ΗΥ340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

```
VAR i:Integer;
    FUNCTION(Symbol) replicate
        x = (function(x,y){return x+y;});
        class DelFunctor: public std::unary_function<
```

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

ΗΥ340 : ΓΛΩΣΣΕΣ ΚΑΙ ΜΕΤΑΦΡΑΣΤΕΣ

Διάλεξη 16η
ΘΕΜΑΤΑ ΒΕΛΤΙΣΤΟΠΟΗΣΗΣ

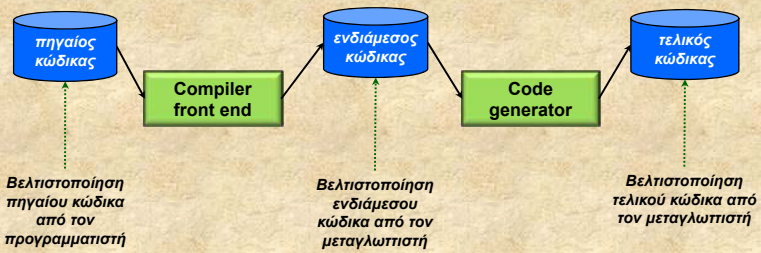
Περιεχόμενα

Εισαγωγή (1/4)

- *Κριτήρια επιλογής τεχνικών βελτιστοποίησης*
  - Οι καταλληλότεροι μετασχηματισμοί του προγράμματος είναι αυτοί που πετυχαίνουν το καλύτερο δυνατό αποτέλεσμα με τη μικρότερη προσπάθεια
    - Επιπλέον, είναι επιθυμητό να είναι και κατασκευαστικά απλοί
  - Κάθε τέτοιος μετασχηματισμός θα πρέπει κατά μέσο όρο να επιταχύνει το πρόγραμμα κατά μία μετρήσιμη ποσότητα
    - Περιστασιακά μπορεί μία βελτιστοποίηση να επιβραδύνει το πρόγραμμα, καθόσον κατά μέσο όρο πάντα οδηγεί σε βελτίωση
  - Πρέπει να αξίζει τον κόπο, σταθμίζοντας το φόρτο κατασκευής με το τελικό επίτευγμα βελτίωσης ταχύτητας

Εισαγωγή (2/4)

- Η βελτιστοποίηση θα μας απασχολήσει ως αυτοματοποιημένη διαδικασία που εφαρμόζεται από τον μεταγλωττιστή
  - Ωστόσο τα πιο θεαματικά αποτελέσματα πετυχαίνονται όταν συνδυαστεί η αυτόματη βελτιστοποίηση με τη χειροκίνητη στο επίπεδο του πηγαίου κώδικα

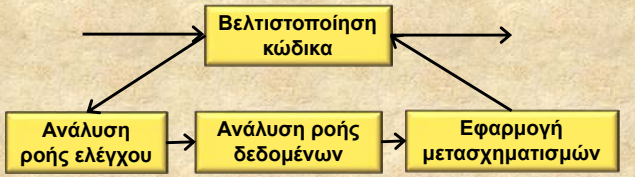


Εισαγωγή (3/4)

- Σε μερικές περιπτώσεις οι αλγοριθμικές βελτιώσεις οδηγούν σε θεαματικά αποτελέσματα
- Επίσης, η γνώση του προγραμματιστή ως προς τα σχήματα μεταγλώττισης του πηγαίου κώδικα σε τελικό κώδικα είναι αρκετή ώστε οδηγήσει στην χειροκίνητη βελτίωση του κώδικα
  - και αυτό συγκαταλέγεται στις περιπτώσεις hacking
- Θα μελετήσουμε δύο περιπτώσεις βελτιστοποίησης
  - μικρής κλίμακας – ***peephole optimization***
  - μετασχηματισμοί ανάλυσης κώδικα – ***code transformation***

Εισαγωγή (4/4)

- Επιλέγουμε την εφαρμογή βελτιστοποιήσεων στον ενδιάμεσο κώδικα διότι είναι εφικτές χωρίς να επηρεάζεται η παραγωγή τελικού κώδικα
- Θα μελετήσουμε επιπλέον το κυριότερο θέμα βελτιστοποιήσεων στον τελικό κώδικα που είναι η ανάθεση καταχωρητών
  - μικρής κλίμακας βελτιστοποιήσεις είναι εφαρμόσιμες και στον τελικό κώδικα



Περιεχόμενα

Βελτιστοποίηση μικρής κλίμακας (1/10)

- Η παραγωγή τελικού κώδικα ανά εντολή συνήθως παράγει πλεονάζοντες εντολές και μη βελτιστοποιημένες δομές κώδικα
- Μία απλή αλλά αποτελεσματική τεχνική για την βελτίωση για την τοπική βελτίωση του τελικού κώδικα είναι το ***peephole optimization***
  - βασίζεται στην εξέταση μικρού τμήματος κώδικα και στην αντικατάσταση του με κάποιο μικρότερο η ταχύτερο
  - η τεχνική εφαρμόζεται τόσο στον τελικό κώδικα όσο και στον ενδιάμεσο κώδικα

Κατηγορίες peephole optimization

1. Πλεονάζοντες εντολές	2. Συνεχόμενες αλλαγές ροής ελέγχου
3. Απρόσιτος κώδικας	4. Αλγεβρικές απλοποιήσεις
5. Εναλλακτικές εντολές	6. Εκμετάλλευση ιδιωμάτων της μηχανής

Βελτιστοποίηση μικρής κλίμακας (2/10)

Πλεονάζοντες εντολές. Σε αρκετές περιπτώσεις παράγονται εντολές εκχώρησης ώστε να υποστηρίζεται η χρήση του αποτελέσματος της έκφρασης. Ωστόσο, εάν το αποτέλεσμα δεν χρησιμοποιείται, η εκτέλεση της εντολής δεν προσφέρει τίποτε στον υπολογισμό εκτός από χρονική καθυστέρηση. Π.χ.

`x=10;`	`assign 10 x` `assign x _t1`
`f();`	`call f` `getretval _t1`
`i++;`	`assign i _t1` `add 1 i i`

•Στο συγκεκριμένο παράδειγμα οι μισές εντολές που παράγονται είναι εντελώς άχρηστες καθώς δεν χρησιμοποιείται το αποτέλεσμα. Απαιτείται προφανώς η «απομάκρυνσή» τους ώστε ο κώδικας να γίνει ταχύτερος.
•Δεν χρειάζεται να αφαιρέσουμε μία τέτοια εντολή, αλλά *απλώς να θέσουμε ένα ignore flag* ώστε να μην γίνει target code generation για αυτό το quad.
•Σε τέτοια quads πρέπει να φροντίσουμε στην παραγωγή τελικού κώδικα να θέτουμε απλώς ως *taddress* την επόμενη εντολή τελικού κώδικα, για το ενδεχόμενο κάποιων jumps με προορισμό την ενδιάμεση εντολή που έχει «αφαιρεθεί» (αυτό θα συμβεί μόνο στην περίπτωση που το αφαιρούμενο quad είναι το πρώτο ενός σχήματος εντολών ενδιάμεσου κώδικα, όπως στο i++).

Βελτιστοποίηση μικρής κλίμακας (3/10)

Έστω η εντολή εκχώρησης σε κρυφή μεταβλητή t με index ***k****.*
i = ***k***+1, *ignore* = **false**, *f*=1;
while *forever* **do** {
  if instruction *i* is ***funcstart*** **then** ++*f*;
  if *i*=*total* **or** instruction *i* is ***funcend*** **and** --*f*=0 **then**
    ignore = **true**;
  else /* *Εκτός peephole ή χρησιμοποιείται το t**/
  if instruction *i* is conditional jump **or** instruction *i* reads *t* **then**
    return;
  else
  if instruction *i* writes *t* **then**
    ignore = **true**;

  if *ignore*=**true then** { *ignore_instruction*(*k*); **return**; }

  if instruction *i* jumps unconditionally to *j* **then**
    i = *j*; /* *Προχώρησε στο jump* */
  else
    i = *i*+1; /* *Προχώρησε στην επόμενη εντολή* */
}

Εξάλειψη πλεοναζόντων assignments (ευρεστικός αλγόριθμος για τη γλώσσα alpha) σε κρυφές μεταβλητές με peephole optimization

Βελτιστοποίηση μικρής κλίμακας (4/10)

Συνεχόμενες αλλαγές ροής ελέγχου. Στην περίπτωση παραγωγής κώδικα για τις συναρτήσεις, παράγονται και ειδικά jumps στο quad ακριβώς μετά το τέλος της συνάρτησης. Εάν υπάρχουν συνεχόμενες τέτοιες συναρτήσεις, τα jumps ουσιαστικά θα είναι διαδοχικά. Η βελτιστοποίησης στην περίπτωση αυτή έγκειται στην εξάλειψη τέτοιων διαδοχικών jumps και η αλλαγή τους με το τελικό.

```
function f(){}      1: jump 4
function g(){}      2: funcstart f
function h(){}      3: funcend f
                    4: jump 7
                    5: funcstart g
                    6: funcend g
                    7: jump 10
                    8: funcstart h
                    9: funcend h
                    10:
```

•Στην περίπτωση αυτή, πέρα από την αλλαγή του αρχικού jump, γνωρίζουμε ότι κάθε ενδιάμεσο unconditional jump που προηγείται ενός *funcstart* προέρχεται από την παραγωγή κώδικα για συναρτήσεις.

•Επομένως, δεν είναι ποτέ δυνατό να έχουμε jumps σε αυτές τις εντολές «από αλλού», άρα μπορούμε να αγνοήσουμε όλα αυτά τα ενδιάμεσα jumps.

Βελτιστοποίηση μικρής κλίμακας (5/10)

Αλλαγή ροής ελέγχου λόγω συναρτήσεων. Λόγω του γεγονότος ότι στη γλώσσα alpha ο ορισμός των συναρτήσεων επιτρέπεται παντού (ακόμη και ως actual argument), είμαστε υποχρεωμένοι να παράγουμε τα jumps τα οποία ουσιαστικά παρακάμπτουν τον κώδικα της συνάρτησης που παράγεται τη στιγμή που γίνεται parsed η συνάρτηση. Αυτό όμως προκαλεί έναν συνωστισμό από τέτοιου είδους jumps τα οποία δεν ανάγονται όλα στην προηγούμενη περίπτωση. Π.χ.

```
x=y;                    1:  assign y x
function f(){}          2:  jump 5
y=z;                    3:  funcstart f
function g(){}          4:  funcend f
z=w;                    5:  assign z y
function h(){}          6:  jump 9
w=q;                    7:  funcstart g
                        8:  funcend g
                        9:  assign w z
                        10: jump 13
                        11: funcstart h
                        12: funcend h
                        13: assign q w
                        14:
```

- **Εδώ η βελτιστοποίηση απαιτεί την μεταφορά των συναρτήσεων σε τέτοιο σημείο ώστε ο κώδικάς του να μην βρίσκεται ποτέ εντός κάποιου block ή του καθολικού κώδικα και έπειτα την εξάλειψη των εντολών jump ως ignored.**
- **Αυτό προϋποθέτει την πλήρη αναδιάταξη του κώδικα για τις συναρτήσεις και είναι καλό να συνιστά την πρώτη περίπτωση βελτιστοποίησης που θα εφαρμοστεί.**
- **<u>Ωστόσο</u> οι εντολές unconditional jump είναι ιδιαίτερα γρήγορες (<u>οι ταχύτερες</u>) ώστε να θεωρεί κανείς ότι επιβαρύνουν την εκτέλεση σοβαρά. Συνεπώς, δύσκολα θα αναγκαστείτε να κάνετε τέτοιου είδους βελτιστοποίηση.**

Βελτιστοποίηση μικρής κλίμακας (6/10)

Απρόσιτος κώδικας. Πρόκειται για κώδικα που δεν υπάρχει καμία περίπτωση να εκτελεστεί. Π.χ. το if τμήμα με τιμή έκφρασης που είναι πάντα false ή αντίστοιχα το else τμήμα με τιμή έκφρασης που είναι πάντα true. Το ίδιο ισχύει και για τις ανακυκλώσεις. Θα μπορούσαμε να πούμε ότι το ίδιο ισχύει και για τις εντολές return σε συνάρτηση, καθώς εάν έχουμε κώδικα μετά από return δεν μπορεί ποτέ αυτός να εκτελεστεί. Ωστόσο αυτό εξαρτάται και από το σημείο στο οποίο είναι «φωλιασμένο» το return, γεγονός που απαιτεί πιο λεπτομερή ανάλυση της ροής ελέγχου (δεν είναι peephole optimization).

```
1: if_eq true false 3
2: jump 10
3:
4:
…
9:
10:
```

Στον κώδικα αυτό, έχω conditional jump ακολουθούμενο από unconditional jump. Τότε:

- Εάν το αποτέλεσμα είναι πάντα false και δεν υπάρχει jump από άλλο σημείο του κώδικα σε κάποια από τις εντολές 3…9, οι εντολές αυτές συνιστούν απρόσιτο κώδικα.

Προσέξτε ότι σε αυτή την περίπτωση τελικά ούτε οι εντολές 1 και 2 χρειάζονται πλέον. Γενικά είναι πολύ σπάνιες τέτοιου είδους περιπτώσεις στον πηγαίο κώδικα και εν γένει εξαλείφονται ευκολότερα στο επίπεδο παραγωγής ενδιάμεσου κώδικα (γνωρίζουμε πότε έχουμε constant boolean expression).

Βελτιστοποίηση μικρής κλίμακας (7/10)

Συγχώνευση εκχωρήσεων. Λόγω των αλγορίθμων παραγωγής ενδιάμεσου κώδικα, όταν έχουμε αριθμητικές ή λογικές εκφράσεις ως r-value θα υπάρχει μία κρυφή μεταβλητή που θα λαμβάνει το αποτέλεσμα και αυτή έπειτα θα εκχωρείται στο l-value. Π.χ.

x=y+z; γίνεται *t=y+z; x=t;*

Σε τέτοιες περιπτώσεις χρήσης κρυφής μεταβλητής, θα μπορούσαμε να έχουμε συγχώνευση των δύο εντολών απευθείας ως *x=y+z;* και επιπλέον ενδεχομένως να καταργούσαμε αργότερα και την κρυφή μεταβλητή *t*, δηλ. λιγότερες εντολές και μνήμη. Το «παράξενο» είναι ότι η συγχώνευση μπορεί να γίνει «τυφλά» χωρίς κανένα άλλο έλεγχο, καθώς δεν υπάρχει περίπτωση να χρησιμοποιείται παρακάτω η μεταβλητή *t* για την ίδια έκφραση (εξάλειψη κοινών εκφράσεων κάνουμε αργότερα).

- Για την υλοποίηση της συγχώνευσης απλώς κάνουμε ignore την *x=t;* και αλλάζουμε το operand *t* του πρώτου quad σε *x*.

Βελτιστοποίηση μικρής κλίμακας (8/10)

Αλγεβρικές απλοποιήσεις. Σπάνιες περιπτώσεις, προέρχονται κυρίως από εξειδικευμένους αλγόριθμους παραγωγής κώδικα παρά από το πηγαίο πρόγραμμα. Ωστόσο, επειδή είναι εύκολο να εφαρμοστούν τις υλοποιούμε.

```
add x 0 x       /* x=x+0 */     ignore
sub x 0 x       /* x=x-0 */     ignore
mul x 1 x       /* x=x*1 */     ignore
div x 1 x       /* x=x/1 */     ignore
mul x 0 x       /* x=x*0 */     assign 0 x
div 0 x x       /* x=0/x */     assign 0 x
```

Βελτιστοποίηση μικρής κλίμακας (9/10)

Εναλλακτικές εντολές. Εφαρμόζεται σε τελικό κώδικα και βασίζεται στην ύπαρξη ταχύτερων εξειδικευμένων εντολών μηχανής. Σε μερικές γλώσσες ο ίδιος ο προγραμματιστής μπορεί να πετύχει το επιθυμητό αποτέλεσμα στον πηγαίο κώδικα, ωστόσο σε πιο υψηλού επιπέδου γλώσσες μπορεί αυτό να μην είναι εφικτό παρά μόνο με βελτιστοποίηση από τον μεταγλωττιστή.

`add x 1 x`	`inc x`
`sub x 1 x`	`dec x`
`add x 2 x`	`inc x`
	`inc x`
`mul x 2 x`	`shl x 1` `/* x = x<<1 */`
`div x 4 x`	`shr x 2` `/* x = x>>2 */`
`mod x 16 y`	`and x 15 y` `/* x = x & 15 */`

Βελτιστοποίηση μικρής κλίμακας (10/10)

Σε μερικές περιπτώσεις η επιλογή εναλλακτικών εντολών μπορεί να απαιτεί επιπλέον επεξεργασία. Π.χ. για τη βελτιστοποίηση πολλαπλασιασμών μπορεί να έχουμε διάσπαση σε έως και δύο παράγοντες εάν πρόκειται για δυνάμεις του δύο. Ωστόσο θέλει καλές μετρήσεις ώστε να πιστοποιήσουμε τη βελτίωση. Π.χ. για την εντολή σε πηγαίο κώδικα *x=y*96* έχουμε:

```
t = y * 96;
x = t;
```

Όμως $y * 96 = y * (64 + 32) = y*64 + y*32 = y<<6 + y<<5;$

```
t1 = y << 6;
t2 = y << 5;
x = t1+t2;      Μετά το assignment merging
```

«Δυστυχώς» ο πολλαπλασιασμός ακεραίων είναι ήδη γρήγορος όμως η διαίρεση που είναι πολύ πιο απαιτητική δεν χρήζει τέτοιας βελτίωσης.

Περιεχόμενα

Βασικά τμήματα και γράφοι ροής ελέγχου (1/6)

- Για την υλοποίηση τεχνικών βελτιστοποίησης πέραν της μικρής κλίμακας απαιτείται η εσωτερική αναπαράσταση του προγράμματος σε μία μορφή που επιτρέπει την εύκολη επεξεργασία και τροποποίηση της ίδιας της δομής του προγράμματος
- Τέτοιου είδους δομή είναι ο *γράφος ροής ελέγχου* (***control flow graph*** – CFG) ο οποίος έχει ως κόμβους τα ονομαζόμενα *βασικά τμήματα* (***basic blocks***)
- Θα δούμε πρώτα την κατασκευή του γράφου ροής ελέγχου και στη συνέχεια θα μελετήσουμε τις τεχνικές βελτιστοποίησης που βασίζονται στην επεξεργασία του
- Οι τεχνικές αυτές εφαρμόζουν ανάλυση της συμπεριφοράς του προγράμματος κατά την εκτέλεση

Βασικά τμήματα και γράφοι ροής ελέγχου (2/6)

ΟΡΙΣΜΟΣ

Βασικό τμήμα – basic block. Είναι μία ακολουθία από συνεχόμενες εντολές ενδιάμεσου με τις εξής ιδιότητες:

(a) Η μόνη εντολή jump που μπορεί να υπάρχει είναι η τελευταία εντολή
(b) Δεν υπάρχει εξωτερική εντολή jump σε ενδιάμεση εντολή του τμήματος
(c) Δεν τερματίζει το πρόγραμμα σε καμία εντολή του τμήματος
(d) Είναι η μεγαλύτερη ακολουθία εντολών που έχει αυτές τις ιδιότητες

Η εκτέλεση ενός βασικού τμήματος αρχίζει με την πρώτη εντολή ενώ συμπληρώνεται όταν εκτελεστεί και η τελευταία εντολή, εκτελώντας όλες τις εντολές με τη σειρά.

Μία ενδιάμεση εντολή της μορφής ***x = y op z*** λέμε ότι κάνει ***define*** το *x* και ***use*** τα *y* και *z*. Ένα όνομα είναι ***live*** σε ένα κάποιο σημείο εάν η τιμή του χρησιμοποιείται (διαβάζεται) μετά από αυτό το σημείο , ίσως σε κάποιο άλλο βασικό τμήμα.

Βασικά τμήματα και γράφοι ροής ελέγχου (3/6)

ΚΑΤΑΣΚΕΥΗ

I. ***Τεμαχισμός σε βασικά τμήματα***. Ως είσοδος είναι μία ακολουθία από εντολές ενδιάμεσου κώδικα (quads). Η έξοδος είναι μία λίστα από βασικά τμήματα με κάθε quad να ανήκει σε ένα και μόνο βασικό τμήμα.

1. Πρώτα προσδιορίζουμε το σύνολο των *leaders*, τις πρώτες εντολές των βασικών τμημάτων, ως εξής:
   a. Η πρώτη εντολή είναι *leader*
   b. Κάθε εντολή που είναι προορισμός αλλαγής ροής ελέγχου ή έπεται τέτοιες εντολής είναι *leader*
2. Για κάθε *leader*, το βασικό του τμήμα περιέχει το ίδιο καθώς και όλες τις εντολές που ακολουθούν μέχρι την εντολή ακριβώς πριν τον επόμενο *leader*

II. ***Δημιουργία γράφου ροής ελέγχου***. Περιέχει βασικά τμήματα. Ως αρχικός κόμβος ορίζεται το βασικό τμήμα του οποίου leader είναι η πρώτη εντολή. Έπειτα εισάγουμε μία κατευθυνόμενη ακμή μεταξύ δύο βασικών τμημάτων $A \rightarrow B$ εάν:

1. Υπάρχει εντολή αλλαγής ροής ελέγχου από την τελευταία εντολή του *A* στην πρώτη εντολή του *B*, ή
2. Ο *leader* του *B* έπεται της τελευταίας εντολής *A* στον ενδιάμεσο κώδικα, η οποία τελευταία εντολή του *A* δεν είναι εντολή αλλαγής ροής ελέγχου χωρίς συνθήκη.

Βασικά τμήματα και γράφοι ροής ελέγχου (4/6)

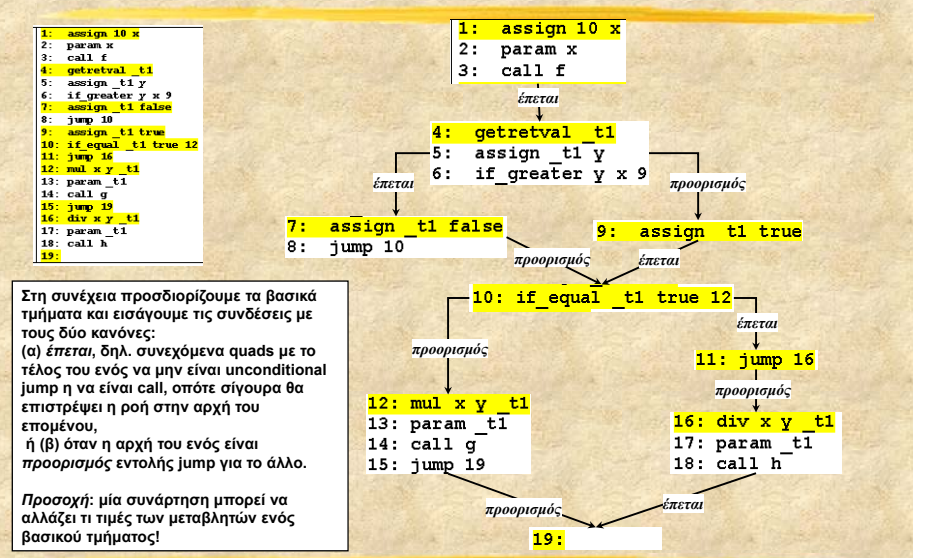
ΠΑΡΑΔΕΙΓΜΑ

Πρώτα προσδιορίζουμε τους leaders

```
x = 10;
y = f(x);
if (y > x)
     g(x*y);
else
     h(x/y);
```

```
1:  assign 10 x
2:  param x
3:  call f
4:  getretval _t1
5:  assign _t1 y
6:  if_greater y x 9
7:  assign _t1 false
8:  jump 10
9:  assign _t1 true
10: if_equal _t1 true 12
11: jump 16
12: mul x y _t1
13: param _t1
14: call g
15: jump 19
16: div x y _t1
17: param _t1
18: call h
19:
```


Βασικά τμήματα και γράφοι ροής ελέγχου (5/6)



Στη συνέχεια προσδιορίζουμε τα βασικά τμήματα και εισάγουμε τις συνδέσεις με τους δύο κανόνες:
(α) *έπεται*, δηλ. συνεχόμενα quads με το τέλος του ενός να μην είναι unconditional jump η να είναι call, οπότε σίγουρα θα επιστρέψει η ροή στην αρχή του επομένου,
ή (β) όταν η αρχή του ενός είναι *προορισμός* εντολής jump για το άλλο.

Προσοχή: μία συνάρτηση μπορεί να αλλάζει τι τιμές των μεταβλητών ενός βασικού τμήματος!

Βασικά τμήματα και γράφοι ροής ελέγχου (6/6)

- Θα δούμε ορισμένους μετασχηματισμούς στα βασικά τμήματα, δηλ. μετασχηματισμούς τοπικού χαρακτήρα – *local optimizations*
- Κάθε βασικό τμήμα υπολογίζει ένα σύνολο από εκφράσεις
  - Ουσιαστικά αυτές οι εκφράσεις είναι οι τιμές των ονομάτων που είναι *live* μετά την έξοδο από το τμήμα
- Δύο βασικά τμήματα είναι ισοδύναμα εάν υπολογίζουν ακριβώς τις ίδιες εκφράσεις
  - Οι μετασχηματισμοί βελτιστοποίησης εφαρμόζουν την τροποποίηση ενός βασικού τμήματος ώστε να προκύψει ένα ισοδύναμο αλλά καλύτερο

Περιεχόμενα

- Εισαγωγή
- Βελτιστοποίηση μικρής κλίμακας
- Βασικά τμήματα και γράφοι ροής ελέγχου
- *Τεχνικές βελτιστοποίησης τμημάτων*

Τεχνικές βελτιστοποίησης τμημάτων (1/19)

- Δυσκολότερα
  - Εξάλειψη κοινών εκφράσεων
  - Διανομή αντιγράφων
  - Εξάλειψη νεκρού κώδικα
- Ευκολότερα
  - Διανομή σταθερών τιμών
  - Αλγεβρικές απλοποιήσεις
  - Αποδυνάμωση εκφράσεων

Τεχνικές βελτιστοποίησης τμημάτων (2/19)

- *Στο πρόγραμμα εμπλέκονται δύο ειδών μεταβλητές (1/2)*
  - Κρυφές μεταβλητές: εισάγονται από τον μεταγλωττιστή, με τις εξής ιδιότητες
    - Μεταφορά τιμών μόνο σε ένα βασικό τμήμα – μπορούμε να παρατηρήσουμε ότι δεν έχουμε κρυφές μεταβλητές οι οποίες γίνονται write σε έναν βασικό τμήμα και η τιμή τους γίνεται read σε ένα άλλο
    - Χρησιμοποιούνται για τον τεμαχισμό υπολογισμού εκφράσεων σε quads – αυτό λέγεται και ***instruction flattening***
    - Σε ένα βασικό τμήμα εκχωρούνται τιμή μία φορά, εκτός από τις περιπτώσεις επαναχρησιμοποίησης
    - *Εισάγονται λόγω μετασχηματισμών βελτιστοποίησης*

Τεχνικές βελτιστοποίησης τμημάτων (3/19)

- *Στο πρόγραμμα εμπλέκονται δύο ειδών μεταβλητές (2/2)*
  - Μεταβλητές του προγράμματος: ορίζονται από το χρήστη
    - Μπορούν να εκχωρηθούν τιμές πολλές φορές
    - Μεταφέρουν τιμές μεταξύ διαφορετικών βασικών τμημάτων, δηλ. μπορούν να γίνουν write σε ένα βασικό τμήμα και έπειτα read σε ένα άλλο
- ✓ Από τα παραπάνω προκύπτει η τοπικότητα χρήσης των κρυφών μεταβλητών εσωτερικά ενός βασικού τμήματος
  - Μπορούμε να μετασχηματίσουμε ένα βασικό τμήμα σε ένα ισοδύναμο (δηλ. ένα που υπολογίζει την ίδια έκφραση) απλώς μετονομάζοντας τις κρυφές μεταβλητές
  - Ένα τέτοιο βασικό τμήμα με μετονομασία κρυφών μεταβλητών ονομάζεται *τμήμα κανονικής μορφής* – ***normal-form block***

Τεχνικές βελτιστοποίησης τμημάτων (4/19)

- *Εξάλειψη κοινών εκφράσεων – common sub expression elimination*
- Έστω ένα κανονικοποιημένο βασικό τμήμα. Τότε όλες οι εντολές είναι ουσιαστικά της μορφής:
  - *var* = *var* **op** *var*
    - Παρατηρήστε ότι αυτή η μορφή καλύπτει πλήρως και την περίπτωση πρόσβασης σε στοιχεία πίνακα, αρκεί να δώσουμε ρόλους *array*, *index*, *content* ή *result* στις μεταβλητές
  - *var* = **op** *var*
  - *var* = *var*
- Θα δούμε ένα τρόπο συμβολικής εκτέλεσης των εντολών ενός βασικού τμήματος. Σε αυτή την εκτέλεση:
  - Θα εξάγουμε συμπεράσματα για τις τιμές των μεταβλητών του τμήματος
  - Θα εντοπίσουμε χαρακτηριστικά της εκτέλεσης που μας ενδιαφέρουν κατά περίπτωση

Τεχνικές βελτιστοποίησης τμημάτων (5/19)

- Θα χρησιμοποιήσουμε *συμβολικές* η αλλιώς *εικονικές* τιμές (symbolic or virtual values)
  - Αυτό σημαίνει ότι θα έχουμε σύμβολα για τις τιμές με κάθε διαφορετικό σύμβολο να αντιστοιχεί σε διαφορετική τιμή
  - Αυτό ο τρόπος απαρίθμησης τιμών με διαφορετικά διακριτά σύμβολα ονομάζεται αρίθμηση τιμών – ***value numbering***
    - Υλοποιείται εύκολα με αναπαράσταση λογικής τιμής ως μοναδικό ακέραιο αριθμό
- Θα εφαρμόσουμε εξομοίωση της εκτέλεσης ενός βασικού τμήματος
  - εκχωρώντας μία εικονική τιμή σε κάθε μεταβλητή και έκφραση
  - το χαρακτηριστικό που θέλουμε να εντοπίσουμε είναι *ποιες μεταβλητές και εκφράσεις φέρουν την ίδια εικονική τιμή*
  - η χρηστικότητα της τεχνικής αυτής είναι για εξάλειψη κοινών εκφράσεων

Τεχνικές βελτιστοποίησης τμημάτων (6/19)

- Στην τεχνική θα εφαρμόσουμε επιπλέον δύο βασικούς μετασχηματισμούς πάνω στον αυθεντικό κώδικα ενός βασικού τμήματος
  - Χρήση επιπλέον κρυφών μεταβλητών για την αποθήκευση των υπολογιζόμενων τιμών
  - Αντικατάσταση των εκφράσεων με την τις αντίστοιχες κρυφές μεταβλητές όταν η τιμή μίας χρησιμοποιούμενης έκφρασης έχει ήδη υπολογιστεί
- Κάθε έκφραση από την εντολή ενδιάμεσου κώδικα θα γράφεται σε νέα κρυφή μεταβλητή
- Κάθε υπάρχουσα μεταβλητή θα τις προσδίδουμε για πρώτη φορά μία νέα εικονική τιμή v_i

Τεχνικές βελτιστοποίησης τμημάτων (7/19)

Αρχικό βασικό τμήμα	Μετασχηματισμένο βασικό τμήμα
`a = x+y;`	`a = x+y;`
`b = z+a;`	`t1 = a;`
`b = x+b;`	`b = z+a;`
`c = z+a;`	`t2 = b;`
	`b = x+b;`
	`t3 = b;`
	`c = t2;`

Συμβολική εκτέλεση βασικού τμήματος. Βασίζεται στην ακολουθιακή επεξεργασία των εντολών ενός βασικού τμήματος με ταυτόχρονη παραγωγή του μετασχηματισμένου βασικού τμήματος, χρησιμοποιώντας τρεις πίνακες αντιστοίχησης. Η επεξεργασία / εξομοίωση ακολουθεί τους εξής κανόνες:

- Κάθε νέα προκύπτουσα τιμή λόγω της παρούσας εντολής εκχωρείται σε μία νέα κρυφή μεταβλητή. Έτσι το **a=x+y;** γίνεται **a=x+y; t=a;**.
  - Η κρυφή μεταβλητή διατηρεί την τιμή της αρχικής έκφρασης ακόμη και αν η αυθεντική μεταβλητή που αρχικά εκχωρήθηκε το αποτέλεσμα έχει τώρα αλλάξει. Έτσι το **a=x-y; a=z-a; c=x-y;** μετασχηματίζεται σε **a=x-y; t1=a; a=z-a; t2=a; c=t1;**

Οι πίνακες αντιστοίχησης κατασκευάζονται και αλλάζουν δυναμικά κατά τη συμβολική εκτέλεση / εξομοίωση των εντολών ενός βασικού τμήματος

Πίνακες αντιστοίχησης

Var → Value	Expr → Value	Expr → Temp
`x : v1`	`v1+v2 : v3`	`v1+v2 : t1`
`y : v2`	`v3+v4 : v5`	`v3+v4 : t2`
`a : v3`	`v1+v5 : v6`	`v1+v5 : t3`
`z : v4`		
`b : v5 v6`		
`c : v5`		

Τεχνικές βελτιστοποίησης τμημάτων (8/19)

Οι πίνακες αντιστοίχησης χρησιμοποιούνται ως εξής

- *Var → Value*, περιέχει τη συμβολική τιμή μίας μεταβλητής ανά πάσα στιγμή, με αρχικά επιλεγόμενη μοναδική εικονική τιμή. Αλλάζει περιεχόμενο τιμής ανάλογα με την επεξεργασία της εκάστοτε εντολής.

- *Expr → Value*, προσδιορίζει τη συμβολική τιμή εκφράσεων. Εάν για την εκάστοτε εντολή η έκφραση, π.χ., *a+z*, με τις συμβολικές τιμές των μεταβλητών, π.χ. *v3+v4*, δεν υφίσταται στον πίνακα, δημιουργείται μία νέα εισαγωγή με νέα εικονική τιμή, π.χ. *v5*, ενώ ταυτόχρονα έχουμε και εισαγωγή / μεταβολή τιμής για την εκχωρούμενη μεταβλητή, δηλ. αλλαγή του πίνακα *Var →Value*.

- *Expr → Temp*, προσδιορίζει την κρυφή μεταβλητή που περιέχει το αποτέλεσμα μία έκφρασης. Εισαγωγές γίνονται κάθε φορά που το αποτέλεσμα μίας έκφρασης το αποθηκεύουμε σε νέα κρυφή μεταβλητή.
  - Όταν βρίσκουμε μία έκφραση με συμβολικές τιμές για την οποία έχουμε στοιχείο σε αυτό τον πίνακα, τότε αυτή η έκφραση δεν χρειάζεται να υπολογιστεί, αλλά χρησιμοποιούμε απευθείας την κρυφή μεταβλητή.

Τεχνικές βελτιστοποίησης τμημάτων (9/19)

Κύριες ιδιότητες

1. Εντοπίζει κοινές εκφράσεις ακόμη και όταν εμπλέκονται διαφορετικές μεταβλητές, καθώς υπολογίζει βάσει συμβολικών τιμών και όχι ονομαστώ εμπλεκομένων μεταβλητών. Π.χ.
   - **a=x+y; b=y; c=x+b;** μετασχηματίζεται σε
   - **a=x+y; t=a; b=y; c=t;**

2. Εντοπίζει κοινές εκφράσεις ακόμη και όταν η μεταβλητή που φέρει το αποτέλεσμα αλλάζει περιεχόμενο, καθώς χρησιμοποιεί νέες κρυφές μεταβλητές για την αποθήκευση των επιμέρους αποτελεσμάτων. Π.χ.
   - **a=x-y; a=z-a; c=x-y;** μετασχηματίζεται σε
   - **a=x-y; t1=a; a=z-a; t2=a; c=t1;**

Τεχνικές βελτιστοποίησης τμημάτων (10/19)

Προβλήματα / θέματα

1. Ο αλγόριθμος οδηγεί στην εισαγωγή μίας νέας κρυφής μεταβλητής για κάθε νέα προκύπτουσα τιμή. Έτσι για κάθε εντολή *a=b **op** c;* θα έχουμε *a = b **op** c; t = a;*.

   Προφανώς κάτι τέτοιο οδηγεί στη δημιουργία πολλών κρυφών μεταβλητών καθώς και σε πολλές εντολές εκχώρησης (copy) σε κρυφές μεταβλητές. Όμως σε πολλές περιπτώσεις οι κρυφές μεταβλητές και οι αντίστοιχες εντολές εκχώρησης δεν είναι απαραίτητες καθώς δεν χρησιμοποιούνται οι μεταβλητές.
   - Αυτό λύνεται με την εφαρμογή βελτιστοποίησης για διανομή αντιγράφων και εξάλειψη νεκρού κώδικα.

2. Καθώς η επεξεργασία αυτή εφαρμόζεται μετά την παραγωγή ενδιάμεσου κώδικα, δεν έχουμε πληροφορία «παρούσας εμβέλειας» ώστε να εισάγουμε τις κρυφές μεταβλητές στον ανάλογο «χώρο» (τοπικές σε συγκεκριμένη συνάρτηση ή καθολικές).
   - Αυτό λύνεται εύκολα με πολλούς τρόπους. Π.χ. διατρέχουμε προς τα πάνω τα quads ξεκινώντας από το leader statement έως ότου βρούμε το πρώτο *funcstart* για το οποίο δεν έχουμε συναντήσει *funcend*.

Τεχνικές βελτιστοποίησης τμημάτων (11/19)

- *Διανομή αντιγράφων – copy propagation*. Εφαρμόζεται μετά την εξάλειψη κοινών εκφράσεων με σκοπό να περιορίσει όσο το δυνατόν τη χρήση των κρυφών μεταβλητών
  - Ο στόχος είναι να χρησιμοποιεί την αυθεντική μεταβλητή, παρά την κρυφή, εάν αυτό είναι εφικτό
  - Αυτό σημαίνει το προσδιορισμό των αυθεντικών μεταβλητών που δεν αλλάζουν μεταξύ των εντολών εκχώρησης και του σημείου χρήσης της υπολογιζόμενης τιμής
    - Εάν δεν επέλθει κάποια μεταβολή, τότε χρησιμοποιούμε την αυθεντική μεταβλητή
  - Εφαρμόζει και πάλι εξομοίωση της εκτέλεσης του μετασχηματισμένου βασικού τμήματος

Τεχνικές βελτιστοποίησης τμημάτων (12/19)

Βασίζεται στη διατήρηση δύο πινάκων *Temp → Var*, που ορίζει ποια μεταβλητή να χρησιμοποιηθεί στη θέση μίας κρυφής μεταβλητής, και *Var → Temp*, που αντίστροφα προσδιορίζει την κρυφή μεταβλητή που αντιστοιχεί στο συγκεκριμένο *Var* βάσει του πρώτου πίνακα. Χρησιμοποιούνται ως εξής:

- Με κάθε εκχώρηση σε μία αυθεντική μεταβλητή, εάν υπάρχει το *Var→Temp* και είναι έστω *t*, τότε θέτουμε στον άλλο πίνακα *Temp→Var* για το κλειδί *t* την κρυφή μεταβλητή *t*, δηλ.*t→t*, ενώ αφαιρούμε την προηγούμενη αντιστοίχηση από τον *Var→Temp*, διαγράφουμε το *x→t*.
- Με κάθε εκχώρηση σε κρυφή μεταβλητή *t* από αυθεντική *x* εισάγουμε τα *x→t* και *t→x* στους πίνακες *Var→Temp* και *Temp→Var* αντίστοιχα.

Αρχικό μετασχηματισμένο βασικό τμήμα	Νέο μετασχηματισμένο βασικό τμήμα	Πίνακες	
		Temp → Var	*Var → Temp*

`a = x+y;`
`t1 = a;`
`b = z+a;`
`t2 = b;`
`b = x+b;`
`t3 = b;`
`c = t1;`
	`a=x+y;`		

Τεχνικές βελτιστοποίησης τμημάτων (13/19)

Αρχικό μετασχηματισμένο βασικό τμήμα	Νέο μετασχηματισμένο βασικό τμήμα	Πίνακες	
		Temp → Var	*Var → Temp*

`a = x+y;`
`t1 = a;`
`b = z+a;`
`t2 = b;`
`b = x+b;`
`t3 = b;`
`c = t1;`
	`a=x+y;`
	`t1=a;`
		$\{ t1 : a \}$	$\{ a : t1 \}$

Αρχικό μετασχηματισμένο βασικό τμήμα	Νέο μετασχηματισμένο βασικό τμήμα	Πίνακες	
		Temp → Var	*Var → Temp*

`a = x+y;`
`t1 = a;`
`b = z+a;`
`t2 = b;`
`b = x+b;`
`t3 = b;`
`c = t1;`
	`a=x+y;`
	`t1=a;`
	`b = z+a;`
	`t2 = b;`
		$\{ t1 : a \}$,
		$\{ t2 : b \}$
			$\{ a : t1 \}$,
			$\{ b : t2 \}$

Τεχνικές βελτιστοποίησης τμημάτων (14/19)

Αρχικό μετασχηματισμένο βασικό τμήμα	Νέο μετασχηματισμένο βασικό τμήμα	Πίνακες	
		Temp → Var	*Var → Temp*

`a = x+y;`
`t1 = a;`
`b = z+a;`
`t2 = b;`
`b = x+b;`
`t3 = b;`
`c = t1;`
	`a=x+y;`
	`t1=a;`
	`b = z+a;`
	`t2 = b;`
	`b = x+b;`
		$\{ t1 : a \}$,
		$\{ t2 : t2 \}$
			$\{ a : t1 \}$,
			~~$\{ b : t2 \}$~~

Αρχικό μετασχηματισμένο βασικό τμήμα	Νέο μετασχηματισμένο βασικό τμήμα	Πίνακες	
		Temp → Var	*Var → Temp*

`a = x+y;`
`t1 = a;`
`b = z+a;`
`t2 = b;`
`b = x+b;`
`t3 = b;`
`c = t1;`
	`a=x+y;`
	`t1=a;`
	`b = z+a;`
	`t2 = b;`
	`b = x+b;`
	`t3=b;`
	`c = a;`
		$\{ t1 : a \}$,
		$\{ t2 : t2 \}$,
		$\{ t3 : b \}$
			$\{ a : t1 \}$,
			$\{ b : t3 \}$

Τεχνικές βελτιστοποίησης τμημάτων (15/19)

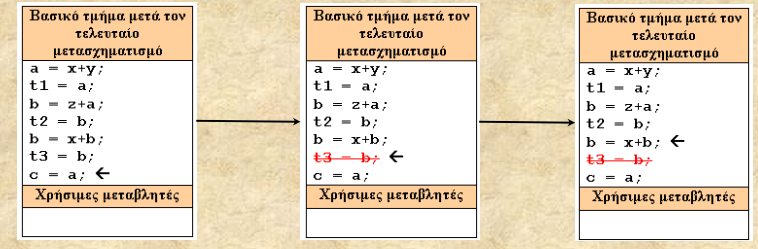
- *Εξάλειψη νεκρού κώδικα*. Ο σκοπός της βελτιστοποίησης αυτής είναι όπως έχουμε ήδη αναφέρει η απομάκρυνση εκχωρήσεων σε μεταβλητές που δεν χρησιμοποιούνται ποτέ
  - Εδικά στην προηγούμενη τεχνική, η διανομή αντιγράφων τιμών δεν επηρεάζει τον αριθμό των κρυφών μεταβλητών
  - Ενώ παραμένουν αρκετές κρυφές μεταβλητές οι οποίες δεν χρησιμοποιούνται ποτέ
  - Η εξάλειψη νεκρού κώδικα θα απομακρύνει τέτοιου είδους εντολές και κατά συνέπεια μπορούμε στη συνέχεια να απομακρύνουμε εντελώς αυτές τις κρυφές μεταβλητές από τον κώδικα μειώνοντας τις ανάγκες μνήμης του προγράμματος
  - Η τεχνική που θα δούμε εφαρμόζεται και πάλι σε βασικά τμήματα και είναι πολύ καλύτερη από την ευρεστική πού έχουμε δει για peephole optimization
  - Δεν θα εξομοιώσουμε την εκτέλεση αλλά απλώς θα εντοπίσουμε τις «χρήσιμες μεταβλητές»

Τεχνικές βελτιστοποίησης τμημάτων (16/19)

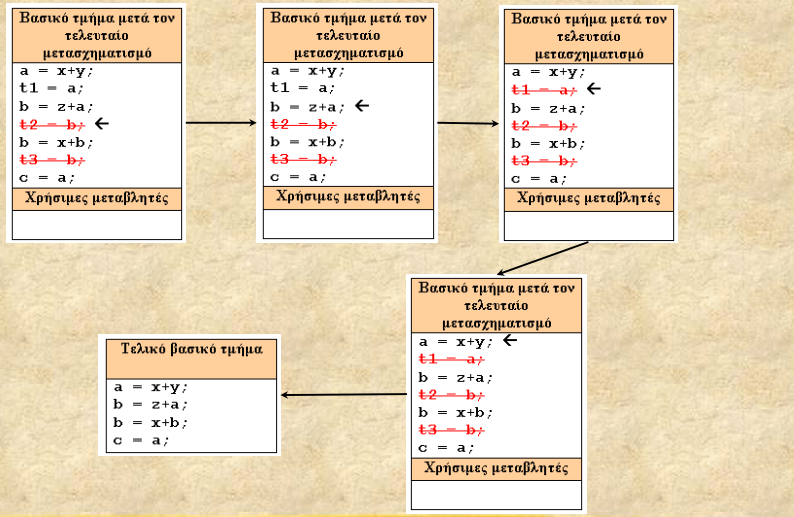
Dead code elimination

Εξάλειψη νεκρού κώδικα. Η τακτική είναι σχετικά απλή και σκιαγραφείται αλγοριθμικά ως εξής:

- Διέτρεξε τις εντολές του βασικού τμήματος από το τέλος προς την αρχή.
- Βάλε κάθε κρυφή μεταβλητή που εμφανίζεται στο δεξί τμήμα (όρισμα) σε μία λίστα «χρήσιμων μεταβλητών».
- Αφαίρεσε κάθε εντολή εκχώρησης σε κρυφή μεταβλητή η οποία δεν ανήκει στο σύνολο των «χρήσιμων μεταβλητών».



Τεχνικές βελτιστοποίησης τμημάτων (17/19)



Τεχνικές βελτιστοποίησης τμημάτων (18/19)

- *Διανομή σταθερών τιμών – constant propagation*. Εφαρμόζεται εύκολα σε επίπεδο βασικών τμημάτων.
  - Βασίζεται στη διατήρηση μίας λίστας από μεταβλητές οι οποίες έχουν εκχωρηθεί σταθερές τιμές
  - Για κάθε εντολή εκχώρησης όπου όλα τα ορίσματα είναι ή σταθερές τιμές είτε μεταβλητές που ανήκουν στη λίστα:
    - υπολογίζουμε το αποτέλεσμα
    - βάζουμε την μεταβλητή στη λίστα μαζί με την υπολογιζόμενη τιμή
    - μαρκάρουμε την εντολή εκχώρησης για διαγραφή
  - Για κάθε εντολή όπου το όρισμα είναι μεταβλητή από τη λίστα το αντικαθιστούμε με την τιμή του
  - Στο τέλος του βασικού τμήματος, για κάθε μη κρυφή μεταβλητή του βασικού τμήματος εισάγουμε εντολές εκχώρησης με την αντίστοιχη σταθερή τιμή
    - Αυτό είμαστε υποχρεωμένοι να το κάνουμε καθώς μπορούν οι μη κρυφές μεταβλητές να χρησιμοποιούνται σε πολλά βασικά τμήματα

Τεχνικές βελτιστοποίησης τμημάτων (19/19)