GLU$^\sharp$/C++ : Θεωρία και Υλοποίηση*

Νικόλαος Σ. Παπασπύρου
(nickie@softlab.ntua.gr)

Ιωάννης Τ. Κασσιός
(ykass@softlab.ntua.gr)

Εθνικό Μετσόβιο Πολυτεχνείο
Τμήμα Ηλεκτρολόγων Μηχ. και Μηχ. Υπολογιστών
Τομέας Πληροφορικής, Εργαστήριο Τεχνολογίας Λογισμικού
Πολυτεχνειούπολη, 15780 Ζωγράφου.

Περίληψη

Η ενσωμάτωση μιας μικρής αλλά εκφραστικής γλώσσας *νοηματικού συναρτησιακού* προγραμματισμού σε μια καθιερωμένη γλώσσα *αντικειμενοστρεφούς* προγραμματισμού έχει ως αποτέλεσμα το συνδυασμό δυο ριζικά διαφορετικών προγραμματιστικών μοντέλων. Σε αυτή την εργασία ορίζουμε με τυπικό τρόπο τη σύνταξη και τη σημασιολογία της GLU$^\sharp$, η οποία μπορεί να θεωρηθεί ως ο νοηματικός πυρήνας των γλωσσών Lucid και GLU. Στη συνέχεια περιγράφουμε την υλοποίηση της GLU$^\sharp$ μέσω της ενσωμάτωσής της στη C++. Τέλος, με μια σειρά παραδειγμάτων, διατυπώνουμε τον ισχυρισμό ότι ο συνδυασμός των δυο μοντέλων προγραμματισμού όχι απλώς είναι συμβατός αλλά αποφέρει μια νέα, πολύ ενδιαφέρουσα και ιδιαίτερα ισχυρή γλώσσα.

1 Εισαγωγή

Η δομή της εργασίας είναι η ακόλουθη. Στην ενότητα 1 παρουσιάζεται μια σύντομη ανασκόπηση των γλωσσών νοηματικού προγραμματισμού Lucid και GLU και η περιγραφή του αντικειμένου της παρούσας έρευνας. Παράλληλα δίνονται αναφορές στη σχετική βιβλιογραφία. Στην ενότητα 2 δίνεται ο ορισμός της γλώσσας GLU$^\sharp$, ενώ η ενότητα 3 ασχολείται με την υλοποίησή της GLU$^\sharp$ μέσω της ενσωμάτωσής της στη C++. Τέλος, στην ενότητα 4 δίνονται παραδείγματα προγραμμάτων GLU$^\sharp$/C++ ενώ στην ενότητα 5 γίνεται μια πρώιμη αξιολόγηση της προσπάθειας.

1.1 Lucid, GLU και νοηματικός προγραμματισμός

Η *Lucid* [Ashc77, Wadg85] είναι μια γλώσσα προγραμματισμού βασισμένη στη ροή δεδομένων (dataflow programming language). Η αρχική της σχεδίαση έγινε από τους Ed Ashcroft και Bill Wadge, το 1974 στο University of Waterloo του Καναδά. Ξεκίνησε ως μια απλή, όχι προστακτική γλώσσα χρονικού προγραμματισμού που αποσκοπούσε κυρίως στην απόδειξη ιδιοτήτων των προγραμμάτων, κατέληξε όμως να εξελιχθεί σε μια εγγενώς παράλληλη πολυδιάστατη γλώσσα

*Στην παρούσα εργασία περιγράφεται μέρος των αποτελεσμάτων έρευνας που πραγματοποιήθηκε στο πλαίσιο του ερευνητικού προγράμματος με τίτλο "Εκτελέσιμες Νοηματικές Γλώσσες και Ευφυείς Εφαρμογές Πολυμέσων, Υπερμέσων και Εικονικής Πραγματικότητας", το οποίο χρηματοδοτείται από το πρόγραμμα ενίσχυσης ερευνητικού δυναμικού (ΠΕΝΕΔ'99) της Γενικής Γραμματείας Έρευνας και Τεχνολογίας. Κωδικός προγράμματος: 99ΕΔ265.

[Du93]. Επιπλέον, ένα νέο μοντέλο προγραμματισμού που ονομάζεται *νοηματικός προγραμματισμός* (intensional programming) έχει τις απαρχές του στη Lucid και σημαντικές εφαρμογές στις περιοχές των προγραμμάτων πραγματικού χρόνου [Faus88, Plai93a], των επιστημονικών προγραμμάτων [Rao94, Paqu99], των χρονικών βάσεων δεδομένων [Paqu94] και λογιστικών φύλλων [Du90], της επεξεργασίας σημάτων [Agi96], των κατηγορικών γραμματικών [Tao94], της διαχείρισης εκδόσεων λογισμικού [Plai93b] καθώς και της δημιουργίας ιστοσελίδων στο World Wide Web [Yild97]. Η γλώσσα αναπτύχθηκε κυρίως στη δεκαετία του 1980 και στην αρχή αυτής του 1990.

Η σημασιολογία της Lucid είναι ριζικά διαφορετική από αυτή άλλων γλωσσών προγραμματισμού, όπως για παράδειγμα της C ή της ML. Στη Lucid, ο προγραμματιστής ορίζει φίλτρα ή συναρτήσεις μετασχηματισμού που εφαρμόζονται πάνω σε χρονικά μεταβαλλόμενες ροές δεδομένων. Η Lucid υποστηρίζει ένα πολύ μικρό σύνολο τύπων δεδομένων (ακέραιους αριθμούς, πραγματικούς αριθμούς και σύμβολα) και είναι σαφώς επηρεασμένη από το μοντέλο συναρτησιακού προγραμματισμού. Κύρια χαρακτηριστικά της είναι η απουσία παρενεργειών (side effects) στις εκφράσεις και η οκνηρή αποτίμηση (lazy evaluation).

Η σύνταξη της Lucid είναι επίσης διαφορετική από αυτή των καθιερωμένων γλωσσών προγραμματισμού. Το γεγονός αυτό δεν είναι τυχαίο αλλά αντικατοπτρίζει την επιλογή των σχεδιαστών της γλώσσας να αποθαρρύνουν τους προγραμματιστές από την εφαρμογή τεχνικών προστακτικού προγραμματισμού και να τους προσανατολίσουν προς τον προγραμματισμό ροών δεδομένων, που συμπεριφέρονται ως άπειρα αντικείμενα.

Ο διάδοχος της Lucid, που θεωρείται σήμερα κύριος εκπρόσωπος του μοντέλου πολυδιάστατου νοηματικού προγραμματισμού, είναι η γλώσσα *GLU* (Granular Lucid) [Ashc91, Ashc95]. Η GLU διευκολύνει τη γρήγορη ανάπτυξη πρωτοτύπων για παράλληλες εφαρμογές [Agi96, Rao94]. Υλοποιήσεις της είναι διαθέσιμες για τις περισσότερες πλατφόρμες Unix από το Εργαστήριο Επιστήμης Υπολογιστών του SRI International (`http://www.csl.sri.com/`).

1.2 Αντικείμενο της έρευνας

Η παρούσα έρευνα αποσκοπεί αφενός στη μελέτη της σύνταξης και της σημασιολογίας των νοηματικών γλωσσών προγραμματισμού και αφετέρου στη μελέτη πρόσφορων τρόπων υλοποίησης τέτοιων γλωσσών.

Οι νοηματικές γλώσσες διαθέτουν αυξημένες δυνατότητες περιγραφής δυναμικών συστημάτων, των οποίων η κατάσταση μεταβάλλεται σε μία ή περισσότερες διαστάσεις. Έχουν ως θεωρητική τους βάση τη *νοηματική λογική* (intensional logic) [Thom74, Dowt81], η εκφραστικότητα της οποίας σύντομα οδήγησε σε μια προσπάθεια απελευθέρωσης του προγραμματιστή από την ενασχόληση με εργασίες χαμηλού επιπέδου, που δεν σχετίζονται άμεσα με το προς επίλυση πρόβλημα. Οι νοηματικές γλώσσες που έχουν σχεδιαστεί μέχρι σήμερα ενσωματώνουν συνήθως χαρακτηριστικά της νοηματικής λογικής σε γλώσσες που εκφράζουν το μοντέλο είτε του *λογικού προγραμματισμού* (logic programming) είτε του *συναρτησιακού προγραμματισμού* (functional programming). Στην παρούσα εργασία το ερευνητικό ενδιαφέρον εστιάζεται σε νοηματικές γλώσσες συναρτησιακής φύσης. Ως αξιόλογες εργασίες στην περιοχή του νοηματικού λογικού προγραμματισμού αξίζει να αναφερθούν οι [Wadg88, Orgu91, Orgu92, Orgu97].

Τα τελευταία χρόνια έντονο ερευνητικό ενδιαφέρον περιστρέφεται γύρω από τις *γλώσσες εξειδικευμένου πεδίου εφαρμογών* (domain-specific languages — DSL). Το χαρακτηριστικό των γλωσσών αυτών είναι η ικανότητά τους να αποδίδουν με ακρίβεια τη σημασιολογία ενός πεδίου εφαρμογών, το εύρος του οποίου ποικίλει, και να επιτρέπουν στον προγραμματιστή να αναπτύσσει εφαρμογές

σε αυτό το πεδίο εύκολα, γρήγορα και αποδοτικά. Τα προγράμματα που παράγονται με τη χρήση DSL είναι συνήθως μικρά σε μέγεθος και είναι εύκολο κανείς να τα καταλάβει, να μελετήσει τις ιδιότητές τους αλλά και να τα συντηρήσει. Από την άλλη όμως πλευρά, απαιτείται στην πράξη σημαντικό κόστος για τη δημιουργία του υπόβαθρου που είναι αναγκαίο για τον προγραμματισμό σε DSL. Σε αυτό εμπεριέχεται το κόστος της υλοποίησης των γλωσσών αλλά και της μετάβασης των προγραμματιστών από ένα προγραμματιστικό περιβάλλον γενικού σκοπού στο περιβάλλον των DSL.

Για την εξομάλυνση αυτών των προβλημάτων, έχει προταθεί η κατασκευή DSL *ενσωματωμένων* (embedded) σε υπάρχουσες γλώσσες προγραμματισμού γενικού σκοπού [Huda98]. Με τον τρόπο αυτό αφενός επιτυγχάνεται η επαναχρησιμοποίηση της σύνταξης, σημασιολογίας, κώδικα, εργαλείων υλοποίησης και γενικά όλου του υπόβαθρου μιας υπάρχουσας γλώσσας προγραμματισμού, αφετέρου διευκολύνεται ο προγραμματισμός εξειδικευμένων εφαρμογών με τη χρήση των ενσωματωμένων εξειδικευμένων γλωσσικών στοιχείων.

Παρά το γεγονός ότι οι νοηματικές γλώσσες έχουν ευρύ πεδίο εφαρμογών, η ενσωμάτωσή τους σε υπάρχουσες γλώσσες προγραμματισμού παρέχει τα σημαντικά πλεονεκτήματα που αναφέρθηκαν προηγουμένως. Επιπλέον, γλώσσες όπως η Lucid είναι ουσιαστικά βασισμένες πάνω σε ένα πυρήνα μικρού μεγέθους προερχόμενο από την νοηματική λογική, ο οποίος τις καθιστά κατάλληλες για τον προγραμματισμό δυναμικών συστημάτων. Τα επιπλέον χαρακτηριστικά που διαθέτουν είναι δανεισμένα από κάποια άλλη γλώσσα προγραμματισμού (συναρτησιακή, στην περίπτωση της Lucid).

Ακολουθώντας αυτή τη λογική, στο πλαίσιο της έρευνας που περιγράφεται στην παρούσα εργασία έγινε κατ' αρχήν προσπάθεια να απομονωθεί και να μελετηθεί ο πυρήνας των νοηματικών χαρακτηριστικών των γλωσσών Lucid και GLU. Ο πυρήνας αυτός αποτελεί από μόνος του μια μικρή νοηματική γλώσσα συναρτησιακού προγραμματισμού, την οποία ονομάζουμε GLU$^\sharp$. Η σύνταξη και η σημασιολογία της GLU$^\sharp$ ορίζονται στην ενότητα 2. Στη συνέχεια, η GLU$^\sharp$ ενσωματώθηκε στη γλώσσα προγραμματισμού γενικού σκοπού C++. Η επιλογή της τελευταίας βασίστηκε στα ακόλουθα επιχειρήματα:

- Η C++ είναι μια ιδιαίτερα δημοφιλής γλώσσα προγραμματισμού που υποστηρίζει το αντικειμενοστρεφές μοντέλο ανάπτυξης λογισμικού. Χρησιμοποιείται ευρέως στην πράξη για τον προγραμματισμό συστημάτων λογισμικού μεγάλης κλίμακας και υπάρχει διαθέσιμος μεγάλος αριθμός βιβλιοθηκών και εργαλείων.

- Η C++ προσφέρεται για την ενσωμάτωση DSL. Σε αυτό συνιστούν αφενός η εκφραστικότητα του αντικειμενοστρεφούς μοντέλου προγραμματισμού, αφετέρου κάποια ιδιαίτερα χαρακτηριστικά της C++ όπως ο παραμετρικός πολυμορφισμός με τη χρήση των templates και ο μηχανισμός διαχείρισης των εξαιρέσεων (exceptions). Επιπλέον, η ύπαρξη πληθώρας τελεστών και η δυνατότητα υπερφόρτωσης (overloading) αυτών των τελεστών επιτρέπουν την ενσωμάτωση τμημάτων κώδικα DSL με σύνταξη απλή, φυσική και εύκολη στην κατανόηση. Η ύπαρξη του προεπεξεργαστή (preprocessor) της C++ διευκολύνει ακόμα περισσότερο προς αυτή την κατεύθυνση.

Η σχεδίαση της GLU$^\sharp$ αποσκοπεί κατ' αρχήν, όπως ήδη αναφέρθηκε, στην απομόνωση των νοηματικών χαρακτηριστικών των γλωσσών Lucid και GLU. Το τελικό αποτέλεσμα είναι μια γλώσσα σημαντικά απλοποιημένη σε σχέση με τις δυο προηγούμενες. Χαρακτηριστικά του συναρτησιακού μοντέλου προγραμματισμού έχουν αφαιρεθεί εντελώς. Έχει όμως διατηρηθεί ένας ελάχιστος αριθμός μη νοηματικών χαρακτηριστικών (αριθμητικές και λογικές πράξεις, αναδρομι-

κοί ορισμοί), ούτως ώστε η GLU$^\sharp$ να μπορεί να χρησιμοποιηθεί αυτόνομα για τον προγραμματισμό μη τετριμμένων εφαρμογών.

Η προσθήκη συναρτήσεων (πρώτης ή και υψηλής τάξης) στην GLU$^\sharp$ θα διευκόλυνε σημαντικά τη χρήση της γλώσσας ως DSL για τον προγραμματισμό δυναμικών συστημάτων. Στην παρούσα εργασία αποφασίστηκε να μη γίνει η προσθήκη αυτή, παρά το γεγονός ότι οι αντίστοιχες τροποποιήσεις στη σύνταξη και τη σημασιολογία της GLU$^\sharp$ είναι σχετικά απλές. Η απόφαση αυτή βασίστηκε σε τρεις λόγους:

- Η ύπαρξη συναρτήσεων στην GLU$^\sharp$ δεν είναι απαραίτητη για τον προγραμματισμό εφαρμογών. Με χρήση των συναρτήσεων της C++ είναι δυνατό να ορισθούν συναρτήσεις των οποίων οι παράμετροι και τα αποτελέσματα να έχουν νοηματικό περιεχόμενο. Αυτό φαίνεται στα παραδείγματα της ενότητας 4. Πρέπει όμως να σημειωθεί ότι η κλήση συναρτήσεων στη C++ δεν είναι οκνηρή, κάτι που είναι γενικά αντίθετο με τη σημασιολογία της GLU$^\sharp$. Για να διατηρηθεί αυτή η σημασιολογία λαμβάνονται ιδιαίτερα μέτρα που περιγράφονται στην ενότητα 3.5.

- Η C++ δεν υποστηρίζει συναρτήσεις ως αντικείμενα πρώτης τάξης (first-class objects). Αυτό καθιστά ιδιαίτερα πολύπλοκη την αποδοτική υλοποίηση συναρτήσεων στην GLU$^\sharp$ που να ακολουθούν πιστά το μοντέλο συναρτησιακού προγραμματισμού.

- Προηγούμενα ερευνητικά αποτελέσματα [Rond97, Rond99] οδηγούν στο συμπέρασμα ότι με την παράλειψη των συναρτήσεων μια νοηματική γλώσσα δε στερείται εκφραστικότητας σε σημαντικό βαθμό, καθώς είναι δυνατός (υπό προϋποθέσεις) ο μετασχηματισμός ενός νοηματικού προγράμματος με συναρτήσεις n-τάξης σε ένα ισοδύναμο νοηματικό πρόγραμμα με συναρτήσεις $(n-1)$-τάξης, για $n > 0$.

Στην προσπάθεια ενσωμάτωσης της GLU$^\sharp$ στη C++ δόθηκε έμφαση σε δυο παράγοντες καθοριστικής σημασίας. Κατ' αρχήν η ενσωμάτωση έπρεπε να γίνει έτσι ώστε η χρήση των νοηματικών χαρακτηριστικών της GLU$^\sharp$ μέσα στη C++ να είναι φυσική. Με αυτό εννοούμε ότι ο ενσωματωμένος κώδικας στο πρόγραμμα C++ δε θα έπρεπε να διαφέρει σημαντικά από τον κώδικα που θα έγραφε κανείς στο αντίστοιχο υποσύνολο της Lucid ή της GLU. Επιπλέον, η υλοποίηση θα έπρεπε να είναι αποδοτική, δηλαδή οι χρόνοι εκτέλεσης των προγραμμάτων C++ με ενσωματωμένο κώδικα GLU$^\sharp$ δε θα έπρεπε να είναι σημαντικά μεγαλύτεροι από αυτούς των αντιστοίχων προγραμμάτων σε Lucid ή GLU, σε σύγκριση με διαθέσιμες υλοποιήσεις για αυτές τις γλώσσες.

2 Περιγραφή της GLU$^\sharp$

Η γλώσσα GLU$^\sharp$ είναι ένα μικρό υποσύνολο της GLU. Υποστηρίζει όλους τους νοηματικούς τελεστές της GLU και διατηρεί τον πολυδιάστατο χαρακτήρα της γλώσσας. Από την άλλη πλευρά όμως, διαθέτει ελάχιστα μη νοηματικά χαρακτηριστικά: μια οκνηρή γλώσσα εκφράσεων με δυο βασικούς τύπους δεδομένων (πραγματικούς αριθμούς και λογικές τιμές) και μια υποτυπώδη γλώσσα αναδρομικών ορισμών.

Ένα *πρόγραμμα* στη γλώσσα GLU$^\sharp$ είναι μια (πιθανώς κενή) ακολουθία *ορισμών*, ακολουθούμενη από μια *έκφραση*. Κάθε ορισμός ορίζει μια *μεταβλητή* και έχει τη μορφή:

```
type x = e;
```

όπου `type` είναι ο *τύπος* της μεταβλητής που ορίζεται, `x` το όνομά της και `e` μια έκφραση. Ο τύπος της έκφρασης `e` πρέπει να είναι ίδιος με τον τύπο της μεταβλητής `x`. Τα ονόματα των μεταβλητών αποτελούνται από γράμματα και ψηφία και αρχίζουν υποχρεωτικά με γράμμα. Δεν μπορούν να ταυτίζονται με τις παρακάτω λέξεις κλειδιά της γλώσσας:

```
true     false    not    and    or     if      then    else
first    next     fby    asa    wvr    real    bool
```

Οι ορισμοί μπορούν να είναι αναδρομικοί και η αναδρομή αυτή μπορεί να είναι άμεση ή έμμεση. Η έκφραση `e` μπορεί να περιέχει ονόματα μεταβλητών, ακόμα και της `x` ή άλλων μεταβλητών που δεν έχουν ακόμα ορισθεί. Οι ορισμοί γίνονται με οκνηρό τρόπο, δηλαδή η τιμή της έκφρασης `e` δεν αποτιμάται παρά μόνο όταν αργότερα στο πρόγραμμα χρειαστεί να υπολογιστεί η τιμή της μεταβλητής `x`. Το ίδιο ισχύει και γενικά για όλες τις τιμές των εκφράσεων στην GLU♯.

Οι τύποι δεδομένων που υποστηρίζει η GLU♯ είναι δυο: `real` και `bool`. Ο πρώτος αντιστοιχεί στους πραγματικούς αριθμούς, με όποιους περιορισμούς αναπαράστασης επιβάλλει η αρχιτεκτονική του υπολογιστή. Οι αριθμητικές σταθερές ανήκουν στον τύπο `real` και έχουν τη μορφή αριθμών στο δεκαδικό σύστημα αρίθμησης, με προαιρετική υποδιαστολή και δεκαδικό μέρος καθώς και με προαιρετικό δεκαδικό εκθέτη, με ή χωρίς πρόσημο. Όλα τα παρακάτω παραδείγματα περιγράφουν αριθμητικές σταθερές που αντιστοιχούν στον αριθμό 42:

```
42    42.0    42.00    4.2e1    4.2e+1    0.042e+3    42000e-3
```

Ο τύπος `bool` αντιστοιχεί σε λογικές τιμές αλήθειας και αποτελείται από δύο μόνο τιμές, που περιγράφονται από τις σταθερές `true` και `false`.

Όπως σε κάθε νοηματική γλώσσα προγραμματισμού, η τιμή μιας μεταβλητής στην GLU♯ είναι δυναμική και μεταβάλλεται ως προς ένα πεπερασμένο αριθμό *διαστάσεων*. Οι διαστάσεις περιγράφονται από ονόματα, όπως αυτά των μεταβλητών, ο ορισμός τους όμως δεν είναι ρητός αλλά γίνεται αυτόματα την πρώτη φορά που εμφανίζεται το όνομα μιας διάστασης. Για παράδειγμα, ο ορισμός:

```
real nat = 0 fby.t nat + 1;
```

εκτός του ότι ορίζει μια μεταβλητή `nat` πραγματικού τύπου, υποδηλώνει έμμεσα και την ύπαρξη της διάστασης `t` ως προς την οποία μεταβάλλεται η τιμή της μεταβλητής `nat`.

Οι εκφράσεις σχηματίζονται με την εφαρμογή *τελεστών* σε μεταβλητές, σταθερές ή άλλες εκφράσεις. Οι τελεστές διακρίνονται αφενός σε νοηματικούς και μη νοηματικούς τελεστές, αφετέρου ανάλογα με το πλήθος των τελουμένων. Οι μη νοηματικοί τελεστές αντιστοιχούν στις συνήθεις αριθμητικές και λογικές πράξεις με τελούμενα τύπου `real` και `bool`. Συνοψίζονται στον πίνακα 1, όπου επίσης φαίνεται ο αριθμός των τελουμένων κάθε τελεστή, η θέση του σχετικά με τα τελούμενα (προθεματική–prefix ή ενθεματική–infix) και η προσεταιριστικότητα του. Η προτεραιότητα των τελεστών ορίζεται στον πίνακα σε φθίνουσα σειρά από πάνω προς τα κάτω. Αυτό σημαίνει ότι ο τελεστής προσήμου - έχει τη μεγαλύτερη προτεραιότητα ενώ ο τελεστής `or` τη μικρότερη. Η χρήση παρενθέσεων επιτρέπει το σχηματισμό εκφράσεων ξεπερνώντας τους περιορισμούς προτεραιότητας και προσεταιριστικότητας των τελεστών.

Οι τελεστές των πρώτων τριων γραμμών του πίνακα 1 αντιστοιχούν στις συνήθεις πράξεις μεταξύ πραγματικών αριθμών. Τα τελούμενα πρέπει να είναι τύπου `real` και το ίδιο είναι και το

Πίνακας 1: Μη νοηματικοί τελεστές.

Τελεστής	Αριθμός τελουμένων	Θέση	Προσεταιριστικότητα
-	1	προθεματική	
* /	2	ενθεματική	αριστερή
+ -	2	ενθεματική	αριστερή
== != > < >= <=	2	ενθεματική	καμία
not	1	προθεματική	
and	2	ενθεματική	αριστερή
or	2	ενθεματική	αριστερή
if ... then ... else ...	3	ειδική	

Πίνακας 2: Νοηματικοί τελεστές.

Τελεστής	Αριθμός τελουμένων	Θέση	Προσεταιριστικότητα
#.d	0		
@.d	2	ενθεματική	καμία
first.d next.d	1	προθεματική	
fby.d asa.d wvr.d	2	ενθεματική	καμία

αποτέλεσμα. Οι τελεστές της τέταρτης γραμμής του πίνακα είναι οι σχεσιακοί τελεστές. Τα τελούμενα πρέπει να ανήκουν στον ίδιο τύπο (`real` ή `bool`) και το αποτέλεσμα είναι τύπου `bool`. Για την ανισότητα μεταξύ λογικών τιμών ορίζεται αυθαίρετα ότι `true > false`. Οι τελεστές των τριων επόμενων γραμμών του πίνακα είναι οι συνήθεις λογικοί τελεστές. Τα τελούμενα πρέπει να είναι τύπου `bool` και το ίδιο είναι και το αποτέλεσμα. Τέλος, ο τελεστής `if-then-else` είναι ο μοναδικός τελεστής με τρία τελούμενα και ειδικό τρόπο σύνταξης. Το πρώτο τελούμενο, η συνθήκη, πρέπει να είναι τύπου `bool`. Τα επόμενα δυο τελούμενα πρέπει να ανήκουν στον ίδιο τύπο και το αποτέλεσμα είναι επίσης του ίδιου τύπου.

Ο πίνακας 2 περιέχει τους νοηματικούς τελεστές της γλώσσας GLU♯. Καθένας από αυτούς συνοδεύεται από το όνομα μιας διαστασης (στον πίνακα έχει χρησιμοποιηθεί το γράμμα `d`). Η λειτουργία των νοηματικών τελεστών περιγράφεται στη σχετική βιβλιογραφία για τις γλώσσες Lucid [Ashc77, Wadg85] και GLU [Ashc91, Ashc95]. Η ακριβής σύνταξη και σημασιολογία τους ορίζεται με τυπικό τρόπο στις επόμενες παραγράφους.

2.1 Σύνταξη της GLU♯

Η σύνταξη της GLU♯ ορίζεται σε μορφή Extended BNF στο σχήμα 1. Η γραμματική που δίνεται σε αυτό περιέχει πληροφορίες για την προτεραιότητα και την προσεταιριστικότητα των τελεστών και μπορεί να χρησιμοποιηθεί άμεσα για την κατασκευή συντακτικών αναλυτών για τη γλώσσα. Για να είναι απλούστερη η περιγραφή της σημασιολογίας της GLU♯ στις επόμενες παραγράφους, θα χρησιμοποιηθεί η γραμματική του σχήματος 2 η οποία περιγράφει την *αφηρημένη σύνταξη* (ab-

$$
\begin{array}{lll}
\mathit{prog} & ::= & (\mathit{defn})^*\ \mathit{expr} \\
\mathit{defn} & ::= & \mathit{type}\ \mathit{var}\ \text{“=”}\ \mathit{expr}\ \text{“;”} \\
\mathit{type} & ::= & \texttt{real} \mid \texttt{bool} \\
\mathit{expr}_0 & ::= & \mathit{var} \mid \mathit{num} \mid \text{“}\texttt{true}\text{”} \mid \text{“}\texttt{false}\text{”} \mid \text{“}\texttt{(}\text{”}\ \mathit{expr}\ \text{“}\texttt{)}\text{”} \\
\mathit{expr}_1 & ::= & \mathit{expr}_0 \mid \text{“}\texttt{-}\text{”}\ \mathit{expr}_1 \\
\mathit{expr}_2 & ::= & \mathit{expr}_1 \mid \mathit{expr}_1\ \text{“}\texttt{*}\text{”}\ \mathit{expr}_2 \mid \mathit{expr}_1\ \text{“}\texttt{/}\text{”}\ \mathit{expr}_2 \\
\mathit{expr}_3 & ::= & \mathit{expr}_2 \mid \mathit{expr}_2\ \text{“}\texttt{+}\text{”}\ \mathit{expr}_3 \mid \mathit{expr}_2\ \text{“}\texttt{-}\text{”}\ \mathit{expr}_3 \\
\mathit{expr}_4 & ::= & \mathit{expr}_3 \mid \mathit{expr}_3\ \text{“}\texttt{==}\text{”}\ \mathit{expr}_3 \mid \mathit{expr}_3\ \text{“}\texttt{!=}\text{”}\ \mathit{expr}_3 \\
 & \mid & \mathit{expr}_3\ \text{“}\texttt{>}\text{”}\ \mathit{expr}_3 \mid \mathit{expr}_3\ \text{“}\texttt{<}\text{”}\ \mathit{expr}_3 \mid \mathit{expr}_3\ \text{“}\texttt{>=}\text{”}\ \mathit{expr}_3 \mid \mathit{expr}_3\ \text{“}\texttt{<=}\text{”}\ \mathit{expr}_3 \\
\mathit{expr}_5 & ::= & \mathit{expr}_4 \mid \text{“}\texttt{not}\text{”}\ \mathit{expr}_5 \\
\mathit{expr}_6 & ::= & \mathit{expr}_5 \mid \mathit{expr}_5\ \text{“}\texttt{and}\text{”}\ \mathit{expr}_6 \\
\mathit{expr}_7 & ::= & \mathit{expr}_6 \mid \mathit{expr}_6\ \text{“}\texttt{or}\text{”}\ \mathit{expr}_7 \\
\mathit{expr}_8 & ::= & \mathit{expr}_7 \mid \text{“}\texttt{if}\text{”}\ \mathit{expr}\ \text{“}\texttt{then}\text{”}\ \mathit{expr}\ \text{“}\texttt{else}\text{”}\ \mathit{expr}_8 \\
\mathit{expr}_9 & ::= & \mathit{expr}_8 \mid \text{“}\texttt{\#}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim} \\
\mathit{expr}_{10} & ::= & \mathit{expr}_9 \mid \mathit{expr}_9\ \text{“}\texttt{@}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_9 \\
\mathit{expr}_{11} & ::= & \mathit{expr}_{10} \mid \text{“}\texttt{first}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_{11} \mid \text{“}\texttt{next}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_{11} \\
\mathit{expr}_{12} & ::= & \mathit{expr}_{11} \mid \mathit{expr}_{11}\ \text{“}\texttt{fby}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_{11} \\
 & \mid & \mathit{expr}_{11}\ \text{“}\texttt{asa}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_{11} \mid \mathit{expr}_{11}\ \text{“}\texttt{wvr}\text{”}\ \text{“}\texttt{.}\text{”}\ \mathit{dim}\ \mathit{expr}_{11} \\
\mathit{expr} & ::= & \mathit{expr}_{12}
\end{array}
$$

Σχήμα 1: Συγκεκριμένη σύνταξη (concrete syntax) της GLU♮.

$$
\begin{array}{lll}
\mathit{prog} & ::= & (\mathit{defn})^*\ \mathit{expr} \\
\mathit{defn} & ::= & \mathit{type}\ \mathit{var} = \mathit{expr} \\
\mathit{type} & ::= & \texttt{real} \mid \texttt{bool} \\
\mathit{expr} & ::= & \mathit{var} \mid \mathit{num} \mid \texttt{true} \mid \texttt{false} \mid \mathit{unop}\ \mathit{expr} \mid \mathit{expr}\ \mathit{binop}\ \mathit{expr} \\
 & \mid & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \mid \texttt{value.}\mathit{dim} \mid \texttt{first.}\mathit{dim}\ \mathit{expr} \\
 & \mid & \texttt{next.}\mathit{dim}\ \mathit{expr} \mid \mathit{expr}\ \texttt{fby.}\mathit{dim}\ \mathit{expr} \mid \mathit{expr}\ \texttt{at.}\mathit{dim}\ \mathit{expr} \\
 & \mid & \mathit{expr}\ \texttt{asa.}\mathit{dim}\ \mathit{expr} \mid \mathit{expr}\ \texttt{wvr.}\mathit{dim}\ \mathit{expr} \\
\mathit{unop} & ::= & \texttt{neg} \mid \texttt{not} \\
\mathit{binop} & ::= & \texttt{times} \mid \texttt{div} \mid \texttt{mod} \mid \texttt{plus} \mid \texttt{minus} \mid \texttt{eq} \mid \texttt{ne} \mid \texttt{lt} \mid \texttt{gt} \mid \texttt{le} \mid \texttt{ge} \mid \texttt{and} \mid \texttt{or}
\end{array}
$$

Σχήμα 2: Αφηρημένη σύνταξη (abstract syntax) της GLU♮.

stract syntax) της GLU$^\natural$. Ο μετασχηματισμός από την συγκεκριμένη στην αφηρημένη σύνταξη είναι αρκετά απλός και παραλείπεται.

2.2 Στατική σημασιολογία της GLU$^\natural$

Σκοπός αυτής της παραγράφου είναι ο ορισμός του *περιβάλλοντος τύπων* (type environment) που αντιστοιχεί στους ορισμούς ενός προγράμματος. Το περιβάλλον αυτό απεικονίζει έναν τύπο της GLU$^\natural$ για κάθε ορισμένη μεταβλητή του προγράμματος και χρησιμοποιείται για τον έλεγχο τύπων στην επόμενη παράγραφο.

Αρχίζουμε με κάποιους ορισμούς πεδίων. Τα στοιχεία των πεδίων ***Var*** και ***Dim*** είναι τα έγκυρα ονόματα μεταβλητών και διαστάσεων της GLU$^\natural$, αντίστοιχα. Το πεδίο ***Type*** περιέχει μόνο δύο στοιχεία, τους τύπους δεδομένων της GLU$^\natural$. Τέλος, το πεδίο ***Ent*** είναι το πεδίο των περιβαλλόντων τύπων.[1]

$$
\begin{array}{lcll}
v & : & \textbf{\textit{Var}} & \text{(πεδίο μεταβλητών)} \\
d & : & \textbf{\textit{Dim}} & \text{(πεδίο διαστάσεων)} \\
\tau & : & \textbf{\textit{Type}} & = \{\, \textit{real}, \textit{bool} \,\} \\
\Gamma & : & \textbf{\textit{Ent}} & = \mathcal{P}(\textbf{\textit{Var}} \times \textbf{\textit{Type}})
\end{array}
$$

Τα στοιχεία του ***Ent*** είναι σύνολα που περιέχουν ζεύγη μεταβλητών και τύπων της GLU$^\natural$.

Με Γ_0 συμβολίζεται το κενό περιβάλλον τύπων.

$$
\begin{array}{lcl}
\Gamma_0 & : & \textbf{\textit{Ent}} \\
\Gamma_0 & = & \emptyset
\end{array}
$$

Για κάθε μεταβλητή var που εμφανίζεται ως συντακτικό σύμβολο σε ένα πρόγραμμα GLU$^\natural$, με $\{\!|var|\!\}$ συμβολίζουμε το στοιχείο του πεδίου ***Var*** που αντιστοιχεί σε αυτή τη μεταβλητή.

$$\{\!|var|\!\} \;:\; \textbf{\textit{Var}}$$

Το ίδιο κάνουμε για τα συντακτικά σύμβολα των διαστάσεων και τα στοιχεία του πεδίου ***Dim***.

$$\{\!|dim|\!\} \;:\; \textbf{\textit{Dim}}$$

Ομοίως, σε κάθε συντακτικό σύμβολο τύπου της GLU$^\natural$ αντιστοιχούμε ένα στοιχείο του πεδίου ***Type***.

$$
\begin{array}{lcl}
\{\!|type|\!\} & : & \textbf{\textit{Type}} \\
\{\!|\texttt{real}|\!\} & = & \textit{real} \\
\{\!|\texttt{bool}|\!\} & = & \textit{bool}
\end{array}
$$

Ορίζουμε τώρα τη στατική σημασιολογική συνάρτηση, που απεικονίζει ακολουθίες ορισμών της GLU$^\natural$ σε περιβάλλοντα τύπων. Η κενή ακολουθία απεικονίζεται στο κενό περιβάλλον, ενώ η παράθεση ορισμών οδηγεί στην ένωση των αντίστοιχων περιβαλλόντων.

[1] Με $\mathcal{P}(A)$ συμβολίζεται το *δυναμοσύνολο* (powerset) του συνόλου A.

$$\{\!|(defn)^*|\!\} \ : \ \textbf{\textit{Ent}}$$
$$\{\!|\epsilon|\!\} \ = \ \Gamma_0$$
$$\{\!|defn\ (defn)^*|\!\} \ = \ \{\!|defn|\!\} \cup \{\!|(defn)^*|\!\}$$

Τέλος, κάθε ορισμός της GLU$^\natural$ απεικονίζεται σε ένα περιβάλλον τύπων που περιέχει μόνο ένα στοιχείο.

$$\{\!|defn|\!\} \ : \ \textbf{\textit{Ent}}$$
$$\{\!|type\ var = expr|\!\} \ = \ \{\ \langle \{\!|var|\!\}, \{\!|type|\!\} \rangle\ \}$$

2.3 Σημασιολογία τύπων της GLU$^\natural$

Η σημασιολογία τύπων της GLU$^\natural$ αποσκοπεί στην εύρεση σφαλμάτων τύπων σε προγράμματα. Περιγράφεται μέσω προτάσεων της μορφής $\Gamma \vdash P : \theta$, όπου Γ είναι ένα περιβάλλον τύπων, P ένα τμήμα του προγράμματος βάσει της αφηρημένης σύνταξης και θ ένας *τύπος πρότασης* (phrase type). Ο τύπος θ μπορεί να είναι ένας από τους $\textbf{\textit{prog}}[\tau]$, $\textbf{\textit{defn}}$, $\textbf{\textit{exp}}[\tau]$. Ο πρώτος και ο τρίτος παριστάνουν αντίστοιχα προγράμματα και εκφράσεις που υπολογίζουν αποτελέσματα τύπου τ, ενώ ο δεύτερος παριστάνει ορισμούς ή ακολουθίες ορισμών.

Ένα πρόγραμμα είναι έγκυρο όταν η ακολουθία ορισμών και η έκφραση που πρέπει να υπολογιστεί είναι έγκυρες στο περιβάλλον τύπων που αντιστοιχεί στην ακολουθία ορισμών.

$$\frac{\Gamma = \{\!|(defn)^*|\!\} \qquad \Gamma \vdash (defn)^* : \textbf{\textit{defn}} \qquad \Gamma \vdash expr : \textbf{\textit{exp}}[\tau]}{\vdash (defn)^*\ expr : \textbf{\textit{prog}}[\tau]}$$

Οι παρακάτω κανόνες περιγράφουν την εγκυρότητα ορισμών και ακολουθιών ορισμών. Σε κάθε ορισμό, ο τύπος της μεταβλητής πρέπει να συμφωνεί με τον τύπο της έκφρασης.

$$\frac{}{\vdash \epsilon : \textbf{\textit{defn}}} \qquad \frac{\Gamma \vdash defn : \textbf{\textit{defn}} \qquad \Gamma \vdash (defn)^* : \textbf{\textit{defn}}}{\vdash defn\ (defn)^* : \textbf{\textit{defn}}}$$

$$\frac{\tau = \{\!|type|\!\} \qquad \Gamma \vdash expr : \textbf{\textit{exp}}[\tau]}{\Gamma \vdash type\ var = expr : \textbf{\textit{defn}}}$$

Η εγκυρότητα των εκφράσεων περιγράφεται με τη σειρά κανόνων που ακολουθούν. Οι περιπτώσεις των μεταβλητών, αριθμητικών και λογικών σταθερών είναι αρκετά απλές. Ο τύπος κάθε μεταβλητής αναζητάται στο περιβάλλον τύπων Γ.

$$\frac{\langle \{\!|var|\!\}, \tau \rangle \in \Gamma}{\Gamma \vdash var : \textbf{\textit{exp}}[\tau]} \qquad \frac{}{\Gamma \vdash num : \textbf{\textit{exp}}[\textbf{\textit{real}}]}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \textbf{\textit{exp}}[\textbf{\textit{bool}}]} \qquad \frac{}{\Gamma \vdash \texttt{false} : \textbf{\textit{exp}}[\textbf{\textit{bool}}]}$$

Οι επόμενοι κανόνες περιγράφουν την εγκυρότητα εκφράσεων που προκύπτουν από την εφαρμογή των μη νοηματικών τελεστών. Σε κάθε κανόνα περιγράφεται ο τύπος των τελουμένων και αυτός του αποτελέσματος.

$$\frac{\Gamma \vdash expr : \mathbf{exp}[\mathbf{real}]}{\Gamma \vdash \texttt{neg}\ expr : \mathbf{exp}[\mathbf{real}]} \qquad \frac{\Gamma \vdash expr : \mathbf{exp}[\mathbf{bool}]}{\Gamma \vdash \texttt{not}\ expr : \mathbf{exp}[\mathbf{bool}]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\mathbf{real}] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\mathbf{real}] \quad binop \in \{\texttt{plus}, \texttt{minus}, \texttt{times}, \texttt{div}\}}{\Gamma \vdash expr_1\ binop\ expr_2 : \mathbf{exp}[\mathbf{real}]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\tau] \quad binop \in \{\texttt{eq}, \texttt{ne}, \texttt{gt}, \texttt{lt}, \texttt{ge}, \texttt{le}\}}{\Gamma \vdash expr_1\ binop\ expr_2 : \mathbf{exp}[\mathbf{bool}]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\mathbf{bool}] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\mathbf{bool}] \quad binop \in \{\texttt{and}, \texttt{or}\}}{\Gamma \vdash expr_1\ binop\ expr_2 : \mathbf{exp}[\mathbf{bool}]}$$

$$\frac{\Gamma \vdash expr : \mathbf{exp}[\mathbf{bool}] \quad \Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\tau]}{\Gamma \vdash \texttt{if}\ expr\ \texttt{then}\ expr_1\ \texttt{else}\ expr_2 : \mathbf{exp}[\tau]}$$

Τέλος, οι επόμενοι κανόνες περιγράφουν την εγκυρότητα εκφράσεων που προκύπτουν από την εφαρμογή των νοηματικών τελεστών. Σε κάθε κανόνα περιγράφεται πάλι ο τύπος των τελουμένων και αυτός του αποτελέσματος.

$$\frac{\Gamma \vdash expr : \mathbf{exp}[\tau]}{\Gamma \vdash \texttt{first.}dim\ expr : \mathbf{exp}[\tau]} \qquad \frac{\Gamma \vdash expr : \mathbf{exp}[\tau]}{\Gamma \vdash \texttt{next.}dim\ expr : \mathbf{exp}[\tau]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\tau]}{\Gamma \vdash expr_1\ \texttt{fby.}dim\ expr_2 : \mathbf{exp}[\tau]} \qquad \frac{}{\Gamma \vdash \texttt{value.}dim : \mathbf{exp}[\mathbf{real}]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\mathbf{real}]}{\Gamma \vdash expr_1\ \texttt{at.}dim\ expr_2 : \mathbf{exp}[\tau]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\mathbf{bool}]}{\Gamma \vdash expr_1\ \texttt{asa.}dim\ expr_2 : \mathbf{exp}[\tau]}$$

$$\frac{\Gamma \vdash expr_1 : \mathbf{exp}[\tau] \quad \Gamma \vdash expr_2 : \mathbf{exp}[\mathbf{bool}]}{\Gamma \vdash expr_1\ \texttt{wvr.}dim\ expr_2 : \mathbf{exp}[\tau]}$$

2.4 Δυναμική σημασιολογία της GLU$^\sharp$

Στην παράγραφο αυτή ορίζεται ο τρόπος εκτέλεσης των προγραμμάτων της GLU$^\sharp$, για τα οποία έχει προηγηθεί ο έλεγχος τύπων. Έστω $\boldsymbol{N}$, $\boldsymbol{R}$ και $\boldsymbol{T}$ τα πεδία που αντιστοιχούν στους φυσικούς αριθμούς, τους πραγματικούς αριθμούς και τις λογικές τιμές αλήθειας ***true*** και ***false*** αντίστοιχα. Κάθε συντακτικό σύμβολο αριθμητικής σταθεράς της GLU$^\sharp$ απεικονίζεται σε ένα στοιχείο του πεδίου $\boldsymbol{R}$.

$$\{\!|num|\!\} \ : \ \boldsymbol{R}$$

Θεωρούμε ότι $\boldsymbol{N} \subset \boldsymbol{R}$. Η συνάρτηση $\mathit{round} : \boldsymbol{R} \to \boldsymbol{N}$ επιστρέφει για κάθε πραγματικό αριθμό τον πλησιέστερο φυσικό αριθμό. Αν υπάρχουν δυο πλησιέστεροι, επιστρέφει τον μεγαλύτερο.

Ορίζουμε στη συνέχεια τα ακόλουθα σημασιολογικά πεδία:[2]

[2]Για τις ανάγκες αυτής της εργασίας, τα πεδία μπορούν να θεωρηθούν ως *πλήρεις μερικές διατάξεις* (complete partial orders — cpo's). Κάθε πεδίο D διαθέτει ένα ελάχιστο στοιχείο, που συμβολίζεται με $\bot_D$ και αντιστοιχεί σημασιολογικά

$$
\begin{array}{lcll}
v & : & \boldsymbol{V} & = & \boldsymbol{R} \oplus \boldsymbol{T} \\
w & : & \boldsymbol{W} & = & \boldsymbol{Dim} \to \boldsymbol{N} \\
 & & \boldsymbol{D} & = & \boldsymbol{W} \to \boldsymbol{V} \\
\rho & : & \boldsymbol{Env} & = & \boldsymbol{Var} \to \boldsymbol{D}
\end{array}
$$

Το πεδίο $\boldsymbol{V}$ περιέχει όλες τις δυνατές τιμές αποτελέσματος μιας έκφρασης ή ενός προγράμματος, που μπορούν να είναι πραγματικές ή λογικές. Τα στοιχεία του πεδίου $\boldsymbol{W}$ είναι οι *δυνατοί κόσμοι* (possible worlds). Κάθε δυνατός κόσμος w είναι μια συνάρτηση που επιστρέφει το τρέχον σημείο σε κάθε διάσταση. Το πεδίο $\boldsymbol{D}$ χρησιμοποιείται για την απόδοση ερμηνείας στις εκφράσεις της $\mathsf{GLU}^\natural$. Η τιμή μιας έκφρασης μεταβάλλεται ως προς κάποιες διαστάσεις και συνεπώς η ερμηνεία της έκφρασης είναι μια συνάρτηση από το πεδίο των δυνατών κόσμων στο πεδίο των τιμών αποτελέσματος. Τα στοιχεία του πεδίου $\boldsymbol{Env}$ είναι τα *περιβάλλοντα τιμών* (value environments), που απεικονίζουν κάθε ορισμένη μεταβλητή του προγράμματος στην ερμηνεία της αντίστοιχης έκφρασης.

Ο κόσμος w_0 είναι ένα ιδιαίτερο στοιχείο του πεδίου $\boldsymbol{W}$. Στον κόσμο αυτό, το τρέχον σημείο όλων των διαστάσεων είναι το $\bot_{\boldsymbol{N}}$. Ο κόσμος αυτός θεωρείται ο αρχικός κόσμος στον οποίο γίνεται η αποτίμηση ενός προγράμματος.

$$
\begin{array}{lcl}
w_0 & : & \boldsymbol{W} \\
w_0 & = & \lambda d : \boldsymbol{Dim}.\ \bot_{\boldsymbol{N}}
\end{array}
$$

Το περιβάλλον ρ_0 είναι επίσης ένα ιδιαίτερο στοιχείο του πεδίου $\boldsymbol{Env}$ και αντιστοιχεί όλες τις μεταβλητές στο $\bot_{\boldsymbol{D}}$. Αυτό σημαίνει ότι όλες οι μεταβλητές δεν έχουν ακόμα ορισθεί.

$$
\begin{array}{lcl}
\rho_0 & : & \boldsymbol{Env} \\
\rho_0 & = & \lambda v : \boldsymbol{Var}.\ \bot_{\boldsymbol{D}}
\end{array}
$$

Αν $f : A \to B$ είναι μια συνάρτηση, χρησιμοποιούμε το συμβολισμό $f\{a \mapsto b\}$ για τη συνάρτηση $f' : A \to B$ που διαφέρει από την f μόνο στο ότι $f'(a) = b$.

$$
f\{a \mapsto b\} \ = \ \lambda x : A.\ \boldsymbol{if}\ x = a\ \boldsymbol{then}\ b\ \boldsymbol{else}\ f\ x
$$

Η ερμηνεία του προγράμματος είναι η τελική τιμή της έκφρασης που υπολογίζεται. Το περιβάλλον τιμών στο οποίο θα γίνει ο υπολογισμός προκύπτει από την ακολουθία ορισμών του προγράμματος. Ο τελεστής ***fix*** επιτρέπει την ύπαρξη αναδρομής στους ορισμούς.

$$
\begin{array}{l}
[\![\, prog \,]\!] \ : \ \boldsymbol{V} \\
[\![\, (defn)^*\ expr \,]\!] \ = \ [\![\, expr \,]\!]\ (\boldsymbol{fix}\ (\lambda \rho : \boldsymbol{Env}.\ [\![\, (defn)^* \,]\!]\ \rho\ \rho_0))\ w_0
\end{array}
$$

Η ερμηνεία ενός ορισμού ή μιας ακολουθίας ορισμών είναι μια συνάρτηση που παίρνει κατ' αρχήν ως παράμετρο το περιβάλλον τιμών ρ, μεσα στο οποίο θα γίνονται οι αποτιμήσεις των εκφράσεων. Επιστρέφει μια συνάρτηση η οποία ενημερώνει το τρέχον περιβάλλον τιμών ρ' προσθέτοντας τους ορισμούς των μεταβλητών.

σε μια αόριστη τιμή (χωρίς να γίνεται διάκριση ανάμεσα στο μη τερματισμό ενός υπολογισμού και στην εμφάνιση σφάλματος εκτέλεσης). Η θεώρηση των πεδίων ως cpo's επιτρέπει τη χρήση του τελεστή *ελάχιστου σταθερού σημείου* (least fixed point), που συμβολίζεται με ***fix***.

$$[\![(defn)^*]\!] \;:\; \mathbf{Env} \to \mathbf{Env} \to \mathbf{Env}$$

$$[\![\epsilon]\!] \;=\; \lambda\rho : \mathbf{Env}.\ id$$
$$[\![defn\ (defn)^*]\!] \;=\; \lambda\rho : \mathbf{Env}.\ [\![(defn)^*]\!]\,\rho \;\circ\; [\![defn]\!]\,\rho$$

$$[\![defn]\!] \;:\; \mathbf{Env} \to \mathbf{Env} \to \mathbf{Env}$$

$$[\![type\ var = expr]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda\rho' : \mathbf{Env}.\ \mathbf{let}\ v = \{\!|var|\!\}\ \mathbf{in}\ \rho'\{v \mapsto [\![expr]\!]\,\rho\}$$

Η ερμηνεία μιας έκφρασης είναι μια συνάρτηση που δέχεται ως παράμετρο ένα περιβάλλον τιμών και επιστρέφει ένα στοιχείο του πεδίου $\boldsymbol{D}$.

$$[\![expr]\!] \;:\; \mathbf{Env} \to \boldsymbol{D}$$

Ο ορισμός αυτής της συνάρτησης είναι αρκετά απλός για τις εκφράσεις που δεν περιέχουν νοηματικούς τελεστές. Οι τιμές των μεταβλητών διαβάζονται από το περιβάλλον. Ο υπολογισμός των υποεκφράσεων μιας έκφρασης γίνεται στον ίδιο δυνατό κόσμο w στον οποίο πρέπει να γίνει ο υπολογισμός ολόκληρης της έκφρασης.

$$[\![var]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \mathbf{let}\ v = \{\!|var|\!\}\ \mathbf{in}\ \rho\, v$$
$$[\![num]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ \{\!|num|\!\}$$
$$[\![\texttt{true}]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ true$$
$$[\![\texttt{false}]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ false$$
$$[\![unop\ expr]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ [\![unop]\!]\,([\![expr]\!]\,\rho\, w)$$
$$[\![expr_1\ binop\ expr_2]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ [\![binop]\!]\,\langle [\![expr_1]\!]\,\rho\, w, [\![expr_2]\!]\,\rho\, w\rangle$$
$$[\![\texttt{if}\ expr\ \texttt{then}\ expr_1\ \texttt{else}\ expr_2]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.$$
$$\quad \mathbf{if}\ [\![expr]\!]\,\rho\, w\ \mathbf{then}\ [\![expr_1]\!]\,\rho\, w\ \mathbf{else}\ [\![expr_2]\!]\,\rho\, w$$

Κάθε μη νοηματικός τελεστής με ένα ή δυο τελούμενα αντιστοιχεί σε μια σημασιολογική συνάρτηση της μορφής:

$$[\![unop]\!] \;:\; \boldsymbol{V} \to \boldsymbol{V}$$
$$[\![binop]\!] \;:\; \boldsymbol{V} \times \boldsymbol{V} \to \boldsymbol{V}$$

Ο ορισμός αυτών των συναρτήσεων είναι εύκολος και παραλείπεται. Ο έλεγχος τύπων εξασφαλίζει ότι οι παράμετροι θα έχουν τον κατάλληλο τύπο.

Ο τελεστής $\texttt{value.}dim$ επιστρέφει το τρέχον σημείο της διάστασης dim, το οποίο διαβάζει από τον τρέχοντα δυνατό κόσμο w.

$$[\![\texttt{value.}dim]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ \mathbf{let}\ d = \{\!|dim|\!\}\ \mathbf{in}\ w\, d$$

Η τιμή της έκφρασης $\texttt{first.}dim\ expr$ είναι απλά η τιμή της έκφρασης $expr$ υπολογισμένη στο δυνατό κόσμο όπου η διάσταση dim βρίσκεται στο σημείο 0.

$$[\![\texttt{first.}dim\ expr]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \boldsymbol{W}.\ \mathbf{let}\ d = \{\!|dim|\!\}\ \mathbf{in}\ [\![expr]\!]\,\rho\, w\{d \mapsto 0\}$$

Με παρόμοιο τρόπο, η τιμή της έκφρασης `next.`*dim expr* είναι η τιμή της έκφρασης *expr* υπολογισμένη στο δυνατό κόσμο όπου η διάσταση *dim* βρίσκεται στο επόμενο σημείο από αυτό που βρισκόταν στον τρέχοντα δυνατό κόσμο w.

$$[\![\texttt{next.}\mathit{dim}\ \mathit{expr}]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \mathbf{W}.\ \mathbf{let}\ d = \{\!|\mathit{dim}|\!\}\ \mathbf{in}\ [\![\mathit{expr}]\!]\,\rho\,w\{d \mapsto w\ d + 1\}$$

Επίσης, η τιμή της έκφρασης expr_1 `at.`*dim* expr_2 είναι η τιμή της έκφρασης expr_1 υπολογισμένη στο δυνατό κόσμο όπου η διάσταση *dim* βρίσκεται στο σημείο που δίνεται από την τιμή της έκφρασης expr_2. Ο υπολογισμός της τελευταίας γίνεται στον κόσμο w.

$$\begin{aligned}&[\![\mathit{expr}_1\ \texttt{at.}\mathit{dim}\ \mathit{expr}_2]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \mathbf{W}.\ \mathbf{let}\ d = \{\!|\mathit{dim}|\!\}\ \mathbf{in}\\ &\quad \mathbf{let}\ n = \mathbf{round}\ ([\![\mathit{expr}_2]\!]\,\rho\,w)\ \mathbf{in}\ [\![\mathit{expr}_1]\!]\,\rho\,w\{d \mapsto n\}\end{aligned}$$

Με χρήση του τελεστή `fby.`*dim* είναι δυνατή η κατασκευή αυθαίρετων εκφράσεων που μεταβάλλονται ως προς τη διάσταση *dim*. Η τιμή της έκφρασης expr_1 `fby.`*dim* expr_2 εξαρτάται από το σημείο της διάστασης *dim* στον τρέχοντα δυνατό κόσμο w. Αν αυτό είναι το 0, τότε το αποτέλεσμα είναι η τιμή της έκφρασης expr_1 στο ίδιο σημείο. Διαφορετικά, είναι η τιμή της έκφρασης expr_2 υπολογισμένη στο δυνατό κόσμο όπου η διάσταση *dim* βρίσκεται στο προηγούμενο σημείο από αυτό που βρισκόταν στον w.

$$\begin{aligned}&[\![\mathit{expr}_1\ \texttt{fby.}\mathit{dim}\ \mathit{expr}_2]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \mathbf{W}.\ \mathbf{let}\ d = \{\!|\mathit{dim}|\!\}\ \mathbf{in}\\ &\quad \mathbf{if}\ w\ d = 0\ \mathbf{then}\ [\![\mathit{expr}_1]\!]\,\rho\,w\ \mathbf{else}\ [\![\mathit{expr}_2]\!]\,\rho\,w\{d \mapsto w\ d - 1\}\end{aligned}$$

Ο τελεστής `asa.`*dim* διαβάζεται "μόλις στη διάσταση *dim*" (as soon as). Η τιμή της έκφρασης expr_1 `asa.`*dim* expr_2 είναι η τιμή της έκφρασης expr_1 υπολογισμένη στο πρώτο σημείο της διάστασης *dim* όπου η τιμή της έκφρασης expr_2 είναι αληθής. Αν η τιμή της expr_2 είναι ψευδής σε όλα τα σημεία της διάστασης *dim*, τότε το αποτέλεσμα είναι απροσδιόριστο. Η ερμηνεία της έκφρασης δίνεται μέσω του τελεστή ***fix***.

$$\begin{aligned}&[\![\mathit{expr}_1\ \texttt{asa.}\mathit{dim}\ \mathit{expr}_2]\!] \;=\; \lambda\rho : \mathbf{Env}.\ \lambda w : \mathbf{W}.\ \mathbf{let}\ d = \{\!|\mathit{dim}|\!\}\ \mathbf{in}\\ &\quad \mathbf{fix}\ (\lambda f : \mathbf{N} \to \mathbf{V}.\ \lambda n : \mathbf{N}.\\ &\qquad \mathbf{if}\ [\![\mathit{expr}_2]\!]\,\rho\,w\{d \mapsto n\}\ \mathbf{then}\\ &\qquad\quad [\![\mathit{expr}_1]\!]\,\rho\,w\{d \mapsto n\}\\ &\qquad \mathbf{else}\\ &\qquad\quad f\ (n+1)\\ &\quad)\ 0\end{aligned}$$

Ο πιο πολύπλοκος νοηματικός τελεστής είναι ο `wvr.`*dim*, που διαβάζεται "όποτε στη διάσταση *dim*" (whenever). Η τιμή της έκφρασης expr_1 `wvr.`*dim* expr_2 είναι η τιμή της έκφρασης expr_1 υπολογισμένη όμως μόνο σε σημεία της διάστασης *dim* όπου η τιμή της έκφρασης expr_2 είναι αληθής. Αν η τιμή της expr_2 είναι αληθής μόνο σε πεπερασμένο αριθμό σημείων της διάστασης *dim*, τότε το αποτέλεσμα στα υπόλοιπα σημεία είναι απροσδιόριστο. Η ερμηνεία της έκφρασης δίνεται και πάλι μέσω του τελεστή ***fix***.

$$
\begin{array}{l}
[\![expr_1 \texttt{ wvr . } dim\ expr_2]\!] \ = \ \lambda\rho : \textbf{\textit{Env}}.\ \lambda w : \textbf{\textit{W}}.\ \textbf{\textit{let}}\ d = \{\!|\, dim \,|\!\}\ \textbf{\textit{in}} \\
\quad \textbf{\textit{fix}}\ (\lambda f : \textbf{\textit{N}} \to \textbf{\textit{N}} \to \textbf{\textit{V}}.\ \lambda n : \textbf{\textit{N}}.\ \lambda m : \textbf{\textit{N}}. \\
\qquad \textbf{\textit{if}}\ [\![expr_2]\!]\ \rho\ w\{d \mapsto n\}\ \textbf{\textit{then}} \\
\qquad\quad \textbf{\textit{if}}\ w\ d = m\ \textbf{\textit{then}} \\
\qquad\qquad [\![expr_1]\!]\ \rho\ w\{d \mapsto n\} \\
\qquad\quad \textbf{\textit{else}} \\
\qquad\qquad f\ (n+1)\ (m+1) \\
\qquad \textbf{\textit{else}} \\
\qquad\quad f\ (n+1)\ m \\
\quad)\ 0\ 0
\end{array}
$$

2.5 Εξάρτηση από διαστάσεις

Οι παρακάτω συναρτήσεις υπολογίζουν το σύνολο των διαστάσεων από τις οποίες εξαρτώνται οι διάφορες συντακτικές κλάσεις της GLU$^\sharp$, δηλαδή το πρόγραμμα, η ακολουθία ορισμών, ο ορισμός και η έκφραση. Η πληροφορία αυτή δεν είναι χρήσιμη στον ορισμό της σημασιολογίας της GLU$^\sharp$, είναι όμως ιδιαίτερα χρήσιμη στην υλοποίηση της γλώσσας.

Τα στοιχεία του πεδίου ***End*** ονομάζονται *περιβάλλοντα εξαρτήσεων* και απεικονίζουν τις μεταβλητές ενός προγράμματος στα σύνολα των διαστάσεων από τις οποίες εξαρτώνται.

$$\delta \ : \ \textbf{\textit{End}} \ = \ \textbf{\textit{Var}} \to \mathcal{P}(\textbf{\textit{Dim}})$$

Με δ_0 συμβολίζεται το ιδιαίτερο στοιχείο του ***End*** που αντιστοιχεί το κενό σύνολο σε όλες τις μεταβλητές.

$$
\begin{array}{lcl}
\delta_0 & : & \textbf{\textit{End}} \\
\delta_0 & = & \lambda v : \textbf{\textit{Var}}.\ \emptyset
\end{array}
$$

Το σύνολο των διαστάσεων από τις οποίες εξαρτάται ένα πρόγραμμα υπολογίζεται εφαρμόζοντας πρώτα τον τελεστή ***fix*** για να βρούμε το περιβάλλον εξαρτήσεων της ακολουθίας ορισμών.[3] Στη συνέχεια, με αυτό το περιβάλλον εξαρτήσεων υπολογίζουμε το σύνολο εξαρτήσεων για την έκφραση που αποτελεί το σώμα του προγράμματος.

$$
\begin{array}{l}
\mathcal{D}[\![prog]\!] \ : \ \mathcal{P}(\textbf{\textit{Dim}}) \\
\mathcal{D}[\![(defn)^*\ expr]\!] \ = \ \mathcal{D}[\![expr]\!]\ (\textbf{\textit{fix}}\ (\lambda\delta : \textbf{\textit{End}}.\ \mathcal{D}[\![(defn)^*]\!]\ \delta\ \delta_0))
\end{array}
$$

Η συνάρτηση $\mathcal{D}[\![\ \cdot\ ]\!]$ για έναν ορισμό ή μια ακολουθία ορισμών επιστρέφει μια συνάρτηση που δέχεται ως παράμετρο το περιβάλλον εξαρτήσεων, βάσει του οποίου θα υπολογιστούν οι εξαρτήσεις των ορισμών. Επιστρέφει μια συνάρτηση η οποία ενημερώνει το τρέχον περιβάλλον εξαρτήσεων δ' με τις επιπλέον εξαρτήσεις που προκύπτουν από τους ορισμούς των μεταβλητών.

[3] Στο σημείο αυτό, θεωρούμε ότι το $\mathcal{P}(\textbf{\textit{Dim}})$ έχει τη δομή πεδίου διατεταγμένου με βάση τη σχέση υποσυνόλου. Το $\emptyset$ είναι το ελάχιστο στοιχείο του.

$$\mathcal{D}[\![(defn)^*]\!] \;:\; \textbf{\textit{End}} \to \textbf{\textit{End}} \to \textbf{\textit{End}}$$

$$\mathcal{D}[\![\epsilon]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ id$$
$$\mathcal{D}[\![defn\ (defn)^*]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \mathcal{D}[\![(defn)^*]\!]\ \delta \circ \mathcal{D}[\![defn]\!]\ \delta$$

$$\mathcal{D}[\![defn]\!] \;:\; \textbf{\textit{End}} \to \textbf{\textit{End}} \to \textbf{\textit{End}}$$

$$\mathcal{D}[\![type\ var = expr]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \lambda\delta' : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ v = \{\!|var|\!\}\ \textbf{\textit{in}}\ \delta'\{v \mapsto \mathcal{D}[\![expr]\!]\ \delta\}$$

Τέλος, οι διαστάσεις από τις οποίες εξαρτάται μια έκφραση υπολογίζονται εύκολα βάσει της δομής της έκφρασης.

$$\mathcal{D}[\![expr]\!] \;:\; \textbf{\textit{End}} \to \mathcal{P}(\textbf{\textit{Dim}})$$

$$\mathcal{D}[\![var]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ v = \{\!|var|\!\}\ \textbf{\textit{in}}\ \delta\ v$$
$$\mathcal{D}[\![num]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \emptyset$$
$$\mathcal{D}[\![\texttt{true}]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \emptyset$$
$$\mathcal{D}[\![\texttt{false}]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \emptyset$$

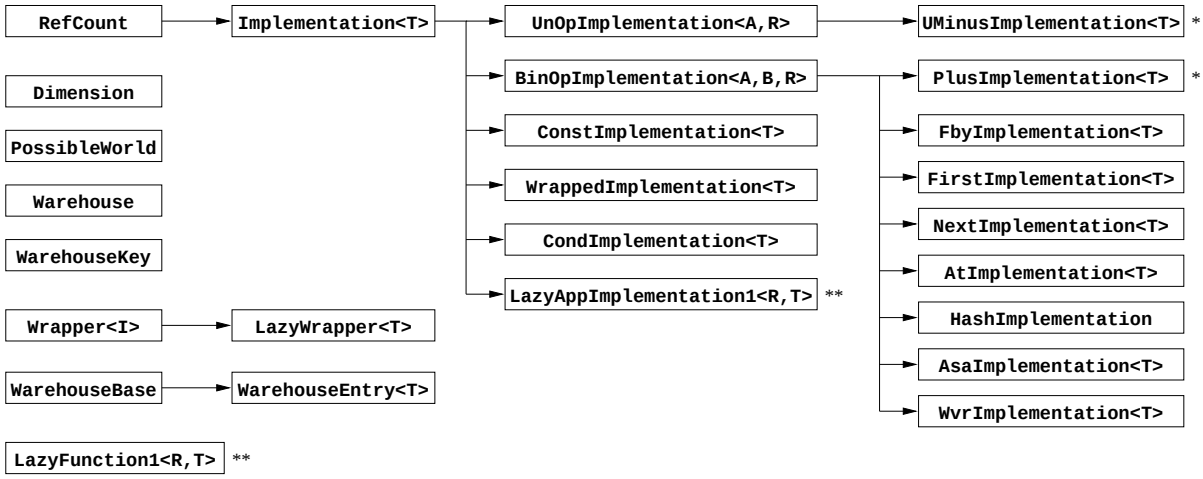
$$\mathcal{D}[\![unop\ expr]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \mathcal{D}[\![expr]\!]\ \delta$$
$$\mathcal{D}[\![expr_1\ binop\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \mathcal{D}[\![expr_1]\!]\ \delta \cup \mathcal{D}[\![expr_2]\!]\ \delta$$
$$\mathcal{D}[\![\texttt{if}\ expr\ \texttt{then}\ expr_1\ \texttt{else}\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \mathcal{D}[\![expr]\!]\ \delta \cup \mathcal{D}[\![expr_1]\!]\ \delta \cup \mathcal{D}[\![expr_2]\!]\ \delta$$

Οι ενδιαφέρουσες περιπτώσεις είναι αυτές των νοηματικών τελεστών, όπου οι εξαρτήσεις προκύπτουν άμεσα από τη σημασιολογία κάθε τελεστή.

$$\mathcal{D}[\![\texttt{value.}\,dim]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ d = \{\!|dim|\!\}\ \textbf{\textit{in}}$$
$$\quad \{\,d\,\}$$
$$\mathcal{D}[\![\texttt{first.}\,dim\ expr]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ d = \{\!|dim|\!\}\ \textbf{\textit{in}}$$
$$\quad \mathcal{D}[\![expr]\!]\ \delta - \{\,d\,\}$$
$$\mathcal{D}[\![\texttt{next.}\,dim\ expr]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.$$
$$\quad \mathcal{D}[\![expr]\!]\ \delta$$
$$\mathcal{D}[\![expr_1\ \texttt{fby.}\,dim\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ d = \{\!|dim|\!\}\ \textbf{\textit{in}}$$
$$\quad \mathcal{D}[\![expr_1]\!]\ \delta \cup \mathcal{D}[\![expr_2]\!]\ \delta \cup \{\,d\,\}$$
$$\mathcal{D}[\![expr_1\ \texttt{at.}\,dim\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ d = \{\!|dim|\!\}\ \textbf{\textit{in}}$$
$$\quad (\mathcal{D}[\![expr_1]\!]\ \delta - \{\,d\,\}) \cup \mathcal{D}[\![expr_2]\!]\ \delta$$
$$\mathcal{D}[\![expr_1\ \texttt{asa.}\,dim\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.\ \textbf{\textit{let}}\ d = \{\!|dim|\!\}\ \textbf{\textit{in}}$$
$$\quad (\mathcal{D}[\![expr_1]\!]\ \delta \cup \mathcal{D}[\![expr_2]\!]\ \delta) - \{\,d\,\}$$
$$\mathcal{D}[\![expr_1\ \texttt{wvr.}\,dim\ expr_2]\!] \;=\; \lambda\delta : \textbf{\textit{End}}.$$
$$\quad \mathcal{D}[\![expr_1]\!]\ \delta \cup \mathcal{D}[\![expr_2]\!]\ \delta$$

3 Υλοποίηση

Στις επόμενες παραγράφους περιγράφεται ο τρόπος ενσωμάτωσης της GLU$^\sharp$ στη C++. Η ιεραρχία των κλάσεων και των templates κλάσεων φαίνεται στο σχήμα 3. Οι κλάσεις που σημειώνονται με $*$ επαναλαμβάνονται με παρόμοιο τρόπο για κάθε τελεστή της C++ που είναι επιθυμητό να χρησιμοποιείται σε ενσωματωμένο κώδικα GLU$^\sharp$. Επίσης, οι κλάσεις που σημειώνονται με $**$ επαναλαμβάνονται με παρόμοιο τρόπο για κάθε αριθμό παραμέτρων, όπως περιγράφεται στην ενότητα 3.5. Τα αρχεία της υλοποίησης καθώς και τα παραδείγματα της επόμενης ενότητας είναι διαθέσιμα



Σχήμα 3: Ιεραρχία κλάσεων στην υλοποίηση της GLU♮.

στη διεύθυνση `http://www.softlab.ntua.gr/~nickie/Research/Intensional/`. Στη συνέχεια δίνεται μια σύντομη περιγραφή της υλοποίησης. Για κάθε κλάση περιγράφονται οι επικεφαλίδες των μελών της αλλά όχι ο αντίστοιχος κώδικας που τα υλοποιεί. Στις επικεφαλίδες χρησιμοποιούνται τα παρακάτω ενδεικτικά σύμβολα:

- ★ Σχέση κληρονομικότητας.
- ■ Πεδίο (attribute) της κλάσης.
- ▶ Μέθοδος της κλάσης ή άλλου είδους ορισμός.

3.1 Περιβλήματα και μέτρηση αναφορών

Προκειμένου να αποφευχθεί η αντιγραφή αντικειμένων, διαδικασία χρονοβόρα και δαπανηρή σε μνήμη, η υλοποίηση της GLU♮ κάνει διάκριση ανάμεσα σε αντικείμενα *περιβλήματα* (wrappers) και αντικείμενα *υλοποιήσεις* (implementations).[4] Τα αντικείμενα που αντιγράφονται είναι περιβλήματα, τα οποία είναι πολύ μικρού μεγέθους και συνήθως περιέχουν μόνο ένα δείκτη σε κάποια υλοποίηση. Οι υλοποιήσεις δεν αντιγράφονται, παρά μόνο οι δείκτες προς αυτές. Με τον τρόπο αυτό, περισσότερα περιβλήματα μπορούν να αναφέρονται στην ίδια υλοποίηση και είναι αναγκαία η ύπαρξη ενός μηχανισμού διαχείρισης των πολλαπλών αναφορών. Αυτό επιτυγχάνεται στο χαμηλότερο επίπεδο με τις κλάσεις `RefCount` και `Wrapper`.

Κλάση: `RefCount`

Η κλάση αυτή είναι η βασική κλάση που κληρονομείται από όλες τις υλοποιήσεις. Ο ρόλος της είναι η μέτρηση αναφορών σε αντικείμενα αυτών των κλάσεων και η καταστροφή όσων αντικειμένων είναι άχρηστα. Η ύπαρξη του μηχανισμού αυτού εξασφαλίζει σχετικά εύκολα την αποφυγή διαρροών μνήμης. Ο μηχανισμός μέτρησης αναφορών είναι αρκετά απλός και αποφεύγονται οι κυκλικές αναφορές. Τα μέλη της κλάσης είναι τα ακόλουθα.

[4] Το μοντέλο αυτό είναι γνωστό στην ανάπτυξη αντικειμενοστρεφών προγραμμάτων ως μοντέλο *γράμματος/φακέλου* (letter/envelope).

- `int` **`myCount`** [mutable, private]
- **`RefCount`** `()` [inline, protected]
- ∼**`RefCount`** `()` [inline, protected, virtual]
- `RefCount &` **`refUsed`** `() const` [inline]
- `void` **`refDiscarded`** `() const` [inline]

Ο αριθμός αναφορών στο δοθέν αντικείμενο αποθηκεύεται στο πεδίο `myCount`. Οι μέθοδοι `refUsed` και `refDiscarded` αυξάνουν και μειώνουν αντίστοιχα τον αριθμό των αναφορών. Επιπλέον, αν μετά από μια κλήση της `refDiscarded` ο μετρητής γίνει ίσος με μηδέν, τότε το αντικείμενο καταστρέφεται αυτόματα.

Template κλάσης: `Wrapper<I>`

Η κλάση `Wrapper<I>` είναι η βασική κλάση που κληρονομείται από όλα τα περιβλήματα υλοποιήσεων τύπου `I`. Η κλάση `I` πρέπει να κληρονομεί την `RefCount`. Στην `Wrapper<I>` γίνεται η διαχείριση των αναφορών. Τα πεδία της κλάσης είναι τα ακόλουθα.

- `I *` **`impl`** [protected]
- **`Wrapper`** `()` [inline]
- **`Wrapper`** `(const Wrapper & w)` [inline]
- **`Wrapper`** `(I & i)` [inline]
- ∼**`Wrapper`** `()` [inline]
- `Wrapper<I> &` **`operator=`** `(const Wrapper<I> & w)` [inline]
- `Wrapper<I> &` **`operator=`** `(const I & i)` [inline]
- **`operator I &`** `() const` [inline]
- `ostream &` **`operator`**`<< (ostream & str, const Wrapper<I> & w)` [friend]

Κάθε περίβλημα μπορεί να είναι συσχετισμένο με μια υλοποίηση, η αναφορά της οποίας αποθηκεύεται στο πεδίο `impl`. Η συσχέτιση ενός περιβλήματος με μια υλοποίηση προκαλεί κλήση της μεθόδου `refUsed` της υλοποίησης. Ομοίως, η αποσυσχέτιση ενός περιβλήματος από μια υλοποίηση προκαλεί κλήση της `refDiscarded` και πιθανώς την καταστροφή της υλοποίησης.

3.2 Διαστάσεις και δυνατοί κόσμοι

Η υλοποίηση των διαστάσεων και των δυνατών κόσμων γίνεται με τις κλάσεις που περιγράφονται σε αυτή την παράγραφο.

Συνώνυμο τύπου: `DimIdentifier`

Χρησιμοποιείται για τα αναγνωριστικά διαστάσεων, αντί ονομάτων.

- `typedef unsigned int` **`DimIdentifier`**

Συνώνυμο τύπου: `DimValue`

Χρησιμοποιείται για τις τιμές των σημείων μιας διάστασης, οι οποίες είναι φυσικοί αριθμοί.

▶ `typedef unsigned int` **`DimValue`**

Κλάση: `Dimension`

Η κλάση `Dimension` υλοποιεί μια διάσταση. Η μόνη πληροφορία που περιέχει είναι το μοναδικό αναγνωριστικό της διάστασης, το οποίο αποδίδεται κατά την κατασκευή της διάστασης. Τα πεδία της κλάσης είναι τα ακόλουθα.

■ `DimIdentifier` **`dimId`** [protected]

■ `DimIdentifier` **`nextDimId`** [static, protected]

▶ **`Dimension`** `()` [inline]

▶ **`Dimension`** `(const Dimension & d)` [inline]

▶ `Dimension &` **`operator=`** `(const Dimension & d)` [inline]

▶ `class` **`PossibleWorld`** [friend]

▶ `ostream &` **`operator`**`<<` `(ostream & str, const Dimension & d)` [friend]

▶ `bool` **`operator==`** `(const Dimension & x, const Dimension & y)` [friend]

▶ `bool` **`operator`**`<` `(const Dimension & x, const Dimension & y)` [friend]

Κλάση: `PossibleWorld`

Η κλάση `PossibleWorld` υλοποιεί ένα δυνατό κόσμο. Τα αντικείμενά αυτής της κλάσης αντιστοιχούν κάθε διάσταση στο τρέχον σημείο της. Οι λεπτομέρειες υλοποίησης της κλάσης `PossibleWorld` παραλείπονται.

■ `const PossibleWorld` **`empty`** [static]

Αντιστοιχεί στον κόσμο w_0.

▶ **`PossibleWorld`** `()` [inline]

▶ **`PossibleWorld`** `(const PossibleWorld & w)` [inline]

▶ ∼**`PossibleWorld`** `()` [inline]

▶ `PossibleWorld &` **`operator=`** `(const PossibleWorld & w)` [inline]

▶ `void` **`set`** `(const Dimension & d, DimValue dimVal)` [inline]

Κάνει το τρέχον σημείο της διάστασης `d` ίσο με `dimVal`.

▶ `DimValue` **`get`** `(const Dimension & d,` [inline]
`bool required = true) const`

Επιστρέφει το τρέχον σημείο της διάστασης `d`. Σε περίπτωση που η διάσταση δεν είναι ορισμένη, αν η παράμετρος `required` είναι αληθής τότε προκύπτει σφάλμα εκτέλεσης. Διαφορετικά επιστρέφεται μια ειδική τιμή που αντιστοιχεί σε μη υπαρκτό σημείο.

▶ `size_t` **`hash`** `() const` [inline]

Χρησιμοποιείται από την κλάση `WarehouseKey`.

▶ `class` **`WarehouseKey`** [friend]

▶ `bool` **`operator==`** `(const PossibleWorld & w1, const PossibleWorld & w2)` [friend]

▶ `ostream &` **`operator<<`** `(ostream & str, const PossibleWorld & w)` [friend]

3.3 Αποθήκη υπολογισμένων τιμών

Για τη βελτίωση της απόδοσης εκτέλεσης ενός προγράμματος C++ με ενσωματωμένο κώδικα GLU♮, η υλοποίηση συμπεριλαμβάνει μια *αποθήκη* (warehouse) στην οποία αποθηκεύονται οι υπολογισμένες τιμές των εκφράσεων, ώστε να αποφεύγεται ο μελλοντικός επανυπολογισμός τους. Αυτό είναι εφικτό λόγω του συναρτησιακού χαρακτήρα της GLU♮, ο οποίος εξασφαλίζει ότι η τιμή μιας έκφρασης σε ένα δοθέντα δυνατό κόσμο δεν μπορεί να μεταβάλλεται.

Κάθε φορά που ζητείται να υπολογισθεί μια έκφραση σε ένα δυνατό κόσμο, ελέγχεται αν η τιμή υπάρχει ήδη στην αποθήκη. Σε αντίθετη περίπτωση, γίνεται ο υπολογισμός και αποθηκεύεται στην αποθήκη μια εγγραφή η οποία περιέχει:

- Ένα αναγνωριστικό της έκφρασης που υπολογίστηκε.
- Το δυνατό κόσμο στον οποίο έγινε ο υπολογισμός. Από τον κόσμο αυτό δεν αποθηκεύονται απαραίτητα τα σημεία όλων των διαστάσεων, αλλά μόνο αυτών από τις οποίες εξαρτάται η έκφραση. Έτσι εξασφαλίζεται αφενός χώρος στη μνήμη, αφετέρου ότι ο υπολογισμός δε θα επαναληφθεί ακόμα και σε διαφορετικούς δυνατούς κόσμους, που δε διαφέρουν όσον αφορά στις κρίσιμες διαστάσεις.
- Την τιμή που υπολογίστηκε.

Το μέγεθος της αποθήκης είναι πεπερασμένο και, όταν γεμίσει, οι νεώτερες υπολογισθείσες τιμές αντικαθιστούν τις παλαιότερες.

Κλάση: `Warehouse`

Υλοποιεί την αποθήκη. Χρησιμοποιεί ένα πίνακα κατακερματισμού για τη γρήγορη προσπέλαση στις εγγραφές της αποθήκης.

■ `hash_map<WarehouseKey, WarehouseBase *>` **`myMap`** [protected]

■ `unsigned long` **`mySize`** [protected]

Το μέγεθος της αποθήκης (σε εγγραφές).

■ `unsigned long` **`myCount`** [protected]

Ο αριθμός των εγγραφών που χρησιμοποιείται.

▶ **`Warehouse`** `(unsigned long size)` [inline]

▶ **`~Warehouse`** `()` [inline]

▶ `void` **`insert`** `(const WarehouseKey & key, WarehouseBase * value)` [inline]

Προσθήκη μιας εγγραφής στην αποθήκη.

▶ `WarehouseBase *` **`lookup`** `(const WarehouseKey & key)` [inline]

Αναζήτηση μιας εγγραφής στην αποθήκη.

Κλάση: `WarehouseBase`

Η αφηρημένη αυτή κλάση χρησιμοποιείται ως βάση για την κατασκευή όλων των εγγραφών της αποθήκης. Είναι απαραίτητη η χρήση πολυμορφισμού γιατί οι υπολογισθείσες τιμές που τοποθετούνται στην αποθήκη είναι γενικά αντικείμενα διαφορετικών κλάσεων.

- ▶ **`WarehouseBase`** `()` [inline, protected]
- ▶ ∼**`WarehouseBase`** `()` [inline, virtual]

Template κλάσης: `WarehouseEntry<T>`

Υλοποιεί μια εγγραφή, η υπολογισθείσα τιμή της οποίας είναι ένα αντικείμενο της κλάσης `T`.

- ★ Κληρονομεί την κλάση: `WarehouseBase` [public]
- ■ `T *` **`element`** [protected]
- ▶ **`WarehouseEntry`** `()` [inline, protected]
- ▶ **`WarehouseEntry`** `(const T & t)` [inline]
- ▶ ∼**`WarehouseEntry`** `()` [inline, virtual]
- ▶ `const T *` **`fetch`** `() const` [inline]

Κλάση: `WarehouseKey`

Η κλάση αυτή υλοποιεί το κλειδί στον πίνακα κατακερματισμού που χρησιμοποιείται για την αποθήκη υπολογισμένων τιμών. Περιέχει ως πληροφορία το αναγνωριστικό της έκφρασης που υπολογίστηκε και το δυνατό κόσμο στον οποίο έγινε ο υπολογισμός.

- ▶ `typedef unsigned long int` **`Identifier`** [protected]
- ■ `Identifier` **`myId`** [protected]
- ■ `PossibleWorld` **`myWorld`** [protected]
- ▶ `Identifier` **`uniqueId`** [static, protected]
- ▶ **`WarehouseKey`** `(Identifier id)` [inline]
- ▶ **`WarehouseKey`** `(const WarehouseKey & k)` [inline]
- ▶ `WarehouseKey &` **`operator=`** `(const WarehouseKey & k)` [inline]
- ▶ `Identifier` **`getUniqueId`** `()` [inline, static]

  Χρησιμοποιείται για την κατασκευή μοναδικών αναγνωριστικών έτσι ώστε να υπάρχει ένα για κάθε διαφορετική έκφραση που υπολογίζεται.

- ▶ `void` **`addBindings`** `(const set<Dimension> & depSet, const PossibleWorld & w)` [inline]

  Προσθέτει στον δυνατό κόσμο `myId` τα σημεία των διαστάσεων που βρίσκονται στο σύνολο `depSet`, παίρνοντάς τα από τον κόσμο `w`. Κατ' αυτό τον τρόπο, αν το `depSet` είναι το σύνολο των διαστάσεων από τις οποίες εξαρτάται μια έκφραση, αποθηκεύονται στην εγγραφή μόνο τα σημεία αυτών των διαστάσεων.

- ▶ `class` **`hash`**`<`**`WarehouseKey`**`>` [friend]

▶ `bool` **`operator==`** `(const WarehouseKey & k1, const WarehouseKey & k2)` [friend]

▶ `ostream &` **`operator<<`** `(ostream & str, const WarehouseKey & k)` [friend]

Στιγμιότυπο template κλάσης: `hash<WarehouseKey>`

Η κλάση αυτή υλοποιεί μια απλή συνάρτηση κατακερματισμού για αντικείμενα της κλάσης `WarehouseKey`.

▶ `size_t` **`operator()`** `(const WarehouseKey & key) const` [inline]

3.4 Οκνηρές υλοποίησεις και περιβλήματα

Η ενσωμάτωση της GLU♮ στη C++ βασίζεται σε μια ιεραρχία κλάσεων και templates κλάσεων που αντιπροσωπεύουν οκνηρές υλοποιήσεις και περιβλήματα. Με τον τρόπο αυτό υλοποιούνται στη C++ τα *οκνηρά αντικείμενα* (lazy objects) της GLU♮, η τιμή των οποίων υπολογίζεται μόνο όταν χρειάζεται να χρησιμοποιηθεί και γενικά εξαρτάται από το δυνατό κόσμο στον οποίο γίνεται ο υπολογισμός.

Template κλάσης: `Implementation<T>`

Τα αντικείμενα της κλάσης `Implementation<T>` είναι υλοποιήσεις οκνηρών αντικειμένων τύπου `T`. Η κλάση αυτή είναι αφηρημένη και δεν υλοποιεί παρά ελάχιστες μεθόδους. Αυτό γίνεται στις κλάσεις που την κληρονομούν.

★ Κληρονομεί την κλάση: `RefCount` [public]

■ `set<Dimension>` **`depSet`** [protected]

Το σύνολο των διαστάσεων από τις οποίες εξαρτάται η τιμή ενός οκνηρού αντικειμένου.

■ `enum {` **`READY, AGAIN, WAIT`** `}` **`depFlag`** [protected]

Το πεδίο αυτό υποδηλώνει αν έχει βρεθεί το σύνολο `depSet`. Οι δυνατές περιπτώσεις είναι: να έχει βρεθεί το σύνολο (`READY`), να μην έχει ακόμα βρεθεί (`AGAIN`) ή ο υπολογισμός να μην έχει ακόμα ολοκληρωθεί (`WAIT`). Ο αλγόριθμος υπολογισμού του συνόλου απαιτεί τη διάκριση των δυο τελευταίων περιπτώσεων για να αντιμετωπισθούν οι αναδρομικοί ορισμοί.

▶ **`Implementation`** `()` [inline, protected]

▶ `~`**`Implementation`** `()` [inline, protected, virtual]

▶ `T` **`evaluate`** `()` [pure virtual]

Η συνάρτηση αυτή υπολογίζει την τιμή του αντικειμένου στον τρέχοντα δυνατό κόσμο. Για το σκοπό αυτό, συνεργάζεται με την αποθήκη υπολογισμένων τιμών, μέσω των συναρτήσεων `whLookup` και `whInsert`.

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

Υπολογίζει το σύνολο `depSet`.

▶ `void` **`depInsert`** `(const Dimension & d)` [inline, protected]

Προσθέτει στο σύνολο `depSet` τη διάσταση `d`.

▶ `void` **`depExclude`** `(const Dimension & d)` [inline, protected]

Αφαιρεί από το σύνολο `depSet` τη διάσταση `d`.

▶ `void` **`depAdd`** `(const set<Dimension> & s)` [inline, protected]

Προσθέτει στο σύνολο `depSet` όλες τις διαστάσεις που περιέχονται στο σύνολο `s`.

▶ `const set<Dimension> &` **`depGet`** `(bool exhaustive)` [inline]

Υπολογίζει, αν αυτό είναι απαραίτητο, και επιστρέφει το σύνολο `depSet`. Η παράμετρος `exhaustive` υποδεικνύει αν ο υπολογισμός πρέπει να γίνει εξαντλητικά, υπολογίζοντας και τις εξαρτήσεις όλων των υλοποιήσεων από τις οποίες εξαρτάται η παρούσα. Στην περίπτωση αυτή, ο υπολογισμός επαναλαμβάνεται μέχρις ότου το αποτέλεσμα δε μεταβληθεί μεταξύ δυο διαδοχικών επαναλήψεων.

▶ `const set<Dimension> &` **`depRaw`** `() const` [inline]

Επιστρέφει την τρέχουσα τιμή του `depSet`.

▶ `bool` **`depMustRecalc`** `() const` [inline]

Επιστρέφει `true` αν το σύνολο `depSet` πρέπει να υπολογισθεί και πάλι, διαφορετικά `false`.

▶ `void` **`printOn`** `(ostream & str) const` [protected, pure virtual]

▶ `class` **`LazyWrapper<T>`** [friend]

▶ `ostream &` **`operator<<`** `(ostream & str,` [friend]
`Implementation<T> & impl)`

Αντικείμενο: `theWorld`

Παριστάνει ανά πάσα στιγμή τον τρέχοντα δυνατό κόσμο, στον οποίο γίνεται η αποτίμηση όλων των οκνηρών αντικειμένων.

■ `PossibleWorld` **`theWorld`**

Template συνάρτησης: `whLookup`

Ελέγχει αν η τιμή της υλοποίησης `i`, στην οποία έχει αποδοθεί το κλειδί `id`, έχει προηγουμένως υπολογισθεί στο δυνατό κόσμο `w`. Αν ναι, επιστρέφεται ένας δείκτης στο ήδη υπολογισμένο αποτέλεσμα, διαφορετικά επιστρέφεται `NULL`.

▶ `template <class T>`
`const T *` **`whLookup`** `(Implementation<T> & i,` [inline]
`WarehouseKey::Identifier id,`
`const PossibleWorld & w)`

Template συνάρτησης: `whInsert`

Αποθηκεύει στην αποθήκη την τιμή `t` για την υλοποίηση `i`, στην οποία έχει αποδοθεί το κλειδί `id`. Ο κόσμος `w` είναι ο δυνατός κόσμος στον οποίο έγινε ο υπολογισμός.

▶ `template <class T>`
`const T &` **`whInsert`** `(Implementation<T> & i,` [inline]
`WarehouseKey::Identifier id,`
`const PossibleWorld & w,`
`const T &t)`

Template κλάσης: `LazyWrapper<T>`

Για κάθε τύπο `T` της C++, το template αυτό ορίζει ένα τύπο `LazyWrapper<T>`, τα στοιχεία του οποίου είναι οκνηρά αντικείμενα του τύπου `T`. Το συνώνυμο `GLU` μπορεί να χρησιμοποιείται στη θέση του `LazyWrapper`. Οι τύποι δεδομένων `real` και `bool` της GLU$^\sharp$ μπορούν να μεταφρασθούν ως `GLU<double>` και `GLU<bool>` αντίστοιχα.

★ Κληρονομεί την κλάση: `Wrapper< Implementation<T> >` [public]

Η κλάση περιέχει υλοποιήσεις της μορφής `Implementation<T>`. Οι περισσότερες μέθοδοι απλά καλούν τις αντίστοιχες μεθόδους της υλοποίησης. Ειδική μέριμνα λαμβάνεται για την αντιμετώπιση αναδρομικών ορισμών.

▶ `WrappedImplementation<T> * wrapped` [mutable, protected]

Αν η υλοποίηση που αντιστοιχεί στο περίβλημα δεν έχει ακόμα ορισθεί, το πεδίο αυτό περιέχει ένα δείκτη σε μια ενδιάμεση υλοποίηση, μέσω της οποίας είναι δυνατό να υλοποιηθούν αναδρομικοί ορισμοί.

▶ **`LazyWrapper`** `()` [inline]

▶ **`LazyWrapper`** `(const LazyWrapper & w)` [inline]

▶ **`LazyWrapper`** `(Implementation<T> & i)` [inline]

▶ **`LazyWrapper`** `(const T & t)` [inline]

Κατασκευάζει ένα οκνηρό αντικείμενο με σταθερή τιμή, χρησιμοποιώντας την υλοποίηση `ConstImplementation<T>`.

▶ `const LazyWrapper<T> &` **`operator=`** `(const LazyWrapper<T> & w)` [inline]

Τελεστής ανάθεσης που μεταξύ άλλων φροντίζει ώστε να ενημερωθεί η ενδιάμεση υλοποίηση με την τελική έκφραση που ορίζει το αντικείμενο.

▶ **`operator Implementation &`** `() const` [inline]

Επιστρέφει την τρέχουσα υλοποίηση που περιέχεται στο περίβλημα. Επιστρέφει την ενδιάμεση υλοποίηση, αν το περίβλημα δεν περιέχει ακόμα κάποια υλοποίηση.

▶ `T` **`evaluate`** `() const` [inline]

▶ `void` **`depCalc`** `() const` [inline, protected]

Template κλάσης: `ConstImplementation<T>`

Υλοποίηση ενός σταθερού αντικειμένου, η τιμή του οποίου είναι ήδη υπολογισμένη και δεν εξαρτάται από τον τρέχοντα δυνατό κόσμο. Τέτοιες σταθερές τιμές δεν αποθηκεύονται.

★ Κληρονομεί την κλάση: `Implementation<T>` [public]

■ `T` **`constant`** [protected]

▶ **`ConstImplementation`** `(const T & t)` [inline]

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, virtual]

Template κλάσης: `WrappedImplementation<T>`

Υλοποίηση ενός αντικειμένου το οποίο παίρνει την τιμή του άμεσα από μια άλλη υλοποίηση. Χρησιμοποιείται ως ενδιάμεση υλοποίηση μεταβλητών που δεν έχουν ακόμα ορισθεί. Τέτοιες υλοποιήσεις χρησιμοποιούνται αναπόφευκτα σε αναδρομικούς ορισμούς.

★ Κληρονομεί την κλάση: `Implementation<T>` [public]

■ `Implementation<T> * patched` [protected]

Περιέχει την τελική τιμή της ενδιάμεσης υλοποίησης, όταν αυτή τελικά γίνει γνωστή.

▶ **`WrappedImplementation ()`** [inline, protected]

▶ **`~WrappedImplementation ()`** [inline, virtual, protected]

▶ `void` **`depCalc ()`** [inline, protected, virtual]

▶ `T` **`evaluate ()`** [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

▶ `void` **`patch`** `(Implementation<T> * i)` [inline, protected]

Ενημερώνει την τιμή του πεδίου `patched`, μέσω του οποίου η ενδιάμεση υλοποίηση παίρνει την τελική της τιμή.

▶ `class` **`LazyWrapper<T>`** [friend]

Template κλάσης: `UnOpImplementation<A, R>`

Υλοποίηση τελεστή με ένα τελούμενο τύπου `A` και αποτέλεσμα τύπου `R`. Χρησιμοποιείται ως βασική κλάση για τις υλοποιήσεις συγκεκριμένων τελεστών.

★ Κληρονομεί την κλάση: `Implementation<R>` [public]

■ `WarehouseKey::Identifier` **`whIdentifier`** [protected]

Το κλειδί με το οποίο αποθηκεύεται η υπολογισμένη τιμή αυτού του αντικειμένου.

■ `const char *` **`opName`** [protected]

Το όνομα του τελεστή.

■ `Implementation<A> *` **`implArg`** [protected]

Η υλοποίηση του τελούμενου.

▶ **`UnOpImplementation`** `(const char * n, Implementation<A> & a)` [inline, protected]

▶ **`~UnOpImplementation ()`** [inline, protected, virtual]

▶ `void` **`depCalc ()`** [inline, protected, virtual]

Template κλάσης: `UMinusImplementation<T>`

Υλοποίηση του τελεστή αρνητικού προσήμου. Με παρόμοιο τρόπο υλοποιούνται και οι υπόλοιποι τελεστές με ένα τελούμενο. Σε καθένα από αυτούς αντιστοιχεί μια κλάση.

★ Κληρονομεί την κλάση: `UnOpImplementation<T, T>` [public]

▶ **`UMinusImplementation`** `(Implementation<A> & a)` [inline]

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, virtual]

Template συνάρτησης: `operator-`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή αρνητικού προσήμου σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`operator-`** `(const LazyWrapper<T> & a)` [inline]

Template κλάσης: `BinOpImplementation<A, B, R>`

Υλοποίηση τελεστή με δυο τελούμενα τύπου `A` και `B` και αποτέλεσμα τύπου `R`. Χρησιμοποιείται ως βασική κλάση για τις υλοποιήσεις συγκεκριμένων τελεστών.

★ Κληρονομεί την κλάση: `Implementation<R>` [public]

■ `WarehouseKey::Identifier` **`whIdentifier`** [protected]

Το κλειδί με το οποίο αποθηκεύεται η υπολογισμένη τιμή αυτού του αντικειμένου.

■ `const char *` **`opName`** [protected]

Το όνομα του τελεστή.

■ `Implementation<A> *` **`implLeft`** [protected]

Η υλοποίηση του αριστερού τελουμένου.

■ `Implementation<B> *` **`implRight`** [protected]

Η υλοποίηση του δεξιού τελουμένου.

▶ **`BinOpImplementation`** `(const char * n,` [inline, protected]
`Implementation<A> & a,`
`Implementation<B> & b)`

▶ **`~BinOpImplementation`** `()` [inline, protected, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

Template κλάσης: `PlusImplementation<T>`

Υλοποίηση του τελεστή της πρόσθεσης. Με παρόμοιο τρόπο υλοποιούνται και οι υπόλοιποι τελεστές με δύο τελούμενα. Σε καθένα από αυτούς αντιστοιχεί μια κλάση.

★ Κληρονομεί την κλάση: `BinOpImplementation<T, T, T>` [public]

▶ **`PlusImplementation`** `(Implementation<A> & a,` [inline]
`Implementation<B> & b)`

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, virtual]

Template συνάρτησης: `operator+`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή της πρόσθεσης σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`operator+`** `(const LazyWrapper<T> & a,` [inline]
`const LazyWrapper<T> & b)`

Template κλάσης: `CondImplementation<T>`

Η κλάση αυτή υλοποιεί τον τελεστή συνθήκης `if ... then ... else ...`.

★ Κληρονομεί την κλάση: `Implementation<T>` [public]

■ `WarehouseKey::Identifier` **`whIdentifier`** [protected]

Το κλειδί με το οποίο αποθηκεύεται η υπολογισμένη τιμή αυτού του αντικειμένου.

■ `Implementation<bool> *` **`implCond`** [protected]

Η υλοποίηση της συνθήκης.

■ `Implementation<T> *` **`implThen`** [protected]

Η υλοποίηση του ενός εναλλακτικού αντικειμένου.

■ `Implementation<T> *` **`implElse`** [protected]

Η υλοποίηση του άλλου εναλλακτικού αντικειμένου.

▶ **`CondImplementation`** `(Implementation<bool> & c,` [inline]
`Implementation<T> & a,`
`Implementation<T> & b)`

▶ **`~CondImplementation`** `()` [inline, protected, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

Template συνάρτησης: `cond`

Η συνάρτηση αυτή επιτρέπει την κατασκευή οκνηρών αντικειμένων της μορφής `if ... then ... else ...`. Δυστυχώς δεν είναι δυνατή η χρήση του τελεστή `?:` της C++, η υπερφόρτωση του οποίου δεν επιτρέπεται. Επιπλέον, δεν είναι δυνατή η χρήση της εντολής `if` της γλώσσας γιατί αυτό θα οδηγούσε στην άμεση αποτίμηση της συνθήκης και δε θα ήταν σύμφωνο με την οκνηρή σημασιολογία της GLU[♯].

▶ `template <class T>`
`LazyWrapper<T>` **`cond`** `(const LazyWrapper<bool> & c,` [inline]
`const LazyWrapper<T> & a,`
`const LazyWrapper<T> & b)`

Template κλάσης: `FbyImplementation<T>`

Υλοποίηση του νοηματικού τελεστή `fby`.

★ Κληρονομεί την κλάση: `BinOpImplementation<T, T, T>` [public]

■ `Dimension` **`dim`** [protected]

▶ **FbyImplementation** (const Dimension & d, [inline]
Implementation<T> & a,
Implementation<T> & b)

▶ T **evaluate** () [inline, virtual]

▶ void **depCalc** () [inline, protected, virtual]

▶ void **printOn** (ostream & str) const [inline, protected, virtual]

Template συνάρτησης: fby

Συνάρτηση που επιτρέπει τη χρήση του τελεστή fby σε οκνηρά αντικείμενα.

▶ template <class T>
LazyWrapper<T> **fby** (const Dimension & d, [inline]
const LazyWrapper<T> & a,
const LazyWrapper<T> & b)

Template κλάσης: FirstImplementation<T>

Υλοποίηση του νοηματικού τελεστή first.

★ Κληρονομεί την κλάση: UnOpImplementation<T, T> [public]

■ Dimension **dim** [protected]

▶ **FirstImplementation** (const Dimension & d, [inline]
Implementation<T> & a)

▶ T **evaluate** () [inline, virtual]

▶ void **depCalc** () [inline, protected, virtual]

▶ void **printOn** (ostream & str) const [inline, protected, virtual]

Template συνάρτησης: first

Συνάρτηση που επιτρέπει τη χρήση του τελεστή first σε οκνηρά αντικείμενα.

▶ template <class T>
LazyWrapper<T> **first** (const Dimension & d, [inline]
const LazyWrapper<T> & a)

Template κλάσης: NextImplementation<T>

Υλοποίηση του νοηματικού τελεστή next.

★ Κληρονομεί την κλάση: UnOpImplementation<T, T> [public]

■ Dimension **dim** [protected]

▶ **NextImplementation** (const Dimension & d, [inline]
Implementation<T> & a)

▶ T **evaluate** () [inline, virtual]

▶ void **depCalc** () [inline, protected, virtual]

▶ void **printOn** (ostream & str) const [inline, protected, virtual]

Template συνάρτησης: `next`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή `next` σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`next`** `(const Dimension & d,` [inline]
`const LazyWrapper<T> & a)`

Template κλάσης: `AtImplementation<T>`

Υλοποίηση του νοηματικού τελεστή `at`.

★ Κληρονομεί την κλάση: `BinOpImplementation<T, DimValue, T>` [public]

■ `Dimension` **`dim`** [protected]

▶ **`AtImplementation`** `(const Dimension & d,` [inline]
`Implementation<T> & a,`
`Implementation<DimValue> & b)`

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

Template συνάρτησης: `at`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή `at` σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`at`** `(const Dimension & d,` [inline]
`const LazyWrapper<T> & a,`
`const LazyWrapper<DimValue> & b)`

Κλάση: `HashImplementation`

Υλοποίηση του νοηματικού τελεστή `value`.

★ Κληρονομεί την κλάση: `Implementation<DimValue>` [public]

■ `Dimension` **`dim`** [protected]

▶ **`HashImplementation`** `(const Dimension & d)` [inline]

▶ **`DimValue evaluate`** `()` [inline, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

Συνάρτηση: `value`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή `value` ως οκνηρού αντικειμένου.

▶ `LazyWrapper<DimValue> value (const Dimension & d)` [inline]

Template κλάσης: `AsaImplementation<T>`

Υλοποίηση του νοηματικού τελεστή `asa`.

★ Κληρονομεί την κλάση: `BinOpImplementation<T, bool, T>` [public]

■ `Dimension` **`dim`** [protected]

▶ **`AsaImplementation`** `(const Dimension & d,` [inline]
`Implementation<T> & a,`
`Implementation<bool> & b)`

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

Template συνάρτησης: `asa`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή `asa` σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`asa`** `(const Dimension & d,` [inline]
`const LazyWrapper<T> & a,`
`const LazyWrapper<bool> & b)`

Template κλάσης: `WvrImplementation<T>`

Υλοποίηση του νοηματικού τελεστή `wvr`.

★ Κληρονομεί την κλάση: `BinOpImplementation<T, bool, T>` [public]

■ `Dimension` **`dim`** [protected]

▶ **`WvrImplementation`** `(const Dimension & d,` [inline]
`Implementation<T> & a,`
`Implementation<bool> & b)`

▶ `T` **`evaluate`** `()` [inline, virtual]

▶ `void` **`depCalc`** `()` [inline, protected, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, protected, virtual]

Template συνάρτησης: `wvr`

Συνάρτηση που επιτρέπει τη χρήση του τελεστή `wvr` σε οκνηρά αντικείμενα.

▶ `template <class T>`
`LazyWrapper<T>` **`wvr`** `(const Dimension & d,` [inline]
`const LazyWrapper<T> & a,`
`const LazyWrapper<bool> & b)`

3.5 Οκνηρές συναρτήσεις

Οι συναρτήσεις της C++ μπορούν γενικά να χρησιμοποιηθούν για την υλοποίηση νοηματικών συναρτήσεων που δεν υποστηρίζονται απευθείας από την GLU♯. Υπάρχει όμως μια σημασιολογική αντίφαση μεταξύ των οκνηρών νοηματικών συναρτήσεων που θα περίμενε κανείς από την GLU♯ και των πρόθυμων συναρτήσεων της C++, που πάντοτε υπολογίζουν τις παραμέτρους και προχωρούν

στην εκτέλεση της συνάρτησης. Το πρόβλημα γίνεται αντιληπτό με το παράδειγμα που ακολουθεί. Έστω ότι πρόκειται να υλοποιηθεί η συνάρτηση `f` η οποία ορίζεται (σε GLU) ως εξής:

```
f(x) = x fby.t f(x) + 1;
```

όπου υποθέτουμε ότι η παράμετρος `x` καθώς και το αποτέλεσμα της συνάρτησης είναι ακέραιοι αριθμοί. Εύκολα διαπιστώνει κανείς ότι η έκφραση `f(x)` περιγράφει ένα οκνηρό αντικείμενο, η τιμή του οποίου μεταβάλλεται ως προς τη διάσταση `t` παίρνοντας κατ' αύξουσα σειρά όλες τις ακέραιες τιμές που είναι μεγαλύτερες ή ίσες του `x`. Για την υλοποίηση της `f` σε C++ με ενσωματωμένη GLU♮ θα μπορούσε κανείς να γράψει τον εξής κώδικα:

```
GLU<int> f (GLU<int> x)
{
   return fby(t, x, f(x) + 1);
}
```

Ο κώδικας όμως αυτός δεν είναι σωστός. Στη C++, μια έκφραση της μορφής `f(x)` προκαλεί κατ' αρχήν τον υπολογισμό του `x`, που σωστά ως οκνηρό αντικείμενο δεν υπολογίζεται παρά μόνο με κλήση της μεθόδου `evaluate`, όμως στη συνέχεια προκαλεί την κλήση της συνάρτησης `f`. Αυτό με τη σειρά του οδηγεί αμέσως στον υπολογισμό της έκφρασης `fby(t, x, f(x) + 1)`, ο οποίος συμπεριλαμβάνει και πάλι τον υπολογισμό της έκφρασης `f(x)`. Κατά συνέπεια, η εκτέλεση του προγράμματος δε θα τερματιστεί ποτέ.

Στην περίπτωση αυτή πρέπει προφανώς να επιτευχθεί η καθυστέρηση της αποτίμησης της έκφρασης `f(x)` μέσα στο σώμα της συνάρτησης `f`. Αυτό γίνεται με χρήση του template συνάρτησης `lazy`, το οποίο ορίζεται σε αυτή την παράγραφο και επιτρέπει την οκνηρή εφαρμογή συναρτήσεων. Με χρήση του `lazy`, ο κώδικας της `f` γίνεται:

```
GLU<int> f (GLU<int> x)
{
   return fby(t, x, lazy(f)(x) + 1);
}
```

Η έκφραση `lazy(f)` είναι στην ουσία μια συνάρτηση ισοδύναμη της `f`, η εφαρμογή όμως της οποίας γίνεται με οκνηρό τρόπο. Έτσι, η εκτέλεση της συνάρτησης `f` θα γίνει στην πραγματικότητα όταν θα χρειαστεί να αποτιμηθεί το οκνηρό αντικείμενο `lazy(f)(x)`, μέσω μιας κλήσης της μεθόδου `evaluate`.

Template κλάσης: `LazyAppImplementation1<R, T>`

Τα αντικείμενα της κλάσης `LazyAppImplementation1<R, T>` υλοποιούν οκνηρά αντικείμενα τύπου `R`, η τιμή των οποίων προκύπτει από την εφαρμογή κάποιας συνάρτησης πάνω σε μια παράμετρο τύπου `T`.

★ Κληρονομεί την κλάση: `Implementation<R>`

▶ `typedef LazyWrapper<R>` **`FunctionType`** `(T)`

Ο τύπος της συνάρτησης που καλείται.

■ `FunctionType *` **`function`** [protected]

Ένας δείκτης στη συνάρτηση.

■ `T` **`argument`** [protected]

Η υπολογισμένη τιμή της παραμέτρου.

▶ **`LazyAppImplementation1`** `(FunctionType * f, const T & arg)` [inline]

▶ `R` **`evaluate`** `()` [inline, virtual]

▶ `void` **`printOn`** `(ostream & str) const` [inline, virtual]

Template κλάσης: `LazyFunction1<R, T>`

Η κλάση `LazyFunction1<R, T>` υλοποιεί οκνηρές συναρτήσεις με μια παράμετρο τύπου `T` και οκνηρό αποτέλεσμα τύπου `R`.

▶ `typedef LazyWrapper<R>` **`FunctionType`** `(T)`

Ο τύπος της συνάρτησης που πρόκειται να μετατραπεί σε οκνηρή.

■ `FunctionType *` **`function`** [protected]

Ένας δείκτης στη συνάρτηση.

▶ **`LazyFunction1`** `(FunctionType * f)` [inline]

▶ **`LazyFunction1`** `(const LazyFunction1<R, T> & lf)` [inline]

▶ `LazyWrapper<R>` **`operator()`** `(const T & arg) const` [inline]

Ο τελεστής εφαρμογής της συνάρτησης επιστρέφει ένα οκνηρό περίβλημα μιας υλοποίησης τύπου `LazyAppImplementation1<R, T>`.

▶ `class` **`LazyAppImplementation1<R, T>`** [friend]

Template συνάρτησης: `lazy`

Χρησιμοποιείται για τη μετατροπή μιας κοινής συνάρτησης της C++ σε οκνηρή συνάρτηση.

▶
```
template <class R, class T>
LazyFunction1<R, T> lazy (LazyWrapper<R> (* f) (T))
```

Τα παραπάνω templates `LazyAppImplementation1<R, T>`, `LazyFunction1<R, T>` και `lazy` υλοποιούν οκνηρές συναρτήσεις που δέχονται μια παράμετρο. Με παρόμοιο τρόπο έχουν υλοποιηθεί templates για οκνηρές συναρτήσεις χωρίς παραμέτρους ή με $1 < n \leq 3$ παραμέτρους.

3.6 Μετάφραση GLU♯ σε C++

Η ενσωμάτωση κώδικα GLU♯ σε C++ δεν μπορεί να ακολουθεί ακριβώς τους κανόνες σύνταξης της GLU♯, αφενός λόγω περιορισμών στη σύνταξη της C++, αφετέρου λόγω εγγενών αδυναμιών στον τρόπο υλοποίησης των οκνηρών αντικειμένων. Σε αυτή την παράγραφο περιγράφεται ο τρόπος μετάφρασης του κώδικα GLU♯ ώστε να μπορεί να ενσωματωθεί σε ένα πρόγραμμα C++. Η μετάφραση είναι εύκολο να γίνει μηχανικά και δεν επιφέρει σημαντικές αλλαγές στον κώδικα. Ο τρόπος μετάφρασης των εκφράσεων της GLU♯ φαίνεται στο σχήμα 4.

Οι διαστάσεις που χρησιμοποιούνται σε ένα πρόγραμμα GLU♯ πρέπει στη C++ να δηλώνονται ρητά. Αυτό σημαίνει ότι για κάθε διάσταση d πρέπει να υπάρχει μια δήλωση της μορφής:

```
Dimension d;
```

Έκφραση GLU♯: *expr*	Αντίστοιχη έκφραση C++ : $\overline{expr}$
var	*var*
num	*num*
`true`	`true`
`false`	`false`
`(` *expr* `)`	`(` $\overline{expr}$ `)`
unop expr	*unop* $\overline{expr}$
$expr_1$ *binop* $expr_2$	$\overline{expr_1}$ *binop* $\overline{expr_2}$
`if` *expr* `then` $expr_1$ `else` $expr_2$	`cond(`$\overline{expr}$`,` $\overline{expr_1}$`,` $\overline{expr_2}$`)`
`#` `.` *dim*	`value(`*dim*`)`
$expr_1$ `@` `.` *dim* $expr_2$	`at(`*dim*`,` $\overline{expr_1}$`,` $\overline{expr_2}$`)`
`first` `.` *dim expr*	`first(`*dim*`,` $\overline{expr}$`)`
`next` `.` *dim expr*	`next(`*dim*`,` $\overline{expr}$`)`
$expr_1$ `fby` `.` *dim* $expr_2$	`fby(`*dim*`,` $\overline{expr_1}$`,` $\overline{expr_2}$`)`
$expr_1$ `asa` `.` *dim* $expr_2$	`asa(`*dim*`,` $\overline{expr_1}$`,` $\overline{expr_2}$`)`
$expr_1$ `wvr` `.` *dim* $expr_2$	`wvr(`*dim*`,` $\overline{expr_1}$`,` $\overline{expr_2}$`)`

Σχήμα 4: Μετάφραση εκφράσεων GLU♯ σε C++.

Οι τύποι της GLU♯ μεταφράζονται στη C++ σε τύπους οκνηρών αντικειμένων. Ο τύπος `real` γράφεται `GLU<double>` και ο τύπος `bool` γράφεται `GLU<bool>`. Για παράδειγμα, ο ορισμός

```
real min = if x < y then x else y;
```

μπορεί να μεταφραστεί ως

```
GLU<double> min = cond(x < y, x, y);
```

Ιδιαίτερη προσοχή χρειάζεται στην περίπτωση που η ακολουθία ορισμών του προγράμματος περιέχει πρωθύστερα ή αναδρομικούς ορισμούς. Αν η έκφραση που ορίζει τη μεταβλητή `x` αναφέρεται σε μια μεταβλητή `y` που ορίζεται αργότερα, τότε στο πρόγραμμα C++ πρέπει να έχει προηγηθεί μια δήλωση της μεταβλητής `y` με τον κατάλληλο τύπο. Το ίδιο ισχύει αν η έκφραση αναφέρεται στην ίδια τη μεταβλητή `x` που ορίζεται. Για να διευκολυνθεί η αυτοματοποίηση της μετάφρασης, είναι απλούστερο να δηλώνονται στην αρχή όλες οι μεταβλητές με τους τύπους τους, και στη συνέχεια να ακολουθούν οι ορισμοί. Το παρακάτω παράδειγμα ορίζει τις ακολουθίες των περιττών και άρτιων φυσικών αριθμών.

```
real even = 0 fby.t odd + 1;
real odd  = even + 1;
```

και μπορεί να μεταφραστεί ως

```
Dimension t;

GLU<double> even;
GLU<double> odd;
```

```
even = fby(t, 0, odd + 1);
odd  = even + 1;
```

4 Παραδείγματα

Τα παραδείγματα που παρουσιάζονται σε αυτή την ενότητα αφορούν προβλήματα επιστημονικών υπολογισμών και περιγράφονται αναλυτικά στην εργασία [Paqu99]. Μικρές απαραίτητες αλλαγές έχουν γίνει, στην περίπτωση προγραμμάτων GLU που περιέχουν συναρτήσεις ή δομές `where`.

4.1 Αριθμοί Fibonacci

Το παράδειγμα αυτό υπολογίζει τον 17ο αριθμό Fibonacci. Η ακολουθία των αριθμών Fibonacci ορίζεται με τον τύπο:

$$\begin{aligned} \text{fib}(0) &= 0 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \qquad , \quad \forall n \geq 2 \end{aligned}$$

Το πρόγραμμα που ακολουθεί χρησιμοποιεί έναν αμοιβαία αναδρομικό ορισμό και υπολογίζει την ακολουθία των αριθμών Fibonacci στη μεταβλητή `fib`, κατά μήκος της διάστασης `t`.

```
#include "glu.hpp"

int main ()
{
    Dimension t;

    GLU<double> fib;
    GLU<double> g;

    fib = fby(t, 0, g);
    g   = fby(t, 1, fib + g);

    theWorld.set(t, 17);
    cout << "fib(17) = " << fib.evaluate() << endl;

    return 0;
}
```

Το αποτέλεσμα της εκτέλεσης του προγράμματος είναι το ακόλουθο:

```
fib(17) = 1597
```

4.2 Το κόσκινο του Ερατοσθένη

Ένας από τους πιο αποδοτικούς τρόπους υπολογισμού πρώτων αριθμών είναι το *κόσκινο του Ερατοσθένη* (Eratosthene's sieve). Το πρόγραμμα που ακολουθεί υπολογίζει τους πρώτους αριθμούς στη μεταβλητή `prime`, κατά μήκος της διάστασης `y`. Ας σημειωθεί ότι η υλοποίηση του αλγορίθμου αποτελείται από τρεις γραμμές κώδικα GLU[‡], η πρώτη από τις οποίες ορίζει την ακολουθία των φυσικών αριθμών που είναι μεγαλύτεροι του 2.

```
#include "glu.hpp"

int main ()
{
    Dimension x, y;

    GLU<unsigned long int> ints;
    GLU<unsigned long int> sieve;
    GLU<unsigned long int> prime;

    ints  = fby(x, 2, ints + 1);
    sieve = fby(y, ints, wvr(x, sieve, sieve % prime != 0));
    prime = first(x, sieve);

    for (DimValue i=0; i<100; i++) {
        theWorld.set(y, i);
        cout << prime.evaluate() << ", ";
        if (i % 12 == 11) cout << endl;
    }
    cout << "done." << endl;

    return 0;
}
```

Το αποτέλεσμα της εκτέλεσης του προηγούμενου προγράμματος είναι οι εκατό μικρότεροι πρώτοι αριθμοί:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223,
227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281,
283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359,
367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433,
439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503,
509, 521, 523, 541, done.
```

4.3 Αριθμοί Hamming

Αριθμοί Hamming ονομάζονται οι αριθμοί της μορφής:

$2^i \cdot 3^j \cdot 5^k \quad , \quad \forall i, j, k \in \mathbb{N}$

Το παρακάτω πρόγραμμα είναι γραμμένο σε GLU (όχι GLU$^\natural$) και υπολογίζει τους αριθμούς Hamming σε αύξουσα σειρά στη μεταβλητή `hamming` κατά μήκος της διάστασης `t`.

```
dimension t;

merge.d (x, y) = if xx < yy then xx else yy fi
   where xx = x upon.d (yy >= xx);
         yy = y upon.d (xx >= yy);
   end;

hamming = 1 fby.t merge.t (
                     merge.t (2 * hamming,
                              3 * hamming),
                     5 * hamming);
```

Πρέπει να σημειωθεί ότι το παραπάνω πρόγραμμα χρησιμοποιεί τρία στοιχεία της GLU που δεν υπάρχουν στην GLU$^\natural$: συναρτήσεις, το νοηματικό τελεστή `upon.d` και τη δομή `where` για τον ορισμό τοπικών μεταβλητών. Μπορεί όμως εύκολα να μετατραπεί σε ένα πρόγραμμα GLU$^\natural$ ενσωματωμένο σε C++. Για την υλοποίηση αυτών των τριών χαρακτηριστικών της GLU που λείπουν από τη GLU$^\natural$ χρησιμοποιούμε κατάλληλα χαρακτηριστικά της C++. Συγκεκριμένα:

- Οι συναρτήσεις της C++ μπορούν να χρησιμοποιηθούν για την υλοποίηση της `merge`. Η πρώτη παράμετρος αυτής της συνάρτησης είναι διάσταση ενώ οι άλλες δυο είναι κοινές (οκνηρά αντικείμενα). Επιπλέον, στη C++ είναι δυνατό να ορισθεί η `merge` ως template συνάρτησης. Αυτό αντιστοιχεί σε μια πολυμορφική νοηματική συνάρτηση, κάτι που η GLU δεν υποστηρίζει.
- Ο τελεστής `upon.d` είναι και αυτός ουσιαστικά μια συνάρτηση που δέχεται μια παράμετρο διάσταση και δυο κοινές παραμέτρους. Η σημασιολογία του γίνεται αντιληπτή στην υλοποίηση που ακολουθεί. Φυσικά θα μπορούσε να ενσωματωθεί και αυτός ο τελεστής στην GLU$^\natural$, αν κάτι τέτοιο θεωρηθεί σκόπιμο για λόγους ταχύτητας εκτέλεσης.
- Οι μεταβλητές που ορίζονται στη δομή `where` απλά υλοποιούνται ως τοπικές μεταβλητές της C++ στο template συνάρτησης `merge`.

Χρησιμοποιώντας λοιπόν όλα αυτά, το παρακάτω πρόγραμμα C++ με ενσωματωμένη GLU$^\natural$ εκτυπώνει τους εκατό μικρότερους αριθμούς Hamming σε αύξουσα σειρά.

```
#include "glu.hpp"

template <class T>
inline GLU<T> upon (const Dimension & d,
                    const GLU<T> & x,
                    const GLU<bool> & y)
{
    GLU<DimValue> w;
```

```
    w = fby(d, 0, cond(y, w+1, w));
    return at(d, x, w);
}

template <class T>
GLU<T> merge (const Dimension & d,
              const GLU<T> & x,
              const GLU<T> & y)
{
    GLU<T> xx;
    GLU<T> yy;

    xx = upon(d, x, xx <= yy));
    yy = upon(d, y, yy <= xx));

    return cond(xx < yy, xx, yy);
}

int main ()
{
    Dimension t;

    GLU<int> hamming;

    hamming = fby(t, 1, merge(t,
                              merge(t,
                                    2 * hamming,
                                    3 * hamming),
                              5 * hamming));

    for (DimValue i=0; i<100; i++) {
        theWorld.set(t, i);
        cout << hamming.evaluate() << ", ";
        if (i % 10 == 9) cout << endl;
    }
    cout << "done." << endl;

    return 0;
}
```

Το αποτέλεσμα της εκτέλεσης του προγράμματος είναι το ακόλουθο:

```
1, 2, 3, 4, 5, 6, 8, 9, 10, 12,
15, 16, 18, 20, 24, 25, 27, 30, 32, 36,
40, 45, 48, 50, 54, 60, 64, 72, 75, 80,
81, 90, 96, 100, 108, 120, 125, 128, 135, 144,
150, 160, 162, 180, 192, 200, 216, 225, 240, 243,
```

```
250, 256, 270, 288, 300, 320, 324, 360, 375, 384,
400, 405, 432, 450, 480, 486, 500, 512, 540, 576,
600, 625, 640, 648, 675, 720, 729, 750, 768, 800,
810, 864, 900, 960, 972, 1000, 1024, 1080, 1125, 1152,
1200, 1215, 1250, 1280, 1296, 1350, 1440, 1458, 1500, 1536,
done.
```

4.4 Πολλαπλασιασμός πινάκων

Το παρακάτω πρόγραμμα υπολογίζει το γινόμενο δυο πινάκων `a` και `b` με διαστάσεις 4×4.

$$\begin{bmatrix} 72 & 6 & 37 & 91 \\ 14 & 67 & 81 & 37 \\ 26 & 60 & 59 & 32 \\ 16 & 44 & 28 & 86 \end{bmatrix} \cdot \begin{bmatrix} 45 & 7 & 26 & 0 \\ 59 & 90 & 22 & 73 \\ 34 & 94 & 25 & 49 \\ 85 & 51 & 20 & 58 \end{bmatrix} = \begin{bmatrix} 12587 & 9163 & 4749 & 7529 \\ 10482 & 15629 & 4603 & 11006 \\ 9436 & 12760 & 4111 & 9127 \\ 11578 & 11090 & 3804 & 9572 \end{bmatrix}$$

Οι πίνακες μεταβάλλονται ως προς τις διαστάσεις `x` και `y`. Ο αλγόριθμος που χρησιμοποιείται προσφέρεται για εκτέλεση σε περιβάλλον παράλληλης επεξεργασίας, κάτι που φυσικά δεν υποστηρίζεται άμεσα από τη C++. Οι πίνακες πρέπει να είναι τετράγωνοι και το μέγεθός τους να είναι ίσο με κάποια δύναμη του 2.

```
#include <math.h>
#include "glu.hpp"

template <class T>
GLU<T> firstOfPair (const Dimension & a, const GLU<T> & z)
{
    return at(a, z, value(a) * 2);
}

template <class T>
GLU<T> secondOfPair (const Dimension & a, const GLU<T> & z)
{
    return at(a, z, value(a) * 2 + 1);
}

template <class T>
GLU<T> sum (const Dimension & d, const GLU<T> & x, int n)
{
    Dimension t;
    DimValue log2n = (DimValue) (log(n) / log(2));
    GLU<T> y;

    y = fby(t, x, firstOfPair(d, y) + secondOfPair(d, y));

    return at(t, y, log2n);
}
```

```
template <class T>
GLU<T> redim (const Dimension & a, const Dimension & b,
              const GLU<T> & x)
{
    return at(a, x, value(b));
}

template <class T>
GLU<T> product (const Dimension & d1, const Dimension & d2,
                const Dimension & d3, const GLU<T> & m,
                const GLU<T> & n)
{
    return redim(d2, d3, m) * redim(d1, d3, n);
}

template <class T>
GLU<T> mm (const Dimension & x, const Dimension & y,
           const GLU<T> & m1, const GLU<T> & m2, int n)
{
    Dimension z;

    return first(z, sum(z, product(x, y, z, m1, m2), n));
}

int main ()
{
    Dimension x, y;

    GLU<double> a, b;
    GLU<double> eod = -1.0;

    a = fby(x, fby(y, 72, fby(y,  6, fby(y, 37, fby(y, 91, eod)))),
        fby(x, fby(y, 14, fby(y, 67, fby(y, 81, fby(y, 37, eod)))),
        fby(x, fby(y, 26, fby(y, 60, fby(y, 59, fby(y, 32, eod)))),
        fby(x, fby(y, 16, fby(y, 44, fby(y, 28, fby(y, 86, eod)))),
               eod))));

    b = fby(x, fby(y, 45, fby(y,  7, fby(y, 26, fby(y,  0, eod)))),
        fby(x, fby(y, 59, fby(y, 90, fby(y, 22, fby(y, 73, eod)))),
        fby(x, fby(y, 34, fby(y, 94, fby(y, 25, fby(y, 49, eod)))),
        fby(x, fby(y, 85, fby(y, 51, fby(y, 20, fby(y, 58, eod)))),
               eod))));

    GLU<double> c;

    c = mm(x, y, a, b, 4);
```

```
    for (int i=0; i<4; i++) {
        theWorld.set(x, i);
        for (int j=0; j<4; j++) {
            theWorld.set(y, j);
            cout << c.evaluate() << " ";
        }
        cout << endl;
    }
}
```

Το αποτέλεσμα της εκτέλεσης του προγράμματος είναι το ακόλουθο:

```
12587 9163 4749 7529
10482 15629 4603 11006
9436 12760 4111 9127
11578 11090 3804 9572
```

Στο σημείο αυτό, αξίζει να σημειωθεί ότι το παραπάνω πρόγραμμα μπορεί να γραφεί σε ισοδύναμη μορφή χωρίς τη χρήση συναρτήσεων. Το πρόγραμμα που ακολουθεί είναι ισοδύναμο, πολύ μικρότερο αλλά και αρκετά δυσκολότερο στην κατανόηση.

```
#include "glu.hpp"

int main ()
{
    Dimension x, y, z, t;

    GLU<double> a, b;
    GLU<double> eod = -1;

    a = fby(x, fby(y, 72, fby(y,  6, fby(y, 37, fby(y, 91, eod)))),
        fby(x, fby(y, 14, fby(y, 67, fby(y, 81, fby(y, 37, eod)))),
        fby(x, fby(y, 26, fby(y, 60, fby(y, 59, fby(y, 32, eod)))),
        fby(x, fby(y, 16, fby(y, 44, fby(y, 28, fby(y, 86, eod)))),
               eod))));

    b = fby(x, fby(y, 45, fby(y,  7, fby(y, 26, fby(y,  0, eod)))),
        fby(x, fby(y, 59, fby(y, 90, fby(y, 22, fby(y, 73, eod)))),
        fby(x, fby(y, 34, fby(y, 94, fby(y, 25, fby(y, 49, eod)))),
        fby(x, fby(y, 85, fby(y, 51, fby(y, 20, fby(y, 58, eod)))),
               eod))));

    GLU<double> mm, s, p, w, fst, snd;

    mm = first(z, s);
    p = at(y, a, value(z)) * at(x, b, value(z));
    s = at(t, w, 2);                                    // 2 = log(4)
    w = fby(t, p, fst + snd);
```

```
        fst = at(z, w, 2 * value(z));
        snd = at(z, w, 2 * value(z) + 1);

        for (int i=0; i<4; i++) {
            theWorld.set(x, i);
            for (int j=0; j<4; j++) {
                theWorld.set(y, j);
                cout << mm.evaluate() << " ";
            }
            cout << endl;
        }
    }
```

5 Συμπεράσματα

Η ενσωμάτωση της GLU♯ στη C++ οδηγεί σε μια πολύ ενδιαφέρουσα και ιδιαίτερα ισχυρή γλώσσα προγραμματισμού. Η εκφραστική ικανότητά της γίνεται εμφανής στα παραδείγματα της ενότητας 4, τα οποία προέρχονται σχεδόν αυτούσια από την πρόσφατη διδακτορική διατριβή του Joey Paquet [Paqu99] με θέμα την εφαρμογή του νοηματικού προγραμματισμού για την κωδικοποίηση επιστημονικών αλγορίθμων. Η GLU♯/C++ συνδυάζει την πλήρη εκφραστική δύναμη της ιδιαίτερα διαδεδομένης γλώσσας αντικειμενοστρεφούς προγραμματισμού C++ με ένα μικρό και καθαρά συναρτησιακό νοηματικό πυρήνα. Εκμεταλλεύεται την εκφραστικότητα του νοηματικού προγραμματιστικού μοντέλου, διατηρώντας παράλληλα τη συμβατότητα με την περιβάλλουσα γλώσσα C++.

Η αξιολόγηση της υλοποίησής μας δεν έχει ακόμα ολοκληρωθεί. Ο τρόπος ενσωμάτωσης της GLU♯ στη C++ που έχουμε επιτύχει είναι ιδιαίτερα φυσικός και μάλλον μικρές βελτιώσεις μπορούμε να αναμένουμε στο μέλλον προς αυτή την κατεύθυνση. Αν και τα μέχρι τώρα αποτελέσματα των μετρήσεων επίδοσης στην εκτέλεση των νοηματικών προγραμμάτων είναι ιδιαίτερα ενθαρρυντικά, πιστεύουμε ότι υπάρχουν σημαντικά περιθώρια βελτίωσης της υλοποίησής μας σε αυτό τον τομέα, με την καλύτερη σχεδίαση της αποθήκης υπολογισμένων τιμών.

Η σύγκριση της υλοποίησής μας γίνεται αφενός ως προς το μεταγλωττιστή της GLU, που διατίθεται από το SRI και μεταφράζει την GLU σε C, αφετέρου ως προς υπάρχουσες υλοποιήσεις οκνηρών αντικειμένων σε C++. Απ' όσο γνωρίζουμε, η μόνη υλοποίηση οκνηρών αντικειμένων που υποστηρίζει νοηματικά χαρακτηριστικά είναι αυτή που περιγράφεται στο [Zhao97]. Τα παραδείγματα που χρησιμοποιούμε ως benchmarks προέρχονται κυρίως από την περιοχή του επιστημονικού προγραμματισμού. Η συνέχεια αυτής της αξιολόγησης θα αποτελέσει βασικό στόχο της μελλοντικής μας έρευνας.

Βιβλιογραφία

[Agi96] I. Agi, "GLU for Multidimensional Signal Processing", in M. A. Orgun and E. A. Ashcroft, editors, *Intensional Programming I*, pp. 135–148, World Scientific, Singapore, 1996.

[Ashc77] E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration", *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, July 1977.

[Ashc91] E. A. Ashcroft, A. A. Faustini and R. Jagannathan, "An Intensional Language for Parallel Applications Programming", in B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pp. 11–49, ACM Press, 1991.

[Ashc95] E. A. Ashcroft, A. A. Faustini, R. Jagannathan and W. W. Wadge, *Multidimensional Programming*, Oxford University Press, 1995.

[Dowt81] D. R. Dowty, R. E. Wall and S. Peters, *Introduction to Montague Semantics*, Kluwer Academic Publishers, 1981.

[Du90] W. Du and W. W. Wadge, "A 3D Spreadsheet Based on Intensional Logic", *IEEE Software*, vol. 7, no. 3, pp. 78–89, May 1990.

[Du93] W. Du, "An Intensional Approach to Parallel Programming", *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 1, no. 3, pp. 22–32, August 1993.

[Faus88] A. A. Faustini and E. B. Lewis, "Toward a Real-Time Dataflow Language", in J. A. Stankovic and K. Ramamrithan, editors, *Hard Real-Time Systems*, pp. 139–145, IEEE Computer Society Press, 1988.

[Huda98] P. Hudak, "Modular Domain Specific Languages and Tools", in P. Devanbu and J. Poulin, editors, *Proceedings of the 5th International Conference on Software Reuse*, pp. 134–142, Victoria, BC, Canada, June 1998.

[Orgu91] M. A. Orgun, *Intensional Logic Programming*, Ph.D. thesis, Department of Computer Science, University of Victoria, December 1991.

[Orgu92] M. A. Orgun and W. W. Wadge, "Towards a Unified Theory of Intensional Logic Programming", *Journal of Logic Programming*, vol. 13, no. 4, pp. 413–440, August 1992.

[Orgu97] M. A. Orgun and W. Du, "Multi-Dimensional Logic Programming: Theoretical Foundations", *Theoretical Computer Science*, vol. 185, no. 2, pp. 319–345, October 1997.

[Paqu94] J. Paquet and J. Plaice, "On the Design of an Indexical Query Language", in *Proceedings of the 7th International Symposium on Lucid and Intensional Programming*, pp. 28–36, 1994.

[Paqu99] J. Paquet, *Intensional Scientific Programming*, Ph.D. thesis, Université Laval, Québec, Départment d' Informatique, April 1999.

[Plai93a] J. Plaice, R. Khédri and R. Lalement, "From Abstract Time to Real Time", in *Proceedings of the 6th International Symposium on Lucid and Intensional Programming*, pp. 83–83, 1993.

[Plai93b] J. Plaice and W. W. Wadge, "A New Approach to Version Control", *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 268–276, March 1993.

[Rao94] P. Rao and R. Jagannathan, "Developing Scientific Applications in GLU", in *Proceedings of the 7th International Symposium on Lucid and Intensional Programming*, pp. 45–52, 1994.

[Rond97] P. Rondogiannis and W. W. Wadge, "First-Order Functional Languages and Intensional Logic", *Journal of Functional Programming*, vol. 7, no. 1, pp. 73–101, January 1997.

[Rond99] P. Rondogiannis and W. W. Wadge, "Higher-Order Functional Languages and Intensional Logic", *Journal of Functional Programming*, vol. 9, no. 5, pp. 527–564, May 1999.

[Tao94] S. Tao, *Indexical Attribute Grammars*, Ph.D. thesis, Department of Computer Science, University of Victoria, 1994.

[Thom74] R. H. Thomason, editor, *Formal Philosophy: Selected Papers by Richard Montague*, Yale University Press, New Haven, CT, 1974.

[Wadg85] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*, Academic Press, 1985.

[Wadg88] W. W. Wadge, "Tense Logic Programming: A Respectable Alternative", in *Proceedings of the 1988 International Symposium on Lucid and Intensional Programming*, pp. 26–32, 1988.

[Yild97] T. Yildirim, "Intensional HTML", Master's thesis, Department of Computer Science, University of Victoria, 1997.

[Zhao97] Q. Zhao, "Implementation of an Object-Oriented Intensional Programming System", Master's thesis, University of New Brunswick, Canada, September 1997.