένη, διότι μεταβάλλεται συνεχώς από μια επαναληπτική διαδικασία που ελέγχεται από μια δυναμική έκφραση. Για να λύσουμε το πρόβλημα οφείλουμε την μεταβλητή X και κάθε μεταβλητή με ανάλογες ιδιότητες να την χαρακτηρίσουμε δυναμική. Ονομάζουμε πεπερασμένη μια διαίρεση που μας εξασφαλίζει πεπερασμένο πλήθος στοιχείων στο σύνολο poly.

Σε αυτή την περίπτωση που μελετάμε λοιπόν οφείλουμε να θεωρήσουμε την μεταβλητή X δυναμική, και το εναπομένον πρόγραμμα θα είναι τότε ίδιο με το αρχικό. Η διαδικασία της κατηγοριοποίησης μιας μεταβλητής ως δυναμική, ακόμα κι αν η απαίτηση για ταιριαστή διαίρεση μας επέτρεπε να της αποδώσουμε τον χαρακτηρισμό στατική, ονομάζεται γενίκευση. Όπως είδαμε η γενίκευση είναι απαραίτητη για να εξασφαλίσουμε ότι η εξειδίκευση θα τερματίσει, όμως ο καθορισμός της βέλτιστης διαίρεσης αποδεικνύεται ότι δεν είναι εφικτό να υπολογιστεί (το πρόβλημα ανάγεται στο halting problem).

3.9 Mix

Μέχρι στιγμής έχουμε περιγράψει τις τεχνικές που όλες μαζί συγκροτούν τον μηχανισμό της εξειδίκευσης των προγραμμάτων. Παρουσιάστηκε η κάθε τεχνική μόνη της αλλά τώρα σκοπός μας είναι να τις συνδυάσουμε ώστε να κατασκευάσουμε ένα πρόγραμμα που θα επιτελεί την εξειδίκευση εφαρμόζοντας όλες τις μεθόδους στην σειρά. Πιο συγκεκριμένα θέλουμε τον αλγόριθμο που πληρεί τις εξής προϋποθέσεις:

- Είσοδος: Ένα αρχικό πρόγραμμα, ένα μέρος της εισόδου του προγράμματος και βάση αυτής μια διαίρεση των μεταβλητών του σε στατικές και δυναμικές.
- Έξοδος: Ένα εναπομένον πρόγραμμα στο οποίο με τη σειρά του, θα του δίνουμε το υπόλοιπο μέρος της εισόδου και θα μας δίνει στην έξοδο το ίδιο αποτέλεσμα, που θα αναμέναμε από το αρχικό πρόγραμμα αν του δίναμε όλα τα δεδομένα εισόδου.

Μπορούμε να περιγράψουμε τον ζητούμενο αλγόριθμο ως εξής:

- Υπολογισμός συνόλου poly και παραγωγή κώδικα για κάθε σημείο του εξειδικευμένου προγράμματος που εισάγουμε στο σύνολο
- Εφαρμογή της τεχνικής του τυλίγματος μεταβάσεων
- Αλλαγή των ετικετών του εναπομείναντος προγράμματος ώστε στα σημεία του προγράμματος να αντιστοιχούν φυσικοί αριθμοί σαν ετικέτες

Είδαμε όμως και πιο πάνω ότι στην πράξη δεν θέλουμε να ξεχωρίσουμε το στάδιο το υ τυλίγματος καταστάσεων από το στάδιο παραγωγής κώδικα. Εφαρμόζοντας το τέχνασμα που αναφέραμε όταν μιλούσαμε για το πότε είναι ασφαλές να τυλίξουμε μια μετάβαση θα μπορούμε να υπολογίζουμε το σύνολο poly, να παράγουμε κώδικα και να τυλίγουμε τις μεταβάσεις σε ένα στάδιο μόνο. Επιπλέον θα θεωρήσουμε δύο σύνολα, το pending και το marked, το πρώτο θα περιέχει τα σημεία του εξειδικευμένου προγράμματος τα οποία δεν τα έχουμε ακόμα επεξεργαστεί, δηλαδή δεν έχουμε αποφασίσει αν θα πρέπει να παράγουμε ή όχι κώδικα, τί ενέργειες θα γίνουν εσωτερικά από το πρόγραμμα που κάνει την εξειδίκευση ούτε ποιες είναι οι πιθανές επόμενες καταστάσεις, και το δεύτερο θα περιέχει εκείνα που έχουμε επεξεργαστεί.

Ο συνολικός αλγόριθμος ακολουθεί, αν κι έχουμε παραλείψει το στάδιο της αλλαγής ετικετών, που είναι εύκολο να γίνει.

```
read(program, division, vs0);
pending := { (pp0, vs0) }; (* pp0 is program's initial program point *)
marked := ∅;
while pending != ∅ do
begin
  Pick an element (pp, vs) ∈ pending and remove it;
  marked := marked ⋃ { (pp, vs) };
  bb := lookup (pp, program); (* Find the basic block labeled by pp in program*)
                              (* Now generate residual code for bb given vs *)
  code := initial code(pp, vs); (* An empty basic block with label (pp, vs) : *)

  while bb is not empty do
  begin
    command := first command (bb);
    bb := rest (bb);

    case command of

      X := exp:
        if X is classified as static by division
          then vs := vs[X ↦ eval(exp, vs)]; (* Static assignment *)
          else code := extend (code, X := reduce(exp, vs)); (* Dynamic assignment *)

      goto pp':
        bb := lookup (pp', program); (* Compress the transition *)
```

```
      if exp then goto pp' else goto pp'':
        if exp is static by division (* Static conditional *)
          then begin
            if eval (exp, vs) = true
            then bb := lookup (pp', program); (* Compress the transition *)
            else bb := lookup (pp'', program); (* Compress the transition *)
          end
          else begin (* Dynamic conditional *)
            pending := pending ⋃ ({(pp', vs)} \ marked);
            pending := pending ⋃ ({(pp'',vs)} \ marked);
            code    := extend (code, if reduce(exp, vs)
                                              goto (pp', vs)
                                              else (pp'', vs) );
          end

      return exp:
        code := extend (code, return reduce (exp, vs));

      otherwise error;
    end; (* while bb is not empty *)

  residual := extend(residual, code); (* add new residual basic block *)
end (* while pending != ∅ *)
```

3.10 Παράδειγμα εφαρμογής του *mix*

Σε αυτό το σημείο μπορούμε να παρουσιάσουμε ένα πραγματικό παράδειγμα. Θα γράψουμε το παρακάτω πρόγραμμα που είναι ένας διερμηνέας της μηχανής του Turing γραμμένος στην γλώσσα που ορίσαμε πριν. Είναι γνωστό πως η μηχανή Turing διαβάζει ως είσοδο μια ταινία από 0, 1 και κενά και η έξοδός της θεωρείται πως είναι η τιμή που βρίσκεται από το στοιχείο στο οποίο δείχνει η κεφαλή ανάγνωσης μέχρι και πριν το πρώτο κενό στοιχείο όταν τερματιστεί η εκτέλεση του προγράμματος, όταν δηλαδή δεν υπάρχει επόμενη εντολή προς εκτέλεση. Εξίσου απλά είναι και τα προγράμματα της μηχανής Turing που αποτελούνται μόνο από 5 βασικές λειτουργίες

- `right` Μετακινεί την κεφαλή ανάγνωσης στο επόμενο στοιχείο
- `left` Μετακινεί την κεφαλή ανάγνωσης στο προηγούμενο στοιχείο
- `write a` Γράφει το στοιχείο a στο σημείο που δείχνει η κεφαλή ανάγνωσης
- `goto i` Μεταφέρεται ο έλεγχος στην εντολή i
- `if a goto i` Αν το στοιχείο κάτω από την κεφαλή ανάγνωσης είναι το a τότε μεταφέρεται ο έλεγχος στην εντολή i

Ένα απλό πρόγραμμα της μηχανής του Turing είναι αυτό που ακολουθεί

```
0: if 0 goto 3
1: right
2: goto 0
3: write 1
```

Πρέπει να επισημάνουμε ότι ως υπολογιστική κατάσταση στην της μηχανής του Turing θεωρείται η τρέχουσα εντολή που θα εκτελεστεί και μια ταινία απείρου μήκους της μορφής $...a_{-2}a_{-1}a_0a_1a_2..$ όπου η κεφαλή ανάγνωσης δείχνει στο a_0 Αν θεωρήσουμε ότι εκτελούμε το παραπάνω πρόγραμμα και σαν είσοδο δίνουμε την ταινία 110101 όπου υποθέτουμε ότι η αρίθμηση ξεκινάει από το 0 και ότι όλα τα άλλα στοιχεία είναι κενά, τότε αναμένουμε το αποτέλεσμα της εκτέλεσης να είναι το 1101.

Κωδικοποιούμε το πρόγραμμα αυτό στη μορφή `Q = (0: if 0 goto 3, 1: right, 2: goto 0, 3: write 1)`, δηλαδή σε μια λίστα με στοιχεία της μορφής Label: Operator. Θεωρούμε επίσης χωρίς βλάβη της γενικότητας ότι η είσοδος του προγράμματος είναι μια ταινία με κενά όλα τα στοιχεία πριν την κεφαλή ανάγνωσης. Με αυτές τις προϋποθέσεις παρουσιάζουμε παρακάτω τον κώδικα του διερμηνέα της μηχανής Turing.

```
read (Q, Right);
init:     Qtail := Q; Left := '();

loop:     if Qtail = '() goto stop else cont;
cont:     Instruction := first instruction(Qtail);
          Qtail       := rest(Qtail);
          Operator    := hd(tl(Instruction));

          if Operator =  'right goto do-right else cont1;
cont1:    if Operator =  'left  goto do-left  else cont2;
cont2:    if Operator =  'write goto do-write else cont3;
cont3:    if Operator =  'goto  goto do-goto  else cont4;
cont4:    if Operator =  'if    goto do-if    else error;

do-right: Left      := cons(firstsym(Right), Left);
          Right     := tl(Right); goto loop;
do-left:  Right     := cons(firstsym(Left), Right);
          Left      := tl(Left); goto loop;
do-write: Symbol    := hd(tl(tl(Instruction)));
          Right     := cons(Symbol,tl(Right)); goto loop;
do-goto:  Nextlabel := hd(tl(tl(Instruction)));
          Qtail     := new_tail(Nextlabel, Q); goto loop;
do-if:    Symbol    := hd(tl(tl(Instruction)));
          Nextlabel := hd(tl(tl(tl(tl(Instruction)))));
          if Symbol = firstsym(Right) goto jump else loop;

jump:     Qtail     := new_tail(Nextlabel,Q); goto loop;

error:    return ('syntax-error: Instruction);

stop:     return right;
```

Το μόνο που χρήζει της προσοχής μας είναι ότι ο διερμηνέας κάνει χρήση έτοιμων συναρτήσεων, της new_tail, η οποία στην είσοδό της δέχεται μια ετικέτα και το πρόγραμμα Q και δίνει στην έξοδο το υποσύνολο του προγράμματος που ξεκινάει από αυτή την ετικέτα, της first_instuction, που επιστρέφει την πρώτη εντολή από μια ακολουθία εντολών και την rest που επιστρέφει τις υπόλοιπες εντολές της ακολουθίας. Υποθέσαμε ακόμα ότι `firstsym () = B` και `tl () = ()`. Για να γίνουν όλα αυτά κατανοητά ακολουθούν μερικά συνοπτικά παραδείγματα

$$
\begin{aligned}
[\![int]\!]_L(Q, 1101) &= 1 \\
\text{new\_tail(2, Q)} &= \text{(2: goto 0 3: write 1)} \\
\text{first\_instruction(Q)} &= \text{(0: if 0 goto 3)} \\
\text{rest(Q)} &= \text{(1: right 2: goto 0 3: write 1)}
\end{aligned}
$$

Ξέρουμε ότι $[\![s]\!]_S(d) = [\![int]\!]_L s(s,d) = [\![([\![mix]\!]_L(int, s))]\!]_L(d) = [\![target]\!]_L(d)$ και έχουμε γνωστά τα mix, int και s. Άρα μπορούμε να κατασκευάσουμε το $target = [\![mix]\!]_L(int, s)$ με βάση τα όσα έχουμε πει και περιγράψει πιο πριν. Υποθέτουμε ότι έχουμε την διαίρεση, ή καλύτερα ότι αυτή υπολογίζεται αυτόματα από τον κώδικα του int, οπότε εφαρμόζουμε τον αλγόριθμο που παρουσιάσαμε, και το αποτέλεσμα είναι αυτό που ακολουθεί.

```
read (Right):
lab0: Left := '();
      if '0 = firstsym(Right) goto lab2 else lab1;
lab1: Left := cons(first(Right), Left);
      Right := tl(Right);
      if '0 = firstsym(Right) goto lab2 else lab1;
lab2: Right := cons('1, tl(Right));
      return(Right);
```

Με δεδομένα τα mix και int όμως, η δεύτερη προβολή Futamura $compiler = [\![mix]\!]_L(mix, int)$, μας λέει ότι μπορούμε να πάμε ένα βήμα ακόμα πιο κάτω και να κατασκευάσουμε έναν μεταγλωττιστή για την μηχανή Turing

3.11 Εξειδικεύοντας το mix

Όταν θέλουμε να εξειδικεύσουμε το mix σε σχέση με το int πρέπει να αποφασίσουμε μια διαίρεση για τις μεταβλητές του mix. Σκοπός μας εδώ δεν είναι να αντιμετωπίσουμε το ζήτημα στην πλήρη του έκταση όπως κάναμε με την μηχανή Turing γιατί ο κώδικας του mix είναι μεγαλύτερος αλλά να εντοπίσουμε τα σημεία που μπορεί να μας δυσκολέψουν κατά την εξειδίκευση. Στη συνέχεια θα ασχοληθούμε με τον αλγόριθμο mix με βάσει τον κώδικα που παραθέσαμε πιο πάνω.

Το ερώτημα που τίθεται τώρα είναι τί πληροφορία θα είναι διαθέσιμη στο mix_1 όταν η παραγωγή του μεταγλωττιστή λαμβάνει χώρα, όπως φαίνεται παρακάτω. Παρουσιάζουμε συγκεκριμένα τα ορίσματα div_{mix} και div_{int} για να γίνει αντιληπτή πλήρως η σημασία της ερώτησης.

$compiler = [\![mix_1]\!]_L[mix_2, div_{mix}, [int, div_{int}]]$

Σε αυτή την διαδικασία λοιπόν, το mix_1 είναι ο ενεργός εξειδικευτής που εκτελείται με τρία ορίσματα. Το πρώτο όρισμα είναι ο κώδικας του mix_2 ο οποίος είναι ταυτόσημος με τον mix_1. Το δεύτερο όρισμα είναι η διαίρεση των μεταβλητών του mix_2. Το τρίτο όρισμα είναι αρχικές τιμές των στατικών μεταβλητών εισόδου του mix_2. Οι δύο από τις 3 μεταβλητές εισόδου του mix_2 είναι στατικές, συγκεκριμένα η program που η τιμή της είναι ο διερμηνέας int και η division που η τιμή της είναι η διαίρεση dim_{mix}. Δηλαδή στο mix_2 δίνουμε τον κώδικα του διερμηνέα και την διαίρεση των μεταβλητών του, αλλά όχι τις αρχικές τιμές των δεδομένων εισόδου του διερμηνέα.

Θυμόμαστε ότι το mix εφαρμοζόμενο σε έναν διερμηνέα και τον κώδικα ενός προγράμματος δίνει ένα εναπομένον πρόγραμμα. Όταν εκτελείται το $[\![mix_1]\!]_L[mix_2, div_{int}, int]$, μόνο ο διερμηνέας είναι διαθέσιμος στο mix_2, οπότε μπορεί να κάνει μόνο όσες ενέργειες εξαρτώνται από αυτόν και όχι από το αρχικό πρόγραμμα. Είναι σημαντικό για την αποδοτικότητα του παραγόμενου μεταγλωττιστή να μπορεί ο mix_2 να εκτελέσει κάποιους από τους υπολογισμούς, κατά την παραγωγή του μεταγλωττιστή.

Τώρα θα εξετάσουμε τις πιο σημαντικές μεταβλητές του mix_2 για να αποφανθούμε ποιες έχουν αρκετές πληροφορίες κατά την παραγωγή του μεταγλωττιστή ώστε να μπορέσουμε να την χαρακτηρίσουμε ως στατικές.

Εξ αρχής οι μεταβλητές program και division είναι στατικές. Οι μεταβλητές vs και vs_0 προορίζονται να περιέχουν τις τιμές μερικών παραμέτρων του διερμηνέα, αυτή η πληροφορία δεν είναι διαθέσιμη μέχρι να δοθεί και το αρχικό πρόγραμμα, άρα είναι δυναμικές. Η απαίτηση για ταιριαστή διαίρεση επιτάσσει οι μεταβλητές *pending*, *marked*, *code* και *residual* να χαρακτηριστούν δυναμικές. Αυτές οι μεταβλητές λοιπόν δεν θα ελαττωθούν από το mix_1 και θα εμφανίζονται στο εναπομένον πρόγραμμα που αντιστοιχεί στον διερμηνέα, δηλαδή στον μεταγλωττιστή.

Ας θεωρήσουμε τώρα τις γραμμές 5-7 στον αλγόριθμο της εξειδίκευσης. Η μεταβλητή pp παίρνει την τιμή της από το pending άρα είναι δυναμική. Η μεταβλητή bb παίρνει την τιμή της αναζητώντας το pp μέσα στο program. Παρόλο που το program είναι στατικό και στο bb ανατίθεται μια τιμή από αυτό, ο περιορισμός του ταιριάσματος μας αναγκάζει να κατατάξουμε το bb ως δυναμική μεταβλητή αφού και η pp είναι δυναμική. Τα αποτελέσματα από αυτή την ενέργεια δεν είναι θετικά. Η μεταβλητή *command* θα πρέπει κι αυτή να θεωρηθεί δυναμική και σαν συνέπεια με δυσκολία μπορεί να γίνει οποιοσδήποτε υπολογισμός κατά την στιγμή της παραγωγής του μεταγλωττιστή.

Η μεταβλητή *pp* λέμε ότι ανήκει σε μια φραγμένη στατική κατανομή, που σημαίνει ότι μπορούμε να θεωρήσουμε ότι λαμβάνει τιμές από ένα σύνολο με πεπερασμένο πλήθος στοιχείων, και επιπλέον αυτό το σύνολο μπορεί να υπολογιστεί στατικά. Εν προκειμένω η *pp* πρέπει να είναι μία από τις ετικέτες στον κώδικα του διερμηνέα που μας δίνει τη δυνατότητα να εφαρμόσουμε ένα προγραμματιστικό κόλπο, που σαν αποτέλεσμα θα έχει η bb και άρα και η command να γίνουν στατικές. Το κόλπο αυτό είναι πολύ συνηθισμένο και παρουσιάζεται σε επόμενο κεφάλαιο. Για την ώρα θα δεχτούμε χωρίς εξήγηση ότι οι bb και command είναι στατικές μεταβλητές.

3.12 Η δομή του παραγόμενου μεταγλωττιστή

Αν εξετάσουμε την δομή του παραγόμενου μεταγλωττιστή για δούμε ότι μοιάζει αρκετά με αυτή ενός μεταγλωττιστή γραμμένου στο χέρι χρησιμοποιώντας την παραδοσιακή μέθοδο της αναδρομικής κατάβασης (recusive descent). Έχουμε δει ήδη το παράδειγμα του τελικού κώδικα που προκύπτει από την εξειδίκευση του διερμηνέα, και η εξισώσεις του mix μας εξασφαλίζουν ότι ο παραγόμενος μεταγλωττιστής δουλεύει με ακριβώς τον ίδιο τρόπο. Αυτό που μας νοιάζει πλέον είναι η δομή και η επιδόσεις αυτού του μεταγλωττιστή.

Ακολουθεί ο κώδικας του μεταγλωττιστή που παίρνουμε από τον διερμηνέα της μηχανής Turing. Έχουμε κάνει ελάχιστες συντακτικές αλλαγές για να βελτιώσουμε την αναγνωσιμότητά του.

```
read(Q);
pending := { ('init, Q) };
marked := ∅;
while pending != '() do
begin
  Pick an element (pp, vs) ∈ pending and remove it;
  marked := marked ⋃ { (pp, vs) };

  case pp of

    init:
      Qtail := Q; (* vs = Q *)
      generate initializing code;
      while Qtail != '() do
11:   begin
```

```
        Instruction := hd(Qtail);
        Qtail := tl(Qtail);

        case Instruction of

          right:     code := extend (code,
                                     left := cons(firstsym(right), left),
                                     right := tl(right))

          left:      code := extend (code,
                                     right := cons(firstsym(left),right),
                                     left := tl(left))

          write s:   code := extend (code, right := cons(s, tl(right)))

          goto lbl:  Qtail := new_tail(lbl, Q);

          if s goto lbl:
                     pending := pending ⋃ ({('cont,Qtail)} \ marked);
                     pending := pending ⋃ ({('jump, lbl)} \ marked );
                     code    := extend (code, if s = firstsym(right)
                                              goto ('jump, lbl)
                                              else ('cont, Qtai));

          otherwise:
                     error
      end;

    cont: if Qtail != '() goto line 11 (* vs = Qtail *)

    jump: Qtail := new_tail(lbl, Q);
          if Qtail != '() goto line 11 (* vs = lbl *)

    otherwise: error;
  residual := extend(residual, code)
end;
```

Αυτός ο μεταγλωττιστής αντιπροσωπεύει ένα ενδιαφέρον μείγμα ανάμεσα στον μερικό αποτιμητή *mix* και τον διερμηνέα της μηχανής Turing. Ο εσωτερικός βρόχος while, αναπαριστά πιστά τον βρόχο του κώδικα του διερμηνέα. Οι συνθήκες των εκφράσεων που χειρίζονται την συντακτική ανάλυση προέρχονται απευθείας από τον διερμηνέα. Ο κώδικας ενδιάμεσα που δημιουργεί τις εκφράσεις προφανώς δεν είναι ίδιος με αυτόν του διερμηνέα αλλά η σχέση είναι στενή. Ο κώδικας αυτός είναι ακριβώς οι οδηγίες που θα είχε εκτελέσει ο διερμηνέας.

Ο εσωτερικός βρόχος που χειρίζεται τη σύνταξη και την παραγωγή κώδικα φαίνεται αναμενόμενος, εκτός ίσως από τις ενέργειες για την μεταγλώττιση των εντολών if. Αυτό διαφέρει από ένα χειρόγραφο μεταγλωττιστή που έχοντας γνώση για τα επόμενα σύμβολα (tokens) θα έκανε γραμμική σάρωση του κειμένου και ταυτόχρονα θα παρήγαγε κώδικα, ενώ στο τέλος τα ακολουθούσε το backpathing.

Από την άλλη βέβαια αναφερόμαστε σε έναν μεταγλωττιστή ο οποίος παράχθηκε αυτόματα από έναν διερμηνέα, και για αυτό τον λόγο έχει κληρονομήσει κάποια από τα χαρακτηριστικά του. Όπως ένας διερμηνέας δεν κάνει γραμμική σάρωση του κώδικα των προγραμμάτων, αλλά ακολουθεί την ροή του ελέγχου όπως αυτή προκύπτει με βάση τη σημασιολογία, έτσι δουλεύει

και ο μεταγλωττιστής που παράγεται από αυτό το διερμηνέα. Αν η ροή ελέγχου μπορεί να προδιαγραφεί μονάχα από τον κώδικα του προγράμματος, τότε μια γραμμική ακολουθία κώδικα παράγεται.

Όταν συναντάται μια εντολή if αυτό δεν είναι πλέον εφικτό να συμβεί εφόσον πρέπει να παραχθεί κώδικας και για τους δύο κλάδους της εντολής. Παρατηρούμε ότι ο μεταγλωττιστής χρησιμοποιεί τα σύνολα pending και marked για να ελέγχει ποια κομμάτια του αρχικού κώδικα έχει μεταγλωττίσει και ποια απομένουν για μεταγλώττιση. Μετά την μεταγλώττιση μιας εντολής if η μεταγλώττιση έχει δύο σημεία από τα οποία πρέπει να συνεχίσει. Το ένα, που θα εκτελεστεί αν η συνθήκη ελέγχου προκύψει να είναι ψευδής, χαρακτηρίζεται από το Qtail, ενώ το άλλο, όταν η συνθήκη ελέγχου είναι αληθής, χαρακτηρίζεται από το lbl, την ετικέτα προς την οποία γίνεται το άλμα υπό συνθήκη. Για αυτό το λόγο δύο ζευγάρια (cont, Qtail) και (jump, lbl) προστίθενται στο σύνολο pending, αν δεν βρίσκονται ήδη εκεί ούτε τα έχουμε ήδη επεξεργαστεί και τοποθετήσει στο σύνολο marked.

Ένα σημείο που χρειάζεται να εξηγήσουμε είναι το γιατί τα ζευγάρια (init, Q), (cont, Qtail) και (jump, lbl) λέμε πως είναι της μορφής (pp, vs). Αν κι εκ πρώτης όψεως δεν φαίνεται λογικό αφού το vs θα έπρεπε να περιέχει τις τιμές όλων των στατικών μεταβλητών του διερμηνέα, εντούτοις είναι σωστό γιατί η μια απλή ανάλυση για ενεργές και ανενεργές στατικές μεταβλητές, που θα την δούμε πιο κάτω, μας λέει ότι οι μόνες στατικές μεταβλητές του διερμηνέα είναι οι Q, Qtail και lbl.

3.13 Ανάλυση στιγμής δέσμευσης

Ένα ερώτημα που έχουμε αφήσει αναπάντητο μέχρι στιγμής είναι το πώς υπολογίζουμε με αυτοματοποιημένο τρόπο μία σωστή διαίρεση Στη συνέχεια παρουσιάζουμε διάφορες τεχνικές που μας επιτρέπουν να κάνουμε την καλύτερη δυνατή κατηγοριοποίηση των μεταβλητών του προγράμματός μας, ξέροντας εκ των προτέρων ποιες μεταβλητές εισόδου έχουν γνωστές τιμές.

3.13.1 Απλή ανάλυση στιγμής δέσμευσης

Θεωρώντας ότι η ίδια διαίρεση βρίσκει εφαρμογή σε κάθε σημείο του προγράμματος και αγνοώντας το ότι πρέπει η διαδικασία της εξειδίκευσης να τερματίζει είναι εύκολο να υπολογίσουμε την διαίρεση για όλες τις μεταβλητές του προγράμματος αν μας δοθεί η διαίρεση των μεταβλητών εισόδου. Η διαδικασία αυτή ονομάζεται ανάλυση της στιγμής δέσμευσης (binding time analysis) γιατί αποφασίζει από ποια στιγμή η τιμή μιας μεταβλητής μπορεί να υπολογιστεί, δηλαδή την στιγμή που μια γνωστή τιμή έχει δεσμευτεί σε μια μεταβλητή.

Ας ονομάσουμε τις μεταβλητές του προγράμματος $X_1, X_2, ..., X_N$, και ας θεωρήσουμε ότι η μεταβλητές εισόδου είναι οι πρώτες n, όπου $0 \leq n \leq N$. Υποθέτουμε ότι μας δίνουν τις στιγμές δέσμευσης $(\bar{b}_1, ..., \bar{b}_n)$ για τις μεταβλητές εισόδου, όπου $\bar{b}_j$ είναι είτε S δηλαδή στατική, είτε D δηλαδή δυναμική. Ο σκοπός μας τώρα είναι να βρούμε μια ταιριαστή διαίρεση για όλες τις μεταβλητές: $B = (b_1, ..., b_N)$ που ικανοποιεί τη σχέση $\bar{b}_i = D \Rightarrow b_i = D$ για $0 \leq i \leq n$, δηλαδή για τις μεταβλητές εισόδου. Αυτό μπορεί να γίνει βάση του παρακάτω αλγορίθμου:

- 1. Κατασκευάζουμε την αρχική διαίρεση $\bar{B} = (\bar{b}_1, ..., \bar{b}_n, S, ..., S)$ και θέτουμε $B = \bar{B}$
- 2. Αν το πρόγραμμα περιέχει μια ανάθεση X_k `= exp`, όπου το X_j εμφανίζεται στην έκφραση exp και $b_j = D$ στο B, τότε θέτουμε $b_k = D$ στο B.
- 3. Επαναλαμβάνουμε το βήμα 2 μέχρι το B να μην αλλάζει άλλο πια. Τότε ο αλγόριθμος τερματίζει και η B είναι μια ταιριαστή διαίρεση

3.13.2 Ανάλυση στιγμής δέσμευσης για την αυτοεφαρμογή

Για να είναι επιτυχημένη η αυτοεφαρμογή είναι κρίσιμο ένα προστάδιο που ονομάζουμε ανάλυση της στιγμής δέσμευσης. Η έξοδος από αυτή τη διαδικασία είναι μια διαίρεση, μια κατηγοριοποίηση κάθε μεταβλητής του προγράμματος ως στατική ή δυναμική. Ο μερικός αποτιμητής χρησιμοποιεί αυτή τη διαίρεση για να αποφασίσει ποια είναι τα στατικά μέρη του αρχικού προγράμματος από πριν, αντί να αναλύει το πρόγραμμα κατά τη διάρκεια της εξειδίκευσης. Αυτό έχει σαν αποτέλεσμα τα αποτελέσματα της εξειδίκευσης να είναι μικρότερα και αποδοτικότερα προγράμματα. Το σημαντικό σημείο είναι ότι τα στατικά μέρη του αρχικού προγράμματος αποφασίζονται πριν την εξειδίκευση και ότι ο μερικός αποτιμητής μπορεί να χρησιμοποιήσει αυτή την πληροφορία όταν εκτελείται. Το να παρέχεται έτοιμη η διαίρεση των μεταβλητών στον μερικό αποτιμητή είναι ο ευκολότερος τρόπος για να έχει γνώση αυτής της πληροφορίας η διαδικασία της εξειδίκευσης.

3.13.3 Φραγμένη στατική κατανομή

Συμβαίνει συχνά κατά την εξειδίκευση μία μεταβλητή να φαίνεται πως είναι δυναμική καθώς εξαρτάται άμεσα ή έμμεσα από δυναμικά δεδομένα εισόδου, αλλά να της ανατίθενται τιμές από ένα πεπερασμένο πλήθος τιμών. Σε μια τέτοια περίπτωση μια διαφορετική σύνταξη του προγράμματος μπορεί να δώσει πολύ καλύτερα αποτελέσματα σε ότι έχει να κάνει με την εξειδίκευση. Αυτή η διαφορετική σύνταξη, ή τροποποίηση του προγράμματος, η οποία δεν αλλοιώνει το αρχικό νόημα του προγράμματος αλλά έχει σαν αποτέλεσμα καλύτερα εναπομείναντα προγράμματα ονομάζεται βελτίωση της στιγμής δέσμευσης. Ο όρος βελτίωση αναφέρεται στον στόχο της τροποποίησης, δηλαδή στον να χαρακτηριστούν ως στατικές περισσότερες μεταβλητές και κατ' επέκταση εκφράσεις, κι έτσι να ελαττωθούν από τον μερικό αποτιμητή. Παρακάτω θα δείξουμε ένα κλασσικό παράδειγμα που εφαρμόζεται και στο ίδιο το mix, και το είχαμε αφήσει σε εκκρεμότητα από πριν.

Αναφερόμαστε στην επόμενη γραμμή από τον αλγόριθμο του mix, που αναζητά και βρίσκει ένα σημείο του προγράμματος με την ετικέτα pp:

```
bb := lookup (pp, program);
```

Θυμίζουμε ότι η μεταβλητή pp προέρχεται από το σύνολο pending και για αυτό έχει κατηγοριοποιηθεί ως δυναμική. Ξέρουμε όμως ότι η λειτουργία της lookup είναι να βρίσκει ένα βασικό μπλοκ μέσα στο πρόγραμμα με βάση την ετικέτα του. Το πρόγραμμα έχει πεπερασμένο μέγεθος, άρα και πεπερασμένο πλήθος από ετικέτες pp και για αυτό έχουμε την ευχέρεια να διατυπώσουμε διαφορετικά την προηγούμενη έκφραση. Υλοποιούμε την lookup με ως εξής:

```
pp' := pp0; (* first label (static) *)
while pp != pp' do
  pp' = next label after pp'; (* pp' remains static *)
  bb := basic block at label pp'; (* bb is static too *)
<computations involving pp>
```

Αυτό που συμβαίνει τώρα είναι ότι το mix συγκρίνει τη δυναμική μεταβλητή pp με όλες τις πιθανές τιμές που θεωρούμε ότι μπορεί να πάρει και παράγει κώδικα για κάθε κάθε δυνατό αποτέλεσμα της lookup $(bb_0, bb_1, ...)$. Το σημαντικό είναι ότι η επιλογή γίνεται τη στιγμή της εκτέλεσης, αλλά οι το εύρος των πιθανών τιμών είναι γνωστό από τη στιγμή της εξειδίκευσης.

Για παράδειγμα, στον μεταγλωττιστή που προκύπτει από την αυτοεφαρμογή του mix συναντάμε κώδικα που έχει την παρακάτω μορφή

```
case pp of
        pp0: <code specializes wirth respect to bb0>
        pp1: <code specializes wirth respect to bb1>
        ...
        ppn: <code specializes wirth respect to bbn>
```

```
    end case
```

Το κόλπο που κάναμε εδώ για να εκμεταλλευτούμε τις μεταβλητές με φραγμένη στατική κατανομή είναι απαραίτητο για να αποφύγουμε την περίπτωση της τετριμμένες αυτοεφαρμογής του *mix*. Σε μερικούς αποτιμητές που εξειδικεύουν ποιο περίπλοκες γλώσσες από την δική μας αυτή η τεχνική χρησιμοποιείται κατά κόρον σε πολλά σημεία, αν κι εμείς το χρησιμοποιήσαμε μόνο για ετικέτες

3.13.4 Διαίρεση ανά σημείο προγράμματος

Μέχρι τώρα έχουμε θεωρήσει πως ο σκοπός της ανάλυσης τις στιγμής δέσμευσης είναι να υπολογίσουμε μία διαίρεση που να είναι έγκυρη για όλα τα σημεία του προγράμματος. Για τα περισσότερα μικρά προγράμματα είναι μια αρκετά λογική απαίτηση να θέλουμε μια διαίρεση που να κατηγοριοποιεί τις μεταβλητές ολόκληρου του προγράμματος σε στατικές ή δυναμικές. Υπάρχουν παρ' όλα αυτά μερικές ενστάσεις σε αυτή την απλοποιημένη αντίληψη της ανάλυσης τις στιγμής δέσμευσης

Θεωρούμε το παρακάτω κομμάτι κώδικα και την αρχική διαίρεση των μεταβλητών εισόδου (S, D):

```
read (X Y);
init: X := X + 1;
      Y := Y - 1;
      goto cout:
cont: Y := 3;
next: ...
```

Προφανώς μια ταιριαστή και ενιαία διαίρεση θα ήταν η (S, D), αλλά κρίνοντας από το πρόγραμμα μια πιο λεπτομερής διαίρεση της μορφής init:(S, D), cont:(S, D), next:(S, S) θα ήταν επίσης ασφαλής. Θα ονομάζουμε μια τέτοια διαίρεση διαίρεση ανά σημείο του προγράμματος. Εν αντιθέσει με την απλούστερη περίπτωση ανάλυσης της στιγμής δέσμευσης, όπου υπολογίζουμε μια ενιαία διαίρεση, η ανάλυση για να υπολογίσουμε την διαίρεση ανά σημείο του προγράμματος οφείλει να λαμβάνει υπόψιν της τον έλεγχο της ροής του προγράμματος.

Ο αλγόριθμος *mix* χρειάζεται μια μικρή επέκταση για να μπορέσει να χειριστεί μια διαίρεση ανά σημείο προγράμματος. Αφενός η μεταβλητή εισόδου division θα πρέπει να αντικατασταθεί από μια λίστα division-list και αφετέρου κάθε φορά που παίρνουμε ένα σημείο του εξειδικευμένου προγράμματος (pp, vs) από το σύνολο pending, θα πρέπει να υπολογίζουμε και την ανάλογη διαίρεση `division = lookup(pp, division-list`. Πιο πριν απαιτήσαμε ότι για να έχουμε καλά αποτελέσματα από την αυτοεφαρμογή θα πρέπει η μεταβλητή division να είναι στατική. Από τη στιγμή που εδώ η μεταβλητή pp είναι στατική, η χρήση της διαίρεσης ανά σημείο προγράμματος δεν χαλάει τα αποτελέσματα της αυτοεφαρμογής.

3.13.5 Ενεργές και ανενεργές στατικές μεταβλητές

Η χρήση προστατικών γλωσσών προγραμματισμού εισάγει ένα σοβαρό πρόβλημα, την εξειδίκευση σε σχέση με ανενεργές στατικές μεταβλητές. Αυτό το πρόβλημα δεν παρουσιάζεται μόνο στην αυτοεφαρμογή, αλλά και κατά την εξειδίκευση κάθε προγράμματος που που έχει ένα σεβαστό μέγεθος. Ας θεωρήσουμε ένα σημείο του εξειδικευμένου προγράμματος (pp, vs) όπου κατά τα γνωστά το vs περιέχει τις τιμές εκείνων των μεταβλητών του προγράμματος που έχουν κατηγοριοποιηθεί ως στατικές. Μερικές από αυτές τις στατικές τιμές μπορεί να είναι τελείως άσχετες με το τους υπολογισμούς που γίνονται σε ένα σημείο του προγράμματος pp. Για ένα απλό παράδειγμα θα υποθέσουμε ότι έχουμε το ακόλουθο απόσπασμα κώδικα:

```
start: if <dynamic condition>
       then a := 1; <commands using a>; goto next;
```

```
      else a := 2; <commands using a>; goto next;
next: <commands not referring a>;
```

Σε ότι αφορά τον παραπάνω κώδικα η μεταβλητή a θεωρείτε στατική, άρα η τιμή της είναι μέσα στο vs, ακόμα κι αν η τιμή της είναι τελείως άχρηστη για τους υπόλοιπους υπολογισμούς. Το αποτέλεσμα είναι το σημείο του εξειδικευμένου προγράμματος (start, ...) να έχει δύο διαδόχους, το (next, ... 1 ...) και το (next, ... 2 ...) οι οποίοι προφανώς είναι ίδιοι, γιατί η τιμή της μεταβλητής a δεν έχει καμία σημασία πλέον.

Ευτυχώς αυτό το πρόβλημα μπορεί να αποφευχθεί χρησιμοποιώντας μια τεχνική που είναι γνωστή με το όνομα ανάλυση ενεργών μεταβλητών, που αποσκοπεί στο να αναγνωρίσει ποιες στατικές μεταβλητές μπορούν να επηρεάσουν μελλοντικούς υπολογισμούς ή τη ροή ελέγχου του προγράμματος και έτσι η μερική αποτίμηση να γίνει λαμβάνοντας υπόψιν της μόνο αυτές τις μεταβλητές. Η γενική ιδέα είναι ξεκινάμε υπολογίζοντας μια ενιαία διαίρεση και διαδοχικά να ανακατηγοριοποιούμε τις μη ενεργές μεταβλητές σε κάθε σημείο του προγράμματος ώστε να μετατρέψουμε την ενιαία διαίρεση, σε διαίρεση ανά σημείο του προγράμματος.

3.13.6 Πολυμεταβλητές διαιρέσεις

Ακόμα κι οι διαιρέσεις ανά σημείο του προγράμματος μπορούν να κατηγοριθούν ότι είναι περισσότερο αυστηρές από ότι χρειάζεται. Κάτι τέτοιο μπορεί να συμβεί όταν ο χαρακτηρισμός ως στατική ή δυναμική σε μια μεταβλητή, σε κάποιο σημείο του προγράμματος μπορεί να αποδοθεί ανάλογα με το πώς φτάσαμε σε αυτό το σημείο, δηλαδή από τις προηγούμενες καταστάσεις. Ας πάρουμε για παράδειγμα το ακόλουθο απόσπασμα όπου θεωρούμε αρχικά την διαίρεση (S, D).

```
read (X Y);
init: if Y > 42 goto xsd else dyn
dyn:  X := Y;
      goto xsd;
xsd: X := X + 17;
     ...
```

Μια ταιριαστή, ανά σημείο προγράμματος διαίρεση θα περιείχε σαν συμπέρασμα το xsd:(D, D). Μια πολυμεταβλητή διαίρεση αναθέτει σε κάθε ετικέτα ένα σύνολο από διαιρέσεις Για το παραπάνω πρόγραμμα, μια ταιριαστή, πολυμεταβλητή διαίρεση θα ήταν η: init:(S, D), dyn:(D, D), xsd:{(S, D), (D, D)}. Μια διαίρεση που δεν είναι πολυμεταβλητή θα την ονομάσουμε μονομεταβλητή.

Ο υπολογισμός μιας πολυμεταβλητής διαίρεσης δεν είναι δύσκολος, αλλά η απορία μας είναι πώς το *mix* μπορεί να εκμεταλλευτεί αυτή την παραπάνω πληροφορία που του παρέχουμε. Όταν θα παράγουμε κώδικα για το σημείο (pp, vs) θα πρέπει να διαλέξουμε την κατάλληλη διαίρεση για το pp. Εκτός κι αν το (pp, vs) είναι το αρχικό σημείο του εξειδικευμένου προγράμματος, υπάρχει ένα (pp', vs') τέτοιο ώστε $(pp, vs) \in successors((pp', vs'))$ και για αυτό η σωστή διαίρεση για το vs εξαρτάται από την διαίρεση που θεωρήσαμε για το vs'. Ένας τρόπος να κρατάμε αυτή την πληροφορία για τις διαιρέσεις είναι να επεκτείνουμε τον ορισμό του σημείου του εξειδικευμένου προγράμματος, έτσι ώστε να περιέχει και την διαίρεση βάση της οποίας έγινε η εξειδίκευσή του. Έπειτα εφαρμόζοντας την ίδια τεχνική που περιγράψαμε και πριν μπορούμε να σιγουρέψουμε ότι η μεταβλητή division θα είναι στατική ώστε να σιγουρέψουμε τα αποτελέσματα της αυτοεφαρμογής.

Μια διαφορετική αντιμετώπιση είναι να δημιουργήσουμε αντίγραφα των μπλοκ από τον κώδικα του αρχικού προγράμματος που είναι δυνατό να έχουν περισσότερες από μία διαιρέσεις, μέσα στον ίδιο τον κώδικα πριν από την εξειδίκευση, ώστε κάθε αντίγραφο να αντιστοιχεί σε μία μόνο διαίρεση Με αυτό τον τρόπο το παραπάνω πρόγραμμα μεταμορφώνεται ως εξής:

```
read (X Y)
init:  if X > 42 goto xsd-s else dyn
dyn:   X := Y;
       goto xsd-d;
xsd-s: X := X + 17;
       ...
xsd-d: X := X + 17;
       ...
```

Αυτή η δημιουργία αντιγράφων όπως την περιγράψαμε στην δεύτερη περίπτωση, στην πρώτη περίπτωση γίνεται αυτόματα κατά την εξειδίκευση, δηλαδή δεν πρέπει να θεωρήσουμε ότι ωθούμε την μερικό αποτιμητή να παράγει δύο φορές των ίδιο κώδικα. Μπορούμε να θεωρήσουμε αυτή τη μεταμόρφωση του κώδικα ότι είναι μια μορφή εξειδίκευσης του αρχικού προγράμματος με βάση τις πληροφορίες από ανάλυση της στιγμής δέσμευσης. Το βασικό πλεονέκτημα από αυτή την υπόθεση είναι ότι ο κυρίως αλγόριθμος του mix παραμένει όσο πιο απλός γίνεται.

3.13.7 Αποσφαλμάτωση της στιγμής δέσμευσης

Δύο προγράμματα που είναι σημασιολογικά, ακόμα και λειτουργικά, ισότιμα σε ότι έχει να κάνει με τη διάρκεια της εκτέλεσης και τον μέγεθος τους, είναι δυνατό να εξειδικεύονται με πολύ διαφορετικά, παράγονταν εναπομείναντα προγράμματα με μεγάλες διαφορές στις επιδόσεις, στο μέγεθος ακόμα και στη χρήση μνήμης κατά την εκτέλεση. Για αυτό όταν πρόκειται να γίνει μερική αποτίμηση των προγραμμάτων μας οφείλουμε να χρησιμοποιούμε κάποιο σωστό προγραμματιστικό ύφος που να επιτρέπει στα προγράμματά μας να εξειδικεύονται καλά. Στη συνέχεια θα αναφερθούμε σε μερικά στοιχεία αυτού που αποκαλέσαμε σωστό προγραμματιστικό ύφος.

Το σωστό ύφος εξαρτάται από το συγκεκριμένο μερικό αποτιμητή που χρησιμοποιείτε κάθε φορά, αλλά ένα κοινό χαρακτηριστικό είναι ότι η ανάλυση της στιγμής δέσμευσης πρέπει να περιπλέκεται όσο το δυνατόν λιγότερο. Μερικώς στατικά αντικείμενα είναι δυσκολότερο να τα χειριστούμε από ότι τα ολικώς στατικά, και η μια δυναμική επιλογή μεταξύ στατικών δεδομένων είναι δυσκολότερη από μια στατική επιλογή. Για παράδειγμα θεωρούμε τις δύο ισοδύναμες εκφράσεις

```
fun f1 x y = (x + 1) + y;
fun f2 x y = (x + y) + 1;
```

Αν η μεταβλητή x είναι στατική και η y δυναμική, ένας μερικός αποτιμητής θα μπορέσει να ελαττώσει το $x + 1$ στην f1 αλλά δεν θα μπορεί να κάνει το ίδιο για την f2. Για να μπορούσε να το κάνει αυτό θα έπρεπε να εφαρμόσει την επιμεριστική και την προσεταιριστική ιδιότητα, είτε κατά την εξειδίκευση είτε σε κάποιο προστάδιο αυτής.

Μια μεταμόρφωση του αρχικού προγράμματος η οποία διατηρεί τη σημασιολογία του αλλά το κάνει πιο κατάλληλο για εξειδίκευση ονομάζεται βελτίωση της στιγμής δέσμευσης. Οι βελτιώσεις της στιγμής δέσμευσης είναι μεταμορφώσεις του προγράμματος που εφαρμόζονται είτε αυτόματα είτε με το χέρι στα αρχικά προγράμματα πριν το στάδιο της εξειδίκευσης. Πρέπει να προσέξουμε ότι μια αλλαγή του `car (cons E1 E2)` σε `E1` δεν είναι αποδεκτή γιατί μπορεί να αλλάξει τη σημασιολογία του προγράμματος, για παράδειγμα ο υπολογισμός του E2 μπορεί να μην τερματίζει.

Προηγουμένως είδαμε μια μεταμόρφωση του κώδικα του αρχικού προγράμματος που έχει τα ίδια αποτελέσματα κατά την εξειδίκευση, με την πολυμεταβλητή διαίρεση Μια τέτοια μεταμόρφωση μπορεί να θεωρηθεί βελτίωση του χρόνου δέσμευσης. Είναι προφανές ότι επιθυμούμε να γίνονται αυτόματα όσες το δυνατό περισσότερες βελτιώσεις από κάποιο στάδιο προεπεξεργασίας του αρχικού κώδικα. Κάθε φορά που καταφέρνουμε να συστηματοποιήσουμε την παραγωγή μιας βελτίωσης μπορούμε να την εισάγουμε στον αλγόριθμο της εξειδίκευσης. Άρα

πολλές φορές είναι άσκοπο να χρησιμοποιούμε κάποιο προγραμματιστικό στυλ το οποίο ο μερικός αποτιμητής μας δεν έχει πλέον ανάγκη για να παράγει καλύτερα εναπομείναντα προγράμματα. Αυτό ενισχύει τα όσα λέγαμε πριν, ότι δηλαδή πρέπει να προγραμματίζουμε λαμβάνοντας υπόψιν μας τις ικανότητες των αλγόριθμων ανάλυσης της στιγμής δέσμευσης και εξειδίκευσης που πρόκειται να χρησιμοποιήσουμε.

Σε αυτό το σημείο μπορούμε να εισάγουμε την έννοια της αποσφαλμάτωσης της ανάλυσης της στιγμής δέσμευσης με την έννοια ότι πριν κάνουμε την εξειδίκευση εξασφαλίζουμε ότι το στάδιο της ανάλυσης έχει εξάγει τα καλύτερα δυνατά αποτελέσματα. Αυτό γίνεται αν παρατηρήσουμε τις στιγμές δέσμευσης του αρχικού προγράμματος και συνεχώς κάνουμε αλλαγές στον κώδικά του, ώστε κάθε φορά να παράγουμε πιο κατάλληλα δεδομένα. Η διαδικασία αυτή είναι κυκλική, δηλαδή είναι μια ακολουθία ανάλυση → αξιολόγηση → βελτίωση → ανάλυση → ..., όπου η αξιολόγησή μπορεί να περιέχει και την μερική αποτίμηση. Στο επόμενο κεφάλαιο παρουσιάζουμε ένα χρήσιμο παράδειγμα αυτής της διαδικασίας.

3.14 Εξειδικεύοντας έναν απλό αλγόριθμο ταιριάσματος συμβολοακολουθιών

Σε αυτό το κεφάλαιο θα παρουσιάσουμε πως η εξειδίκευση και η βελτίωση της στιγμής δέσμευσης θα μας βοηθήσουν να κατασκευάσουμε τον αλγόριθμο των Knuth, Morris και Pratt για το ταίριασμα συμβολοακολουθιών, ξεκινώντας από ένα απλοϊκό πρόγραμμα που κάνει αυτή τη δουλειά. Η αρχική έκδοση αυτού του παραδείγματος που αποτελεί κοινή αναφορά σε κάθε προσπάθεια επίδειξης των βελτιώσεων που μπορούν να γίνουν κατά τη στιγμή δέσμευσης έχει κατασκευαστεί από του Consel και Danvy.

Χρησιμοποιώντας την σύνταξη της Scheme, ένα απλοϊκό πρόγραμμα που ταιριάζει συμβολοακολουθίες είναι το παρακάτω.

```
(define (kmp p d)   (loop p d p d))
(define (loop p d pp dd)
  (cond
   ((null? p) 'yes)
   ((null? d) 'no)
   ((equal? (car p) (car d))
    (loop (cdr p) (cdr d) pp dd))
   (else (kmp pp (cdr dd)))))
```

Σαν είσοδο λαμβάνει ένα μοτίβο p και μια αρχική συμβολοακολουθία d και στην έξοδό του δίνει yes αν το μοτίβο υπάρχει στην συμβολοακολουθία ή no διαφορετικά. Στην μεταβλητή pp αποθηκεύεται ένα αντίγραφο του αρχικού μοτίβου και στην dd ένα αντίγραφο του υπόλοιπου της συμβολοακολουθίας από από το σημείο που άρχισε η τρέχουσα απόπειρα για ταίριασμα και μετά. Η χρονική πολυπλοκότητα του αλγόριθμου είναι $O(m \times n)$ όπου m το μήκος του μοτίβου και n το μήκος της αρχικής συμβολοακολουθίας.

Αν εξειδικεύσουμε τη συνάρτηση kmp σε σχέση με ένα στατικό μοτίβο p και μια δυναμική συμβολοακολουθία d, το εναπομένον πρόγραμμα θα έχει και πάλι πολυπλοκότητα $O(m \times n)$. Αν εκμεταλλευτούμε την πληροφορία που περιέχεται στον αλγόριθμο, ότι στην περίπτωση που το ταίριασμα αποτύχει τότε οι χαρακτήρες μέχρι το σημείο της διαφωνίας στα p και d είναι ίδιοι, τότε μπορούμε να επιδιώξουμε καλύτερα αποτελέσματα. Αυτό που θα κάνουμε είναι να συγκεντρώνουμε αυτή την πληροφορία, για παράδειγμα το κοινό στατικό πρόθεμα των p και d που είναι γνωστό σε κάποια δεδομένη στιγμή, και να συγκρίνουμε το μοτίβο πρώτα με αυτό το πρόθεμα και να συνεχίζουμε με το υπόλοιπο της συμβολοακολουθίας d μόνο αφού έχουμε ταιριάξει ολόκληρο το πρόθεμα. Η βελτίωση είναι δυνατή γιατί το ταίριασμα με ένα πρόθεμα είναι στατική έκφραση και μπορεί να αποτιμηθεί κατά το χρόνο εξειδίκευσης. Παρουσιάζουμε τη βελτιωμένη έκδοση:

```
(define (kmp p d)    (loop p d p '() '()))
(define (loop p d pp f ff)
  (cond
   ((null? p) 'yes)
   ((null? f)
    (cond
     ((null? d) 'no)
     ((equal? (car p) (car d))
      (loop (cdr p) (cdr d) pp '() (snoc ff (car p))))
     ((null? ff)
      (kmp pp (cdr d)))
     (else
      (loop pp d pp (cdr ff) (cdr ff)))))
   ((equal? (car p) (car f))
    (loop (cdr p) d pp (cdr f) ff))
   (else
    (loop pp d pp (cdr ff) (cdr ff)))))
```

Η μεταβλητή ff αντιστοιχεί στο πρόθεμα και είναι φανερό ότι ανήκει σε μια φραγμένη στατική κατανομή. Η μεταβλητή f παίζει τον ίδιο ρόλο όπως η d με την dd. Η συνάρτηση snoc προσθέτει ένα στοιχείο στο τέλος μιας λίστας, ο κώδικάς της είναι απλός και δεν παρουσιάζεται.

Ο χαρακτήρας που προκαλεί τη διαφωνία αγνοείται και για αυτό το λόγο αναμένουμε μερικούς περιττούς ελέγχους στους εναπομένον πρόγραμμα. Αυτοί μπορούν να αποφευχθούν με μια μικρή αλλαγή στο loop κάνοντάς το να εκμεταλλευτεί και αυτή την αρνητική πληροφορία. Ακολουθεί το τροποποιημένο πρόγραμμα που κάνει ακριβώς αυτό. Η μεταβλητή neg είναι μια λίστα από χαρακτήρες που δεν ταιριάζουν με τον πρώτο χαρακτήρα του d.

```
(define (kmp p d)
  (loop p d p '() '() '()))
(define (loop p d pp f ff neg)
  (cond
   ((null? p) 'yes)
   ((null? f)
    (cond
     ((and (not (null? neg)) (member (car p) neg))
      (if (null? ff)
          (kmp pp (cdr d))
          (loop pp d pp (cdr ff) (cdr ff) neg)))
     ((and (null? neg) (null? d)) 'no)
     ((equal? (car p) (car d))
      (loop (cdr p) (cdr d) pp '() (snoc ff (car p)) '()))
     ((null? ff)
      (kmp pp (cdr d)))
     (else
      (loop pp d pp (cdr ff) (cdr ff) (cons (car p) neg)))))
   ((equal? (car p) (car f))
    (loop (cdr p) d pp (cdr f) ff neg))
   (else
    (loop pp d pp (cdr ff) (cdr ff) neg))))
```

Στη συνέχει παραθέτουμε την εξειδικευμένη μορφή του παραπάνω προγράμματος για δεδομένο μοτίβο $p = (abab)$. Το εναπομένον αυτό πρόγραμμα έχει την ίδια δομή με αυτό που

προκύπτει με βάση την τεχνική των Knuth, Morris και Pratt για την παραγωγή αλγόριθμων που ταιριάζουμε συμβολοακολουθίες. Η πολυπλοκότητα του εξειδικευμένου προγράμματος είναι $O(n)$, όπου το n υποδηλώνει το μήκος της συμβολοακολουθίας που δίνεται στην είσοδο. Η επιτάχυνση που καταφέραμε είναι γραμμική, δηλαδή το εξειδικευμένο πρόγραμμα τρέχει ένα σταθερό αριθμό γρηγορότερα από τον αρχικό απλοϊκό αλγόριθμο.

```
(define (kmp-0 d_0)
  (define (loop-0-1 d_0)
    (cond ((null? d_0) 'no)
          ((equal? 'a (car d_0)) (loop-0-2 (cdr d_0)))
          (else (loop-0-1 (cdr d_0)))))
  (define (loop-0-2 d_0)
    (cond ((null? d_0) 'no)
          ((equal? 'b (car d_0))
           (let ((d_1 (cdr d_0)))
             (cond ((null? d_1) 'no)
                   ((equal? 'a (car d_1))
                    (let ((d_2 (cdr d_1)))
                      (cond ((null? d_2) 'no)
                            ((equal? 'b (car d_2))
                             (begin (cdr d_2) 'yes))
                            (else (loop-0-5 d_2)))))
                   (else (loop-0-1 (cdr d_1))))))
          (else (loop-0-5 d_0))))
  (define (loop-0-5 d_0)
    (if (equal? 'a (car d_0))
        (loop-0-2 (cdr d_0))
        (loop-0-1 (cdr d_0))))
  (loop-0-1 d_0)))
```

Κεφάλαιο 4

Ανάλυση στο πεδίο του χρόνου

Σε αυτό το κεφάλαιο θα εξετάσουμε την μερική αποτίμηση από την πλευρά του χρόνου. Δεν πρέπει να ξεχνάμε ότι μερική αποτίμηση έχει αξία επειδή μειώνει τον χρόνο για την ολοκλήρωση ενός υπολογισμού. Άλλωστε η βελτιστότητα είναι μια βασική απαίτηση από την μερική αποτίμηση.

Ορίζουμε ως χρόνο εκτέλεσης ενός προγράμματος p με αρχικά δεδομένα εισόδου d την τιμή $time_p(d)$, και εννοούμε ότι από τη στιγμή μου αρχίζει και μέχρι να ολοκληρωθεί η υπολογιστική διαδικασία περνάει χρόνος $time_p(d)$.

Από τον παραπάνω ορισμό προκύπτει ότι για την εκτέλεση ενός p γραμμένου στη γλώσσα S με τη βοήθεια ενός διερμηνέα int αυτής της γλώσσας και για αρχικά δεδομένα εισόδου d ο χρόνος εκτέλεσης είναι $time_{int}(p, d)$. Αυτό το χρόνο θα πρέπει να τον αντιπαραβάλουμε με τον άθροισμα $time_{mix}(int, p)+time_{int_p}(d)$ τον χρόνο δηλαδή που απαιτείται για την εξειδίκευση του διερμηνέα και την εκτέλεση του εναπομείναντος προγράμματος. Υπενθυμίζουμε ότι όπως και πριν είπαμε αν $time_{mix}(int, p) + time_{int_p}(d) \geq time_{int}(p, d) \geq time_{int_p}(d)$ η μερική αποτίμηση εξακολουθεί να παρουσιάζει ενδιαφέρον, αρκεί το εναπομένον πρόγραμμα να να φανεί χρήσιμο περισσότερες από μία φορές.

4.1 Ορισμός επιτάχυνσης

Υποθέτουμε πως έχουμε στη διάθεσή μας ένα πρόγραμμα p και δύο μεταβλητές εισόδου την στατική s, την οποία ξέρουμε κατά την εξειδίκευση και την δυναμική d. Θεωρούμε ότι $|s|$ και $|d|$ είναι τα μήκη αυτών των μεταβλητών και ότι το $time_{int}(s, d)$ αυξάνει όσο αυξάνουν αυτά τα μήκη. Εμείς αυτό που επιδιώκουμε είναι να βρούμε κάποια σχέση μεταξύ των χρόνων $t_p(s, d)$ και $t_{p_s}(d)$, της εκτέλεσης του αρχικού και του εξειδικευμένου προγράμματος αντίστοιχα. Η διαδικασία δεν είναι πολύ απλή γιατί δεν είναι τόσο εύκολο να συγκρίνουμε δύο συναρτήσεις με πολλά ορίσματα, συναρτήσει των ορισμάτων αυτών, αλλά και γιατί σε αρκετές περιπτώσεις ο χρόνος της μερικής αποτίμησης είναι σημαντικός.

Δεδομένου ενός προγράμματος p που δέχεται δύο μεταβλητές εισόδου την στατική s και την δυναμική d ορίζουμε την συνάρτηση επιτάχυνσης $su_s(d)$ ως εξής:

$$su_s(d) = \frac{t_p(s, d)}{t_{p_s}(d)}$$

και το όριο επιτάχυνσης $sb(s)$

$$sb(s) = \lim_{|d|\to\infty} su_s(d)$$

Ως αποτέλεσμα του παραπάνω βλέπουμε ότι ο μερικός αποτιμητής mix προκαλεί γραμμική επιτάχυνση των αρχικών προγραμμάτων, αρκεί το $sb(s)$ να είναι πεπερασμένο για όλα τα s.

4.2 Μέτρηση επιτάχυνσης

Είναι προφανές ότι η μερική αποτίμηση έχει οφέλη κάθε φορά που $su_s(d) > 1$ για όλα τα s και d ή αν το d μεταβάλλεται ταχύτερα από το s. Για να εκμεταλλευτούμε την δεύτερη περίπτωση

κατασκευάζουμε περισσότερα του ενός εναπομείναντα προγράμματα, εξειδικεύοντας κάθε φορά το αρχικό σε σχέση με τα διαφορετικά s. Αν γνωρίζουμε ότι ο ρυθμός μεταβολής του s είναι n φορές μικρότερος από τον αριθμό μεταβολής του d, και άρα αναμένουμε ότι για κάθε s θα υπάρξουν n διαφορετικά ζευγάρια εισόδου (s, d), τότε μπορούμε να πούμε ότι η μερική αποτίμηση μας ωφελεί αν

$$\frac{1}{n} \times t_{mix}(p, s) + t_{p_s}(d) < t_p(s, d)$$

Ο παραπάνω κανόνας έχει δύο σημαντικές περιπτώσεις:

- $n = 1$ Άρα στην ουσία το εναπομένον πρόγραμμα προορίζεται να εκτελεστεί μόνο μία φορά, οπότε στους υπολογισμούς μας υπεισέρχεται ολόκληρο το χρονικό κόστος της εξειδίκευσης.

- $n = \infty$ Δηλαδή το εναπομένον πρόγραμμα δεν πρόκειται να αλλάξει πότε, και ως έχει θα βρίσκει πάντοτε εφαρμογή, οπότε όσο και να διαρκεί η εξειδίκευση τελικά δεν μας επηρεάζει στην λήψη των αποφάσεών μας.

Τα παραπάνω ισχύουν αν μελετάμε την επιτάχυνση συναρτήσει του d για διαφορετικές τιμές του s. Αυτό σημαίνει ότι η συνάρτηση που μας δίνει την επιτάχυνση su_s είναι ανεξάρτητη του s. Το ίδιο συμβαίνει και στην περίπτωση που εξειδικεύουμε ένα διερμηνέα σε σχέση με κάποιο αρχικό πρόγραμμα. Υπάρχουν όμως και περιπτώσεις όπου η συνάρτηση της επιτάχυνσης εξαρτάται σημαντικά από το s.

Επιλογή διαφορετικού αλγορίθμου

Μπορούμε να πετύχουμε σημαντική επιτάχυνση κατά την εξειδίκευση αν αλλάξουμε τον αλγόριθμο του χρησιμοποιεί το πρόγραμμά μας. Μια τέτοια αλλαγή πιθανώς να μας ωφελούσε και χωρίς την μερική αποτίμηση, όμως αν προσέξουμε και εφαρμόσουμε τις κατάλληλες τεχνικές, τότε μπορούμε κερδίσουμε ακόμα μεγαλύτερη επιτάχυνση από την εξειδίκευση. Χαρακτηριστικό παράδειγμα είναι ο υπολογισμός των αριθμών Fibonacci όπου μετά τον υπολογισμό κάθε αριθμού μπορούμε να τον αποθηκεύουμε έτσι ώστε την επόμενη φορά να ανατρέχουμε σε κάποια δομή για να τον ανακτήσουμε χωρίς να χρειαστεί να τον υπολογίσουμε από την αρχή. Με αυτό τον τρόπο άλλωστε μειώνουμε την πολυπλοκότητα της διαδικασίας από εκθετική σε γραμμική.

Ο τετριμμένος μερικός αποτιμητής

Μία ακόμα περίπτωση είναι ο τετριμμένος μερικός αποτιμητής. Στην ουσία δεν πρόκειται για μερική αποτίμηση, αλλά για ανάθεση των αρχικών στατικών τιμών σε εσωτερικές μεταβλητές του εναπομείναντος προγράμματος. Δεν μπορούμε να έχουμε την απαίτηση για καλύτερες επιδόσεις από αυτή τη διαδικασία. Προφανώς ισχύει ότι $t_p(s, d) = t_{p_s}(d)$ για κάθε s, οπότε και $su_s(d) = 1$ πάντα.

Προσθετικός χρόνος εκτέλεσης

Αν ο χρόνος που χρειάζεται για την εκτέλεση ενός προγράμματός είναι μια συνάρτηση της μορφής $t_p(s, d) = f(|s|) + g(|d|)$, τότε η μερική αποτίμηση δεν μπορεί να προσφέρει μεγάλη επιτάχυνση στον υπολογισμό. Ακόμα και να μπορέσουμε να μειώσουμε όλες τις στατικές εκφράσεις και κάθε s θα ισχύει ότι:

$$sb(s) = \lim_{|d| \to \infty} su_s(d) = \lim_{|d| \to \infty} \frac{f(|s|) + g(|d|)}{constant + g(|d|)} = 1$$

Επιτάχυνση διερμηνέων

Κατά την εξειδίκευση ενός μεταγλωττιστή σε σχέση με ένα αρχικό πρόγραμμα ισχύει η παρακάτω σχέση για όλα τα d

$$\alpha_p \times t_p(d) \leq t_{int}(p, d)$$

Η τιμή της παραμέτρου α_p είναι εξαρτάται μόνο από το p και καθόλου από το d. Συνήθως έχει τη μορφή $\alpha_p = c + f(p)$ όπου το c είναι μια σταθερά που έχει να κάνει με την υλοποίηση του του διερμηνέα και το $f(p)$ με τον χρόνο που αναλώνεται στην αποθήκευση και ανάκτηση των μεταβλητών. Κατάλληλες δομές μπορούν να μας εξασφαλίσουν ότι το α_p θα μεταβάλλεται πολύ αργά σε σχέση με το $|p|$. Οι διερμηνείς λοιπόν παρουσιάζουν μεγάλο ενδιαφέρον ώστε να ασχοληθούμε και να καταβάλουμε ιδιαίτερη προσπάθεια για την εξειδίκευσή τους. Η μερική αποτίμηση των διερμηνέων μας εξασφαλίζει γραμμική επιτάχυνση και συνήθως $sb(p) \geq c$ μιας και όπως είδαμε το $su_s = c + f(p)$ εξαρτάται ελάχιστα από το p.

Ταίριασμα συμβολοακολουθιών

Τέλος συχνά συναντάμε και τις διαδικασίες ταιριάσματος συμβολοακολουθιών. Σε αυτές τις περιπτώσεις η μερική αποτίμηση ρίχνει την υπολογιστική πολυπλοκότητα από το $O(m \times n)$ στο $O(n)$, όπου m το μήκος του μοτίβου και n το μήκος της συμβολοακολουθίας. Κάτι αντίστοιχο συμβαίνει και στους χρόνους εκτέλεσης οπότε μπορούμε να πούμε πως το όριο της επιτάχυνσης για αυτό το πρόβλημα είναι $sb(s) = m = |s|$, το οποίο είναι και πάλι γραμμικό, αφού για κάθε δεδομένο s το $|s|$ είναι μια σταθερά.

4.3 Ανάλυση επιτάχυνσης

Το πόσο θα αυξήσει τις επιδόσεις ενός προγράμματος η μερική αποτίμηση δεν μπορούμε να το ξέρουμε εκ των προτέρων. Πρέπει να γίνει, να εκτελέσουμε το εναπομένον πρόγραμμα και μετά να βγάλουμε το πόρισμα. Όμως όπως είδαμε πιο πριν πρέπει να ξέρουμε αν αξίζει να γίνει η εξειδίκευση πριν πάρουμε την απόφαση να την κάνουμε. Το μόνο που μπορούμε να κάνουμε είναι να υπολογίσουμε μια εκτίμηση της επιτάχυνσης που θα μας προσφέρει το εναπομένον πρόγραμμα, αγνοώντας την δυναμική είσοδο, κάνοντας αναλυση του κώδικα του αρχικού προγράμματος.

Η ακριβής πρόβλεψη για την επιτάχυνση είναι άλυτο πρόβλημα για προφανείς λόγους που συναντήσαμε και στο προηγούμενο κεφάλαιο, κατασκευάζοντας τον αλγόριθμο του μερικού αποτιμητή. Ακόμα και η εκτίμηση της επιτάχυνσης είναι πολύπλοκη διαδικασία, όμως στη συνέχεια θα παρουσιάσουμε τις βασικές αρχές της ανάλυσης της επιτάχυνσης. Το αποτέλεσμα στο οποίο θα καταλήγουμε θα είναι της μορφής $[l, h]$ που θα σημαίνει πως το εναπομένον πρόγραμμα θα εκτελείται από l μέχρι h φορές ταχύτερα από ότι το αρχικό. Στην περίπτωση που $h = \infty$ θα λέμε ότι έχουμε απεριόριστα περιθώρια για αύξηση των επιδόσεων του αρχικού προγράμματος. Κάτι τέτοιο συμβαίνει για παράδειγμα όταν το αρχικό πρόγραμμα είναι ένας μοναδικός βρόχος και ο αριθμός των επαναλήψεων καθορίζεται από μια στατική έκφραση.

4.4 Ασφαλή διαστήματα επιτάχυνσης

Ένα διάστημα επιτάχυνσης $[u, v]$ θα λέμε ότι είναι ασφαλές για το πρόγραμμα p αν η επιτάχυνση $su_s(d)$ συγκλίνει σε κάποια τιμή μέσα σε αυτό το διάστημα όταν επιλέγουμε τα s και d ώστε $|p(s, d)| \to \infty$. Έχουμε τη δυνατότητα να εκφράσουμε και διαστήματα της μορφής $[u, u]$ για να δείξουμε πως για κάθε s η επιτάχυνση που προσφέρει η μερική αποτίμηση είναι σταθερή και ίση με u.

Ορίζουμε πως ένα διάστημα $[u, v]$ είναι ασφαλές για το p όταν για κάθε ακολουθία (s_i, d_i) : $|p(s_i, d_i)| \to \infty$ έπεται πως

$$\forall \epsilon > 0 : \exists k : \forall j > k : \frac{|p(s_j, d_j)|}{|p_{s_j}(d_j)|} \in [u - \epsilon, v + \epsilon]$$

4.5 Επιτάχυνση απλών βρόχων

Θεωρούμε ένα πρόγραμμα γραμμένο στην γλώσσα που περιγράψαμε στο προηγούμενο κεφάλαιο. Ένας βρόχος είναι μια ακολουθία σημείων του προγράμματος $pp_1 \to pp_2 \to ... \to pp_k$ στα οποία περνάει διαδοχικά ο έλεγχος του προγράμματος και ισχύει ότι $pp_1 = pp_k$. Υποθέτουμε δηλαδή ότι η τελευταία εντολή του μπλοκ bb_i είναι ένα υπό συνθήκη ή όχι άλμα προς το pp_{i+1} για $1 \leq i \leq k$. Ένας απλός βρόχος ορίζεται όπως και ο βρόχος με την επιπλέον προϋπόθεση ότι $pp_i \neq pp_j$ για όλα τα i, j τέτοια ώστε $1 \leq i < j \leq k$.

Για κάθε σημείο του προγράμματος pp_i ορίζουμε το κόστος $C(pp_i)$ ως το άθροισμα των χρόνων εκτέλεσης κάθε εντολής που ανήκει στο μπλοκ που δείχνει η ετικέτα pp_i. Ορίζουμε το $C_s(pp_i)$ για να αναφερθούμε στο άθροισμα μόνο των στατικών εκφράσεων του μπλοκ, και όμοια ορίζουμε το $C_d(pp_i)$ για τις δυναμικές μεταβλητές.

Έστω ότι $l = pp_1 \to pp_2 \to ... \to pp_k$ είναι ένας απλός βρόχος. Ορίζουμε την επιτάχυνση $SU(l)$ στο βρόχο l ως:

$$SU(l) = \begin{cases} \dfrac{C_s(l) + C_d(l)}{C_d(l)} & \text{if } C_d(l) \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

όπου $C_s(l) = \sum_{i=1}^{k-1} C_s(pp_i)$ και $C_d(l) = \sum_{i=1}^{k-1} C_d(pp_i)$. Προφανώς η επιτάχυνση $SU(l)$ είναι ανεξάρτητη από τις τιμές των μεταβλητών.

4.6 Εις βάθος ανάλυση

Αν θεωρήσουμε τώρα ότι από ένα πρόγραμμα p παίρνουμε το σύνολο $\mathcal{L}$ που περιέχει όλους τους απλούς βρόχους. Τότε μπορούμε να πούμε ότι η συνολική επιτάχυνση του προγράμματος προέρχεται από την επιτάχυνση αυτών των βρόχων και μόνο. Με δεδομένη την απαίτησή μας το πρόγραμμα να εκτελείται για ένα αρκετά μεγάλο χρονικό διάστημα είναι αναμενόμενο ότι ο χρόνος που θα σπαταληθεί στις εντολές που δεν περιέχονται σε κάποιο βρόχο είναι αμελητέος.

Υπολογίζουμε το $SU(l)$ για κάθε απλό βρόχο που υπάρχει στο σύνολο $\mathcal{L}$. Το ασφαλές διάστημα επιτάχυνσης για το συνολικό πρόγραμμα p είναι τότε το μικρότερο $[u, v]$ για το οποίο να ισχύει ότι $\forall l \in \mathcal{L} : SU(l) \in [u, v]$.

Θέλουμε τώρα να αποδείξουμε ότι η επιτάχυνση που προκύπτει από τους μη απλούς βρόχους βρίσκεται στο ασφαλές διάστημα που ορίσαμε πριν. Υποθέτουμε ότι έχουμε ένα βρόχο l που προκύπτει από τον συνδυασμό δύο απλών βρόχων l_1 και l_2, και έστω ότι το ασφαλές διάστημα που προκύπτει για τα l_1 και l_2 είναι το $[u, v]$. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι $SU(l_1) \leq SU(l_2) < \infty$ και ξέρουμε ότι $C_s(l) = C_s(l_1) + C_s(l_2)$ και $C_d(l) = C_d(l_1) + C_d(l_2)$, σύμφωνα με τους ορισμούς των C_s και C_d. Τότε μπορούμε να πούμε ότι

$$SU(l_1) \leq SU(l) = \frac{C_s(l) + C_d(l)}{C_d(l)} \leq SU(l_2)$$

και άρα στο όριο η επιτάχυνση του βρόχου l είναι φραγμένη από τις επιταχύνσεις των απλών βρόχων l_1 και l_2, που βρίσκονται στο ασφαλές διάστημα, οπότε είμαστε σίγουροι ότι το $[u, v]$ είναι ασφαλές και για τον βρόχο l.

Μέχρι εδώ αγνοήσαμε την επιτάχυνση που επιτυγχάνουμε εξειδικεύοντας τα τμήματα του προγράμματος που δεν βρίσκονται μέσα σε βρόχους. Η επιλογή μας αυτή είναι δικαιολογημένη

όπως είπαμε και πιο πάνω, όμως χρειάζεται μια μετατροπή για να καλύψει την περίπτωση της ανάλυσης της επιτάχυνση των προγραμμάτων χωρίς βρόχους.

Ακολουθεί ένα μικρό παράδειγμα για να κατανοήσουμε την μέθοδο που περιγράψαμε. Θεωρούμε το παρακάτω πρόγραμμα

```
read (m, n);
1: sum := n;
2: if (zero? m) goto 4 else 3;
3: sum := sum + 1; m := m - 1; goto 2;
4: return sum;
```

Μετράμε μία μονάδα χρόνου για κάθε εντολή για να απλοποιήσουμε το παράδειγμα. Στην πράξη κάθε εντολή έχει το δικό της βάρος και αυτό εξαρτάται εκτός από την υλοποίηση του διερμηνέα και από την γλώσσα στην οποία είναι γραμμένος. Ο μοναδικός απλός βρόχος στο πρόγραμμα αυτό είναι ο $2 \to 3 \to 2$ και έστω ότι η m είναι στατική μεταβλητή και η n δυναμική. Τότε και το sum είναι δυναμικό οπότε μέσα στο βρόχο έχουμε τρεις στατικές και μια δυναμική έκφραση, έτσι ο βρόχος επιδέχεται επιτάχυνσης 4, δηλαδή το ασφαλές διάστημα είναι το $[4, 4]$. Αυτό μας δίνει το δικαίωμα να λέμε ότι το συνολικό πρόγραμμα μπορεί να τρέξει 4 φορές γρηγορότερα αν το εξειδικεύσουμε σε σχέση με το m. Αυτό βέβαια με την προϋπόθεση ότι ο βρόχος θα επαναληφθεί πολλές φορές.

Ασφαλές διάστημα για όλο το p

Πρέπει να είμαστε σίγουροι πως αν από την ανάλυση της επιτάχυνσης του προγράμματος p προκύψει το διάστημα $[u, v]$ τότε αυτό είναι ασφαλές για το πρόγραμμα p. Έστω η ακολουθία c σημείων pp_i του προγράμματος p τα οποία επισκεπτόμαστε διαδοχικά κατά τον υπολογισμό $p(s, d)$:

$$c = pp_1 \to pp_2 \to ... \to pp_k$$

Από την ακολουθία c διαγράφουμε όλους τους απλούς βρόχους, δηλαδή αντικαθιστούμε κάθε υποακολουθία της μορφής $pp_{i-1} \to pp_i \to pp_{i+1} \to ... \to pp_{i+j} = pp_i$ με το $pp_{i-1} \to pp_i \to pp_{i+j+1}$. Αφού φτάσουμε στο σημείο που δεν υπάρχει άλλος βρόχος για να διαγράψουμε το c θα περιέχει εκείνα τα σημεία του προγράμματος από τα οποία ο έλεγχος περνάει μόνο μια φορά. Επειδή το αρχικό πλήθος των σημείων του προγράμματος είναι πεπερασμένο, και το πλήθος των στοιχείων στο c θα είναι πεπερασμένο. Αναφερόμαστε στα σημεία pp_i που έχουν απομείνει στο c με τον όρο NL και ορίζουμε πως $nlstat = \sum_{n \in NL} C_s(n)$ και $nldyn = \sum_{n \in NL} C_d(n)$. Υπολογίζουμε πως η επιτάχυνση συνολικά στο πρόγραμμα δίνεται από την σχέση

$$SU = \frac{C_s(\mathcal{L}) + nlstat + C_d(\mathcal{L}) + nldyn}{C_d(\mathcal{L}) + nldyn} = \frac{\frac{C_s(\mathcal{L}) + C_d(\mathcal{L})}{C_d(\mathcal{L})}}{\frac{C_d(\mathcal{L}) + nldyn}{C_d(\mathcal{L})}} + \frac{\frac{nlstat + nldyn}{nldyn}}{\frac{C_d(\mathcal{L}) + nldyn}{nldyn}}$$

Επιλέγουμε μια ακολουθία (s_i, d_i) τέτοια ώστε $|p(s_i, d_i)| \to \infty$ και εξετάζουμε τους όρους της παραπάνω σχέσης. Ο αριθμητής $\frac{nlstat + nldyn}{nldyn}$ είναι φραγμένος και για τον παρονομαστή έχουμε $\frac{C_d(\mathcal{L}) + nldyn}{nldyn} \to \infty$ (θεωρούμε ότι εκτελούνται υπολογισμοί μέσα στους βρόχους άρα $C_d(\mathcal{L}) \to \infty$) άρα συνολικά ο δεύτερος όρος τείνει στο μηδέν. Από την άλλη ο παρονομαστής $\frac{C_d(\mathcal{L}) + nldyn}{C_d(\mathcal{L})} \to 1$ κι έτσι τελικά

$$SU \to \frac{C_s(\mathcal{L}) + C_d(\mathcal{L})}{C_d(\mathcal{L})}$$

Αφού το $\mathcal{L}$ περιέχει όλους τους απλούς βρόχους, έχουμε εξασφαλίσει από πριν ότι είναι $\frac{C_s(\mathcal{L}) + C_d(\mathcal{L})}{C_d(\mathcal{L})} \in [u, v]$, οπότε αποδείξαμε ότι στο διάστημα που βρίσκουμε πως είναι ασφαλές για όλους τους απλούς βρόχους του προγράμματος p, είναι ασφαλές και για όλο το πρόγραμμα p.

4.7 Βελτιστότητα του *mix*

Τα προγράμματα που εκτελούνται μέσω διερμηνέα είναι κατά κανόνα πιο χρονοβόρα από αυτά που μεταγλωττίζονται και εκτελούνται με την νέα τους μορφή ή που εκτελούνται απευθείας, με την έννοια ότι τα διερμηνεύει ο υπολογιστής. Ξέρουμε ότι η διαφορά είναι τέτοια που να αξίζει να πεδευτούμε για την κατασκευή ενός μερικού αποτιμητή με απότερο στόχο να τα μεταγλωττίσουμε. Όμως πρέπει να βρούμε και κάποιον τρόπο να επιβεβαιώνουμε ότι ο μερικός αποτιμητής που κατασκευάζουμε είναι αρκετά καλός στην δουλειά του, δηλαδή όντως τα εναπομείναντα προγράμματα είναι ταχύτερα των αρχικών. Αυτό που ζητάμε είναι η εξειδίκευση να αφαιρεί όλη την περιττή υπολογιστική πολυπλοκότητα λόγο διερμήνευσης και ο τρόπος για να ελέγχουμε ότι κάτι τέτοιο γίνεται είναι με την χρήση ενός αυτοδιερμηνέα sint. Όπως και πριν αναμένουμε ότι ο χρόνος εκτέλεσης του sint θα πρέπει να είναι περίπου $\alpha_p \times t_p(d)$ και το α_p πρέπει να είναι τέτοιο που να αξίζει να γίνει εξειδίκευση.

Θυμίζουμε πως για την περίπτωση του αυτοδιερμηνέα ισχύει η σχέση

$$[\![p]\!](d) = [\![sint]\!](p, d) = [\![[\![mix]\!](sint, p)]\!](d)$$

έτσι το $p' = [[\![mix]\!](sint, p)$ είναι ισότιμο του p. Επόμενο είναι λοιπόν ότι αν το mix κάνει αρκετά καλά τη δουλειά του, το p' θα είναι τουλάχιστον όσο αποδοτικό είναι και το p. Αν το πετύχουμε αυτό θα σημαίνει ότι το mix αφαίρεσε όλους του περιττούς υπολογισμούς από την διερμηνεία του $sint$, με άλλα λόγια $\alpha_p = 1$. Με άλλα λόγια το mix είναι βέλτιστο αν δεδομένου ενός αυτοδιερμηνέα $sint$

$$\forall p. \exists p' = [\![mix]\!](sint, p). \forall d. t'_p(d) \leq t_p(d)$$

Το παραπάνω κριτήριο δεν είναι τόσο αυστηρό όσο φαίνεται, καθώς στην πράξη έχουμε κατασκευάσει πολλά mix τα οποία το ικανοποιούν. Ακόμα να τονίσουμε πως αν και προφανώς ένα mix που θα έκανε τετριμμένη αποτίμηση ικανοποιεί την συνθήκη, δεν μπορούμε να πούμε πως είναι βέλτιστο.

Κεφάλαιο 5

Online και offline μερική αποτίμηση

Σε αυτό το κεφάλαιο μας απασχολούν οι ορισμοί, η χρήση και η σύγκριση των online και offline τεχνικών μερικής αποτίμησης. Οι πρώτοι μερικοί αποτιμητές ήταν online αλλά οι offline τεχνικές φαίνεται να είναι απαραίτητες για μπορέσουμε να επιτύχουμε την αυτοεφαρμογή και την παραγωγή γεννητόρων προγραμμάτων. όπως είναι οι μεταγλωττιστές. Στην ανάπτυξη του κεφαλαίου θα γίνουν φανερά τα πλεονέκτήματα της κάθε αντιμετώπισης για την μερική αποτίμηση, ενώ προς το τέλος θα περιγράψουμε μερικές ιδέες για τον συνδιασμό των καλύτερων χαρακτηριστικών και από τους δύο κόσμους.

5.1 Λήψη αποφάσεων σε ένα προστάδιο

Ο μερικός αποτιμητής πρέπει να λάβει μια απόφαση για το τί θα πράξει σε κάθε μία από τις παρακάτω περιπτώσεις

- Για κάθε τελεστή (if, +, cons, ...) αν μπορεί και πρέπει να τον ελαττώσει την στιγμή της εξειδίκευσης ή να παράγει κώδικα για το εναπομένον πρόγραμμα.

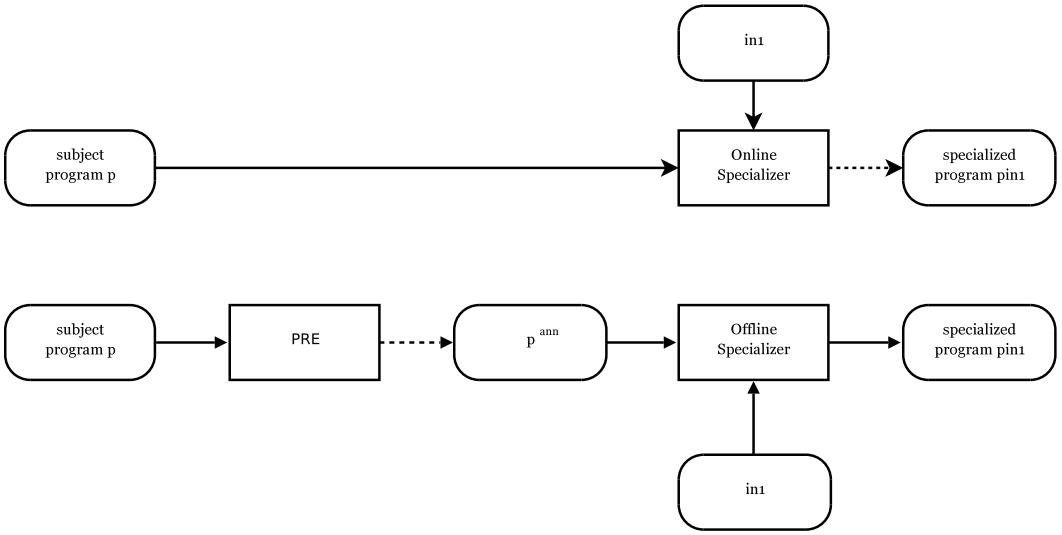
- Για κάθε μεταβλητή σε κάθε σημείο του προγράμματος αν μπορεί και πρέπει να την θεωρήσει στατική ή αν είναι δυναμική.

- Για κάθε κλήση συνάρτησης ή άλμα αν μπορεί να τυλίξει την μετάβαση της κατάστασης ή θα πρέπει να παράγει κώδικα.

Οι offline μερικοί αποτιμητές βασίζονται σε ένα προστάδιο που περιλαμβάνει την ανάλυση της στιγμής δέσμευσης για να λύσει το πρόβλημα του ότι πρέπει να διαλέξει τις σωστές ενέργειες, ανεξάρτητα από τις τιμές που έχουν οι στατικές μεταβλητές. Στο σχήμα που ακολουθεί βλέπουμε τη ροή που ακολουθεί η offline και η online μερική αποτίμηση.

Μέχρι στιγμής έχουμε περιγράψει offline μερικούς αποτιμητές και μόνο. Οι στατικοί και οι δυναμικοί υπολογισμοί διακρίνονται από την διαίρεση των μεταβλητών (ή από επισημειώσεις του αρχικού κώδικα, που είναι ακριβώς το ίδιο πράγμα με την την διαίρεση μόνο που εκφράζεται με διαφορετικό τρόπο. Στο εξής όπου συναντάμε το p_{ann} θα καταλαβαίνουμε ότι στη διάθεσή μας έχουμε ένα πρόγραμμα και την διαίρεση των μεταβλητών του.) που δημιουργήθηκε ανεξάρτητα από το ποιες είναι οι τιμές των στατικών μεταβλητών. Το τύλιγμα των μεταβάσεων εφαρμόζεται παντού εκτός από τους κλάδους μιας εντολής άλματος που ελέγχεται από κάποια δυναμική έκφραση.

Η online μερική αποτίμηση χαρακτηρίζεται από το γεγονός ότι δεν προϋπολογίζεται κάποια διαίρεση των μεταβλητών και όλες οι αποφάσεις λαμβάνονται κατά τη στιγμή της αποτίμησης όπου ταυτόχρονα είναι γνωστές και οι τιμές των στατικών μεταβλητών. Η αντιμετώπιση αυτή αν και εκ πρώτης όψεως φαίνεται λογικότερη και αποδοτικότερη στην πράξη συναντάει κάποια σημαντικά προβλήματα οπότε όπως γίνεται πάντα αναγκαζόμαστε να εφαρμόζουμε υβριδικές λύσεις.



Σχήμα 5.1: Online και Offline μερική αποτίμηση

5.2 Online και offline μείωση των εκφράσεων

Σε αυτό το κεφάλαιο θα συγκρίνουμε την μείωση των εκφράσεων όπως γίνεται από τους online και από τους offline μερικούς αποτιμητές. Θα αναφερθούμε σε τυπικούς αλγόριθμους online και offline μερικής αποτίμησης και θα δώσουμε παραδείγματα των πλεονεκτημάτων της κάθε μεθόδου. Υπενθυμίζουμε ότι η εξειδίκευση κατά την offline μεθοδολογία γίνεται στο επισημειωμένο p_{ann} ενώ κατά την online στο απλό αρχικό πρόγραμμα p. Στο εξής θα χρησιμοποιούμε τα ονόματα OffPE και OnPE για να αναφερόμαστε στον offline και τον online μερικό αποτιμητή αντίστοιχα.

Το επισημειωμένο πρόγραμμα p_{ann} εξασφαλίζει στον OffPE ότι μπορεί να διακρίνει άμεσα τί προσφέρεται για ελάττωση και τί όχι. Από την άλλη αυτή η πληροφορία απουσιάζει από το αρχικό πρόγραμμα που τροφοδοτούμε στον OnPE και για αυτό πρέπει να βρούμε ένα τρόπο να ξεχωρίζει αν κάποια έκφραση που συναντάει στο πρόγραμμα πρέπει να την ελαττώσει ή όχι. Η ιδέα που βρήκαμε για να λύσουμε αυτό το πρόβλημα είναι ο OnPE να προσθέτει μια χαρακτηριστική ένδειξη (tag) στα δεδομένα που επεξεργάζεται.

5.3 Online μερική αποτίμηση

Τα βασικά πλεονεκτήματα των online μεθόδων εξειδίκευσης προέρχονται από το γεγονός ότι δεν κάνουμε καμιά εκτίμηση εκ των προτέρων για το ποια δεδομένα και εκφράσεις θα είναι στατικά και ποια δυναμικά.

Μια ασφαλής ανάλυση της στιγμής δέσμευσης κατά την offline αποτίμηση πάντοτε παράγει μια ταιριαστή διαίρεση που χαρακτηρίζει στατικές αρκετές πληροφορίες ώστε να μπορούν να γίνουν οι υπολογισμοί. Είδαμε ότι η ανάλυση αυτή μπορεί να ενισχυθεί και με διάφορες βελτιώσεις και την αποσφαλμάτωση της διαδικασίας, αλλά σε κάθε περίπτωση η ανάλυση της στιγμής δέσμευσης οφείλει να λαμβάνει υπόψιν της τη χειρότερη περίπτωση. Για αυτό το λόγο, για να εξασφαλίσει τον τερματισμό της αποτίμησης μεταξύ άλλων, μερικές εκφράσεις θα κατηγοριοποιηθούν ως δυναμικές ακόμα κι αν μερικές φορές θα μπορούσαν να αποτιμηθούν την στιγμή της εξειδίκευσης.

Οι online μερικοί αποτιμητές έχουν περισσότερες πληροφορίες στην διάθεσή τους για να εξετάσουν την στατικότητα κατά τη στιγμή της εξειδίκευσης, και μπορούν να τις εκμεταλλευτούν για να αποφανθούν πιο καλά για την στιγμή δέσμευσης. Με άλλα λόγια οι online μερική αποτίμηση μπορεί να κάνει στατικούς υπολογισμούς τη στιγμή που, τους ίδιου υπολο-

γισμούς, ένας offline αλγόριθμος θα τους είχε αποκλείσει ως δυναμικούς. Στη συνέχεια θα παρουσιάσουμε μερικά παραδείγματα για να γίνει φανερό αυτό που λέμε.

Φυσικά τα μεγαλύτερα περιθώρια για στατικούς υπολογισμούς περιέχουν πάντα τον κίνδυνο ότι μπορεί η διαδικασία της εξειδίκευσης να ξεπεράσει τα όρια και να μην τερματίσει ποτέ. Για αυτό τον λόγο οι OnPE χρησιμοποιούν συχνά πιο συντηρητικές μεθόδους ή πολύ πιο προηγμένες τεχνικές για ξεπεράσουν το πρόβλημα της ατέρμονης μείωσης των καταστάσεων. Αυτό είναι μειονέκτημα των των online αλγορίθμων καθώς στον offline κόσμο το πρόβλημα έχει αντιμετωπιστεί και επιλυθεί με τη χρήση της ανάλυσης της στιγμής δέσμευσης, αν κι αυτό το στάδιο θέτει ένα όριο χαμηλότερο από το βέλτιστο δυνατό για το πόσο αποδοτική μπορεί να είναι η μερική αποτίμηση.

Συνθήκες με μικτή στιγμή δέσμευσης

Σε μια μονομεταβλητή ανάλυση της στιγμής δέσμευσης μια έκφραση χαρακτηρίζεται ως πάντοτε στατική ή πάντοτε δυναμική. Αν όμως μια συνάρτηση καλείται πότε με στατικά και πότε με δυναμικά δεδομένα, αναγκαζόμαστε να τη θεωρήσουμε δυναμική έκφραση αν και είναι εμφανές ότι θα μπορούσαμε να επωφεληθούμε από την επιπλέον πληροφόρηση που διαθέτουμε. Οι πολυμεταβλητές αναλύσεις της στιγμής δέσμευσης λύνουν αυτό το πρόβλημα για τις offline τεχνικές, αλλά και πάλι δεν μας επιτρέπουν να έχουμε κάποια κατηγορία μεταβλητών πέρα από τις στατικές και τις δυναμικές.

Ας σκεφτούμε με βάση την έκφραση `e = (if e1 e2 e3)` όπου οι e1 και e2 είναι στατικές εκφράσεις και το e3 δυναμική. Η ανάλυση θα κατηγοριοποιήσει το e ως δυναμικό. Ένας online έλεγχος όμως θα ανακάλυπτε ότι το e είναι στατικό αν το e1 αποτιμάται σε true. Αν με τη σειρά της η έκφραση e ήταν όρισμα στην κλήση μιας συνάρτησης f, ο OnPE θα τύλιγε την κλήση της συνάρτησης κάθε φορά που το e1 θα ήταν true αλλά ακόμα και μια πολυμεταβλητή ανάλυση θα ήταν δύσκολο να κάνει μια τέτοια πρόβλεψη. Στην έκφραση `(f (if e1 e2 e3))` η πολυμεταβλητή ανάλυση δεν μπορεί να δώσει τα αναμενόμενα αποτελέσματα, γιατί το όρισμα έχει ήδη κατηγοριοποιηθεί δυναμικό. Βέβαια μια μεταμόρφωση της έκφρασης στην ισοδύναμη `(if e1 (f e2) (f e3))` θα οδηγούσε την πολυμεταβλητή ανάλυση στο επιθυμητό αποτέλεσμα.

Δυναμικές συνθήκες με στατικά αποτελέσματα

Θεωρούμε μια έκφραση με συνθήκη `e = (if e1 e2 e3)` και έστω ότι η έκφραση e1 είναι δυναμική. Τότε ολόκληρη η έκφραση είναι δυναμική και πρέπει να παραχθεί κώδικας για το εναπομένον πρόγραμμα. Γενικά ούτε οι online ούτε οι offline τεχνικές έχουν κάποιο τρόπο να συμπεράνουν κάτι για μια τέτοια έκφραση με συνθήκη Παρ' όλα αυτά ο OnPE μπορεί να κάνει κάτι παραπάνω στην περίπτωση που οι εκφράσεις e2 και e3 είναι στατικές και έχουν κοινά χαρακτηριστικά. Κάτι ανάλογο γίνεται και για τις κλήσεις συναρτήσεων όπου είναι δυνατό η τιμή επιστροφής να είναι στατική μεταβλητή ακόμα κι αν η κλήση έγινε με δυναμικά ορίσματα. Όταν έχουμε αναδρομικές κλήσεις μιας συναρτήσεις φαντάζει λογικό ο μερικός αποτιμητής να μη μπει στη διαδικασία να τυλίξει μεταβάσεις λόγο του κινδύνου μη τερματισμού. Την περίπτωση όμως που οι αναδρομικές κλείσεις έχουν σαν αποτέλεσμα την ενημέρωση και μόνο μιας δομής, για την αποτίμηση μιας έκφρασης από ένα διερμηνέα για παράδειγμα, οπότε και το ενδεχόμενο μη τερματισμού δεν υπάρχει, ο OnPE είναι σε θέση να την ανίχνευση και να τυλίξει τις ανάλογες μεταβάσεις στο εναπομένον πρόγραμμα.

Αυτό που ισχύει στην περίπτωση του OffPE είναι ότι το πρόβλημα ξεκινάει από την ανάλυση της στιγμής δέσμευσης, οπότε με αποσφαλμάτωσή της μπορούμε να κερδίσουμε λίγο ακόμα σε επιδόσεις.

Σε επόμενο κεφάλαιο θα δούμε ποιο αναλυτικά το θέμα της αυτοεφαρμογής με σκοπό την παραγωγή μεταγλωττιστών και γιατί σε αυτό τον τομέα κυριαρχεί η offline μερική αποτίμηση. Συνοπτικά σαν μια εισαγωγή μπορούμε να πούμε ότι η offline μερική αποτίμηση έκανε την αυτοεφαρμογή εφικτή γιατί η online παρουσιάζει το πρόβλημα με τον μη τερματισμό. Κάθε παραγόμενος μεταγλωττιστής από αυτή τι διαδικασία όμως έχει ένα πρόβλημα. Δεν κάνει αρκετές βελτιστοποιήσεις κατά την μεταγλώττιση όπως για παράδειγμα θα ήταν το ξεδίπλωμα των σταθερών στο παραγόμενο πρόγραμμα. Αυτό συμβαίνει γιατί με βάση την ροή της OffPE πριν την μετατροπή του διερμηνέα σε μεταγλωττιστή λόγω της αυτοεφαρμογής όλες οι μεταβλητές του διερμηνέα έχουν κατηγοριοποιηθεί σε στατικές και δυναμικές, δηλαδή η απόφαση για το τι θα υπολογιστεί κατά την εξειδίκευση και τι κατά την εκτέλεση του παραγόμενου μεταγλωττιστή έχει παρθεί από την αρχή. Αυτή η κατηγοριοποίηση είναι σταθερή και προφανώς ανεξάρτητη του αρχικού προγράμματος που θα μεταγλωττίσει ο μεταγλωττιστής που παράγεται.

Από την άλλη πλευρά ακόμα και να αντιμετωπίσουμε το πρόβλημα με τον τερματισμό του OnPE με ευφυείς τεχνικές υπάρχει ένα επιπλέον εμπόδιο που καθιστά την online μερική αποτίμηση ακατάλληλη για την παραγωγή μεταγλωττιστών. Οι μεταγλωττιστές που προκύπτουν από την εξειδίκευση είναι μεγάλοι σε όγκο και χαρακτηριστικά αργοί. Ένας συνδυασμός των online και offline τεχνικών ίσως οδηγούσε σε γρήγορους μεταγλωττιστές που να είχαν την ικανότητα να βελτιστοποιούν το παραγόμενο πρόγραμμα ανάλογα με το αρχικό. Οι Ruf και Weise έχουν πάρει ένα διαφορετικό δρόμο όμως και προσπαθούν χρησιμοποιώντας ένα ισχυρό online μερικό αποτιμητή να εξειδικεύσουν έναν ασθενέστερο σε σχέση με τον διερμηνέα για να πάρουν τελικά τον μεταγλωττιστή. Αυτή είναι και η πρώτη αναφορά μας σε μια μορφή της δεύτερης προβολή του Futamura, όπου οι δύο λειτουργικοί ρόλοι που παίζει το mix δεν εκπληρώνονται από το ίδιο πρόγραμμα.

5.4 Offline μερική αποτίμηση

Η offline μερική αποτίμηση εφευρέθηκε το 1984 και έκανε την αυτοεφαρμογή εφικτή στην πράξη. Από το 1971 ο Futamura είχε σκεφτεί τις εφαρμογές της εξειδίκευσης του μερικού αποτιμητή για να παράγει μεταγλωττιστές, αλλά τα μέσα, οι περιορισμοί μνήμης και αποθηκευτικού χώρου σε συνδυασμό με την έλλειψη της κατάλληλης τεχνικής δεν είχαν βοηθήσει την έρευνα να αποφέρει καρπούς. Αργότερα στο κεφάλαιο θα δείξουμε πως οι offline τεχνικές παράγουν ένα μικρό και αποδοτικό μεταγλωττιστή, εδώ όμως θα περιοριστούμε στην παρουσίαση των προτερημάτων των offline μερικών αποτιμητών.

Το συστατικό του OffPE που εκτελεί την ελάττωση του επισημιωμένου προγράμματος καλείται πυρήνας της εξειδίκευσης. Το άλλο σημαντικό συστατικό της offline μερικής αποτίμησης είναι η ανάλυση της στιγμής δέσμευσης. Η διάσπαση της μερικής αποτίμησης σε δύο τελείως διαχωρισμένες φάσεις, στο στάδιο σχεδιασμού και στο στάδιο υλοποίησης, έχει αποδειχτεί πολύ χρήσιμο στην υ λοποίηση αρκετών μερικών αποτιμητών.

Έχουν γίνει πολλές έρευνες που στοχεύουν το ένα από τα δύο στάδια κάθε φορά και αυτό έχει συμβάλει στο να κατανοήσουμε καλύτερα τα κεντρικά προβλήματα της μερικής αποτίμησης. Ένας πυρήνας εξειδίκευσης μπορεί να δοκιμαστεί χωρίς το στάδιο της ανάλυσης της στιγμής δέσμευσης, παρέχοντάς του τις απαραίτητες επισημειώσεις με το χέρι και αντίστοιχα τα αποτελέσματα της ανάλυσης μπορούμε να τα αξιολογήσουμε κοιτώντας το επισημειωμένο πρόγραμμα.

Η ανάπτυξη των τεχνικών για την ανάλυση της στιγμής της δέσμευσης είναι μεγάλη και σε συνδυασμό με την διακριτή προσέγγιση της offline μερικής αποτίμησης, το σύνολο ανάλυση και εξειδίκευση έχει καταστεί αποδοτικότερο από την online μερική αποτίμηση. Το γιατί είναι προφανές αν αναλογιστούμε ότι κατά την online μερική αποτίμηση γίνονται όχι μόνο περισσότεροι έλεγχοι αλλά υπάρχει και η επιπλέον πολυπλοκότητα για την ανάθεση χαρακτηριστικών στις τιμές.

Η εξειδίκευση του μερικού αποτιμητή μπορεί και επιταχύνει την διαδικασία της μερικής αποτίμησης, κατά τον ίδιο τρόπο που η μεταγλώττιση και εκτέλεση του παραγόμενου προγράμματος είναι ταχύτερη από την διερμηνεία. Αν ο εξειδικευμένος μερικός αποτιμητής χρησιμοποιηθεί πολλές φορές μάλιστα, τα κέρδη είναι σημαντικά. Η εξειδίκευση σε πολλά στάδια φαίνεται στον παρακάτω πίνακα

1.	p_{ann}	:=	$[\![BTA]\!](p)$	Preprocessing time
2.	$p{-}gen$	:=	$[\![mix]\!](SPEC, p_{ann})$	
3.	p_{in1}	:=	$[\![p{-}gen]\!](in1)$	Specialization time
4.	$output$	:=	$[\![p_{in1}]\!](in2)$	Run time

Η βασική μας επιδίωξη είναι να κάνουμε το στάδιο 4 γρήγορο που σημαίνει να παράγουμε γρήγορα εξειδικευμένα προγράμματα. Σε δεύτερη μοίρα είναι η ταχύτητα του τρίτου σταδίου, δηλαδή να εξειδικεύουμε γρήγορα. Δεν μας απασχολεί πάρα πολύ αν τα δύο πρώτα στάδια δεν είναι πολύ γρήγορα αν είναι αυτό να επιταχύνει το τρίτο. γιατί τότε απλά θα καθυστερούμε λίγο περισσότερο αλλά στο τέλος θα έχουμε ένα καλό μερικό αποτιμητή και αυτό θα χρειαστεί να το κάνουμε μόνο μια φορά.

Ως χρήστες ενός μερικού αποτιμητή πολλές φορές επιθυμούμε να καθορίσουμε εμείς τη στιγμή δέσμευσης κάθε μεταβλητής. Ο αναλυτής της στιγμής δέσμευσης μας δίνει αυτή την ευχέρεια γιατί μας επιτρέπει να δούμε και να μεταβάλουμε την διαίρεση που παράγει. Αν η διαίρεση δεν είναι αυτή που επιθυμούμε έχουμε δει τί ενέργειες πρέπει να κάνουμε για να την βελτιώσουμε. Για παράδειγμα αν αναλύουμε ένα διερμηνέα και δούμε στα αποτελέσματα ότι τα ορίσματα που γνωρίζουμε καταλήγουν να κατηγοριοποιούνται δυναμικά αντιλαμβανόμαστε ότι κάτι δεν πάει καλά. Είναι προφανές ότι αυτό είναι πάρα πολύ πιο εύκολο να το καταλάβουμε κοιτώντας το επισημειωμένο πρόγραμμα που παράγεται στο ενδιάμεσο στάδιο της offline μερικής αποτίμησης παρά εξετάζοντας το εναπομένον πρόγραμμα της online μερικής αποτίμησης. Μια βελτίωση αυτής της ιδέας είναι η αποσφαλμάτωση της στιγμής δέσμευσης όπου πλέον εξετάζουμε συστηματικά τα αποτελέσματα της ανάλυσης.

Αν θυμηθούμε τα όσα είπαμε στο προηγούμενο κεφάλαιο βλέπουμε ότι από τα επισημειωμένα προγράμματα μπορούν να γίνουν εκτιμήσεις για το ασφαλές διάστημα επιτάχυνσης, ενώ η online εξειδίκευση δεν μας παρέχει ανάλογες δυνατότητες.

Η ανάλυση της στιγμής δέσμευσης συνήθως συνοδεύεται κι από άλλες αναλύσεις για την ικανοποίηση επιπλέον απαιτήσεων πέρα από την ταιριαστή διαίρεση. Τυπικά παραδείγματα είναι η απαίτηση για τερματισμό της διαδικασίας, η αποφυγή περιττών υπολογισμών και η αποφυγή των ανενεργών στατικών μεταβλητών. Αν και είναι φυσιολογικό να συνδυάζουμε όλες τις αναλύσεις με την απλή ανάλυση της στιγμής δέσμευσης δεν είναι αναγκαίο ότι εξαρτώνται από τις πληροφορίες που αυτή παράγει. Για παράδειγμα η ανάλυση για την ανεύρεση ενεργών και μη ενεργών μεταβλητών μπορεί να χρησιμοποιηθεί και στην online μερική αποτίμηση. Η αντιμετώπιση του OnPE για αυτό το ζήτημα είναι η εξής, οι μη ενεργές μεταβλητές είναι δυναμικές και οι ενεργές ελέγχονται κάθε φορά για να αποφασιστεί η κατάταξή τους. Με αυτό τον τρόπο και επιτυγχάνεται η απαραίτητη γενίκευση όπως κάνει ο OffPE και εξακολουθεί να υπάρχει η δυνατότητα να εκμεταλλευτούμε τα προτερήματα της online μερικής αποτίμησης.

Από την άλλη πλευρά όμως κάποιες αναλύσεις εξαρτώνται από την πληροφορία για τη στιγμή δέσμευσης. Για παράδειγμα μια μορφή γενίκευσης είναι αυτή που γενικεύει οποιαδήποτε μεταβλητή δεν λαμβάνει μέρος στη λήψη αποφάσεων. Είναι εμφανές ότι η ανάλυση αυτή προκύπτει από τον συνδυασμό ανάλυσης της στιγμής δέσμευσης και ανάλυσης για ενεργές μεταβλητές.

Οι OnPE συνήθως κρατάνε ένα μεγάλο μέρος των προηγούμενων υπολογισμών τους οποίους συγκρίνουν με τον τρέχον για να αποφασίσουν και να αποφύγουν το ενδεχόμενο η διαδικασία να εισέλθει σε ένα ατέρμον βρόχο. Ορίζονται κριτήρια επικινδυνότητας και αν ένα τέτοιο ικανοποιηθεί (για παράδειγμα το όρισμα μιας αναδρομικής συνάρτησης συνεχώς μεγαλώνει) τότε πραγματοποιείται γενίκευση και το μέρος του προγράμματος που οφείλεται για αυτή την

συνεχή επανάληψη προκύπτει και στο εναπομένον πρόγραμμα. Οι OffPE δεν μπορούν εξ ορισμού να το κάνουν αυτό και για αυτό απαιτείται κατά την ανάλυση της στιγμής δέσμευσης να εξασφαλιστεί ότι η διαίρεση που θα προκύψει θα οδηγήσει σε τερματισμό και την την διαδικασία εξειδίκευσης.

Γενικά η ανάλυση του προγράμματος είναι ποιο διαδεδομένη στην offline μερική αποτίμηση γιατί είτε είναι αναγκαία είτε συνεισφέρει στα καλύτερα αποτελέσματα. Η online μερική αποτίμηση θα μπορούσε να επωφεληθεί από τέτοιες μορφές ανάλυσης γιατί υπάρχουν ακόμα μεγάλα περιθώρια βελτίωσης.

5.5 Ποια μεθοδολογία είναι καλύτερη

Με τα ως τώρα δεδομένα είναι αδύνατο να αποφανθούμε ποια μεθοδολογία είναι καλύτερη, η online ή η offline μερική αποτίμηση. Παραθέσαμε τα πλεονεκτήματα της κάθε μεθόδου αλλά το ποια είναι προτιμότερη εξακολουθεί να εξαρτάται από το κάθε συγκεκριμένο πρόβλημα και το πως θα αξιολογούνται τα χαρακτηριστικά του. Το να πούμε ότι μια από τις δύο τεχνικές υπερτερεί της άλλης είναι δύσκολο. Αντί αυτού προτιμάμε να μελετήσουμε και τις δύο κατευθύνσεις που μπορούμε να ακολουθήσουμε κατά την μερική αποτίμηση και να μάθουμε από αυτές.

Συμβιβαστικές λύσεις υπάρχουν, όπως για παράδειγμα η στρατηγική της μικτής μερικής αποτίμησης όπου η ανάλυση της στιγμής δέσμευσης χωρίζει τις μεταβλητές σε τρεις κατηγορίες. Πέρα από τις στατικές, που εδώ έχουν το ρόλο "πάντα στατική" και τις δυναμικές, δηλαδή τις "πάντα δυναμικές" υπάρχει και μια κατηγορία M, όπου ο ρόλος των μεταβλητών είναι μικτός και η απόφαση λαμβάνεται κατά την εξειδίκευση με χρησιμοποιώντας online τεχνικές.

5.6 Μεταγλώττιση κάνοντας online μερική αποτίμηση

Στη συνέχει θα αποδείξουμε ότι δεν ήταν θέμα τύχης που καταφέραμε να κάνουμε την αυτοεφαρμογή ενός offline μερικού αποτιμητή και να πάρουμε ένα μεταγλωττιστή που παράγει μικρά και γρήγορα προγράμματα, ενώ κάτι τέτοιο δεν είναι τόσο εύκολο με τους online. Η θεωρεία δεν θέτει κάποιο περιορισμό στους online μερικούς αποτιμητές όμως πειράματα μας έχουν δείξει ότι με τις τεχνικές της online εξειδίκευσης τα παραγόμενα προγράμματα είναι ως και τρεις φορές μεγαλύτερα και ως έξι φορές πιο αργά.

Δεδομένου ενός προγράμματος source ας προσπαθήσουμε να το μεταγλωττίσουμε. Έχουμε δει ήδη πολλές φορές ότι αυτό γίνεται ως εξής $target = [\![mix]\!]_L(int, source)$, όπου το int είναι προφανώς ο αντίστοιχος διερμηνέας. Το πρόγραμμα mix είναι ένας online αποτιμητής. Ας θεωρήσουμε ότι το κώδικα του μεταγλωττιστή συναντάμε κάπου την εντολή

```
E = (cons (car names) (car values))
```

Ο ρόλος της εντολής μπορεί να είναι να συνδέσει την τιμή value με την μεταβλητή name, να ενημερώσει δηλαδή την αποθήκη. Και το name και το value είναι μεταβλητές του διερμηνέα. Το σημαντικό που πρέπει να προσέξουμε είναι ότι οι δύο μεταβλητές έχουν διαφορετική στιγμή δέσμευσης. Η μεταβλητή name μπορεί να υπολογιστεί στατικά από το αρχικό πρόγραμμα source, ενώ για τη value αυτό είναι αδύνατο. Έτσι η μεταβλητή name θα είναι στατική, και χωρίς βλάβη της γενικότητας έστω ότι αντιστοιχεί στην τιμή a, ενώ στη value αντιστοιχεί μια ελαττωμένη έκφραση. Τελικά θα παραχθεί κώδικας για την εναπομείνουσα έκφραση

```
(cons 'a (car exp))
```

5.7 Αυτοεφαρμογή ενός Online μερικού αποτιμητή

Σύμφωνα με την δεύτερη προβολή του Futamura το $mix_{int} = [\![mix]\!]_L(mix, int)$ είναι ένα μεταγλωττιστής από την γλώσσα που διερμηνεύει ο int στην L. Θα θεωρήσουμε ότι ο διερμηνέας int

περιέχει την εντολή που αναλύσαμε πριν και θα δούμε τι λειτουργίες γίνονται όταν εφαρμόζουμε τον μερικό αποτιμητή mix στον εαυτό του.

Θεωρούμε ότι έχουμε την έκφραση `E = (cons (car names) (car values))` ως μέρος του int. Η συνάρτηση μέσα στο mix που θα τη χειριστεί είναι η OnPE, οπότε στον μεταγλωττιστή mix_{int}, θα συναντήσουμε κώδικα που προέρχεται από την από την εξειδίκευση της OnPE σε σχέση με την E. Συγκρίνοντας τον αρχικό και το εξειδικευμένο κώδικα διαπιστώνουμε ότι όλοι οι υπολογισμοί που αφορούν την είσοδο έχουν γίνει κατά την εξειδίκευση, αλλά η εντολή case που ελέγχει για στατικές μεταβλητές έχει απομείνει ως είχε γιατί τα ορίσματά της είναι δυναμικά.

Μπορούμε να κάνουμε δύο παρατηρήσεις για αυτόν τον μεταγλωττιστή. Πρώτα από όλα δεν κάνει καμιά διάκριση για την στιγμή δέσμευσης και χειρίζεται όλες τις εκφράσεις του διερμηνέα με τον ίδιο ακριβώς τρόπο, ακόμα κι όταν μερικές υποεκφράσεις θα μπορούσαν να είναι χαρακτηρισμένες "πάντα δυναμικές" ή "πάντα στατικές". Την συμπεριφορά αυτή την κληρονομεί από το mix που που όντας online μερικός αποτιμητής κάνει ακριβώς το ίδιο και και χειρίζεται και τις δυναμικές και τις στατικές εκφράσεις με κλήσεις προς τη συνάρτηση OnPE. Η συνάρτηση αυτή είναι σε θέση να επιστρέψει και τιμές και ελλατωμένες εκφράσεις (και τους αναθέτει και τον ανάλογο τίτλο ταυτόχρονα). Παρατηρούμε όμως ότι κατά την εκτέλεση του μεταγλωττιστή η μεταβλητή $lookup_{names}$ θα είναι πάντα στατική (φραγμένη κατανομή), οπότε η έκφραση `(reduce-car (`$lookup_{names}\rho$`))` πάντοτε θα αποτιμάται σε κάποια σταθερή τιμή, δηλαδή θα είναι κι αυτή με τη σειρά της στατική. Αυτό το συμπέρασμα είναι δυνατό να βγει κατά τη στιγμή που παράγεται ο μεταγλωττιστής και για αυτό το λόγο χρησιμοποιούμε σαφείς πληροφορίες για τις στιγμές δέσμευσης.

Η δεύτερη παρατήρηση έχει να κάνει με την υπερβολική γενικότητα του μεταγλωττιστή. Το τμήμα του κώδικα του μεταγλωττιστή που παρουσιάσαμε μπορεί να εξειδικεύσει το int με δύο διαφορετικούς τρόπους τουλάχιστον. Ο ένας τρόπος είναι ο επιθυμητός και αναμενόμενος, να παράγει το τελικό πρόγραμμα target από το αρχικό πρόγραμμα αλλά με άγνωστα τα δεδομένα εισόδου του προγράμματος. Μπορεί να παράγει όμως κι ένα περίεργο πρόγραμμα target που θα προκύψει από τα στατικά δεδομένα εισόδου χωρίς όμως των κώδικα του προγράμματος εισόδου! Το παραγόμενο πρόγραμμα τότε θα πάρει το αρχικό πρόγραμμα και θα παράγει την ίδια έξοδο που θα παρήγαγε η εκτέλεση του αρχικού προγράμματος με τα ίδια δεδομένα εισόδου:

$$[\![crazy]\!]_L(source) = [\![int]\!]_L(source, data) = [\![source]\!]_S(data) = target$$

Προφανώς το παραπάνω δεν παραβιάζει κανένα κανόνα από όσους περιγράψαμε μέχρι στιγμής όμως ο μεταγλωττιστής που παράγεται είναι τελείως άχρηστος. Σαν συμπέρασμα μπορούμε να πούμε με σιγουριά λοιπόν ότι το mix_{int} είναι πολύ πιο γενικό από ότι θα ήταν απαραίτητο.

Η γενικότητα αυτή οφείλεται στο γεγονός ότι δεν γίνεται διάκριση μεταξύ των ενεργειών του διερμηνέα που παραδοσιακά γίνονται κατά τη στιγμή της μεταγλώττισης, όπως είναι η συντακτική ανάλυση, και αυτών που γίνονται κατά τη στιγμή της εκτέλεσης, όπως είναι η αποτίμηση των εκφράσεων. Το mix που χρησιμοποιήσαμε αυτές τις αποφάσεις τις παίρνει τη στιγμή που συναντάει την κάθε έκφραση και είναι αναμενόμενο πως ο μεταγλωττιστής που θα προκύψει από αυτό θα συμπεριφέρεται με όμοιο τρόπο.

Για να επιτύχουμε λοιπόν τις επιδόσεις των κλασσικών μεθόδων κατασκευής μεταγλωττιστών πρέπει η έκφραση `(car names)` που είδαμε στην αρχή να αποτιμηθεί τη στιγμή της μεταγλώττισης και για την έκφραση `(car values)` πρέπει να ελαττωθεί και να παραχθεί ο αντίστοιχος κώδικας στο εναπομένον πρόγραμμα. Σε καμία από τις δύο περιπτώσεις δεν θα πρέπει να προηγηθεί έλεγχος για να αποφασιστεί αν η εκφράσεις είναι στατικές ή όχι.

5.8 Online μερική αποτίμηση με επισημειώσεις

Στη συνέχεια θα θεωρήσουμε ότι τα mix_1 και mix_2 δεν είναι κατ' ανάγκη όμοια. Τη διαδικασία παραγωγής ενός μεταγλωττιστή μπορούμε να την περιγράψουμε ως

$$comp = [\![mix_1]\!]_L(mix_2, int)$$

όπου ο mix_2 εξειδικεύεται σε σχέση με το int, από τον mix_1. Υπενθυμίζουμε ότι ο mix_2 είναι online μερικός αποτιμητής οπότε διενεργεί συνεχώς ελέγχους για να αποφανθεί αν πρέπει να χειριστεί της μεταβλητές και τις εκφράσεις ως στατικές ή δυναμικές. Μπορούμε να αντιληφθούμε ότι αυτό είναι λάθος, καθώς η πληροφόρηση ότι η συντακτική ανάλυση του διερμηνέα είναι στατική κατά την μεταγλώττιση, θα έπρεπε να υπάρχει τη στιγμή της εξειδίκευσης (στιγμή παραγωγής μεταγλωττιστή) Για να μπορέσουμε να ανάγουμε τους ελέγχους αυτούς σε στατικούς στον mix_2 πρέπει να του παρέχουμε επιπλέον πληροφορίες γιατί είναι αδύνατο να συμπεράνει μόνος του ότι θέλουμε ο μεταγλωττιστής comp που προκύπτει να δουλεύει μόνο σε κανονικές περιπτώσεις και όχι και στην περίεργη περίπτωση που παρουσιάσαμε πριν.

Ο διερμηνέας int δέχεται δύο ορίσματα, το source και το data, οπότε ο mix_2 πρέπει να ξέρει ότι ο μεταγλωττιστής πρόκειται να εφαρμοστεί στο source. Ένας βολικός τρόπος να περάσουμε αυτή την πληροφορία στον mix_2 είναι να επισημειώσουμε τον κώδικα του int και να χαρακτηρίσουμε εκ των προτέρων τις ενέργειές του είτε στατικές είτε δυναμικές. Με αυτό το τρόπο η ανάλυση της στιγμής δέσμευσης του int φαίνεται σαν τη σύνταξη και η συντακτική ανάλυση του mix_2 σίγουρα μειώνεται από τον mix_1. Ισοδύναμα θα μπορούσαμε να πούμε ότι εφοδιάζουμε τον διερμηνέα με μια σαφή διαίρεση για κάθε συνάρτησή του. Στην πράξη αυτό που περιγράψαμε είναι ότι ο mix_2 θα πρέπει να είναι offline μερικός αποτιμητής οπότε και το int θα πρέπει να επισημειωθεί.

Ας θεωρήσουμε ότι ο *offmix* είναι ένας offline μερικός αποτιμητής και χρησιμοποιεί τη συνάρτηση OffPE για να μειώσει τις εκφράσεις. Η παραγωγή του μεταγλωττιστή από τον επισημειωμένο κώδικα του διερμηνέα γίνεται προφανώς από τον υπολογισμό

$$[\![offmix]\!]_L(offmix_{ann}, int_{ann})$$

Οι αποφάσεις στο $offmix_{ann}$ για το αν θα γίνει αποτίμηση ή παραγωγή κώδικα δεν εξαρτώνται από τη μη διαθέσιμη μέχρι στιγμής είσοδο του *offmix*, δηλαδή το source, αλλά μόνο από τις επισημειώσεις στον διερμηνέα. Έτσι οι αποφάσεις είναι δεδομένες για το πως θα κατασκευαστεί ο μεταγλωττιστής και δεν αλλάζουν ανάλογα με το εκάστοτε πρόγραμμα source το οποίο εφαρμόζεται στο μεταγλωττιστή.

Σε αυτό το σημείο πρέπει να παρατηρήσουμε ότι το OffPE δεν παράγει τις ίδιες μειωμένες εκφράσεις με το OnPE. Το αποτέλεσμα του OffPE εξαρτάται άμεσα από τη δοσμένη διαίρεση, και για αυτό αν αλλάξουμε τη διαίρεση θα αλλάξει και το εναπομένον πρόγραμμα. Αντίθετα το OnPE που δεν λαμβάνει υπόψιν του καμιά επιπλέον πληροφορία σε κάθε περίπτωση θα παράγει το ίδιο πρόγραμμα. Επιπλέον, με δεδομένο ότι οι περισσότερες λειτουργίες του διερμηνέα αποτιμώνται κατά την εξειδίκευση μπορούμε να δυναμώσουμε την λειτουργικότητα του OffPE βάζοντάς τον να κάνει περισσότερους ελέγχους, για παράδειγμα να ελαττώνει περισσότερο τις δυναμικές εκφράσεις με βάση κανόνες που θα του περιγράψουμε, και αυτό όχι μόνο δεν θα έχει καμία αρνητική επίπτωση στο μέγεθος και την ταχύτητα του παραγόμενου μεταγλωττιστή αλλά θα βοηθήσει το OffPE να εκτελεί τους επιπλέον υπολογισμούς που κάνει το OnPE.

5.9 Μια συνταγή για αυτοεφαρμογή

Η κατασκευή ενός μερικού αποτιμητή με αυτοεφαρμογή για μια καινούρια γλώσσα είναι μια εργασία που χρειάζεται προσοχή. Μπορεί να χρειαστούμε πολλές επαναλήψεις μέχρι να αναπτύξουμε τις μεθόδους αποτίμησης ώστε να τελειοποιήσουμε τον παραγόμενο μεταγλωττιστή. Ακολουθεί μια διαδικασία που στηρίζεται στη χρήση offline εξειδίκευσης και μας βοηθάει να βάλουμε τα πράγματα σε μια σειρά. Είναι όμως σίγουρο ότι σε κάθε προσπάθεια από το μηδέν δεν θα πάνε τα πάντα βάση σχεδίου και θα πρέπει να είμαστε έτοιμοι να αντιμετωπίσουμε τις δυσκολίες που θα προκύψουν.

Προτείνεται στην αρχή να γραφτεί ο αυτοδιερμηνέας. Ακολουθούν οι λόγοι γιατί αυτό το βήμα είναι κατάλληλο για την παραγωγή γεννητόρων προγραμμάτων με βάση τη δεύτερη προβολή του Futamura $p{-}gen = [\![mix]\!](mix, p)$

- Είναι δύσκολο να πάρουμε σωστά αποτελέσματα από την εφαρμογή $[\![mix]\!](mix, p)$. Ακόμα πιο δύσκολο είναι να πάρουμε καλά αποτελέσματα.
- Ακολουθούμε την τακτική που λέει ότι για να λύσεις ένα δύσκολο πρόβλημα λύσε πρώτα ένα παρόμοιο αλλά ευκολότερο και μετά γενίκευσε στη συνέχεια την λύση.
- Για να μπορεί να εκτελεί μη τετριμμένους υπολογισμούς και εξειδικεύσεις ένα μερικός αποτιμητής mix πρέπει να εμπεριέχει έναν αυτοδιερμηνέα sint.
- Ένα απλούστερο πρόβλημα του υπολογισμού του $p{-}gen = [\![mix]\!](mix, p)$ είναι ο υπολογισμός του

$$p' = [\![mix]\!](sint, p)$$

- Μια λύση στο παραπάνω πρόβλημα θεωρείται καλή και αποδεκτή αν ικανοποιεί το κριτήριο της βελτιστότητας που περιγράψαμε σε περασμένο κεφάλαιο. Αν αδυνατούμε να παράγουμε ένα p' με τις ίδιες επιδόσεις με το p, τότε δεν θα μπορέσουμε να κατασκευάσουμε ούτε το $p{-}gen$.

Η διαδικασία περιγράφεται από τα παρακάτω βήματα:

1. Σκεφτόμαστε προσεχτικά τα παρακάτω ερωτήματα:
   - Ποια δεδομένα είναι γνωστά και ποια όχι.
   - Μπορεί κάθε μεταβλητή να θεωρηθεί πάντα στατική ή πάντα δυναμική ή θα πρέπει άλλοτε να την κατατάσσουμε στατική και άλλοτε δυναμική;
   - Τί είναι ένα σημείο στο εξειδικευμένο πρόγραμμα.
2. Γράφουμε έναν αυτοδιερμηνέα sint με απλό και καθαρό τρόπο, ίσως χρησιμοποιώντας μόνο ένα υποσύνολο της γλώσσας μας.
3. Εξειδικεύουμε με το χέρι και χρησιμοποιώντας απλές μεθόδους ένα δυο απλά προγραμματάκια για να δούμε πως γίνεται η διαδικασία στην γλώσσα μας και τι αποτελέσματα δίνει.
4. Κατασκευάζουμε μια διαίρεση και με αυτή την επιπλέον πληροφορία για το αρχικό πρόγραμμα το εξειδικεύουμε ως εξής:
   - κάνουμε τους υπολογισμούς για τις εκφράσεις που χαρακτηρίσαμε στατικές
   - παράγουμε κώδικα για τις εκφράσεις που χαρακτηρίσαμε δυναμικές
5. Ελέγχουμε (με το χέρι) αν μπορούμε να βρούμε μια διαίρεση για τα προγράμματα με τα οποία πειραματιστήκαμε στο 3ο βήμα που να παράγει το σωστό εναπομένον πρόγραμμα.
6. Βρίσκουμε μια διαίρεση για τον αυτοδιερμηνέα
7. Γράφουμε έναν μερικό αποτιμητή mix που να εκτελεί αυτόματα τις λειτουργίες του 4ο βήματος. Δεν είναι υποχρεωτικό να γραφεί στην ίδια γλώσσα.
8. Χρησιμοποιούμε το mix για να εξειδικεύσουμε τον αυτοδιερμηνέα με την διαίρεση που ορίσαμε σε σχέση με διάφορα απλά προγράμματα και μετά με τον εαυτό του. Αν το κριτήριο βελτιστότητας ισχύει έχουμε ξεπεράσει τα πιο δύσκολα μέρη του προβλήματος.

9. Αν στο βήμα 7 δεν χρησιμοποιήσαμε την ίδια γλώσσα, ξαναγράφουμε το mix στην γλώσσα την οποία θέλουμε να εξειδικεύσουμε.

10. Παρακολουθούμε τον τρόπο που κατασκευάζεται η διαίρεση και κατασκευάζουμε έναν αλγόριθμο ώστε να κατασκευάζεται όσο το δυνατόν πιο πιστά με αυτόματο τρόπο.

11. Δοκιμάζουμε το $[\![mix]\!](mix, p)$ για διάφορα προγράμματα p.

Μια συμβουλή είναι αν η γλώσσα μας περιέχει δομές ελέγχου ανώτερου επιπέδου, ταίριασμα προτύπων ή βαθιά φωλιασμένες δομές, τότε η δουλειά μας θα γίνει πολύ πιο εύκολη αν πρώτα υλοποιήσουμε την βασική λειτουργικότητα και στη συνέχεια σταδιακά προσθέσουμε όλα αυτά τα χαρακτηριστικά

Κεφάλαιο 6

Εφαρμογές της μερικής αποτίμησης

Σε αυτό το κεφάλαιο θα αναζητήσουμε τους λόγους της επιτυχίας ή αποτυχίας της εφαρμογής της μερικής αποτίμησης και θα αναφερθούμε πιο συγκεκριμένα στα προβλήματα που η μερική αποτίμηση λύνει με επιτυχία.

Υποθέτουμε ότι ένα πρόγραμμα p υπολογίζει μια συνάρτηση $f(s, d)$ και το s είναι στατικό, δηλαδή γνωστό κατά την στιγμή της εξειδίκευσης, ενώ το d δυναμικό. Ο τερματισμός της διαδικασίας εξειδίκευσης καθώς και το μέγεθος και η αποτελεσματικότητα του εναπομείναντος προγράμματος p_s εξαρτώνται άμεσα από τον τρόπο που το p χρησιμοποιεί τα στατικά και τα δυναμικά δεδομένα εισόδου του.

Η πρώτη περίπτωση είναι το p να μην έχει καθόλου στατικά δεδομένα. Ακόμα και σε αυτή την περίπτωση η μερική αποτίμηση μπορεί να προσφέρει θετικά αποτελέσματα όπως θα δούμε αργότερα σε παραδείγματα. Μια δεύτερη περίπτωση είναι να γενικεύσουμε μια ιδέα της θεωρίας της πολυπλοκότητας και να αναζητήσουμε σε ποιες περιπτώσει πρέπει να περιμένουμε καλά αποτελέσματα από την μερική αποτίμηση. Μία ανυποψίαστη μηχανή Turing (oblivious) είναι αυτή της οποία η κίνηση της κεφαλής ανάγνωσης εξαρτάται μόνο από το μήκος της ταινίας εισόδου και είναι ανεξάρτητη από τα περιεχόμενά της.

Λέμε ένα πρόγραμμα p είναι ανυποψίαστο (σε σχέση με την διαίρεση των μεταβλητών εισόδου) όταν η ροή ελέγχου του εξαρτάται μόνο από στατικές τιμές, για παράδειγμα ποτέ δεν κάνει έλεγχο της τιμής κάποιας δυναμικής μεταβλητής. Αντιλαμβανόμαστε ότι σε ένα τέτοιο πρόγραμμα υπάρχει μόνο μια δυνατή ακολουθία σημείων προγράμματος από τα οποία θα περάσει ο έλεγχος κι η μερική αποτίμησή του απλά θα ακολουθήσει αυτό το μοναδικό μονοπάτι. Ως συνέπεια ο τερματισμός εξαρτάται από τις τιμές των στατικών μεταβλητών σε ένα μόνο συνδυασμό, δηλαδή δεν μπορούμε να έχουμε περισσότερους συνδυασμούς στατικών δεδομένων. Το ποιες από τις τιμές θα έχουν καθοριστικό ρόλο εξαρτάται από τα δυναμικά δεδομένα.

Το μέγεθος και ο χρόνος εκτέλεσης του p_s είναι ανάλογα του πόσες δυναμικές μεταβλητές και εκφράσεις συναντάμε στο πρόγραμμα p. Ο χρόνος της εξειδίκευσης είναι ανάλογος του χρόνου που απαιτείται για να γίνουν οι στατικοί υπολογισμοί του p συν τον χρόνο για την παραγωγή του εναπομείναντος προγράμματος

Τα προγράμματα φαίνεται αν είναι ανυποψίαστα ή όχι μόνο σε σχέση με την διαίρεση των μεταβλητών τους. Είναι λοιπόν προφανές ότι με τη χρήση ενός online μερικού αποτιμητή δεν μπορεί να γίνει καμία πρόβλεψη από την παραπάνω.

Σε ένα υποψιασμένο πρόγραμμα (non oblivious) η ροή ελέγχου μπορεί να ακολουθήσει πολλά πιθανά μονοπάτια, ανάλογα με τις τιμές των δυναμικών μεταβλητών εισόδου. Ο μερικός αποτιμητής πρέπει να λάβει υπόψιν του όλες τις περιπτώσεις και να παράγει εξειδικευμένο κώδικα για κάθε συνδυασμό, δηλαδή και για τους δύο κλάδους κάθε εντολής διακλάδωσης που ελέγχεται από δυναμική έκφραση. Αυτό είναι πιθανό πως θα μας οδηγήσει σε μεγάλα εναπομείναντα προγράμματα p_s, αλλά ακόμα κι έτσι μάλλον ταχύτερα από τα αρχικά p.

Οι διερμηνείς είναι υποψιασμένα προγράμματα υπό αυτή την έννοια επειδή παράγουν κώδικα που κάνει ελέγχους, ευτυχώς όμως όπως θα δείξουμε στη συνέχει ανήκουν σε μια κατηγορία, τα ασθενώς υποψιασμένα προγράμματα, και για αυτό επιδέχονται μεγαλύτερης εξειδίκευσης.

6.1 Μερική αποτίμηση προγραμμάτων χωρίς στατικά δεδομένα

Η μερική αποτίμηση μπορεί να βρει εφαρμογή ακόμα κι αν δεν υπάρχουν στατικά δεδομένα εισόδου στο πρόγραμμα. Ένα παράδειγμα είναι κάποιο πρόγραμμα που απαρτίζεται από πολλές αυτοτελείς μονάδες (modules) για να είναι περισσότερο παραμετροποιήσιμο. Ένα άλλο είναι ότι η μερική αποτίμηση περιλαμβάνει πολλές από τις τεχνικές βελτιστοποίησης που χρησιμοποιούν και οι παραδοσιακοί μεταγλωττιστές.

Η διάδοση των σταθερών είναι μια οικία βελτιστοποίηση και προκύπτει ακόμα και σε σημεία που δεν μπορεί να τα ελέγξει ο προγραμματιστής. Είναι απαραίτητη για να παραχθεί αποδοτικός κώδικας για την πρόσβαση σε έναν πίνακα για παράδειγμα, ο ενδιάμεσος κώδικας για την έκφραση `A[I,1]:=B[2,3]+A[I,1]` θα έχει πολλές πράξεις μεταξύ σταθερών. Η ελάττωση των σταθερών είναι σίγουρα μια μορφή μερικής αποτίμησης, όπως είναι και πολλές άλλες λειτουργίες χαμηλότερου επιπέδου.

Η μερική αποτίμησή μάλιστα είναι σε θέση να κάνει βελτιστοποιήσεις ακόμα και έξω από τα όρια μιας συνάρτησης, δηλαδή να βελτιστοποιήσει την συνδυασμένη κλήση δύο ή περισσότερων συναρτήσεων, ακόμα και να μειώσει τις κλήσεις συναρτήσεων.

6.2 Ανυποψίαστοι Αλγόριθμοι

Ας υποθέσουμε ότι θέλουμε να υπολογίσουμε το γινόμενο δύο πινάκων, το παρακάτω πρόγραμμα κάνει αυτό ακριβώς όπου θεωρούμε ότι οι A και B είναι πίνακες διαστάσεων $p \times p$. Το πρόγραμμα είναι ανυποψίαστο ως προς την διάσταση p.

```
prod(p, A, B):
  for i := 1 to p do
  for j := 1 to p do {
    C[i, j] := 0;
    for k := 1 to p do
      C[i, j] := C[i, j] + A[i, k] * B[k, j];
    write C[i, j]
  }
```

Μπορούμε να ελέγξουμε αν το πρόγραμμα είναι ανυποψίαστο αφού πρώτα αναλύσουμε τη στιγμή δέσμευσης κάθε μεταβλητής. Τότε το p είναι ανυποψίαστο αν δεν περιέχει κανένα έλεγχο για δυναμικές μεταβλητές.

Ας θεωρήσουμε τώρα ότι το p_s ως το αποτέλεσμα της εξειδίκευσης του p σε σχέση με το s. Αν το p είναι ανυποψίαστο τότε δεν θα περιέχει μεταβάσεις ελέγχου ροής από τη στιγμή που όλοι οι έλεγχοι είναι στατικοί κι ως εκ τούτου αποτιμώνται τη στιγμή της εξειδίκευσης. Με βάση τον παραπάνω κώδικα μπορούμε να πούμε ότι το p_n έχει πολυπλοκότητα $O(n)$. Για παράδειγμα το p_2 είναι το

```
prod_2(A, B):
  write A[1, 1] * B[1, 1] + A[1, 2] * B[2, 1];
  write A[1, 1] * B[1, 2] + A[1, 2] * B[2, 2];
  write A[2, 1] * B[1, 1] + A[2, 2] * B[2, 1];
  write A[2, 2] * B[1, 2] + A[2, 2] * B[2, 2];
```

Η μερική αποτίμηση ανυποψίαστων προγραμμάτων παράγει μια μεγάλη ακολουθία εντολών χωρίς εντολές διακλάδωσης. Αυτό σημαίνει ότι παράγει μεγάλα βασικά μπλοκ και για αυτά ξέρουμε πως υπάρχουν πολύ ανεπτυγμένες τεχνικές μεταγλώττισης και βελτιστοποίησης. Ειδικά μεγάλα οφέλη παρουσιάζονται σε αρχιτεκτονικές που γίνεται χρήση σωληνώσεων. Επίσης και σε παράλληλες αρχιτεκτονικές ποιο αποδοτικά εκτελούνται τα προγράμματα χωρίς άλματα και βρόχους. Η κατανομή του κώδικα των μπλοκ σε πολλά μηχανήματα μπορεί να μειώσει κατακόρυφα την πολυπλοκότητα του πολλαπλασιασμού πινάκων ακόμα και σε $O(log(n))$.

6.2.1 Ασθενώς Ανυποψίαστοι Αλγόριθμοι

Ένα πρόγραμμα p το λέμε ασθενώς ανυποψίαστο αν οι αλλαγές στα δυναμικά δεδομένα δεν επηρεάζουν τις ακολουθίες των τιμών που ανατίθενται στα στατικά δεδομένα. Αυτός ο ορισμός είναι προφανώς πιο χαλαρός από τον ορισμό των ανυποψίαστων αλγορίθμων.

Για να καταλάβουμε καλύτερα τον ορισμό ας δούμε ένα παράδειγμα. Θεωρούμε ένα πρόγραμμα Bsort που κάνει ταξινόμηση φυσαλίδας. Το Bsort είναι ασθενώς ανυποψίαστο σε σχέση με το μήκος n της λίστας που θα ταξινομηθεί, γιατί αν και υπάρχουν δυναμικοί έλεγχοι (συγκρίσεις) και ανταλλαγές τιμών, παρ' όλα αυτά δεν επηρεάζουν τις τιμές των στατικών μεταβλητών. Όπως και πριν το μέγεθος του p_s είναι ανάλογο του πλήθους των δυναμικών μεταβλητών και εκφράσεων στο p. Ο χρόνος εκτέλεσής του όμως ενδέχεται να είναι μεγαλύτερος όμως επειδή υπάρχουν δυναμικοί έλεγχοι και βρόχοι.

Τα ασθενώς ανυποψίαστα προγράμματα έχουν πολλά κοινά με τα ανυποψίαστα. Για παράδειγμα αν δεν επιστρέφουν μόνο ένα μεγάλο βασικό μπλοκ χωρίς άλματα, το εκάστοτε p_s εξακολουθεί να απαρτίζεται από λίγα σχετικά μεγάλα μπλοκ. Για αυτό το λόγο και πάλι μπορούν να εκμεταλλευτούν σε μεγάλο βαθμό τις αρχιτεκτονικές με σωληνώσεις καθώς και τις παράλληλες αρχιτεκτονικές. Το μέγεθός του εξακολουθεί να είναι προβλέψιμο σίγουρα όχι όσο προβλέψιμο είναι το μέγεθος των ανυποψίαστων προγραμμάτων, αλλά πολύ περισσότερο από αυτό των υποψιασμένων.

Ένα απλό πρόγραμμα που δεν είναι ασθενώς ανυποψίαστο είναι το ακόλουθο

```
double(x) = f(x, 0)
f(x, y)   = if x = 0 then y else f(x-1, y+2)
```

όπου το x είναι δυναμικό. Η τιμή της μεταβλητής y είναι αρχικά 0 και μετά αυξάνεται κατά μια σταθερή, οπότε μια απλοϊκή ανάλυση της στιγμής δέσμευσης θα την κατηγοριοποιούσε στατική.

Ακόμα κι αν η y δεν εξαρτάται άμεσα από την x, η ακολουθία των τιμών που της ανατίθενται στην πράξη αποφασίζεται από το x. Όντας δυναμική ο μερικός αποτιμητής πρέπει κάθε φορά να λαμβάνει υπόψιν του και τα δυο πιθανά ενδεχόμενα του ελέγχου και άρα η εξειδίκευση θα πρέπει να γίνει για άπειρες τιμές του y.

Για ένα ακόμα παράδειγμα ας θεωρήσουμε το πρόγραμμα p που κάνει δυαδική αναζήτηση σε ένα πίνακα $T_0, ..., T_{2^n-1}$ και με αρχική κλήση την `Find(T, 0, m, x)` και $m = 2^n - 1$. Αν υποθέσουμε πως το delta είναι στατικό και το i δυναμικό, τότε το πρόγραμμα είναι ασθενώς ανυποψίαστο αφού οι συγκρίσεις με το x δεν επιδρούν στις τιμές που ανατίθενται στο delta.

```
Find(T, i, delta, x) =
  Loop: if delta = 0 then
          if x = T[i] then return(i) else return(NOTFOUND);
        if x >= T[i + delta] then i := i + delta;
        delta := delta / 2;
        goto Loop;
```

Αν το αποτιμήσουμε μερικώς σε σχέση με την τιμή delta=4 παίρνουμε το εναπομένον πρόγραμμα

```
if x >= T[i+4] then i := i + 4;
if x >= T[i+2] then i := i + 2;
if x >= T[i+1] then i := i + 1;
if x  = T[i] then return(i) else return(NOTFOUND)
```

Γενικά το p_n εκτελείται σε χρόνο της τάξης $O(log(n))$ με καλύτερη σταθερή συνιστώσα όμως από το αρχικό πρόγραμμα, που έχει την ίδια πολυπλοκότητα. Όμως και το μέγεθός του είναι της τάξης του $O(log(n))$ πλέον.

6.3 Υποψιασμένοι Αλγόριθμοι

Πολλά προβλήματα δεν είναι ανυποψίαστα υπό καμία έννοια και αυτό μπορεί να οδηγήσει σε απρόβλεπτα αποτελέσματα κατά την αποτίμηση. Έχουμε δει ότι το p_s μπορεί να γίνει τεράστιο ή ακόμα και να μην τελειώσει ποτέ η διαδικασία εξειδίκευσής του επειδή όλοι οι πιθανοί συνδυασμού στατικών μεταβλητών πρέπει να ληφθούν υπόψιν αν και πολύ λίγοι από αυτούς θα χρειαστούν τελικά για τον υπολογισμό $[\![p]\!](s,d)$ για οποιαδήποτε μεταβλητή d.

Για να δείξουμε τα προβλήματα που μπορεί να παρουσιαστούν, ας θεωρήσουμε το προηγούμενο πρόγραμμα δυαδικής αναζήτησης αλλά με στατικό το n. Θα μπορούσαμε να κατηγοριοποιήσουμε το i στατικό αφού είναι φραγμένο στο σύνολο $0, 1, ..., n-1$. Το πρόγραμμα υπό αυτές τις συνθήκες δεν είναι ανυποψίαστο, αφού οι έλεγχοι στο x επηρεάζουν την τιμή του στατικού i.

Αν το αποτιμήσουμε μερικώς σε σχέση με τις τιμές delta=4 και i=0 παίρνουμε αυτή τη φορά το εναπομένον πρόγραμμα

```
if x >= T[4] then
  if x >= T[6] then
    if x >= T[7] then
      if x = T[7] then return(7) else return(NOTFOUND)
      else if x = T[6] then return(6) else return(NOTFOUND)
    else if x >= T[5] then
      if x = T[5] then return(5) else return(NOTFOUND)
      else if x = T[4] then return(4) else return(NOTFOUND)
    else if x >= T[2] then
      if x >= T[3] then
        if x = T[3] then return(3) else return(NOTFOUND)
        else if x = T[2] then return(2) else return(NOTFOUND)
      else x >= T[1] then
        if x = T[1] then return(1) else return(NOTFOUND)
        else if x = T[0] then return(0) else return(NOTFOUND)
```

Το εξειδικευμένο πρόγραμμα και πάλι εκτελείται σε χρόνο $O(log(n))$ και με ακόμα καλύτερη σταθερή συνιστώσα από ότι πριν. Από την άλλη όμως το μέγεθός του είναι πλέον της τάξης του $O(n)$ εκθετικά μεγαλύτερο δηλαδή από την ανυποψίαστη εκδοχή του. Όμως οι επιπτώσεις δεν είναι πάντοτε αρνητικές. Στη συνέχεια ακολουθούν δύο παραδείγματα που παρουσιάζουν κάποια από τα προβλήματα και τον τρόπο που τα λύνουμε.

Υποθέτουμε ότι έχουμε ένα πρόγραμμα p για να κάνουμε τον υπολογισμό Find(G, A, B), όπου G είναι ένας γράφος και A, B είναι οι κόμβοι αρχής και τέλους. Το αποτέλεσμα θα είναι ένα μονοπάτι από τον κόμβο A στον κόμβο B, αν φυσικά κάτι τέτοιο υπάρχει, διαφορετικά μια ένδειξη σφάλματος. Το αποτέλεσμα της εξειδίκευσης του p σε σχέση με τα στατικά (γνωστά) G και B θα είναι ένα πρόγραμμα που θα βρίσκει γρήγορα την διαδρομή από οποιοδήποτε κόμβο A στον B.

Μια απλή εξειδίκευση πιθανώς θα μας οδηγήσει σε έναν αργό αλγόριθμο μιας και ο αλγόριθμος του Dijkstra ψάχνει όλα τα πιθανά μονοπάτια ξεκινώντας από το δυναμικό A μέχρι να φτάσει το στατικό B. Εναλλακτικά θα μπορούσαμε να εκμεταλλευτούμε το γεγονός ότι το A ανήκει σε μια φραγμένη κατανομή για να πάρουμε καλύτερα αποτελέσματα. Η ιδέα είναι να ενσωματώσουμε το p μέσα σε ένα άλλο πρόγραμμα που θα καλεί την συνάρτηση Find μόνο με πλήρως στατικά ορίσματα:

```
function Paths-to(G, A, B) =
  let nodes = Node-list(G) in
    forall A1 in nodes do
      if A = A1 then Find(G, A1, B)
```

```
function Find(G, A, B) = ...
```

Τώρα οι μεταβλητές nodes και A1 είναι στατικές και το αποτέλεσμα της εξειδίκευσης σε σχέση με τα G και B θα μπορούσε να είναι ένα πρόγραμμα της μορφής

```
function Paths-to-Copenhagen(A) =
  if A = Hamburg then [Hamburg, C1, ..., Copenhagen] else
  if A = London  then [London, D1, ..., Copenhagen]  else
  ...                                                 else
  if A = Paris   then [Paris, E1, ..., Copenhagem]    else NOPATH
```

Οι διερμηνείς είναι υποχρεωτικά υποψιασμένα προγράμματα αν η γλώσσα που υλοποιούν περιέχει ελέγχους αλλά έχουμε δει ότι εξειδικεύονται αρκετά καλά. Αυτό βέβαια δεν συμβαίνει πάντα αλλά η εμπειρία μας έχει δείξει ότι αν προγραμματίζουμε ακολουθώντας κάποιες συγκεκριμένες τεχνικές τότε τα αποτελέσματα θα είναι καλά. Στη συνέχεια θα αναφερθούμε σε ορισμένα σημαντικά χαρακτηριστικά

Πρώτα απ' όλα οι διερμηνείς συνήθως γράφονται με ένα συνθετικό τρόπο, ώστε το αποτέλεσμα μιας ενέργειας μέσα σε μια έκφραση να είναι συνδυασμός των αποτελεσμάτων που επιστρέφονται από ενέργειες πάνω σε υποεκφράσεις της. Η συνθετικότητα είναι είναι μια βασική υπόθεση της σημασιολογίας όταν το βασικό κίνητρο είναι να κάνουμε εφικτές τις αποδείξεις βασιζόμενοι στην επαγωγή των δομών του συντακτικού της γλώσσας.

Από την δική μας σκοπιά η συνθετικότητα δηλώνει ότι ένας διερμηνέας μεταχειρίζεται μόνο κομμάτια του αρχικού προγράμματος. Από τη στιγμή που αυτά είναι πεπερασμένα στο πλήθος μπορούν να χρησιμοποιηθούν για την μερική αποτίμηση.

Στην πράξη η προϋπόθεση για συνθετικότητα μπορεί να χαλαρώσει, όσο τα στατικά δεδομένα παραμένουν σε μια φραγμένη κατανομή, εννοώντας ότι για κάθε δεδομένο στατικό πρόγραμμα εισόδου σε ένα διερμηνέα, όλες οι μεταβλητές που χαρακτηρίζονται στατικές μπορούν να αποθηκεύσουν ένα πεπερασμένο πλήθος από διαφορετικές τιμές και έτσι εξασφαλίζεται ότι η διαδικασία της μερικής αποτίμησης θα τερματίσει.

Οι διερμηνείς που κατασκευάζονται με άλλους τρόπους, για παράδειγμα παράγουν καινούριο κώδικα κατά την εκτέλεσή τους, δεν είναι εύκολο, πιθανώς αδύνατο, να εξειδικευτούν με σημαντική βελτίωση των επιδόσεών τους.

Κεφάλαιο 7

Επίλογος

Στα προηγούμενα κεφάλαια αναλύσαμε την ιδέα της μερικής αποτίμησης. Είναι σημαντικό να πούμε ότι όσα αναφέραμε είναι η βάση της θεωρίας, ενώ η έρευνα έχει προχωρήσει πολύ πιο πέρα με σημαντικές επιτυχίες

Δώσαμε ιδιαίτερη προσοχή στην κατασκευή μεταγλωττιστών μέσω εξειδίκευσης, γιατί αυτή είναι μια πλευρά της μερικής αποτίμησης που είναι αρκετά ενδιαφέρουσα και ταυτόχρονα πάρα πολύ χρήσιμη. Είναι πολύ βολικό και συνηθισμένο να ορίζουμε μια καινούρια γλώσσα γράφοντας έναν διερμηνέα ή αυτοδιερμηνέα για αυτή. Με αυτόν τον τρόπο η σημασιολογία της γλώσσας προκύπτει άμεσα, ώστε και να μπορούμε να την αντιλαμβανόμαστε πλήρως και να μπορούμε να την ορίζουμε σωστά όπως αρχικά σκοπεύαμε. Με τις κλασσικές μεθόδους ανάπτυξης μεταγλωττιστών όμως, ο αυτοδιερμηνέας δεν είχε καμιά αξία στο δρόμο για την επίτευξη του στόχου της κατασκευής ενός μεταγλωττιστή. Αντίθετα με τις τεχνικές που έχουν αναπτυχθεί γύρω από την μερική αποτίμηση, ο ορισμός του αυτοδιερμηνέα είναι αυτοσκοπός. Από εκεί και πέρα η κατασκευή του μεταγλωττιστή είναι μια πλήρως αυτοματοποιημένη διεργασία.

Τα τελευταία χρόνια στον χώρο της πληροφορικής και των γλωσσών προγραμματισμού ειδικότερα, έχει εμφανιστεί μια καινούρια έννοια, αυτή της μεταγλώττισης ακριβώς την στιγμή που χρειάζεται (Just In Time compilation, JIT). Η κεντρική ιδέα της μεταγλώττισης τη στιγμή που χρειάζεται, είναι ότι αντί να μεταγλωττίζουμε το πρόγραμμά μας στην τελική γλώσσα και να παράγουμε ένα εκτελέσιμο πρόγραμμα, χρησιμοποιούμε έναν ειδικό διερμηνέα, που συνήθως ονομάζουμε εικονική μηχανή (virtual machine), για να εκτελέσουμε το αρχικό πρόγραμμα, ή κάποια ενδιάμεση αναπαράσταση αυτού. Αυτό που ξεχωρίζει τους διερμηνείς που χρησιμοποιήσαμε στα προηγούμενα κεφάλαια από μια εικονική μηχανή είναι ότι η εικονική μηχανή στην ουσία κάνει μεταγλώττιση του αρχικού προγράμματος, όμως αντί να παράγει ένα εκτελέσιμο πρόγραμμα, εκτελεί απευθείας η ίδια τον κώδικα που παρήγαγε.

Η ιδέα της εικονικής μηχανής βλέπουμε ότι μοιράζεται πολλά κοινά με την έννοια της μερικής αποτίμησης. Μια εικονική μηχανή δεν μεταγλωττίζει αδιακρίτως τον κώδικα, αλλά λαμβάνει υπόψιν της το χρονικό κόστος της μεταγλώττισης και αν είναι ασύμφορο προτιμάει την διερμήνευση. Κατά τη μεταγλώττιση ενός τμήματος του κώδικα λαμβάνει υπόψιν της όλη τη γνώση για τις τιμές των μεταβλητών που είναι γνωστές σε εκείνο το σημείο. Αν μεταγλωττίσει μια συνάρτηση θα φυλάξει το αποτέλεσμα, για την περίπτωση που ξανακληθεί η ίδια συνάρτηση. Όλες αυτές οι τεχνικές μοιάζουν γνωστές και στην ουσία προέρχονται από τον κόσμο της μερικής αποτίμησης. Είναι συχνό φαινόμενο να αμφισβητούνται οι επιδόσεις των εικονικών μηχανών σε σχέση με τα αντίστοιχα εκτελέσιμα προγράμματα, και γενικά αν η μεταγλώττιση την στιγμή που χρειάζεται είναι ανταγωνιστική της κλασσικής μεταγλώττισης, όπως αμφισβητήσιμα είναι τα αποτελέσματα στης μερικής αποτίμησης σε σχέση με την χειροκίνητη εξειδίκευση των αλγορίθμων.

Πέρα από τις γλώσσες προγραμματισμού, η μερική αποτίμηση έχει βοηθήσει σε πολλού τομείς της επιστήμης. Έχει χρησιμοποιηθεί για την φωτοσκίαση τρισδιάστατων μοντέλων, όπου ο αλγόριθμος φωτοσκίασης εξειδικεύεται σε σχέση με το μοντέλο και το αποτέλεσμα εκτελείται για διαφορετικές συνθήκες φωτισμού. Στην αριθμητική ανάλυση έχει συμβάλει στην απλοποίηση των μεθόδων επίλυσης αραιών συστημάτων εξισώσεων.

Ακόμα συμβαίνει συχνά διαφορετικές ερευνητικές ομάδες να αναπτύσσουν κάποια μοντέλα

από κοινού και στην συνέχεια η καθεμία να παραμετροποιεί το μοντέλο για να το προσαρμόσει στις δικές της ανάγκες. Η μερική αποτίμησης του μοντέλου σε σχέση με την εκάστοτε παραμετροποίηση μπορεί να βελτιώσει τις επιδόσεις.

Τέλος υπάρχει και μια κατηγορία προβλημάτων τα οποία είναι από τον ορισμό τους διερμηνευτικής φύσης, όπως για παράδειγμα η εξομοίωση δικτύων. Με χρήση της μερικής αποτίμησης και αν θεωρήσουμε πως κάθε στοιχείο στο δίκτυο έχει δεδομένα χαρακτηριστικά, μπορούμε να εξειδικεύσουμε τον αλγόριθμο εξομοίωσης σε σχέση με το δίκτυο και να πάρουμε ένα πρόγραμμα που μας δίνει την απόκριση του δικτύου για κάθε διέγερση.

Το συμπέρασμα στο οποίο καταλήγουμε είναι ότι η μερική αποτίμηση έχει προσφέρει πολλά και μπορεί να προσφέρει ακόμα περισσότερα. Πολλές τεχνικές που χρησιμοποιούμε ενδέχεται να περιέχουν κάποιες από τις έννοιες της εξειδίκευσης προγραμμάτων και αλγορίθμων και είναι πιθανό πολλοί από μας να εφαρμόζουμε τις τεχνικές της μερικής αποτίμησης πολύ πιο συχνά από ότι αντιλαμβανόμαστε. Η μερική αποτίμηση δεν είναι δύσκολο να γίνει κατανοητή και είναι ένα χρήσιμο εργαλείο στα χέρια όποιου μπορεί να την χειριστεί σωστά.

Βιβλιογραφία

[Gluc03] Robert Glück, "The translation power of the Futamura projections", in Manfred Broy and Alexander V. Zamulin, editors, *Perspectives of System Informatics. Proceedings*, vol. 2890 of *Lecture Notes in Computer Science*, pp. 133–147, Springer-Verlag, 2003.

[Jone93] Neil D. Jones, Carsten K. Gomard and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[Jone95] Neil D. Jones, "MIX Ten Years Later", in William L. Scherlis, editor, *Proceedings of PEPM '95*, pp. 24–38, ACM, ACM Press, 1995.

[Jone96] Neil D. Jones, "An Introduction to Partial Evaluation", *ACM Computing Surveys*, vol. 28, no. 3, pp. 480–504, September 1996.

[Jone97] Neil D. Jones, *Computability and Complexity from a Programming Perspective*, Foundations of Computing, MIT Press, Boston, London, 1 edition, 1997.