Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Ο Νοηματικός Μετασχηματισμός ως Τεχνική Υλοποίησης Συναρτησιακών Γλωσσών Προγραμματισμού

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΓΕΩΡΓΙΟΣ ΦΟΥΡΤΟΥΝΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2014

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Ο Νοηματικός Μετασχηματισμός ως Τεχνική Υλοποίησης Συναρτησιακών Γλωσσών Προγραμματισμού

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

ΓΕΩΡΓΙΟΣ ΦΟΥΡΤΟΥΝΗΣ

Συμβουλευτική Επιτροπή: Νικόλαος Παπασπύρου
Γεώργιος Κολέτσος
Ευστάθιος Ζάχος

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 3η Ιουλίου 2014.

.....	.....	.....
Ν. Παπασπύρου	Γ. Κολέτσος	Ε. Ζάχος
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.
.....	.....	.....
Κ. Σαγώνας	Π. Ροντογιάννης	Ι. Σμαραγδάκης
Αν. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Κ.Π.Α.	Αν. Καθηγητής Ε.Κ.Π.Α.

.....
Δ. Βυτινιώτης
Ερευνητής Microsoft Research

Αθήνα, Ιούλιος 2014

Γεώργιος Φουρτούνης
Διδάκτωρ Πληροφορικής Ε.Μ.Π.

Περίληψη

Οι γλώσσες προγραμματισμού με μη αυστηρή σημασιολογία χρησιμοποιούνται για τη συγγραφή προγραμμάτων στα οποία μια έκφραση μπορεί να αντιστοιχίζεται σε ένα όνομα, χωρίς όμως να υπολογίζεται στο σημείο της δέσμευσης. Γλώσσες αυτού του τύπου χρησιμοποιούνται αρκετά σε ερευνητικά θέματα αλλά και σε πρακτικές εφαρμογές γιατί επιτρέπουν τη συγγραφή γρήγορων και κομψών προγραμμάτων, όπου οι υπολογισμοί ορίζονται όπου έχει νόημα αλλά εκτελούνται μόνο όταν χρειάζονται.

Αυτή η διδακτορική διατριβή εξετάζει τον νοηματικό μετασχηματισμό (intensional transformation) ως εναλλακτική τεχνική υλοποίησης μη αυστηρών συναρτησιακών γλωσσών, καταλήγοντας στα εξής αποτελέσματα:

- Περιγράφεται ο *γενικευμένος νοηματικός μετασχηματισμός*, το βασικό νέο θεωρητικό αποτέλεσμα που αποτελεί γενίκευση του κλασικού νοηματικού μετασχηματισμού και, σε συνδυασμό με τον μετασχηματισμό απαλοιφής συναρτήσεων (defunctionalization), επιτυγχάνει να μετασχηματίσει προγράμματα υψηλότερης τάξης με αυθαίρετες δομές δεδομένων σε προγράμματα ροής δεδομένων (dataflow) μηδενικής τάξης.
- Αποδεικνύεται η εκφραστική ισοδυναμία των δύο κλασικών εκδοχών του νοηματικού μετασχηματισμού (πρώτης τάξης και υψηλότερης τάξης) χρησιμοποιώντας τον μετασχηματισμό defunctionalization.
- Δίνεται μια αποδοτική υλοποίηση του γενικευμένου νοηματικού μετασχηματισμού με τη μορφή ενός μεταγλωττιστή για τη γλώσσα Haskell. Η υλοποίηση είναι κατάλληλη για δημοφιλείς αρχιτεκτονικές υλικού και μπορεί να συγκριθεί σε ταχύτητα με άλλους διαθέσιμους μεταγλωττιστές της Haskell.
- Περιγράφεται μια νέα, οικονομική σε μνήμη, κωδικοποίηση των δομών του χρόνου εκτέλεσης, για την αρχιτεκτονική υλικού AMD64. Αυτή η κωδικοποίηση βελτιώνει την ταχύτητα και τη χρήση κρυφής μνήμης των προγραμμάτων που παράγονται, με αποτέλεσμα η υλοποίηση να είναι συγκρίσιμη σε ταχύτητα με τον μεταγλωττιστή GHC, ο οποίος αποτελεί την ντε φάκτο υλοποίηση της Haskell.
- Ως σήμερα, ο μετασχηματισμός defunctionalization και ο νοηματικός μετασχηματισμός έχουν περιγραφεί ως μετασχηματισμοί χωρίς δυνατότητα ξεχωριστής μεταγλώττισης. Στη διατριβή αυτή αποδεικνύεται πώς αυτό το χαρακτηριστικό μπορεί να προστεθεί και στους δύο, ώστε να μπορούν να επεξεργαστούν προγράμματα με τη μορφή μονάδων κώδικα Haskell, φτάνοντας έτσι στον *τμηματικό νοηματικό μετασχηματισμό* και στον *τμηματικό μετασχηματισμό defunctionalization*.

Λέξεις κλειδιά

Αρχιτεκτονικές ροής δεδομένων, μετασχηματισμός defunctionalization, νοηματικός μετασχηματισμός, νοηματικός προγραμματισμός, οκνηρή αποτίμηση, συναρτησιακός προγραμματισμός.

Abstract

Non-strict programming languages are used to write programs where an expression may be bound to a name, but it is not necessarily evaluated on the spot. Such languages have been successful in both research and applications, since they allow programmers to write efficient and elegant programs where computations are declared where it makes sense but only run when needed.

This thesis describes the intensional transformation, a technique that translates higher-order non-strict programs to dataflow programs that can still be implemented using familiar lazy evaluation techniques on mainstream computers. The main contributions of this work are:

- We present the *generalized intensional transformation*, an extension of the classic intensional transformation that can handle languages with user-defined data types. Having data types, we then use the defunctionalization transformation to add support for higher-order functions in the intensional transformation, addressing the other open problem of the classic intensional transformation, that of supporting closures.
- We prove that the two flavors of the classic intensional transformation are equally expressive, using defunctionalization.
- We demonstrate that the intensional transformation is an efficient implementation technique on mainstream hardware by building a compiler that is competitive with other compilers of the Haskell programming language.
- We present a compact representation of program runtime structures that takes advantage of the AMD64 hardware architecture and results in fast and memory-efficient programs.
- We show how to combine separate compilation with defunctionalization and the intensional transformation, two transformations so far considered whole-program. In particular, we show how Haskell-style modules can be separately compiled and then linked in variants of both transformations.

Key words

Dataflow architectures, defunctionalization, functional programming, intensional transformation, intensional programming, lazy evaluation.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τα μέλη του Εργαστηρίου Τεχνολογίας Λογισμικού, για την παρέα, τη δουλειά μαζί και τα flame wars, αυτά τα χρόνια: τον Δημήτρη Βεκρή, τον Άγγελο Γιάντσιο, τον Παναγιώτη Θεοφιλόπουλο, τον Γιώργο Κορφιάτη, τον Λάμπρο Μουσελίμη, τον Χάρη Νάκο, τον Μιχάλη Παπακυριάκου, τον Παναγιώτη Προκοπίου, την Κατερίνα Ρουκουνάκη και τον Γιάννη Τσιούρη. Θα ήθελα επίσης να ευχαριστήσω τον Νίκο Παπασπύρου, για την άψογη συνεργασία και επίβλεψη της διατριβής, και τον Πάνο Ροντογιάννη, για την ενθάρρυνση και τα χρήσιμα σχόλιά του.

Αυτή η διδακτορική έρευνα έχει συγχρηματοδοτηθεί από την Ευρωπαϊκή Ένωση (European Social Fund – ESF) και από ελληνικά κονδύλια μέσω του επιχειρησιακού προγράμματος "Εκπαίδευση και Δια Βίου Μάθηση" του Εθνικού Στρατηγικού Πλαισίου Αναφοράς (ΕΣΠΑ), με το πρόγραμμα χρηματοδότησης ΘΑΛΗΣ (αριθμός έργου: MIS 380153).

Στο αρχικό μέρος αυτής της έρευνας, βοήθησε η υποτροφία *EEA FM EL0086 NTUA Mobility and Scholarship Program*, που μου έδωσε την ευκαιρία να επισκεφτώ την ομάδα Precise Modelling and Analysis (PMA) στο Πανεπιστήμιο του Όσλο. Ευχαριστώ τον Peter Ölveczky και τον Olaf Owe, καθώς και τον Lucian Bentea, τον Vegard Nossum, τον Volker Stolz και την Ka I Violet Pun, για τη φιλοξενία και τη συνεργασία.

Ευχαριστώ, επίσης, την ομάδα SEMI στο Πανεπιστήμιο του Μονς, για την παρέα στο low-level hacking και τις pizza Tuesdays: τον Ricardo Chessini Bose, τον Xin Chang, τον Fernando Escobar, τον Naim Harb, τον Laurent Jojczyk, τον Papy Ndungidi, τον Paulo Da Cunha Possa και τον Carlos Valderrama.

Τέλος, το μεγαλύτερο ευχαριστώ πάει στη Μαίρη, τη Νάσια και την Αγγελική· χωρίς τη βοήθειά τους, αυτή η διατριβή θα ήταν πολύ πιο δύσκολη.

Γεώργιος Φουρτούνης,

Αθήνα, 3η Ιουλίου 2014

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-2-14-gr, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2014.

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

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

Αυτή η διδακτορική διατριβή περιγράφει τη σχέση μεταξύ των συναρτησιακών γλωσσών προγραμματισμού με μη αυστηρή σημασιολογία (non-strict semantics) και των νοηματικών γλωσσών προγραμματισμού. Το θεωρητικό εργαλείο που χρησιμοποιείται για να εξεταστεί αυτή η σχέση είναι ο νοηματικός μετασχηματισμός.

Σε αυτό το κεφάλαιο γίνεται μια εισαγωγή σε βασικές έννοιες αυτών των δύο διαφορετικών οικογενειών που θα μελετηθούν με τη βοήθεια του μετασχηματισμού: των μη αυστηρών γλωσσών προγραμματισμού (Ενότητα 1.1 και των νοηματικών γλωσσών προγραμματισμού (Ενότητα 1.2). Θα κλείσουμε παρουσιάζοντας το στόχο αυτής της διατριβής, που είναι να συσχετιστούν αυτά τα δύο πεδία, και θα περιγράψουμε τη δομή που θα ακολουθήσουμε στα επόμενα κεφάλαια (Ενότητα 1.3).

1.1 Μη αυστηρές συναρτησιακές γλώσσες προγραμματισμού

Οι περισσότερες γλώσσες προγραμματισμού που χρησιμοποιούνται στην πράξη έχουν αυστηρή σημασιολογία (strict semantics): όταν σε μια μεταβλητή αντιστοιχίζεται κάποια έκφραση, η έκφραση αυτή υπολογίζεται επιτόπου και το αποτέλεσμα τοποθετείται στη μεταβλητή. Αν όμως η μεταβλητή αυτή τελικά δε χρησιμοποιηθεί στο υπόλοιπο του προγράμματος, ο υπολογισμός δε χρειαζόταν και μπορεί να θεωρηθεί επιζήμιος (αν είχε κόστος σε χρόνο ή άλλους πόρους). Επιπλέον, αν ο υπολογισμός της έκφρασης δεν τερμάτισε ποτέ ή αν προέκυψε κάποιο σφάλμα, τότε θα έχει δημιουργηθεί πρόβλημα στο πρόγραμμα που εκτελείται χωρίς λόγο.

Η μη αυστηρότητα (non-strictness) αντιμετωπίζει προβλήματα σαν αυτά που περιγράφηκαν παραπάνω: οι γλώσσες προγραμματισμού με μη αυστηρή σημασιολογία (non-strict semantics) επιτρέπουν τη συγγραφή προγραμμάτων όπου οι εκφράσεις στον κώδικα υπολογίζονται μόνο όταν χρειάζεται και μια έκφραση που έχει παρενέργειες (ή σφάλματα) επηρεάζει το πρόγραμμα μόνο αν αυτή όντως χρησιμοποιηθεί. Με αυτόν τον τρόπο, η μη αυστηρότητα επιτρέπει τον αποδοτικό χειρισμό πολύ μεγάλων ή ακόμα και άπειρων δομών δεδομένων, και αγνοεί τις άχρηστες παραμέτρους στις κλήσεις συναρτήσεων. Αυτό έχει ως αποτέλεσμα ο προγραμματιστής να μπορεί ελεύθερα να δεσμεύει εκφράσεις σε μεταβλητές, γνωρίζοντας ότι αυτές θα αποτιμηθούν μόνο όπου χρειάζονται.

Η πρώτη γλώσσα προγραμματισμού που επίσημα εξερεύνησε την επιλεκτική αποτίμηση εκφράσεων ήταν η ALGOL 60 με τις παραμέτρους *κλήσης κατ' όνομα* (*call-by-name*) [146]. Αυτές δεν υπολογίζονταν στο σημείο κλήσης μιας συνάρτησης, αλλά η έκφραση που περιείχαν αντιγραφόταν στα σημεία που αναφερόταν το όνομα της παραμέτρου, στο σώμα της συνάρτησης. Αυτό είχε ως αποτέλεσμα να υπολογίζεται η τιμή τους τόσες φορές, όσες φορές

γινόταν χρήση του ονόματος στο οποίο είχαν δεσμευτεί. Η ALGOL 60 εισήγαγε επίσης τον όρο *thunk*, για μια έκφραση που αναμένει να υπολογιστεί [129].

Η αποτίμηση κατ' όνομα είχε πρωτοεμφανιστεί πιο πριν, από τον Alonzo Church, στον λ-λογισμό του [61, 291], ο οποίος αποτέλεσε τη βασική θεωρία στην οποία στηρίχτηκε η ανάπτυξη των γλωσσών συναρτησιακού προγραμματισμού (functional programming) [27]. Αργότερα εμφανίστηκαν και άλλες στρατηγικές αποτίμησης που διατηρούσαν τη μη αυστηρή σημασιολογία, με πιο δημοφιλή την *οκνηρή αποτίμηση* (*lazy evaluation*) ή *κλήση κατ' ανάγκη* (*call-by-need*), όπου αποφεύγονται οι πολλαπλοί υπολογισμοί της ίδιας έκφρασης [101, 122, 285, 292]. Η οκνηρή αποτίμηση προτιμάται από τις περισσότερες υλοποιήσεις της δημοφιλούς γλώσσας προγραμματισμού Haskell [125], ενώ έχει επίσης προστεθεί ως επιπλέον χαρακτηριστικό σε άλλες δημοφιλείς γλώσσες όπως η Scala [198] ή οι γλώσσες που εκτελούνται στην πλατφόρμα .NET της Microsoft [173].

Σε αυτήν τη διατριβή θα ασχοληθούμε με τη Haskell ως κατεξοχήν εκπρόσωπο των μη αυστηρών γλωσσών προγραμματισμού. Η Haskell είναι μια αρκετά επιτυχημένη γλώσσα: αποτελεί βασική πλατφόρμα τις τελευταίες δεκαετίες για ερευνητικά θέματα γλωσσών προγραμματισμού, αλλά ταυτόχρονα χρησιμοποιείται όλο και περισσότερο στην πράξη για προσωπικά ή επαγγελματικά έργα λογισμικού, μιας και έχει μια πολύ αποδοτική υλοποίηση και διατίθενται πολλές υποστηρικτικές βιβλιοθήκες λογισμικού. Η Haskell είναι μια αγνή (pure) γλώσσα προγραμματισμού, τα προγράμματα της οποίας γράφονται σε συναρτησιακό στυλ και οι παρενέργειες (όπως η είσοδος και η έξοδος) ελέγχονται από δομές της ίδιας της γλώσσας.

Στη συνέχεια αυτής της εισαγωγής θα δούμε πώς η μη αυστηρότητα μπορεί να χρησιμοποιηθεί στην πράξη. Έστω το παρακάτω παράδειγμα κώδικα σε Haskell, που αθροίζει τα στοιχεία μιας λίστας αριθμών:

```
result     = sumList [1, 5, 4, 2, 30]
sumList ls = case ls of
               []       → 0
               (x : xs) → x + (sumList xs)
```

Στον παραπάνω κώδικα ορίζεται η μεταβλητή `result`, η οποία θα περιέχει και το αποτέλεσμα του προγράμματος.[1] Ορίζεται επίσης η συνάρτηση `sumList`, η οποία παίρνει μια λίστα `ls` και την εξετάζει: αν είναι η κενή λίστα `[]` επιστρέφει 0, ενώ αν είναι η λίστα με αρχικό στοιχείο x και πιθανόν και υπόλοιπα στοιχεία `xs`, προσθέτει την τιμή του x στο άθροισμα των υπόλοιπων στοιχείων της λίστας. Η αποτίμηση της `result` θα δώσει `42`. Λέμε ότι η συνάρτηση `sumList` ορίζεται αναδρομικά (*recursively*) γιατί καλεί τον εαυτό της με παράμετρο κάποια υποέκφραση που έχει λάβει η ίδια. Αυτός ο τρόπος προγραμματισμού χρησιμοποιείται συχνά στον συναρτησιακό προγραμματισμό για να περιγράψει επαναλαμβανόμενους υπολογισμούς.

Θα ορίσουμε τώρα την εξής συνάρτηση:

```
theNumbersFrom x = x : (theNumbersFrom (x+1))
```

Αυτή η συνάρτηση παίρνει έναν αριθμό και δημιουργεί μια λίστα όπου το πρώτο στοιχείο είναι αυτός ο αριθμός και τα επόμενα είναι η συνάρτηση εφαρμοσμένη στον επόμενο αριθμό. Αν προσπαθήσουμε να υπολογίσουμε την έκφραση `(theNumbersFrom 0)`, θα προκύψει η άπειρη λίστα που ακολουθεί:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,...
```

[1] Θα κάνουμε αυτήν την παραδοχή και στη συνέχεια του κειμένου, για να απλοποιηθούν τα παραδείγματά μας.

Η παραπάνω λίστα είναι άπειρη σε μέγεθος και δεν μπορούμε να τη χρησιμοποιήσουμε απευθείας. Μπορούμε όμως να ορίσουμε μια άλλη συνάρτηση, η οποία να επιστρέφει μέρος αυτής της άπειρης λίστας:

```
takeFirst i (x : xs) =
  if i==0 then [] else x : (takeFirst (i−1) xs)
```

Η έκφραση `(takeFirst 12 (theNumbersFrom 0))`, αν υπολογιστεί, θα επιστρέψει μόνο τα 12 πρώτα στοιχεία της λίστας:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Αυτό συνέβη γιατί η λίστα παράχθηκε από την `theNumbersFrom` και καταναλώθηκε από την `takeFirst`, στοιχείο-στοιχείο, σταματώντας στο 12ο στοιχείο, επειδή μόνο αυτά τα στοιχεία χρειάζονται για το αποτέλεσμα της `takeFirst`.

Ακολουθεί το πλήρες πρόγραμμα που αθροίζει τα πρώτα 12 στοιχεία της άπειρης λίστας αριθμών:

```
result               = sumList (takeFirst 12 (theNumbersFrom 0))
sumList ls           = case ls of
                           []       → 0
                           (x : xs) → x + (sumList xs)
theNumbersFrom x     = x : (theNumbersFrom (x+1))
takeFirst i (x : xs) = if i==0 then
                           []
                       else
                         x : (takeFirst (i−1) xs)
```

Ένα άλλο χαρακτηριστικό της Haskell είναι ότι η γλώσσα αντιμετωπίζει τις συναρτήσεις ως *τιμές πρώτης τάξης* (*first-class values*): μια συνάρτηση μπορεί να πάρει μια άλλη συνάρτηση ως παράμετρο, ή να επιστρέψει συνάρτηση. Έστω για παράδειγμα το εξής πρόγραμμα:

```
result   = map inc [1, 14, 31]
inc x    = x + 1
map f ls = case ls of
             []       → []
             (x : xs) → (f x) : (map f xs)
```

Εδώ η συνάρτηση `map` παίρνει μια συνάρτηση `f` και την εφαρμόζει στο πρώτο στοιχείο μιας λίστας `ls` (και αναδρομικά και στα υπόλοιπα στοιχεία της λίστας). Ο υπολογισμός της `result` θα εφαρμόσει επομένως την `inc` σε όλα τα στοιχεία της λίστας `[1, 14, 31]`, δίνοντας σαν αποτέλεσμα `[2, 15, 32]`. Η Haskell επιτρέπει επίσης τον σχηματισμό συναρτήσεων με *μερική εφαρμογή* (*partial application*), όπως στην εφαρμογή της `add` που φαίνεται παρακάτω:

```
result   = map (add 1) [1, 14, 31]
add x y  = x + y
map f ls = case ls of
             []       → []
             (x : xs) → (f x) : (map f xs)
```

Ο παραπάνω κώδικας εφαρμόζει την `add` (που παίρνει δύο παραμέτρους) στην παράμετρο `1`, με αποτέλεσμα να προκύπτει μια νέα συνάρτηση που παίρνει μια παράμετρο `y`, στην οποία προσθέτει `1`. Η νέα αυτή συνάρτηση μπορεί να περαστεί στην `map` ως συνήθως.

Εκφράσεις που είναι ονόματα συναρτήσεων ή μερικές εφαρμογές αυτών ονομάζονται *εκφράσεις υψηλότερης τάξης* (*higher-order expressions*) και οι γλώσσες που τις υποστηρίζουν

αποκαλούνται *γλώσσες προγραμματισμού υψηλότερης τάξης* (*higher-order programming languages*). Αντίθετα, γλώσσες που δεν υποστηρίζουν αυτού του τύπου τις εκφράσεις ονομάζονται *γλώσσες προγραμματισμού πρώτης τάξης* (*first-order programming languages*).

1.2 Νοηματικές γλώσσες προγραμματισμού

Ο *νοηματικός προγραμματισμός* (*intensional programming*) είναι ένα προγραμματιστικό παράδειγμα, στο οποίο τα προγράμματα περιέχουν τιμές που εξαρτώνται έμμεσα από συμφραζόμενα [286, 288]. Βασίζεται στη νοηματική λογική (intensional logic) [181], έναν φορμαλισμό που δημιούργησε ο γλωσσολόγος Richard Montague για να μελετήσει τη σημασιολογία της εξάρτησης από συμφραζόμενα στις φυσικές γλώσσες (όπως οι χρονικές προτάσεις).

Η πρώτη γλώσσα προγραμματισμού που ενσωμάτωσε ιδέες της νοηματικής λογικής ήταν η Lucid [23], στόχος της οποίας ήταν η εξαγωγή συμπερασμάτων για ροές τιμών, οι οποίες εξαρτώνταν από γραμμικό χρόνο [22]. Η Lucid εξερεύνησε πολλές ιδέες σχετικές με τον προγραμματισμό με εξάρτηση από τα συμφραζόμενα (context-dependent programming) και με τον προγραμματισμό με ροές (stream programming). Αν και νοηματική γλώσσα, η Lucid θεωρήθηκε επίσης *γλώσσα προγραμματισμού ροής δεδομένων* (*dataflow programming language*) [289], σε μια περίοδο που η κατασκευή και ο προγραμματισμός των υπολογιστών ροής δεδομένων (dataflow hardware) [18, 116] ήταν ενεργό ερευνητικό πεδίο, και ο νοηματικός προγραμματισμός αναπτύχθηκε ως επέκταση του προγραμματισμού ροής δεδομένων [21].

Στη Lucid ο προγραμματιστής χρησιμοποιεί τιμές που εξαρτώνται από τα συμφραζόμενα και δίνονται ειδικοί νοηματικοί τελεστές που χειρίζονται αυτά τα συμφραζόμενα. Για παράδειγμα, η ακολουθία των αριθμών Fibonacci μπορεί να δοθεί από τον εξής κώδικα σε Lucid:

```
fib = 0 fby (1 fby fib + next fib)
```

Εδώ η `fib` δεν είναι συνάρτηση, αλλά μία "ιστορία" (ροή) τιμών. Επίσης, εκφράσεις όπως η `0` και η `1` δεν είναι απλά αριθμοί αλλά οι σταθερές ροές `0, 0, ...` και `1, 1, ...` Τα έμμεσα συμφραζόμενα εδώ είναι ο διακριτός γραμμικός χρόνος, στις στιγμές του οποίου ορίζονται οι ροές των τιμών. Οι αριθμητικοί τελεστές εφαρμόζονται επίσης πάνω σε ροές: ο τελεστής `+` παίρνει δύο ροές και παράγει μια νέα ροή, η τιμή της οποίας σε κάθε χρονική στιγμή είναι ίση με το άθροισμα των τιμών των δύο ροών τη στιγμή αυτή. Τα συμφραζόμενα μπορούν να αλλάξουν, με τους δύο ειδικούς τελεστές `fby` και `next`. Ο τελεστής `fby` παίρνει δύο ροές και παράγει μια νέα ροή, όπου το πρώτο στοιχείο είναι το πρώτο στοιχείο της πρώτης ροής και ακολουθούν τα στοιχεία της δεύτερης ροής. Ο τελεστής `next` ζητάει την τιμή μιας ροής την επόμενη χρονική στιγμή.

Στην πράξη, το παράδειγμα μπορεί να διαβαστεί ως εξής: «το `fib` είναι αρχικά μηδέν, μετά ένα, και μετά το άθροισμα της τρέχουσας τιμής του και της επόμενης τιμής του», που είναι ένας καθαρά δηλωτικός τρόπος να περιγραφούν οι τιμές της ακολουθίας Fibonacci. Ο υπολογισμός του αριθμού Fibonacci στη θέση i της ακολουθίας, γίνεται ζητώντας την τιμή του `fib` τη στιγμή i. Η υλοποίηση της γλώσσας τότε θα τον υπολογίσει, υπολογίζοντας και τις τιμές που χρειάζονται σε άλλα συμφραζόμενα (σε αυτό το παράδειγμα, τις τιμές του `fib` σε προηγούμενες χρονικές στιγμές). Αυτό το μοντέλο υπολογισμού που καθοδηγείται από τη ζήτηση τιμών (demand-driven computation model) [220] ονομάζεται *eduction* [21] και μοιάζει με τη μη αυστηρή αποτίμηση που είδαμε στην προηγούμενη ενότητα: για να υπολογιστεί ένα αποτέλεσμα υπολογίζονται μόνο όσα τμήματα μιας ροής χρειάζεται.

Σημαντικό χαρακτηριστικό των νοηματικών γλωσσών είναι ότι οι επαναλαμβανόμενοι υπολογισμοί (όπως οι διαφορετικές τιμές της `fib` στο παράδειγμα) εκφράζονται με την επα-

νάληψη (iteration) πάνω σε ακολουθίες τιμών. Αυτό έρχεται σε αντιδιαστολή με την αναδρομή που χρησιμοποιούν οι συναρτησιακές γλώσσες προγραμματισμού, όπως είδαμε στην Ενότητα 1.1.

Η Lucid ακολουθήθηκε από πολλές παρόμοιες γλώσσες και εργαλεία. Κάποιες παραλλαγές της χρησιμοποιούσαν παραπάνω από μία διαστάσεις για να έχουν πολλαπλά συμφραζόμενα [132], ή επέτρεπαν στα συμφραζόμενα να είναι και αυτά τιμές πρώτης τάξης μέσα στη γλώσσα [149, 293].

Μια ιδιαίτερα επιτυχημένη παραλλαγή της Lucid είναι η Lustre [52] (και οι ερευνητικές παραλλαγές της, Lucid-Synchrone [53, 228] και Zélus [42]), μια γλώσσα πραγματικού χρόνου (real-time) για την ανάπτυξη συστημάτων αυτοματισμού, η οποία ελέγχει αν τα προγράμματά της μπορούν να εκτελεστούν σε φραγμένο χώρο μνήμης ή αν οι υπολογισμοί τους μπορούν να εκτελούνται πάντα μέσα σε συγκεκριμένα χρονικά όρια.[2] Η Lustre έχει αρκετές βιομηχανικές εφαρμογές και έχει χρησιμοποιηθεί για τον προγραμματισμό ασφαλούς λογισμικού ελέγχου για αεροσκάφη, τρένα, και εργοστάσια παραγωγής ενέργειας [84].

Η TransLucid [226] είναι μια άλλη σύγχρονη γλώσσα προγραμματισμού που βασίστηκε σε νοηματικές ιδέες και πολλαπλές διαστάσεις. Η γλώσσα γενικεύει τις διαστάσεις ώστε αυτές να μπορούν να είναι οποιεσδήποτε ατομικές τιμές [222] ενώ διαθέτει και παράλληλη υλοποίηση με πολλαπλά νήματα [229]. Πρόσφατα προστέθηκε άλλο ένα μέλος στην οικογένεια των επηρεασμένων από τη Lucid γλωσσών: η Forensic Lucid, η οποία στοχεύει σε εφαρμογές ηλεκτρονικής εγκληματολογίας (cyberforensics) [177, 178] και βασίζεται σε μια κατανεμημένη υλοποίηση [118].

Οι ιδέες του νοηματικού προγραμματισμού διαδόθηκαν επίσης και σε άλλα πεδία, όπως τα λογιστικά φύλλα [80], οι βάσεις δεδομένων [269], τα συστήματα πρακτόρων [8], τα συστήματα αρχείων [81], η διαχείριση εκδόσεων [149], το υπερκείμενο [224], ο χρονικός λογικός προγραμματισμός [201, 239] και ο διαδικαστικός προγραμματισμός [208, 237]. Επίσης, γίνεται προσπάθεια να δημιουργηθεί ένα πρότυπο που να συσχετίζει με τυπικό τρόπο τα διαφορετικά μέλη της οικογένειας των γλωσσών που κατάγονται από τη Lucid [210].

1.3 Σχέση μεταξύ των δύο πεδίων

Στις προηγούμενες δύο ενότητες περιγράφηκαν δύο στυλ προγραμματισμού, τα οποία επιτρέπουν τη συγγραφή μη αυστηρών προγραμμάτων. Ο συναρτησιακός προγραμματισμός βασίζεται στις *συναρτήσεις* και περιγράφει τους επαναλαμβανόμενους υπολογισμούς χρησιμοποιώντας *αναδρομή*, ενώ ο νοηματικός προγραμματισμός βασίζεται σε *ροές τιμών* και εκφράζει τους επαναλαμβανόμενους υπολογισμούς χρησιμοποιώντας *ακολουθίες τιμών* και *συμφραζόμενα*.

Βασική θέση αυτής της διατριβής είναι ότι αυτοί οι δύο διαφορετικοί τρόποι προγραμματισμού μπορούν να συσχετιστούν και ότι συναρτησιακά προγράμματα μπορούν να μετασχηματιστούν σε ισοδύναμα νοηματικά προγράμματα. Η υπόλοιπη διατριβή δομείται ως εξής:

- Στο Κεφάλαιο 2 που ακολουθεί, θα περιγράψουμε τους δύο μετασχηματισμούς προγραμμάτων που πρόκειται να χρησιμοποιήσουμε: τον νοηματικό μετασχηματισμό και τον μετασχηματισμό defunctionalization.

[2] Οι γλώσσες που ανήκουν στην οικογένεια της Lustre αποκαλούνται *σύγχρονες γλώσσες ροής δεδομένων* (*synchronous data-flow languages*) [51].

- Στο Κεφάλαιο 3 θα παρουσιάσουμε τον γενικευμένο νοηματικό μετασχηματισμό, που μπορεί να μετατρέψει μη αυστηρά συναρτησιακά προγράμματα με τύπους δεδομένων σε ισοδύναμα νοηματικά προγράμματα.

- Στο Κεφάλαιο 4 περιγράφουμε πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί με οκνηρό τρόπο χρησιμοποιώντας κλασικές ιδέες των νοηματικών υλοποιήσεων. Ως αποτέλεσμα, δίνουμε έναν διερμηνέα που υποστηρίζει οκνηρή αποτίμηση και περιγράφεται η σχέση αυτής της τεχνικής με υλοποιήσεις γλωσσών ροής δεδομένων και νοηματικού προγραμματισμού.

- Στο Κεφάλαιο 5 δείχνουμε ότι ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί αποδοτικά στις δημοφιλείς αρχιτεκτονικές υπολογιστών IA-32 και AMD64, χρησιμοποιώντας μια παραλλαγή της τεχνικής υλοποίησης γλωσσών με εγγραφές δραστηριοποίησης.

- Στο Κεφάλαιο 6 περιγράφουμε μια οικονομική αναπαράσταση της μνήμης κατά τον χρόνο εκτέλεσης, για την αρχιτεκτονική υλικού AMD64. Η τεχνική αυτή βελτιώνει την ταχύτητα και τη συμπεριφορά κρυφής μνήμης των προγραμμάτων που παράγονται.

- Στο Κεφάλαιο 7 περιγράφουμε πώς η υλοποίηση του γενικευμένου νοηματικού μετασχηματισμού μπορεί να υποστηρίξει τμηματική μεταγλώττιση και σύνδεση μονάδων κώδικα Haskell. Αυτό επιτυγχάνεται τροποποιώντας τον γενικευμένο νοηματικό μετασχηματισμό και τον μετασχηματισμό defunctionalization, ώστε να μπορούν να εφαρμοστούν σε μεμονωμένα τμήματα κώδικα, τα οποία στη συνέχεια μπορούν να συνδεθούν για να προκύψει το τελικό εκτελέσιμο πρόγραμμα.

- Κλείνουμε με το Κεφάλαιο 8, αναφέροντας κατευθύνσεις για περαιτέρω έρευνα, με αφετηρία τα αποτελέσματα αυτής της διατριβής.

Κεφάλαιο 2

Οι δύο μετασχηματισμοί προγραμμάτων

Στο προηγούμενο κεφάλαιο έγινε μια εισαγωγή στον συναρτησιακό και στον νοηματικό προγραμματισμό, δύο προγραμματιστικά παραδείγματα που ενθαρρύνουν διαφορετικούς τρόπους συγγραφής προγραμμάτων. Εκ πρώτης όψεως δείχνουν δύσκολο να συσχετιστούν: το πρώτο βασίζεται στις συναρτήσεις (χρησιμοποιώντας ακόμα και μερική εφαρμογή αυτών) και στην αναδρομή, ενώ το δεύτερο βασίζεται στις ροές τιμών και στις ακολουθίες υπολογισμών.

Αυτό το κεφάλαιο παρουσιάζει το θεωρητικό υπόβαθρο που θα χρησιμοποιηθεί για να συσχετιστούν τα δύο προγραμματιστικά παραδείγματα. Αρχικά θα παρουσιάσουμε τον κλασικό νοηματικό μετασχηματισμό, μια τεχνική που μπορεί να μετασχηματίσει κάποια συναρτησιακά προγράμματα σε νοηματικά (Ενότητα 2.1). Στη συνέχεια θα παρουσιάσουμε τον μετασχηματισμό defunctionalization, μια τεχνική που αντικαθιστά τις εκφράσεις που είναι μερικές εφαρμογές συναρτήσεων, με δομές δεδομένων (Ενότητα 2.2). Τέλος, θα εξετάσουμε τον συνδυασμό τους (Ενότητα 2.3): θα δείξουμε ότι οι δύο κλασικές εκδοχές του νοηματικού μετασχηματισμού έχουν ίδια εκφραστική δύναμη (Ενότητα 2.3.1) και θα περιγράψουμε τι λείπει για να μπορέσουμε να μετασχηματίσουμε συναρτησιακά προγράμματα σε νοηματικά στη γενική περίπτωση (Ενότητα 2.3.2).

2.1 Ο νοηματικός μετασχηματισμός

Ο νοηματικός μετασχηματισμός (intensional transformation) μετατρέπει συναρτησιακά προγράμματα σε νοηματικά προγράμματα. Σαν τεχνική, πρωτοχρησιμοποιήθηκε από τον Calvin Ostrum στο Πανεπιστήμιο του Waterloo, για την υλοποίηση συναρτησιακών γλωσσών πρώτης τάξης με βάση το μοντέλο υπολογισμού eduction, το οποίο είχε χρησιμοποιηθεί και στην υλοποίηση της ίδιας της Lucid [48, 244]. Ο Ali Yaghi στη συνέχεια περιέγραψε την τεχνική με τυπικό τρόπο χρησιμοποιώντας νοηματική λογική στη διδακτορική του διατριβή [302] και ο Πάνος Ροντογιάννης την επέκτεινε και απέδειξε την ορθότητά της [244] ως μέρος της δικής του διδακτορικής διατριβής [236].

Σε αυτήν την ενότητα θα περιγράψουμε συνοπτικά τον νοηματικό μετασχηματισμό για γλώσσες πρώτης τάξης (Ενότητα 2.1.1)· περισσότερες λεπτομέρειες περιλαμβάνονται στις αντίστοιχες δημοσιεύσεις [244, 302]. Στη συνέχεια θα αναφερθούμε στον νοηματικό μετασχηματισμό υψηλότερης τάξης (Ενότητα 2.1.2) και στις εφαρμογές του μετασχηματισμού (Ενότητα 2.1.3).

2.1.1 Ο μετασχηματισμός πρώτης τάξης

Η είσοδος στον κλασικό μετασχηματισμό πρώτης τάξης [244, 302] είναι ένα συναρτησιακό πρόγραμμα πρώτης τάξης που χρησιμοποιεί μόνο βασικούς τύπους δεδομένων (όπως

είναι οι ακέραιοι ή οι τιμές αλήθειας). Το πρόγραμμα τότε με απλά βήματα μετασχηματίζεται σε ένα νοηματικό πρόγραμμα μηδενικής τάξης που περιλαμβάνει μόνο δηλώσεις μεταβλητών που δεν παίρνουν παραμέτρους (nullary). Το πρόγραμμα που προκύπτει είναι όντως νοηματικό γιατί περιέχει δύο ειδικούς τελεστές που ενεργούν πάνω στα συμφραζόμενα του προγράμματος και σε λίγο θα δούμε αναλυτικά τη σημασιολογία τους. Ο μετασχηματισμός μπορεί με απλό τρόπο να περιγραφεί ως εξής [244]:

1. Έστω ότι η `f` είναι συνάρτηση που ορίζεται στο αρχικό συναρτησιακό πρόγραμμα. Απαριθμούνται στο κείμενο του προγράμματος οι κλήσεις της `f`, αρχίζοντας από το 0 (οι κλήσεις στο σώμα της ίδιας της `f` συμπεριλαμβάνονται).

2. Η κλήση i της `f` στο πρόγραμμα αντικαθίσταται από την έκφραση `call_i(f)`. Οι τυπικές παράμετροι στον ορισμό της `f` διαγράφονται, ώστε η `f` να ορίζεται πια σαν απλή μεταβλητή.

3. Για κάθε τυπική παράμετρο της `f` δημιουργείται ένας νέος ορισμός. Στο δεξιό μέλος του βρίσκεται ο τελεστής `actuals` που εφαρμόζεται στη λίστα όλων των πραγματικών παραμέτρων που περνιούνται στη θέση αυτής της τυπικής παραμέτρου, με την ίδια σειρά που έγινε η αρίθμηση των κλήσεων.

Ο αλγόριθμος μπορεί να φανεί καλύτερα με το παρακάτω παράδειγμα. Έστω το εξής συναρτησιακό πρόγραμμα πρώτης τάξης:

```
result = f 3 + f 5
f x    = g (x − 1)
g y    = y + 2
```

Ο μετασχηματισμός παράγει το εξής τελικό πρόγραμμα:

```
result = call_0(f) + call_1(f)
f      = call_0(g)
g      = y + 2
x      = actuals(3, 5)
y      = actuals(x − 1)
```

Ο παραπάνω νοηματικός κώδικας μπορεί να αποτιμηθεί χρησιμοποιώντας αρχικά κενά *συμφραζόμενα* (*context*), τα οποία στην πραγματικότητα αναπαριστώνται από μια λίστα αριθμών. Η λίστα αυτή μπορεί να θεωρηθεί σαν μια δομή που κρατάει πληροφορία για το πού βρίσκεται η εκτέλεση στο δέντρο των αναδρομικών κλήσεων των συναρτήσεων. Οι τελεστές `call_i` και `actuals` αλλάζουν τα συμφραζόμενα (context-switching operators). Ο τελεστής `call_i` προσθέτει στην αρχή μιας λίστας w τον αριθμό i. Αντίθετα, ο τελεστής `actuals` επιλέγει το στοιχείο i από τις παραμέτρους του, όπου i είναι το πρώτο στοιχείο των συμφραζομένων λίστας, και αφαιρεί το i από τη λίστα των συμφραζομένων.

Μπορούμε τώρα να ορίσουμε μια συνάρτηση *EVAL* που να υλοποιεί το νοηματικό πρόγραμμα που προκύπτει από τον μετασχηματισμό, όπως φαίνεται στην Εικόνα 2.1. Η συνάρτηση παίρνει επίσης σαν παράμετρο το (αμετάβλητο) πρόγραμμα P που αποτιμάται, το οποίο συχνά θα παραλείπεται για λόγους απλότητας. Η συνάρτηση *lookup*(v, P) επιστρέφει την έκφραση που αντιστοιχεί στη μεταβλητή v, όπως ορίζεται στο πρόγραμμα P. Η αποτίμηση των κλασικών δομών των συναρτησιακών γλωσσών (όπως η έκφραση if-then-else και οι αριθμητικές εκφράσεις) περιγράφεται από τον κανόνα για τις σταθερές c που παίρνουν n παραμέτρους (όπου, αν $n = 0$, καλύπτεται και η περίπτωση των σταθερών μηδενικής τάξης, όπως οι αριθμοί, οι χαρακτήρες, κλπ.). Η εκτέλεση του νοηματικού προγράμματος του

παραδείγματός μας εμφανίζεται στην Εικόνα 2.2. Θεωρούμε ότι κάθε πρόγραμμα έχει μια μεταβλητή `result` που αντιστοιχεί στην τιμή του προγράμματος που θέλουμε να υπολογίσουμε.

$$
\begin{array}{lcl}
\mathit{EVAL}_P(v, w) & = & \mathit{EVAL}_P(\mathit{lookup}(v, P), w) \\
\mathit{EVAL}_P(\texttt{call\_i}(E), w) & = & \mathit{EVAL}_P(E, i : w) \\
\mathit{EVAL}_P(\texttt{actuals}(E_0, \ldots, E_{n-1}), i : w) & = & \mathit{EVAL}_P(E_i, w) \\
\mathit{EVAL}_P(c(E_0, \ldots, E_{n-1}), w) & = & c(\mathit{EVAL}_P(E_0, w), \ldots, \mathit{EVAL}_P(E_{n-1}, w))
\end{array}
$$

Εικόνα 2.1: Η συνάρτηση *EVAL* της νοηματικής γλώσσας.

$$
\begin{array}{ll}
 & \mathit{EVAL}(\texttt{result}, [\,]) \\
= & \mathit{EVAL}(\texttt{call\_0(f)+ call\_1(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{call\_0(f)}, [\,]) + \mathit{EVAL}(\texttt{call\_1(f)}, [\,]) \\
= & \mathit{EVAL}(\texttt{f}, [0]) + \mathit{EVAL}(\texttt{f}, [1]) \\
= & \mathit{EVAL}(\texttt{call\_0(g)}, [0]) + \mathit{EVAL}(\texttt{call\_0(g)}, [1]) \\
= & \mathit{EVAL}(\texttt{g}, [0, 0]) + \mathit{EVAL}(\texttt{g}, [0, 1]) \\
= & \mathit{EVAL}(\texttt{y}, [0, 0]) + \mathit{EVAL}(\texttt{2}, [0, 0]) + \mathit{EVAL}(\texttt{y}, [0, 1]) + \mathit{EVAL}(\texttt{2}, [0, 1]) \\
= & \mathit{EVAL}(\texttt{actuals(x-1)}, [0, 0]) + 2 + \\
 & \mathit{EVAL}(\texttt{actuals(x-1)}, [0, 1]) + 2 \\
= & \mathit{EVAL}(\texttt{x-1}, [0]) + 2 + \mathit{EVAL}(\texttt{x-1}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{x}, [0]) - \mathit{EVAL}(\texttt{1}, [0]) + 2 + \\
 & \mathit{EVAL}(\texttt{x}, [1]) - \mathit{EVAL}(\texttt{1}, [1]) + 2 \\
= & \mathit{EVAL}(\texttt{actuals(3, 5)}, [0]) - 1 + 2 + \\
 & \mathit{EVAL}(\texttt{actuals(3, 5)}, [1]) - 1 + 2 \\
= & \mathit{EVAL}(\texttt{3}, [\,]) - 1 + 2 + \mathit{EVAL}(\texttt{5}, [\,]) - 1 + 2 \\
= & 3 - 1 + 2 + 5 - 1 + 2 \\
= & 10
\end{array}
$$

Εικόνα 2.2: Εκτέλεση του τελικού νοηματικού προγράμματος.

Το πρόγραμμα που προέκυψε είναι νοηματικό γιατί δεν περιέχει πια συναρτήσεις αλλά μόνο μεταβλητές, η τιμή των οποίων αλλάζει ανάλογα με τα συμφραζόμενα. Τα συμφραζόμενα όμως δεν είναι πια ο γραμμικός χρόνος της κλασικής Lucid αλλά *διακλαδιζόμενος χρόνος* (*branching time*): οι αριθμοί αναπαριστούν διάφορα «επόμενα» συμφραζόμενα που μπορούν να είναι προσβάσιμα από τα τρέχοντα συμφραζόμενα. Οι τιμές των μεταβλητών είναι δηλαδή γενικευμένες ροές τιμών που δεικτοδοτούνται από ένα διακλαδιζόμενο δένδρο. Μια χρονική στιγμή είναι μια λίστα αριθμών που δείχνουν τους κλάδους που ακολουθήθηκαν από την αρχική στιγμή (η οποία θεωρείται ότι είναι τα κενά συμφραζόμενα).

2.1.2 Ο μετασχηματισμός υψηλότερης τάξης

Ο μετασχηματισμός πρώτης τάξης δεν μπορεί να μετασχηματίσει συναρτησιακά προγράμματα υψηλότερης τάξης, που είναι πολύ συχνά στον συναρτησιακό προγραμματισμό. Μια σημαντική προσπάθεια να ξεπεραστεί αυτό το εμπόδιο ήταν ο νοηματικός μετασχηματισμός υψηλότερης τάξης των Ροντογιάννη και Wadge [245, 287]. Ο μετασχηματισμός αυτός δέχεται όλα τα συναρτησιακά προγράμματα υψηλότερης τάξης, στα οποία το όνομα μιας

συνάρτησης μπορεί να περαστεί σε μια άλλη συνάρτηση, όπως στο εξής παράδειγμα, δεν μπορεί όμως να εφαρμοστεί μερικώς σε λιγότερα ορίσματα:[1]

```
result = (g inc 8) + (g dec 5)
g f x  = f (x+1)
inc y  = y+1
dec a  = a−1
```

Εδώ οι εκφράσεις `inc` και `dec` είναι ονόματα συναρτήσεων που περνιούνται στην `g` μέσω της τυπικής παραμέτρου `f`. Ο μετασχηματισμός υψηλότερης τάξης παράγει τότε ένα νοηματικό πρόγραμμα που έχει δύο διαστάσεις, μια για κάθε τάξη των τυπικών παραμέτρων που υπήρχαν στο αρχικό πρόγραμμα:

```
result = call_(1,0)call_(0,0)(g) + call_(1,1)call(0,1)(g)
g      = call_(0,0)(f)
inc    = y+1
dec    = a−1
f      = case_1(actuals_(1,0)call_(0,0)(inc),
                actuals_(1,1)call_(0,0)(dec))
z      = case_0(actuals_(0,0)(x+1))
y      = case_0(actuals_(0,0)call_(1,0)(z))
a      = case_0(actuals_(0,0)call_(1,1)(z))
x      = case_0(actuals_(0,0)actuals_(1,0)(8),
                actuals_(0,1)actuals_(1,1)(5))
```

Ο νοηματικός μετασχηματισμός υψηλότερης τάξης είναι πιο πολύπλοκος από τον αντίστοιχο πρώτης τάξης: οι τελεστές `call` και `actuals` περιέχουν περισσότερη πληροφορία, για να φαίνεται ποια διάσταση χρησιμοποιείται, και υπάρχει ένας νέος νοηματικός τελεστής `case`. Δε θα περιγράψουμε εδώ τον μετασχηματισμό υψηλότερης τάξης· λεπτομερής τυπικός ορισμός και απόδειξη ορθότητας περιλαμβάνονται στο αντίστοιχο άρθρο των Ροντογιάννη και Wadge [245]. Όπως θα δούμε και παρακάτω, στην Ενότητα 2.3.1, οι δύο νοηματικοί μετασχηματισμοί (πρώτης τάξης και υψηλότερης τάξης) έχουν την ίδια εκφραστική δύναμη.

2.1.3 Εφαρμογές

Ο νοηματικός μετασχηματισμός αρχικά προτάθηκε ως ένας τρόπος για να εκτελούνται συναρτησιακά προγράμματα σε *αρχιτεκτονικές ροής δεδομένων μονάδων με ετικέτες* (*tagged-token dataflow*) [18]. Οι υπολογιστές αυτής της αρχιτεκτονικής περιείχαν υλικό που υποστήριζε τον χειρισμό τιμών μαζί με ετικέτες, τις *μονάδες με ετικέτες* (*tagged tokens*): το υλικό μπορούσε να χρησιμοποιηθεί σαν πλατφόρμα εκτέλεσης συναρτησιακών γλωσσών [277] και οι ετικέτες μπορούσαν να χρησιμοποιηθούν ως συμφραζόμενα για την υλοποίηση νοηματικών γλωσσών [244, 302]. Στη συνέχεια ο νοηματικός μετασχηματισμός αποδείχτηκε αποδοτικός και σε κλασικές αρχιτεκτονικές υπολογιστών, παράγοντας γρηγορότερα προγράμματα σε σχέση με άλλες υλοποιήσεις συναρτησιακών γλωσσών [55, 242]. Ο νοηματικός μετασχηματισμός χρησιμοποιήθηκε επίσης στον χρονικό λογικό προγραμματισμό και στις βάσεις δεδομένων [238].

Ο κλασικός νοηματικός μετασχηματισμός έχει όμως δύο σημαντικούς περιορισμούς που εμποδίζουν τη χρήση του στην πράξη:

- Δεν υποστηρίζει αυθαίρετα πολύπλοκους τύπους δεδομένων που να ορίζονται από τον χρήστη, ένα χαρακτηριστικό που χρησιμοποιείται πολύ συχνά στον συναρτησιακό προ-

[1] Το παράδειγμα προέρχεται από το αντίστοιχο άρθρο των Ροντογιάννη και Wadge [245], με μικρές αλλαγές.

γραμματισμό. Αυτό είναι γενικό πρόβλημα των νοηματικών γλωσσών, για παράδειγμα η Lucid ποτέ δεν υποστήριξε αυθαίρετους τύπους δεδομένων [289, §7.1], ενώ ακόμα και νεότερες γλώσσες όπως η TransLucid έχουν περιορισμένη ποικιλία σε τύπους [78]. Οι γλώσσες που στοχεύουν σε υλοποίηση μέσω λογισμικού υποστηρίζουν συνήθως περισσότερα χαρακτηριστικά τύπων δεδομένων· για παράδειγμα, η Objective Lucid και το σύστημα νοηματικού προγραμματισμού GIPSY μπορούν να διαβάζουν κλάσεις της Java [177], ενώ η Lucid Synchrone ενσωματώνει τύπους δεδομένων με αυστηρή σημασιολογία στο στυλ της ML [228]. Στον προγραμματισμό ροής δεδομένων, αν και ήδη από το 1974 ο Dennis είχε προτείνει τρόπους για τον χειρισμό σύνθετων δομών δεδομένων [72], η υποστήριξη αυθαίρετων τύπων δεδομένων από το υλικό ροής δεδομένων θεωρείται ακόμα ανοιχτό πρόβλημα [127].

- Ο νοηματικός μετασχηματισμός υψηλότερης τάξης δεν υποστηρίζει τη μερική εφαρμογή συναρτήσεων. Αυτό στην πράξη αποκλείει τον μετασχηματισμό μιας μεγάλης κλάσης συναρτησιακών προγραμμάτων, κάνοντας την τεχνική ακατάλληλη για ρεαλιστικές νοηματικές υλοποιήσεις συναρτησιακών γλωσσών. Αν και έχουν γίνει κάποιες προσπάθειες να επεκταθεί ο νοηματικός μετασχηματισμός σε περισσότερα συναρτησιακά προγράμματα υψηλότερης τάξης [78, 243], δεν έχει βρεθεί ικανοποιητική λύση για τη γενική περίπτωση.[2] Και αυτός ο περιορισμός φαίνεται ότι είναι συνηθισμένος στις γλώσσες βασισμένες σε ροές: οι εκφράσεις υψηλότερης τάξης δεν ταιριάζουν με τη θεώρηση όλων των τιμών του προγράμματος σαν μεταβαλλόμενες τιμές που ανήκουν σε κάποιον απλό τύπο. Παρόλα αυτά, έχει γίνει έρευνα πάνω στην υποστήριξη εκφράσεων υψηλότερης τάξης από γλώσσες ροής δεδομένων: οι Lee και Parks περιέγραψαν οπτικές γλώσσες ροής δεδομένων υψηλότερης τάξης [155], οι Colaço *et al.* εισήγαγαν κάποιες εκφράσεις υψηλότερης τάξης στη Lustre και στη Lucid-Synchrone [62], ενώ οι Uustalu και Vene έδωσαν σημασιολογία βασισμένη στα monads για ροή δεδομένων υψηλότερης τάξης [279].

Τα δύο παραπάνω προβλήματα δεν είναι ανεξάρτητα: γνωρίζουμε ότι υπάρχει ο μετασχηματισμός *defunctionalization*, ο οποίος μετατρέπει εκφράσεις υψηλότερης τάξης, όπως οι μερικές εφαρμογές συναρτήσεων, σε δομές δεδομένων. Στην επόμενη ενότητα θα περιγράψουμε τον μετασχηματισμό και θα εξετάσουμε κατά πόσο μπορεί να αντιμετωπίσει τα παραπάνω προβλήματα.

2.2 Ο μετασχηματισμός defunctionalization

Η τεχνική της *απαλοιφής συναρτήσεων*[3] (defunctionalization) μετασχηματίζει προγράμματα υψηλότερης τάξης σε ισοδύναμα προγράμματα πρώτης τάξης. Ανακαλύφθηκε από τον John Reynolds [234] και αρχικά χρησιμοποιήθηκε για την υλοποίηση διερμηνέων για γλώσσες υψηλότερης τάξης χρησιμοποιώντας γλώσσες πρώτης τάξης.

Ο μετασχηματισμός defunctionalization αναπαριστά τις εκφράσεις υψηλότερης τάξης με δομές δεδομένων μέσα στη γλώσσα πρώτης τάξης. Για παράδειγμα, έστω το εξής πρόγραμμα υψηλότερης τάξης:

```
result    = (twice inc 1) + (twice (mult 3) 2)
```

[2] Η πιο πλήρης λύση έχει παρουσιαστεί για την TransLucid και χρησιμοποιεί έναν αυθαίρετο αριθμό διαστάσεων και ειδικές δομές, αλλά δεν έχει εφαρμοστεί από όσο γνωρίζουμε για την υλοποίηση κάποιας συναρτησιακής γλώσσας προγραμματισμού [225].

[3] Στη συνέχεια θα χρησιμοποιείται ο καθιερωμένος αγγλικός όρος αντί του περιφραστικού ελληνικού.

```
twice f x = f (f x)
inc y     = y + 1
mult a b  = a + b
```

Σε αυτό το πρόγραμμα η συνάρτηση `twice` παίρνει μια άλλη συνάρτηση `f` σαν όρισμα. Επίσης, στο σώμα της `result`, υπάρχουν άλλες δύο εκφράσεις υψηλότερης τάξης: η `inc`, που αναφέρεται στην ομώνυμη συνάρτηση, και η `(mult 3)`, που είναι η μερική εφαρμογή της `mult` στην τιμή `3`.

Με χρήση του defunctionalization, οι δύο εκφράσεις υψηλότερης τάξης αντικαθίστανται από δύο νέους κατασκευαστές ενός τύπου δεδομένων (data type constructors) `Inc` και `Mult`, οι οποίοι εφαρμόζονται μέσω μιας νέας συνάρτησης `apply`:

```
data Defunc = Inc | Mult Int
result      = (twice Inc 1) + (twice (Mult 3) 2)
twice f x   = apply f (apply f x)
inc y       = y + 1
mult a b    = a + b
apply c z   = case c of
                Inc    → inc z
                Mult m → mult m z
```

Ο μετασχηματισμός defunctionalization είναι η βασική τεχνική που μπορεί να απαλείψει όλες τις εκφράσεις υψηλότερης τάξης που βρίσκονται σε ένα πρόγραμμα,[4] σε αντίθεση με άλλες τεχνικές που έχουν μόνο μερική επιτυχία, όπως η τεχνική η-επέκτασης και εξειδίκευσης συναρτήσεων των Chin και Darlington [60], ο μετασχηματισμός firstification του Nelan [190], η παραλλαγή του αλγόριθμου deforestation του Nishimura [197] ή ο αλγόριθμος Firstify των Mitchell και Runciman [176]. Ο μετασχηματισμός όμως απαιτεί η γλώσσα πρώτης τάξης να υποστηρίζει τύπους με πολλαπλούς κατασκευαστές, όπως ο τύπος δεδομένων `Defunc` του παραπάνω παραδείγματος.

Έχει χρησιμοποιηθεί στην υλοποίηση αρκετών γλωσσών προγραμματισμού, όπου ο μεταγλωττιστής χρησιμοποιεί μια ενδιάμεση γλώσσα πρώτης τάξης [38, 54, 123, 276]. Αυτές οι υλοποιήσεις μπορεί να είναι πολύ γρήγορες όταν εκτελούνται σε συγκεκριμένους επεξεργαστές, να χειρίζονται αποδοτικά τις εκφράσεις υψηλότερης τάξης και να προσφέρουν ευκαιρίες για απαλοιφή κλήσεων κώδικα (code inlining) [227]. Η τεχνική ανακαλύφθηκε ξεχωριστά στις γλώσσες λογικού προγραμματισμού υψηλότερης τάξης [295] και χρησιμοποιήθηκε στην υλοποίηση της γλώσσας HiLog [57]. Έχει επίσης χρησιμοποιηθεί στην υλοποίηση αλγόριθμων συλλογής σκουπιδιών που διατηρούν τους τύπους (type-preserving garbage collectors) [294] και μια παραλλαγή της χρησιμοποιήθηκε στην υποστήριξη προγραμμάτων υψηλότερης τάξης από το εργαλείο ανάλυσης πρώτης τάξης Catch [176]. Τέλος, έχει χρησιμοποιηθεί σε εφαρμογές και γλώσσες βάσεων δεδομένων για την υποστήριξη εκφράσεων υψηλότερης τάξης [114, 142] και σε συστήματα απόδειξης θεωρημάτων με τη βοήθεια του υπολογιστή [305].

Ο μετασχηματισμός defunctionalization αποτελεί το αντίστροφο της κωδικοποίησης του Church [67] και μπορεί να χρησιμοποιηθεί για να αναλυθούν προγράμματα με βάση δομές δεδομένων αντί για συναρτήσεις υψηλότερης τάξης. Οι Ager, Biernacki, Danvy και Midtgaard τον χρησιμοποίησαν για να εξάγουν ορισμούς αφηρημένων μηχανών (abstract machines) και μεταγλωττιστές από διερμηνείς [4, 6], ενώ ο Danvy τον χρησιμοποίησε σαν έναν σημαντικό μετασχηματισμό που αποτελεί μέρος μιας συλλογής τεχνικών για την ανάλυση της αναπα-

[4] Η τεχνική του Turner [278] μπορεί επίσης να το κάνει αυτό αλλά αλλάζει ριζικά τη γλώσσα του προγράμματος.

ράστασης και της σειράς αποτίμησης προγραμμάτων, και για τη μελέτη αφηρημένων μηχανών [66]. Ο μετασχηματισμός έχει επίσης χρησιμοποιηθεί στην απόδειξη μετασχηματισμών προγραμμάτων που βασίζονται στη ροή ελέγχου (control flow) για συναρτησιακές γλώσσες προγραμματισμού [30]. Αρχικά ο μετασχηματισμός παρουσιάστηκε από τον Reynolds για γλώσσες χωρίς τύπους [234], ενώ στη συνέχεια αποδείχτηκε ότι ο μετασχηματισμός είναι ορθός [30, 193] και διατηρεί τους τύπους, σε γλώσσες με συστήματα απλών [33] ή πολυμορφικών τύπων [227].

Ο Reynolds προειδοποίησε (και οι Ager *et al*. έδειξαν αναλυτικά [5]) ότι η σειρά αποτίμησης της γλώσσας πρώτης τάξης μπορεί να επηρεάσει τη σειρά αποτίμησης της γλώσσας υψηλότερης τάξης που ορίζεται με τη βοήθεια του defunctionalization. Στο υπόλοιπο κείμενο θα υποθέτουμε ότι εφαρμόζουμε τον μετασχηματισμό αυτό μόνο μέσα στην ίδια τη γλώσσα, δηλαδή σαν τελική γλώσσα χρησιμοποιούμε μόνο το υποσύνολο πρώτης τάξης της γλώσσας υψηλότερης τάξης του προγράμματος εισόδου.

2.3 Συνδυάζοντας τους δύο μετασχηματισμούς

Σε αυτήν την ενότητα θα δείξουμε ότι οι δύο κλασικές εκδοχές του νοηματικού μετασχηματισμού είναι εκφραστικά ισοδύναμες χρησιμοποιώντας τον μετασχηματισμό defunctionalization. Θα δούμε επίσης γιατί ο συνδυασμός του νοηματικού μετασχηματισμού και του μετασχηματισμού defunctionalization δεν είναι αρκετά ισχυρός ώστε να μπορεί να μετασχηματίσει πάντα προγράμματα υψηλότερης τάξης σε νοηματικά προγράμματα.

2.3.1 Defunctionalization με ακέραιους

Έστω το εξής πρόγραμμα υψηλότερης τάξης:

```
result = (f inc 5) + (g dec 6)
f g x  = g (g x)
inc a  = a + 1
dec b  = b − 1
```

Το πρόγραμμα αυτό μπορεί να μετασχηματιστεί σε ένα ισοδύναμο πρόγραμμα πρώτης τάξης μέσω του defunctionalization ως εξής:

```
Data Defunc = Inc | Dec
result      = (f Inc 5) + (f Dec 6)
f g x       = apply g (apply g x)
inc a       = a + 1
dec b       = b − 1
apply h y   = case h of
                Inc → inc y
                Dec → dec y
```

Εδώ παρατηρούμε ότι το αρχικό πρόγραμμα υψηλότερης τάξης έχει ένα ιδιαίτερο χαρακτηριστικό: δεν εμφανίζεται καθόλου η μερική εφαρμογή συναρτήσεων και οι μόνες εκφράσεις υψηλότερης τάξης που περνούν ως παράμετροι είναι ονόματα συναρτήσεων. Με άλλα λόγια, πρόκειται για τη γλώσσα που όπως είδαμε στην Ενότητα 2.1.2 είναι η γλώσσα του νοηματικού μετασχηματισμού υψηλότερης τάξης [245].

Σε αυτήν τη γλώσσα, όλοι οι κατασκευαστές που δημιουργούνται από τον μετασχηματισμό defunctionalization –όπως ο `Inc` και ο `Dec` στο παράδειγμα– δεν έχουν παραμέτρους.

Αυτό σημαίνει ότι ο τύπος δεδομένων `Defunc` είναι μια απαρίθμηση (enumeration) και μπορεί να αναπαρασταθεί από ένα πεπερασμένο σύνολο από ακέραιους. Για παράδειγμα, αν ο κατασκευαστής `Inc` αντιστοιχηθεί στον αριθμό `1` και ο `Dec` στον αριθμό `2`, τότε το πρόγραμμα μπορεί να ξαναγραφεί χρησιμοποιώντας μόνο αριθμούς (όπου η ειδική συνάρτηση `error` τυπώνει ένα μήνυμα λάθους):

```
result    = (f 1 5) + (f 2 6)
f g x     = apply g (apply g x)
inc a     = a + 1                      ── συνάρτηση 1
dec b     = b − 1                      ── συνάρτηση 2
apply h y = if h==1 then
              inc y
            else if h==2 then
              dec y
            else
              error "unknown␣function␣h"
```

Τα παραπάνω δείχνουν ότι μπορούμε να μετασχηματίσουμε κάθε πρόγραμμα υψηλότερης τάξης που χρησιμοποιεί μόνο ονόματα συναρτήσεων σαν εκφράσεις υψηλότερης τάξης, σε ισοδύναμο πρόγραμμα πρώτης τάξης σε μια γλώσσα που υποστηρίζει μόνο ακέραιους.[5] Αυτό σημαίνει ότι *προγράμματα γραμμένα στη γλώσσα εισόδου του νοηματικού μετασχηματισμού υψηλότερης τάξης μπορούν να μετασχηματιστούν σε προγράμματα στη γλώσσα εισόδου του νοηματικού μετασχηματισμού πρώτης τάξης*. Σε όρους νοηματικού προγραμματισμού, αυτό σημαίνει ότι τα νοηματικά προγράμματα πολλαπλών διαστάσεων του νοηματικού μετασχηματισμού υψηλότερης τάξης μπορούν να μετασχηματιστούν σε νοηματικά προγράμματα μιας διάστασης.

Επομένως οι δύο κλασικές εκδοχές του νοηματικού μετασχηματισμού μετασχηματίζουν την ίδια κλάση προγραμμάτων άρα έχουν την ίδια εκφραστική δύναμη. Στο υπόλοιπο αυτής της διατριβής θα χρησιμοποιήσουμε σαν θεωρητικό εργαλείο τον νοηματικό μετασχηματισμό πρώτης τάξης, μιας και είναι απλούστερος από τον νοηματικό μετασχηματισμό υψηλότερης τάξης.

2.3.2 Κατασκευαστές: το χαρακτηριστικό που λείπει

Τι συμβαίνει όταν προσπαθήσουμε να εφαρμόσουμε τον μετασχηματισμό defunctionalization σε προγράμματα με μερικές εφαρμογές συναρτήσεων; Αν και υπάρχει το *lightweight defunctionalization* [30], μια παραλλαγή του μετασχηματισμού που μπορεί να απαλείψει παραμέτρους από τους κατασκευαστές ώστε να ισχύουν οι περιορισμοί της Ενότητας 2.3.1, αυτό δε λειτουργεί πάντα. Στη γενική περίπτωση η τεχνική defunctionalization απαιτεί η τελική γλώσσα πρώτης τάξης να υποστηρίζει κατασκευαστές με παραμέτρους. Στις μη αυστηρές γλώσσες που είναι και το αντικείμενο αυτής της διατριβής, απαιτείται επιπλέον η αποτίμηση των κατασκευαστών να είναι μη αυστηρή, δηλαδή οι παράμετροι των κατασκευαστών να υπολογίζονται μόνο όταν χρειάζεται.

Υπάρχουν δύο λύσεις σε αυτό το πρόβλημα:

1. Αφού η γλώσσα πρώτης τάξης υποστηρίζει ακέραιους αριθμούς, θα μπορούσαμε να χρησιμοποιήσουμε κάποια κωδικοποίηση (όπως αυτή του Gödel [188]) για να αναπαραστήσουμε τους μη αυστηρούς κατασκευαστές με αριθμούς. Δυστυχώς αυτή η τεχνική, εκτός από το ότι θα έκανε πολύ πιο δυσανάγνωστο το παραγόμενο νοηματικό

[5] Η ιδέα ότι ένα όνομα που χρησιμοποιείται ως έκφραση υψηλότερης τάξης μπορεί να αναπαρασταθεί ως μια μοναδική σταθερά δεν είναι καινούρια· αναφέρεται π.χ. σε εργασία των Epstein, Black και Peyton-Jones [82].

πρόγραμμα, έχει και ένα σοβαρότερο μειονέκτημα: θα χειροτέρευε την πολυπλοκότητα του τελικού προγράμματος, μιας και προγράμματα με ακέραιους και προγράμματα με κατασκευαστές έχουν γενικά διαφορετική πολυπλοκότητα [135].

2. Μπορούμε να προσθέσουμε μη αυστηρούς κατασκευαστές και ταίριασμα προτύπων (pattern matching) στον νοηματικό μετασχηματισμό και στη νοηματική του γλώσσα. Έτσι υποστηρίζονται οι τύποι αθροίσματος που απαιτούνται από τον μετασχηματισμό defunctionalization και οι δύο μετασχηματισμοί μπορούν να συνδυαστούν. Για να γίνει αυτό χρειάζεται ο νοηματικός μετασχηματισμός να υποστηρίξει μια γλώσσα πρώτης τάξης με τύπους δεδομένων, ώστε να αποτελέσει ένα επόμενο στάδιο μετά τον μετασχηματισμό defunctionalization.

Σε αυτήν τη διατριβή επιλέξαμε τη δεύτερη λύση: στο επόμενο κεφάλαιο θα δείξουμε πώς μπορούμε να μετασχηματίσουμε μια συναρτησιακή γλώσσα πρώτης τάξης με τύπους δεδομένων σε νοηματική γλώσσα και πώς αυτό μπορεί να προστεθεί σαν χαρακτηριστικό στον νοηματικό μετασχηματισμό.

Περίληψη

Σε αυτό το κεφάλαιο περιγράψαμε τα δύο θεωρητικά εργαλεία που θα χρησιμοποιήσουμε σε αυτήν τη διατριβή, τον νοηματικό μετασχηματισμό και τον μετασχηματισμό defunctionalization. Δείξαμε επίσης ότι οι δύο διαφορετικές εκδοχές του κλασικού μετασχηματισμού έχουν την ίδια εκφραστική δύναμη αλλά δεν μπορούν να συνδυαστούν απευθείας με τον μετασχηματισμό defunctionalization, επειδή δεν υποστηρίζουν κατασκευαστές με παραμέτρους.

Στο επόμενο κεφάλαιο θα δείξουμε πώς ο κλασικός νοηματικός μετασχηματισμός πρώτης τάξης μπορεί να επεκταθεί για να υποστηρίζει τύπους δεδομένων, ώστε να μπορεί να μετασχηματίζει την τελική γλώσσα του μετασχηματισμού defunctionalization.

Κεφάλαιο 3

Ο γενικευμένος νοηματικός μετασχηματισμός

Αυτό το κεφάλαιο περιγράφει τον γενικευμένο νοηματικό μετασχηματισμό, ο οποίος αποτελεί επέκταση του κλασικού νοηματικού μετασχηματισμού πρώτης τάξης και υποστηρίζει τύπους δεδομένων, κατασκευαστές και ταίριασμα προτύπων.[1]

Αρχικά θα περιγράψουμε τι είναι οι αυθαίρετα ορισμένοι τύποι δεδομένων και το ταίριασμα προτύπων (Ενότητα 3.1). Θα δείξουμε με ένα παράδειγμα πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να μετασχηματίσει ένα πρόγραμμα με τύπους δεδομένων (Ενότητα 3.2) και στη συνέχεια θα δώσουμε έναν τυπικό ορισμό του γενικευμένου μετασχηματισμού (Ενότητα 3.3). Τέλος, θα παρουσιάσουμε τα χαρακτηριστικά και την τελική εκφραστική δύναμη του νέου μετασχηματισμού (Ενότητα 3.4).

3.1 Τύποι δεδομένων και ταίριασμα προτύπων

Το υλικό ενός υπολογιστή συνήθως προσφέρει λίγους και βασικούς πρωτόγονους τύπους δεδομένων, όπως οι ακέραιοι από κάποιο συγκεκριμένο σύνολο ή οι αριθμοί κινητής υποδιαστολής πεπερασμένης ακρίβειας. Αυτοί οι τύποι δεν είναι αρκετοί για να εκφράσουν την ποικιλία των μορφών που έχουν τα δεδομένα κάθε προγράμματος: πρέπει η γλώσσα προγραμματισμού να παρέχει στον χρήστη τη δυνατότητα κατασκευής νέων, εκφραστικότερων τύπων δεδομένων, που να περιγράφουν τις τιμές και τις ιδιότητες των αντικειμένων που περιέχουν τα δεδομένα του προγράμματος [47].

Οι τύποι δεδομένων είναι βασικό χαρακτηριστικό των γλωσσών προγραμματισμού· η γλώσσα Plankalkül που αναπτύχθηκε τη δεκαετία του 1940 από τον Konrad Zuse ήδη υποστήριζε δομές δεδομένων που να ορίζονται από τον χρήστη [108], ενώ η ALGOL 60 ήταν η πρώτη γλώσσα που χρησιμοποίησε την πληροφορία τύπων για να ελέγξει την ορθότητα ενός προγράμματος στον χρόνο μεταγλώττισης [47]. Η Haskell υποστηρίζει *αλγεβρικούς τύπους δεδομένων* (*algebraic data types*), δηλαδή τύπους δεδομένων που μπορούν να προκύψουν από τη σύνθεση άλλων (ή αναδρομικά των ίδιων) τύπων, χρησιμοποιώντας αλγεβρικές πράξεις όπως το άθροισμα (sum) και το γινόμενο (product). Για παράδειγμα, ο τύπος των λιστών από ακέραιους των 32 bit μπορεί να δηλωθεί ως εξής:

```
data IntList = Nil | Cons Int32 IntList
```

Αυτή η δήλωση περιγράφει τον τύπο `IntList`, ο οποίος είναι το άθροισμα των πεδίων τιμών που μπορούν να δημιουργηθούν χρησιμοποιώντας τους δύο *κατασκευαστές* `Nil` και `Cons`. Ο κατασκευαστής `Nil` δεν δέχεται παραμέτρους, άρα μπορεί να κατασκευάσει μόνο μια τιμή, ενώ ο `Cons` μπορεί να κατασκευάσει γινόμενα των δύο τύπων `Int32` και `IntList`. Στην τελευταία περίπτωση, βλέπουμε ότι η Haskell παρέχει έναν αφηρημένο τύπο δεδομένων `Int32`

[1] Το κεφάλαιο περιλαμβάνει υλικό που παρουσιάστηκε το 2013 σε άρθρο των Γ. Φουρτούνη, Ν. Παπασπύρου και Π. Ροντογιάννη [97]. Μέρος της τεχνικής παρουσιάστηκε επίσης σε άρθρο του 2011 από τους ίδιους συγγραφείς [96].

για να περιγράψει[2] τον πρωτόγονο τύπο δεδομένων ακεραίων 32 bit [63]. Για παράδειγμα, η μεταβλητή `ilist` μπορεί να δηλωθεί ότι περιέχει τη λίστα που αποτελείται από τους αριθμούς 1, 2, 3 ως εξής:

```
ilist = Cons 1 (Cons 2 (Cons 3 Nil))
```

Έστω ότι κάποιο άλλο τμήμα του προγράμματος πρέπει να εξετάσει την `ilist`: πρέπει τότε να μπορεί να βρει αν η έκφραση που περιέχεται στη μεταβλητή είναι κατασκευασμένη από τον `Nil` ή τον `Cons`. Επιπλέον, στην περίπτωση του `Cons`, πρέπει να μπορεί να χρησιμοποιήσει την πληροφορία που περιέχει στα *πεδία* (*data type fields*) που περιέχει. Αυτό γίνεται μέσω μιας έκφρασης *ταιριάσματος προτύπων* (*pattern matching*), η οποία στη Haskell δίνεται με την έκφραση `case`:

```
head = case ilist of
         Nil       → error "Empty␣list!"
         Cons i il → i
```

Εδώ εξετάζεται αν η τιμή της `ilist` ταιριάζει είτε με το πρότυπο `Nil` ή με το πρότυπο `Cons i il`. Κάθε πρότυπο αντιστοιχεί σε έναν κατασκευαστή και, αν κάποιος κατασκευαστής έχει πεδία, το πρότυπο τότε περιέχει ονόματα νέων μεταβλητών, οι οποίες θα δεσμευτούν στις τιμές των αντίστοιχων πεδίων. Στο παράδειγμα, η `i` θα αποκτήσει την τιμή 1.

Το ταίριασμα προτύπων ως ιδέα προτάθηκε ανεξάρτητα από τους Burstall και Turner [216, §4.1.4] και αποτελεί πια βασικό χαρακτηριστικό των συναρτησιακών γλωσσών προγραμματισμού. Στις μη αυστηρές γλώσσες το ταίριασμα προτύπων έχει μια επιπλέον ιδιότητα: η έκφραση που εξετάζεται[3] πρέπει να αποτιμηθεί μόνο μέχρι να βρεθεί ο κατασκευαστής της [101]. Η πληροφορία αυτή είναι αρκετή για να ταιριάξει με κάποιο πρότυπο και να συνεχιστεί η αποτίμηση του προγράμματος ακολουθώντας τον κατάλληλο κλάδο της έκφρασης `case`. Τα πεδία του κατασκευαστή θα δεσμευτούν στις νέες μεταβλητές του προτύπου και θα αποτιμηθούν αργότερα, αν χρειαστούν [216].

Μια ισχυρή γενίκευση των αλγεβρικών τύπων δεδομένων είναι οι *γενικευμένοι αλγεβρικοί τύποι δεδομένων* (*Generalized Algebraic Data Types*, *GADTs*) [215, 255, 300], οι οποίοι περιέχουν κατάλληλη πληροφορία τύπων, ώστε να είναι γνωστές ιδιότητες της χρήσης τους κατά τον χρόνο μεταγλώττισης. Αυτή η επιπλέον πληροφορία θα επέτρεπε για παράδειγμα την απαλοιφή του περιττού κλάδου με το πρότυπο `Nil` στην συνάρτηση `head` του παραπάνω παραδείγματος [140]. Στη Haskell, οι γενικευμένοι αλγεβρικών τύποι δεδομένων δηλώνονται με μια παραλλαγή της σύνταξης των απλών αλγεβρικών τύπων, που χρησιμοποιεί σημειώσεις τύπων. Για παράδειγμα, το παραπάνω παράδειγμα θα γινόταν:

```
data Empty
data NonEmpty
data IntList a where
  Nil  :: IntList Empty
  Cons :: Int32 → IntList a → IntList NonEmpty
head :: IntList NonEmpty → Int32
head = case ilist of
         Cons i il → i
```

Στο υπόλοιπο αυτής της διατριβής, θα χρησιμοποιήσουμε την κατάλληλη σύνταξη της Haskell (`data` ή `data ... where`), ανάλογα με τον τύπο που θα δηλώνουμε κάθε φορά

[2] Η Haskell παρέχει επίσης τους τύπους δεδομένων `Int`, για ακέραιους αριθμούς γενικά μικρότερης ακρίβειας που καθορίζεται από την υλοποίηση, και `Integer`, για ακέραιους αριθμούς αυθαίρετου μεγέθους (bignums). Οι δύο αυτοί τύποι χρησιμοποιείται συνήθως περισσότερο στην πράξη.

[3] Στη Haskell συνήθως ονομάζεται *scrutinee* [249] ή *discriminator* [119].

(απλό ή γενικευμένο). Η σύνταξη για τους γενικευμένους αλγεβρικούς τύπους δεδομένων μπορεί να χρησιμοποιηθεί για τη γενική περίπτωση, κάτι που θα εκμεταλλευτούμε αργότερα, στο Κεφάλαιο 7.

3.2 Μετασχηματισμός ενός προγράμματος που περιέχει τύπους δεδομένων

Όπως αναφέραμε στην Ενότητα 2.1, ο κλασικός νοηματικός μετασχηματισμός ποτέ δεν γενικεύτηκε ώστε να μπορεί να εφαρμοστεί σε μια γλώσσα υψηλότερης τάξης ή σε μια γλώσσα που να υποστηρίζει δομές δεδομένων που να ορίζονται από τον χρήστη. Οι εκφράσεις υψηλότερης τάξης και οι δομές δεδομένων σχετίζονται όσον αφορά την υλοποίησή τους, μιας και ο μετασχηματισμός defunctionalization μπορεί να μετασχηματίσει ένα πρόγραμμα υψηλότερης τάξης σε ένα πρώτης τάξης, προσθέτοντας κάποιες δομές δεδομένων. Τα δύο προβλήματα δηλαδή μπορούν να λυθούν ταυτόχρονα αν γενικεύσουμε τον νοηματικό μετασχηματισμό ώστε να εφαρμόζεται σε προγράμματα πρώτης τάξης με οριζόμενους από τον χρήστη τύπους δεδομένων. Για παράδειγμα, έστω το εξής πρόγραμμα υψηλότερης τάξης σε Haskell:

```
result  = inc (add 1) 2 + inc sq 3
inc f x = f (x + 1)
add a b = a + b
sq z    = z * z
```

Ο κώδικάς του μετασχηματίζεται στον εξής κώδικα πρώτης τάξης:

```
result     = inc (fadd 1) 2 + inc fsq 3
inc f x    = apply f (x + 1)
add a b    = a + b
sq z       = z * z

data Func  = Fadd Int | Fsq
fadd c     = Fadd c
fsq        = Fsq

apply cl d = case cl of
               Fadd c → add c d
               Fsq    → sq d
```

Τα παραπάνω αποτελούν συνηθισμένη εφαρμογή του defunctionalization, με δύο μικρές παραλλαγές. Αρχικά προσθέσαμε δύο συναρτήσεις `fadd` και `fsq`, οι οποίες χρησιμοποιούνται σε όσα σημεία καλούνταν οι κατασκευαστές `Fadd` και `Fsq`. Επίσης, στο πρότυπο (pattern) της `case` που αντιστοιχεί στον κατασκευαστή `Fadd`, έχουμε χρησιμοποιήσει την ίδια μεταβλητή `c` που εμφανίζεται στον ορισμό της `fadd`. Αυτές οι δύο συμβάσεις (που θα εξεταστούν αναλυτικότερα στην Ενότητα 3.3) επιτρέπουν στον νοηματικό μετασχηματισμό να εφαρμοστεί και να προκύψει το νοηματικό πρόγραμμα μηδενικής τάξης, όπως και στον κλασικό νοηματικό μετασχηματισμό:

```
result = call_0(inc) + call_1(inc)
inc    = call_0(apply)
add    = a+b
sq     = z*z
```

```
fadd   = Fadd
fsq    = Fsq

apply  = case cl of
           Fadd → call_0(add)
           Fsq  → call_0(sq)

f      = actuals(call_0(fadd), fsq)
x      = actuals(2, 3)
a      = actuals(c)
b      = actuals(d)
z      = actuals(d)
c      = actuals(1)
cl     = actuals(f)
d      = actuals(x+1)
```

Το παραπάνω πρόγραμμα μπορεί να εκτελεστεί χρησιμοποιώντας τις γνωστές αρχές της Ενότητας 2.1, κατασκευάζοντας έναν διερμηνέα με βάση μια συνάρτηση $EVAL_p(e, w)$, η οποία θα οριστεί με τυπικό τρόπο στην Ενότητα 3.3 που ακολουθεί.

3.3 Τυπικός ορισμός του γενικευμένου μετασχηματισμού

Σε αυτήν την ενότητα θα παρουσιάσουμε τον γενικευμένο νοηματικό μετασχηματισμό με τυπικό τρόπο, ενώ θα παραλείψουμε τον μετασχηματισμό defunctionalization, ως γνωστή τεχνική. Υποθέτουμε ότι η αρχική γλώσσα του νοηματικού μετασχηματισμού είναι η τελική γλώσσα του defunctionalization, δηλαδή μια μη αυστηρή γλώσσα προγραμματισμού πρώτης τάξης με τύπους δεδομένων που ορίζονται από τον χρήστη. Τη γλώσσα αυτή την αποκαλούμε FOFL (First-Order Functional Language).

Η σύνταξη της FOFL ορίζεται από την ακόλουθη γραμματική χωρίς συμφραζόμενα, όπου το f και το v είναι μεταβλητές, το c είναι σταθερά, το κ είναι κατασκευαστής και $n, m \geq 0$. Όταν $n = 0$, θα παραλείπουμε τις κενές παρενθέσεις.

$p ::= d_0, \ldots, d_n$	*πρόγραμμα*
$d ::= f(v_0, \ldots, v_{n-1}) = e$	*ορισμός*
$e ::= c(e_0, \ldots, e_{n-1}) \mid f(e_0, \ldots, e_{n-1})$	*έκφραση*
$\quad \mid \kappa(e_0, \ldots, e_{n-1}) \mid$ `case` e `of {` b_0 `; ... ;` b_n `} \mid` $\#^m(v)$	
$b ::= \kappa(v_0, \ldots, v_{n-1}) \to e$	*ταίριασμα προτύπου*

Υποθέτουμε επίσης ότι όλες οι τυπικές παράμετροι των συναρτήσεων έχουν μοναδικά ονόματα (κάτι που εξασφαλίζεται με ένα βήμα μετονομασίας πριν τον μετασχηματισμό). Επίσης για κάθε κατασκευαστή κ με n παραμέτρους, υπάρχει μια συνάρτηση που ορίζεται ως εξής:

$$f_\kappa(v_0, \ldots, v_{n-1}) = \kappa(v_0, \ldots, v_{n-1})$$

και όλες οι εμφανίσεις του κ στο πρόγραμμα αντικαθιστώνται από την f_κ. Επιπλέον, όλα τα πρότυπα που αντιστοιχούν στον κατασκευαστή κ σε όλες τις εκφράσεις `case` θα χρησιμοποιούν τις ίδιες μεταβλητές $v_0, \ldots, v_{n-1}$ που εμφανίζονται στον ορισμό της f_κ. Δυστυχώς αυτό δεν μπορεί απλά να εξασφαλιστεί με μετονομασία, γιατί μπορεί να υπάρχουν εμφωλευμένες εκφράσεις `case` (nested case expressions). Για αυτό θα χρησιμοποιήσουμε μια ειδική

μορφή εκφράσεων $\#^m(v)$ που θα χειρίζεται αυτά τα θέματα ορατότητας ονομάτων δεσμευμένων μεταβλητών.

Η $\#^m(v)$ αντιστοιχεί στη μεταβλητή v που δεσμεύεται σε ένα πρότυπο της εμφωλευμένης έκφρασης `case` σε *βάθος* (*nesting depth*) m. Για παράδειγμα, η συνάρτηση `apply` στο παράδειγμα της προηγούμενης ενότητας μπορεί να γραφεί ως εξής:

$$
\begin{array}{lcl}
\mathit{apply}(\mathit{cl}, d) & = & \texttt{case } \mathit{cl} \texttt{ of \{} \\
& & \quad \mathit{Add}(c) \rightarrow \mathit{add}(\#^0(c), d); \\
& & \quad \mathit{Sq} \rightarrow \mathit{sq}(d) \\
& & \texttt{\}}
\end{array}
$$

όπου η $\#^0(c)$ αντιστοιχεί στη μεταβλητή c που δεσμεύεται από το πρότυπο $\mathit{Add}(c)$ της έκφρασης `case`. Παρακάτω ακολουθεί ένα παράδειγμα με εμφωλευμένες εκφράσεις `case`, όπου η έκφραση στο αριστερό μέλος (σε σύνταξη Haskell, που υπολογίζει το άθροισμα των πρώτων δύο στοιχείων μιας λίστας) μπορεί να μετατραπεί σε αυτή του δεξιού μέλους:

```
case l of
  Nil → 0
  Cons x xs →
    case xs of
      Nil → x
      Cons y ys → x+y
```

$$
\begin{array}{l}
\texttt{case } l \texttt{ of \{} \\
\quad \mathit{Nil} \rightarrow 0; \\
\quad \mathit{Cons}(h, t) \rightarrow \\
\qquad \texttt{case } \#^0(t) \texttt{ of \{} \\
\qquad\quad \mathit{Nil} \rightarrow \#^1(h); \\
\qquad\quad \mathit{Cons}(h, t) \rightarrow +(\#^1(h), \#^0(h)) \\
\qquad \texttt{\}} \\
\quad \texttt{\}}
\end{array}
$$

Παρατηρούμε εδώ ότι τα ίδια ονόματα μεταβλητών (h, t) χρησιμοποιούνται και στα δύο πρότυπα του *Cons* και ότι η `x` και η `y`, οι οποίες αντιστοιχούν και οι δύο στην h, διακρίνονται από την τιμή m, το βάθος των εκφράσεων `case`.

3.3.1 Η γενικευμένη NVIL

Τα προγράμματα σε FOFL μετασχηματίζονται σε νοηματικά προγράμματα μηδενικής τάξης στη γλώσσα NVIL (Nullary Variables Intensional Language). Περισσότερες πληροφορίες για αυτή τη γλώσσα αναφέρονται στην Ενότητα 1.2 ή στις πρώτες ενότητες του άρθρου των Ροντογιάννη και Wadge για τον κλασικό νοηματικό μετασχηματισμό πρώτης τάξης [244]. Η μόνη διαφορά της NVIL σε σχέση με την αντίστοιχη γλώσσα που ορίζεται στο [244] είναι ότι η πρώτη υποστηρίζει τύπους δεδομένων που ορίζονται από τον χρήστη. Η σύνταξη της NVIL δίνεται από τη γραμματική χωρίς συμφραζόμενα που ακολουθεί. Παρατηρούμε ότι η σύνταξη των νοηματικών τελεστών (`call` και `actuals`) διαφέρει λίγο από αυτήν που χρησιμοποιήσαμε στην Ενότητα 2.1 και ότι οι εκφράσεις $\#^m(v)$ έχουν αντικατασταθεί από τις γενικότερες $\#^m(e)$.

$$
\begin{array}{lll}
p ::= d_0, \ \ldots, \ d_n & & \textit{πρόγραμμα} \\
d ::= f = e & & \textit{ορισμός} \\
e ::= c(e_0, \ \ldots, \ e_{n-1}) \mid f \mid \kappa & & \textit{έκφραση} \\
\quad \mid \ \texttt{case } e \texttt{ of \{ } b_0 \texttt{ ; } \ldots \texttt{ ; } b_n \texttt{ \}} \mid \#^m(e) \mid \texttt{call}_\ell(e) & & \\
\quad \mid \ \texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}) & & \\
b ::= \kappa \rightarrow e & & \textit{ταίριασμα προτύπου}
\end{array}
$$

$$
\begin{aligned}
&\mathit{EVAL}_p(c(e_0,\ \ldots,\ e_{n-1}), w) \ = \ c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w))\\
&\mathit{EVAL}_p(f, w) \ = \ \mathit{EVAL}_p(\mathit{body}(f, p), w)\\
&\mathit{EVAL}_p(\kappa, w) \ = \ \langle \kappa, w\rangle\\
&\mathit{EVAL}_p(\texttt{case}\ e\ \texttt{of}\ \{\kappa_0 \to e_0;\ \ldots;\ \kappa_n \to e_n\}, \langle \ell, w, \mu\rangle) \ = \ \mathit{EVAL}_p(e_i, \langle \ell, w, w' : \mu\rangle)\\
&\qquad \text{εάν } \mathit{EVAL}_p(e, \langle \ell, w, \mu\rangle) = \langle \kappa_i, w'\rangle\\
&\mathit{EVAL}_p(\#^m(e), \langle \ell, w, \mu\rangle) \ = \ \mathit{EVAL}_p(e, \mu_m)\\
&\mathit{EVAL}_p(\texttt{call}_\ell(e), w) \ = \ \mathit{EVAL}_p(e, \langle \ell, w, \bullet\rangle)\\
&\mathit{EVAL}_p(\texttt{actuals}(\langle e_\ell\rangle_{\ell\in I}), \langle \ell, w, \mu\rangle) \ = \ \mathit{EVAL}_p(e_\ell, w)
\end{aligned}
$$

Εικόνα 3.1: Σημασιολογία της NVIL.

Στην Ενότητα 2.1, ο τελεστής `call` είχε σαν δείκτη έναν φυσικό αριθμό i και ο τελεστής `actuals` δεχόταν μια ακολουθία από εκφράσεις, που δεικτοδοτούνταν από το i. Εδώ γενικεύουμε τον δείκτη να είναι οποιοδήποτε στοιχείο ℓ από ένα κατάλληλο σύνολο *Labels*. Επομένως, οι δείκτες της `call` ανήκουν στο ℓ και η `actuals` παίρνει μια ακολουθία από εκφράσεις e_ℓ που δεικτοδοτούνται από ετικέτες που ανήκουν σε ένα υποσύνολο $I \subseteq$ *Labels*. Αναπαριστούμε αυτήν την ακολουθία ως $\langle e_\ell\rangle_{\ell\in I}$. Αυτή η σύμβαση δεν επηρεάζει τη σημασιολογία της NVIL αλλά θα είναι χρήσιμη στον ορισμό του νοηματικού μετασχηματισμού.

Η σημασιολογία της NVIL δίνεται στην Εικόνα 3.1. Όπως και στην Ενότητα 2.1, ορίζεται με τη βοήθεια μιας συνάρτησης αποτίμησης $\mathit{EVAL}_p(e, w)$, όπου p είναι το πρόγραμμα, e είναι η έκφραση που πρόκειται να αποτιμηθεί και w τα νοηματικά συμφραζόμενα. Σε αντίθεση με την απλή δομή των συμφραζομένων (λίστες από ετικέτες) που χρησιμοποιούνταν στον νοηματικό μετασχηματισμό πρώτης τάξης [244], οι τύποι δεδομένων απαιτούν ένα πιο πολύπλοκο είδος συμφραζομένων. Αν και μοιάζουν με τις λίστες με δείκτες προς τα πίσω (lists with backpointers ή b-lists) που είχε χρησιμοποιήσει ο Yaghi για να υλοποιήσει εμφωλευμένες δηλώσεις [302], τα συμφραζόμενά μας είναι διαφορετικά και αναπαριστούν μια ειδική μορφή δένδρων. Στα δένδρα αυτά, κάθε κόμβος αναπαριστά συμφραζόμενα, ένας κόμβος είναι παιδί ενός άλλου όταν το πεδίο w του πρώτου δείχνει στον δεύτερο, και έχουν προστεθεί ακμές από κόμβους συμφραζομένων κατασκευαστών σε προγονικούς κόμβους συμφραζομένων εκφράσεων `case` μέσω του πεδίου μ.

Τα συμφραζόμενα ορίζονται από τη γραμματική που ακολουθεί. Το νέο στοιχείο είναι το μ, που είναι η λίστα από *εμφωλευμένα συμφραζόμενα* (*nested contexts*) που αντιστοιχούν σε εμφωλευμένες εκφράσεις `case`.

$$
\begin{aligned}
w \ &::= \ \bullet \mid \langle \ell, w, \mu\rangle\\
\mu \ &::= \ \bullet \mid w : \mu
\end{aligned}
$$

Το αποτέλεσμα της συνάρτησης $\mathit{EVAL}_p(e, w)$ μπορεί να είναι είτε απλή τιμή (ground value), που επιστρέφεται από κάποιο τελεστή c (όπως για παράδειγμα ένας ακέραιος αριθμός), είτε ένα ζεύγος της μορφής $\langle \kappa, w\rangle$, που αντιστοιχεί σε τιμή ενός τύπου δεδομένων που ορίστηκε από τον χρήστη. Στην τελευταία περίπτωση, κ είναι ο κατασκευαστής που χρησιμοποιήθηκε για να δημιουργηθεί η τιμή και w είναι τα συμφραζόμενα που πρέπει να χρησιμοποιηθούν για να υπολογιστούν οι παράμετροι που περιέχονται στον κατασκευαστή. Αυτή η σημασιολογία εμφανίζεται στην εξίσωση της $\mathit{EVAL}_p(\kappa, w)\cdot$ εδώ πρέπει να τονιστεί ότι, όπως είπαμε και πιο πριν, αυτές οι εκφράσεις μπορούν να εμφανιστούν μόνο στο σώμα των συναρτήσεων f_κ που κατασκευάστηκαν για κάθε κατασκευαστή κ.

Η σημασιολογία της `call` και της `actuals` χειρίζεται τα συμφραζόμενα όπως έχει ήδη αναφερθεί στην Ενότητα 2.1· η `call` προσθέτει μια νέα ετικέτα στα συμφραζόμενα και η `actuals` επιλέγει την έκφραση που θα υπολογίσει με βάση την τρέχουσα ετικέτα που αφαι-

ρεί από τα συμφραζόμενα. Τα πιο ενδιαφέροντα μέρη της νέας σημασιολογίας είναι οι εξισώσεις για τις εκφράσεις `case` και $\#^m$. Στην πρώτη, η έκφραση που πρόκειται να εξεταστεί αποτιμάται και βρίσκεται να είναι της μορφής $\langle \kappa_i, w' \rangle$ για κάποιον κατασκευαστή κ_i που εμφανίζεται σε έναν από τους κλάδους ταιριάσματος προτύπων της `case`. (Αυτό εξασφαλίζεται από το ότι το πρόγραμμα έχει περάσει από τον έλεγχο τύπων και οι εκφράσεις `case` ελέγχουν εξαντλητικά τα πρότυπα· δε θα ασχοληθούμε περαιτέρω με αυτά τα θέματα.) Η αποτίμηση συνεχίζει με το σώμα e_i του κλάδου αυτού αλλά τα συμφραζόμενα w' προστίθενται στη λίστα μ από συμφραζόμενα που αντιστοιχούν σε εμφωλευμένες εκφράσεις `case`. Αν αργότερα, κατά τον υπολογισμό της e_i, βρεθεί μια έκφραση της μορφής $\#^m(e)$, τα συμφραζόμενα μ_m που βρίσκονται στη θέση m της λίστας μ θα χρησιμοποιηθούν για να αποτιμηθεί η e, αντί για τα τρέχοντα συμφραζόμενα.

Για παράδειγμα, έστω το εξής πρόγραμμα που επιστρέφει το πρώτο στοιχείο μιας λίστας:

Haskell

```
data List = Nil
          | Cons Int List
result = head l
l = Cons 10 Nil
head e = case e of
           Cons x xs → x
```

FOFL

result $=$ *head* l
$l =$ *cons* 10 *nil*
cons $h\ t =$ *Cons* $h\ t$
nil $=$ *Nil*
head $e =$ `case` e `of {`
  Cons$(h, t) \rightarrow \#^0(h)$
`}`

NVIL

result $= \texttt{call}_{\ell_0}$(*head*)
$l = \texttt{call}_{\ell_1}$(*cons*)
cons $=$ *Cons*
nil $=$ *Nil*
head $=$ `case` e `of {`
  Cons $\rightarrow \#^0(h)$
`}`
$e = \texttt{actuals}(\ell_0 : l)$
$h = \texttt{actuals}(\ell_1 : 10)$
$t = \texttt{actuals}(\ell_1 :$ *nil*$)$

Η μεταβλητή *result* του προγράμματος NVIL αποτιμάται ως εξής:

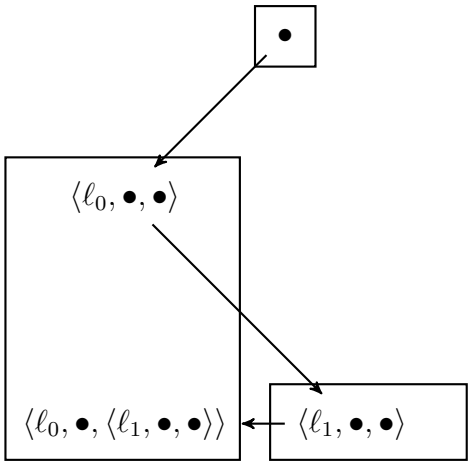
$$
\begin{aligned}
\mathit{EVAL}_p(\mathit{result}, \bullet) &= \mathit{EVAL}_p(\texttt{call}_{\ell_0}(\mathit{head}), \bullet) \\
&= \mathit{EVAL}_p(\mathit{head}, \langle \ell_0, \bullet, \bullet \rangle) \\
&= \mathit{EVAL}_p(\texttt{case}\ e\ \texttt{of}\ \{\ \mathit{Cons} \rightarrow \#^0(h)\ \}, \langle \ell_0, \bullet, \bullet \rangle)
\end{aligned}
$$

Αποτίμηση της e:

$$
\begin{aligned}
\mathit{EVAL}_p(e, \langle \ell_0, \bullet, \bullet \rangle) &= \mathit{EVAL}_p(\texttt{actuals}(\ell_0 : l), \langle \ell_0, \bullet, \bullet \rangle) \\
&= \mathit{EVAL}_p(l, \bullet) \\
&= \mathit{EVAL}_p(\texttt{call}_{\ell_1}(\mathit{cons}), \bullet) \\
&= \mathit{EVAL}_p(\mathit{cons}, \langle \ell_1, \bullet, \bullet \rangle) \\
&= \mathit{EVAL}_p(\mathit{Cons}, \langle \ell_1, \bullet, \bullet \rangle) \\
&= \langle \mathit{Cons}, \langle \ell_1, \bullet, \bullet \rangle \rangle
\end{aligned}
$$

$$
\begin{aligned}
&= \mathit{EVAL}_p(\#^0(h), \langle \ell_0, \bullet, \langle \ell_1, \bullet, \bullet \rangle \rangle) \\
&= \mathit{EVAL}_p(h, \langle \ell_1, \bullet, \bullet \rangle) \\
&= \mathit{EVAL}_p(\texttt{actuals}(\ell_1 : 10), \langle \ell_1, \bullet, \bullet \rangle) \\
&= \mathit{EVAL}_p(10, \langle \ell_1, \bullet, \bullet \rangle) \\
&= 10
\end{aligned}
$$

Το δένδρο των συμφραζομένων φαίνονται στην Εικόνα 3.2, όπου τα βέλη δείχνουν την πορεία της αποτίμησης και τα πλαίσια τους κόμβους. Τα συμφραζόμενα που διαφέρουν μόνο στο πεδίο μ, όπως τα $\langle \ell_0, \bullet, \bullet \rangle$ (πριν την αποτίμηση της έκφρασης e) και $\langle \ell_0, \bullet, \langle \ell_1, \bullet, \bullet \rangle \rangle$ (μετά την αποτίμηση της e), απεικονίζονται ως ο ίδιος κόμβος.



Εικόνα 3.2: Δένδρο συμφραζομένων κατά την αποτίμηση.

3.3.2 Ο νοηματικός μετασχηματισμός από FOFL σε NVIL

Αρχίζουμε ορίζοντας το σύνολο *labels*(f, p), που είναι το σύνολο των ετικετών των κλήσεων της f στο πρόγραμμα p. Αυτές οι ετικέτες θα αποτελέσουν τους δείκτες των τελεστών `call`. Ειδικότερα, η ετικέτα μιας κλήσης συνάρτησης $f(e_0, \ldots, e_{n-1})$ είναι απλά η ακολουθία $\langle e_0, \ldots, e_{n-1}\rangle$ των παραμέτρων της. Με άλλα λόγια, η κλήση $f(e_0, \ldots, e_{n-1})$ μετασχηματίζεται σε $\texttt{call}_\ell$ όπου $\ell = \langle e_0, \ldots, e_{n-1}\rangle$. Αυτή η παραδοχή είναι κάπως διαφορετική από αυτή που παρουσιάστηκε στην Ενότητα 2.1.1 αλλά βοηθάει σε δύο σημεία. Καταρχήν, χρησιμοποιώντας αυτήν την παραδοχή, δύο ίδιες κλήσεις συναρτήσεων στο πρόγραμμα αποκτούν την ίδια ετικέτα. Επιπλέον, επειδή μια ετικέτα ℓ είναι μια ακολουθία των πραγματικών παραμέτρων μιας κλήσης συνάρτησης, μπορούμε να γράψουμε ℓ_m για να ορίσουμε την πραγματική παράμετρο στη θέση m αυτής της κλήσης, με αποτέλεσμα να έχουμε απλούστερη σύνταξη. Επομένως:

$$\textit{labels}(f, p) \;=\; \{\langle e_0, \ldots, e_{n-1}\rangle \mid f(e_0, \ldots, e_{n-1}) \in p\}$$

Μπορούμε τώρα να ορίσουμε τον συνολικό μετασχηματισμό από FOFL σε NVIL, όπως αυτός φαίνεται στην Εικόνα 3.3. Δεδομένου ενός προγράμματος p, η συνάρτηση *Trans*(p) αφαιρεί τις τυπικές παραμέτρους όλως των ορισμών και προσθέτει έναν επιπλέον ορισμό για κάθε τυπική παράμετρο κάθε συνάρτησης του προγράμματος. Αυτοί οι νέοι ορισμοί ορίζονται από τη συνάρτηση *actdefs*. Ειδικότερα, αν δοθεί μια συνάρτηση f με τυπικές παραμέτρους $v_0, \ldots, v_{n-1}$, η συνάρτηση *actdefs*(f, p) κατασκευάζει έναν ορισμό `actuals` για κάθε v_j, ο οποίος περιέχει την ακολουθία όλων των (μετασχηματισμένων) πραγματικών παραμέτρων της f στο p που αντιστοιχούν στη θέση j. Τέλος, έχουμε τις συναρτήσεις $\mathcal{E}$ και $\mathcal{B}$, που επεξεργάζονται εκφράσεις και κλάδους εκφράσεων `case`. Ο κύριος ρόλος αυτών των δύο συναρτήσεων είναι να αντικαθιστούν κλήσεις συναρτήσεων με αντίστοιχες εμφανίσεις του τελεστή `call`.

3.4 Χαρακτηριστικά του νέου μετασχηματισμού

Το βασικότερο χαρακτηριστικό του νοηματικού μετασχηματισμού, τόσο στις κλασικές όσο και στη γενικευμένη εκδοχή του, είναι ότι *η δυναμικότητα των κλήσεων συναρτήσεων που δεσμεύουν παραμέτρους σε τυπικές μεταβλητές αντικαθίσταται από μια επιλογή ανάμεσα σε πεπερασμένες περιπτώσεις που αντιστοιχούν στις ετικέτες των κλήσεων στο πρόγραμμα*.

$$
\begin{array}{lcl}
\mathcal{E}(c(e_0,\ldots,e_{n-1})) & = & c(\mathcal{E}(e_0),\ldots,\mathcal{E}(e_{n-1})) \\
\mathcal{E}(f) & = & f \\
\mathcal{E}(f(e_0,\ldots,e_n)) & = & \texttt{call}_\ell(f) \quad \text{όπου } \ell = \langle e_0,\ldots,e_n\rangle \\
\mathcal{E}(\kappa(e_0,\ldots,e_{n-1})) & = & \kappa \\
\mathcal{E}(\texttt{case } e \texttt{ of } \{b_0\texttt{;}\ldots\texttt{;}b_n\}) & = & \texttt{case } \mathcal{E}(e) \texttt{ of } \{\mathcal{B}(b_0);\ \ldots;\ \mathcal{B}(b_n)\} \\
\mathcal{E}(\#^m(e)) & = & \#^m(\mathcal{E}(e)) \\
\mathcal{B}(\kappa(v_0,\ \ldots,\ v_{n-1}) \to e) & = & \kappa \to \mathcal{E}(e)
\end{array}
$$

$$
\textit{actdefs}(f,p) \ = \bigcup_{j=0}^{n-1} \{v_j = \texttt{actuals}(\langle \mathcal{E}(l_j)\rangle_{l\in I})\}
$$

όπου $v_0,\ldots,v_{n-1}$ είναι οι τυπικές παράμετροι της f και $I = \textit{labels}(f,p)$

$$
\textit{Trans}(p) \quad = \bigcup_{f(v_0,\ldots,v_{n-1})=e\ \in\ p} \{f = \mathcal{E}(e)\} \ \cup\ \textit{actdefs}(f,p)
$$

Εικόνα 3.3: Ο αλγόριθμος μετασχηματισμού από FOFL σε NVIL.

Αυτό αποτελεί περίπτωση *concretization*, ενός όρου που δημιουργήθηκε από τους Pottier και Gauthier για να περιγράψει την απαλοιφή δυναμικότητας με χρήση επιλογής από ένα πεπερασμένο σύνολο περιπτώσεων [227]. Ένας άλλος μετασχηματισμός που ανήκει σε αυτήν την κατηγορία είναι ο μετασχηματισμός defunctionalization [227] και στην πραγματικότητα οι δύο μετασχηματισμοί αποτελούν χρήσιμο συνδυασμό: η τεχνική defunctionalization απαλείφει τις εκφράσεις υψηλότερης τάξης από τη γλώσσα, κάνοντάς την πρώτης τάξης, ενώ ο νοηματικός μετασχηματισμός απαλείφει τις παραμέτρους από τις συναρτήσεις και τους κατασκευαστές της, κάνοντάς τη γλώσσα τελικά μηδενικής τάξης.

Οι δύο μετασχηματισμοί συμφιλιώνουν τα δύο πεδία που αναφέρθηκαν στο Κεφάλαιο 1: τις μη αυστηρές συναρτησιακές γλώσσες προγραμματισμού υψηλότερης τάξης και τις νοηματικές γλώσσες (και το συγγενές τους μοντέλο ροής δεδομένων με μονάδες με ετικέτες). Αυτό επιτυγχάνεται χωρίς να προστεθούν στοιχεία που είναι ασύμβατα με τις αρχιτεκτονικές ροής δεδομένων, όπως οι ρητές αναπαραστάσεις εκφράσεων υψηλότερης τάξης [241].

Ο γενικευμένος νοηματικός μετασχηματισμός που παρουσιάσαμε έχει σημαντική εκφραστική δύναμη. Αν συνδυαστεί με τον μετασχηματισμό πολυμορφικού defunctionalization και τις κωδικοποιήσεις για εγγραφές (records) και κλάσεις τύπων (type classes) των Pottier και Gauthier [227], μπορεί να μετατρέψει σε νοηματικές, μη αυστηρές πολυμορφικές γλώσσες με πολυμορφικές εγγραφές και γενικευμένους αλγεβρικούς τύπους δεδομένων. Επομένως, ο συνδυασμός των μετασχηματισμών είναι τόσο εκφραστικός που μπορεί να υλοποιήσει ολόκληρη τη Haskell 98 [138] και περιέχει χαρακτηριστικά του Συστήματος F_C, στο οποίο βασίζεται η εσωτερική αναπαράσταση του μεταγλωττιστή GHC της Haskell [264]. Επιπλέον, ο γενικευμένος νοηματικός μετασχηματισμός καθοδηγείται από τη σύνταξη του προγράμματος και όχι από τους τύπους (όπως ο κλασικός νοηματικός μετασχηματισμός υψηλότερης τάξης). Αυτό σημαίνει ότι μπορεί να μετασχηματίσει μη αυστηρά συναρτησιακά προγράμματα ακόμα και σε γλώσσες με δυναμικούς τύπους (όπως για παράδειγμα η R [183]).

Ο γενικευμένος νοηματικός μετασχηματισμός, απαλείφοντας τις συναρτήσεις πρώτης τάξης από το πρόγραμμα, αφαιρεί και την ανάγκη για *αντικατάσταση* (*substitution*). Η κλασική σημασιολογία των συναρτησιακών γλωσσών βασίζεται στον λ-λογισμό και στην αντικατάσταση τυπικών μεταβλητών από πραγματικές παραμέτρους, κατά την αποτίμηση του προγράμματος [31]. Τα νοηματικά προγράμματα, αντίθετα, προσομοιώνουν την αντικατάσταση,

(α) χρησιμοποιώντας ως οδηγό τα συμφραζόμενα και (β) απαιτώντας όλες οι τυπικές μεταβλητές να έχουν διαφορετικό όνομα και να χρησιμοποιείται πάντα το ίδιο όνομα για τα πεδία ενός κατασκευαστή.

Οι οκνηροί κατασκευαστές ενσωματώνουν συμφραζόμενα στον γενικευμένο νοηματικό μετασχηματισμό, η NVIL είναι δηλαδή μια νοηματική γλώσσα υψηλότερης τάξης [223]. Επομένως, η διατήρηση συμφραζομένων ως ενδιάμεσες τιμές κατά το ταίριασμα προτύπων, αρκεί για τη μετατροπή συναρτησιακών προγραμμάτων υψηλότερης τάξης σε νοηματικά.

Όπως παρατηρούν οι Marlow και Peyton Jones, οι γλώσσες υψηλότερης τάξης που υποστηρίζουν μερική εφαρμογή συνήθως υλοποιούνται με βάση κάποιο από τα ακόλουθα μοντέλα αποτίμησης για την κλήση συναρτήσεων: (α) push/enter, στο οποίο ωθείται πληροφορία σε μια στοίβα και στη συνέχεια καλείται μια συνάρτηση, και (β) eval/apply, στο οποίο μια έκφραση αποτιμάται και το αποτέλεσμά της, αν είναι συνάρτηση, εφαρμόζεται σε έναν αριθμό ορισμάτων [166]. Αν και η γλώσσα του γενικευμένου νοηματικού μετασχηματισμού είναι πρώτης τάξης και χωρίς υποστήριξη μερικής εφαρμογής, η σημασιολογία του μετασχηματισμού προτείνει επίσης ένα μοντέλο push/enter: η κλήση μιας συνάρτησης f του αρχικού προγράμματος μετατρέπεται σε μια έκφραση $\texttt{call}_\ell(f)$, η οποία θα δημιουργήσει νέα συμφραζόμενα, ωθώντας τον δείκτη ℓ στα τρέχοντα συμφραζόμενα, παρόμοια με μια στοίβα.[4] Επομένως, ο συνδυασμός του μετασχηματισμού defunctionalization και του γενικευμένου νοηματικού μετασχηματισμού οδηγεί σε μια υλοποίηση push/enter για γλώσσες υψηλότερης τάξης με υποστήριξη μερικής εφαρμογής. Επίσης, ο Peyton Jones παρατηρεί ότι η διαφορά μεταξύ των μοντέλων push/enter και eval/apply είναι λιγότερο εμφανής σε υλοποιήσεις γλωσσών πρώτης τάξης και οδηγεί στην παραγωγή παρόμοιου κώδικα [137].

Ο γενικευμένος νοηματικός μετασχηματισμός δεν υποστηρίζει τοπικές δηλώσεις (π.χ., με χρήση `let` ή `where` στη Haskell), όλες δηλαδή οι συναρτήσεις του προγράμματος δηλώνονται στο ανώτερο επίπεδο του προγράμματος (*supercombinators* [126]). Η χρήση τοπικών δηλώσεων είναι πολύ χρήσιμη στην πράξη στον συναρτησιακό προγραμματισμό· στις υλοποιήσεις που θα περιγραφούν στα επόμενα κεφάλαια, χρησιμοποιούμε την τεχνική lambda-lifting [134], που μετατρέπει τις τοπικές δηλώσεις του αρχικού προγράμματος σε δηλώσεις νέων συναρτήσεων στο ανώτερο επίπεδο του προγράμματος.[5] Ο συνδυασμός των μετασχηματισμών defunctionalization και lambda-lifting πρόσφατα (2014) προτάθηκε και από τους Trancón y Widemann και Lepper, σε εργασία τους πάνω στη σημασιολογία των γλωσσών ροής δεδομένων [301].

Περίληψη

Σε αυτό το κεφάλαιο δείξαμε πώς η υποστήριξη αυθαίρετων τύπων δεδομένων και το ταίριασμα προτύπων μπορούν να προστεθούν με φυσικό τρόπο στον νοηματικό μετασχηματισμό και στη νοηματική του γλώσσα, επιτρέποντας τη χρήση μη αυστηρών δομών δεδομένων και εκφράσεων υψηλότερης τάξης στον νοηματικό προγραμματισμό και στον προγραμματισμό ροής δεδομένων μονάδων με ετικέτες.

Στο επόμενο κεφάλαιο θα δούμε πώς λειτουργεί στην πράξη μια νοηματική υλοποίηση που εκτελεί προγράμματα γραμμένα στη γλώσσα NVIL του γενικευμένου νοηματικού μετασχηματισμού.

[4] Η υποστήριξη αναδρομικών συναρτήσεων από τον νοηματικό μετασχηματισμό προϋποθέτει μια μορφή στοίβας από την υλοποίηση, κάτι που είχαν ήδη εντοπίσει οι Ghosh *et al.* από το 1984 για τον προγραμματισμό ροής δεδομένων με ετικέτες [107].

[5] Περισσότερες λεπτομέρειες για την υλοποίηση του lambda-lifting στον γενικευμένο νοηματικό μετασχηματισμό είναι διαθέσιμες στην πτυχιακή εργασία του Παναγιώτη Θεοφιλόπουλου [274].

Κεφάλαιο 4

Υλοποίηση με αποθήκη

Στο προηγούμενο κεφάλαιο παρουσιάσαμε τον γενικευμένο νοηματικό μετασχηματισμό με τυπικό τρόπο· σε αυτό το κεφάλαιο θα δούμε πώς λειτουργεί μια πραγματική υλοποίηση.[1]

Θα περιγράψουμε δύο χρήσιμους μηχανισμούς της νοηματικής υλοποίησης: (α) την αποθήκη (Ενότητα 4.1), η οποία διατηρεί ήδη υπολογισμένες τιμές, και (β) τον πίνακα δέσμευσης συμφραζομένων (Ενότητα 4.2), που αναπαριστά τα νοηματικά συμφραζόμενα. Θα αναφέρουμε επίσης μια βελτιστοποίηση που βασίζεται σε μια απλή στατική ανάλυση μοιράσματος (Ενότητα 4.3). Στη συνέχεια θα αναφέρουμε τη σχέση της τεχνικής υλοποίησης που παρουσιάζεται σε αυτό το κεφάλαιο με τις αρχιτεκτονικές ροής δεδομένων μονάδων με ετικέτες (Ενότητα 4.4) και με υλοποιήσεις άλλων νοηματικών γλωσσών (Ενότητα 4.5).

4.1 Η αποθήκη

Όπως είδαμε στην Ενότητα 1.2, οι νοηματικές γλώσσες συνήθως βασίζονται στο υπολογιστικό μοντέλο eduction, το οποίο καθοδηγείται από τη ζήτηση τιμών κατά την εκτέλεση. Σε αυτό το μοντέλο, υπολογίζονται οι τιμές των μεταβλητών σε διαφορετικά συμφραζόμενα, ανάλογα με το αν χρειάζονται. Στην πράξη όμως, σε μια υλοποίηση, η ταχύτητα εκτέλεσης μπορεί να επηρεαστεί αρνητικά αν η τιμή μιας μεταβλητής στα ίδια συμφραζόμενα ζητηθεί πολλές φορές και κάθε φορά υπολογίζεται πάλι από την αρχή, καταλήγοντας στην ίδια τιμή. Για αυτόν τον λόγο, οι υλοποιήσεις του eduction συνήθως χρησιμοποιούν μια *αποθήκη* (*warehouse*), που είναι ένας μηχανισμός που αποθηκεύει τιμές που έχουν ήδη υπολογιστεί.

Η αποθήκη κρατά μόνο υπολογισμένες τιμές για τυπικές μεταβλητές. Δε χρειάζεται να κρατά τις υπολογισμένες τιμές μεταβλητών που αντιστοιχούν σε συναρτήσεις του αρχικού προγράμματος, αφού δεν πρόκειται να χρειαστούν: κάθε συνάρτηση γενικά έχει διαφορετικό δείκτη και θα υπολογιστεί σε διαφορετικά συμφραζόμενα.[2]

Στην Εικόνα 4.1 φαίνεται η σημασιολογία της NVIL όταν χρησιμοποιείται αποθήκη.[3] Σε σχέση με την αρχική σημασιολογία της NVIL (Εικόνα 3.1), παρατηρούμε ότι αλλάζει μόνο ο κανόνας για την αποτίμηση μιας μεταβλητής, ο οποίος εισάγει τη χρήση της αποθήκης W. Θεωρούμε ότι η αποθήκη είναι ένα σύνολο από τριάδες της μορφής (v, w, val), κάθε μια από τις οποίες αντιστοιχεί στην ήδη υπολογισμένη τιμή *val* μιας μεταβλητής v σε συμφραζόμενα w. Παρατηρούμε ότι η αποθήκη W δεν είναι παράμετρος της *EVAL* αλλά αποτελεί εξωτερικό αποθηκευτικό χώρο στον οποίο επιτρέπονται οι παρενέργειες και ο οποίος λειτουργεί σαν

[1] Το κεφάλαιο περιλαμβάνει υλικό που παρουσιάστηκε το 2013 σε άρθρο των Γ. Φουρτούνη, Ν. Παπασπύρου και Π. Ροντογιάννη [97].

[2] Για λόγους απλότητας, αγνοούμε την περίπτωση δύο κλήσεις να έχουν τον ίδιο δείκτη, όπως στην Ενότητα 3.3.2.

[3] Έχουμε υλοποιήσει έναν οκνηρό διερμηνέα με αποθήκη για τη γλώσσα NVIL, ως μέρος του GIC, της υλοποίησής μας που είναι διαθέσιμη στη διεύθυνση `https://github.com/gfour/gic`.

$$
\begin{aligned}
&\mathit{EVAL}_p(c(e_0,\ \ldots,\ e_{n-1}), w) \ = \ c(\mathit{EVAL}_p(e_0, w), \ldots, \mathit{EVAL}_p(e_{n-1}, w))\\
&\mathit{EVAL}_p(f, w) \ = \ \mathit{EVAL}_p(\mathit{body}(f, p), w), \text{εάν η } f \text{ είναι συνάρτηση}\\
&\mathit{EVAL}_p(f, w) \ = \ \mathit{val}, \text{εάν η } f \text{ είναι τυπική μεταβλητή και } (f, w, \mathit{val}) \in W\\
&\mathit{EVAL}_p(f, w) \ = \ W(f, w) \mapsto \mathit{EVAL}_p(\mathit{body}(f, p), w)\\
&\mathit{EVAL}_p(\kappa, w) \ = \ \langle \kappa, w \rangle\\
&\mathit{EVAL}_p(\texttt{case}\ e\ \texttt{of}\ \{\kappa_0 \to e_0;\ \ldots;\ \kappa_n \to e_n\}, \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e_i, \langle \ell, w, w' : \mu \rangle)\\
&\qquad \text{εάν } \mathit{EVAL}_p(e, \langle \ell, w, \mu \rangle) = \langle \kappa_i, w' \rangle\\
&\mathit{EVAL}_p(\#^m(e), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e, \mu_m)\\
&\mathit{EVAL}_p(\texttt{call}_\ell(e), w) \ = \ \mathit{EVAL}_p(e, \langle \ell, w, \bullet \rangle)\\
&\mathit{EVAL}_p(\texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}), \langle \ell, w, \mu \rangle) \ = \ \mathit{EVAL}_p(e_\ell, w)
\end{aligned}
$$

Εικόνα 4.1: Σημασιολογία της NVIL με αποθήκη.

μηχανισμός απομνημόνευσης κάποιων αποτελεσμάτων της *EVAL*. Η έκφραση $W(f, w) \mapsto$ *val* απομνημονεύει μια τριάδα (f, w, val) στην αποθήκη και επιστρέφει την τιμή *val*.

Ανακύκλωση της μνήμης της αποθήκης. Οι υπολογιστές έχουν πεπερασμένο μέγεθος μνήμης και μια αποθήκη συχνά γεμίζει από υπολογισμένες τιμές κατά τη διάρκεια της εκτέλεσης ενός νοηματικού προγράμματος, με αποτέλεσμα να είναι αναγκαίος ένας μηχανισμός χειρισμού της μνήμης. Για αυτόν τον λόγο, η αποθήκη πρέπει συχνά να αποδεσμεύει μέρος του χώρου μνήμης που χρησιμοποιεί, ώστε να μπορεί το πρόγραμμα που εκτελείται να συνεχίσει. Υπάρχουν δύο κύριες στρατηγικές για την επαναχρησιμοποίηση της μνήμης από μια αποθήκη:

- **Σχέδιο συνταξιοδότησης (retirement plan).** Κάθε αποθηκευμένη τιμή έχει μια *ηλικία συνταξιοδότησης* (*retirement age*) [88]. Όταν πρέπει να αποδεσμευτεί μνήμη, γίνεται αναζήτηση στην αποθήκη για τιμές που έχουν πολύ μεγάλη ηλικία συνταξιοδότησης, οι οποίες και μπορούν να διαγραφούν και, αν χρειαστούν στο μέλλον, να υπολογιστούν πάλι. Αυτή η στρατηγική υποθέτει ότι γενικά μια τιμή μπορεί να διαγραφεί και να ξαναϋπολογιστεί αν χρειαστεί. Η αποθήκη συμπεριφέρεται δηλαδή σαν μια μεγάλη κρυφή μνήμη (cache) που κρατά μόνο τις τιμές για ζεύγη μεταβλητών και συμφραζομένων που χρησιμοποιήθηκαν πρόσφατα. Το πλεονέκτημα αυτής της στρατηγικής είναι ότι πάντα μπορεί να αποδεσμεύσει χώρο στην αποθήκη, με κόστος όμως επανυπολογισμού ήδη γνωστών αποτελεσμάτων. Η στρατηγική αυτή αποτελούσε καθιερωμένη λύση διαχείρισης μνήμης σε υλοποιήσεις της Lucid [88, 290] και χρησιμοποιήθηκε και σε υλοποιήσεις του νοηματικού μετασχηματισμού [113, 240]. Γενικά, αυτή η προσέγγιση αφήνει την υλοποίηση ελεύθερη να επιλέξει διάφορες στρατηγικές έξωσης από την κρυφή μνήμη (cache eviction) [230, 307].

- **Συλλογή σκουπιδιών (garbage collection).** Ένας συλλέκτης σκουπιδιών [136] μπορεί να αναλάβει τη διαχείριση της μνήμης της αποθήκης και να διαγράφει μόνο τιμές σε συμφραζόμενα που δεν είναι πια προσβάσιμα από τα τρέχοντα συμφραζόμενα, και άρα δεν πρόκειται να χρησιμοποιηθούν ξανά. Μια παρόμοια προσέγγιση ακολουθείται στις περισσότερες υλοποιήσεις συναρτησιακών γλωσσών προγραμματισμού. Το πλεονέκτημά της είναι ότι ποτέ δε διαγράφει μια υπολογισμένη τιμή που μπορεί να χρησιμοποιηθεί ξανά. Το μειονέκτημά της είναι ότι μπορεί να αποτύχει να αποδεσμεύσει μνήμη αν δε βρεθούν μη προσβάσιμα συμφραζόμενα.

Σχέση με την οκνηρή αποτίμηση. Στον συναρτησιακό προγραμματισμό με μη αυστηρή σημασιολογία και κλήση κατ' όνομα, οι υλοποιήσεις καλούνται να λύσουν ακριβώς το ίδιο

πρόβλημα του επανυπολογισμού τιμών· για αυτό το λόγο, οι περισσότερες υλοποιήσεις τέτοιων γλωσσών επιλέγουν μια οκνηρή στρατηγική αποτίμησης.[4] Σε μια οκνηρή υλοποίηση της FOFL, μια παράμετρος συνάρτησης ή κατασκευαστή θα αναπαρίσταται με ένα thunk· την πρώτη φορά που ζητείται η τιμή της παραμέτρου, αυτή θα υπολογιστεί και θα τοποθετηθεί στο thunk, ενώ τις επόμενες φορές που θα χρειαστεί, μπορεί να διαβαστεί κατευθείαν από εκεί. Στην υλοποίηση που περιγράφουμε, είπαμε ότι οι παράμετροι συναρτήσεων είναι οι μόνες μεταβλητές των οποίων οι τιμές διατηρούνται. Αυτές οι μεταβλητές αντιστοιχούν ακριβώς στις παραμέτρους συναρτήσεων και κατασκευαστών στην FOFL. Επίσης, η τιμή μιας τυπικής μεταβλητής σε κάποια συμφραζόμενα αντιστοιχεί στην τιμή της ίδιας μεταβλητής σε μια κλήση συνάρτησης στο αρχικό πρόγραμμα. Επομένως, η αποθήκη που περιγράψαμε παραπάνω μπορεί να μιμηθεί την οκνηρή αποτίμηση των συναρτησιακών προγραμμάτων, αρκεί να επιλεγεί η συλλογή σκουπιδιών ως στρατηγική ανακύκλωσης της μνήμης.

Σε αυτήν τη διατριβή δε θα χρησιμοποιήσουμε τη στρατηγική σχεδίου συνταξιοδότησης γιατί είναι δύσκολο να ελέγξουμε το κόστος των υπολογισμών που επαναλαμβάνονται. Θα επιλέξουμε επομένως τη συλλογή σκουπιδιών, η οποία έχει τη γνώριμη συμπεριφορά των οκνηρών υλοποιήσεων.

4.2 Αναπαράσταση των συμφραζομένων

Τα συμφραζόμενα του γενικευμένου νοηματικού μετασχηματισμού που παρουσιάστηκαν στην Ενότητα 3.2 πρέπει να αναπαριστώνται από την υλοποίηση με τρόπο που να επιτρέπει τον αποδοτικό χειρισμό και τη γρήγορη αποθήκευσή τους. Σε αυτήν την ενότητα θα παρουσιαστεί ένας μηχανισμός για τη δημιουργία νέων συμφραζομένων και για τη διαχείρισή τους, με μια κωδικοποίηση που δεν σπαταλά χώρο.

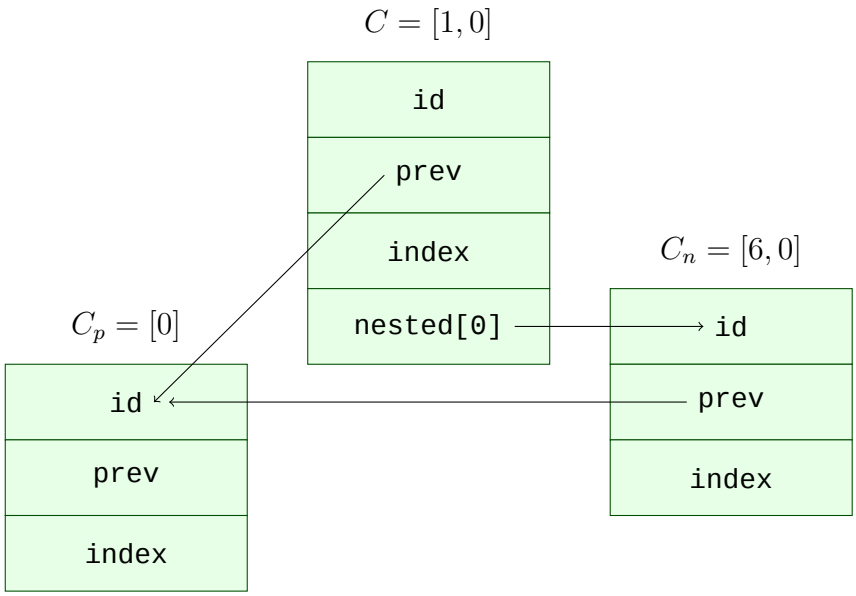
Στον γενικευμένο νοηματικό μετασχηματισμό, τα συμφραζόμενα αναπαριστώνται ως μία λίστα που μπορεί να περιέχει δείκτες προς άλλα συμφραζόμενα. Στην υλοποίησή μας, τα συμφραζόμενα έχουν τέσσερα πεδία: ένα μοναδικό αναγνωριστικό *id*, έναν δείκτη *prev* προς τα προηγούμενα συμφραζόμενα από τα οποία έγινε κλήση συνάρτησης, την αρχική νοηματική ετικέτα *index*, και τον πίνακα *nested* που χρειάζεται για το ταίριασμα προτύπων και δείχνει σε άλλα, εμφωλευμένα συμφραζόμενα.

Για παράδειγμα, έστω τα εξής τρία συμφραζόμενα: $C_p = [0]$, $C = [1, 0]$, και $C_n = [6, 0]$, όπου C_n είναι τα μοναδικά εμφωλευμένα συμφραζόμενα του C και τα δύο συμφραζόμενα C και C_n έχουν δημιουργηθεί μετά το C_p. Η υλοποίηση θα δώσει σε κάθε περίπτωση συμφραζομένων ένα μοναδικό αριθμητικό *id*, το οποίο και θα χρησιμοποιείται για να υποδηλώσει κάποια συμφραζόμενα σε οποιοδήποτε πεδίο των τετράδων που αποθηκεύονται σε έναν *πίνακα δέσμευσης συμφραζομένων* (*context allocation table*). Η σχέση μεταξύ των συμφραζομένων και των πραγματικών περιεχομένων του πίνακα φαίνεται στην Εικόνα 4.2.

Εδώ πρέπει να σημειωθεί ότι επειδή η αποτίμηση του προγράμματος πάντα ξεκινά από τη μεταβλητή *result* στα κενά συμφραζόμενα `[]`, το μέγεθος του *nested* σε αυτά τα συμφραζόμενα πρέπει να είναι ίσο με το βάθος ταιριάσματος προτύπων N της *result*.

Η παραπάνω αναπαράσταση είναι μια απλοποιημένη έκδοση της τεχνικής με συνένωση-κατακερματισμό (hash-consing) που ακολουθήθηκε από τον Ροντογιάννη και τον Γρίβα στις υλοποιήσεις τους του κλασικού νοηματικού μετασχηματισμού [113, 240], που στην πράξη είχε μέτρια ταχύτητα [55]. Η τεχνική hash-consing [9] αναπαριστά δομές που μοιράζονται στοιχεία με οικονομικό τρόπο στη μνήμη αλλά δεν είναι δυνατό να χρησιμοποιηθεί στον γε-

[4] Αυτό είναι τόσο συνηθισμένο που γλώσσες με μη αυστηρή σημασιολογία όπως η Haskell, συχνά αποκαλούνται οκνηρές γλώσσες προγραμματισμού (lazy programming languages).



Συμφραζόμενα	*id*	*prev*	*index*	*nested*
[ ]	0	—	—	πίνακας με N πεδία
C_p	1	0	0	—
C	2	1	1	[3]
C_n	3	1	6	—

Εικόνα 4.2: Τρία νοηματικά συμφραζόμενα και ο πίνακας δέσμευσης συμφραζομένων που τα περιέχει.

νικευμένο νοηματικό μετασχηματισμό. Αυτό οφείλεται στο ότι η τεχνική λειτουργεί όταν τα ίδια συμφραζόμενα χρησιμοποιούνται από τον τελεστή `call`, κάτι που συνέβαινε συχνά στον κλασικό νοηματικό μετασχηματισμό αλλά δε συμβαίνει στον γενικευμένο μετασχηματισμό, μιας και τα συμφραζόμενα πια διατηρούν εσωτερικά πληροφορία στο πεδίο *nested*, η οποία συμπληρώνεται μετά τη δημιουργία των συμφραζομένων κατά την κλήση συνάρτησης.

Ο πίνακας των συμφραζομένων πρέπει επίσης να διαχειρίζεται τη μνήμη που χρησιμοποιεί και να διαγράφει συμφραζόμενα αν η μνήμη δεν επαρκεί για την εκτέλεση του προγράμματος. Αυτό πρέπει να γίνεται προσεκτικά, ώστε να μη διαγράφονται συμφραζόμενα που είναι προσβάσιμα από την τρέχουσα κατάσταση του προγράμματος.

Αν η αποθήκη θεωρηθεί μια αφηρημένη μοντελοποίηση της μνήμης, η κωδικοποίηση των συμφραζομένων που παρουσιάσαμε είναι ο μοναδικός τρόπος να δημιουργούνται “διευθύνσεις μνήμης”. Δεδομένου ότι η υλοποίηση με αποθήκη δε χρειάζεται δείκτες προς κώδικα, η προσεκτική αναπαράσταση της αποθήκης μπορεί να χρησιμοποιηθεί ακόμα και σε ετερογενή (heterogeneous systems) ή σε κατανεμημένα συστήματα (distributed systems), όπου διαφορετικά μέρη ενός προγράμματος μπορεί να βλέπουν διαφορετικούς χώρους διευθύνσεων.

4.3 Ανάλυση μοιράσματος

Στην κλήση κατ’ ανάγκη, κάθε φορά που ζητείται η τιμή μιας τυπικής μεταβλητής, ελέγχεται αν αυτή έχει υπολογιστεί· αν ναι, τότε επιστρέφεται η τιμή, αλλιώς η τιμή υπολογίζεται,

αποθηκεύεται και επιστρέφεται στο σημείο που τη ζήτησε. Αυτός ο έλεγχος είναι περιττός αν μια τυπική μεταβλητή χρησιμοποιείται μόνο μια φορά – σε αυτήν την περίπτωση πρέπει κατευθείαν να υπολογιστεί η τιμή της χωρίς έλεγχο και αποθήκευση, δηλαδή πρέπει να αποτιμηθεί κατ' όνομα. Στην πράξη, έχει παρατηρηθεί ότι το 70% των εκφράσεων αναπαριστούν τιμές που θα χρησιμοποιηθούν μόνο μια φορά κατά τη διάρκεια της εκτέλεσης [165].

Αν και γενικά δεν μπορούμε να βρούμε αν μια τυπική μεταβλητή χρησιμοποιείται μια ή παραπάνω φορές, υπάρχει μια απλή και συντηρητική στατική ανάλυση μοιράσματος (sharing analysis) που αποδεικνύεται χρήσιμη στην πράξη. Όσες τυπικές μεταβλητές χρησιμοποιούνται το πολύ μια φορά στο σώμα μιας συνάρτησης στο αρχικό πρόγραμμα, μπορούν να αποτιμηθούν κατ' όνομα, γλιτώνοντας το κόστος ελέγχου της κατάστασής τους, καθώς και το κόστος αποθήκευσής τους. Με αυτόν τον τρόπο, στην υλοποίηση του νοηματικού μετασχηματισμού, μπορεί να παρακάμπτεται η αποθήκη κατά τον υπολογισμό ορισμένων τυπικών παραμέτρων.

Σε αυτό το σημείο, πρέπει να σημειώσουμε ότι στις συναρτήσεις `apply` του μετασχηματισμού defunctionalization, όλες οι τυπικές παράμετροι χρησιμοποιούνται ακριβώς μια φορά στο σώμα της συνάρτησης. Έτσι, η αποτίμηση των τυπικών παραμέτρων γίνεται πάντα κατ' όνομα όταν χρησιμοποιείται αυτή η βελτιστοποίηση.

Δε θα περιγράψουμε εδώ αυτήν την ανάλυση· περισσότερες λεπτομέρειες δίνονται από τους Fairbairn και Wray [86], οι οποίοι την περιγράφουν για την Three Instruction Machine (TIM), μια από τις πρώτες πειραματικές μηχανές αποδοτικής εκτέλεσης οκνηρών γλωσσών.

4.4 Νοηματικός μετασχηματισμός και αρχιτεκτονικές ροής δεδομένων

Όπως αναφέραμε στην Ενότητα 2.1.3, ο κλασικός νοηματικός μετασχηματισμός είχε προταθεί για την υλοποίηση συναρτησιακών γλωσσών στις αρχιτεκτονικές ροής δεδομένων της δεκαετίας του 1980 [17, 18, 155]. Η υλοποίηση που περιγράφουμε σε αυτό το κεφάλαιο συνεχίζει αυτές τις ιδέες και προσφέρεται επίσης για υλοποίηση σε αυτές τις αρχιτεκτονικές.

Στο μοντέλο υπολογισμού ροής δεδομένων, η εκτέλεση ενός προγράμματος αναλογεί στην παράλληλη ροή δεδομένων σε έναν *γράφο ροής δεδομένων* (*dataflow graph*) [74, 282]. Αυτός ο γράφος αποτελείται από *κόμβους* (*nodes*) που συνδέονται με *ακμές* (*arcs*) ή *κανάλια επικοινωνίας* (*communication channels*). Κάθε κόμβος μπορεί να έχει πολλές *θύρες* (*ports*) εισόδου και εξόδου, στις οποίες καταλήγουν ακμές. Ένα από τα πιο βασικά μοντέλα ροής δεδομένων είναι το μοντέλο ροής δεδομένων μονάδων με ετικέτες, όπου δεδομένα που μεταφέρονται διαμέσου των ακμών κουβαλούν *ετικέτες* (*tags*). Ένας κόμβος μπορεί να επεξεργαστεί σε μια δεδομένη στιγμή μόνο τα δεδομένα εισόδου που έχουν την ίδια ετικέτα, ώστε να παράγει νέα δεδομένα στις ακμές εξόδου του.

Ο μηχανισμός των μονάδων με ετικέτες επέτρεπε σε μια ακμή να περιέχει δεδομένα από διαφορετικές στιγμές της εκτέλεσης του προγράμματος, κάτι που στη βιβλιογραφία των γλωσσών ροής δεδομένων ονομάστηκε *χρωματισμός* (*coloring*) [72]. Αυτό έκανε εύκολη την αναπαράσταση τιμών από διαφορετικές κλήσεις της ίδιας αναδρομικής συνάρτησης (η οποία αντιστοιχούσε σε κάποιον κόμβο του γράφου) και υπήρξε βασικό στοιχείο προσπαθειών να μεταγλωττιστούν συναρτησιακές γλώσσες για υπολογιστές ροής δεδομένων [18, 116]. Υπήρξαν μάλιστα προσπάθειες να υλοποιηθούν γλώσσες υψηλότερης τάξης για αρχιτεκτονικές ροής δεδομένων, θεωρώντας ότι το υλικό περιλάμβανε ειδικούς κόμβους (παρόμοιους με τις συναρτήσεις χειρισμού κλεισιμάτων του μετασχηματισμού defunctionalization) που αναλάμβαναν την εφαρμογή των εκφράσεων υψηλότερης τάξης [128, 221, 277].

Σε αυτό το σημείο πρέπει να αναφέρουμε τη γλώσσα ροής δεδομένων Id, η οποία είχε αρχικά υλοποιηθεί σε αρχιτεκτονικές ροής δεδομένων με ετικέτες [194], στη συνέχεια οδήγησε στην ανάπτυξη της pHluid [91] και της Parallel Haskell (pH) [50, 196], οι οποίες στόχευαν στην υλοποίηση σε παράλληλο υλικό υπολογιστών αρχιτεκτονικής von Neumann. Η pH είχε τη σύνταξη της Haskell και ήταν μη αυστηρή αλλά ακολουθούσε μια παράλληλη στρατηγική αποτίμησης (parallel evaluation ή lenient evaluation) [2], η οποία διαφέρει αρκετά από την οκνηρή αποτίμηση [250]. Σε αυτήν τη στρατηγική αποτίμησης, οι παράμετροι των συναρτήσεων μπορούν να αποτιμηθούν παράλληλα, ακόμα και αν δεν χρειάζεται η τιμή τους εκείνη τη στιγμή· η υλοποίηση εξασφαλίζει ότι άχρηστοι υπολογισμοί που γίνονται παράλληλα δεν επηρεάζουν το τελικό αποτέλεσμα του προγράμματος. Στα πρώτα στάδια της ανάπτυξης της Id είχε επίσης γίνει προσπάθεια να υλοποιηθούν οκνηρές δομές δεδομένων στον προγραμματισμό ροής δεδομένων [121].

Η τεχνική που έχουμε παρουσιάσει μέχρι τώρα δείχνει πώς μια μη αυστηρή (ή οκνηρή) γλώσσα προγραμματισμού μπορεί να υλοποιηθεί σε αυτό το μοντέλο ροής δεδομένων, χρησιμοποιώντας την ιδέα του χρωματισμού, χωρίς ειδικούς κόμβους στον γράφο του προγράμματος και υποστηρίζοντας με φυσικό τρόπο εκφράσεις υψηλότερης τάξης, κατασκευαστές, και ταίριασμα προτύπων.

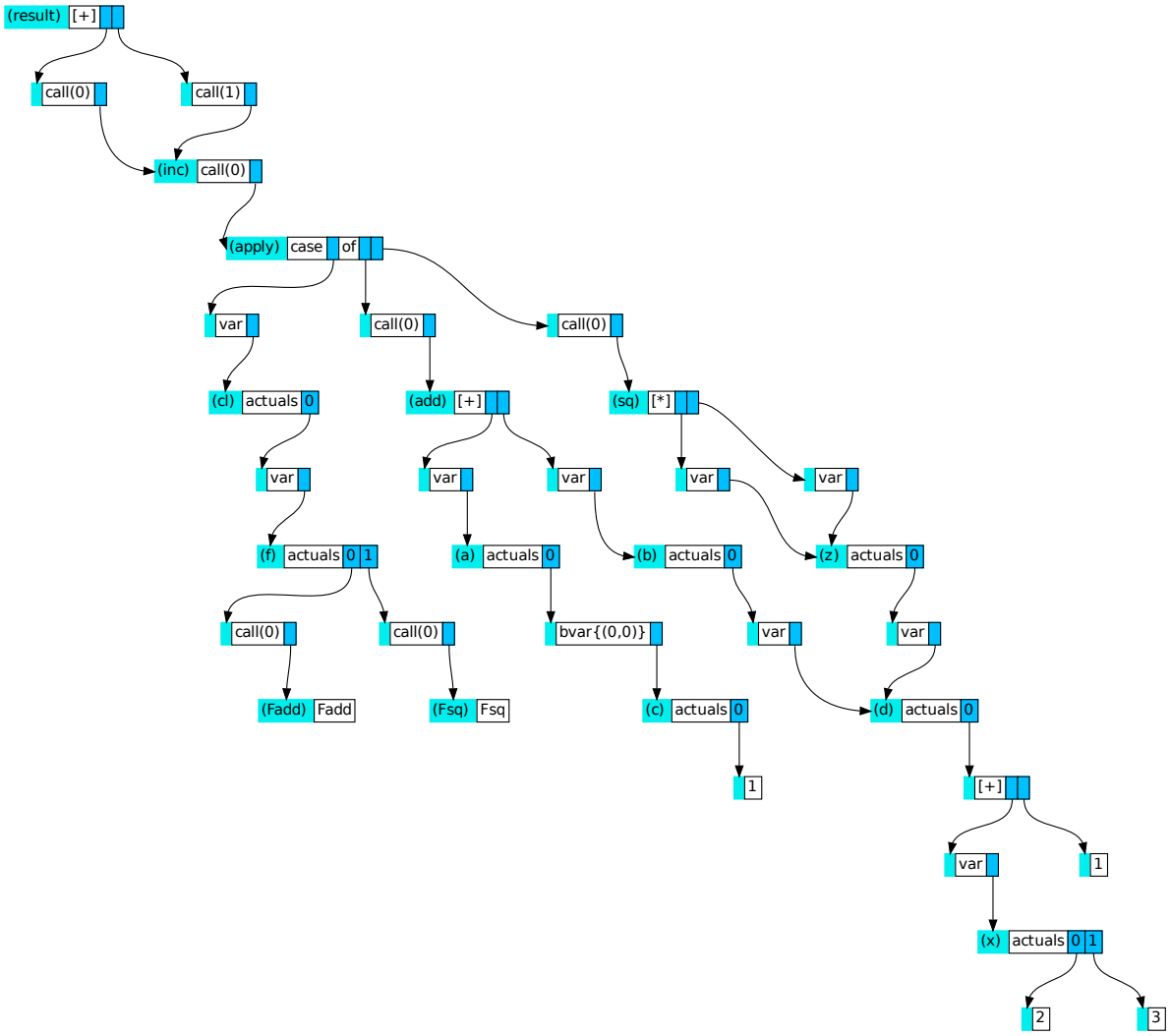
Τα προγράμματα που παράγει ο γενικευμένος νοηματικός μετασχηματισμός μπορούν να αναπαρασταθούν ως γράφοι ροής δεδομένων: για παράδειγμα, το πρόγραμμα NVIL της Ενότητας 3.2 μπορεί να αναπαρασταθεί όπως φαίνεται στην Εικόνα 4.3. Κάθε κόμβος αυτού του γράφου αποτελεί μια εντολή ροής δεδομένων που συνδέεται με άλλες εντολές μέσω ακμών και ανταλλάσσει αιτήματα για ζεύγη μεταβλητών-συμφραζομένων και υπολογισμένες τιμές. Στην τεχνική υλοποίησης που περιγράψαμε σε αυτό το κεφάλαιο, η αποθήκη και ο πίνακας συμφραζομένων, μαζί με τον μηχανισμό διαχείρισης μνήμης, είναι η μοναδική μνήμη που χρειάζεται για την εκτέλεση ενός προγράμματος. Το ίδιο το πρόγραμμα δεν αλλάζει ποτέ και μπορεί να αναπαρασταθεί ως ένας αμετάβλητος γράφος ροής δεδομένων (όπως αυτός της Εικόνας 4.3), με τους κόμβους του να επικοινωνούν μέσω τιμών με ετικέτες. Αυτές οι τιμές που περιλαμβάνουν ετικέτες αντιστοιχούν απευθείας στα ζεύγη μεταβλητών και συμφραζομένων του νοηματικού μετασχηματισμού. Αυτή είναι μια σημαντική διαφορά της τεχνικής μας σε σχέση με τις υλοποιήσεις συναρτησιακών γλωσσών προγραμματισμού που είναι βασισμένες στην αναγωγή γράφου (graph reduction) [216], όπου το πρόγραμμα θεωρείται γράφος που μετασχηματίζεται μέχρι να βρεθεί σε μια κανονική μορφή, η οποία και θα είναι το αποτέλεσμά του.

Οι δύο μηχανισμοί που περιγράφηκαν σε αυτό το κεφάλαιο αντιστοιχούν σε υποσυστήματα των αρχιτεκτονικών ροής δεδομένων:

1. Ο χειρισμός των οκνηρών εκφράσεων μέσω της αποθήκης έχει ομοιότητες με τις I-δομές (I-structures) [19, 262]. Οι I-δομές χρησιμοποιήθηκαν στον κλασικό προγραμματισμό ροής δεδομένων για να παρέχουν ασφαλή ταυτόχρονη αποτίμηση δομών δεδομένων εξασφαλίζοντας οκνηρή αποτίμηση· αν και η υλοποίηση που παρουσιάσαμε σε αυτό το κεφάλαιο δεν είναι παράλληλη, η αποθήκη εξασφαλίζει την οκνηρή αποτίμηση με παρόμοιο μηχανισμό με τις I-δομές.

2. Ο μηχανισμός αναπαράστασης συμφραζομένων μοιάζει με τον μηχανισμό δέσμευσης μνήμης που περιέγραψαν οι Arvind και Culler [17], ο οποίος χρησιμοποιήθηκε στην αρχιτεκτονική ροής δεδομένων Monsoon [65, 206].

Ιστορικά, αν και αυτές οι αρχιτεκτονικές ροής δεδομένων έδειχναν θεαματικά αποτελέσματα όταν προγραμματίζονταν με γλώσσες ροής δεδομένων, δεν μπορούσαν να εκτελέσουν



Εικόνα 4.3: Το πρόγραμμα NVIL της Ενότητας 3.2, ως γράφος ροής δεδομένων.

αποδοτικά τις δημοφιλείς γλώσσες προγραμματισμού της εποχής και ήταν δύσκολο να μεταφερθούν υλοποιήσεις γλωσσών σε αυτές [16, 154]. Από την άλλη πλευρά, οι κλασικοί υπολογιστές με επεξεργαστές αρχιτεκτονικής von Neumann εκμεταλλεύτηκαν τον νόμο του Moore για να επικρατήσουν λόγω της ταχύτητάς τους [64, 182]. Οι αρχιτεκτονικές ροής δεδομένων εξακολουθούν και σήμερα να είναι ενεργό ερευνητικό πεδίο με εξειδικευμένες εφαρμογές [104, 267] και το μοντέλο υπολογισμού τους επιστρέφει ως πιθανή απάντηση σε προβλήματα παράλληλου προγραμματισμού. Σε αυτή τη διατριβή δε θα ασχοληθούμε άλλο με υλοποιήσεις που απαιτούν υλικό ροής δεδομένων· περισσότερες πληροφορίες πάνω σε αυτό το αντικείμενο και πιθανές ερευνητικές κατευθύνσεις αναφέρονται στην Ενότητα 8.3.2 του επιλόγου.

4.5 Σχετικές νοηματικές τεχνικές υλοποίησης

Παρόμοιες τεχνικές υλοποίησης έχουν χρησιμοποιηθεί και σε άλλα συστήματα νοηματικού προγραμματισμού. Ειδικότερα, το σύστημα νοηματικού προγραμματισμού GIPSY [179, 209, 270, 281] έχει αναπτύξει μια κατανεμημένη αποθήκη μεγάλης κλίμακας, ενώ η νοηματική γλώσσα TransLucid περιέχει επίσης μια αποθήκη [222] που ο Ditu απέδειξε [77] ότι περιέχει την ελάχιστη απαιτούμενη πληροφορία που χρειάζεται για την εκτέλεση ενός προ-

γράμματος. Η TransLucid έχει επίσης μια πολυνηματική υλοποίηση που χρησιμοποιεί αποθήκη [229]. Στη διπλωματική του [113], ο Γρίβας περιέγραψε την υλοποίηση μιας αποθήκης για τον νοηματικό μετασχηματισμό υψηλότερης τάξης. Σε σχετική δημοσίευση του 2011, οι Φουρτούνης, Ölveczky και Παπασπύρου έδωσαν έναν φορμαλισμό για μια απλοποιημένη παράλληλη έκδοση της αποθήκης που περιγράψαμε, χρησιμοποιώντας το εργαλείο Maude που είναι βασισμένο στη λογική αναγραφής (rewriting logic) [94]. Οι παράλληλες υλοποιήσεις όμως δε θα μας απασχολήσουν άλλο σε αυτήν τη διατριβή, που θα περιοριστεί σε σειριακές υλοποιήσεις.

Περίληψη

Σε αυτό το κεφάλαιο είδαμε πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί ακολουθώντας ιδέες των νοηματικών υλοποιήσεων και των γλωσσών ροής δεδομένων.

Ο νοηματικός μετασχηματισμός έχει όμως υλοποιηθεί με επιτυχία στο παρελθόν και για κλασικό υλικό αρχιτεκτονικής von Neumann [55, 242]. Στο επόμενο κεφάλαιο θα παρουσιαστεί η υλοποίηση του γενικευμένου νοηματικού μετασχηματισμού για μια τέτοια αρχιτεκτονική, η οποία θα οδηγήσει σε χρήσιμα συμπεράσματα για το πώς λειτουργεί το μοντέλο μας σε σύγχρονες δημοφιλείς αρχιτεκτονικές υπολογιστών.

Κεφάλαιο 5

Υλοποίηση με εγγραφές δραστηριοποίησης

Σε αυτό το κεφάλαιο θα δούμε πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί στην πλατφόρμα x86, μια διαδεδομένη αρχιτεκτονική υλικού για σύγχρονους υπολογιστές (πρότυπα IA-32 [130]) και AMD64 [169]), η οποία δεν προβλέπει ειδική υποστήριξη για νοηματικές γλώσσες ή γλώσσες ροής δεδομένων.[1]

Η τεχνική που περιγράφεται σε αυτό το κεφάλαιο βασίζεται στην αντίστοιχη τεχνική των Ροντογιάννη και Wadge [242], η οποία στη συνέχεια βελτιώθηκε από τους Χαραλαμπίδη, Γρίβα, Παπασπύρου και Ροντογιάννη [55], για την αποδοτική υλοποίηση του νοηματικού μετασχηματισμού στην πλατφόρμα αυτή. Η βασική ιδέα αυτής της τεχνικής είναι ότι τα συμφραζόμενα μιας συνάρτησης μπορούν να αναπαρασταθούν από μια εγγραφή δραστηριοποίησης (activation record) [7], οι παράμετροι της οποίας όμως υπολογίζονται όταν χρειάζονται, ακολουθώντας το μοτίβο της οκνηρής αποτίμησης.

Στη συνέχεια θα εξετάσουμε την κωδικοποίηση αυτή των συμφραζομένων (Ενότητα 5.1) και των παραμέτρων που αποτιμούνται κατ' ανάγκη (Ενότητα 5.2). Θα περιγράψουμε μια βελτιστοποίηση που βασίζεται σε μια απλή στατική ανάλυση (Ενότητα 5.3) και θα αναφέρουμε επιπλέον μέρη που χρειάστηκαν για την υλοποίηση (Ενότητα 5.4). Τέλος, θα συγκρίνουμε στην πράξη πώς η υλοποίησή μας συγκρίνεται με άλλες δημοφιλείς υλοποιήσεις της Haskell (Ενότητα 5.5) και θα αναφέρουμε σχετική έρευνα με την τεχνική μας (Ενότητα 5.6).

5.1 Οκνηρές εγγραφές δραστηριοποίησης

Σε αυτήν την ενότητα θα περιγράψουμε το πώς η υλοποίηση αναπαριστά τα συμφραζόμενα στη μνήμη. Η βασική ιδέα της τεχνικής της υλοποίησης είναι ότι για κάθε ορισμό του νοηματικού προγράμματος σε NVIL, παράγεται ένα κομμάτι κώδικα σε C, το οποίο παίρνει ως παράμετρο τα τρέχοντα συμφραζόμενα. Στην πράξη, αυτός ο κώδικας C είναι μια πιο αποδοτική υλοποίηση της συνάρτησης *EVAL* που φαίνεται στην Εικόνα 3.1. Το σύστημα χρόνου εκτέλεσης (runtime system) χρησιμοποιεί μια στοίβα (stack) και έναν σωρό (heap). Σε αντίθεση με άλλες τεχνικές που αναπαριστούν τα δεδομένα των κατασκευαστών ως αντικείμενα στον σωρό, τα μόνα αντικείμενα που αποθηκεύονται στη μνήμη (στη στοίβα και στον σωρό) είναι *οκνηρές εγγραφές δραστηριοποίησης* (ΟΕΔ) (*Lazy Activation Records* ή LARs) [55]. Μια ΟΕΔ δημιουργείται όταν κατά την εκτέλεση του προγράμματος συναντάται μια έκφραση της μορφής $\texttt{call}_\ell(f)$. Οι ΟΕΔ μοιάζουν με τις κλασικές εγγραφές δραστηριοποίησης, στις οποίες αποθηκεύονται οι παράμετροι των συναρτήσεων. Κάποια όμως από τα πεδία μιας ΟΕΔ δεν συμπληρώνονται κατά την κλήση της συνάρτησης, αλλά μόνο όταν η τιμή τους απαιτηθεί κατά τη διάρκεια της εκτέλεσης του προγράμματος. Επιπλέον, όταν η τιμή μιας τυπικής μεταβλητής ζητηθεί *πάλι* στα ίδια συμφραζόμενα, η ήδη υπολογισμένη τιμή

[1] Το κεφάλαιο περιλαμβάνει υλικό που παρουσιάστηκε το 2013 σε άρθρο των Γ. Φουρτούνη, Ν. Παπασπύρου και Π. Ροντογιάννη [97], καθώς και υλικό από άρθρο του 2014 των Γ. Φουρτούνη και Ν. Παπασπύρου [95].

της διαβάζεται από την ΟΕΔ. Αυτό σημαίνει ότι οι ΟΕΔ υλοποιούν σημασιολογία κλήσης κατ' ανάγκη και αποτελούν έναν εναλλακτικό τρόπο να αναπαρίσταται και να χρησιμοποιείται η αποθήκη του Κεφαλαίου 4.

Μια ΟΕΔ αντιστοιχεί σε συμφραζόμενα της μορφής $w = \langle \ell, w', \mu \rangle$ στον ορισμό της συνάρτησης *EVAL* στην Εικόνα 3.1. Αναλυτικότερα, περιέχει τα εξής πεδία:

- *prev*: δείκτης στην προηγούμενη ΟΕΔ, δηλαδή στην ΟΕΔ της συνάρτησης που κάλεσε την τρέχουσα συνάρτηση (*access link* [7]). Αντιστοιχεί στην παράμετρο w'.

- $arg_0, \ldots, arg_{n-1}$: κάθε arg_i είναι δείκτης προς κώδικα που πρόκειται να υπολογίσει την τυπική παράμετρο στη θέση i της κλήσης συνάρτησης που κατασκεύασε αυτήν την ΟΕΔ. Πρόκειται για κωδικοποίηση της παραμέτρου ℓ της τυπικής σημασιολογίας της NVIL.

- $val_0, \ldots, val_{n-1}$: κάθε val_i αποθηκεύει την τιμή της αντίστοιχης παραμέτρου arg_i. Αρχικά τα πεδία είναι άδεια και συμπληρώνονται όταν χρειάζεται: αν κάποια στιγμή ο κώδικας, στον οποίο δείχνει η arg_i, εκτελεστεί και επιστρέψει κάποια τιμή, αυτή η τιμή θα αποθηκευτεί στην val_i για μελλοντική χρήση. Με αυτόν τον τρόπο υλοποιούνται thunks με σημασιολογία κλήσης κατ' ανάγκη.

- *nested*: αυτό το πεδίο αντιστοιχεί απευθείας στο πεδίο μ της σημασιολογίας και είναι ένας πίνακας που αποθηκεύει τις τιμές των εκφράσεων που χρησιμοποιούνται σε εμφωλευμένες δομές `case`. Ειδικότερα, όταν μια έκφραση της μορφής $\#^m(e)$ συναντάται σε κάποιο σώμα ταιριάσματος προτύπου, το πεδίο *nested*$[m]$ δείχνει στην ΟΕΔ που θα χρησιμοποιηθεί για να υπολογιστεί η e. Το πεδίο αυτό χρησιμοποιείται δηλαδή για να προσπελαστούν τιμές που αρχικά δημιουργήθηκαν σε άλλα περιβάλλοντα.[2]

Συνοψίζοντας, ένα πρόγραμμα NVIL μεταγλωττίζεται σε κώδικα C ακολουθώντας πιστά τη σημασιολογία της $EVAL_p$ που έχει δοθεί στην Εικόνα 3.1.

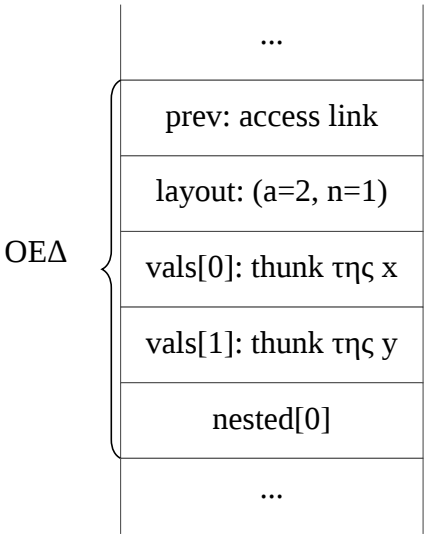
Μια σημαντική διαφορά της τεχνικής μας σε σχέση με άλλες κλασικές τεχνικές υλοποίησης μη αυστηρών συναρτησιακών γλωσσών είναι η απουσία *κλεισιμάτων* (closures). Σε κλασικές υλοποιήσεις της κλήσης κατ' ανάγκη, το πεδίο arg_i θα περιείχε ένα κλείσιμο που αποτελείται από: (α) έναν δείκτη προς τον κώδικα που υπολογίζει την παράμετρο στη θέση i, και (β) ένα περιβάλλον (environment) που δίνει τις τιμές των δεσμευμένων μεταβλητών που θα χρησιμοποιήσει ο κώδικας. Αντίθετα, στη δική μας υλοποίηση, η παράμετρος arg_i είναι απλά ένας δείκτης προς κώδικα. Το περιβάλλον δεν υπάρχει πια ρητά, μιας και ο νοηματικός μετασχηματισμός το έχει κωδικοποιήσει μέσω των συμφραζομένων (δείκτης προς μια ΟΕΔ) που θα περαστούν ως παράμετρος κατά την κλήση της arg_i. Όλες οι μεταβλητές αντιστοιχούν στους ορισμούς μηδενικής τάξης της NVIL και τα συμφραζόμενα καθοδηγούν την εκτέλεση του προγράμματος και οδηγούν στον υπολογισμό των σωστών τιμών για αυτές τις μεταβλητές.

Για παράδειγμα, έστω η εξής συνάρτηση `f`, η οποία δέχεται δύο παραμέτρους και περιλαμβάνει μια έκφραση ταιριάσματος προτύπων:

```
f x y = case x of
          []   → [1]
          a:as → [a + y]
```

Όταν κληθεί η `f`, θα δημιουργηθεί μια ΟΕΔ που θα περιέχει τον δείκτη *prev*, δύο thunk με τις παραμέτρους, και ένα πεδίο *nested*, όπως φαίνεται στην Εικόνα 5.1. Στην πράξη, η ΟΕΔ θα

[2] Στη βιβλιογραφία των μεταγλωττιστών, ένα τέτοιο πεδίο ονομάζεται *display structure* [7].



Εικόνα 5.1: Η οκνηρή εγγραφή δραστηριοποίησης (ΟΕΔ) μιας συνάρτησης που παίρνει δύο παραμέτρους και έχει μια έκφραση ταιριάσματος προτύπων.

περιέχει επίσης μεταδεδομένα (πεδίο *layout*) που περιγράφουν πόσες παραμέτρους (a) και πόσα πεδία *nested* (n) έχει, ώστε να μπορεί να τη διαβάσει σωστά κάποιος ακριβής συλλέκτης σκουπιδιών.

5.2 Υλοποίηση της κλήσης κατ' ανάγκη

Για κάθε τυπική παράμετρο μιας ΟΕΔ, κρατείται χώρος για το αντίστοιχο thunk, το οποίο αποτελεί τη δομή που εξασφαλίζει ότι η παράμετρος θα υπολογιστεί μόνο όταν χρειαστεί και το πολύ μια φορά. Ένα thunk έχει τρία πεδία:

σημαία: Ένα πεδίο με τιμές *Αληθές*/*Ψευδές* που δείχνει αν το thunk έχει ήδη υπολογιστεί.

κώδικας: Ένας δείκτης προς τον κώδικα που πρέπει να εκτελεστεί για να υπολογιστεί η τιμή του thunk.

τιμή: Μια αποθηκευμένη τιμή, αν το thunk έχει ήδη αποτιμηθεί. Η τιμή αυτή μπορεί να είναι ένας οκνηρός κατασκευαστής[3] ή μια *πρωτόγονη τιμή* (ακέραιος ή άλλος τύπος δεδομένων, ισομορφικός με τους ακέραιους). Ένας οκνηρός κατασκευαστής έχει δύο πεδία: ένα αναγνωριστικό (όπως το `Cons` ή το `Nil`, αν πρόκειται για έναν κλασικό τύπο δεδομένων λίστας) και την ΟΕΔ που περιέχει τις παραμέτρους του κατασκευαστή. Στην υλοποίησή μας, ένας κατασκευαστής είναι απλά μια συνάρτηση που επιστρέφει ένα ζεύγος από το αναγνωριστικό του κατασκευαστή και την ίδια την ΟΕΔ της.

Στην πράξη, χρησιμοποιούνται τρεις λέξεις για κάθε thunk: (α) ο δείκτης του κώδικα, (β) το αναγνωριστικό του κατασκευαστή ή η πρωτόγονη τιμή, και (γ) ο δείκτης του κατασκευαστή (ή μηδέν αν το thunk περιέχει κάποια πρωτόγονη τιμή). Ο δείκτης του κώδικα παίζει και τον ρόλο της σημαίας του thunk, με τον δείκτη `NULL` να σημαίνει ότι το thunk έχει ήδη υπολογιστεί. Σε αυτήν την αναπαράσταση, οι ακέραιοι, που είναι πάντα πρωτόγονες τιμές, έχουν την ίδια μορφή με έναν τύπο δεδομένων που έχει μόνο κατασκευαστές χωρίς παραμέτρους.

[3] Λόγω της χρήσης του μετασχηματισμού defunctionalization, τα κλεισίματα συναρτήσεων αναπαριστώνται και αυτά με κατασκευαστές.

Όταν η τιμή ενός thunk ζητηθεί, ελέγχεται η σημαία του: αν δείχνει ότι η τιμή έχει ήδη υπολογιστεί, τότε αυτή είναι ήδη αποθηκευμένη στο thunk, οπότε διαβάζεται και επιστρέφεται στον κώδικα που τη ζήτησε. Αν η σημαία δείχνει ότι το thunk δεν έχει υπολογιστεί ακόμα, τότε υπολογίζεται η τιμή του και στη συνέχεια αποθηκεύεται στο thunk, πριν επιστραφεί. Αυτός ο τρόπος οκνηρής αποτίμησης ονομάζεται *μοντέλο κελιού* (*cell model*) [137].

5.3 Ανάλυση διαφυγής και δέσμευση μνήμης

Σε αυτήν την ενότητα θα περιγράψουμε μια απλή στατική ανάλυση που επιλέγει αν μια ΟΕΔ θα αποθηκευτεί στη στοίβα ή στον σωρό.

Η τιμή που μπορεί να επιστρέψει μια συνάρτηση μπορεί να είναι δύο ειδών: είτε ένας οκνηρός κατασκευαστής που περιέχει κάποια συμφραζόμενα (και άρα τα συμφραζόμενα διαφεύγουν από τη συνάρτηση που τα όρισε και ίσως χρησιμοποιηθούν αργότερα), είτε κάποια τιμή που δεν περιέχει συμφραζόμενα. Τιμές του τελευταίου τύπου είναι οι ακέραιοι και όσοι τύποι δεδομένων είναι ισομορφικοί με αυτούς, όπως οι τύποι που περιλαμβάνουν μόνο κατασκευαστές χωρίς παραμέτρους (π.χ. ο τύπος `Bool` με κατασκευαστές `True` και `False`).

Λόγω της παραπάνω ιδιότητας των τιμών χωρίς συμφραζόμενα, γνωρίζουμε πόσο ζουν τα συμφραζόμενα μιας συνάρτησης που υπολογίζει μια τέτοια τιμή: μόνο όσο η συνάρτηση εκτελείται. Από τη στιγμή που η συνάρτηση επιστρέψει το αποτέλεσμά της, τα συμφραζόμενα πια δεν χρειάζονται, άρα μπορούν να τοποθετηθούν στη στοίβα κατά την κλήση της συνάρτησης, ώστε να αποδεσμευτούν αυτόματα κατά την επιστροφή από αυτή. Αντίθετα, οι συναρτήσεις που επιστρέφουν τιμές, οι οποίες μπορεί να είναι κατασκευαστές με παραμέτρους, τοποθετούν τα συμφραζόμενά τους στον σωρό κατά την κλήση τους. Αν κάποια στιγμή ο σωρός γεμίσει, θα χρειαστεί να γίνει συλλογή σκουπιδιών για να απελευθερωθεί χώρος.

Η παραπάνω στατική ανάλυση είναι μια απλή ανάλυση διαφυγής (escape analysis), που σε κάθε συνάρτηση, αποδίδει στατικά έναν χώρο αποθήκευσης για την ΟΕΔ της. Η βελτιστοποίηση αυτή επιτρέπει σε προγράμματα που χρησιμοποιούν ακέραιους ή άλλους απλούς τύπους δεδομένων να εκτελούνται χρησιμοποιώντας μόνο τη στοίβα. Λόγω του defunctionalization, σε αυτά περιλαμβάνονται και όσα προγράμματα χρησιμοποιούν μόνο ονόματα συναρτήσεων ως εκφράσεις υψηλότερης τάξης (όπως αναφέρθηκε στην Ενότητα 2.3.1). Αυτά είναι και τα προγράμματα που μπορούσε να εκτελέσει και η υλοποίηση με ΟΕΔ των Χαραλαμπίδη *et al.* για τον κλασικό νοηματικό μετασχηματισμό υψηλότερης τάξης, η οποία χρησιμοποιούσε μόνο τη στοίβα [55].

Όπως και η προηγούμενη βελτιστοποίηση της Ενότητας 4.3, αυτή η απλή ανάλυση διαφυγής που περιγράψαμε παραπάνω χρησιμοποιήθηκε στην υλοποίηση της TIM [15].

5.4 Κώδικας υποστήριξης της υλοποίησης

Στην ενότητα αυτή περιγράφονται κάποια μέρη της υλοποίησης που είναι απαραίτητα για τη λειτουργία της αλλά δεν εξετάζονται σε αυτήν τη διατριβή. Περισσότερες λεπτομέρειες πάνω στα δύο πρώτα θέματα είναι διαθέσιμες στην πτυχιακή εργασία του Παναγιώτη Θεοφιλόπουλου [274].

Μορφές CAF. Οι συναρτήσεις χωρίς παραμέτρους θεωρείται ότι βρίσκονται σε μορφή CAF (Constant Applicative Form) [126] και αποτελούν ειδική περίπτωση για τις υλοποιήσεις της κλήσης κατ' ανάγκη. Αν θεωρηθούν συναρτήσεις, όταν η τιμή τους υπολογίζεται

ξανά, θα είναι πάντα η ίδια, με αποτέλεσμα να σπαταλούνται άσκοπα υπολογιστικοί πόροι. Για αυτόν τον λόγο, οι περισσότερες υλοποιήσεις τις θεωρούν μεταβλητές και τις αποτιμούν με διαφορετικό τρόπο. Στην υλοποίησή μας, τις θεωρούμε μεταβλητές με καθολική εμβέλεια και δεσμεύουμε στην αρχή του προγράμματος μια ειδική καθολική ΟΕΔ, που περιέχει όλες τις μεταβλητές σε μορφή CAF του προγράμματος. Στη συνέχεια, τις αποτιμούμε κατά τα γνωστά, χρησιμοποιώντας τον μηχανισμό αποτίμησης κατ' ανάγκη των τυπικών μεταβλητών.

Συλλογή σκουπιδιών. Η τεχνική μας επιτρέπει να χρησιμοποιηθεί είτε ακριβής συλλογή σκουπιδιών, είτε συντηρητική. Και στις δύο περιπτώσεις, το σύνολο των ριζών (roots) της συλλογής σκουπιδιών βρίσκεται από τις ΟΕΔ που βρίσκονται στη στοίβα ή υπάρχουν δείκτες προς αυτές σε καταχωρητές. Αυτές οι ΟΕΔ αντιστοιχούν σε συναρτήσεις που μπορεί να μην έχει τελειώσει ακόμα η εκτέλεσή τους. Ρίζες θεωρούνται επίσης οι μεταβλητές σε μορφή CAF. Για ακριβή συλλογή σκουπιδιών, χρησιμοποιήθηκε ο αλγόριθμος συλλογής σκουπιδιών αντιγραφής (semi-space) [136]. Για συντηρητική συλλογή σκουπιδιών, ενσωματώθηκε ο αλγόριθμος συντηρητικής συλλογής σκουπιδιών των Boehm-Demers-Weiser [36], ο οποίος αγνοεί το πεδίο *layout* των ΟΕΔ.

Αυστηρή αποτίμηση. Η υλοποίησή μας υποστηρίζει σημειώσεις αυστηρότητας (strictness annotations) σε τυπικές μεταβλητές και πεδία κατασκευαστών, χρησιμοποιώντας το σύμβολο `!` μπροστά από το όνομα της μεταβλητής ή του πεδίου. Για παράδειγμα, στον ακόλουθο κώδικα, η τυπική μεταβλητή `n` θα υπολογιστεί κατά την κλήση της συνάρτησης, έχοντας αυστηρή σημασιολογία:

```
fib !n = if n > 2 then fib (n−1) + fib (n−2) else 1
```

Η αυστηρή αποτίμηση μιας τυπικής μεταβλητής υλοποιείται με την αποτίμηση της μεταβλητής κατά την είσοδο στη συνάρτηση που χρησιμοποιείται, πριν αποτιμηθεί το σώμα της συνάρτησης. Το χαρακτηριστικό αυτό είναι αντίθετο με τη σημασιολογία του νοηματικού μετασχηματισμού γενικά αλλά προστέθηκε για να μπορεί να χρησιμοποιηθεί στο μέλλον κάποια στατική ανάλυση αυστηρότητας (strictness analysis) [186, 254]. Επίσης, λόγω της αναπαράστασης των κατασκευαστών από συναρτήσεις (που αναφέρθηκε στην Ενότητα 3.3), η παραπάνω δυνατότητα επιτρέπει και τη δήλωση αυστηρών πεδίων κατασκευαστών.

Ανάλυση μοιράσματος. Η αναπαράσταση με ΟΕΔ μιμείται την αποθήκη του προηγούμενου κεφαλαίου και άρα μπορεί να χρησιμοποιηθεί η ανάλυση μοιράσματος της Ενότητας 4.3.

5.5 Μετρήσεις

Σε αυτήν την ενότητα συγκρίνουμε την ταχύτητα των προγραμμάτων που παράγει η υλοποίησή μας με άλλους τέσσερις μεταγλωττιστές της Haskell:

- Glasgow Haskell Compiler (GHC): ο πιο δημοφιλής μεταγλωττιστής της Haskell [217].
- Utrecht Haskell Compiler (UHC): μεταγλωττιστής που έχει υλοποιηθεί με την τεχνική των attribute grammars, ο οποίος υποστηρίζει τα περισσότερα χαρακτηριστικά των προτύπων Haskell 98 και Haskell 2010 [76].
- NHC98: ένας μικρός και φορητός μεταγλωττιστής της Haskell 98 [235].

Πρόγραμμα	GIC	GIC-llvm	GHC7	GHC6	NHC	UHC	JHC
ack	2.47	1.25	0.62	0.48	6.18	40.03	0.05
church	3.55	2.09	0.61	0.55	11.58	68.37	0.17
collatz	0.69	0.41	1.07	2.66	84.28	46.90	0.16
digits_of_e1	2.30	2.09	0.77	1.74	60.71	75.29	–[1]
fast-reverse	3.03	1.95	1.74	1.82	1.35	9.41	–[2]
fib	1.35	1.12	0.50	0.51	10.43	55.55	0.17
naive-reverse	3.02	2.87	0.49	0.42	0.79	3.56	0.75
ntak	8.62	5.87	2.91	3.65	154.74	91.95	7.18
primes	2.55	1.58	2.19	2.30	172.45	173.81	0.73
queens-num	0.33	0.23	0.31	0.33	21.16	12.43	0.14
queens	3.92	3.24	0.44	0.48	27.17	123.98	0.82
quick-sort	3.18	2.77	1.92	1.90	1.51	5.42	8.58
tree-sort	2.19	1.97	0.39	0.33	0.91	6.58	0.72
GMR[3]	1.38	1.00	0.51	0.57	7.28	18.49	0.33

[1] jhc: σφάλμα μεταγλώττισης, [2] jhc: σφάλμα χρόνου εκτέλεσης.
[3] Γεωμετρικός μέσος των λόγων, σύγκριση με τον GIC-llvm.

Εικόνα 5.2: Σύγκριση χρόνου εκτέλεσης για 13 δοκιμαστικά προγράμματα. Οι χρόνοι είναι σε δευτερόλεπτα.

- JHC: ένας πειραματικός και γρήγορος μεταγλωττιστής της Haskell, που υλοποιήθηκε για να δοκιμαστούν διάφορες νέες βελτιστοποιήσεις πάνω στη γλώσσα [171].

Η σύγκριση βασίζεται σε 13 προγράμματα, τα περισσότερα από τα οποία είναι κλασικά δοκιμαστικά προγράμματα οκνηρών γλωσσών προγραμματισμού (π.χ. από τη σουίτα δοκιμαστικών προγραμμάτων NoFib [213]). Κάποια από τα προγράμματα κάνουν μόνο αριθμητικές πράξεις (όπως τα προγράμματα `ack`, `fib`, `primes` και `queens-num`), μόνο επεξεργασία λιστών (όπως τα `naive-reverse` και `fast-reverse`), αριθμητικές πράξεις μαζί με επεξεργασία λιστών ή συναρτήσεις υψηλότερης τάξης (όπως τα `church`, `ntak`, `collatz`, `digits_of_e1` και `quick-sort`), και επεξεργασία άλλων τύπων δεδομένων που έχουν οριστεί από τον χρήστη (όπως τα `queens` και `tree-sort`).

Οι μετρήσεις έγιναν σε έναν υπολογιστή με τέσσερις τετραπύρηνους επεξεργαστές Intel Xeon E7340 στα 2.40GHz και με 16 GB μνήμη, με λειτουργικό σύστημα Debian 6.0.5. Οι εκδόσεις των μεταγλωττιστών που χρησιμοποιήθηκαν ήταν ο GHC 7.4.1 και ο GHC 6.12.1, ο UHC/EHC 1.1.4, ο NHC98 1.22 και ο JHC 0.8.0. Ο μεταγλωττιστής μας εμφανίζεται στον πίνακα των μετρήσεων ως `GIC` (Generalized Intensional Compiler). Όλα τα δοκιμαστικά προγράμματα εκτελέστηκαν πέντε φορές και καταγράφηκε ο μέσος χρόνος εκτέλεσης. Η επίδραση του συλλέκτη σκουπιδιών ελαχιστοποιήθηκε σε όλους τους μεταγλωττιστές, θέτοντας ένα μεγάλο όριο στο μέγεθος του σωρού — στην πράξη όλα τα προγράμματα είτε δεν έκαναν καθόλου συλλογή σκουπιδιών, είτε έκαναν πολύ λίγο. Τέλος, απενεργοποιήσαμε τη στατική ανάλυση αυστηρότητας (strictness analysis) σε όλους τους μεταγλωττιστές, ώστε να μελετήσουμε μόνο την ταχύτητα εκτέλεσης τους στην περίπτωση της κλήσης κατ' ανάγκη.

Τα αποτελέσματα των μετρήσεων φαίνονται στην Εικόνα 5.2. Στον πίνακα αυτό, ως `GIC-llvm` εμφανίζεται ο μεταγλωττιστής μας, ο παραγόμενος κώδικας C του οποίου μεταγλωττίζεται με το εργαλείο `llvm-gcc`, που συνδέει τον GCC με τον μεταγλωττιστή του LLVM [152]. Χρησιμοποιήσαμε τον GCC 4.4.5 και το LLVM 2.6. Οι μετρήσεις δείχνουν τα εξής:

- Η μεταγλώττιση του παραγόμενου κώδικα C του GIC με το `llvm-gcc` παράγει πιο γρήγορο κώδικα σε σχέση με τον απλό GCC. Παρόμοια αποτελέσματα προκύπτουν και με τον μεταγλωττιστή Clang. Στη συνέχεια του κειμένου, όταν θα αναφερόμαστε στην υλοποίησή μας, θα αναφερόμαστε στον `GIC-llvm`.

- Η νοηματική υλοποίηση είναι κατά μέσο όρο 2-3 πιο αργή από τις πλήρως βελτιστοποιημένες υλοποιήσεις του `GHC6` και του `GHC7`. Ειδικότερα, στα προγράμματα `collatz`, `primes` και `queens-num`, η νοηματική μας υλοποίηση εκτελείται γρηγορότερα σε σχέση με τον `GHC6` και τον `GHC7`. Μιας και ο νοηματικός μεταγλωττιστής δεν περιλαμβάνει προχωρημένες βελτιστοποιήσεις ή αντίστοιχες τεχνικές μετασχηματισμού του κώδικα, θεωρούμε ότι υπάρχουν δυνατότητες βελτίωσης της υλοποίησης.

- Σε κάποια προγράμματα (όπως το `ack` και το `church`) ο `GHC6` είναι ταχύτερος του `GHC7`. Αυτό αναφέρθηκε ως σφάλμα στην ομάδα ανάπτυξης του GHC (με αριθμό #5888 στο σύστημα παρακολούθησης σφαλμάτων του GHC) και είναι σχετικό με μια βελτιστοποίηση του GHC πάνω στους ακεραίους, η οποία δεν λειτουργούσε σωστά και διορθώθηκε στην έκδοση 7.6.1.

Γενικά, τα αποτελέσματα των μετρήσεων της νοηματικής υλοποίησης είναι ενθαρρυντικά, ειδικά λόγω του ότι ο μεταγλωττιστής μας είναι πολύ λιγότερο ώριμος σε σχέση με τους υπόλοιπους, και ότι η υλοποίηση που περιγράψαμε παραπάνω στοχεύει στην απλότητα και όχι στην ταχύτητα.

5.6 Σχετική έρευνα

Η τεχνική που περιγράψαμε σε αυτό το κεφάλαιο συσχετίζει τις έννοιες του γενικευμένου νοηματικού μετασχηματισμού με αντίστοιχες έννοιες κλασικών υλοποιήσεων γλωσσών προγραμματισμού: (α) τα συμφραζόμενα αντιστοιχούν σε οκνηρές εγγραφές ενεργοποίησης, (β) η αποθήκη κατανέμεται σε πεδία από thunk στις εγγραφές ενεργοποίησης, (γ) η στοίβα και ο σωρός αποθηκεύουν συμφραζόμενα, και (δ) ο πίνακας δέσμευσης συμφραζομένων γίνεται συνάρτηση δέσμευσης μνήμης για εγγραφές ενεργοποίησης.

Γενικά, η νοηματική υλοποίηση συναρτησιακών γλωσσών διαφέρει στη φιλοσοφία της σε σχέση με τις δημοφιλείς υλοποιήσεις που είναι βασισμένες στην αναγωγή γράφου. Μια εργασία που έχει κοινά στοιχεία με την υλοποίηση που περιγράφεται σε αυτό το κεφάλαιο είναι ο μεταγλωττιστής GRIN του Boquist [38], ο οποίος επίσης χρησιμοποιεί εσωτερικά μια αναπαράσταση βασισμένη στον μετασχηματισμό defunctionalization. Όμως ο GRIN χρησιμοποιεί έναν αριθμό από “ετικέτες” (“tags”) για να χαρακτηρίζει διαφορετικές δομές μιας οκνηρής γλώσσας (κατασκευαστές, εφαρμογές συναρτήσεων, μερικές εφαρμογές συναρτήσεων), ενώ ο γενικευμένος νοηματικός μετασχηματισμός χρησιμοποιεί μια ενιαία αναπαράσταση για αυτούς τους τρεις τύπους δομών. Επιπλέον, ο GRIN βασίζεται σε μια αυστηρή γλώσσα πρώτης τάξης, ενώ ο νοηματικός μετασχηματισμός σε μια μη αυστηρή γλώσσα πρώτης τάξης. Μια άλλη διαφορά στην υλοποίηση είναι ότι ο GRIN μεταγλωττίζει κατευθείαν τη γλώσσα του για αναγωγή γράφου χρησιμοποιώντας ειδικές βελτιστοποιήσεις, όπως ένας αλγόριθμος δέσμευσης καταχωρητών (interprocedural register allocation) ενώ ο GIC παράγει νοηματικό κώδικα μηδενικής τάξης, τον οποίο στη συνέχεια μεταφράζει σε C χρησιμοποιώντας οκνηρές εγγραφές δραστηριοποίησης.

Ο GHC βασίζεται στην εικονική μηχανή STG, η οποία ακολουθεί το μοντέλο αυτοενημέρωσης (self-updating model) για την οκνηρή αποτίμηση [137]· όπως είδαμε στην Ενότητα 5.2, η υλοποίηση με ΟΕΔ βασίζεται στο μοντέλο κελιού, και άρα διαφέρει σημαντικά

ως προς το μοντέλο εκτέλεσής της. Επίσης, η STG (όπως και οι περισσότερες αφηρημένες μηχανές συναρτησιακών γλωσσών) χειρίζεται εκφράσεις υψηλότερης τάξης και μερική εφαρμογή, άρα έχει διαφορετική σημασιολογία σε σχέση με τον συνδυασμό νοηματικού μετασχηματισμού και μετασχηματισμού defunctionalization που χρησιμοποιούμε. Στην πράξη, ο GHC κρατά την τρέχουσα στοίβα εκτέλεσης μέσα σε κάθε thunk που δεν έχει αποτιμηθεί πλήρως [164]· αυτό μοιάζει με την υλοποίηση αυτού του κεφαλαίου που κρατά την τρέχουσα ΟΕΔ μέσα σε έναν οκνηρό κατασκευαστή.

Η υλοποίηση του γενικευμένου νοηματικού μετασχηματισμού μοιάζει με τις εικονικές μηχανές που βασίζονται στα περιβάλλοντα (environment-based abstract machines), όπως αυτές των Friedman και Wise [101], των Henderson και Morris [122], και του Krivine [148], ή οι μηχανές STG που βασίζονται σε περιβάλλοντα των De La Encina και Peña [70]. Μια βασική διαφορά της νοηματικής προσέγγισης σε σχέση με τις παραπάνω μηχανές είναι ότι βασίζεται σε μια γλώσσα εισόδου πρώτης τάξης· παρόλα αυτά, τα συμφραζόμενα του νοηματικού μετασχηματισμού παίζουν κατά κάποιο τρόπο τον ρόλο του περιβάλλοντος, μιας και καθοδηγούν τον μηχανισμό εκτέλεσης ώστε να αντικαταστήσει τη σωστή έκφραση στη σωστή τυπική μεταβλητή στο σώμα μιας συνάρτησης.

Η υλοποίηση με ΟΕΔ μοιάζει με την αφηρημένη μηχανή TIM [86]: και οι δύο, επειδή βασίζονται σε supercombinators, χρησιμοποιούν μια μόνο εγγραφή δραστηριοποίησης ως αναπαράσταση του περιβάλλοντος.[4] Η υλοποίησης με ΟΕΔ όμως είναι πιο απλή και έχει διαφορετική σημασιολογία σε σχέση με την TIM, αφού δε χρειάζεται να χειρίζεται εκφράσεις υψηλότερης τάξης και μερική αποτίμηση. Έτσι, η υλοποίησή μας δεν απαιτεί ειδική πληροφορία στη στοίβα (stack markers) ή τον τελεστή σταθερού σημείου Y της TIM. Η υλοποίηση με ΟΕΔ, υπολογίζει την τρέχουσα έκφραση χρησιμοποιώντας τη στοίβα κλήσεων της C χωρίς να διατηρεί μια "σπονδυλική στήλη" από εκφράσεις (spine) στη μνήμη, είναι δηλαδή spineless, χαρακτηριστικό που μοιράζεται με την TIM [218] και με την STG [137].

Βασικό χαρακτηριστικό της υλοποίησης αυτού του κεφαλαίου είναι ότι κάθε συνάρτηση του αρχικού προγράμματος μετατρέπεται σε μια συνάρτηση σε C που δέχεται μια ΟΕΔ. Το στυλ της C που παράγεται (με τις κατάλληλες βοηθητικές μακροεντολές) μοιάζει με μια οκνηρή παραλλαγή της C και ο κώδικας που παράγεται, παρά τον όγκο του, είναι αναγνώσιμος από τον άνθρωπο.[5] Και άλλοι μεταγλωττιστές της Haskell χρησιμοποιούν εσωτερικά υψηλότερου επιπέδου παραλλαγές της C: ο GHC χρησιμοποιεί την C−− [219], η οποία έχει υποστήριξη για συλλογή σκουπιδιών, εξαιρέσεις, ταυτοχρονισμό και αποσφαλμάτωση [231], ενώ ο Intel Labs Haskell Research Compiler χρησιμοποιεί την Pillar, η οποία είναι επίσης επηρεασμένη από τη C−− [161].

Περίληψη

Σε αυτήν την ενότητα περιγράψαμε μια τεχνική υλοποίησης του γενικευμένου νοηματικού μετασχηματισμού, που είναι κατάλληλη για δημοφιλείς αρχιτεκτονικές υπολογιστών. Δείξαμε ότι ο νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί χρησιμοποιώντας τεχνικές για κλασικό υλικό υπολογιστών και τα προγράμματα που παράγονται εκτελούνται γρήγορα. Με βάση τις ιδέες που παρουσιάστηκαν, στο επόμενο κεφάλαιο θα παρουσιαστεί μια βελτίωση της παραπάνω υλοποίησης, η οποία έχει ακόμα καλύτερη απόδοση.

[4] Αν εξαιρέσουμε την ΟΕΔ των CAF που είναι επίσης ορατή από όλα τα σημεία του κώδικα και αναφέρθηκε στην Ενότητα 5.4.

[5] Σε αυτό βοηθά και η χρήση της επέκτασης του GCC για εκφράσεις C, βλ. `https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html`.

Κεφάλαιο 6

Μια αποδοτική αναπαράσταση κατασκευαστών στα 64 bit

Σε αυτό το κεφάλαιο θα δούμε πώς μπορούμε να κάνουμε οικονομία μνήμης στην υλοποίηση με ΟΕΔ όταν παράγει προγράμματα για την αρχιτεκτονική AMD64.[1]

Το χάσμα στην ταχύτητα μεταξύ του επεξεργαστή και της μνήμης (processor-memory performance gap) [40, 299] είναι σημαντικό πρόβλημα στους σύγχρονους υπολογιστές. Ο επεξεργαστής συνήθως είναι πολύ πιο γρήγορος από τη μνήμη, με αποτέλεσμα τα προγράμματα να πρέπει να περιμένουν τη μνήμη για να διαβάσουν ή να γράψουν δεδομένα. Για να αντιμετωπιστεί αυτό το πρόβλημα, συχνά χρησιμοποιείται κρυφή μνήμη (cache), η οποία είναι μια γρήγορη μνήμη που μεσολαβεί ανάμεσα στον επεξεργαστή και την κύρια μνήμη και προσπαθεί να κρύψει τις καθυστερήσεις δεδομένων που χρησιμοποιούνται συχνά. Οι κρυφές μνήμες όμως δεν μπορούν να έχουν μεγάλο μέγεθος και για αυτό τα προγράμματα πρέπει να έχουν καλή τοπικότητα κρυφής μνήμης (cache locality), μια ιδιότητα που συνδέεται με το μέγεθος του κώδικα και των δεδομένων που αυτός χρησιμοποιεί [157, 306]. Το χάσμα μεταξύ επεξεργαστή και μνήμης είναι ιδιαίτερα αισθητό στις υλοποιήσεις της Haskell, επειδή η γλώσσα βασίζεται σε αμετάβλητες (immutable) δομές δεδομένων και τα προγράμματα δαπανούν πολύ χρόνο για δέσμευση μνήμης και συλλογή σκουπιδιών, με αποτέλεσμα συχνά να μη μπορούν να εκμεταλλευτούν την ταχύτητα της κρυφής μνήμης [191].

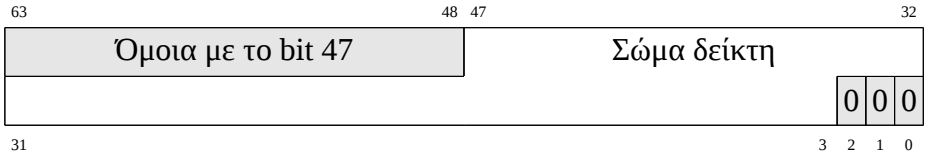
Στη συνέχεια του κεφαλαίου θα δούμε πώς η υλοποίησή μας μπορεί να χρησιμοποιήσει μια συμπαγή αναπαράσταση στη μνήμη, εκμεταλλευόμενη πλεονάζουσα πληροφορία που περιέχεται στους δείκτες στην αρχιτεκτονική AMD64 [169], η οποία αποτελεί μια από τις πιο δημοφιλείς αρχιτεκτονικές υπολογιστών σήμερα. Η αναπαράσταση αυτή προσπαθεί να μειώσει τη χρήση μνήμης τοποθετώντας μαζί (clustering [59]) τις δομές δεδομένων του χρόνου εκτέλεσης και κάνοντας πράξεις σε επίπεδο bit.

Αρχικά περιγράφουμε τα χαρακτηριστικά της αρχιτεκτονικής AMD64 που εκμεταλλευόμαστε (Ενότητα 6.1). Στη συνέχεια δείχνουμε πώς η υλοποίηση τα χρησιμοποιεί στην πράξη (Ενότητα 6.2) και πώς μπορεί να λειτουργήσει η συλλογή σκουπιδιών στη νέα αυτή αναπαράσταση (Ενότητα 6.3). Μετράμε πόσο αποδοτική είναι η νέα αυτή αναπαράσταση, συγκρίνοντάς τη με αυτή που περιγράψαμε στο προηγούμενο κεφάλαιο, αλλά και σε σχέση με τον GHC με πλήρεις βελτιστοποιήσεις (Ενότητα 6.4). Τέλος, εξετάζουμε σχετικές αναπαραστάσεις και τεχνικές υλοποίησης (Ενότητα 6.5) και συνοψίζουμε με σχόλια για περαιτέρω βελτιώσεις της αναπαράστασης (Ενότητα 6.6).

6.1 Πλεονάζουσα πληροφορία στους δείκτες της AMD64

Οι δείκτες στην αρχιτεκτονική AMD64 έχουν τα εξής δύο χαρακτηριστικά:

[1] Το κεφάλαιο περιλαμβάνει υλικό από άρθρο του 2014 των Γ. Φουρτούνη και Ν. Παπασπύρου [95].



Εικόνα 6.1: Ο χώρος σε έναν δείκτη της αρχιτεκτονικής AMD64.

1. Στοιχίζονται στα 8 byte [169, p. 12], με αποτέλεσμα τα τελευταία 3 bit τους να είναι μηδέν.

2. Μπορούν να χρησιμοποιηθούν για να δεικτοδοτήσουν διευθύνσεις των 48 bit [169, Section 3.3.2]. Τα υψηλότερα 16 bit θεωρούνται ίσα με το πιο σημαντικό χρησιμοποιούμενο bit [12, Section 1.1.3].

Κάθε δείκτης χρησιμοποιεί δηλαδή μόνο 45 πραγματικά bit, τα οποία στο εξής θα αποκαλούμε *σώμα δείκτη* (*pointer body*), με αποτέλεσμα να μένουν ελεύθερα 19 bit ανά δείκτη (το 29.69%) που δεν περιέχουν πληροφορία και μπορούν να επαναχρησιμοποιηθούν. Ο χώρος που καταλαμβάνει ένας δείκτης μπορεί να αναπαρασταθεί όπως φαίνεται στην Εικόνα 6.1, με την σκιασμένη περιοχή να έχει γνωστά περιεχόμενα και άρα να μπορεί να επαναχρησιμοποιηθεί, αρκεί η γνωστή της πληροφορία να ανακατασκευάζεται όταν χρειάζεται.

Οι διαφορετικές περιοχές του χώρου που καταλαμβάνει ένας δείκτης μπορούν να προσπελαστούν με πράξεις bit, όπως οι λογικές ολισθήσεις (logical shifts) και οι πράξεις μεταξύ τιμών αλήθειας (boolean operations). Για παράδειγμα, αν τα bit 48-63 θεωρηθεί ότι περιέχουν έναν ακέραιο των 16 bit χωρίς πρόσημο (του οποίου ο τύπος στη C είναι `uint16_t` [273]), τότε η μακροεντολή της C που ακολουθεί παίρνει έναν δείκτη p και του εφαρμόζει μια πράξη λογικής ολίσθησης για να απομονώσει τον ακέραιο:

```
#define INT16(p) ((uint16_t)((uintptr_t) p >> 48))
```

Στη συνέχεια αυτού του κεφαλαίου θα χρησιμοποιήσουμε κώδικα C όπως ο παραπάνω για να αναπαραστήσουμε τέτοιες λειτουργίες, όπως στο αντίστοιχο άρθρο του Gudeman [115], θεωρώντας ότι ο κώδικας αυτός μεταγλωττίζεται από έναν μεταγλωττιστή της C για την αρχιτεκτονική AMD64. Θα χρησιμοποιήσουμε επίσης τους πρότυπους τύπους `uintptr_t` και `intptr_t` [273] της C για να χειριστούμε δείκτες ως ακέραιους των 64 bit, με ή χωρίς πρόσημο, και να κάνουμε ολισθήσεις αντίστοιχα διατηρώντας ή παραλείποντας το πρόσημο.

6.2 Αναπαράσταση ενός thunk με μια λέξη μηχανής

Όπως περιγράφηκε στην Ενότητα 5.1, ένα thunk έχει τρία πεδία: (α) μια σημαία που δείχνει αν έχει υπολογιστεί η τιμή του, (β) έναν δείκτη προς κώδικα, και (γ) μια υπολογισμένη τιμή. Σε αυτήν την ενότητα θα περιγράψουμε πώς και τα τρία αυτά πεδία μπορούν να χωρέσουν σε μια λέξη μηχανής.

Αρχικά παρατηρούμε ότι ο δείκτης κώδικα χρειάζεται μόνο όταν η τιμή του thunk δεν έχει υπολογιστεί ακόμα, ενώ το αντίθετο ισχύει για την υπολογισμένη τιμή. Επομένως, μπορεί να χρησιμοποιηθεί η ίδια λέξη και για τον δείκτη και για την τιμή, αν το τελευταίο bit της λέξης χρησιμοποιηθεί για τη σημαία που δείχνει αν το thunk έχει αποτιμηθεί.

Ο δείκτης κώδικα χωράει σε μια λέξη μηχανής (εξ ορισμού), με τα τρία χαμηλότερα bit

του να είναι μηδέν.[2] Η υπολογισμένη τιμή είναι είναι ένας οκνηρός κατασκευαστής, δηλαδή ένα ζεύγος από ένα αναγνωριστικό κατασκευαστή και έναν δείκτη προς μια ΟΕΔ. Αν υποθέσουμε ότι κάθε τύπος δεδομένων έχει το πολύ 2^{16} διαφορετικούς κατασκευαστές, μπορούμε να τοποθετήσουμε το αναγνωριστικό στα 16 αχρησιμοποίητα υψηλότερα bit του δείκτη ΟΕΔ, με αποτέλεσμα όλος ο οκνηρός κατασκευαστής να χρειάζεται μόνο τη λέξη μηχανής του δείκτη. Επειδή ο χώρος που χρησιμοποιείται είναι ενός δείκτη, τα τρία χαμηλότερα bit θα είναι επίσης μηδέν. Αφού και στις δύο περιπτώσεις (δείκτης κώδικα και υπολογισμένη τιμή) τα τρία χαμηλότερα bit είναι μηδέν, όντως μπορούμε να χρησιμοποιήσουμε το τελευταίο bit ως σημαία του thunk.

Έχουμε ήδη φτάσει σε μια συμπαγή αναπαράσταση αλλά υπάρχουν άλλα δύο αχρησιμοποίητα bit (1-2). Αυτό το εκμεταλλευόμαστε για να χρησιμοποιήσουμε το bit 1 ως *σημαία τιμής* (*value flag*) για να ξεχωρίζουν οι κανονικοί κατασκευαστές από τις πρωτόγονες τιμές (όπως η έκφραση `True` ή η `42`)· οι δεύτεροι δε χρειάζονται δείκτη ΟΕΔ και μπορούν να χρησιμοποιήσουν όλο το χώρο του δείκτη.[3] Οι πρωτόγονες τιμές χρησιμοποιούν χώρο 62 bit και είναι για παράδειγμα, οι ακέραιοι, οι τιμές αλήθειας, οι αριθμοί κινητής υποδιαστολής, και άλλοι τύποι δεδομένων απαρίθμησης. Η Haskell έχει στατικούς τύπους και αυτή η σημαία κανονικά δε θα χρειαζόταν: ο κώδικας πάντα εφαρμόζεται σε τιμές του κατάλληλου τύπου στον χρόνο εκτέλεσης. Η σημαία αυτή όμως χρειάζεται για τον συλλέκτη σκουπιδιών της επόμενης ενότητας, ο οποίος πρέπει να εξετάζει κάθε thunk μιας ΟΕΔ για να βρίσκει δείκτες.

Η εικόνα 6.2 δείχνει την αναπαράσταση στη μνήμη των thunk που δεν έχουν υπολογιστεί ακόμα και αυτών που είναι ήδη υπολογισμένα· τα δεύτερα αφορούν και οκνηρούς κατασκευαστές και πρωτόγονες τιμές. Παρατηρούμε ότι υπάρχει κάποιος αχρησιμοποίητος χώρος (το bit 2 στους κατασκευαστές και τα bit 1-2 στα thunk που δεν έχουν υπολογιστεί ακόμα). Κρατάμε την ακολουθία `110` των τελευταίων τριών bit για μελλοντική χρήση.

Χειρισμός μη υπολογισμένων thunk. Η εξής μακροεντολή μπορεί να ελέγξει αν ένα thunk έχει ήδη υπολογιστεί, ελέγχοντας τη σημαία του:

```
#define IS_VAL(t) (((uintptr_t) t & 1) == 0)
```

Αν η σημαία είναι αληθής, τότε η μακροεντολή `CODE` μπορεί να χρησιμοποιηθεί για να επιστρέψει τον δείκτη προς τον κώδικα που πρέπει να εκτελεστεί για να υπολογιστεί το thunk:

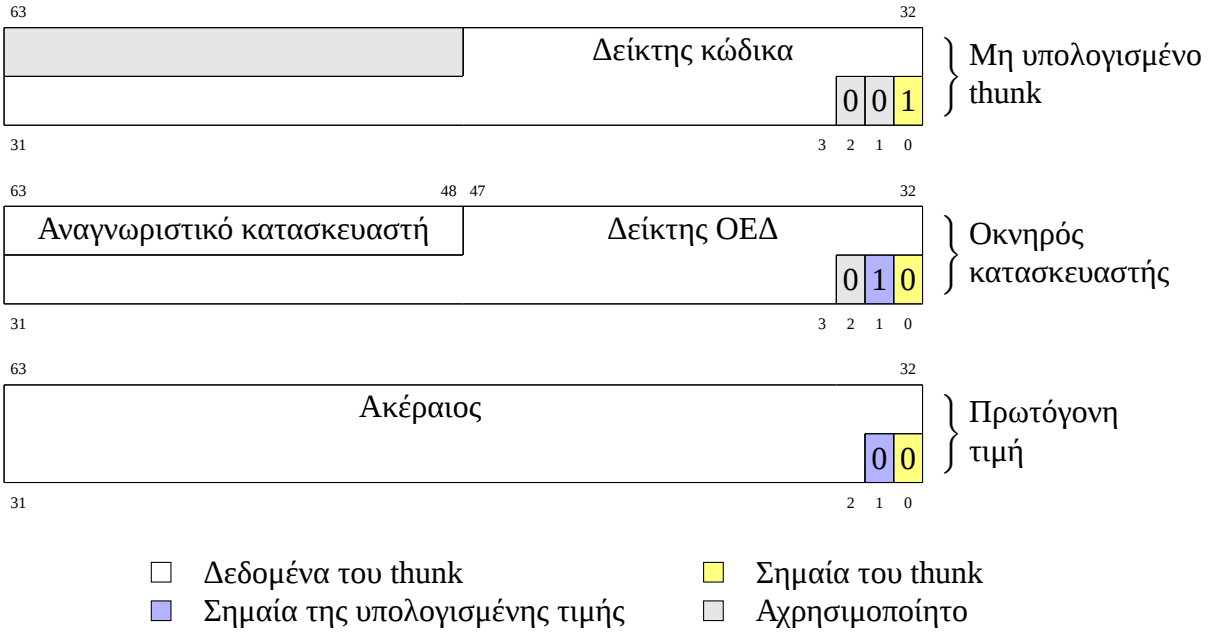
```
#define CODE(t) ((uintptr_t) t & ~1)
```

Αν η σημαία του thunk είναι ψευδής, τότε τα bit 2-63 περιέχουν μια ήδη υπολογισμένη τιμή, η οποία είναι είτε ένας οκνηρός κατασκευαστής, είτε μια πρωτόγονη τιμή, και μπορούν να χρησιμοποιηθούν οι μακροεντολές των επόμενων παραγράφων για να διαβαστούν αυτά τα περιεχόμενα. Όπως έχει ήδη αναφερθεί, ο τύπος μιας τιμής είναι γνωστός στατικά· δε χρειάζεται να ελεγχθεί το bit 1.

Οκνηροί κατασκευαστές. Ένας οκνηρός κατασκευαστής περιέχει δύο πεδία: ένα αναγνωριστικό κατασκευαστή (constructor tag) και έναν δείκτη προς την ΟΕΔ με τα περιεχόμενα του κατασκευαστή (το καθένα από τα οποία είναι επίσης ένα thunk). Το αναγνωριστικό του κατασκευαστή μπορεί να διαβαστεί με απλή ολίσθηση:

[2] Αυτή η ιδιότητα μπορεί να εξασφαλιστεί, π.χ. στον μεταγλωττιστή GCC, με τη χρήση της παραμέτρου `-falign-functions`.

[3] Παρόμοιο χαρακτηρισμό διάφορων τύπων δεδομένων ως ακέραιους ακολουθεί και η υλοποίηση της OCaml [232].



Εικόνα 6.2: Ο χώρος σε ένα thunk των 64 bit.

```
#define CONSTR(p) ((uintptr_t) p >> 48)
```

Ο δείκτης του κατασκευαστή μπορεί να ανακατασκευαστεί από τα bit 3-47. Όπως φαίνεται και στην Εικόνα 6.1, ένας δείκτης μπορεί να αναδημιουργηθεί από το σώμα δείκτη του, αν τα τρία χαμηλότερα bit τεθούν ίσα με μηδέν, και αν το υψηλότερο bit του σώματος επεκταθεί προς τα αριστερά, στα υπόλοιπα bit. Η ακόλουθη μακροεντολή `CPTR` δείχνει αυτή τη λειτουργία, με την μετατροπή σε `(intptr_t)` να χρησιμοποιείται για να κάνει την ολίσθηση προσημασμένη:

```
#define CPTR(p) (((intptr_t) p << 16 >> 16) & ~7)
```

Πρωτόγονες τιμές. Αυτές είναι προσημασμένοι ακέραιοι των 62 bit, που βρίσκονται στα bit 2-63 και μπορούν να διαβαστούν και να γραφτούν με τις εξής μακροεντολές:

```
#define PVAL_R(p) ((intptr_t) p >> 2)
#define PVAL_W(i) ((intptr_t) i << 2)
```

Το κόστος της παραπάνω αναπαράστασης είναι ότι για κάθε βασική πράξη, κάθε παράμετρος πρέπει να ολισθήσει 2 bit προς τα δεξιά, να γίνει η πράξη, και το αποτέλεσμα να ολισθήσει 2 bit προς τα αριστερά, για να αποτελεί έγκυρο thunk. Για κάποιες πράξεις μπορούμε να αποφύγουμε αυτό το κόστος, μιας και τα χαμηλά bit είναι πάντα μηδέν, για παράδειγμα μπορούμε να προσθέσουμε δύο θετικούς ακέραιους χωρίς ολίσθηση ως εξής [115]:

```
#define SUM(p1,p2) ((intptr_t) p1 + (intptr_t) p2)
```

Γιατί χρησιμοποιούμε αυτή την αναπαράσταση των χαμηλών bit; Όπως είδαμε στην προηγούμενη παράγραφο, η αναπαράσταση των ακεραίων έχει πλεονέκτημα, μιας και τα χαμηλά bit είναι μηδέν και κάποιες πράξεις μπορούν να απλοποιηθούν. Μια άλλη επιλογή θα ήταν να θέτουμε τα χαμηλά bit των κατασκευαστών ίσα με μηδέν, ώστε το διάβασμα του δείκτη ΟΕΔ του κατασκευαστή να μη χρησιμοποιεί τη μάσκα `~7`. Αντίστοιχα, θα μπορούσαμε

να κάνουμε μηδέν τα τρία χαμηλότερα bit των thunk που δεν έχουν υπολογιστεί ακόμα, ώστε η `CODE` να μη χρειάζεται τη μάσκα `~1`. Στην πράξη, δεν είδαμε κάποια διαφορά στην ταχύτητα μεταξύ αυτών των τριών επιλογών στην αναπαράσταση.

6.3 Συλλογή σκουπιδιών

Αν και η γεννήτρια κώδικα του μεταγλωττιστή μας παράγει C, δεν μπορεί να χρησιμοποιήσει κατευθείαν έναν συντηρητικό συλλέκτη σκουπιδιών γενικής χρήσης όπως αυτός των Boehm-Demers-Weiser [36], γιατί η πληροφορία που προσθέτουμε στους δείκτες, τους κρύβει από τον συλλέκτη.[4] Χρειαζόμαστε επομένως έναν συλλέκτη σκουπιδιών που να καταλαβαίνει την αναπαράσταση των thunk και των ΟΕΔ. Σε αυτήν την ενότητα θα περιγράψουμε τη βασική πληροφορία που χρειάζεται να παρέχει η αναπαράστασή μας σε έναν συλλέκτη σκουπιδιών αντιγραφής (semi-space garbage collector) [136].

Στην υλοποίησή μας υπάρχει μόνο ένας τρόπο να δεσμευτεί μνήμη: η δημιουργία μιας ΟΕΔ κατά την κλήση μιας συνάρτησης ή ενός κατασκευαστή. Ο ζητούμενος συλλέκτης σκουπιδιών μας τότε είναι απλός: οι ρίζες είναι οι δείκτες προς ΟΕΔ που βρίσκονται σε καταχωρητές ή στη στοίβα και η συλλογή προχωρά αναδρομικά σε όλες τις ΟΕΔ που είναι προσβάσιμες από αυτούς. Δεδομένης μιας ΟΕΔ, δείκτες προς άλλες ΟΕΔ μπορούν να βρεθούν (α) στο πεδίο *prev*, (β) σε υπολογισμένα thunk που περιέχουν κατασκευαστές, και (γ) στα πεδία *nested*.

Όπως περιγράφηκε στην Ενότητα 5.1, οι ΟΕΔ διαφορετικών συναρτήσεων έχουν διαφορετική μορφή: η ΟΕΔ μιας συνάρτησης που παίρνει $a \geq 0$ παραμέτρους και περιέχει $n \geq 0$ εμφωλευμένων ταιριασμάτων προτύπων, έχει a πεδία με thunk και n πεδία *nested*. Αυτή η πληροφορία πρέπει να αποθηκευτεί σε κάθε ΟΕΔ, ώστε ο συλλέκτης σκουπιδιών να ξέρει τι μορφή έχει. Μια συνάρτηση μπορεί να μη δέχεται παραμέτρους ή να μην κάνει καθόλου ταίριασμα προτύπων, επομένως το μόνο μέρος μιας ΟΕΔ που υπάρχει πάντα είναι το πεδίο *prev*. Μπορούμε έτσι να χρησιμοποιήσουμε τα υψηλότερα 16 bit για να αποθηκεύσουμε τα a και n ως ένα ζεύγος αριθμών 8-bit. (Αυτό στην πράξη περιορίζει την υλοποίησή μας να υποστηρίζει μόνο συναρτήσεις που παίρνουν μέχρι 255 παραμέτρους και έχουν εμφωλευμένες εκφράσεις ταιριάσματος προτύπων μέχρι 255 επίπεδα, κάτι που στην πράξη δεν είναι τόσο περιοριστικό.) Το bit 0 χρησιμοποιείται επίσης από τον συλλέκτη σκουπιδιών και θα παρουσιαστεί αργότερα σε αυτήν την ενότητα.

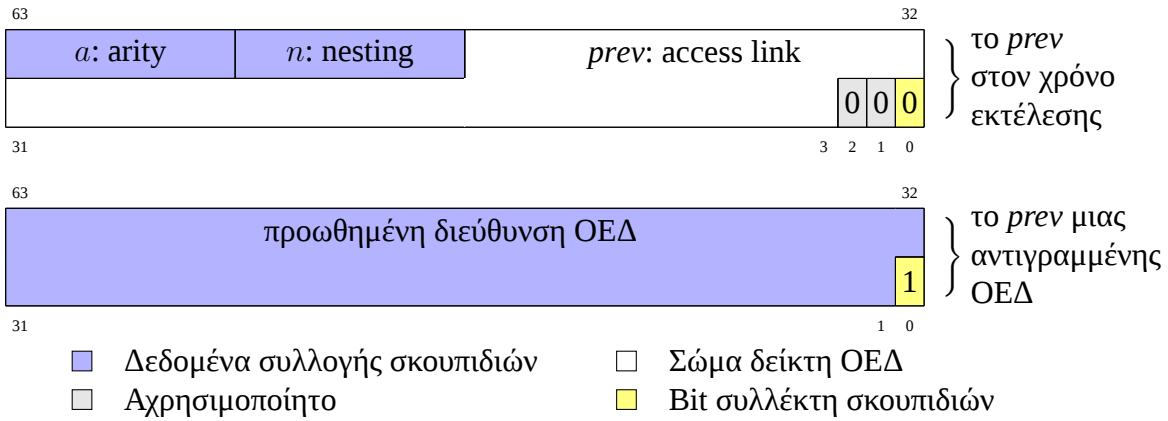
Ο δείκτης *prev* αποκτά τη μορφή που φαίνεται στην Εικόνα 6.3. Στον κώδικα που ακολουθεί, η μάσκα `PTRMASK` απομονώνει το σώμα του δείκτη και η `ARINFO` γεμίζει τα αντίστοιχα πεδία του δείκτη *prev* κατά την κατασκευή της ΟΕΔ:

```
#define PTRMASK          0x0000fffffffffff8
#define ARINFO(a,n,prev) (((uintptr_t) a << 56) \
                        | ((uintptr_t) n << 48) \
                        | ((uintptr_t) prev & PTRMASK))
```

Τα δύο πεδία a και n μπορούν να απομονωθούν με τις εξής μακροεντολές:

```
#define AR_a(p) ((uintptr_t) p >> 56)
#define AR_n(p) (((uintptr_t) p >> 48) & 0xff)
```

[4] Μπορούμε να τροποποιήσουμε έναν τέτοιο συλλέκτη σκουπιδιών ώστε να μπορεί να αναγνωρίσει τους δείκτες μας, αλλά δεν είναι ξεκάθαρο πόσο καλά θα λειτουργούσε κάτι τέτοιο για συντηρητική συλλογή σκουπιδιών.



Εικόνα 6.3: Η πληροφορία συλλογής σκουπιδιών που ενσωματώνεται στο πεδίο *prev*.

Τα τρία χαμηλότερα bit του δείκτη *prev* είναι αρχικά μηδέν λόγω της `PTRMASK` και το bit του συλλέκτη σκουπιδιών είναι πάντα μηδέν όταν ο συλλέκτης δεν εκτελείται. Έτσι, όταν το πρόγραμμα δεν κάνει συλλογή σκουπιδιών, ο δείκτης *prev* μπορεί να διαβαστεί χωρίς να χρειάζεται να μηδενιστούν τα χαμηλά bit του:

```
#define AR_prev(p) ((intptr_t) p << 16 >> 16)
```

Ένας συλλέκτης σκουπιδιών αντιγραφής χρειάζεται χώρο στη μνήμη για να αποθηκεύσει την *προωθημένη διεύθυνση* (*forwarded address*) [136] ενός ήδη αντιγραμμένου αντικειμένου — στην πράξη, οι υλοποιήσεις συχνά επαναχρησιμοποιούν προσωρινά κάποιον χώρο στο αντικείμενο προς αντιγραφή για να κρατούν αυτό το πεδίο. Στη δική μας περίπτωση, επαναχρησιμοποιούμε τον δείκτη *prev* για να αποθηκεύσουμε την προωθημένη διεύθυνση της ΟΕΔ και θέτουμε σε αληθές το bit 0 του πεδίου *prev* για να δείξουμε ότι η ΟΕΔ έχει πια προωθηθεί και άρα μπορούμε να πάρουμε τη νέα της διεύθυνση, με τις ακόλουθες μακροεντολές `IS_FORWARDED` και `FORWARDED_ADDR`:[5]

```
#define IS_FORWARDED(lar)   ((uintptr_t) lar->prev & 1)
#define FORWARDED_ADDR(lar) ((uintptr_t) lar->prev & ~1)
```

Η αναπαράσταση που προκύπτει φαίνεται στο κάτω μέρος της Εικόνας 6.3.

6.4 Μετρήσεις

Μετρήσαμε την υλοποίησή μας με δύο τρόπους: (α) συγκρίναμε την απόδοσή της με τον GHC, για να βεβαιωθούμε ότι είναι αρκετά γρήγορη, και (β) τη συγκρίναμε με την προηγούμενη αναπαράσταση που περιγράφηκε στο Κεφάλαιο 5.

6.4.1 Σύγκριση με τον GHC

Μετρήσαμε την ταχύτητα των προγραμμάτων που προκύπτουν από την υλοποίησή μας χρησιμοποιώντας ένα σύνολο από πρότυπα δοκιμαστικά προγράμματα Haskell που μεταγλωττίστηκαν από τον GIC και τον GHC, συγκρίνοντας τον χρόνο εκτέλεσής τους. Φυσικά τα δοκιμαστικά αυτά προγράμματα δεν συγκρίνουν μόνο την αποδοτικότητα της αναπαράστασης μνήμης που παρουσιάζουμε σε αυτό το κεφάλαιο. Βασίζονται στην αποδοτικότητα

[5] Η χρήση κατά τη συλλογή σκουπιδιών ενός bit για να κρατά προσωρινή πληροφορία είναι αρκετά συχνή: για παράδειγμα, η τεχνική είχε ήδη εφαρμοστεί στον πρώτο συλλέκτη σκουπιδιών της LISP 1.5 [170].

ολόκληρου του GIC, δηλαδή εξαρτώνται από τον μετασχηματισμό defunctionalization και τον νοηματικό μετασχηματισμό.

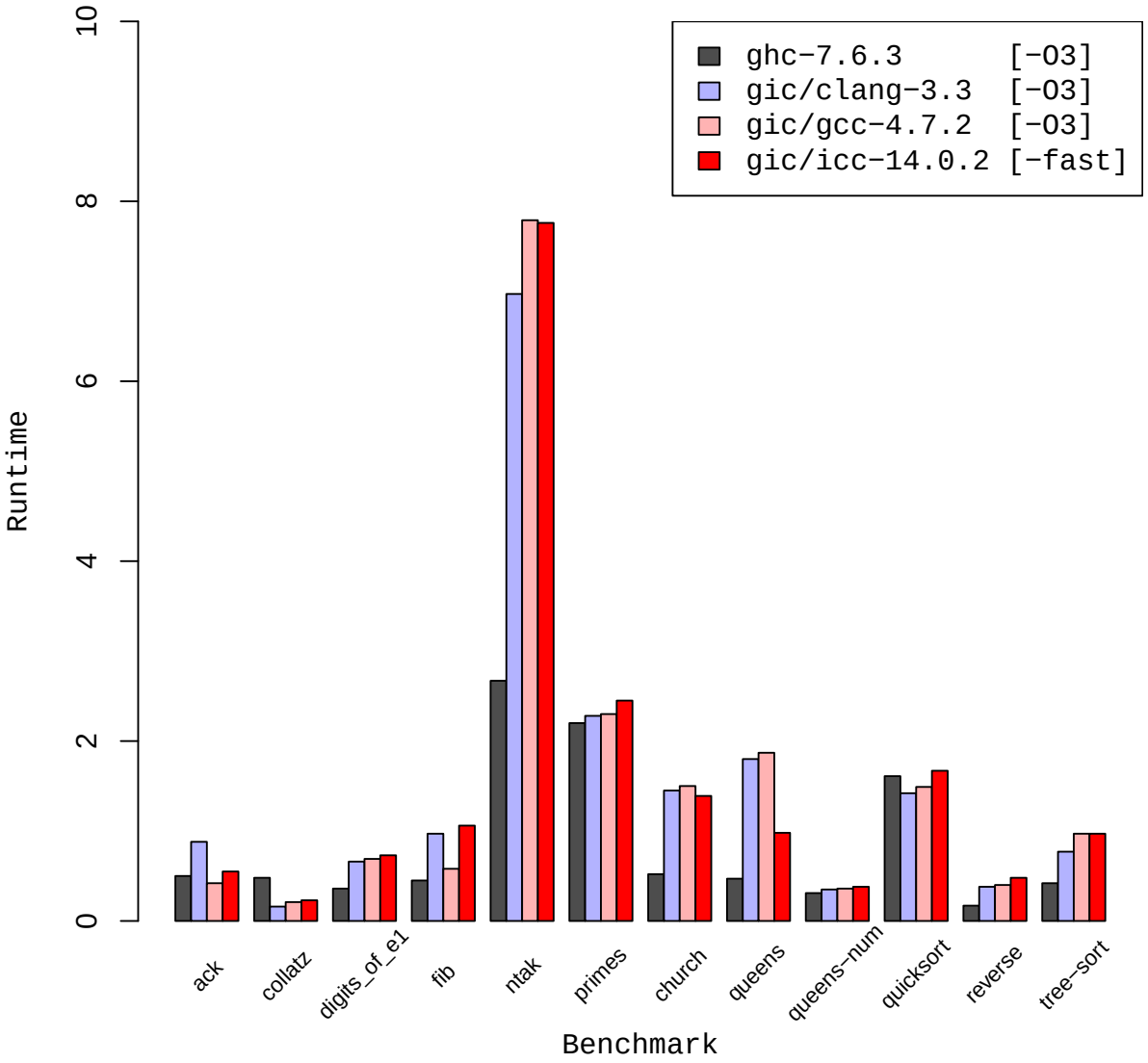
Ο GHC είναι ο κορυφαίος μεταγλωττιστής της Haskell και αποτελεί καρπό δεκαετιών έρευνας πάνω σε βελτιστοποιήσεις κώδικα και σε τεχνικές υλοποίησης οκνηρών γλωσσών. Αντίθετα, ο δικός μας μεταγλωττιστής είναι στα αρχικά στάδια της ανάπτυξής του και του λείπουν πολλά χαρακτηριστικά· για παράδειγμα δεν κάνει κανέναν μετασχηματισμό βελτιστοποίησης κώδικα (π.χ. fusion ή inlining). Χρησιμοποιεί όμως δύο απλές στατικές αναλύσεις για βελτιστοποίηση: (i) την ανάλυση μοιράσματος που αναφέραμε στην Ενότητα 4.3 και (ii) την ανάλυση διαφυγής που αναφέραμε στην Ενότητα 5.3.

Οι μετρήσεις έγιναν σε έναν υπολογιστή με τέσσερις επεξεργαστές Intel Xeon® E7340 (2.40 GHz), με συνολικά 16 πυρήνες, 4 MB κρυφή μνήμη και 16 GB RAM, και λειτουργικό σύστημα Debian GNU/Linux 7.3. Ως μεταγλωττιστή αναφοράς της Haskell χρησιμοποιήσαμε τον GHC 7.6.3 και ως μεταγλωττιστή της C για τον GIC τον LLVM 3.3/Clang. Μεταγλωττίσαμε επίσης τα δοκιμαστικά προγράμματά μας με τον GCC και τον Intel C Compiler, για να αποκλείσουμε τυχαίες διαφορές στην ταχύτητα ανάμεσα σε διαφορετικούς μεταγλωττιστές. Ο GHC χρησιμοποιήθηκε με ενεργοποιημένες όλες τις βελτιστοποιήσεις (`-O3`), χρησιμοποιώντας την προκαθορισμένη του γεννήτρια κώδικα. Προστέθηκαν σημειώσεις τύπων σε όλα τα δοκιμαστικά προγράμματα, ώστε να αφαιρεθεί πολυμορφισμός που δεν χρειαζόταν (και που ειδικά ο GHC εισάγει όταν μεταγλωττίζει προγράμματα που κάνουν αριθμητικές πράξεις). Ο κώδικας C που παρήγαγε ο GIC μεταγλωττίστηκε από όλους τους μεταγλωττιστές της C, πάντα με όλες τις βελτιστοποιήσεις τους ενεργοποιημένες.

Η Εικόνα 6.4 δείχνει την σύγκριση διάφορων δοκιμαστικών προγραμμάτων, μεταγλωττισμένα με τον GHC και τον GIC. Βλέπουμε ότι (α) το `collatz` εκτελείται γρηγορότερα όταν μεταγλωττίζεται με τον GIC, (β) τα `primes`, `queens-num` και `quicksort` έχουν περίπου την ίδια ταχύτητα και (γ) τα υπόλοιπα (`ack`, `digits_of_e1`, `fib`, `ntak`, `church`, `queens`, `reverse`, `tree-sort`) εκτελούνται γρηγορότερα από τον GHC. Διαπιστώνουμε επίσης ότι ο Clang, ο GCC και ο Intel C Compiler δεν έχουν σημαντικές διαφορές στην ταχύτητα αυτών των προγραμμάτων, αν και υπάρχουν περιπτώσεις που ο πρώτος (`ntak`, `tree-sort`), ο δεύτερος (`fib`), ή ο τρίτος (`ack`, `queens`) έχουν καλύτερη ταχύτητα.

Η Εικόνα 6.5 δείχνει την συμπεριφορά κρυφής μνήμης των δοκιμαστικών προγραμμάτων, όπως μετράται από το εργαλείο Cachegrind [191]. Βλέπουμε ότι (α) το `collatz`, το `digits_of_e1` και το `queens-num` έχουν λιγότερες αστοχίες όταν μεταγλωττίζονται με τον GIC, (β) το `ntak`, το `primes` και το `queens` έχουν παρόμοιες αστοχίες και με τους δύο μεταγλωττιστές, ενώ (γ) τα `ack`, `church`, `fib`, `quick-sort`, `reverse` και `tree-sort` έχουν λιγότερες αστοχίες όταν μεταγλωττίζονται με τον GHC.

Το πρόγραμμα `church` χρησιμοποιεί πολλά κλεισίματα, κάτι που πιέζει την απλή υλοποίηση του μετασχηματισμού defunctionalization που χρησιμοποιούμε (όπως σημειώσαμε στην Ενότητα 2.2, η κωδικοποίηση Church είναι ο αντίστροφος μετασχηματισμός του defunctionalization). Η χειρότερη ταχύτητα που έχει ο GIC σε σχέση με τον GHC μπορεί να οφείλεται σε κάποια βελτιστοποίηση που λείπει ή να σημαίνει ότι ο μετασχηματισμός defunctionalization που υλοποιήσαμε δεν αρκεί για προγράμματα που χρησιμοποιούν πολλά κλεισίματα. Το τελευταίο θα μπορούσε να αντιμετωπιστεί αν χρησιμοποιούσαμε ειδικές βελτιστοποιήσεις, όπως η ανάλυση ροής του μεταγλωττιστή MLton, ο οποίος επίσης χρησιμοποιεί τον μετασχηματισμό defunctionalization [54]. Πρέπει να τονίσουμε όμως ότι ο μετασχηματισμός defunctionalization δεν συνδέεται με την αναπαράσταση που παρουσιάζουμε σε αυτό το κεφάλαιο· επιλέξαμε να δείξουμε αυτό το δοκιμαστικό πρόγραμμα για λόγους πληρότητας, επειδή η συγγραφή προγραμμάτων υψηλότερης τάξης είναι βασικό χαρακτηριστικό του στυλ προγραμματισμού σε Haskell.



Εικόνα 6.4: Χρόνος εκτέλεσης των δοκιμαστικών προγραμμάτων που μεταγλωττίστηκαν με τον GHC και τον GIC (οι μικρότερες τιμές είναι καλύτερες).

Κατά τη διάρκεια των μετρήσεων, βρήκαμε επίσης ότι οι απλοποιημένες πράξεις αριθμητικής της Ενότητας 6.2 (όπως η `SUM`) δεν έδειξαν κάποια διαφορά στην ταχύτητα εκτέλεσης των δοκιμαστικών προγραμμάτων.

Γενικά οι μετρήσεις δείχνουν καλή ταχύτητα για την υλοποίησή μας, μιας και είναι σχετικά απλή (7908 γραμμές σε Haskell και C)[6] και δεν περιέχει τις περισσότερες από τις βελτιστοποιήσεις που περιλαμβάνει ο GHC. Στα μικρά προγράμματα που δοκιμάσαμε, η αναπαράσταση των thunk και των ΟΕΔ έχει σαν αποτέλεσμα ταχύτητα συγκρίσιμη με αυτή του κορυφαίου μεταγλωττιστή GHC.

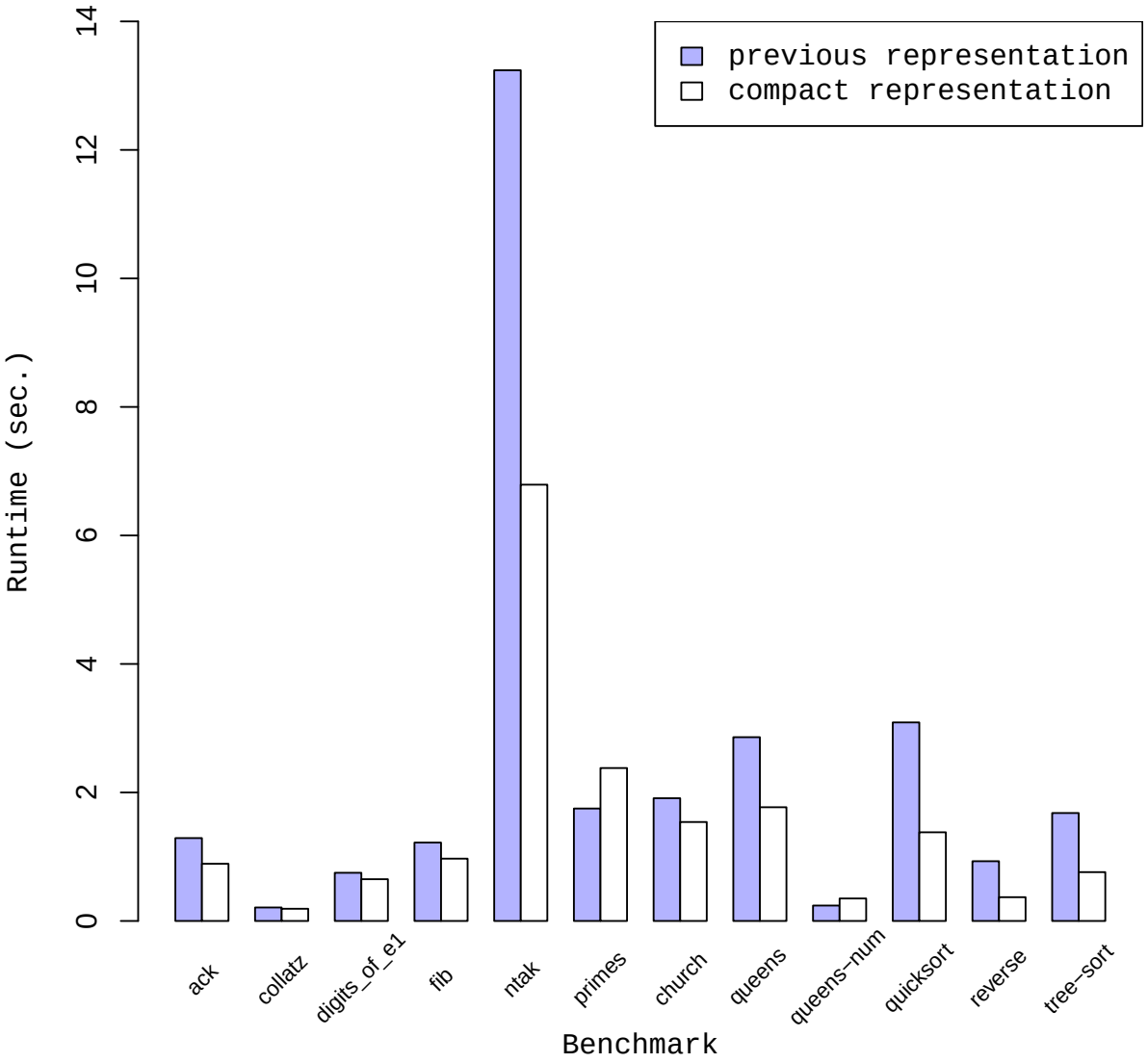
6.4.2 Σύγκριση με την προηγούμενη αναπαράσταση

Συγκρίναμε την συμπαγή αναπαράσταση που παρουσιάσαμε με την αναπαράσταση που περιγράψαμε στην Ενότητα 5.2, όπου κάθε thunk χρησιμοποιούσε περισσότερο χώρο (τρεις λέξεις). Οι ΟΕΔ κρατούσαν επίσης πληροφορία για τον συλλέκτη σκουπιδιών, σε μια ξεχω-

[6] Οι γραμμές κώδικα μετρήθηκαν από το εργαλείο `SLOCCount` του David A. Wheeler.

	GHC					GIC (προηγούμενη αναπαράσταση)					GIC (αποδοτική αναπαράσταση)				
	I1	LLi	D1	LLd	LL	I1	LLi	D1	LLd	LL	I1	LLi	D1	LLd	LL
ack	0.0	0.0	6.1	2.1	0.3	0.0	0.0	10.1	0.0	0.0	0.0	0.0	10.8	0.0	0.0
church	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
collatz	0.0	0.0	0.7	0.0	0.0	0.0	0.0	0.3	0.2	0.1	0.0	0.0	0.1	0.1	0.0
digits_of_e1	0.0	0.0	3.2	0.0	0.0	0.0	0.0	2.4	1.8	0.6	0.0	0.0	1.3	0.8	0.2
fib	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ntak	0.0	0.0	1.6	0.0	0.0	0.0	0.0	2.4	1.9	0.8	0.0	0.0	1.0	0.9	0.2
primes	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0	0.1	0.1	0.0
queens	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.8	0.6	0.2	0.0	0.0	0.4	0.3	0.0
queens-num	0.0	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
quick-sort	0.0	0.0	4.2	0.5	0.1	0.0	0.0	11.2	3.9	1.6	0.0	0.0	8.2	1.7	0.5
reverse	0.0	0.0	8.2	0.0	0.0	0.0	0.0	23.4	8.0	3.3	0.0	0.0	15.0	3.6	1.0
tree-sort	0.0	0.0	7.3	0.0	0.0	0.0	0.0	14.4	3.4	1.4	0.0	0.0	7.9	1.6	0.4

Εικόνα 6.5: Ποσοστά αστοχίας της κρυφής μνήμης (cache miss rates) που αναφέρονται από το εργαλείο Cachegrind (%). Οι μηδενικές τιμές εμφανίζονται με γκρίζο χρώμα.
I1: κρυφή μνήμη εντολών πρώτου επιπέδου (first-level instruction cache).
LLi: κρυφή μνήμη εντολών τελευταίου επιπέδου (last-level instruction cache).
D1: κρυφή μνήμη δεδομένων πρώτου επιπέδου (first-level data cache).
LLd: κρυφή μνήμη δεδομένων τελευταίου επιπέδου (last-level data cache).
LL: συνδυασμένη κρυφή μνήμη τελευταίου επιπέδου (last-level combined cache).



Εικόνα 6.6: Σύγκριση χρόνου εκτέλεσης ανάμεσα στις δύο αναπαραστάσεις.

ριστή λέξη μηχανής, στο πεδίο *layout*.

Μεταγλωττίσαμε τα δοκιμαστικά προγράμματα χρησιμοποιώντας και τις δύο αναπαραστάσεις και τα εκτελέσαμε, συγκρίνοντας χρόνους εκτέλεσης, χρήσης μνήμης και συμπεριφορά κρυφής μνήμης.

Η πρώτη μέτρηση φαίνεται στην Εικόνα 6.6 και συγκρίνει τους χρόνους εκτέλεσης, όπου η συμπαγής αναπαράσταση εκτελείται γρηγορότερα για τα περισσότερα προγράμματα, εκτός από το `primes` και το `queens-num`. Η χειρότερη ταχύτητα των δύο τελευταίων προγραμμάτων μάλλον δείχνει ότι οι συχνές αριθμητικές πράξεις (όπως ο έλεγχος για πρώτους αριθμούς και οι έλεγχοι για N βασίλισσες) μπορεί να έχουν σημαντικό κόστος σε μια συμπαγή αναπαράσταση.

Οι μετρήσεις στην Εικόνα 6.5 εμφανίζουν τη συμπεριφορά της κρυφής μνήμης των δύο αναπαραστάσεων: εδώ η συμπαγής αναπαράσταση προκαλεί λιγότερες αστοχίες στην προσομοίωση ιεραρχίας κρυφής μνήμης του Cachegrind, με την εξαίρεση του `ack`, που έχει λίγο περισσότερες αστοχίες κρυφής μνήμης δεδομένων πρώτης τάξης. Κάποια προγράμματα δείχνουν μικρή βελτίωση (όπως το `primes`), ενώ άλλα σημαντικό πλεονέκτημα (όπως το `quick-sort`, το `reverse` και το `tree-sort`).

Όπως περιμέναμε, η νέα αναπαράσταση χρειάζεται λιγότερο χώρο στη μνήμη σε σχέση

Χρήση σωρού σε byte			
Πρόγραμμα	Προηγούμενη	Συμπαγής	Μείωση
collatz,	41,360,250	13,200,080	68.1%
digits_of_e1	602,368,070	192,246,560	68.1%
ntak	11,516,398,810	3,636,757,520	68.4%
church	2,198,570	672,690	69.4%
queens	1,503,132,770	498,798,920	66.8%
queens-num	8,140	2,600	68.1%
quicksort	2,788,938,330	875,448,910	68.6%
reverse	685,380,260	216,432,080	68.4%
tree-sort	1,652,780,690	525,870,220	68.2%

Εικόνα 6.7: Η χρήση μνήμης των δύο αναπαραστάσεων, χωρίς συλλογή σκουπιδιών.

με την προηγούμενη αναπαράσταση. Στον πίνακα της Εικόνας 6.7 φαίνεται πόσος χώρος στον σωρό χρειάστηκε από τα προγράμματα, θεωρώντας ότι δεν έγινε συλλογή σκουπιδιών. Δεν δείχνουμε τα `ack`, `fib` και `primes`, γιατί αυτά χρησιμοποιούν μόνο τη στοίβα. Γενικά υπάρχει μια μείωση κατά μέσο όρο 68.2% στη χρήση του σωρού· αυτή η μείωση είναι πάνω από τα δύο τρίτα της μνήμης και εξηγείται γιατί οι συναρτήσεις παίρνουν λίγες παραμέτρους με αποτέλεσμα να είναι εμφανής επιπλέον η εξοικονόμηση σε χώρο λόγω της απάλειψης των ξεχωριστών μεταδεδομένων συλλογής σκουπιδιών των ΟΕΔ. Δε δείχνουμε μετρήσεις από τη στοίβα· όλα τα προγράμματα έτρεξαν με μέγιστο μέγεθος στοίβας τα 262,144 byte, και παρατηρείται και εκεί παρόμοια μείωση, επειδή στη στοίβα αποθηκεύονται τα ίδια δεδομένα (ΟΕΔ με πεδία thunk).

Τα αποτελέσματα δείχνουν ότι στις περισσότερες περιπτώσεις η συμπαγής αναπαράσταση είναι καλύτερη από την προηγούμενη: χρειάζεται το ένα τρίτο της μνήμης, είναι συνήθως γρηγορότερη και έχει καλύτερα χαρακτηριστικά κρυφής μνήμης. Επίσης, η μικρή χρήση μνήμης κάνει την τεχνική μας κατάλληλη για περιβάλλοντα με περιορισμένη μνήμη, όπως τα ενσωματωμένα συστήματα (embedded systems).

6.5 Σχετική έρευνα

Η απόκρυψη πληροφορίας στα χαμηλά bit δεικτών που δείχνουν σε στοιχισμένες διευθύνσεις στη μνήμη (*pointer tagging*) είναι παλιά και δοκιμασμένη τεχνική των υλοποιήσεων γλωσσών προγραμματισμού [115]. Για παράδειγμα, η Smalltalk ενσωμάτωνε προσημασμένους ακεραίους των 15 bit (αντικείμενα της κλάσης *SmallInteger*) σε δείκτες προς στοιχισμένες διευθύνσεις των 16 bit [110] και οι κατασκευαστές της ML είχαν υλοποιηθεί με χρήση τέτοιων δεικτών (tagged pointers) [45]. Ένα πιο σύγχρονο παράδειγμα είναι το σύστημα χρόνου εκτέλεσης του περιβάλλοντος Erlang/OTP, που χρησιμοποιεί ένα περίπλοκο σύστημα ετικετών για να αναπαριστά πολλά αντικείμενα χρησιμοποιώντας μια λέξη μηχανής (ή ακόμα και μισή λέξη σε συστήματα 64-bit) [207].

Οι υλοποιήσεις των οκνηρών γλωσσών προγραμματισμού έχουν επίσης χρησιμοποιήσει tagged pointers [151]. Ειδικότερα, ο GHC χρησιμοποιεί τέτοιους δείκτες [168] για να κωδικοποιεί κατασκευαστές τύπων δεδομένων με λίγα διαφορετικά αναγνωριστικά κατασκευαστών. Η τεχνική μας μπορεί να θεωρηθεί γενίκευση αυτής της τεχνικής, επιτρέποντας τη χρήση τύπων δεδομένων με πολλούς κατασκευαστές. Ο GHC βασίζεται στην αφηρημένη

μηχανή STG [137], η οποία υποθέτει ότι τα thunk έχουν περισσότερη πληροφορία από τα δικά μας, περιλαμβάνοντας έναν δείκτη (*info pointer*) προς χρήσιμα μεταδεδομένα (όπως πληροφορίες αποσφαλμάτωσης και παράλληλης εκτέλεσης). Η αναπαράστασή μας δεν υποστηρίζει πληροφορίες αποσφαλμάτωσης. Για να υποστηρίξει ταυτόχρονο υπολογισμό ενός thunk, μια λύση θα ήταν να προστεθεί ένα επιπλέον πεδίο στο thunk που να περιλαμβάνει ένα lock ή κάποιον άλλο ισοδύναμο μηχανισμό συγχρονισμού. Μπορούμε όμως να κρατήσουμε τα συμπαγή thunk μιας λέξης, αν χρησιμοποιούσαμε μια εντολή test-and-set στο υψηλότερο bit του thunk (μειώνοντας τα αναγνωριστικά κατασκευαστών στα 15 bit και τις πρωτόγονες τιμές στα 61 bit), υλοποιώντας στην πράξη ένα spin lock με αυτό το bit. Εναλλακτικά, ένα ή περισσότερα bit μπορούν να χρησιμοποιηθούν για να υλοποιηθεί πιο πολύπλοκος συγχρονισμός, όπως η τεχνική μαύρης τρύπας (black-hole) που χρησιμοποιείται από τον GHC [167], οπότε και το σύστημα του χρόνου εκτέλεσης πρέπει να κρατά επιπλέον μνήμη για διαχειριστικούς λόγους (όπως είναι για παράδειγμα οι ουρές ανενεργών υπολογισμών). Η ενσωμάτωση της σημαίας του thunk μέσα σε έναν δείκτη έχει επίσης χρησιμοποιηθεί και σε άλλες παλαιότερες υλοποιήσεις οκνηρών γλωσσών, όπως αναφέρει ο Hammond [117].

Η διάδοση της αρχιτεκτονικής AMD64, με τις διευθύνσεις 48 bit που αυτή χρησιμοποιεί, έχει ωθήσει και άλλες υλοποιήσεις γλωσσών με στατικούς τύπους να εκμεταλλευτούν τα ελεύθερα bit υψηλής τάξης στους δείκτες. Για παράδειγμα, τα bit αυτά χρησιμοποιούνται σε νεότερες εκδόσεις του συστήματος χρόνου εκτέλεσης της Objective-C για τα λειτουργικά συστήματα Mac OS X και iOS [20].

Μια άλλη τεχνική που ενσωματώνει πληροφορία σε άλλες δομές είναι η *NaN-boxing*, η οποία εκμεταλλεύεται την πλεονάζουσα πληροφορία στην αναπαράσταση αριθμών κινητής υποδιαστολής IEEE-754 [115]. Και αυτή η τεχνική είναι αρκετά δημοφιλής και χρησιμοποιείται σε πρόσφατες υλοποιήσεις δυναμικών γλωσσών όπως η Lua [204], η JavaScript [26] και η Ruby [184].

Μια ενδιαφέρουσα εφαρμογή της τεχνικής NaN-boxing στη Haskell ήταν η κωδικοποίηση του Caro που ενσωμάτωνε δείκτες των 52 bit μέσα σε αριθμούς κινητής υποδιαστολής διπλής ακρίβειας των 64 bit στη γλώσσα pH [49]. Σε σύγκριση με την αναπαράστασή μας, η κωδικοποίηση αυτή υποστήριζε ένα άλλο υποσύνολο των ακεραίων ($-2^{63} \ldots 2^{52} - 1$), πριμοδοτώντας την απευθείας αναπαράσταση των αριθμών κινητής υποδιαστολής διπλής ακρίβειας. Η αναπαράστασή μας έχει αρκετό χώρο για αριθμούς κινητής υποδιαστολής απλής ακρίβειας· για να υποστηρίξει διπλής ακρίβειας, θα μπορούσε επίσης να χρησιμοποιήσει κάποια τεχνική τύπου NaN-boxing. Η κωδικοποίηση του Caro χειριζόταν I-δομές (I-structures) και M-δομές (M-structures), που έχουν παράλληλη σημασιολογία και δεν είναι μη αυστηρές αλλά υλοποιούνται με lenient evaluation [50]. Η αναπαράσταση αυτή χρειαζόταν επιπλέον χώρο, εκτός των 64 bit, για τις σημαίες του thunk και του συγχρονισμού ενώ η αναπαράστασή μας ξεκινά από τη σειριακή οκνηρή αποτίμηση και χειρίζεται ανεξάρτητα την παράλληλη αποτίμηση, κρατώντας όλη την πληροφορία σε μια λέξη των 64 bit. Ο Caro επίσης τροποποίησε την τεχνική του για να χωρέσει όλα τα δεδομένα σε μια λέξη των 64 bit, μειώνοντας τον χώρο διευθύνσεων στα 44 bit και υποστηρίζοντας περισσότερους ακεραίους ($-2^{63} \ldots 2^{63} - 1 - 2^{51}$) [50].

Ο Appel τόνισε ότι οι υλοποιήσεις των γλωσσών με στατικούς τύπους έχουν όλη την απαραίτητη πληροφορία διαθέσιμη για να αποφύγουν τη διατήρηση μεταδεδομένων για τον συλλέκτη σκουπιδιών κατά τον χρόνο εκτέλεσης [13]. Επειδή μελέτησε μια αυστηρή γλώσσα, η παρατήρησή του δεν περιλαμβάνει τη σημαία του thunk, η οποία υπάρχει με κάποιον τρόπο σε όλες τις υλοποιήσεις οκνηρών γλωσσών προγραμματισμού. Η τεχνική του θα μπορούσε να εφαρμοστεί και στη δική μας υλοποίηση, αν απαλείφαμε τα πεδία a και n από τις ΟΕΔ, καθώς και το πεδίο τιμής από τα thunk, αντικαθιστώντας τα με στατική πληροφορία τύπων

για κάθε ΟΕΔ. Αυτή η πληροφορία θα μπορούσε να ενσωματωθεί σε μια ΟΕΔ ή να κρατηθεί σε ξεχωριστό χώρο. Παρόλα αυτά, ο Appel σημειώνει επίσης ότι μια τέτοια αλλαγή μπορεί να μη βελτιώσει την ταχύτητα στην πράξη.

Η *συμπίεση δεικτών* (*pointer compression*) είναι μια τεχνική που προσπαθεί να μειώσει τον χώρο που χρησιμοποιείται από τους δείκτες, όταν ο χώρος διευθύνσεων μπορεί να συμπιεστεί, όπως για παράδειγμα όταν τα δεδομένα μπορεί να χρειάζονται μόνο τμήμα όλης της μνήμης που μπορεί να δεικτοδοτηθεί [35, 153], ή όταν διαφορετικές διευθύνσεις μοιράζονται κάποιο κοινό αρχικό τμήμα [306]. Η τεχνική έχει εφαρμοστεί με επιτυχία στις εικονικές μηχανές της Java [3, 200, 283, 284]. Η προσέγγισή μας διαφέρει γιατί μπορεί να χρησιμοποιήσει ολόκληρο τον χώρο διευθύνσεων, χωρίς να υποθέτει κάτι για το μέγεθος ή τη χρήση των δομών δεδομένων του προγράμματος.

Τα συστήματα χρόνου εκτέλεσης που χρησιμοποιούν tagged pointers μπορούν να θεωρηθούν λογισμικό που αντιστοιχεί σε αρχιτεκτονικές υλικού με υποστήριξη tags για συγκεκριμένες γλώσσες. Τέτοιες αρχιτεκτονικές είχαν προταθεί στο παρελθόν για την εκτέλεση συναρτησιακών γλωσσών προγραμματισμού (όπως οι Lisp machines [111]) αλλά βρέθηκε ότι δεν χρειάζονταν για τις γλώσσες χωρίς δυναμικούς τύπους που χρησιμοποιούνταν [105]. Οι μετρήσεις μας δείχνουν ότι το υλικό AMD64, αν και ριζικά διαφορετικό, μπορεί να προγραμματιστεί αποδοτικά χρησιμοποιώντας τεχνικές για τέτοιες αρχιτεκτονικές.

Ένα παράπλευρο αποτέλεσμα της δουλειάς που παρουσιάσαμε είναι ότι η C είναι καλή γλώσσα για γεννήτρια κώδικα μιας οκνηρής υλοποίησης και προσφέρει έτοιμες σημαντικές τεχνολογίες μεταγλώττισης, κάτι το οποίο επισήμαναν πρόσφατα και οι Liu *et al.* κατά την ανάπτυξη του Intel Labs Haskell Research Compiler [161].

Τέλος, η αρχιτεκτονική AMD64 προσφέρει εντολές όπως οι `INSERTQ` και `EXTRQ`, οι οποίες λειτουργούν πάνω σε τμήματα λέξεων των 64 bit [11] και μπορούν να χρησιμοποιηθούν σε κάποια γεννήτρια κώδικα μηχανής που να χρησιμοποιεί την αναπαράστασή μας.

6.6 Πρόσθετες παρατηρήσεις πάνω στην αναπαράσταση

Η τεχνική μας δεν εφαρμόζεται μόνο σε σύγχρονα συστήματα AMD64· μελλοντικές επεκτάσεις της αρχιτεκτονικής μπορεί να αλλάξουν τον χώρο διευθύνσεων σε 52 bit [12, §1.1.2], που αφήνει πάλι αρκετό χώρο για τα αναγνωριστικά των κατασκευαστών. Η τεχνική μας μπορεί επίσης να χρησιμοποιηθεί στην υλοποίηση οκνηρών γλωσσών σε επεξεργαστές ARMv8 και SPARC των 64 bit, όπου οι δείκτες επίσης έχουν παρόμοια πλεονάζουσα πληροφορία [159, 199]. Σημειώνουμε επίσης ότι κάποιες εκδόσεις του λειτουργικού συστήματος Microsoft Windows προσφέρουν επιπλέον πλεονάζοντα χώρο στους δείκτες τους, έχοντας μόνο διευθύνσεις των 44 bit σε υλικό AMD64 [58].

Η αναπαράστασή μας μπορεί να έχει πλεονεκτήματα σε κάποιες παράλληλες υλοποιήσεις οκνηρών γλωσσών. Η ταυτόχρονη τροποποίηση διαφορετικών πεδίων από bit μέσα στην ίδια λέξη μηχανής είναι ατομική πράξη (atomic operation) και μπορεί να αποκλείσει κάποια προβλήματα (data races) σε συνθήκες ταυτόχρονης τροποποίησης δεδομένων. Πιστεύουμε επίσης ότι η τεχνική μας μπορεί να προσαρμοστεί για χρήση σε μεταγλωττιστές αυστηρών γλωσσών προγραμματισμού.

Πρέπει να επαναλάβουμε ότι ο μετασχηματισμός defunctionalization δεν χρειάζεται για να λειτουργήσει η αναπαράσταση που δείξαμε αλλά χρησιμοποιήθηκε για να απλοποιηθεί η υλοποίηση. Όπως φαίνεται στην Εικόνα 6.2, το bit 2 ενός υπολογισμένου κατασκευαστή είναι αχρησιμοποίητο και μπορεί να χρησιμοποιηθεί για να διακρίνει μια ξεχωριστή αναπαράσταση για κλεισίματα. Στην πράξη κάτι τέτοιο φαίνεται θα έπρεπε να υλοποιηθεί μιας

και τα αναγνωριστικά κατασκευαστών των 16 bit σημαίνουν ότι μπορούν να υπάρχουν το πολύ 2^{16} διαφορετικοί κατασκευαστές κλεισιμάτων για κάθε συνάρτηση `apply` σε όλο το πρόγραμμα, κάτι που αποτελεί περιορισμό για μεγάλα προγράμματα στην πράξη.

Ακόμα και στην αναπαράσταση που παρουσιάσαμε σε αυτό το κεφάλαιο, εξακολουθεί να υπάρχει πλεονάζουσα πληροφορία. Ειδικότερα, στους δείκτες *prev* περισσεύουν bit που μπορούν να χρησιμοποιηθούν για παράδειγμα σαν επιπλέον χώρος από κάποιον πιο πολύπλοκο συλλέκτη σκουπιδιών ή από έναν μεταγλωττιστή JIT όπως αυτός του Schilling [252]. Οι δείκτες κώδικα έχουν αχρησιμοποίητα τα bit 1-2 και 48-63, όπου θα μπορούσε να αποθηκευτεί πληροφορία για μη υπολογισμένα thunk. Οι δείκτες στα πεδία *nested* υπάρχουν μόνο όταν μια συνάρτηση κάνει ταίριασμα προτύπων και άρα ο αχρησιμοποίητος χώρος τους θα μπορούσε να χρησιμοποιηθεί για σχετικές πληροφορίες. Τέλος, τα 16 bit για αναγνωριστικά κατασκευαστών μπορεί να είναι πολλά στην πράξη – κάποια από αυτά μπορούν να χρησιμοποιηθούν για άλλη πληροφορία σχετική με τους κατασκευαστές.

Περίληψη

Παρουσιάσαμε μια βελτιωμένη αναπαράσταση για την τεχνική υλοποίησης με ΟΕΔ του Κεφαλαίου 5, όπου η πλεονάζουσα πληροφορία των δεικτών της αρχιτεκτονικής AMD64 χρησιμοποιείται για την αποθήκευση ενός thunk σε μια μόνο λέξη μηχανής των 64 bit. Τα παραγόμενα προγράμματα είναι σχετικά γρήγορα συγκρινόμενα με τον κορυφαίο μεταγλωττιστή της Haskell, και έχουν συμπαγή αναπαράσταση στη μνήμη.

Οι υλοποιήσεις που έχουμε δει μέχρι στιγμής μπορούν να δέχονται μόνο ολόκληρα προγράμματα και δεν έχουν υποστήριξη για μερική μεταγλώττιση ή χρήση εξωτερικών βιβλιοθηκών. Στο επόμενο κεφάλαιο, θα δούμε πώς οι δύο μετασχηματισμοί που χρησιμοποιούμε (νοηματικός και defunctionalization) μπορούν να υποστηρίξουν τέτοια χαρακτηριστικά.

Κεφάλαιο 7

Υποστήριξη για τμηματική μεταγλώττιση

Σε αυτό το κεφάλαιο θα δούμε πώς η υλοποίησή μας μπορεί να υποστηρίξει τμηματική μεταγλώττιση, η οποία αποτελεί απαραίτητο χαρακτηριστικό των σύγχρονων υλοποιήσεων γλωσσών προγραμματισμού.[1]

Τμηματική μεταγλώττιση (*separate compilation*) ονομάζεται η δυνατότητα ενός μεταγλωττιστή να μεταγλωττίζει ξεχωριστές μονάδες κώδικα (modules), παράγοντας ξεχωριστά ενδιάμεσα αρχεία,[2] τα οποία μπορούν στη συνέχεια να συνδεθούν από το αντίστοιχο πρόγραμμα (linker) και να δημιουργηθεί το τελικό εκτελέσιμο πρόγραμμα ή βιβλιοθήκη. Οι περισσότεροι σύγχρονοι μεταγλωττιστές υποστηρίζουν τμηματική μεταγλώττιση κώδικα, ο οποίος έχει οργανωθεί σε μονάδες, για δύο κυρίως λόγους:

(α) Εξοικονομείται χρόνος με το να μεταγλωττίζονται μόνο τα αρχεία που άλλαξαν κάθε φορά που γίνεται κάποια αλλαγή από τον προγραμματιστή, όπως για παράδειγμα κάνει το εργαλείο `make` [260]. Αυτή η βελτίωση είναι είναι ιδιαίτερα αισθητή σε μεγάλα σε μέγεθος προγράμματα [1].

(β) Ένα σύνολο ενδιάμεσων αρχείων μπορούν να συνδεθούν σχηματίζοντας μια *βιβλιοθήκη* (*library*), η οποία μπορεί να διανεμηθεί ως εκτελέσιμος κώδικας που θα συνδεθεί αργότερα με άλλα προγράμματα.

Η τμηματική μεταγλώττιση ως ιδέα δεν είναι νέα: υπάρχει τουλάχιστον από τη δεκαετία του 1950 με τη FORTRAN II [28] και αποτελεί βασικό στοιχείο των περισσότερων ρεαλιστικών υλοποιήσεων γλωσσών προγραμματισμού. Επίσης, η δυνατότητα οργάνωσης κώδικα σε μονάδες, εκτός από τη δυνατότητα τμηματικής μεταγλώττισης, προσφέρει εργαλεία αφαίρεσης (abstraction) και διαχωρισμού της λειτουργικότητας του προγράμματος σε καλά ορισμένα πλαίσια [14, 158, 160, 212].

Ο μετασχηματισμός defunctionalization και ο γενικευμένος νοηματικός μετασχηματισμός έχουν μέχρι στιγμής περιγραφεί ως μετασχηματισμοί που πρέπει να έχουν διαθέσιμο ολόκληρο το πρόγραμμα και άρα δεν μπορούν να υποστηρίξουν τμηματική μεταγλώττιση. Αν και ο μετασχηματισμός defunctionalization έχει χρησιμοποιηθεί σε δημοφιλείς μεταγλωττιστές γλωσσών, όπως ο MLton [297] και ο UHC [76], οι τελευταίοι μπορούν να λειτουργούν μόνο σε κατάσταση πλήρους προγράμματος (whole-program) όταν παράγουν τελικό κώδικα μηχανής. Αυτό θεωρείται σημαντικό μειονέκτημα του defunctionalization [180, 227, 253, 276], που τον κάνει ακατάλληλο για πολλές υλοποιήσεις στην πράξη.

[1] Το κεφάλαιο περιλαμβάνει υλικό που παρουσιάστηκε το 2013 σε άρθρο των Γ. Φουρτούνη και Ν. Παπασπύρου [99], καθώς και υλικό που παρουσιάστηκε το 2014 σε άρθρο των Γ. Φουρτούνη, Ν. Παπασπύρου και Π. Θεοφιλόπουλου [98].

[2] Στην περίπτωση ενός μεταγλωττιστή που παράγει τελικό κώδικα σε δυαδική μορφή, αυτά λέγονται *αρχεία αντικειμενικού κώδικα* (object code). Σε αυτό το κεφάλαιο θα παρουσιάσουμε τμηματική μεταγλώττιση και για γλώσσες υψηλότερου επιπέδου, όπου τα αρχεία μας να είναι πιο αφηρημένα, όπως τα *linksets* του Cardelli [46].

Στη συνέχεια του κεφαλαίου θα δούμε ότι οι δύο κύριοι μετασχηματισμοί που χρησιμοποιούμε μπορούν να τροποποιηθούν ώστε να υποστηρίξουν τμηματική μεταγλώττιση και σύνδεση μονάδων κώδικα. Θα παρουσιαστούν ο *τμηματικός μετασχηματισμός defunctionalization* (Ενότητα 7.1) και ο *τμηματικός νοηματικός μετασχηματισμός* (Ενότητα 7.2). Θα αναφέρουμε επίσης πώς η τεχνική μας χρησιμοποιεί την τεχνολογία τμηματικής μεταγλώττισης και σύνδεσης της C (Ενότητα 7.3).

7.1 Ο τμηματικός μετασχηματισμός defunctionalization

Σε αυτήν την ενότητα θα παρουσιάσουμε τον τμηματικό μετασχηματισμό defunctionalization (modular defunctionalization). Αρχικά θα προσθέσουμε στη γλώσσα FL υποστήριξη για μονάδες κώδικα Haskell (Ενότητα 7.1.1). Στη συνέχεια θα εξετάσουμε γιατί ο μετασχηματισμός defunctionalization, όπως παρουσιάζεται συνήθως, δεν μπορεί να υποστηρίξει τμηματική μεταγλώττιση (Ενότητα 7.1.2). Στη συνέχεια θα παρουσιάσουμε τυπικά τον τμηματικό μετασχηματισμό defunctionalization (Ενότητα 7.1.3), καθώς και μια απλή τεχνική που μειώνει τα δεδομένα που χρειάζεται ο μετασχηματισμός (Ενότητα 7.1.4). Κλείνουμε την ενότητα με αναφορές σε σχετική έρευνα (Ενότητα 7.1.5).

7.1.1 Η αρχική και η τελική γλώσσα

Σε αυτήν την ενότητα θα περιγράψουμε την HL_M, μια συναρτησιακή γλώσσα υψηλότερης τάξης με μονάδες κώδικα, η οποία θα αποτελέσει την αρχική γλώσσα του τμηματικού μετασχηματισμού defunctionalization. Στη συνέχεια θα περιγράψουμε την FL, η οποία είναι το υποσύνολο πρώτης τάξης της, που αποτελεί την τελική γλώσσα του μετασχηματισμού.

Η αρχική γλώσσα HL_M

Η γλώσσα HL_M είναι μια συναρτησιακή γλώσσα υψηλότερης τάξης που μοιάζει με τη Haskell και βασίζεται στο Σύστημα F, υποστηρίζοντας γενικευμένους αλγεβρικούς τύπους δεδομένων και μονάδες κώδικα [75]. Οι γενικευμένοι αλγεβρικοί τύποι δεδομένων είναι απαραίτητοι για την τεχνική μας,[3] η οποία αποτελεί παραλλαγή του πολυμορφικού μετασχηματισμού defunctionalization των Pottier και Gauthier [227].

Ένα πρόγραμμα σε HL_M χωρίζεται σε μονάδες κώδικα, η καθεμία από τις οποίες έχει ένα όνομα, μια λίστα από δηλώσεις εισαγωγής ονομάτων τύπων δεδομένων και συναρτήσεων από άλλες μονάδες, και μια λίστα από ορισμούς τύπων δεδομένων[4] και συναρτήσεων. Η γλώσσα HL_M ορίζεται από την αφηρημένη σύνταξη που ακολουθεί, όπου m είναι όνομα μονάδας, d είναι όνομα τύπου δεδομένων, b είναι βασικός τύπος δεδομένων, x είναι τυπική παράμετρος συνάρτησης ή μεταβλητή προτύπου, *op* είναι ενσωματωμένος σταθερός τελεστής, f είναι όνομα καθολικής συνάρτησης (top-level function), r είναι η πληθικότητα κάποιας συνάρτησης (φυσικός αριθμός), t είναι μεταβλητή τύπου, και κ είναι κατασκευαστής. Ο συμβολισμός με το αστέρι σημαίνει μηδέν ή περισσότερες επαναλήψεις.

[3] Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε τις δηλώσεις `newtype` της Haskell 98, όπως παρατηρούν οι Sulzmann και Wang [265]. Επιλέξαμε τη μέθοδο με τους γενικευμένους αλγεβρικούς τύπους δεδομένων, γιατί αποτελούν γενικά χρήσιμο και δημοφιλές χαρακτηριστικό που χρησιμοποιείται σε αρκετά συναρτησιακά προγράμματα και θα έπρεπε στην πράξη να το χειρίζεται η τεχνική μας.

[4] Όπως είδαμε στην Ενότητα 3.1, θα χρησιμοποιήσουμε τη σύνταξη `data ... where` για τη δήλωση όλων των αλγεβρικών τύπων δεδομένων.

p	$::= M^*$	*πρόγραμμα*
M	$::=$ `module` m `where` $I^*\ D^*\ F^*$	*μονάδα κώδικα*
I	$::=$ `import` $m\ (m.d)^*\ (v/r : \tau)^*$	*δήλωση εισαγωγής*
D	$::=$ `data` $m.d\ t^*$ `where` $(m.\kappa : \tau)^*$	*τύπος δεδομένων*
τ	$::= b \mid t \mid m.d\ \tau^* \mid \tau \to \tau$	*τύπος*
F	$::= m.f\ x^*$ `=` e	*ορισμός*
e	$::= (x \mid v \mid op)\ e^* \mid$ `case` e `of` b^*	*έκφραση*
v	$::= m.f \mid m.\kappa$	*καθολική μεταβλητή*
b	$::= m.\kappa\ x^*\ \to\ e$	*ταίριασμα προτύπου*

Στην HL_M θεωρούμε ότι τα ονόματα τύπων (d), οι καθολικές μεταβλητές (f) και τα ονόματα κατασκευαστών (κ) αρχίζουν πάντα με το όνομα της μονάδας κώδικα (m) στην οποία ορίζονται. Οι τυπικές παράμετροι συναρτήσεων και οι μεταβλητές προτύπων (x) είναι τοπικά ονόματα και δεν αρχίζουν με όνομα μονάδας. Με αυτόν τον τρόπο, κάθε μονάδα κώδικα έχει το δικό της *χώρο ονομάτων* (*namespace*): κάθε καθολική συνάρτηση είναι ξεχωριστή και δύο διαφορετικές μονάδες μπορούν να ορίζουν συναρτήσεις, τύπους δεδομένων ή κατασκευαστές με το ίδιο όνομα, χωρίς προβλήματα συνωνυμίας. Στη συνέχεια, θα ακολουθήσουμε το στυλ της Haskell για τα ονόματα: όλες οι συναρτήσεις και οι μεταβλητές θα αρχίζουν με μικρό γράμμα, ενώ οι τύποι δεδομένων, οι κατασκευαστές και οι μονάδες κώδικα θα αρχίζουν με κεφαλαίο γράμμα.

Για παράδειγμα, ένα πρόγραμμα που αποτελείται από δύο μονάδες `Lib` και `Main` είναι αυτό που ακολουθεί. Θεωρούμε (για λόγους απλότητας) ότι οι τύποι των συναρτήσεων `Lib.high` και `Main.high` είναι μονομορφικοί.

```
module Lib where

Lib.high g x = g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.h 1
Lib.add a b  = a + b
```

```
module Main where

import Lib ( Lib.high/2 :: (Int→Int)→Int→Int
           , Lib.test/0 :: Int
           , Lib.add/2  :: Int→Int→Int )

Main.result = Main.f 10 + Lib.test
Main.f a    = a + Main.high (Lib.add 1) + Lib.high Main.dec 2
Main.high g = g 10
Main.dec x  = x − 1
```

Η τελική γλώσσα FL

Η γλώσσα FL, που είναι η τελική γλώσσα του μετασχηματισμού defunctionalization, είναι το υποσύνολο πρώτης τάξης της HL_M, χωρίς μονάδες. Δηλαδή, στα προγράμματα σε FL ισχύουν οι εξής ιδιότητες:

1. Όλες οι συναρτήσεις και οι κατασκευαστές τύπων δεδομένων είναι πρώτης τάξης.

2. Τα ονόματα μονάδων που τοποθετούνται μπροστά από τα ονόματα συναρτήσεων, τύπων δεδομένων και κατασκευαστών, θεωρούνται πια μέρη του ονόματος.

3. Δεν υπάρχουν όρια μονάδων· τα προγράμματα είναι λίστες από ορισμούς τύπων δεδομένων και συναρτήσεων.

7.1.2 Defunctionalization και τμηματική μεταγλώττιση

Έστω πάλι οι δύο μονάδες κώδικα `Lib` και `Main` που ορίστηκαν στην Ενότητα 7.1.1. Αν εφαρμόσουμε τον μετασχηματισμό defunctionalization ξεχωριστά σε καθεμία από αυτές, προκύπτουν τα εξής δύο κομμάτια κώδικα:

```
-- module Lib where

Lib.high g x = Lib.apply g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.H 1
Lib.add a b  = a + b

data Lib.Closure par res where
  Lib.H :: Lib.Closure Int Int

Lib.apply c z = case c of
                  Lib.H → Lib.h z
```

```
-- module Main where

Main.result = Main.f 10 + Lib.test
Main.f a    = a + Main.high (Lib.Add 1) + Lib.high Main.Dec 2
Main.high g = Main.apply g 10
Main.dec x  = x − 1

data Main.Closure par res where
  Lib.Add1 :: Int → Main.Closure Int Int
  Main.Dec :: Main.Closure Int Int

Main.apply c z = case c of
                   Lib.Add a → Lib.add a z
                   Main.Dec  → Main.dec z
```

Αρχικά, βλέπουμε ότι οι δύο αυτές μονάδες δημιουργούν δύο διαφορετικούς ορισμούς για τον τύπο δεδομένων `Closure`, με κάθε ορισμό να έχει διαφορετικούς γενικά κατασκευαστές. Αν θεωρηθεί ότι αυτά τα δύο κομμάτια κώδικα ενωθούν με επιτυχία, και οι δύο τύποι `Closure` συνενωθούν, θα υπάρχει πρόβλημα όταν πρόκειται να αποτιμηθεί η έκφραση `Lib.high Main .Dec 2`: η `Lib.high` θα καλέσει τη `Lib.apply`, η οποία δε γνωρίζει για τον κατασκευαστή κλεισίματος `Main.Dec` και το πρόγραμμα θα παρουσιάσει σφάλμα.

Φαίνεται ότι πρέπει να χειριστούμε με ιδιαίτερο τρόπο τους τύπους δεδομένων των κλεισιμάτων, τους κατασκευαστές κλεισιμάτων και τις συναρτήσεις `apply`, προκειμένου συναρτήσεις διαφορετικών μονάδων να μπορούν να ανταλλάσσουν εκφράσεις υψηλότερης τάξης. Όλοι οι υπόλοιποι τύποι δεδομένων, κατασκευαστές και συναρτήσεις, μπορούν να μεταγλωττιστούν ξεχωριστά και να συνυπάρχουν κατά τα γνωστά, αφού δεν υπάρχει κίνδυνος συνωνυμίας.

7.1.3 Τυπικός ορισμός του τμηματικού defunctionalization

Το πρόβλημα της προηγούμενης ενότητας λύνεται αν ελέγχεται ο κώδικας που παράγει ο μετασχηματισμός defunctionalization: οι τύποι δεδομένων των κλεισιμάτων, οι κατασκευαστές κλεισιμάτων και οι συναρτήσεις `apply` πρέπει να συγκεντρώνονται από όλες τις μονάδες και να παράγεται ο τελικός κώδικας για αυτά μόνο κατά τον χρόνο σύνδεσης (link-time). Η τεχνική μας εφαρμόζει τον μετασχηματισμό defunctionalization ξεχωριστά σε κάθε μονάδα κώδικα, μετασχηματίζοντας τον αρχικό κώδικα HL_M σε τελικό κώδικα FL, εισάγοντας κατασκευαστές κλεισιμάτων και κλήσεις προς συναρτήσεις `apply`, όπου αυτό απαιτείται. Οι κατασκευαστές που χρειάζονται για κάθε μονάδα καταγράφονται και κρατούνται μαζί με τον τελικό κώδικα που δημιουργείται για κάθε μονάδα κώδικα. Στη συνέχεια, σε ένα τελικό βήμα σύνδεσης (linking), συνενώνονται οι δηλώσεις του τύπου δεδομένων κλεισιμάτων και δημιουργείται ο κώδικας της τελικής συνάρτησης `apply`, με βάση την καταγεγραμμένη πληροφορία.

Ο τμηματικός μετασχηματισμός defunctionalization είναι επομένως ένας μετασχηματισμός δύο βημάτων:

1. *Ξεχωριστός μετασχηματισμός defunctionalization*: Κάθε μονάδα μετασχηματίζεται ξεχωριστά. Αυτό παράγει (i) ένα σύνολο από μετασχηματισμένες δηλώσεις τύπων δεδομένων, (ii) ένα σύνολο από ορισμούς καθολικών συναρτήσεων, και (iii) πληροφορία για τα κλεισίματα που αντιστοιχούν στις καθολικές συναρτήσεις που ορίζονται στη μονάδα κώδικα. Η τρίτη πληροφορία είναι η *διεπαφή defunctionalization* (*defunctionalization interface*) της μονάδας κώδικα. Σε αυτό το σημείο, οι ορισμοί κάθε μονάδας μπορούν να μεταγλωττιστούν σε τελικό κώδικα, θεωρώντας ότι οι κατασκευαστές κλεισιμάτων και η συνάρτηση `apply` είναι σύμβολα που θα επιλυθούν αργότερα, στον χρόνο σύνδεσης. Εναλλακτικά, οι ορισμοί μπορούν να δοθούν στον τμηματικό νοηματικό μετασχηματισμό της επόμενης ενότητας και να γίνουν μονάδες νοηματικού κώδικα, όπως θα δούμε στην Ενότητα 7.2.

2. *Σύνδεση* (*linking*): Ο κώδικας που μετασχηματίστηκε ξεχωριστά συνενώνεται και δημιουργείται ο κώδικας που έλειπε (κατασκευαστές κλεισιμάτων, συνάρτηση `apply`), χρησιμοποιώντας τις διεπαφές defunctionalization του προηγούμενου βήματος. Ο νέος κώδικας μπορεί να μεταγλωττιστεί ξεχωριστά (ή να μετασχηματιστεί από τον τμηματικό μετασχηματισμό της Ενότητας 7.2) και να παραχθεί το τελικό εκτελέσιμο (ή νοηματικό πρόγραμμα).

Στη συνέχεια αυτής της ενότητας θα δώσουμε έναν τυπικό ορισμό του τμηματικού μετασχηματισμού defunctionalization. Τα δύο βήματα που αναφέρθηκαν παραπάνω θα περιγραφούν στις επόμενες υποενότητες.

Ξεχωριστός μετασχηματισμός defunctionalization

Αυτό το βήμα εφαρμόζει τον μετασχηματισμό defunctionalization σε κάθε μονάδα, δημιουργώντας μια λίστα από μετασχηματισμένους ορισμούς τύπων δεδομένων και συναρτήσεων, και μια λίστα όλων των κατασκευαστών κλεισιμάτων που αντιστοιχούν σε καθολικές συναρτήσεις που ορίστηκαν στη μονάδα. Στη συνέχεια, θεωρούμε ότι μετασχηματίζουμε μια μονάδα κώδικα M.

Υποθέτουμε ότι έχει ήδη γίνει ο έλεγχος τύπων και είναι διαθέσιμη η απαραίτητη πληροφορία τύπων. Για λόγους απλότητας κατά την παρουσίαση του μετασχηματισμού, υποθέτουμε ότι όλες οι εκφράσεις συνοδεύονται από τον τύπο τους (π.χ., e^τ) αλλά τις περισσότερες φορές θα παραλείψουμε αυτήν την πληροφορία.

Θεωρούμε επίσης ότι υπάρχει κάποιος μηχανισμός για τη δημιουργία μοναδικών *ονομάτων* κατά τη διάρκεια του μετασχηματισμού. Όλα αυτά τα ονόματα δεν περιέχουν ονόματα μονάδων κώδικα και μπορούν να χρησιμοποιηθούν στην FL. Ειδικότερα:

- Οι συναρτήσεις $\mathcal{N}(m.d)$, $\mathcal{N}(m.f)$ και $\mathcal{N}(m.\kappa)$ δημιουργούν ονόματα για δηλωμένους τύπους, καθολικές συναρτήσεις και κατασκευαστές που εμφανίζονται στον πηγαίο κώδικα μιας μονάδας.

- Η $\mathcal{C}\ell(\tau_0, \tau_1)$ δημιουργεί τον γενικευμένο τύπο δεδομένων που αντιστοιχεί σε κλεισίματα που παίρνουν παραμέτρους τύπου τ_0 και επιστρέφουν αποτελέσματα τύπου τ_1 (χρησιμοποιήσαμε τον συμβολισμό `Closure` $\tau_o\ \tau_1$ στα παραδείγματα);

- Η $\mathcal{C}(v, n)$ δημιουργεί το όνομα ενός κατασκευαστή που αντιστοιχεί στο κλείσιμο της v, που δεσμεύει n παραμέτρους.

- Η $\mathcal{A}$ δημιουργεί το όνομα της συνάρτησης που αναλαμβάνει τη διαχείριση των κλεισιμάτων (την οποία ονομάζουμε `apply` στα παραδείγματα).

Θα χρειαστούν επίσης κάποιες βοηθητικές συναρτήσεις για τον χειρισμό των τύπων:

- Η $\mathsf{arity}_T(\tau)$ επιστρέφει την πληθικότητα (arity) ενός τύπου (πόσες επιπλέον παράμετροι πρέπει να δοθούν μέχρι να επιστραφεί τιμή που να μην είναι υψηλότερης τάξης).

$$
\begin{array}{ll}
\mathsf{arity}_T(b) & \doteq\ 0 \\
\mathsf{arity}_T(t) & \doteq\ 0 \\
\mathsf{arity}_T(m.d\ \tau^*) & \doteq\ 0 \\
\mathsf{arity}_T(\tau_1 \to \tau_2) & \doteq\ 1 + \mathsf{arity}_T(\tau_2)
\end{array}
$$

- Η $\mathsf{arity}_V(v)$ επιστρέφει την πληθικότητα μιας καθολικής συνάρτησης ή ενός κατασκευαστή (πόσες τυπικές παράμετροι υπάρχουν στον αντίστοιχο ορισμό). Αυτό είναι γνωστό από τη σύνταξη, για συναρτήσεις που δηλώνονται στην ίδια μονάδα κώδικα, ή από την πληροφορία που βρίσκεται στη δήλωση `import`. Για μια δηλωμένη συνάρτηση f τύπου τ, ισχύει πάντα ότι $\mathsf{arity}_V(f) \leq \mathsf{arity}_T(\tau)$ – οι δυο πληθικότητες είναι ίσες μόνο όταν το σώμα του ορισμού της συνάρτησης δεν είναι υψηλότερης τάξης.

- Η $\mathsf{ground}(\tau)$ μετατρέπει τύπους υψηλότερης τάξης σε τύπους πρώτης τάξης, αντικαθιστώντας τύπους υψηλότερης τάξης με τους αντίστοιχους μετασχηματισμένους τύπους κλεισιμάτων.

$$
\begin{array}{ll}
\mathsf{ground}(b) & \doteq\ b \\
\mathsf{ground}(t) & \doteq\ t \\
\mathsf{ground}(m.d\ \tau^*) & \doteq\ \mathcal{N}(m.d)\ \mathsf{ground}(\tau)^* \\
\mathsf{ground}(\tau_1 \to \tau_2) & \doteq\ \mathcal{C}\ell(\mathsf{ground}(\tau_1), \mathsf{ground}(\tau_2))
\end{array}
$$

- Η $\mathsf{lower}(\tau)$ μετατρέπει τύπους υψηλότερης τάξης σε τύπους πρώτης τάξης, αντικαθιστώντας, αν χρειάζεται, τις παραμέτρους των τύπων συναρτήσεων με τους αντίστοιχους τύπους κλεισιμάτων.

$$
\begin{array}{ll}
\mathsf{lower}(b) & \doteq\ b \\
\mathsf{lower}(t) & \doteq\ t \\
\mathsf{lower}(m.d\ \tau^*) & \doteq\ \mathcal{N}(m.d)\ \mathsf{ground}(\tau)^* \\
\mathsf{lower}(\tau_1 \to \tau_2) & \doteq\ \mathsf{ground}(\tau_1) \to \mathsf{lower}(\tau_2)
\end{array}
$$

Η διαδικασία του μετασχηματισμού defunctionalization τυποποιείται με τη βοήθεια τεσσάρων μετασχηματισμών $\mathcal{T}(D)$, $\mathcal{D}(F)$, $\mathcal{E}(e)$ και $\mathcal{B}(b)$, για τις δηλώσεις τύπων, τους ορισμούς καθολικών συναρτήσεων, εκφράσεων, και κλάδων ταιριάσματος προτύπων, αντίστοιχα:

$$\begin{aligned}
&\mathcal{T}(\texttt{data}\ m.d\ t^*\ \texttt{where}\ (m.\kappa : \tau)^*) \doteq \\
&\quad \texttt{data}\ \mathcal{N}(m.d)\ t^*\ \texttt{where}\ (\mathcal{N}(m.\kappa) : \mathsf{lower}(\tau))^*
\end{aligned}$$

$$\mathcal{D}(m.f\ x^*\ =\ e)\ \doteq\ \mathcal{N}(m.f)\ x^*\ =\ \mathcal{E}(e)$$

$$\begin{aligned}
\mathcal{E}(x\ e_1\ \ldots\ e_n) &\doteq \mathcal{A} \ldots (\mathcal{A}\ x\ \mathcal{E}(e_1)) \ldots \mathcal{E}(e_n) \\
\mathcal{E}(v\ e_1\ \ldots\ e_n) &\doteq \mathcal{C}(v, n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
&\qquad\qquad \text{εάν } n < n' = \mathsf{arity}_V(v) \\
\mathcal{E}(v\ e_1\ \ldots\ e_n) &\doteq \mathcal{A} \ldots (\mathcal{A}\ (\mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_{n'}))\ \mathcal{E}(e_{n'+1})) \ldots \mathcal{E}(e_n) \\
&\qquad\qquad \text{εάν } n \geq n' = \mathsf{arity}_V(v) \\
\mathcal{E}(\mathit{op}\ e_1\ \ldots\ e_n) &\doteq \mathit{op}\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) \\
\mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b^*) &\doteq \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b)^*
\end{aligned}$$

$$\mathcal{B}(m.\kappa\ x^*\ \rightarrow\ e)\ \doteq\ \mathcal{N}(m.\kappa)\ x^*\ \rightarrow\ \mathcal{E}(e)$$

Οι παραπάνω μετασχηματισμοί κάνουν τα εξής: (i) Οι τύποι δεδομένων μετασχηματίζονται ώστε όλοι οι τύποι υψηλότερης τάξης στις υπογραφές (signatures) κατασκευαστών αντικαθιστώνται από τον αντίστοιχο τύπο δεδομένων κλεισιμάτων. (ii) Οι τυπικές παράμετροι ή μεταβλητές προτύπων που καλούνται ως συναρτήσεις εφαρμόζονται χρησιμοποιώντας την αντίστοιχη συνάρτηση κλεισιμάτων. (iii) Οι μερικές εφαρμογές καθολικών συναρτήσεων και κατασκευαστών αντικαθιστώνται από κατασκευαστές κλεισιμάτων. (iv) Οι κλήσεις συναρτήσεων με περισσότερες παραμέτρους από όσες έχει ο ορισμός τους (over-saturated calls) αντικαθιστώνται από εφαρμογές κλεισιμάτων που επιστρέφονται ως αποτελέσματα συναρτήσεων.

Κατά το πρώτο βήμα του μετασχηματισμού, συλλέγεται χρήσιμη πληροφορία για κάθε κλείσιμο που αντιστοιχεί σε μια καθολική συνάρτηση ή κατασκευαστή. Αυτό επιτυγχάνεται με τη συνάρτηση $\mathcal{F}(v^\tau)$, ο ορισμός της οποίας ακολουθεί. Υποθέτουμε ότι η v είναι καθολική συνάρτηση ή κατασκευαστής, με τύπο τ.

$$\begin{aligned}
&\mathcal{F}(v^\tau)\ \doteq\ \mathsf{info}(v, \tau, [\,]) \\
&\mathsf{info}(v, \tau, \tau^*)\ \doteq\ \{(v/r, \tau^*, \mathsf{ground}(\tau))\} \cup \mathsf{info}(v, \tau_2, \tau^* +\!\!+ [\mathsf{ground}(\tau_1)]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{εάν } \tau = \tau_1 \rightarrow \tau_2, \text{ και } \mathsf{length}(\tau^*) < \mathsf{arity}_V(v) = r \\
&\mathsf{info}(v, \tau, \tau^*)\ \doteq\ \emptyset \qquad\qquad\qquad\qquad \text{σε κάθε άλλη περίπτωση}
\end{aligned}$$

Η συνάρτηση $\mathcal{F}(v^\tau)$ επιστρέφει ένα σύνολο από τριάδες, μια για κάθε πιθανό κλείσιμο της v. Κάθε τριάδα περιέχει: (i) το όνομα v και την πληθικότητα r της αντίστοιχης συνάρτησης, (ii) μια λίστα με τους τύπους των παραμέτρων που έχουν ήδη δοθεί, και (iii) τον τύπο του αντίστοιχου κλεισίματος.

Για παράδειγμα, έστω μια συνάρτηση `add` με τρεις ακέραιες παραμέτρους.

```
add a b c = a + b + c
```

Η συνάρτηση έχει πληθικότητα 3 και τύπο `Int->Int->Int->Int`· μπορεί να χρησιμοποιηθεί σε τρία κλεισίματα, όταν δίνονται 0, 1 ή 2 παράμετροι:

$$\begin{aligned}
&\mathcal{F}(\texttt{add}^{\texttt{Int->Int->Int->Int}}) = \\
&\quad \{\ (\texttt{add}/3, [\,], \texttt{Closure Int (Closure Int (Closure Int Int))}), \\
&\quad\ \ (\texttt{add}/3, [\texttt{Int}], \texttt{Closure Int (Closure Int Int)}), \\
&\quad\ \ (\texttt{add}/3, [\texttt{Int}, \texttt{Int}], \texttt{Closure Int Int})\ \}
\end{aligned}$$

Είναι πιθανό να μη χρησιμοποιηθούν στο τελικό πρόγραμμα όλα τα κλεισίματα που παράγονται από τη συνάρτηση $\mathcal{F}(v^\tau)$. Η υλοποίηση μπορεί να επιλέξει ένα υποσύνολο από αυτά τα κλεισίματα, π.χ. όσα εμφανίζονται στον κώδικα μιας μονάδας. Όμως, το τελικό σύνολο όλων των κλεισιμάτων δεν είναι απλά η ένωση όλων όσων εμφανίζονται στον κώδικα κάθε μονάδας: λόγω της δυνατότητας μερικής εφαρμογής, πρέπει να δημιουργηθούν αυτόματα επιπλέον κλεισίματα. Θα επανέλθουμε σε αυτό το σημείο, όταν περιγράψουμε τη βελτιστοποίηση της Ενότητας 7.1.4.

Σύνδεση

Μετά την ξεχωριστή εφαρμογή του defunctionalization σε κάποιες μονάδες κώδικα, μένουν οι μετασχηματισμένοι ορισμοί των μονάδων και η αντίστοιχη πληροφορία κλεισιμάτων για κάθε μονάδα. Το τελικό πρόγραμμα συνδέεται συλλέγοντας όλους τους μετασχηματισμένους ορισμούς και συμπληρώνοντας αυτόν τον κώδικα με τον κώδικα της συνάρτησης που χειρίζεται τα κλεισίματα. Έστω ότι η I είναι η ένωση όλης της πληροφορίας κλεισιμάτων από όλες τις μονάδες που πρόκειται να συνδεθούν. Να σημειωθεί ότι επειδή παρουσιάζουμε τον μετασχηματισμό σε επίπεδο πηγαίου κώδικα, αρχίζουμε δημιουργώντας τον ορισμό του τύπου δεδομένων για τα κλεισίματα, κάτι που δε χρειάζεται αν απλά συνδέουμε δυαδικό κώδικα.

Ο τύπος δεδομένων των κλεισιμάτων, παραμετροποιημένος με τον τύπο της παραμέτρου a και τον τύπο του αποτελέσματος b ενός κλεισίματος, ορίζεται ως εξής:

$$\texttt{data}\ \mathcal{C}\ell(a,b)\ \texttt{where}\ \{\ \mathcal{C}(v,n) : \tau^* \to \tau \mid (v/r,\tau^*,\tau) \in I, n = \mathsf{length}(\tau^*)\ \}$$

όπου με $\tau^* \to \tau$ κατασκευάζουμε τύπους συναρτήσεων με περισσότερες από μια παραμέτρους, π.χ., $[\tau_1,\tau_2,\tau_3] \to \tau \ \doteq\ \tau_1 \to \tau_2 \to \tau_3 \to \tau$.

Η συνάρτηση χειρισμού των κλεισιμάτων $\mathcal{A}$ δημιουργείται χρησιμοποιώντας πάλι την πληροφορία κλεισιμάτων I. Επειδή το πρόγραμμα είναι κλειστό στον χρόνο σύνδεσης, χρειάζεται απλά να δημιουργήσουμε μια μεγάλη έκφραση `case` με έναν κλάδο ταιριάσματος προτύπων για κάθε κατασκευαστή κλεισίματος στο I. Το αποτέλεσμα κάθε τέτοιου κλάδου μπορεί να έχει έναν από τους εξής δύο τύπους: (i) αν δίνοντας άλλη μια παράμετρο, φτάσαμε την πληθικότητα της συνάρτησης, τότε συμβαίνει μια πλήρης εφαρμογή της συνάρτησης, αλλιώς (ii) αν μπορούν να δοθούν και άλλες παράμετροι, τότε επιστρέφεται ένα νέο κλείσιμο (που περιμένει μια παράμετρο λιγότερο). Αυτές οι δύο εναλλακτικές περιπτώσεις μπορούν να παρουσιαστούν μαζί χρησιμοποιώντας τη βοηθητική συνάρτηση $\mathsf{next}(v/r,n)$ που ορίζεται παρακάτω.

$$\begin{array}{lll} \mathsf{next}(v/r,n) & \doteq & \mathcal{N}(v) \qquad\qquad \text{εάν } n = r \\ \mathsf{next}(v/r,n) & \doteq & \mathcal{C}(v,n) \qquad\qquad \text{εάν } n < r \end{array}$$

Ο ορισμός της $\mathcal{A}$ τότε γράφεται ως εξής:

$$\begin{array}{rcl} \mathcal{A}\ c\ x & = & \texttt{case}\ c\ \texttt{of}\ \{\ \mathcal{C}(v,n)\ y_1\ \ldots\ y_n \to \mathsf{next}(v/r,n+1)\ y_1\ \ldots\ y_n\ x \\ & & \qquad \mid (v/r,\tau^*,\tau) \in I, n = \mathsf{length}(\tau^*)\ \} \end{array}$$

Επιστροφή στο παράδειγμα

Επιστρέφουμε στο παράδειγμα της Ενότητας 7.1.1. Εφαρμόζοντας το πρώτο βήμα της τεχνικής μας, δημιουργούνται δύο μετασχηματισμένα κομμάτια κώδικα και δύο σύνολα από πληροφορίες κλεισιμάτων. Και τα δύο δίνονται στην Εικόνα 7.1. Ο κώδικας μοιάζει αρκετά με τον κώδικα που προκύπτει από τον κλασικό μετασχηματισμό defunctionalization

Μετασχηματισμένη μονάδα Lib

```
f_Lib_high g x = apply g x
f_Lib_h y = y + 1
f_Lib_test = f_Lib_high Cf_Lib_h_0 1
f_Lib_add a b = a + b
```

Πληροφορία κλεισιμάτων για τη μονάδα Lib

```
(f_Lib_add/2,  0, Closure Int (Closure Int Int))
(f_Lib_add/2,  1, Int -> Closure Int Int)
(f_Lib_h/1,    0, Closure Int Int)
(f_Lib_high/2, 0, Closure (Closure Int Int) (Closure Int Int))
(f_Lib_high/2, 1, Closure Int Int -> Closure Int Int)
```

Μετασχηματισμένη μονάδα Main

```
f_Main_result = f_Main_f 10 + f_Lib_test
f_Main_f a = a + f_Main_high (Cf_Lib_add_1 1)
               + f_Lib_high Cf_Main_dec_0 2
f_Main_high g = apply g 10
f_Main_dec x = x - 1
```

Πληροφορία κλεισιμάτων για τη μονάδα Main

```
(f_Lib_add/2,   0, Closure Int (Closure Int Int))
(f_Lib_add/2,   1, Int -> Closure Int Int)
(f_Lib_high/2,  0, Closure (Closure Int Int) (Closure Int Int))
(f_Lib_high/2,  1, Closure Int Int -> Closure Int Int)
(f_Main_dec/1,  0, Closure Int Int)
(f_Main_f/1,    0, Closure Int Int)
(f_Main_high/1, 0, Closure (Closure Int Int) Int)
```

Εικόνα 7.1: Το αποτέλεσμα της εφαρμογής του τμηματικού μετασχηματισμού defunctionalization στο παράδειγμα της Ενότητας 7.1.1.

(Ενότητα 7.1.2)· διαφέρουν τα ονόματα και ότι χρησιμοποιείται μόνο ένας τύπος δεδομένων `Closure` και μια συνάρτηση `apply`. Επίσης, οι ορισμοί των δύο τελευταίων δεν περιλαμβάνονται στον κώδικα που προκύπτει. Μετά τη σύνδεση αυτών των δύο μονάδων κώδικα, οι ορισμοί του τύπου δεδομένων `Closure` και της συνάρτησης `apply` δίνονται στην Εικόνα 7.2.

7.1.4 Μείωση του αριθμού των κατασκευαστών κλεισιμάτων

Σε μεγάλα προγράμματα, η τεχνική που περιγράψαμε στην Ενότητα 7.1.3 μπορεί να δημιουργήσει μεγάλο αριθμό κατασκευαστών κλεισιμάτων, καθώς και μεγάλο σώμα στη συνάρτηση `apply`. Σε αυτήν την ενότητα θα περιγράψουμε μια βελτιστοποίηση που μπορεί να απαλείψει αρκετούς κατασκευαστές κλεισιμάτων (και αντίστοιχους κλάδους στη συνάρτηση `apply`), με βάση μια απλή ανάλυση χρήσης (usage analysis) για κατασκευαστές κλεισιμάτων.

Η ανάλυσή μας βασίζεται στη διαπίστωση ότι η μερική εφαρμογή πάντα προσθέτει παραμέτρους στα δεξιά της λίστας τυπικών παραμέτρων μιας συνάρτησης. Επομένως, τα μόνα κλεισίματα που μπορούν να εμφανιστούν κατά τη διάρκεια της εκτέλεσης ενός προγράμματος είναι αυτά που δημιουργούνται αρχικά στο κείμενο του προγράμματος, ή νέα κλεισίματα

Τύπος δεδομένων κλεισιμάτων

```
data Closure p r where
  Cf_Lib_add_0   :: Closure Int (Closure Int Int)
  Cf_Lib_add_1   :: Int → Closure Int Int
  Cf_Lib_h_0     :: Closure Int Int
  Cf_Lib_high_0  :: Closure (Closure Int Int) (Closure Int Int)
  Cf_Lib_high_1  :: Closure Int Int → Closure Int Int
  Cf_Main_dec_0  :: Closure Int Int
  Cf_Main_f_0    :: Closure Int Int
  Cf_Main_high_0 :: Closure (Closure Int Int) Int
```

Συνάρτηση χειρισμού κλεισιμάτων

```
apply c x = case c of
              Cf_Lib_add_0     → Cf_Lib_add_1 x
              Cf_Lib_add_1 y1  → f_Lib_add y1 x
              Cf_Lib_h_0       → f_Lib_h x
              Cf_Lib_high_0    → Cf_Lib_high_1 x
              Cf_Lib_high_1 y1 → f_Lib_high y1 x
              Cf_Main_dec_0    → f_Main_dec x
              Cf_Main_f_0      → f_Main_f x
              Cf_Main_high_0   → f_Main_high x
```

Εικόνα 7.2: Ο κώδικας που παράγεται κατά τη σύνδεση του παραδείγματος της Ενότητας 7.1.1.

που προκύπτουν από την πρόσθεση επιπλέον παραμέτρων σε υπάρχοντα κλεισίματα.

Ορίζουμε το σύνολο όλων των κατασκευαστών κλεισιμάτων που μπορούν να δημιουργηθούν κατά την εκτέλεση του προγράμματος, ως $C_u = C_0 \cup C_a$, όπου:

1. C_0 είναι το σύνολο όλων των κατασκευαστών που εμφανίζονται στα μετασχηματισμένα σώματα των καθολικών συναρτήσεων.

2. C_a είναι το σύνολο όλων των κατασκευαστών που μπορούν να δημιουργηθούν προσθέτοντας μία ή περισσότερες παραμέτρους σε όλα τα κλεισίματα που αναπαριστώνται από το $c \in C_0$.

Στην πράξη, η αποδοτικότητα αυτής της ανάλυσης εξαρτάται από τον αριθμό των συναρτήσεων που δημιουργούν κλεισίματα στο αρχικό πρόγραμμα και από τις πληθικότητες αυτών των συναρτήσεων. Μιας και τα περισσότερα προγράμματα σε Haskell δε χρησιμοποιούν όλες τους τις συναρτήσεις για να δημιουργούν κλεισίματα, αυτή η ανάλυση, εκτός από απλή και οικονομική σε κόστος, φαίνεται και χρήσιμη στην πράξη.

Για να χρησιμοποιηθεί αυτή η ανάλυση για κάθε μονάδα κώδικα, πρέπει να αλλάξουμε λίγο την τεχνική της Ενότητας 7.1.3. Η διεπαφή defunctionalization μιας μονάδας δεν πρέπει να κρατά πληροφορία για όλα τα κλεισίματα που μπορούν να δημιουργηθούν από τις καθολικές συναρτήσεις και κατασκευαστές που ορίζονται στη μονάδα. Αντίθετα, πρέπει να κρατά πληροφορία μόνο για τα κλεισίματα που η μονάδα πραγματικά χρησιμοποιεί, τα οποία δημιουργούνται είτε από τοπικές συναρτήσεις και κατασκευαστές, είτε από εισαγόμενες συναρτήσεις και κατασκευαστές από άλλες μονάδες. Και σε αυτήν την περίπτωση, το σύνολο όλων των πιθανών κλεισιμάτων που χρησιμοποιούνται στο πρόγραμμα θα είναι η ένωση

Τύπος δεδομένων κλεισιμάτων

```
data Closure p r where
  Cf_Lib_add_1 :: Int → Closure Int Int
  Cf_Lib_h_0 :: Closure Int Int
  Cf_Main_dec_0 :: Closure Int Int
```

Συνάρτηση χειρισμού κλεισιμάτων

```
apply c x = case c of
              Cf_Lib_add_1 y1 → f_Lib_add y1 x
              Cf_Lib_h_0 → f_Lib_h x
              Cf_Main_dec_0 → f_Main_dec x
```

Εικόνα 7.3: Ο κώδικας που δημιουργείται κατά τη σύνδεση του παραδείγματος της Ενότητας 7.1.1, με την ανάλυση χρήσης των κατασκευαστών κλεισιμάτων.

όλης αυτής της πληροφορίας από όλες τις μονάδες κώδικα, που συλλέγεται κατά τη σύνδεση του προγράμματος.

Η εφαρμογή αυτής της ανάλυσης στο παράδειγμα της Ενότητας 7.1.1, οδηγεί σε έναν τύπο δεδομένων `Closure` που περιλαμβάνει μόνο τρεις κατασκευαστές, σε σύγκριση με τους οκτώ κατασκευαστές του αντίστοιχου τύπου δεδομένων της Εικόνας 7.2. Ο τύπος δεδομένων και η συνάρτηση `apply` που προκύπτουν δίνονται στην Εικόνα 7.3.

7.1.5 Σχετική έρευνα

Οι Pottier και Gauthier σημειώνουν ότι ο μετασχηματισμός defunctionalization μπορεί να είναι τμηματικός, αν η γλώσσα στην οποία γίνεται είναι πιο πλούσια από την HL_M και υποστηρίζει αναδρομικές πολυ-μεθόδους (multi-methods) [227]. Η τεχνική μας είναι απλούστερη, γιατί συλλέγει μόνο πληροφορία κατασκευαστών κλεισιμάτων από κάθε μονάδα κώδικα. Στην πράξη, τα δύο βήματα ξεχωριστού μετασχηματισμού defunctionalization και σύνδεσης μπορούν να θεωρηθούν ως μια τεχνική υλοποίησης των πολυ-μεθόδων που χρειάζεται ο τμηματικός μετασχηματισμός defunctionalization.

Το εμπρόσθιο τμήμα (front-end) του μετασχηματιστή GRIN είχε κάποιο βαθμό υποστήριξης για τμηματική μεταγλώττιση, αλλά το οπίσθιο μέρος (back-end) ήταν ένας μεταγλωττιστής ολόκληρου προγράμματος [38]. Ο Utrecht Haskell Compiler (UHC), ο οποίος επίσης βασίζεται στην προσέγγιση του GRIN, επιτρέπει την τμηματική μεταγλώττιση όταν χρησιμοποιείται μια ειδική μορφή κώδικα byte (bytecode), η οποία εκτελείται σε έναν διερμηνέα, αλλά δεν την υποστηρίζει για δυαδικό εκτελέσιμο κώδικα [76]. Στα πλαίσια του μετασχηματισμού εξειδίκευσης (specialization transformation) του UHC, ο Middelkoop παρατήρησε ότι η πλήρης υποστήριξη τμηματικής μεταγλώττισης μαζί με defunctionalization, απαιτεί τη διατήρηση πληροφορίας που μοιάζει με το αφηρημένο συντακτικό δέντρο (abstract syntax tree) μιας συνάρτησης [174]. Η τεχνική μας λειτουργεί με τον ίδιο τρόπο, αλλά κρατώντας μόνο την πληροφορία κατασκευαστών κλεισιμάτων, η οποία αρκεί για να δημιουργηθεί το τελικό αφηρημένο συντακτικό δέντρο του κώδικα που λείπει.

Ο Mitchell [176] πρότεινε μια παραλλαγή του defunctionalization, η οποία δεν εισάγει νέους κατασκευαστές κλεισιμάτων ή κλήσεις προς κάποια συνάρτηση χειρισμού κλεισιμάτων. Αυτό έχει ως αποτέλεσμα να μην έχει προβλήματα στον ξεχωριστό μετασχηματισμό μονάδων κώδικα και να επιτρέπει την τμηματική μεταγλώττιση. Δεν μπορεί όμως να μετασχηματίσει όλα τα προγράμματα υψηλότερης τάξης, ενώ ο τμηματικός μετασχηματισμός

defunctionalization που παρουσιάσαμε είναι το ίδιο ισχυρός με τον κλασικό μετασχηματισμό defunctionalization.

Ο Tolmach περιέγραψε συνοπτικά πώς μια επέκταση της τεχνικής typed closure conversion [275] θα μπορούσε να υποστηρίξει τμηματική μεταγλώττιση. Η σύνοψή του μοιράζεται ιδέες με την προσέγγισή μας, καθυστερώντας τη δημιουργία των κατασκευαστών και των συναρτήσεων χειρισμού τους για τον χρόνο σύνδεσης. Η τεχνική του αφορά όμως μονομορφικές γλώσσες και χρησιμοποιεί επεκτάσιμους τύπους δεδομένων (extensible data types) και μια ειδική ανάλυση (global-vs-local) για τα κλεισίματα και τις συναρτήσεις χειρισμού τους, ενώ η τεχνική μας υποστηρίζει το Σύστημα F με γενικευμένους αλγεβρικούς τύπους δεδομένων και χειρίζεται ομοιόμορφα όλα τα κλεισίματα, χρησιμοποιώντας ονόματα συναρτήσεων που περιέχουν ονόματα μονάδων. Σε επόμενη εργασία του [276], ο Tolmach αναφέρει ότι η τεχνική του μπορεί να εφαρμοστεί σε πολυμορφικά προγράμματα, χρησιμοποιώντας πάλι επεκτάσιμους τύπους δεδομένων.

Επίσης, η τεχνική μας επίσης υποστηρίζει ad-hoc πολυμορφισμό μέσω κλάσεων τύπων (type classes), αν χρησιμοποιηθεί η κωδικοποίηση concretization τους [227]. Αυτό διαχωρίζει την προσέγγισή μας από άλλους μεταγλωττιστές, όπως τον μονομορφισμό (monomorphisation) του MLton [54], την εναλλακτική προσέγγιση του JHC που χρησιμοποιεί ανάλυση ολόκληρου προγράμματος και τον λ-κύβο [171], και τις κλάσεις τύπων των GHC και UHC που υλοποιούνται με λεξικά [25, 76].

7.2 Ο τμηματικός νοηματικός μετασχηματισμός

Σε αυτήν την ενότητα θα δούμε πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υποστηρίξει τμηματική μεταγλώττιση. Θα δούμε πώς ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να μετασχηματίσει ξεχωριστά και στη συνέχεια να συνδέσει κώδικα της FL (Ενότητα 7.2.1). Θα δούμε επίσης πώς ο τμηματικός νοηματικός μετασχηματισμός συνδυάζεται με τον τμηματικό μετασχηματισμό defunctionalization (Ενότητα 7.2.2).

7.2.1 Νοηματικός μετασχηματισμός και τμηματική μεταγλώττιση

Έστω το εξής πρόγραμμα που αποτελείται από δύο ενότητες `Mod` και `Main`:

```
module Mod where

Mod.f x = x + x
Mod.g y = Mod.f y + 1

module Main where

import Mod (Mod.f/1 :: Int → Int,
            Mod.g/1 :: Int → Int)

Main.result = Mod.f 42 + Mod.g 10
```

Η εφαρμογή του γενικευμένου νοηματικού μετασχηματισμού στην ενότητα `Mod` παράγει τον αντίστοιχο νοηματικό κώδικα (θεωρώντας μια ψευδο-NVIL, όπου τα ονόματα συναρτήσεων μπορούν να περιέχουν ονόματα μονάδων):

```
── module Mod where
Mod.f = x + x
```

```
x = actuals(y)
Mod.g = call_0(Mod.f)
y = actuals()
```

Όταν προσπαθήσουμε να εφαρμόσουμε τον μετασχηματισμό στη μονάδα `Main`, εμφανίζεται το εξής πρόβλημα: η βοηθητική συνάρτηση *actdefs* του μετασχηματισμού (Εικόνα 3.3), η οποία δημιουργεί νέους ορισμούς `actuals`, πρέπει να γνωρίζει τις τυπικές παραμέτρους όλων των συναρτήσεων που καλούνται, οι οποίες δεν είναι γνωστές για συναρτήσεις που έχουν εισαχθεί από άλλες μονάδες. Αυτό μπορεί να διορθωθεί αν οι δηλώσεις `import`, αντί απλά για την πληθικότητα r, περιλαμβάνουν ολόκληρη τη λίστα τυπικών παραμέτρων κάθε εισαγόμενης συνάρτησης ή κατασκευαστή (το μήκος της λίστας θα είναι η πληθικότητα r).

Έστω ότι η πληροφορία των τυπικών παραμέτρων κάθε εισαγόμενης συνάρτησης υπάρχει και ότι προχωρούμε στον νοηματικό μετασχηματισμό της `Main`:

```
-- module Main where

Main.result = call_0(Mod.f) + call_0(Mod.g)
x = actuals(42)
y = actuals(10)
```

Αν συνενώσουμε τον παραγόμενο νοηματικό κώδικα για κάθε μονάδα, προσπαθώντας να παράγουμε το τελικό νοηματικό πρόγραμμα, προκύπτει ο εξής νοηματικός κώδικας:

```
-- module Mod where
Mod.f = x + x
x = actuals(y)
Mod.g = call_0(Mod.f)
y = actuals()
-- module Main where
Main.result = call_0(Mod.f) + call_0(Mod.g)
x = actuals(42)
y = actuals(10)
```

Σε αυτό το σημείο προκύπτει άλλο ένα πρόβλημα αν οι δείκτες είναι φυσικοί αριθμοί: υπάρχουν δύο διαφορετικοί ορισμοί για την `x` και δεν μπορούν να συνδυαστούν γιατί και οι δύο ορίζουν την τιμή της `x` για τον ίδιο νοηματικό δείκτη 0. Αυτό οφείλεται στο ότι οι δείκτες είναι αριθμοί που αντιστοιχούν σε σημεία ενός ενιαίου κειμένου προγράμματος· στην περίπτωση των μονάδων κώδικα, θα έπρεπε οι δείκτες που απαριθμούν τις κλήσεις του κειμένου μιας μονάδας να μην έχουν ποτέ προβλήματα συνωνυμίας με δείκτες κλήσεων άλλων μονάδων.

Το πρόβλημα των δεικτών μπορεί να αντιμετωπιστεί αν χρησιμοποιήσουμε ένα πιο πλούσιο σύνολο για τους δείκτες, που να ενσωματώνει πληροφορία για τη μονάδα κώδικα, στην οποία γίνεται κάθε κλήση. Αντί για φυσικοί αριθμοί, οι δείκτες πρέπει να είναι ζεύγη της μορφής (m, n), όπου το m είναι όνομα μονάδας και $n \geq 0$.

Θεωρώντας ότι οι δηλώσεις `import` του αρχικού προγράμματος περιέχουν λίστες τυπικών παραμέτρων, ορίζεται τότε η NVIL_M, μια παραλλαγή της NVIL με τις εξής διαφορές: (α) τα ονόματα συναρτήσεων και κατασκευαστών μπορούν να περιέχουν ονόματα μονάδων κώδικα και (β) το σύνολο *Labels* (που αναφέρθηκε στην Ενότητα 3.3.1) περιέχει πληροφορία μονάδων κώδικα.

$p ::= d_0, \ \ldots, \ d_n$ *πρόγραμμα*

$d ::= m.f = e$ *ορισμός*

$$
\begin{array}{lll}
e ::= c(e_0, \ldots, e_{n-1}) \mid m.f \mid m.\kappa & & \textit{έκφραση} \\
\quad \mid \texttt{case}\ e\ \texttt{of}\ \{\ b_0\ \texttt{;}\ \ldots\ \texttt{;}\ b_n\ \} \mid \#^m(e) \mid \texttt{call}_\ell(e) & & \\
\quad \mid \texttt{actuals}(\langle e_\ell \rangle_{\ell \in I}) & & \\
b ::= m.\kappa \to e & & \textit{ταίριασμα προτύπου}
\end{array}
$$

Αντίστοιχα με τον τμηματικό μετασχηματισμό defunctionalization, μπορούμε να ορίσουμε τον τμηματικό νοηματικό μετασχηματισμό, ο οποίος αποτελείται από δύο βήματα:

1. *Ξεχωριστός νοηματικός μετασχηματισμός:* Εφαρμόζεται ο γενικευμένος νοηματικός μετασχηματισμός σε κάθε μονάδα κώδικα και προκύπτει ένα σύνολο από ορισμούς σε NVIL_M.

2. *Σύνδεση νοηματικού κώδικα:* Ο νοηματικός κώδικας που έχει προκύψει από τον ξεχωριστό μετασχηματισμό του προηγούμενου βήματος συνδυάζεται για την παραγωγή του τελικού νοηματικού κώδικα.

Ξεχωριστός νοηματικός μετασχηματισμός. Χρησιμοποιώντας την NVIL_M, ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να εφαρμοστεί ξεχωριστά στις δύο μονάδες του παραδείγματος ως εξής:

```
-- module Mod where
Mod.f = x + x
x = actuals((Mod, 0):y)
Mod.g = call_(Mod, 0)(Mod.f)
y = actuals()

-- module Main where

Main.result = call_(Main, 0)(Mod.f) + call_(Main, 0)(Mod.g)
x = actuals((Main, 0):42)
y = actuals((Main, 0):10)
```

Σύνδεση νοηματικού κώδικα. Τα ξεχωριστά μετασχηματισμένα κομμάτια τμηματικού κώδικα μπορούν να συνδεθούν ως εξής:

- Οι δηλώσεις συναρτήσεων δεν έχουν κίνδυνο συνωνυμίας και μπορούν να τοποθετηθούν η μια μετά την άλλη, με οποιαδήποτε σειρά.

- Οι δηλώσεις `actuals` τυπικών μεταβλητών συνενώνονται, όταν αφορούν το ίδιο όνομα μεταβλητής – η μορφή των δεικτών δεν προκαλεί προβλήματα συνωνυμίας.

Ο τελικός νοηματικός κώδικας μετά τη σύνδεση θα είναι:

```
Mod.f = x + x
Mod.g = call_(Mod, 0)(Mod.f)
Main.result = call_(Main, 0)(Mod.f) + call_(Main, 0)(Mod.g)
x = actuals((Mod, 0):y, (Main, 0):42)
y = actuals((Main, 0):10)
```

7.2.2 Συνδυασμός του τμηματικού νοηματικού μετασχηματισμού με τον τμηματικό μετασχηματισμό defunctionalization

Ο τμηματικός νοηματικός μετασχηματισμός μπορεί να συνδυαστεί με τον τμηματικό μετασχηματισμό defunctionalization, ώστε η υλοποίησή μας να υποστηρίζει τμηματική τμηματική μεταγλώττιση.

Ο διπλός μετασχηματισμός που προκύπτει από τη σύνθεση των δύο αυτών μετασχηματισμών γίνεται σε δύο βήματα ως εξής:

1. *Ξεχωριστός μετασχηματισμός:* Σε κάθε μονάδα κώδικα, εφαρμόζεται το πρώτο βήμα του τμηματικού μετασχηματισμού defunctionalization, ακολουθούμενο από το πρώτο βήμα του τμηματικού νοηματικού μετασχηματισμού. Προκύπτουν: (α) ο νοηματικός κώδικας που αντιστοιχεί στις καθολικές συναρτήσεις και στις τυπικές παραμέτρους της μονάδας, και (β) η διεπαφή defunctionalization.

2. *Σύνδεση:* Η σύνδεση γίνεται σε τρία βήματα:

   (α) Παράγεται ο κώδικας που λείπει, χρησιμοποιώντας το δεύτερο βήμα του τμηματικού μετασχηματισμού defunctionalization.

   (β) Ο νέος αυτός κώδικας στη συνέχεια θεωρείται μια νέα ξεχωριστή μονάδα (που έμμεσα εισάγουν όλες οι υπόλοιπες μονάδες) και μετασχηματίζεται από το πρώτο βήμα του τμηματικού νοηματικού μετασχηματισμού.

   (γ) Ο νοηματικός κώδικας των αρχικών μονάδων και της νέας μονάδας συνδυάζεται χρησιμοποιώντας το δεύτερο βήμα του τμηματικού νοηματικού μετασχηματισμού και παράγεται το τελικό νοηματικό πρόγραμμα.

Ο παραπάνω διπλός μετασχηματισμός είναι κατάλληλος για την τμηματική μεταγλώττιση γλωσσών υψηλότερης τάξης σε νοηματικά προγράμματα, και άρα μπορεί να χρησιμοποιηθεί στην υλοποίηση του Κεφαλαίου 4.

7.3 Υλοποίηση με ΟΕΔ

Σε αυτήν την ενότητα θα περιγράψουμε πώς υποστηρίζεται η τμηματική μεταγλώττιση στην τεχνική υλοποίησης με ΟΕΔ που περιγράφηκε στο Κεφάλαιο 5.

Η υλοποίηση με ΟΕΔ παράγει κώδικα C · θα περιγράψουμε πώς χρησιμοποιούμε την τεχνολογία της C για να υποστηρίξουμε πιο εύκολα τμηματική μεταγλώττιση (Ενότητα 7.3.1). Επίσης, θα αναφέρουμε δύο λεπτομέρειες που προέκυψαν κατά την υλοποίηση της τμηματικής μεταγλώττισης, που είναι ο χειρισμός των μορφών CAF (Ενότητα 7.3.2) και ο χειρισμός πολλών παραμέτρων κατά την εφαρμογή εκφράσεων υψηλότερης τάξης (Ενότητα 7.3.3).

7.3.1 Τμηματική μεταγλώττιση και C

Όπως αναφέρθηκε στο Κεφάλαιο 5, η υλοποίηση με ΟΕΔ παράγει κώδικα C για κάθε συνάρτηση και τυπική παράμετρο του νοηματικού προγράμματος. Αυτό σημαίνει ότι ο διπλός μετασχηματισμός τμηματικής μεταγλώττισης της Ενότητας 7.2.2, πρέπει να τροποποιηθεί ώστε να παράγει κώδικα C. Οι αλλαγές αυτές είναι οι εξής:

- *Ξεχωριστή μεταγλώττιση σε C:* Κάθε μονάδα κώδικα μετασχηματίζεται ξεχωριστά σε νοηματικό κώδικα και διεπαφή defunctionalization από το πρώτο βήμα του διπλού μετασχηματισμού. Στη συνέχεια, ο νοηματικός κώδικας μετατρέπεται σε κώδικα C, ο οποίος μπορεί να μεταγλωττιστεί σε δυαδικό αρχείο αντικειμενικού κώδικα (αρχείο `.o`). Οι συναρτήσεις της C που αντιστοιχούν σε κατασκευαστές κλεισιμάτων και στη συνάρτηση `apply` του αρχικού προγράμματος θεωρούνται εξωτερικά σύμβολα (external symbols) και στον κώδικα C δηλώνονται ως `extern` [143]. Επίσης, ως `extern` δηλώνονται και οι συναρτήσεις που εισάγονται σε μια μονάδα από άλλες μονάδες κώδικα.

- *Σύνδεση:* Το δεύτερο βήμα του διπλού μετασχηματισμού παράγει τον νοηματικό κώδικα που λείπει, δημιουργώντας μια νέα μονάδα κώδικα, η οποία στη συνέχεια μεταγλωττίζεται σε C και σε αρχείο αντικειμενικού κώδικα. Τα αρχεία αντικειμενικού κώδικα από αυτό και το προηγούμενο βήμα, μπορούν να συνδεθούν από το κατάλληλο πρόγραμμα (όπως ο συνδέτης `ld` του εγχειρήματος GNU [93]) για την παραγωγή του τελικού εκτελέσιμου. Το πρόγραμμα που θα κάνει τη σύνδεση θα αναλάβει να επιλύσει όλα τα εξωτερικά σύμβολα του πρώτου βήματος παραπάνω.

Με τον παραπάνω τρόπο, η υλοποίηση με ΟΕΔ παράγει ενδιάμεσα αρχεία αντικειμενικού κώδικα για κάθε μονάδα κώδικα, μέσω του μεταγλωττιστή της C, και χρησιμοποιεί τον συνδέτη του λειτουργικού συστήματος. Έτσι, βελτιώσεις στον μεταγλωττιστή της C ή στον συνδέτη, μπορούν άμεσα να χρησιμοποιηθούν από την υλοποίηση.

Κάθε μονάδα κώδικα που μεταγλωττίζεται ξεχωριστά, δημιουργεί ένα συνοδευτικό αρχείο που περιέχει τη διεπαφή της (module interface). Αυτή η διεπαφή περιλαμβάνει: (α) τη διεπαφή defunctionalization και (β) πληροφορία τύπων και τυπικών μεταβλητών για κάθε συνάρτηση, ώστε ο χρήστης να μη χρειάζεται να την εισάγει στις δηλώσεις `import`. Ο ρόλος των διεπαφών αυτή μοιάζει με αυτόν που έχουν τα αρχεία `.hi` στον GHC, τα οποία επίσης κρατούν πληροφορία τύπων και πληθικότητας συναρτήσεων [168].

7.3.2 Τμηματική μεταγλώττιση και μορφές CAF

Όπως είδαμε και στη Ενότητα 5.4, οι μορφές CAF είναι οι συναρτήσεις χωρίς ορίσματα που τις θεωρούμε καθολικές οκνηρές μεταβλητές και τις κρατάμε σε μια ειδική ΟΕΔ. Στην τμηματική μεταγλώττιση, κάθε μονάδα κώδικα έχει τις δικές της μορφές CAF και άρα κρατά τη δική της ΟΕΔ για αυτές. Αυτό σημαίνει ότι κατά την έναρξη του προγράμματος, κάθε μονάδα κώδικα αναλαμβάνει να αρχικοποιήσει την ΟΕΔ με τις μορφές CAF που περιλαμβάνει. Επιπλέον, κάθε μονάδα κώδικα πρέπει να διαθέτει μεταδεδομένα που να περιγράφουν σε ποια θέση στην ΟΕΔ της υπάρχει κάθε μορφή CAF, ώστε να μπορούν να τις καλέσουν άλλες μονάδες κώδικα.

7.3.3 Χειρισμός πολλών παραμέτρων κατά την εφαρμογή εκφράσεων υψηλότερης τάξης

Για λόγους απλότητας, ο τμηματικός μετασχηματισμός defunctionalization που περιγράψαμε σε αυτό το κεφάλαιο, χειρίζεται την εφαρμογή μιας έκφρασης υψηλότερης τάξης πάνω σε *μια* παράμετρο. Για παράδειγμα, έστω η εξής συνάρτηση του αρχικού προγράμματος:

```
aux f a b = f a (b+1)
```

Η συνάρτηση αυτή, μετά τον μετασχηματισμό defunctionalization, γίνεται:

```
aux f a b = apply (apply f a) (b+1)
```

Στην πράξη, η υλοποίηση απλοποιεί αυτές τις δύο κλήσεις της `apply`, χρησιμοποιώντας μια νέα συνάρτηση `apply2`, η οποία παίρνει δύο παραμέτρους:

```
aux f a b = apply2 f a (b+1)
```

Αυτό σημαίνει ότι στο τελικό πρόγραμμα, εκτός της κανονικής συνάρτησης χειρισμού κλεισιμάτων, υπάρχουν και άλλες βοηθητικές συναρτήσεις που αντιστοιχούν στην εφαρμογή εκφράσεων υψηλότερης τάξης σε περισσότερες της μιας παραμέτρους. Η κατασκευή αυτών των νέων συναρτήσεων γίνεται με παρόμοιο τρόπο με αυτόν που περιγράψαμε για τη συνάρτηση $\mathcal{A}$ στην Ενότητα 7.1.

7.4 Καταληκτικές παρατηρήσεις

Στο κεφάλαιο αυτό παρουσιάστηκαν δύο παραλλαγές των βασικών μετασχηματισμών αυτής της διατριβής (νοηματικού και μετασχηματισμού defunctionalization), οι οποίες επιτρέπουν την τμηματική μεταγλώττιση μονάδων κώδικα.

Από όσο γνωρίζουμε, η προσέγγισή μας είναι η πρώτη υλοποίηση του μετασχηματισμού defunctionalization, η οποία επιτρέπει την τμηματική μεταγλώττιση πολυμορφικών συναρτησιακών προγραμμάτων σε δυαδικό εκτελέσιμο κώδικα. Η τεχνική αυτή μπορεί να χάνει ευκαιρίες για απαλοιφή κώδικα· αυτό αποτελεί γενικότερο πρόβλημα της τμηματικής μεταγλώττισης και μπορεί να αντιμετωπιστεί με τεχνικές βελτιστοποίησης κατά τον χρόνο σύνδεσης (link-time optimization [41, 69, 89]). Επίσης, αν και περιγράψαμε μόνο την περίπτωση τμηματικής μεταγλώττισης και στατικής σύνδεσης, η τεχνική μας μπορεί να λειτουργήσει χωρίς σημαντικές αλλαγές και σε μια υλοποίηση με δυνατότητα δυναμικής σύνδεσης, θεωρώντας ότι ο δυναμικός φορτωτής (dynamic loader) και ο δυναμικός συνδέτης (dynamic linker) μπορούν να χρησιμοποιήσουν τις διεπαφές defunctionalization.

Επίσης, ο τμηματικός νοηματικός μετασχηματισμός που περιγράψαμε αποτελεί φυσική γενίκευση του νοηματικού μετασχηματισμού με μονάδες κώδικα. Ο μετασχηματισμός αυτός είναι ανεξάρτητος του μετασχηματισμού defunctionalization (αν και όπως είδαμε στην Ενότητα 7.2.2 μπορούν να συνδυαστούν) και μπορεί να χρησιμοποιηθεί για να μετατρέψει μεγάλα συναρτησιακά προγράμματα σε νοηματικά. Από όσο γνωρίζουμε, ο συνδυασμός μονάδων κώδικα και νοηματικού μετασχηματισμού δεν έχει περιγραφεί στη βιβλιογραφία του νοηματικού προγραμματισμού.

Η τεχνική τμηματικής μεταγλώττισης που περιγράψαμε απαιτεί την παραγωγή κάποιου κώδικα FOFL, NVIL και C κατά τον χρόνο σύνδεσης. Η παραγωγή κώδικα στον χρόνο σύνδεσης έχει περιγραφεί πάλι στην εργασία των Swasey *et al.* για την τμηματική μεταγλώττιση μονάδων κώδικα στη συναρτησιακή γλώσσα προγραμματισμού Standard ML [268].

Κεφάλαιο 8

Επίλογος

Στην Ενότητα 8.1 συνοψίζονται τα αποτελέσματα αυτής της διδακτορικής διατριβής και αναφέρονται οι σχετικές δημοσιεύσεις. Με αφετηρία τα αποτελέσματα αυτής της εργασίας, στη συνέχεια θα αναφερθούν μελλοντικές κατευθύνσεις έρευνας σε θεωρητικό επίπεδο (Ενότητα 8.2) αλλά και σε επίπεδο υλοποίησης (Ενότητα 8.3).

8.1 Συνεισφορά

Στη διατριβή αυτή παρουσιάστηκαν ο γενικευμένος νοηματικός μετασχηματισμός, που είναι μια τεχνική που μετασχηματίζει μη αυστηρά συναρτησιακά προγράμματα σε νοηματικά προγράμματα. Αυτός ο νέος νοηματικός μετασχηματισμός, σε συνδυασμό με τον μετασχηματισμό defunctionalization, διορθώνει τα δύο βασικά προβλήματα του κλασικού νοηματικού μετασχηματισμού, που ήταν η υποστήριξη αυθαίρετων τύπων δεδομένων και ο μετασχηματισμός εκφράσεων υψηλότερης τάξης. Αποδείχτηκε επίσης ότι οι δύο κλασικές παραλλαγές του νοηματικού μετασχηματισμού έχουν την ίδια εκφραστική δύναμη. Ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί αποδοτικά σε δημοφιλείς αρχιτεκτονικές υπολογιστών, κάνοντας χρήση μιας φιλικής στη μνήμη αναπαράστασης των δομών του χρόνου εκτέλεσης. Τέλος οι δύο μετασχηματισμοί της υλοποίησης (νοηματικός και defunctionalization) μπορούν να υποστηρίξουν τμηματική μεταγλώττιση.

Από τα αποτελέσματα της διατριβής προέκυψαν πέντε δημοσιεύσεις σε πρακτικά συνεδρίων με κρίση και μία δημοσίευση σε επιστημονικό περιοδικό. Ακολουθεί μία σύντομη ανάλυσή τους:

[1] Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The intensional transformation for functional languages with user-defined data types. In *Proceedings of the 8th Panhellenic Logic Symposium*, pages 38–42, 2011.

Στην εργασία αυτή, παρουσιάσαμε για πρώτη φορά τη βασική ιδέα πίσω από τον γενικευμένο νοηματικό μετασχηματισμό, δίνοντας μια αφηρημένη μηχανή για την εκτέλεση νοηματικών προγραμμάτων, τα οποία περιέχουν κατασκευαστές δεδομένων και ταίριασμα προτύπων.

[2] Georgios Fourtounis, Peter Csaba Ölveczky, and Nikolaos Papaspyrou. Formally specifying and analyzing a parallel virtual machine for lazy functional languages using Maude. In *Proceedings of the 5th International Workshop on High-level Parallel Programming and Applications* (HLPP'11), pages 19–26, 2011.

Με βάση τον κλασικό νοηματικό μετασχηματισμό πρώτης τάξης, δώσαμε μια παράλληλη αφηρημένη μηχανή για την εκτέλεση νοηματικών προγραμμάτων. Παρουσιάσαμε

τα εξής αποτελέσματα: (1) Η σημασιολογία αυτής της παράλληλης μηχανής δόθηκε τυπικά, σε λογική αναγραφής, χρησιμοποιώντας το εργαλείο Maude για τον εντοπισμό σφαλμάτων στην τυπική περιγραφή της μηχανής. (2) Η αποθήκη που παρουσιάσαμε ήταν κατανεμημένη και υποστήριζε ασφαλή ταυτόχρονη οκνηρή αποτίμηση.

Τα αποτελέσματα αυτής της έργασίας δεν αναφέρθηκαν αναλυτικά στην παρούσα διατριβή, γιατί η εφαρμογή τους στον γενικευμένο νοηματικό μετασχηματισμό αφορά μελλοντική έρευνα, όπως αναφέρουμε στην Ενότητα 8.3.2.

[3] Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The generalized intensional transformation for implementing lazy functional languages. In Konstantinos F. Sagonas, editor, *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages* (PADL '13), volume 7752 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2013.

Αυτή η εργασία αποτελεί συνέχεια και επέκταση της εργασίας [1], στην οποία είχαμε παρουσιάσει τον γενικευμένο νοηματικό μετασχηματισμό. Σε σχέση με την εργασία [1], τα νέα αποτελέσματα που αναφέρονται είναι: (1) Η τυπική περιγραφή του γενικευμένου νοηματικού μετασχηματισμού. (2) Η τεχνική υλοποίησής του γενικευμένου νοηματικού μετασχηματισμού με ΟΕΔ. (3) Η μελέτη της αποδοτικότητας της τεχνικής υλοποίησης με ΟΕΔ, συγκρίνοντας την υλοποίηση αυτή με δημοφιλείς μεταγλωττιστές της Haskell.

[4] Georgios Fourtounis and Nikolaos S. Papaspyrou. Supporting separate compilation in a defunctionalizing compiler. In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 39–49, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Σε αυτήν την εργασία, παρουσιάσαμε πώς ο μετασχηματισμός defunctionalization μπορεί να υποστηρίξει τμηματική μεταγλώττιση, για μια γλώσσα με μονομορφικό σύστημα τύπων. Ως αποτέλεσμα, δώσαμε μια τυπική περιγραφή μιας παραλλαγής του μετασχηματισμού defunctionalization των Bell *et al.* [33], προσθέτοντας υποστήριξη για τον ξεχωριστό μετασχηματισμό και στη συνέχεια σύνδεση μονάδων κώδικα.

[5] Georgios Fourtounis, Nikolaos Papaspyrou, and Panagiotis Theofilopoulos. Modular polymorphic defunctionalization. *Computer Science and Information Systems*. Accepted for publication, to appear.

Η εργασία αυτή αποτελεί συνέχεια και επέκταση, με τη μορφή άρθρου σε περιοδικό, της εργασίας [4]. Δημοσιεύτηκε στο ειδικό τεύχος με τα postproceedings του *2nd Symposium on Languages, Applications and Technologies*, κατόπιν επιλογής και εκ νέου κρίσης. Σε σχέση με την εργασία [4], τα νέα αποτελέσματα που αναφέρονται σε αυτή είναι: (1) Η τροποποίηση της τεχνικής ώστε να υποστηρίζεται παραμετρικός πολυμορφισμός, χρησιμοποιώντας μια προσαρμογή της τεχνικής των Pottier and Gauthier [227]. (2) Η προσθήκη υποστήριξης για κλήση συναρτήσεων υψηλής τάξης με περισσότερες πραγματικές παραμέτρους από τις τυπικές που αναφέρονται στον ορισμό τους (over-saturated higher-order function calls). (3) Η περιγραφή ενός εύκολου και αποδοτικού στην υλοποίησή του ευριστικού μηχανισμού για τη σημαντική μείωση του πλήθους των κλεισιμάτων που πρέπει να παραχθούν κατά τον μετασχηματισμό defunctionalization.

[6] Georgios Fourtounis and Nikolaos Papaspyrou. An efficient representation for lazy constructors using 64-bit pointers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on*

Functional High-performance Computing (FHPC'14), 2014. Accepted for presentation, to appear.

Σε αυτήν την εργασία, παρουσιάσαμε μια αποδοτική αναπαράσταση για την τεχνική υλοποίησης με ΟΕΔ, για υπολογιστές αρχιτεκτονικής AMD64. Παρουσιάσαμε τα εξής αποτελέσματα: (1) Μια τεχνική αναπαράστασης ενός thunk με μια λέξη των 64 bit, που περιλαμβάνει όλη την πληροφορία που χρειάζεται για την αποτίμησή του. (2) Μια διεπαφή μεταξύ της αναπαράστασής μας και ενός μηχανισμού για την ακριβή συλλογή σκουπιδιών. (3) Τη σύγκριση της υλοποίησής μας με την δημοφιλέστερη υλοποίηση της Haskell (GHC), με την τελευταία να έχει ενεργοποιημένες όλες τις βελτιστοποιήσεις της.

8.2 Θεωρητικές επεκτάσεις

Ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να μετασχηματίσει πολυμορφικά συναρτησιακά προγράμματα υψηλότερης τάξης αλλά η γλώσσα του δεν περιλαμβάνει τοπικές δηλώσεις `let`. Όπως αναφέρθηκε στην Ενότητα 5.4, η υλοποίηση περιλαμβάνει ένα στάδιο μετασχηματισμού lambda-lifting για τον χειρισμό αυτών των τοπικών δηλώσεων. Η υποστήριξη `let` από τον γενικό νοηματικό μετασχηματισμό αποτελεί μελλοντικό αντικείμενο έρευνας που θα έφερνε τις νοηματικές γλώσσες και τις γλώσσες ροής δεδομένων πιο κοντά στη σύνταξη των συναρτησιακών γλωσσών προγραμματισμού. Όπως αναφέρθηκε στην Ενότητα 3.3.1, ο νοηματικός μετασχηματισμός του Yaghi παρείχε μια μορφή των συμφραζομένων που επέτρεπε εμφωλευμένες δηλώσεις [302].

Ο κλασικός νοηματικός μετασχηματισμός είχε και μια χρονική ερμηνεία στον χρονικό λογικό προγραμματισμό [238, 239], με τα συμφραζόμενα να αντιστοιχούν σε στιγμές ενός διακλαδιζόμενου χρόνου (όπως αναφέρθηκε στην Ενότητα 2.1). Ακολουθώντας αυτήν την ερμηνεία, ο γενικευμένος νοηματικός μετασχηματισμός συνδέει διαφορετικές στιγμές μέσω των εμφωλευμένων συμφραζομένων, με τον μηχανισμό αλλαγής συμφραζομένων των δεσμευμένων μεταβλητών να αντιστοιχεί σε έναν τελεστή επιστροφής στο παρελθόν (ο οποίος αποτελεί ειδική μορφή του τελεστή `at` της Lucid που επέλεγε αυθαίρετες στιγμές σε μια ροή). Θα ήταν ενδιαφέρον να εξερευνηθεί η εφαρμογή του γενικευμένου νοηματικού μετασχηματισμού στον χρονικό λογικό προγραμματισμό, ή η συσχέτισή του με άλλες υπάρχουσες χρονικές λογικές.

Όπως αναφέραμε στην Ενότητα 4.1, η υλοποίηση του γενικευμένου νοηματικού μετασχηματισμού με αποθήκη μπορεί να μιμηθεί, πέρα από την οκνηρή, και άλλες στρατηγικές αποτίμησης, οι οποίες αξίζει να εξερευνηθούν. Για παράδειγμα, η αποθήκη μπορεί να λειτουργήσει παρόμοια με την κρυφή μνήμη του υλικού και να εξερευνηθεί η εκτέλεση προγραμμάτων με διαφορετικές στρατηγικές έξωσης από την κρυφή μνήμη· θα μπορούσε επίσης να μελετηθεί η συμπεριφορά της αποθήκης όταν διαγράφει τυχαία ήδη υπολογισμένες τιμές, κάτι που μπορεί να εκφράσει την ανθεκτικότητά της σε σενάρια αποτυχίας του προγράμματος ή αστοχίας υλικού, ή να περιγράψει προγράμματα που αυτο-βελτιστοποιούνται [278, σ. 18].

Τέλος, τα εμφωλευμένα συμφραζόμενα μοιάζει να σχετίζονται με τις φραγμένες συνέχειες (delimited continuations). Η μη αυστηρή σημασιολογία του ταιριάσματος προτύπων χρησιμοποιεί τα εμφωλευμένα συμφραζόμενα για να οπισθοχωρήσει (backtracking) από τα συμφραζόμενα ενός κλάδου ταιριάσματος προτύπων στα συμφραζόμενα ενός κατασκευαστή. Η διαφορά μεταξύ των δύο συμφραζομένων μοιάζει με μια φραγμένη συνέχεια που χρησιμοποιείται για οπισθοχώρηση [66, σ. 17]. Η αρχική σημασιολογία του γενικευμένου νοηματικού μετασχηματισμού που δόθηκε το 2011 από τους Φουρτούνη, Παπασπύρου και

Ροντογιάννη [96] περιείχε ζεύγη συμφραζομένων που θα μπορούσαν να θεωρηθούν νοηματικά αντίστοιχα φραγμένων συνεχειών· περαιτέρω έρευνα μπορεί να δείξει τη σχέση μεταξύ του γενικευμένου νοηματικού μετασχηματισμού και των φραγμένων συνεχειών.

8.3 Θέματα υλοποίησης

Η υλοποίηση που παρουσιάσαμε σε αυτήν τη διατριβή έχει κάποια σημεία που μπορούν να βελτιωθούν και να επεκταθούν (Ενότητα 8.3.1). Προτείνουμε επίσης πιθανές στρατηγικές παράλληλης υλοποίησης (Ενότητα 8.3.2).

8.3.1 Γενικές βελτιώσεις

Η υλοποίηση αυτή αφορά την HL_M και άρα χρειάζεται κάποιους επιπλέον μετασχηματισμούς για να μπορεί να θεωρηθεί μεταγλωττιστής της Haskell. Στην πράξη, αυτό θα μπορούσε να επιτευχθεί αν η υλοποίησή μας μετατρεπόταν σε οπίσθιο τμήμα (back-end) κάποιου δημοφιλούς μεταγλωττιστή της Haskell, όπως ο GHC. Μια τέτοια προσέγγιση θα επέτρεπε την επαναχρησιμοποίηση της σημαντικής δουλειάς που έχει γίνει από άλλους ερευνητές σε μετασχηματισμούς και βελτιστοποιήσεις προγραμμάτων Haskell.[1]

Όσον αφορά το στάδιο της παραγωγής κώδικα, η υλοποίησή μας θα μπορούσε να συγκρίνει την απόδοση του κώδικα C που παράγει με αντίστοιχο κώδικα bit (bitcode) του δημοφιλούς εγχειρήματος LLVM [152]. Επίσης, η υλοποίησή μας δεν υποστηρίζει κλήσεις ουράς (tail calls), εκτός από αυτές που εισάγει ο μεταγλωττιστής της C · η υποστήριξη βελτιστοποίησης κλήσης ουράς (tail-call optimization) θα βελτίωνε και άλλο την ήδη σημαντική εξοικονόμηση μνήμης του Κεφαλαίου 6. Η κλήση ουράς δεν αφορά μόνο την υλοποίηση με ΟΕΔ αλλά έχει μελετηθεί και στα πλαίσια του νοηματικού προγραμματισμού [102].

8.3.2 Στρατηγικές παράλληλης υλοποίησης

Η τελευταία δεκαετία χαρακτηρίστηκε από τη ραγδαία εξάπλωση των πολυπύρηνων επεξεργαστών (multicores) και άλλου παράλληλου υλικού, μιας και οι απλοί επεξεργαστές ενός πυρήνα δεν μπορούν πια να κλιμακώσουν και άλλο τη συχνότητα εκτέλεσής τους, λόγω θερμικών προβλημάτων [145]. Η εμφάνιση παράλληλων υπολογιστών σε μεγάλη κλίμακα έστρεψε τους προγραμματιστές στις ιδέες του παράλληλου προγραμματισμού, ώστε να χρησιμοποιηθεί αποδοτικά το νέο υλικό. Στις επόμενες παραγράφους θα αναφερθούμε σε κατευθύνσεις υλοποίησης για δύο τύπους υλικού: για παράλληλες αρχιτεκτονικές von Neumann και για παράλληλες αρχιτεκτονικές ροής δεδομένων.

Παράλληλη υλοποίηση με ΟΕΔ

Το πιο δημοφιλές παράλληλο υλικό αρχιτεκτονικής von Neumann είναι η πολυπύρηνοι επεξεργαστές κοινής μνήμης (shared-memory multicores). Η τεχνική υλοποίησης με ΟΕΔ των Κεφαλαίων 5 και 6 μπορεί να υποστηρίξει την εκτέλεση σε αυτό το υλικό, με τις κατάλληλες επεκτάσεις:

- Το σύστημα χρόνου εκτέλεσης πρέπει να υποστηρίζει παράλληλη δέσμευση μνήμης και συλλογή σκουπιδιών.

[1] Η υλοποίησή μας ήδη χρησιμοποιεί την μηχανή ελέγχου και συμπερασμού τύπων (type inference engine) του GHC.

- Κάθε thunk πρέπει να διαθέτει κάποιον μηχανισμό συγχρονισμού που να εξασφαλίζει την οκνηρή ταυτόχρονη αποτίμησή του.

- Πρέπει να προστεθεί τόσο στο σύστημα χρόνου εκτέλεσης, όσο και στην ίδια τη γλώσσα, υποστήριξη για κάποιον ελαφρύ μηχανισμό παράλληλης αποτίμησης εκφράσεων. Παράδειγμα ενός τέτοιου μηχανισμού είναι οι τελεστές `par` και `pseq`, και ο μηχανισμός των *sparks* του GHC [167].

Πιστεύουμε ότι μια τέτοια υλοποίηση θα έπρεπε να μελετηθεί, για να φανεί η συμπεριφορά του μετασχηματισμού defunctionalization και της αναπαράστασης του Κεφαλαίου 6 σε παράλληλα περιβάλλοντα, όπου η πίεση στη μνήμη είναι αυξημένη και η απόδοση του προγράμματος επηρεάζεται από θέματα συνάφειας κρυφής μνήμης (cache coherence).

Υλοποιήσεις για αρχιτεκτονικές ροής δεδομένων

Στην πράξη, ο προγραμματισμός παράλληλων εφαρμογών με κλιμακούμενη απόδοση φάνηκε ότι είναι πολύ διαφορετικός από τον παραδοσιακό προγραμματισμό για έναν επεξεργαστή. Οι προγραμματιστές συχνά συναντούν δύσκολα προβλήματα ταυτοχρονισμού και χρειάζεται να αναπτυχθούν νέα προγραμματιστικά παραδείγματα για να χρησιμοποιηθεί αποδοτικά το παράλληλο υλικό [266]. Επιπλέον, οι πολυπύρηνοι επεξεργαστές φαίνεται να αντιμετωπίζουν και αυτοί προβλήματα όσον αφορά την κλιμάκωση των επιδόσεών τους [39, 83].

Σε αυτές τις συνθήκες, ο προγραμματισμός ροής δεδομένων και το μοντέλο εκτέλεσής του είναι πάλι δημοφιλή, λόγω του τρόπου με τον οποίο εκμεταλλεύονται τον υπάρχοντα παραλληλισμό ενός προγράμματος [34, 73, 124, 156, 163, 175, 185]. Για παράδειγμα, πρόσφατα, οι I-δομές (που μοιράζονται ιδέες με την αποθήκη του Κεφαλαίου 4 και με τις παράλληλες νοηματικές αποθήκες [94]), γενικεύθηκαν από τους Kuper και Newton για την κατασκευή ντετερμινιστικών παράλληλων προγραμμάτων [150], ενώ οι Doeraene και Van Roy παρουσίασαν έναν δηλωτικό πυρήνα ροής δεδομένων για τη γλώσσα Scala, ο οποίος υποστηρίζει οκνηρή αποτίμηση και ταυτοχρονισμό με πέρασμα μηνυμάτων [79].

Το σύγχρονο παράλληλο υλικό επίσης ενσωματώνει έννοιες του προγραμματισμού ροής δεδομένων: συνεχίζουν να αναπτύσσονται νέες αρχιτεκτονικές ροής δεδομένων [87, 104, 112, 144, 248, 267, 271, 303] ενώ τεχνολογία ροής δεδομένων προστίθεται και σε παραλλαγές παλαιότερων αρχιτεκτονικών [85, 172, 214, 246].

Η κομψότητα της σημασιολογίας των συναρτησιακών γλωσσών έχει συνδυαστεί αρκετές φορές με τεχνικές υλοποίησης σε χαμηλό επίπεδο: έχουν υπάρξει αρκετές προσπάθειες υλοποίησης συναρτησιακών γλωσσών σε υλικό [10, 24, 32, 37, 68, 100, 106, 111, 139, 189, 192, 202, 251, 261, 263, 296] ενώ ιδέες του συναρτησιακού προγραμματισμού έχουν ενσωματωθεί σε γλώσσες περιγραφής υλικού (hardware description languages) [187, 195, 211, 256, 258]. Επιπλέον, η διάδοση του επαναδιαμορφώσιμου υλικού (reconfigurable hardware), όπως τα FPGA, προσφέρει νέες επιλογές για τη μεταγλώττιση προγραμμάτων σε υλικό (hardware compilation), για γλώσσες υψηλού επιπέδου [90, 109, 162, 247, 257], και ειδικά για γλώσσες ροής δεδομένων [43, 44, 92, 120, 141, 246, 259, 272, 280, 304]. Σε αυτό το πλαίσιο, ο γενικευμένος νοηματικός μετασχηματισμός προτείνει μια νέα κατεύθυνση για τη μετατροπή μη αυστηρών συναρτησιακών προγραμμάτων σε υλικό, με βάση το μοντέλο ροής δεδομένων με ετικέτες.

Η υλοποίηση μιας αποδοτικής παράλληλης αποθήκης σε υλικό φαίνεται να είναι βασική απαίτηση για μια γρήγορη υλοποίηση σε υλικό ροής δεδομένων. Σε σχετική εργασία με αυτή τη διατριβή, οι Φουρτούνης, Ölveczky και Παπασπύρου, έδωσαν ταυτόχρονη σημασιολογία για μια παράλληλη αποθήκη, καθώς και μια παράλληλη εικονική μηχανή σε λογισμικό [94].

Στην πράξη, η υλοποίηση των μηχανισμών της Ενότητας 4 σε υλικό ροής δεδομένων βασίζεται σε μνήμες προσπελάσιμες με βάση το περιεχόμενο (Content-Addressable Memory, CAM [203]), οι οποίες κατά τη δεκαετία του 1980 ήταν ακριβές [147, 205] αλλά σήμερα είναι πιο οικονομικές [133, 233] και μπορούν να υλοποιηθούν και με επαναδιαμορφώσιμο υλικό [29, 131, 298].

Το μοντέλο ροής δεδομένων σχετίζεται επίσης με το προγραμματιστικό παράδειγμα Map-Reduce [71]: οι Gates *et al*. υλοποίησαν ένα σύστημα ροής δεδομένων πάνω σε μια πλατφόρμα Map-Reduce [103], ενώ οι Χαραλαμπίδης, Παπασπύρου και Ροντογιάννης προτείνουν το μοντέλο ροής δεδομένων με ετικέτες ως μοντέλο του Map-Reduce [56]. Αυτό σημαίνει ότι ο γενικευμένος νοηματικός μετασχηματισμός μπορεί να υλοποιηθεί σε αντίστοιχες πλατφόρμες και να χρησιμοποιήσει αποτελέσματα της έρευνας πάνω σε αλγόριθμους Map-Reduce.

Ευρετήριο

Βιβλιογραφία

[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[2] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Proceedings from the Haskell Workshop at FPCAA '95*, pages 35–49, 1995.

[3] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Michal Cierniak, Marsha Eng, Jesse Fang, Brian T. Lewis, Brian R. Murphy, and James M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 100–110, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Mads Sig Ager, Dariusz Biernacki, Olivier. Danvy, and Jan. Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, March 2003.

[5] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, University of Aarhus, Aarhus, Denmark, March 2003.

[6] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19, New York, NY, USA, 2003. ACM.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Vasu S. Alagar, Joey Paquet, and Kaiyu Wan. Intensional programming for agent communication. In João Leite, Andrea Omicini, Paolo Torroni, and Pınar Yolum, editors, *Declarative Agent Languages and Technologies II*, volume 3476 of *Lecture Notes in Computer Science*, pages 239–255. Springer Berlin Heidelberg, 2005.

[9] John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978.

[10] Makoto Amamiya and Rin-ichiro Taniguchi. An ultra-multiprocessing machine architecture for efficient parallel execution of functional languages. In Akinori Yonezawa and Takayasu Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 491 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 1991.

[11] AMD. AMD64 architecture programmer's manual volume 2: 128-bit and 256-bit media instructions, rev. 3.16. `http://developer.amd.com/wordpress/media/2012/10/26568_APM_v4.pdf`, 2012. Accessed: 31-01-2014.

[12] AMD. AMD64 architecture programmer's manual volume 2: System programming, rev. 3.23. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`, 2012. Accessed: 31-01-2014.

[13] Andrew W. Appel. Runtime tags aren't necessary. *LISP and Symbolic Computation*, 2(2):153–162, 1989.

[14] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 13–23, New York, NY, USA, 1994. ACM.

[15] Guy Argo. Improving the three instruction machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 100–115, New York, NY, USA, 1989. ACM.

[16] Arvind. Dataflow: Passing the token. `http://csg.csail.mit.edu/Users/arvind/ISCAfinal.pdf`, June 2005. Keynote Address at the 32nd Annual International Symposium on Computer Architecture.

[17] Arvind and David E. Culler. Dataflow architectures. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science Vol. 1, 1986*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[18] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39:300–318, March 1990.

[19] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[20] Mike Ash. Friday Q&A 2013-09-27: ARM64 and you. `https://www.mikeash.com/pyblog/friday-qa-2013-09-27-arm64-and-you.html`. Accessed: 24-01-2014.

[21] Edward A. Ashcroft, Anthony A. Faustini, Rangaswamy Jagannathan, and William W. Wadge. *Multidimensional Programming*. Oxford University Press, 1995.

[22] Edward A. Ashcroft and William W. Wadge. Lucid – a formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3):336–354, 1976.

[23] Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[24] Lennart Augustsson. BWM: A concrete machine for graph reduction. In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 36–50, London, UK, UK, 1992. Springer-Verlag.

[25] Lennart Augustsson. Implementing Haskell overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73. ACM Press, 1993.

[26] Louis-Rémi Babé. Firefox 4 performance. `https://hacks.mozilla.org/2011/03/firefox4-performance/`, 2011. Accessed: 31-01-2014.

[27] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[28] John Backus. The history of FORTRAN I, II, and III. *SIGPLAN Notices - Special issue: History of programming languages conference*, 13(8):165–180, August 1978.

[29] Zachary K. Baker and Justin L. Tripp. Tuple spaces in hardware for accelerated implicit routing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 166–173, May 2011.

[30] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer Berlin Heidelberg, 2001.

[31] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Tom S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[32] Henk P. Barendregt, Marko C. J. D. Van Eekelen, Marinus J. Plasmeijer, Pieter H. Hartel, Louis O. Hertzberger, and Willem G. Vree. The Dutch parallel reduction machine project. *Future Generation Computer Systems*, 3(4):261–270, 1987.

[33] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 2nd International Conference on Functional Programming*, pages 25–37. ACM, 1997.

[34] Richard Blewett and Andrew Clymer. TPL Dataflow. In *Pro Asynchronous Programming with .NET*, pages 195–232. Apress, 2013.

[35] Daniel G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975.

[36] Hans Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`, 2014. Accessed: 01-03-2014.

[37] Arjan Boeijink, Philip K.F. Hözenspies, and Jan Kuper. Introducing the PilGRIM: A processor for executing lazy functional languages. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 54–71. Springer Berlin Heidelberg, 2011.

[38] Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proceedings of IFL '96, volume 1268 of LNCS*, pages 58–84. Springer-Verlag, 1996.

[39] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[40] Shekhar Y. Borkar, Hans Mulder, Pradeep Dubey, Stephen S. Pawlowski, Kevin C. Kahn, David J. Kuck, and Justin R. Rattner. Platform 2015: Intel processor and platform evolution for the next decade. Whitepaper, Intel, 2005.

[41] Dominique Boucher. GOld: a link-time optimizer for Scheme. In *Proceedings of the Workshop on Scheme and Functionnal Programming*, pages 1–11, Houston, TX, USA, 2000. Rice University.

[42] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with ODEs. In *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Philadelphia, USA, March 2013.

[43] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, volume 2438 of *Lecture Notes in Computer Science*, pages 853–863. Springer Berlin Heidelberg, 2002.

[44] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 14–26, New York, NY, USA, 2004. ACM.

[45] Luca Cardelli. Compiling a functional language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 208–217, New York, NY, USA, 1984. ACM.

[46] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 266–277, New York, NY, USA, 1997. ACM.

[47] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[48] Tom A. Cargil. Deterministic operational semantics for Lucid. Technical Report CS-76-19, University of Waterloo, Ontario, 1976.

[49] Alejandro Caro. A novel 64 bit data representation for garbage collection and synchronizing memory. Technical Report Computation Structures Group Memo 396, Intel, April 1997.

[50] Alejandro Caro. *Generating Multithreaded Code from Parallel Haskell for Symmetric Multiprocessors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 1999.

[51] Paul Caspi. What can we learn from synchronous data-flow languages? In Oded Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 255–258. Springer Berlin Heidelberg, 1997.

[52] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[53] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[54] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *Programming Languages and Systems*, volume 1782 of *Lecture Notes in Computer Science*, pages 56–71. Springer Berlin Heidelberg, 2000.

[55] Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008.

[56] Angelos Charalambidis, Nikolaos Papaspyrou, and Panos Rondogiannis. Tagged dataflow: a formal model for iterative Map-Reduce. In *Proceedings of the 2014 International Workshop on Algorithms for MapReduce and Beyond (BeyondMR)*. ACM, 2014.

[57] Weidong Chen, Michael Kifer, and David S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

[58] David Chernicoff. The corporate datacenter – it's ready for Windows Server x64 editions. `http://download.microsoft.com/download/4/4/7/44715635-144e-4726-8705-b8ec83c0a34d/The%20Corporate%20Datacenter-It%27s%20Ready%20for%20Windows%20Server%20x64%20Editions.pdf`, 2007. Accessed: 31-01-2014.

[59] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, December 2000.

[60] Wei-Ngan Chin and John Darlington. A higher-order removal method. *Higher-order and Symbolic Computation / Lisp and Symbolic Computation*, 9:287–322, 1996.

[61] Alonzo Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):839–864, October 1933.

[62] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, pages 230–239, New York, NY, USA, 2004. ACM.

[63] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM.

[64] Intel Corporation. Excerpts from a conversation with Gordon Moore: Moore's Law. `http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf`, 2005.

[65] David E. Culler and Gregory M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990. Data-flow Processing.

[66] Olivier Danvy. An analytical approach to programs as data objects. Doctor scientarum thesis, University of Aarhus, 2006.

[67] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM.

[68] John Darlington and Mike Reeve. ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 65–76, New York, NY, USA, 1981. ACM.

[69] Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter. Post-pass compaction techniques. *Communications of the ACM*, 46(8):41–46, August 2003.

[70] Alberto De La Encina and Ricardo Peña. From natural semantics to C: A formal derivation of two STG machines. *Journal of Functional Programming*, 19(01):47–94, 2009.

[71] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[72] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer Berlin Heidelberg, 1974.

[73] Jack B. Dennis. A parallel program execution model supporting modular software construction. In *Proceedings of the Conference on Massively Parallel Programming Models*, MPPM '97, pages 50–60, Washington, DC, USA, 1997. IEEE Computer Society.

[74] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.

[75] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the Haskell 98 module system. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 17–28, New York, NY, USA, 2002. ACM.

[76] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.

[77] Gabriel C. Ditu. *The programming language TransLucid*. PhD thesis, Computer Science & Engineering, Faculty of Engineering, UNSW, 2007.

[78] Gabriel C. Ditu. *Creating TransLucid: A New Intensional Programming Language*. VDM, 2009.

[79] Sébastien Doeraene and Peter Van Roy. A new concurrency model for Scala based on a declarative dataflow core. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 4:1–4:10, New York, NY, USA, 2013. ACM.

[80] Weichang Du and William W. Wadge. An intensional language as the basis of a 3-D spreadsheet design. In *Computer Languages, 1988. Proceedings., International Conference on*, pages 2–9, October 1988.

[81] Paul R. Eggert and D. Stott Parker. An intensional file system. In *Proceedings of the Usenix File Systems Workshop*, pages 145–146, May 1992.

[82] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[83] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[84] Esterel Technologies. SCADE Success Stories, December 2013.

[85] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Task superscalar: An out-of-order task pipeline. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 89–100, December 2010.

[86] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer Berlin Heidelberg, 1987.

[87] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116, June 2011.

[88] Anthony A. Faustini and William W. Wadge. An eductive interpreter for the language pLucid. *SIGPLAN Notices*, 22(7):86–91, July 1987.

[89] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, New York, NY, USA, 1995. ACM.

[90] Paulo Ferreira, João Canas Ferreira, and José Carlos Alves. Computational block templates using functional programming models. *Actas IX Jornadas sobre Sistemas Reconfiguráveis*, 2013.

[91] Cormac Flanagan and Rishiyur S. Nikhil. pHluid: The design of a parallel functional language implementation on workstations. In *Proceedings of the first ACM international conference on functional programming*, pages 169–179. ACM Press, 1996.

[92] Michael J. Flynn, Oliver Pell, and Oskar Mencer. Dataflow supercomputing. In *Field Programmable Logic and Applications (FPL), 22nd International Conference on*, pages 1–3, August 2012.

[93] Free Software Foundation. Documentation for binutils 2.24. `https://sourceware.org/binutils/docs/`, 2014. Accessed: 09-06-2014.

[94] Georgios Fourtounis, Peter Ölveczky, and Nikolaos Papaspyrou. Formally specifying and analyzing a parallel virtual machine for lazy functional languages using Maude. In *Proceedings of the 5th International Workshop on High-level Parallel Programming and Applications*, HLPP' 11, 2011.

[95] Georgios Fourtounis and Nikolaos Papaspyrou. An efficient representation for lazy constructors using 64-bit pointers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, New York, NY, USA, 2014. ACM. To appear.

[96] Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The intensional transformation for functional languages with user-defined data types. In *Proceedings of the 8th Panhellenic Logic Symposium*, 2011.

[97] Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The generalized intensional transformation for implementing lazy functional languages. In Konstantinos F. Sagonas, editor, *PADL*, volume 7752 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2013.

[98] Georgios Fourtounis, Nikolaos Papaspyrou, and Panagiotis Theofilopoulos. Modular polymorphic defunctionalization. *Computer Science and Information Systems*. To appear.

[99] Georgios Fourtounis and Nikolaos S. Papaspyrou. Supporting separate compilation in a defunctionalizing compiler. In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 39–49, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[100] Simon Frankau. *Hardware Synthesis from a Stream-Processing Functional Language*. PhD thesis, University of Cambridge, 2004.

[101] Daniel P. Friedman and David S. Wise. CONS should not evaluate its arguments. In *ICALP*, pages 257–284, 1976.

[102] Jean-Raymond Gagné, Joey Paquet, and John Plaice. Indexical translation of tail-recursive functions. In Manolis Gergatsoulis, editor, *Intensional programming II. Papers from the 12th international symposium on languages for intensional programming (ISLIP'99)*, pages 296–309. World Scientific, 2000.

[103] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, August 2009.

[104] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the TRIPS computer system. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 1–12, New York, NY, USA, 2009. ACM.

[105] Edward F. Gehringer and J. Leslie Keedy. Tagged architecture: How compelling are its advantages? In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, ISCA '85, pages 162–170, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[106] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of Synthesis IV: Compiling affine recursion into static hardware. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 221–233, New York, NY, USA, 2011. ACM.

[107] Sukumar Ghosh, Somprakash Bandyopadhyay, Chandan Mazumdar, and S. Bhattacharya. Handling of recursion in dataflow model. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge*, ACM '84, pages 189–196, New York, NY, USA, 1984. ACM.

[108] Wolfgang K. Giloi. Konrad Zuse's Plankalkül: The first high-level, "non Von Neumann" programming language. *IEEE Annals of the History of Computing*, 19(2):17–24, April 1997.

[109] Christophe Giraud-Carrier. A reconfigurable dataflow machine for implementing functional programming languages. *SIGPLAN Notices*, 29(9):22–28, September 1994.

[110] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[111] Eiichi Goto, T. Soma, Nobuyuki Inada, Tetsuo Ida, Masanori Idesawa, Kei Hiraki, Masayuki Suzuki, Kentaro Shimizu, and B. Philipov. Design of a Lisp Machine – FLATS. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, LFP '82, pages 208–215, New York, NY, USA, 1982. ACM.

[112] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, Sept 2012.

[113] Athanassios Grivas. Implementation of functional languages using the branching dimensions transformation. Diploma dissertation, School of Electrical and Computer Engineering, National Technical University of Athens, October 2004.

[114] Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. Functions are data too (defunctionalization for PL/SQL). *PVLDB*, 6(12):1214–1217, 2013.

[115] David Gudeman. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, October 1993.

[116] John R. Gurd, Chris C. Kirkham, and Ian Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, pages 34–52, 1985.

[117] Kevin Hammond. The Spineless Tagless G-Machine - NOT!, 1993.

[118] Bin Han, Serguei A. Mokhov, and Joey Paquet. Advances in the design and implementation of a multi-tier architecture in the GIPSY environment with Java. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:259–266, 2010.

[119] William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, November 2005.

[120] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 642–649, 2001.

[121] Steven K. Heller. Efficient lazy data-structures on a dataflow machine. Technical Report MIT/LCS/TR-438, MIT, February 1989.

[122] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, POPL '76, pages 95–103, New York, NY, USA, 1976. ACM.

[123] Christoph Armin Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger, and Christian Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, May 1999.

[124] Tony Hoare and John Wickerson. Unifying models of data flow. In Manfred Broy, Christian Leuxner, and Tony Hoare, editors, *Software and Systems Safety - Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 211–230. IOS Press, 2011.

[125] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[126] R. J. M. Hughes. Super-combinators—a new implementation method for applicative languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[127] Ali R. Hurson and Krishna M. Kavi. Dataflow computers: Their history and future. *Wiley Encyclopedia of Computer Science and Engineering*, 2007.

[128] Robert A. Iannucci. Compiling for the hybrid architecture. In *Parallel Machines: Parallel Machine Languages*, volume 96 of *The Kluwer International Series in Engineering and Computer Science*, pages 93–147. Springer US, 1990.

[129] Peter Z. Ingerman. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, January 1961.

[130] Intel Corporation. Intel® 64 and IA-32 architectures software developer manuals, February 2014.

[131] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[132] Rangaswamy Jagannathan, Chris Dodd, and Iskender Agi. GLU: A high-level system for granular data-parallel programming. *Practice and Experience*, 9:1–63, 1996.

[133] Sateh M. Jalaleddine. Associative memories and processors: The exact match paradigm. *Journal of King Saud University - Computer and Information Sciences*, 11(0):45–67, 1999.

[134] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Berlin Heidelberg, 1985.

[135] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11:55–94, 1 2001.

[136] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall/CRC, 1st edition, August 2011.

[137] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine – version 2.5. *Journal of Functional Programming*, 2:127–202, 1992.

[138] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. `http://haskell.org/`, September 2002.

[139] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe*, volume 365 of *Lecture Notes in Computer Science*, pages 193–206. Springer Berlin Heidelberg, 1989.

[140] Georgios Karachalias. Pattern matching exhaustiveness for GADTs. Technical Report CSD-SW-TR-1-14, School of Electrical and Computer Engineering, National Technical University of Athens, 2014.

[141] Hojin Kee, Chung-Ching Shen, Shuvra S. Bhattacharyya, Ian Wong, Yong Rao, and Jacob Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Journal of Signal Processing Systems*, 66(3):285–301, 2012.

[142] Oliver Kennedy, Yanif Ahmad, and Christoph Koch. DBToaster: Agile views for a dynamic data management system. In *CIDR*, pages 284–295. www.cidrdb.org, 2011.

[143] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[144] Changkyu Kim, Simha Sethumadhavan, Madhu S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 381–394, Washington, DC, USA, 2007. IEEE Computer Society.

[145] Laszlo B. Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3–4):144–149, 2002.

[146] Donald E. Knuth. The remaining trouble spots in ALGOL 60. *Communications of the ACM*, 10(10):611–618, October 1967.

[147] Teuvo Kohonen. *Content-Addressable Memories*. Springer, Berlin; New York, second edition, 1987.

[148] Jean-Louis Krivine. Un interpréteur du lambda-calcul.

[149] Peter Kropf and John Plaice. Intensional objects, 1999.

[150] Lindsey Kuper and Ryan R. Newton. LVars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 71–84, New York, NY, USA, 2013. ACM.

[151] Koen Langendoen and Pieter H. Hartel. FCG: A code generator for lazy functional languages. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 278–296. Springer Berlin Heidelberg, 1992.

[152] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.

[153] Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 Workshop on Memory System Performance*, MSP '05, pages 24–35, New York, NY, USA, 2005. ACM.

[154] Ben Lee and Ali R. Hurson. Issues in dataflow computing. *Advances in Computers*, 37:285–333, 1993.

[155] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, may 1995.

[156] Gareth Lee and John Morris. Dataflow Java: Implicitly parallel Java. In *Proceedings of the 5th Australasian Computer Architecture Conference*, ACAC '00, pages 42–50, Washington, DC, USA, 2000. IEEE Computer Society.

[157] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Evaluation of a high performance code compression method. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 32, pages 93–102, Washington, DC, USA, 1999. IEEE Computer Society.

[158] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 109–122, New York, NY, USA, 1994. ACM.

[159] ARM Limited. Principles of ARM memory maps. `http://infocenter.arm.com/help/topic/com.arm.doc.den0001c/DEN0001C_principles_of_arm_memory_maps.pdf`, 2012. Accessed: 29-01-2014.

[160] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[161] Hai Liu, Neal Glew, Leaf Petersen, and Todd A. Anderson. The Intel Labs Haskell Research Compiler. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 105–116, New York, NY, USA, 2013. ACM.

[162] Wayne Luk, David Ferguson, and Ian Page. Structured hardware compilation of parallel programs. In *Selected Papers from the Oxford 1993 International Workshop on Field Programmable Logic and Applications on More FPGAs*, pages 213–224, Oxford, UK, UK, 1994. Abingdon EE&CS Books.

[163] Frederik M. Madsen and Andrzej Filinski. Towards a streaming model for nested data parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 13–24, New York, NY, USA, 2013. ACM.

[164] Simon Marlow. Why can't I get a stack trace? `http://www.youtube.com/watch?v=J0c4L-AURDQ`. Presentation at the Haskell Implementors Workshop 2012.

[165] Simon Marlow. Update avoidance analysis by abstract interpretation. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1993. Springer-Verlag.

[166] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.

[167] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[168] Simon Marlow, Alexey Rodriguez Yakushev, and Simon L. Peyton Jones. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 277–288, New York, NY, USA, 2007. ACM.

[169] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. AMD64 ABI draft 0.99.5, September 3, 2010.

[170] John McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[171] J. Meacham. JHC: John's Haskell compiler, 2007.

[172] Pierre Michaud and André Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 27–36, Washington, DC, USA, 2001. IEEE Computer Society.

[173] Microsoft. Lazy<T> Class, December 2013.

[174] Arie Middelkoop. Uniqueness typing refined. Master's thesis, Universiteit Utrecht, the Netherlands, 2006.

[175] Cupertino Miranda, Antoniu Pop, Philippe Dumont, Albert Cohen, and Marc Duranton. Erbium: A deterministic, concurrent intermediate representation to map data-flow tasks to scalable, persistent streaming processes. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 11–20, New York, NY, USA, 2010. ACM.

[176] Neil Mitchell and Colin Runciman. Losing functions without gaining data. In *Haskell '09: Proceedings of the second ACM SIGPLAN symposium on Haskell*, pages 49–60. ACM, September 2009.

[177] Serguei A. Mokhov. *Intensional Cyberforensics*. PhD thesis, Concordia University, Quebec, Canada, 2013.

[178] Serguei A. Mokhov and Joey Paquet. Formally specifying and proving operational aspects of Forensic Lucid in Isabelle. *CoRR*, abs/0904.3789, 2009.

[179] Serguei A. Mokhov and Joey Paquet. Using the General Intensional Programming System (GIPSY) for evaluation of Higher-Order Intensional Logic (HOIL) expressions. In *Proceedings of the 2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications*, SERA '10, pages 101–109, Washington, DC, USA, 2010. IEEE Computer Society.

[180] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 81–91, New York, NY, USA, 2001. ACM.

[181] Richard Montague. Pragmatics and intensional logic. In Richmond H. Thomason, editor, *Formal Philosophy: Selected Papers of Richard Montague*, pages 119–148. Yale University Press, New Haven, London, 1974.

[182] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[183] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language: Objects and functions for data analysis. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 104–131, Berlin, Heidelberg, 2012. Springer-Verlag.

[184] Mruby developers. Full-length nan boxing on 64bit platforms, #1441. `https://github.com/mruby/mruby/pull/1441`, 2013. Accessed: 31-01-2014.

[185] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[186] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*, pages 269–281, London, UK, UK, 1980. Springer-Verlag.

[187] Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin Heidelberg, 2000.

[188] Ernest Nagel and James R. Newman. *Gödel's proof*. Routledge & Kegan Paul London, 1959.

[189] Matthew Naylor and Colin Runciman. The Reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, 22(4-5):574–613, August 2012.

[190] George C. Nelan. *Firstification*. PhD thesis, Department of Computer Science, Arizona State University, U.S.A., 1991.

[191] Nicholas Nethercote and Alan Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of the 2002 Workshop on Memory System Performance*, MSP '02, pages 44–55, New York, NY, USA, 2002. ACM.

[192] Anja Niedermeier, Rinse Wester, Kenneth Rovers, Christiaan Baaij, Jan Kuper, and Gerard Smit. Designing a dataflow processor using CλaSH. In *Proceedings of the 28th Norchip Conference, NORCHIP 2010*, page 69. IEEE Circuits & Systems Society, November 2010.

[193] Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical report, BRICS, 2000.

[194] Rishiyur S. Nikhil. The parallel programming language Id and its compilation for parallel machines. *International Journal of High Speed Computing*, 5(2):171–223, 1993.

[195] Rishiyur S. Nikhil. Types, functional programming and atomic transactions in hardware design. In Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation*, volume 8000 of *Lecture Notes in Computer Science*, pages 418–431. Springer Berlin Heidelberg, 2013.

[196] Rishiyur S. Nikhil and Arvind. *Implicit parallel programming in pH*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[197] Susumu Nishimura. Deforesting in accumulating parameters via type-directed transformations. In *Asian Workshop on Programming Languages and Systems*, pages 145–159, 2002.

[198] Martin Odersky. The Scala language specification version 2.8, December 2013.

[199] Oracle. SPARC Assembly Language Reference Manual, 5.2: Address Sizes. `http://docs.oracle.com/cd/E26502_01/html/E28387/glyfs.html`. Accessed: 28-01-2014.

[200] Oracle. Compressedoops. `https://wikis.oracle.com/display/HotSpotInternals/CompressedOops`, 2012. Accessed: 14-05-2014.

[201] Mehmet A. Orgun and William W. Wadge. Towards a unified theory of intensional logic programming. *The Journal of Logic Programming*, 13(4):413–440, 1992.

[202] Gerald Ostheimer. *Parallel Functional Programming for Message-Passing Multiprocessors*. PhD thesis, University of St. Andrews, Scotland, 1993.

[203] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: a tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712–727, March 2006.

[204] Mike Pall. LuaJIT 2.0 intellectual property disclosure and research opportunities. `http://lua-users.org/lists/lua-l/2009-11/msg00089.html`, 2009. Accessed: 31-01-2014.

[205] Gregory M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor*. MIT Press, Cambridge, MA, USA, 1991.

[206] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 82–91, May 1990.

[207] Nikolaos Papaspyrou and Konstantinos Sagonas. On preserving term sharing in the Erlang virtual machine. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 11–20, New York, NY, USA, 2012. ACM.

[208] Nikolaos S. Papaspyrou and Ioannis T. Kassios. GLU embedded in C++: a marriage between multidimensional and object-oriented programming. *Software: Practice and Experience*, 34:609–630, June 2004.

[209] Joey Paquet and Peter Kropf. The GIPSY architecture. In Peter G. Kropf, Gilbert Babin, John Plaice, and Herwig Unger, editors, *Distributed Communities on the Web*, volume 1830 of *Lecture Notes in Computer Science*, pages 144–153. Springer Berlin Heidelberg, 2000.

[210] Joey Paquet and Serguei A. Mokhov. Furthering baseline core Lucid standard specification in the context of the history of Lucid, intensional programming, and context-aware computing. *CoRR*, abs/1107.0940, 2011.

[211] Sungwoo Park and Hyeonseung Im. A calculus for hardware description. *Journal of Functional Programming*, 21(1):21–58, January 2011.

[212] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[213] Will Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, 1993.

[214] Oliver Pell, Jacob Bower, Robert Dimond, Oskar Mencer, and Michael J. Flynn. Finite-difference wave propagation modeling on special-purpose dataflow machines. *Parallel and Distributed Systems, IEEE Transactions on*, 24(5):906–915, May 2013.

[215] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM.

[216] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[217] Simon L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 18–44, London, UK, UK, 1996. Springer-Verlag.

[218] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: a tutorial*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[219] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C--: A portable assembly language that supports garbage collection. In *Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, PPDP '99, pages 1–28, London, UK, UK, 1999. Springer-Verlag.

[220] Keshav Pingali and Arvind. Efficient demand-driven evaluation. Part 1. *ACM Transactions on Programming Languages and Systems*, 7(2):311–333, April 1985.

[221] Keshav K. Pingali. Lazy evaluation and the logic variable. In *Proceedings of the 2nd international conference on Supercomputing*, ICS '88, pages 560–572, New York, NY, USA, 1988. ACM.

[222] John Plaice. *Cartesian Programming*. PhD thesis, University of Grenoble, France, January 2011. Habilitation thesis.

[223] John Plaice and Jarryd P. Beck. Higher-order multidimensional programming. Technical Report UNSW-CSE-TR-201215, The University of New South Wales, August 2012.

[224] John Plaice and Blanca Mancilla. Collaborative intensional hypertext. In *Hypertext*, pages 91–92, 2004.

[225] John Plaice and Blanca Mancilla. The practical uses of TransLucid. In *Proceedings of the first international workshop on Context-aware software technology and applications*, CASTA '09, pages 13–16, New York, NY, USA, 2009. ACM.

[226] John Plaice, Blanca Mancilla, and Gabriel Ditu. From Lucid to TransLucid: Iteration, dataflow, intensional and cartesian programming. *Mathematics in Computer Science*, 2(1):37–61, 2008.

[227] François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, 2006. 10.1007/s10990-006-8611-7.

[228] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[229] Toby Rahilly and John Plaice. A multithreaded implementation for TransLucid. In *COMPSAC*, pages 1272–1277. IEEE Computer Society, 2008.

[230] Naila Rahman. Algorithms for hardware caches and TLB. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 171–192. Springer Berlin Heidelberg, 2003.

[231] Norman Ramsey, Simon Peyton Jones, and Christian Lindig. *The C-- Language Specification Version 2.0*, January 2005.

[232] Theophile Ranquet. OCaml internals - implementation of an ML descendant. `http://www.gconfs.fr/media/confs/2013-11-ml-slides.pdf`, 2013. Accessed: 01-02-2014.

[233] Swapan Kumar Ray. Large-capacity high-throughput low-cost pipelined CAM using pipelined CTAM. *IEEE Transactions on Computers*, 55(5):575–587, May 2006.

[234] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Reprinted from the proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[235] Niklas Röjemo. Highlights from `nhc` — a space-efficient Haskell compiler. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 282–292, New York, NY, USA, 1995. ACM.

[236] Panos Rondogiannis. *Higher-Order Tagged Dataflow*. PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994.

[237] Panos Rondogiannis. Adding multidimensionality to procedural programming languages. *Software: Practice and Experience*, 29(13):1201–1221, 1999.

[238] Panos Rondogiannis and Manolis Gergatsoulis. The branching-time transformation technique for chain Datalog programs. *Journal of Intelligent Information Systems*, 17(1):71–94, 2001.

[239] Panos Rondogiannis, Manolis Gergatsoulis, and Themis Panayiotopoulos. Branching-time logic programming: The language Cactus and its applications. *Computer Languages*, 24(3):155–178, October 1998.

[240] Panos Rondogiannis and William W. Wadge. A dataflow implementation technique for lazy typed functional languages. In *Proceedings of the International Symposium on Lucid and Intensional Programming*, pages 23–43, April 1993.

[241] Panos Rondogiannis and William W. Wadge. Compiling higher-order functions for tagged-dataflow. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '94, pages 269–278, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[242] Panos Rondogiannis and William W. Wadge. Higher-order dataflow and its implementation on stock hardware. In *Proceedings of the 1994 ACM symposium on applied computing*, SAC '94, pages 431–435, New York, NY, USA, 1994. ACM.

[243] Panos Rondogiannis and William W. Wadge. Extending the intensionalization algorithm to a broader class of higher-order programs. In *Eighth International Symposium on Languages for Intensional Programming*, 1995.

[244] Panos Rondogiannis and William W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7:73–101, January 1997.

[245] Panos Rondogiannis and William W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, 1999.

[246] Ghislain Roquier, Endri Bezati, and Marco Mattavelli. Hardware and software synthesis of heterogeneous systems from dataflow programs. *Journal of Electrical and Computer Engineering*, January 2012.

[247] Xavier Saint-Mleux, Marc Feeley, and Jean-Pierre David. SHard: a Scheme to hardware compiler. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, 2006.

[248] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *Micro, IEEE*, 23(6):46–51, November 2003.

[249] Patrick M. Sansom. Time profiling a lazy functional compiler. In John T. O'Donnell and Kevin Hammond, editors, *Functional Programming, Glasgow 1993*, Workshops in Computing, pages 252–264. Springer London, 1994.

[250] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs require? In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 216–225, New York, NY, USA, 1995. ACM.

[251] Mark Scheevel. NORMA: A graph reduction processor. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 212–219, New York, NY, USA, 1986. ACM.

[252] Thomas Schilling. Challenges for a trace-based just-in-time compiler for Haskell. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 51–68. Springer Berlin Heidelberg, 2012.

[253] Ulrich Schöpp. On interaction, continuations and defunctionalization. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, volume 7941 of *Lecture Notes in Computer Science*, pages 205–220. Springer Berlin Heidelberg, 2013.

[254] Tom Schrijvers and Alan Mycroft. Strictness meets data flow. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 439–454, Berlin, Heidelberg, 2010. Springer-Verlag.

[255] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, New York, NY, USA, 2009. ACM.

[256] Stefan Schulze and Sergei Sawitzki. Processor design using a functional hardware description language. *Microprocessors and Microsystems*, 36(8):676–694, 2012. Applied Reconfigurable Computing.

[257] Jocelyn Sérot and Greg Michaelson. Harnessing parallelism in FPGAs using the Hume language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 27–36, New York, NY, USA, 2012. ACM.

[258] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, July 2005. `http://www.jucs.org/jucs_11_7/hardware_design_and_functional`.

[259] Jorge Luiz E. Silva and Joelmir José Lopes. A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores. *WSEAS Transactions on Computers*, 9(5):429–444, May 2010.

[260] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make: A Program for Directing Recompilation, for Version 3.81*. Free Software Foundation, 2004.

[261] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Design of a LISP-based microprocessor. *Communications of the ACM*, 23(11):628–645, November 1980.

[262] Kenneth M. Steele. Implementation of an I-structure memory controller. Technical Report MIT / LCS / TR-471, MIT, March 1990.

[263] William Robert Stoye. The implementation of functional languages using custom hardware. Technical Report UCAM-CL-TR-81, University of Cambridge, 1985.

[264] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[265] Martin Sulzmann and Meng Wang. GADTless programming in Haskell 98, 2006.

[266] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[267] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–302, Washington, DC, USA, 2003. IEEE Computer Society.

[268] David Swasey, Tom Murphy VII, Karl Crary, and Robert Harper. A separate compilation extension to Standard ML. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 32–42, New York, NY, USA, 2006. ACM.

[269] Paul Swoboda and John Plaice. An active functional intensional database. In Fernando Galindo, Makoto Takizawa, and Roland Traunmüller, editors, *Database and Expert Systems Applications*, volume 3180 of *Lecture Notes in Computer Science*, pages 56–65. Springer Berlin Heidelberg, 2004.

[270] Lei Tao. Intensional value warehouse and garbage collection in the GIPSY. Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada, 2004.

[271] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *Micro, IEEE*, 22(2):25–35, March 2002.

[272] John Teifel and Rajit Manohar. An asynchronous dataflow FPGA architecture. *Computers, IEEE Transactions on*, 53(11):1376–1392, November 2004.

[273] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, stdint.h – integer types. `http://pubs.opengroup.org/onlinepubs/000095399/basedefs/stdint.h.html`, 2004. Accessed: 28-01-2014.

[274] Panagiotis Theofilopoulos. An efficient implementation of lazy functional programming languages based on the generalized intensional transformation. Master's thesis, National and Kapodistrian University of Athens, Athens, Greece, 2014.

[275] Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, 1997.

[276] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programmming*, 8(4):367–412, July 1998.

[277] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1986.

[278] David A. Turner. A new implementation technique for applicative languages. In *Software - Practice and Experience*, volume 9, pages 31–49, 1979.

[279] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

[280] Rene van Leuken, Tom van Leeuwen, and Huib Lincklaen Arriens. High level synthesis of asynchronous circuits from data flow graphs. In *Proceedings of the 21st International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization, and Simulation*, PATMOS'11, pages 317–330, Berlin, Heidelberg, 2011. Springer-Verlag.

[281] Emil Vassev and Joey Paquet. A general architecture for demand migration in a demand-driven execution engine in a heterogeneous and distributed environment. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference, 2005*, pages 176–182, May 2005.

[282] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, December 1986.

[283] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Object-relative addressing: Compressed pointers in 64-bit Java virtual machines. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 79–100. Springer, 2007.

[284] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(3), September 2007.

[285] Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[286] William W. Wadge. New frontiers in intensional programming. In Janice I. Glasgow, editor, *Proceedings of the 3rd International Symposium on Lucid and Intensional Programming*, Kingston, Ontario, 1990. Queen's University. Held 13–15 May 1990.

[287] William W. Wadge. Higher-order Lucid. In *Proceedings of the 4th International Symposium on Lucid and Intensional Programming*, 1991.

[288] William W. Wadge. Intensional logic in context. *Intensional Programming II: Papers based on the 1999 International Symposium on Lucid and Intensional Programming*, 2000.

[289] William W. Wadge and Edward A. Aschroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[290] Wadge, Bill. Lucid - retirement plan, April 2012.

[291] Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 189–201, New York, NY, USA, 2003. ACM.

[292] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.

[293] Kaiyu Wan, Vasu Alagar, and Joey Paquet. A context theory for intensional programming. In *Proceedings of the CRR'05 Workshop on Context Representation and Reasoning*, 2005.

[294] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 166–178, New York, NY, USA, 2001. ACM.

[295] David H. D. Warren. Higher-order extensions to PROLOG: are they needed? In Donald Michie, editor, *Machine Intelligence*, volume 10, pages 441–454. Edinburgh University Press, 1982.

[296] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant. Flagship: A parallel architecture for declarative programming. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 124–130, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[297] Stephen Weeks. Whole-program compilation in MLton. In *Proceedings of the ML Workshop*, pages 1–1, New York, NY, USA, 2006. ACM. Invited lecture, available from `http://www.mlton.org/guide/20130715/References.attachments/060916-mlton.pdf` (accessed: Mar. 2014).

[298] Steven J.E. Wilton, Cristopher W. Jones, and Julien Lamoureux. An embedded flexible content-addressable memory core for inclusion in a field-programmable gate array. In *Circuits and Systems, 2004. ISCAS '04. Proceedings of the 2004 International Symposium on*, volume 2, pages II–885–8 Vol.2, May 2004.

[299] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[300] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, NY, USA, 2003. ACM.

[301] Baltasar Trancón y Widemann and Markus Lepper. Foundations of total functional data-flow programming. In Neelakantan Krishnaswami and Paul Blain Levy, editors, *Mathematically Structured Functional Programming*, EPTCS. 2014.

[302] Ali A. G. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

[303] Fahimeh Yazdanpanah, Carlos Alvarez-Martinez, Daniel Jimenez-Gonzalez, and Yoav Etsion. Hybrid dataflow/von-Neumann architectures. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints):1, 2013.

[304] Ali Mustafa Zaidi and David J. Greaves. Achieving superscalar performance without superscalar overheads - a dataflow compiler IR for custom computing. In Andrew V. Jones and Nicholas Ng, editors, *ICCSW*, volume 35 of *OASICS*, pages 136–143. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2013.

[305] Noam Zeilberger. Defunctionalizing focusing proofs (or, how Twelf learned to stop worrying and love the Ω-rule). In *International Workshop on Proof-Search in Type Theories (PSTT'09)*, 2009.

[306] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2002.

[307] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, 2001.