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

# Υλοποίηση της Γλώσσας Προγραμματισμού Ασυρμάτων Πρωτοκόλλων Ziria ως DSL Ενσωματωμένη στη Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΕΙΡΗΝΗ ΒΛΑΣΣΗ-ΠΑΝΔΗ**

**Επιβλέπων :** Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2015

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

# Υλοποίηση της Γλώσσας Προγραμματισμού Ασυρμάτων Πρωτοκόλλων Ziria ως DSL Ενσωματωμένη στη Haskell

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΕΙΡΗΝΗ ΒΛΑΣΣΗ-ΠΑΝΔΗ**

**Επιβλέπων :** Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 4η Σεπτεμβρίου 2015.

| | | |
|---|---|---|
| Νικόλαος Σ. Παπασπύρου<br>Αν. Καθηγητής Ε.Μ.Π. | Κωνσταντίνος Σαγώνας<br>Αν. Καθηγητής Ε.Μ.Π. | Δημήτριος Βυτινιώτης<br>Ερευνητής Microsoft Research |

Αθήνα, Σεπτέμβριος 2015

**Ειρήνη Βλάσση-Πανδή**

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

# Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει έναν τρόπο ενσωμάτωσης της γλώσσας προγραμματισμού Ziria στη Haskell. Η Ziria είναι μια νέα, καινοτόμος γλώσσα συγκεκριμένου σκοπού φτιαγμένη για ραδιοεπικοινωνία ορισμένη από λογισμικό.

Αναλυτικότερα, η Ziria είναι μια καινούρια πλατφόρμα που προσφέρει προγραμματιστικές αφαιρέσεις κατάλληλες για ασύρματο προγραμματισμό σε φυσικό επίπεδο. Δίνοντας παράλληλα έμφαση στον επαναπροσδιορισμό της σωλήνωσης (pipeline reconfiguration) και προσφέροντας μια πληθώρα βελτιστοποιήσεων κατά τη μεταγλώττιση, επιτυγχάνει να είναι μια γλώσσα ευέλικτη στον προγραμματισμό και ταυτόχρονα πολύ καλή στην απόδοση, πετυχαίνοντας για παράδειγμα ρυθμούς γραμμής που απαιτούνται για την υλοποίηση του WiFi 802.11a/g σε απλούς επεξεργαστές.

Η Ziria είναι μια αυτόνομη γλώσσα συγκεκριμένου σκοπού με το δικό της συντακτικό και το δικό της μεταγλωττιστή. Ωστόσο κατά τον σχεδιασμό της έχει ήδη συμπεριλάβει πολλές ιδέες από τη γλώσσα γενικού σκοπού Haskell. Η παρούσα διπλωματική εργασία ακολουθεί μια εναλλακτική προσέγγιση και επιχειρεί να ενσωματώσει πλήρως τη Ziria, ως DSL στην Haskell.

Η ενσωμάτωση αφορά ουσιαστικά έναν διερμηνέα της Ziria, ο οποίος μας επιτρέπει να ορίσουμε τους combinators της Ziria στη Haskell, προσφέροντας έτσι έναν εναλλακτικό τρόπο να κάνουμε υπολογισμούς ροών δεδομένων μέσα στη Haskell. Η τελική υλοποίηση είναι σχετικά απλή, σχετικά άμεση ως προς τη σημασιολογία της Ziria, γεγονός που την καθιστά κατάλληλη για πειραματισμό και γρήγορη προτυποποίηση, δίνοντας τη δυνατότητα στον προγραμματιστή να εκμεταλλευτεί την πλήρη ισχύ της Haskell και να χρησιμοποιήσει χαρακτηριστικά που η ίδια η Ziria δεν υποστηρίζει.

Ελέγχουμε την ορθότητα και την πληρότητα της ενσωματωμένης γλώσσας μέσω της υλοποίησης της κρυπτογραφικής συνάρτησης SHA-1 τόσο σε Ziria όσο και σε Haskell αλλά και μέσω άλλων μικρότερων παραδειγμάτων.

## Λέξεις κλειδιά

Ziria, Haskell, ραδιοεπικοινωνία ορισμένη από λογισμικό, γλώσσα συγκεκριμένου σκοπού, ενσωμάτωση γλώσσας συγκεκριμένου σκοπού, free monad transformer, προγραμματισμός σε φυσικό επίπεδο, ροή δεδομένων εισόδου, ροή δεδομένων εξόδου

# Abstract

This diploma thesis presents a way of embedding the programming language Ziria into Haskell. Ziria is a new, innovative domain-specific language made for software-defined radio.

More precisely, Ziria is a new platform that offers programming abstractions suitable for wireless programming in physical layer. Emphasizing in the pipeline reconfiguration and offering a plethora of compile-time optimizations, it achieves flexibility without sacrificing efficiency. As an example, it achieves the line rates that are required for the implementation of WiFi 802.11a/g in commodity CPUs.

Ziria is an independent domain-specific language (DSL) with its own syntax and compiler. Nevertheless, its design has been heavily inspired by the general-purpose programming language Haskell. This diploma thesis follows an alternative approach by attempting to fully embed Ziria in Haskell as a DSL.

The embedding is essentially implemented as a Ziria interpreter, that allows us to define the Ziria combinators in Haskell, thus providing an alternative way of doing data stream computations in Haskell. The implementation is relatively simple and direct in regard to the actual Ziria semantics. Consequently, the embedding is suitable for experimentation and rapid prototyping, thus allowing the programmer to benefit from the full functionality of Haskell and to use features that Ziria itself does not support.

Lastly, we test the correctness and the completeness of the embedded language through the implementation of the cryptographic hash function SHA-1 in Ziria as well as in Haskell.

## Key words

Ziria, Haskell, software-defined radio (SDR), domain-specific language (DSL), embedded domain-specific language, free monad transformer, wireless physical (PHY) layer programming, stream computation

# Ευχαριστίες

Φτάνοντας στο τέλος των προπτυχιακών μου σπουδών θα ήθελα να ευχαριστήσω όλους αυτούς τους ανθρώπους που με βοήθησαν στη μέχρι τώρα πορεία μου και παρά τις όποιες εξωτερικές αντιξοότητες μου επέτρεψαν να συνεχίζω να μαθαίνω και να εξελίσσομαι.

Μέσα από αυτό το σύνολο οφείλω να ξεχωρίσω τον επιβλέποντα καθηγητή μου Νίκο Παπασπύρου. Ο οποίος από την πρώτη μέχρι την τελευταία μέρα στη σχολή αποτέλεσε για εμένα πηγή έμπνευσης πολλαπλά. Στα πρώτα χρόνια των σπουδών μου μετέδωσε εκτός από γνώσεις, την αγάπη και το μεράκι για τις γλώσσες προγραμματισμού. Ενώ κατόπιν μου έδωσε την ευκαιρία να ασχοληθώ με ένα πολύ ενδιαφέρον θέμα, αφού με εμπιστεύτηκε με την εκπόνηση της παρούσας διπλωματικής. Θέλω να τον ευχαριστήσω προσωπικά για την αμέριστη στήριξη σε όλα τα επίπεδα.

Μέσω αυτής της διπλωματικής είχα την επιπλέον τύχη και χαρά να συνεργαστώ με τον εξαίρετο επιστήμονα κ. Δήμητρη Βυτινιώτη, τον οποίο θέλω να ευχαριστήσω επίσης θερμά. Αρχικά, γιατί μου έδωσε τη δυνατότητα να παρακολουθήσω από σχετικά κοντά και ίσως να συμβάλλω ελάχιστα στην ανάπτυξη μιας νέας γλώσσας προγραμματισμού, γεγονός που αποτέλεσε συναρπαστική εμπειρία για εμένα. Και δευτερευόντως, αλλά εξίσου σημαντικό, για τον ενθουσιασμό και την όρεξη για έρευνα που προσφέρει σε όποιον βρίσκεται στο περιβάλλον του.

Σε πιο προσωπικό επίπεδο θα ήθελα να ευχαριστήσω τη μητέρα μου, Κατερίνα, για όλα όσα έχει κάνει για εμένα και την ανατροφή μου αλλά και τους πολύ καλούς φίλους και συμφοιτητές Λυδία και Γιάννη που υπήρξαν εκλεκτοί συνοδοιπόροι εντός κι εκτός σχολής.

Ειρήνη Βλάσση-Πανδή,

Αθήνα, 4η Σεπτεμβρίου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-15, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Σεπτέμβριος 2015.

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

# Περιεχόμενα

# Κατάλογος σχημάτων

# Κατάλογος Κώδικα

# Κεφάλαιο 1

# Εισαγωγή

## 1.1 Αντικείμενο της Εργασίας

Η ραδιοεπικοινωνία ορισμένη από λογισμικό 2.1 έχει τη δυναμική να επιτρέψει σημαντικές καινοτομίες στη σχεδίαση ασύρματων δικτύων. Ωστόσο, το όποιο πιθανό μέχρι τώρα αντίκτυπο έχει περιοριστεί δραστικά λόγω της αυξημένης πολυπλοκότητας των εργαλείων προγραμματισμού που χρησιμοποιούν οι υπάρχουσες πλατφόρμες. Η Ziria σχεδιάστηκε για να αντιμετωπίσει το πρόβλημα αυτό. Είναι μια νέα γλώσσα προγραμματισμού με το δικό της συντακτικό και το δικό της μεταγλωττιστή. Η νέα αυτή πλατφόρμα προσφέρει τις κατάλληλες προγραμματιστικές αφαιρέσεις αλλά και -μέσω πολλαπλών βελτιστοποιήσεων κατά τη μεταγλώττιση- την απαραίτητη ταχύτητα για ασύρματο προγραμματισμό σε φυσικό επίπεδο.

Η Ziria είναι μια αυτόνομη γλώσσα συγκεκριμένου σκοπού 3.1, με ωστόσο σαφείς επιρροές κατά το σχεδιασμό από τη γλώσσα γενικού σκοπού Haskell. Στόχος της παρούσας εργασίας είναι η πλήρης ενσωμάτωση της Ziria ως γλώσσα συγκεκριμένου σκοπού (domain-specific language, DSL) στη Haskell. Τα κίνητρα για να κινηθούμε προς την κατεύθυνση αυτή είναι ότι θα καταφέρουμε να έχουμε μία υλοποίηση (i) σχετικά απλή, (ii) σχετικά άμεση ως προς τη σημασιολογία της Ziria και άρα κατάλληλη για πειραματισμό, (iii) που δίνει τη δυνατότητα να εκμεταλλευτεί ο προγραμματιστής όλη την υπόλοιπη Haskell και να χρησιμοποιήσει δυνατότητες και χαρακτηριστικά που η ίδια η Ziria δεν υποστηρίζει.

## 1.2 Δομή της Διπλωματικής Εργασίας

Το υπόλοιπο της εργασίας διαμορφώνεται ως εξής:

**Κεφάλαιο 2:**
Παρέχουμε το απαραίτητο θεωρητικό υπόβαθρο για τις έννοιες και τις οντότητες που συζητάμε μέσω της εργασίας.

**Κεφάλαιο 3:**
Δίνουμε τον ορισμό των γλωσσών συγκεκριμένου σκοπού και τις υπάρχουσες κατηγορίες και υποκατηγορίες. Συζητάμε επίσης γιατί η Haskell μπορεί να αποτελεί μια καλή επιλογή ως host γλώσσα για ενσωμάτωση γλωσσών συγκεκριμένου σκοπού και δίνουμε μερικά απλά παραδείγματα που υποστηρίζουν την πρόταση αυτή.

**Κεφάλαιο 4:**
Παρουσιάζουμε τη γλώσσα συγκεκριμένου σκοπού Ziria. Πιο αναλυτικά παραθέτουμε τη σύνταξή της, τις προγραμματιστικές αφαιρέσεις που έχει, τον τρόπο με τον οποίο κάνει σύνθεση στο μονοπάτι δεδομένων και στο μονοπάτι ελέγχου, τα μπλοκ επεξεργασίας που χρησιμοποιεί,

τον επαναπροσδιορισμό σωλήνωσης, τις δυνατές βελτιστοποιήσεις και μια συνολική αξιολόγηση.

**Κεφάλαιο 5:**
Σε αυτό το κεφάλαιο περιγράφουμε την τελική υλοποίηση της ενσωμάτωσης της Ziria στη Haskell, τα εργαλεία που χρησιμοποιήσαμε και τον τρόπο που το πετύχαμε. Παραθέτουμε επίσης μερικά παραδείγματα γραμμένα σε Ziria καθώς και τα αντίστοιχα σε Haskell για να συγκρίνουμε, να ελέγξουμε την ορθότητα και να έχουμε μια εποπτεία του συντακτικού.

**Κεφάλαιο 6:**
Παρουσιάζουμε τα συμπεράσματα που προέκυψαν από αυτήν την εργασία καθώς και μελλοντικές ερευνητικές κατευθύνσεις που μπορούν να ακολουθήσουν.

## Κεφάλαιο 2

# Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο, παρουσιάζονται οι βασικές θεωρητικές γνώσεις και μηχανισμοί που χρειάζονται για την κατανόηση της Ziria ως νέας γλώσσας προγραμματισμού αλλά και μερικά εισαγωγικά στοιχεία για τη Haskell που θα χρησιμοποιηθεί ως host γλώσσα για την ενσωμάτωση της Ziria.

Πιο συγκεκριμένα, στην ενότητα 2.1 παρουσιάζουμε πώς ορίζεται η ραδιοεπικοινωνία ορισμένη από λογισμικό και πώς αυτή έχει εξελιχθεί τα τελευταία χρόνια. Τέλος στην ενότητα 2.2 δίνουμε μερικές βασικές έννοιες της Haskell ώστε να επιτραπεί στον αναγνώστη με στοιχειώδες υπόβαθρο στο θέμα να παρακολουθήσει εύκολα την ροή της υλοποίησης.

## 2.1 Ραδιοεπικοινωνία Ορισμένη από Λογισμικό

Η ραδιοεπικοινωνία ορισμένη από λογισμικό (**Software Defined Radio, SDR**) αναφέρεται σε πρωτόκολλα ασύρματης επικοινωνίας στα οποία η περίπλοκη επεξεργασία σήματος που απαιτείται, πραγματοποιείται από λογισμικό αντί της τυπικής υλοποίησης με υλικό. Η χρήση λογισμικού μας επιτρέπει να έχουμε επαναδιαρθώσιμες λειτουργικές μονάδες προσφέροντας μία ευέλικτη, αποδοτική και ασφαλή πλατφόρμα ιδανική για έρευνα και ανάπτυξη νέων πρωτοκόλλων.

Η τεχνολογία SDR έχει ήδη συμβάλει στη σπουδαία πρόοδο που έχει σημειωθεί τα τελευταία χρόνια στον σχεδιασμό και την υλοποίηση ασυρμάτων πρωτοκόλλων στο φυσικό επίπεδο (PHY) [1, 2, 3]. Ωστόσο η πολυπλοκότητα του προβλήματος είναι αυξημένη. Ο απαιτούμενος ρυθμός μετάδοσης επιτάσσει χρήση ιδιαίτερα γρήγορων και αποδοτικών αλγορίθμων, γεγονός που για το λογισμικό μεταφράζεται σε απαίτηση μεγάλης επεξεργαστικής ισχύος της τάξεως των gigabits ανά δευτερόλεπτο. Συνεπώς, ο προγραμματιστής των SDR καλείται από τη μία πλευρά να αποκτήσει βαθειά κατανόηση στους τομείς της επεξεργασίας σήματος και της ασύρματης δικτύωσης και από την άλλη να είναι πλήρως καταρτισμένος σε θέματα αρχιτεκτονικής υπολογιστών και σχεδιασμού συστημάτων. Για αυτόν τον λόγο οι χρήστες των SDR περιορίζονται κυρίως σε λίγες ερευνητικές ομάδες χάνοντας ένα ευρύ και κρίσιμο ερευνητικό κοινό.

Παρακάτω παρουσιάζουμε μερικές από τις πιο δημοφιλείς πλατφόρμες SDR αλλά και τις εκάστοτε προκλήσεις που καλείται να αντιμετωπίσει ο χρήστης. Μια κατηγοριοποίηση αφορά τα προγραμματιστικά εργαλεία για SDR που βασίζονται σε έναν συνδυασμό Matlab, Simulink και κάποιου FPGA compiler, όπως η Warp [4] και η Lyrtech [5]. Τα κύρια πλεονεκτήματα αυτών των περιβαλλόντων είναι η υψηλή επίδοση αλλά και ότι μέσω του Matlab, παρέχουν στον προγραμματιστή χρήσιμες έτοιμες βιβλιοθήκες για επεξεργασία σήματος και προσομοιώσεις πραγματικού κόσμου. Εντούτοις το προγραμματιστικό μοντέλο απαιτεί από τον χρήστη ουσιαστικά να έχει την ικανότητα να προγραμματίζει αποτελεσματικά FPGAs, λαμβάνοντας κάθε φορά υπ'όψιν του τις διαφορετικές καθυστερήσεις διάδοσης και τον τρόπο που αυτές επιδρούν στην επίδοση κάθε συστήματος.

Μία εναλλακτική προσέγγιση είναι αυτή των πλατφορμών SDR που βασίζονται σε επεξεργαστές γενικού σκοπού. Σε αυτήν την περίπτωση συνήθως έχουμε εύκολα προγραμματίσιμα κι επεκτάσιμα

εργαλεία σε βάρος όμως μιας μειούμενης επίδοσης, η οποία καθιστά απαγορευτική τη χρήση τέτοιων εργαλείων για προσομοίωση πραγματικών δικτύων. Στην πραγματικότητα, οι περισσότερες τέτοιες προσεγγίσεις δεν επιχειρούν καν να επεξεργαστούν τα σήματα σε πραγματικό χρόνο. Μέσω της πλατφόρμας SDR κάνουν μία απλή καταγραφή του σήματος, ενώ το επεξεργάζονται κατόπιν, εκτός σύνδεσης, με πολύ μικρότερο ρυθμό από αυτόν που απαιτείται στο PHY. Το τελευταίο διάστημα, ωστόσο, κάποιες πλατφόρμες όπως η Sora [6], η USRP [7] και η GnuRadio [8], έχουν καταφέρει να επεξεργάζονται τα σήματα βάσης με ικανοποιητικό ρυθμό. Για παράδειγμα με τη Sora επιτεύχθηκε για πρώτη φορά η διαλειτουργικότητα μεταξύ εμπορικού WiFi υλικού ενώ ταυτόχρονα η επεξεργασία όλων των σημάτων γίνεται στην κεντρική μονάδα επεξεργασίας (CPU). Οι προσπάθειες αυτές όμως είχαν σαν συνέπεια να ανέβει σημαντικά η πολυπλοκότητα στο γράψιμο του κώδικα. Ο χρήστης αφού γράψει το πρόγραμμα σε C/C++ καλείται να το βελτιστοποιήσει με χειροκίνητο τρόπο μέχρι να επιτευχθεί η επιθυμητή ταχύτητα. Η χρήση C++ template βιβλιοθηκών επιστρατεύτηκε ώστε να υπάρξει ένας περισσότερο αυτοματοποιημένος τρόπος, προκύπτουν ωστόσο πολλοί περιορισμοί σε θέματα μοιράσματος κατάστασης και δυναμικού επανασχεδιασμού.

Μέχρι τώρα ο κύριος όγκος των προσπαθειών για προγραμματισμό SDR έχει επικεντρωθεί στη δημιουργία αποδοτικών υλοποιήσεων των αλγορίθμων ψηφιακής επεξεργασίας σήματος (digital signal processing, DSP) και των υποκείμενων υπολογισμών πινάκων. Ωστόσο τυπικά ο σχεδιασμός στο φυσικό επίπεδο αποτελείται από μια σωλήνωση (pipeline) των σταδίων επεξεργασίας σήματος κι είναι ουσιώδες να δοθεί βάρος, όχι μόνο στα διάφορα στάδια — DSP κώδικας — αλλά και στη δυνατότητα επαναπροσδιορισμού της σωλήνωσης όταν η κατάσταση του συστήματος (π.χ. πληροφορίες καναλιού, ποσοστά κωδικοποίησης και διαμόρφωσης) αλλάζει. Για αυτόν τον λόγο προκύπτει σαν ανάγκη για τον προγραμματιστή η διατήρηση μιας κατάστασης (state) καθ' όλη τη διαδικασία επεξεργασίας των διάφορων οντοτήτων (κεφαλίδα, πακέτα, κτλ.), η οποία θα του επιτρέψει να προσδιορίζει πότε μία κατάσταση αλλάζει και πώς αυτή η αλλαγή επηρεάζει τη λειτουργικότητα του συνολικού συστήματος. Το στοιχείο αυτό του PHY σχεδιασμού έχει εξέχουσα σημασία παρ'όλο που είχε παραβλεφθεί από τις υπάρχουσες δουλειές για SDR. Παραδείγματος χάριν, το πρωτόκολλο 4G/LTE, που αυτή τη στιγμή αποτελεί την τεχνολογία αιχμής για την ασύρματη κινητή τηλεφωνία, περιέχει πάνω από 400 σελίδες προσδιορισμού του PHY επιπέδου και μόνο [9], ενώ το πρωτόκολλο IEEE WiFi χρειάζεται 90 σελίδες κειμένου [10] για να προσδιορίσει πλήρως πώς οι αλλαγές κατάστασης επηρεάζουν την ροή ελέγχου. Μία σωστή και αποδοτική υλοποίηση των προηγουμένων προδιαγραφών δεν είναι ούτε εύκολη ούτε προφανής αφού απαιτείται ιδιαίτερα λεπτός συγχρονισμός μεταξύ του επαναπροσδιορισμού του pipeline και της επεξεργασίας δεδομένων.

Η Ziria [11, 12, 13, 14] είναι μια καινούρια γλώσσα προγραμματισμού, που σχεδιάστηκε ενσωματώνοντας τη δυνατότητα υψηλού επιπέδου επαναπροσδιορισμού της σωλήνωσης και ροής ελέγχου σε πρωτόκολλα PHY χωρίς να χρειαστεί να μετριάσει την απαιτούμενη επίδοση μιας πλατφόρμας SDR σε ό,τι αφορά τον ρυθμό μετάδοσης. Μια αναλυτική περιγραφή της γλώσσας και του τρόπου με τον οποίο το πετυχαίνει αυτό δίνεται στο κεφάλαιο 4.

## 2.2 Εισαγωγικά Στοιχεία της Haskell

Σε αυτήν την υποενότητα θα παρουσιάσουμε μερικά βασικά χαρακτηριστικά της Haskell στα οποία θα στηριχτούμε για να χτίσουμε την υλοποίησή μας.

### 2.2.1 Τύποι και Συναρτήσεις

Στη Haskell όλα είναι *τύποι*. Οι τύποι στη Haskell, όπως και στις άλλες γλώσσες δεν είναι παρά σύνολα των διαρθρωτικών τιμών της γλώσσας με κάποιους περιορισμούς. Για παράδειγμα στη Haskell το `Bool` είναι ο τύπος των τιμών `True` και `False`, ο `Int` είναι ο τύπος των λέξεων — με μέγεθος εξαρτώμενο

από το μηχάνημα που τρέχουμε το πρόγραμμα — ο `Double` είναι ο τύπος για διπλής ακρίβειας IEEE αριθμούς κινητής υποδιαστολής και οι αντιστοιχίσεις συνεχίζουν με τον ίδιο τρόπο, όπως συμβαίνει στη C, C++, Java και τις υπόλοιπες παραδοσιακές γλώσσες προγραμματισμού. Σημειώνουμε ότι όλα τα ονόματα τύπων στη Haskell ξεκινούν με κεφαλαίο γράμμα.

Επιπλέον των απλών αυτών τύπων η Haskell παρέχει δύο συντακτικές μορφές για να εκφράσει ενώσεις τύπων. Κατά πρώτον υπάρχουν τα ζευγάρια, οι τρίπλες και οι μεγαλύτερες δομές που μπορούν να γραφούν χρησιμοποιώντας σύνταξη για τούπλες, δηλαδή τύπους μέσα σε παρενθέσεις οι οποίοι χωρίζονται από κόμμα και οι οποίοι επιτρέπεται να είναι και διαφορετικοί μεταξύ τους. Για παράδειγμα, η `(Int,Bool)` είναι μια δομή που περιέχει μια `Int` και μια `Bool` συνιστώσα. Κατά δεύτερον υπάρχουν οι λίστες, όπου επιτρέπεται να περιέχουν μόνο τιμές του ίδιου τύπου και μπορούν να αναπαρασταθούν εν συντομία με αγκύλες. Δηλαδή, το [Int] είναι μία λίστα από Int.

Η Haskell έχει επίσης κι άλλους container τύπους. Ένας container τύπος που μπορεί να περιέχει έναν `Int` έχει τύπο `Maybe Int`. Οι containers τύποι αυτοί ξεκινούν επίσης με κεφαλαίο γράμμα. Σημειώνουμε ακόμη ότι οι τύποι στην Haskell μπορούν να εμφωλιαστούν σε οποιοδήποτε βάθος, π.χ. μπορούμε να έχουμε ένα `[Maybe (Int,Bool)]`.

Οι πολυμορφικοί τύποι στη Haskell εκφράζονται χρησιμοποιώντας πεζά γράμματα και παίζουν τον ίδιο ρόλο με τους `void*` δείκτες στη C και τα πολυμορφικά ορίσματα στα Java generics. Μπορούμε να ορίσουμε περιορισμούς πάνω σε αυτούς τους πολυμορφικούς τύπους, χρησιμοποιώντας το αντίστοιχο με την ιεραρχία αντικειμένων στη Haskell.

Τέλος, μία συνάρτηση γράφεται χρησιμοποιώντας ένα βελάκι από τον τύπο του ορίσματος στον τύπο του αποτελέσματος, π.χ. στη Haskell μία συνάρτηση που παίρνει σαν όρισμα μια λίστα και γυρίζει λίστα γράφεται σαν: `[a]` $\rightarrow$ `[a]`. Παρακάτω δίνουμε ένα παράδειγμα μιας Haskell συνάρτησης:

```
sort :: (Ord a) ⇒ [a] → [a]
sort []     = []
sort (x:xs) = sort before ++ [x] ++ sort after
  where before = filter (≤ x) xs
        after  = filter (> x) xs
```

Η συνάρτηση αυτή ταξινομεί μια λίστα χρησιμοποιώντας μια παραλλαγή της quicksort, στην οποία το pivot είναι απλά το πρώτο στοιχείο της λίστας:

- Στην πρώτη γραμμή έχουμε τον τύπο της `sort`, ο οποίος μας λέει ότι για $\forall a$ που επιδέχεται ταξινόμησης `Ord a`, η συνάρτηση παίρνει κι επιστρέφει μία λίστα από `a`.
- Η δεύτερη γραμμή μας λέει ότι μια άδεια γραμμή είναι ήδη ταξινομημένη.
- Οι υπόλοιπες γραμμές δηλώνουν ότι μία όχι άδεια λίστα μπορεί να ταξινομηθεί παίρνοντας το πρώτο στοιχείο για pivot και το υπόλοιπο της λίστας, επονομαζόμενα `x` και `xs` αντίστοιχα, ταξινομώντας τις τιμές πριν και μετά από αυτό το pivot κι ενώνοντας αυτές τις ενδιάμεσες τιμές μαζί.
- Οι ενδιάμεσες τιμές αυτές μπορούν να ονομαστούν χρησιμοποιώντας το `where`, στην περίπτωσή μας τα ονομάζουμε `before` και `after` αντίστοιχα.

Όπως φαίνεται και από το παραπάνω παράδειγμα, η Haskell έχει πολύ καθαρό συντακτικό. Μία συνάρτηση από τα ορίσματά της χωρίζεται με κενά, όχι με παρενθέσεις και κόμματα — οι παρενθέσεις χρησιμοποιούνται μόνο για να κάνουμε ένθετες κλήσεις συναρτήσεων. Επίσης, η εξαγωγή τύπων (type inference) που προσφέρει η Haskell μας επιτρέπει σε πολλές περιπτώσεις να παραλείπουμε τις δηλώσεις τύπων παρότι χρησιμοποιούμε μια γλώσσα με ισχυρό σύστημα τύπων.

Η Haskell είναι μια συνοπτική και άμεση γλώσσα. Οι δομές της δηλώνονται χρησιμοποιώντας τύπους, δομούνται και αποδομούνται, αλλά ποτέ δεν ενημερώνονται. Λόγου χάριν, ο τύπος `Maybe` μπορεί να οριστεί χρησιμοποιώντας δύο κατασκευαστές (constructors), τον `Nothing` και τον `Just`:

```
data Maybe where
    Nothing ::      Maybe a
    Just    :: a → Maybe a
```

Το `Nothing` είναι ένα `Maybe` από οτιδήποτε, το `Just` με ένα όρισμα είναι ένα `Maybe` με τον τύπο του ορίσματος. Οι κατασκευαστές αυτοί μπορούν να χρησιμοποιηθούν για δομήσουν και να αποδομήσουν δομές αλλά ποτέ για να τις ανανεώσουν. Όλες οι δομές στην Haskell είναι αμετάβλητες (immutable).

Είναι δυνατόν χρησιμοποιώντας το σύστημα υπερφόρτωσης (overloading) της Haskell να εμπλουτίσουμε ορισμένους τύπους με επιπλέον δυνατότητες όπως της ισότητας και της σύγκρισης, π.χ. μπορούμε να δώσουμε στον τύπο `Maybe` τη δυνατότητα να ελέγχει για ισότητα δημιουργώντας ένα στιγμιότυπο της κλάσης στον τύπο `Eq`:

```
instance Eq a ⇒ Eq (Maybe a) where
  Just a  ≡ Just b  = a ≡ b
  Nothing ≡ Nothing = True
  _       ≡ _       = False
```

Το παραπάνω δηλώνει ότι οποιοσδήποτε τύπος μπορεί να ελεγχθεί για ισότητα, μπορεί επίσης να ελεγχθεί για ισότητα μέσα στο `Maybe`. Χρησιμοποιώντας pattern matching μπορούμε να αποδομήσουμε το `Maybe` και να γυμνώσουμε τον εσωτερικό τύπο από το `Just`.

Στην Haskell οι παρενέργειες (side effects) όπως το γράψιμο στην οθόνη ή το διάβασμα από το πληκτρολόγιο σημειώνονται με χρήση του do:

```
main :: IO ()
main = do
  putStrLn "Hello"
  xs ← getLine
  print xs
```

Σε αυτό το παράδειγμα η `main` χρησιμοποιεί τη σημειογραφία `do` για να περιγράψει αλληλεπίδραση με τον χρήστη. Στην πραγματικότητα η do σημειογραφία εμπερικλείει μια δομή που καλείται *monad* και μας επιτρέπει να αποφεύγουμε να γράφουμε monadic τελεστές όπως $\gg=$ και $\ggg$ και αντί αυτού να κάνουμε bind monads μόνο με αλλαγή γραμμής. Σε επόμενη υποενότητα αναλύουμε και περισσότερο φορμαλιστικά τα monads.

### 2.2.2 Functors, Applicative Functors, Monads

Παραθέτουμε παρακάτω τις υλοποιήσεις κάποιων βασικών typeclasses που θα μας επιτρέψουν στη συνέχεια να δομήσουμε πιο περίπλοκους τύπους.

Η `Functor` typeclass περιλαμβάνει τους τύπους που μπορούν να γίνουν map. Έχει μια μοναδική μέθοδο που λέγεται `fmap` και της ορίζεται ως εξής:

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

Εδώ, το `f` είναι ένας κατασκευαστής τύπων (type constuctor) που παίρνει έναν τύπο για μεταβλητή — όπως γίνεται με το `Maybe Int`, ο οποίος είναι ένας συγκεκριμένος τύπος, ενώ το `Maybe` μόνο του είναι ένας κατασκευαστής τύπων όπου παίρνει έναν τύπο για μεταβλητή. Η `fmap` παίρνει μία συνάρτηση από έναν τύπο σε κάποιον άλλο και έναν functor που έχει εφαρμοστεί στον πρώτο τύπο και επιστρέφει

έναν functor που έχει εφαρμοστεί στο δεύτερο. Ένα γνωστό στιγμιότυπο της κλάσης είναι η `map` όπου ουσιαστικά δεν είναι παρά μία `fmap` που δουλεύει μόνο για λίστες, γεγονός που φαίνεται κι από το type signature της:

```
map :: (a → b) → [a] → [b]
```

Ένα ακόμη σημαντικό στιγμιότυπο της `Functor` που χρησιμοποιείται συχνά είναι ο τύπος της συνάρτησης `(→) r`. Ο τύπος της συνάρτησης `r → a`, επαναγράφεται σαν `(→) r a` για να είμαστε όμως σε θέση να τον κάνουμε στιγμιότυπο της `Functor` χρειάζεται να τον εφαρμοστεί μερικώς στο `(→) r` ώστε να δέχεται ακριβώς έναν τύπο σαν παράμετρο. Το στιγμιότυπο της κλάσης ορίζεται ως εξής:

```
instance Functor ((→) r) where
    fmap f g = (\x → f (g x))
```

Παραθέτουμε επίσης τον ορισμός της `Applicative` typeclass, ώστε να είμαστε σε θέση κατόπιν να ορίσουμε τα `Monads`:

```
class (Functor f) ⇒ Applicative f where
    pure :: a → f a
    (<*>) :: f (a → b) → f a → f b
```

Αρχικά παρατηρούμε ότι τίθεται ο περιορισμός (class constraint) ότι για να είναι κάτι `Applicative` πρέπει να είναι `Functor` πρώτα, έτσι όταν ένας κατασκευαστής τύπων είναι μέρος της `Applicative` typeclass μπορούμε να χρησιμοποιήσουμε την `fmap` σ'αυτόν.

Η πρώτη μέθοδος είναι η `pure` η οποία παίρνει μια τιμή και την τοποθετεί στο ελάχιστο πλαίσιο του applicative functor που εξακολουθεί να αποδίδει την τιμή αυτή.

Η $<*>$ μέθοδος παίρνει έναν functor που έχει μια συνάρτηση μέσα του, τρέχει κι εξάγει αυτήν τη συνάρτηση από τον πρώτο functor και μετά την κάνει map στο δεύτερο functor.

Παραθέτουμε ακόμη τον ορισμό του `Monad` typeclass, όπου θα χρησιμοποιηθεί αρκετές φορές στη συνέχεια.

```
class Monad m where
    return :: a → m a

    (>>=) :: m a → (a → m b) → m b

    (>>) :: m a → m b → m b
    x >> y = x >>= \_ → y

    fail :: String → m a
    fail msg = error msg
```

**Κώδικας 2.1:** Monad Typeclass

Η `return` είναι αντίστοιχη της `pure`, παίρνει μια τιμή και την εμπερικλείει σε ένα `Monad`. Η επόμενη συνάρτηση είναι η $>>=$, ή αλλιώς `bind`, η οποία προσομοιάζει με την εφαρμογή συνάρτησης, μόνο που αντί να παίρνει μια κανονική τιμή και να τη δίνει σε μια κανονική συνάρτηση, παίρνει μια monadic τιμή και τη δίνει σε μια συνάρτηση που παίρνει μια κανονική τιμή αλλά επιστρέφει monadic τιμή. Οι επόμενες δύο συναρτήσεις δεν μας απασχολούν ποτέ ιδιαίτερα, η $>>$ έρχεται με προκαθορισμένη υλοποίηση κι είναι ουσιαστικά μια `bind` που δεν χρησιμοποιεί το πρώτο της όρισμα, ενώ η `fail` χρησιμοποιείται, όπως υποδηλώνει και το όνομά της, από τη Haskell σε κάποιες περιπτώσεις αποτυχίας.

Όλα τα στιγμιότυπα του `Monad` typeclass πρέπει να ακολουθούν στους τρεις παρακάτω `monad` νόμους:

```
Left identity : return a >>= f ≡ f a
Right identity: m >>= return ≡ m
```

```
Associativity : (m >>= f) >>= g ≡ m >>= (\x → f x >>= g)
```

**Κώδικας 2.2:** Monad Laws

Θα μπορούσαμε να επεκταθούμε περισσότερο στην ανάλυση των `Monads` ωστόσο δεν είναι αυτός σκοπός μας εδώ, θα τα εξετάσουμε μόνο υπό το πρίσμα που χρειάζεται για να καταλάβουμε μερικά κύρια κομμάτια της υλοποίησης.

Ωστόσο για να αποκτήσουμε μια καλύτερη διαίσθηση, παραθέτουμε το στιγμιότυπο ενός πολύ συχνά χρησιμοποιούμενου `Monad`, αυτό της λίστας:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

Το `return` επιστρέφει μια λίστα με μόνο ένα στοιχείο. Για το bind κάνουμε map την συνάρτηση f στην αρχική μας λίστα, επειδή όμως η f είναι συνάρτηση από απλή σε monadic τιμή, το map της επιστρέφει μια λίστα από λίστες, οπότε εφαρμόζουμε ένα concat για να έχουμε σαν τελικό αποτέλεσμα μια flat λίστα με τις τιμές που παράχθηκαν από την f.

Ο τύπος της συνάρτησης, όπως αναφέρεται και παραπάνω, είναι functor, applicative functor αλλά και `Monad`:

```
instance Monad ((→) r) where
    return x = \_ → x
    h >>= f = \w → f (h w) w
```

Το `return` κλασικά παίρνει μια τιμή την τοποθετεί στο ελάχιστο περιβάλλον που πάντα θα έχει αυτήν την τιμή σαν αποτέλεσμα. Ο μόνος τρόπος να κατασκευάσεις μια συνάρτηση που έχει μια συγκεκριμένη τιμή σαν το αποτέλεσμά της είναι να αγνοήσεις εντελώς την παράμετρό της. Για την υλοποίηση του `bind` χρειαζόμαστε μια πιο αναλυτική προσέγγιση. Όταν χρησιμοποιούμε το $\gg=$ για να τροφοδοτήσουμε μια monadic τιμή σε μια συνάρτηση, το αποτέλεσμα είναι πάντα monadic τιμή. Οπότε σε αυτήν την περίπτωση όταν τροφοδοτούμε μια συνάρτηση σε μια άλλη συνάρτηση, το αποτέλεσμα είναι επίσης συνάρτηση και αυτός είναι ο λόγος που το αποτέλεσμα ξεκινά με $\lambda$. Για να λάβουμε το αποτέλεσμα από μια συνάρτηση πρέπει να την εφαρμόσουμε κάπου, έτσι εδώ καλούμε τη `(h w)` ώστε να πάρουμε το αποτέλεσμα από τη συνάρτηση και κατόπιν να εφαρμόσουμε την `f` σε αυτό. Η `f` επιστρέφει μια monadic τιμή, όπου εδώ είναι συνάρτηση και για αυτό την εφαρμόζουμε και αυτή στο `w`.

### 2.2.3 Free Monads

Ο διαχωρισμός των δεδομένων από τον ίδιο τον διερμηνέα (interpreter) που επεξεργάζεται τα δεδομένα αυτά είναι δείγμα καλού προγραμματισμού. Οι μεταγλωττιστές αποτελούν παραδείγματα μιας τέτοιας προσέγγισης, αφού τυπικά αναπαριστούν τον πηγαίο κώδικα σαν αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree, AST) που στη συνέχεια περνάει από μία ή περισσότερες διερμηνείες. Μπορούμε να επωφεληθούμε λοιπόν από την αποσύνδεση του διερμηνέα από το συντακτικό δέντρο, αφού πλέον έχουμε τη δυνατότητα να ερμηνεύσουμε το συντακτικό δέντρο με πολλούς τρόπους. Η προσέγγιση αυτή υιοθετείται συχνά σε περιπτώσεις που θέλουμε να ενσωματώσουμε μια γλώσσα συγκεκριμένου σκοπού στη Haskell και πολλές φορές αναφέρεται σαν *Free Monad with Interpreter* pattern.

Η Haskell προσφέρει το `Free` monad σαν έναν πρακτικό τρόπο να παράγει AST που στη συνέχεια μπορούν να προσπελαστούν με χρήση τυποποιημένων διευκολύνσεων των monads, χωρίς να χρειαστεί

να γραφτούν πολλές γραμμές προσαρμοσμένου, καινούριου κώδικα. Σε περιπτώσεις που χρησιμοποιείται για τη δημιουργίας μιας DSL ο τρόπος αυτός μας εξασφαλίζει ότι η τελική γλώσσα θα είναι εύκολη στη σύνθεση: ο προγραμματιστής μπορεί να ορίσει επιμέρους κομμάτια μέσα σε αυτή και μετά να συνθέσει τα μέρη αυτά με έναν δομημένο τρόπο, ο οποίος του επιτρέπει να εκμεταλλευτεί τις προγραμματιστικές αφαιρέσεις της Haskell. Ένα free monad δηλαδή, ικανοποιεί όλα τα monad laws (Κώδικας 2.2), αλλά δεν κάνει κάποιου είδους υπολογισμού, δομεί μόνο μια εμφωλευμένη σειρά περιεχομένων, όπως κάποιο AST. Τελικά Η ευθύνη να δώσει νόημα σε μια τέτοια σύνθεση, να κάνει κάποια ενέργεια με αυτήν περνάει στον προγραμματιστή που τη δημιούργησε αυτός είναι και ο λόγος που καθιστά μια τέτοια επιλογή ιδιαίτερα βολική για τη δημιουργία DSL.

Τα `Free` monads είναι ένας γενικός τρόπος να μετατρέπεις functors σε monads. Αυτό σημαίνει ότι δοθέντος ενός οποιοδήποτε functor `f`, το `Free f` είναι monad.

Αν το `X` είναι `Y` με επιπλέον μερικές ιδιότητες ή δομή `P`, τότε το `free X` είναι ένας τρόπος να πηγαίνεις από το `Y` στο `X` χωρίς να κερδίζεις κάτι παραπάνω.

Για παράδειγμα ας πάρουμε ένα monoid `(X)` κι ένα set `(Y)` με επιπλέον δομή `(P)`, η οποία ουσιαστικά είναι μια πράξη, όπως η πρόθεση κι ένα ουδέτερο στοιχείο, όπως το μηδέν.

Έχουμε δηλαδή, την κλάση των monoids:

```
class Monoid m where
   mempty  :: m
   mappend :: m → m → m
```

τον ορισμό της λίστας:

```
data [a] = [] | a : [a]
```

και παίρνουμε, ότι δοθέντος κάποιου τύπου `t`, είμαστε σε θέση να γνωρίζουμε ότι το `[t]` είναι monoid:

```
instance Monoid [t] where
  mempty  = []
  mappend = (++)
```

Έτσι θα μπορούσαμε να πούμε ότι οι λίστες είναι free monoids πάνω στα sets. Για τα free monads ισχύει η ίδια ιδέα, παίρνουμε έναν functor και δημιουργούμε ένα monad. Ο ορισμός του free monad μοιάζει με αυτόν της λίστας:

```
data Free f a = Pure a | Roll (f (Free f a))
```

το οποίο αυτόματα μπορεί να γίνει και monad, από τη στιγμή που το `f` είναι functor:

```
instance Functor f ⇒ Functor (Free f) where
  fmap f (Pure a) = Pure (f a)
  fmap f (Roll x) = Roll (fmap (fmap f) x)

concatFree :: Functor f ⇒ Free f (Free f a) → Free f a
concatFree (Pure x) = x
concatFree (Roll y) = Roll (fmap concatFree y)

instance Functor f ⇒ Monad (Free f) where
  return = Pure
  x >>= f = concatFree (fmap f x)
```

### 2.2.4 Free Monad Transformers

Κάποιες φορές δεν είναι δυνατόν να καθορίσουμε το συντακτικό δέντρο όλο απευθείας. Για παράδειγμα συχνά θέλουμε να παρεμβάλλουμε στο συντακτικό δέντρο με κάποιο άλλο monad για να παράγουμε streaming υπολογισμούς. Το `FreeT` λύνει το πρόβλημα αυτό επιτρέποντας να αναμείξουμε βήματα από το χτίσιμο του αφηρημένου συντακτικού δέντρου με κλήσεις ενεργειών σε κάποιο βασικό monad:

```
data FreeF f a x = Pure a | Free (f x)

newtype FreeT f m a = FreeT {
    runFreeT :: m (FreeF f a (FreeT f m a)) }
```

**Κώδικας 2.3:** Free Monad Transformer

Για παράδειγμα, ας πούμε ότι θέλουμε να γράψουμε τον δικό μας Python generator:

```
type Generator b m r = FreeT ((,) b) m r
```

Στην περίπτωση αυτή, το συντακτικό μας δέντρο είναι μια απλή λίστα όπου τρέχουμε το βασικό monad για να δημιουργηθεί το επόμενο στοιχείο. Τώρα είμαστε σε θέση να αναπαράγουμε το συντακτικό της Python:

```
yield :: b → Generator b m ()
yield b = liftF (b, ())
```

Σε περίπτωση που θέλουμε να ζητάμε από τον χρήστη να εισάγει κάθε επόμενο στοιχείο της λίστας, μπορούμε να θέσουμε το monad βάσης σε `IO`:

```
prompt :: Generator String IO r
prompt = forever $ do
    lift $ putStrLn "Enter a string:"
    str ← lift getLine
    yield str
```

Στη συνέχεια μπορούμε να ζητήσουμε το επόμενο στοιχείο από το generator μας χρησιμοποιώντας την `runFreeT`:

```
main = do
    x ← runFreeT prompt
    case x of
        Pure       _  → return ()
        Free (str, _) → putStrLn $ "User entered: " ++ str
```

Με αυτόν τον τρόπο δεν απαιτούμε από το χρήστη να δώσει άπειρο αριθμό τιμών, του ζητάμε απλά να εισάγει όσες τιμές απαιτείται από τον generator — μόνο μία, στο παράδειγμά μας.

Εδώ τελειώσαμε με τα εισαγωγικά στοιχεία της Haskell, στο Κεφάλαιο 5 θα δούμε πώς αυτά εφαρμόζονται στην υλοποίησή μας.

## Κεφάλαιο 3

# Γλώσσες Συγκεκριμένου Σκοπού

## 3.1 Εισαγωγή

Μία γλώσσα συγκεκριμένου σκοπού (**Domain Specific Language, DSL**) είναι μια γλώσσα προγραμματισμού εξειδικευμένη σε ένα συγκεκριμένο τομέα εφαρμογής. Μία DSL έχει σχεδιαστεί δηλαδή ειδικά για να εκφράσει λύσεις στα προβλήματα αυτού του δεδομένου τομέα. Από την άλλη πλευρά, οι γλώσσες γενικού σκοπού (General Purpose Language, GPL) στερούνται οποιασδήποτε εξειδίκευσης και μπορούν να εφαρμοστούν για να λύσουν προβλήματα σε πολλούς τομείς. Ωστόσο ο διαχωρισμός δεν είναι πάντοτε σαφής, αφού μία DSL μπορεί τελικά να βρει εφαρμογή σε ευρύτερο πλαίσιο ή αντίστροφα μπορεί κατ'αρχήν μια γλώσσα να έχει δημιουργηθεί ως GPL αλλά πρακτικά να χρησιμοποιείται κυρίως σε έναν συγκεκριμένο τομέα. Στην πρώτη κατηγορία ανήκουν η Perl, η AWK και τα shell scripts. Ενώ στη δεύτερη κατηγορία για παράδειγμα ανήκει η PostScript, όπου είναι μια Turing complete γλώσσα, αλλά πρακτικά χρησιμοποιείται κυρίως ως γλώσσα περιγραφής σελίδων.

Υπάρχουν πολλά παραδείγματα DSL γλωσσών. Οι DSL περιλαμβάνουν όλες τις γλώσσες επερωτήσεων (όπως SQL, XPath), όλες τις γλώσσες προτύπων (Django, Smarty), γλώσσες για αποθήκευση κι ανταλλαγή δεδομένων (XML, YAML), γλώσσες εγγράφων (LaTeX, HTML, CSS), macro γλώσσες (make, ant, rake), γλώσσες για λεκτική και συντακτική ανάλυση γλωσσών (lexx, yacc) ή η γλώσσα των κανονικών εκφράσεων (regular expressions).

## 3.2 Εξωτερικές και Εσωτερικές Γλώσσες Συγκεκριμένου Σκοπού

Ένας σημαντικός και χρήσιμος διαχωρισμός που γίνεται μεταξύ των DSLs είναι ο εξής [15]:

**Εσωτερικές DSLs**
: Οι εσωτερικές (internal) ή διαφορετικά ενσωματωμένες (embedded) γλώσσες είναι γλώσσες που εξαρτώνται από τη host γλώσσα και μπορούν να θεωρηθούν ως επεκτάσεις της. Σε πολλές περιπτώσεις, χρησιμοποιούνται οι πρωταρχικές εντολές της host γλώσσας για έλεγχο ροής. Ο κώδικας της host γλώσσας μπορεί εύκολα να κληθεί κατά βούληση ωστόσο η DSL δεν μπορεί να υπάρξει ξεχωριστά από τη host γλώσσα. Πολλές DSLs έχουν υλοποιηθεί σαν Lisp Macros, ενώ πρόσφατα η κοινότητα της Ruby υιοθέτησε ευρέως την προσέγγιση αυτή. Γλώσσες αντικειμενοστραφούς προγραμματισμού όπως η Java και η C♯ έχουν επίσης τη δυνατότητα να παίξουν αποδοτικά το ρόλο της host γλώσσας ενώ υπάρχει μια μεγάλη παράδοση στην κατασκευή DSL σε μοντέρνες γλώσσες συναρτησιακού προγραμματισμού όπως η ML και η Haskell. Τα πλεονεκτήματα και τα μειονεκτήματα συγκεκριμένα της επιλογής της Haskell ως host καθώς και οι τρόποι που μπορεί να γίνει ενσωμάτωση μιας DSL σε αυτή θα αναλυθούν περαιτέρω στην Ενότητα 3.4.

**Εξωτερικές DSLs**

Οι εξωτερικές (external) DSLs είναι ανεξάρτητες και όχι επεκτάσεις κάποιας συγκεκριμένης host γλώσσας. Όταν κάποιος προγραμματίζει σε μία τέτοια γλώσσα χρησιμοποιεί μόνο τη λειτουργικότητα που έχει προσδιοριστεί από τη γλώσσα. Μερικές από αυτές δίνουν τη δυνατότητα κλήσης κώδικα που είναι υλοποιημένος σε μια GPL, αλλά κάτι τέτοιο είναι προαιρετικό. Οι εξωτερικές DSLs έχουν το δικό τους custom συντακτικό και χρειάζονται ολόκληρο δικό τους parser για να τις επεξεργαστείς. Οι DSLs μπορούν να υλοποιηθούν είτε μέσω διερμηνέα (interpreter) είτε μέσω παραγωγής κώδικα (code generation). Η διερμηνεία, δηλαδή να διαβάζουμε το DSL script και να το τρέχουμε στο χρόνο εκτέλεσης είναι συνήθως πιο εύκολη, αλλά η παραγωγή κώδικα σε μερικές περιπτώσεις είναι εξαιρετικά σημαντική. Συνήθως ο παραγόμενος κώδικας είναι σε μια υψηλού επιπέδου γλώσσα προγραμματισμού όπως C ή Java.

### 3.2.1 Πλεονεκτήματα και Μειονεκτήματα

**Πλεονεκτήματα**

Οι DSLs έχουν γίνει ευρέως αποδεκτές για πολλούς λόγους, δύο από τους πιο σημαντικούς είναι ότι βελτιώνουν άμεσα την παραγωγικότητα των προγραμματιστών ενώ συγχρόνως καθιστούν ευκολότερη την επικοινωνία μεταξύ των ειδικών του εκάστοτε τομέα. Μία προσεκτικά επιλεγμένη DSL συμβάλλει στο να γίνει ένα περίπλοκο τμήμα κώδικα πιο εύληπτο και συνεπώς στο να βελτιώσει την παραγωγικότητα των προγραμματιστών που δουλεύουν μαζί του. Συνεισφέρει επίσης στη βελτίωση της επικοινωνίας μεταξύ των ειδικών, παρέχοντας ένα απλό σχετικά κείμενο που μπορεί να λειτουργήσει σαν εκτελέσιμο λογισμικό και ταυτόχρονα σαν μία περιγραφή όπου μπορούν εύκολα να διαβάσουν και να καταλάβουν τον τρόπο με τον οποίο οι ιδέες τους αναπαριστώνται στο σύστημα. Αυτή η ενδοεπικοινωνία μεταξύ ειδικών είναι ένα όφελος πιο δύσκολο να επιτευχθεί, ωστόσο το κέρδος που προκύπτει είναι πολύ μεγαλύτερο αφού βοηθάει να αποσυμφορηθεί ένα από τα πιο δύσκολα σημεία στην ανάπτυξη λογισμικού, αυτό δηλαδή της συνεννόησης μεταξύ των προγραμματιστών και των πελατών τους. Ένα ακόμη πλεονέκτημα των DSLs είναι ότι επιτρέπουν την επαλήθευση του κώδικα σε επίπεδο που αφορά τον συγκεκριμένο τομέα. Εφόσον οι γλωσσικές δομές που χρησιμοποιούνται είναι ασφαλείς οποιαδήποτε πρόταση γράφεται μέσω αυτών μπορεί να θεωρηθεί ασφαλής.

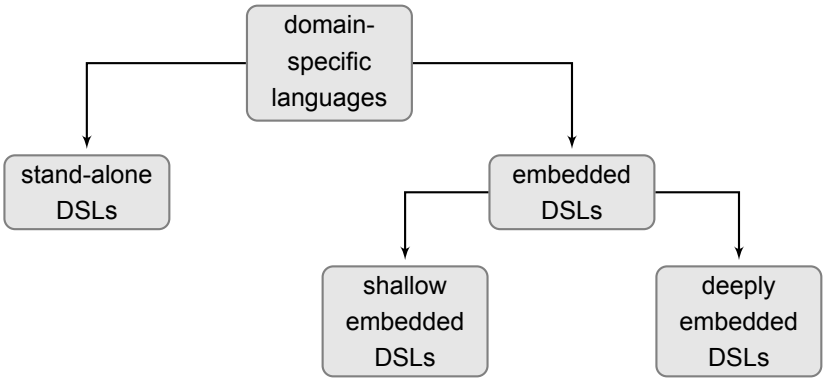
**Μειονεκτήματα**

Τα μειονεκτήματα προκύπτουν από το γεγονός ότι ουσιαστικά μιλάμε για μια καινούρια γλώσσα με ό,τι αυτό συνεπάγεται. Χρειάζεται κάποιος χρόνος εκμάθησης της νέας γλώσσας που όμως έχει περιορισμένη εφαρμοσιμότητα. Υπάρχει κόστος για τον σχεδιασμό, την υλοποίηση και τη διατήρηση της ίδιας της DSL αλλά και των εργαλείων που απαιτούνται για την διαδικασία ανάπτυξης. Άλλα μειονεκτήματα πηγάζουν από το γεγονός ότι μιλάμε για γλώσσα συγκεκριμένου σκοπού οπότε χρειάζεται να βρεθεί, να τεθεί και να διατηρηθεί το αντίστοιχο κατάλληλο πεδίο, ενώ μπορεί συγκριτικά πάντα με την επιλογή μιας GPL να αυξηθεί σημαντικά η δυσκολία ενσωμάτωσης της DSL με τις υπόλοιπες συνιστώσες του IT συστήματος. Ενέχει επίσης η πιθανότητα απώλειας επεξεργαστικής ισχύος σε σχέση με λογισμικό που έχει γραφτεί από την αρχή με το χέρι. Τέλος οι ειδικοί, που τυγχάνει να μην είναι τόσο τεχνικά καταρτισμένοι, πιθανώς να δυσκολευτούν να γράψουν ή να αλλάξουν προγράμματα, που έχουν γραφτεί στην DSL, μόνοι τους αφού εκτός των άλλων είναι και δυσκολότερο να βρει κανείς παραδείγματα κώδικα σε τέτοιες γλώσσες.

Ένας συχνός συσχετισμός που γίνεται όταν μιλάμε για τα οφέλη ή και τα προβλήματα των DSLs είναι η αντικατάσταση της έννοιας DSL με μία βιβλιοθήκη σε γλώσσες γενικού σκοπού. Πολλά από όσα κερδίζει κανείς με μια DSL μπορεί επίσης να τα κερδίσει δημιουργώντας ένα framework. Στην πραγματικότητα, πολλές DSLs είναι μία λεπτή επίστρωση πάνω από μια βιβλιοθήκη ή ένα framework. Υπάρχουν ήδη διάφορες γνωστές τεχνικές που επιτρέπουν τη μείωση του κόστους δημιουργίας μίας DSL δραματικά κι είναι καλό να ληφθούν υπ' όψιν σε μια προσπάθεια δημιουργίας μιας νέας DSL.

## 3.3 Ρηχή και Βαθιά Ενσωμάτωση Ενσωματωμένων Γλωσσών

Οι EDSLs με τη σειρά τους μπορούν να διακριθούν σε δύο βασικές κατηγορίες, ανάλογα με το βαθμό ενσωμάτωσης που παρέχουν, όπως φαίνεται στο Σχήμα 3.1:




**Σχήμα 3.1:** Τύποι των γλωσσών συγκεκριμένου σκοπού

**Ρηχή Ενσωμάτωση**

Η ρηχή ενσωμάτωση (shallow embedding) αποτυπώνει τη σημασιολογία των δεδομένων του εκάστοτε τομέα σε έναν τύπο δεδομένων παρέχοντας μια συγκεκριμένη διερμηνεία για τα δεδομένα αυτά. Δηλαδή, όλες οι λειτουργίες της Haskell μεταφράζονται άμεσα στην τελική γλώσσα στόχο, π.χ. η Haskell έκφραση `a+b` μεταφράζεται σε ένα `String` όπως `"a+b"` που περιέχει την έκφραση στην τελική γλώσσα. Το αποτέλεσμα ενός υπολογισμού σε μια ρηχή ενσωμάτωση είναι μια τιμή (value) που υπολογίζεται απευθείας, χωρίς να περάσει από ενδιάμεσα στάδια.

**Βαθιά Ενσωμάτωση**

Η βαθιά ενσωμάτωση (deep embedding) δεν σταματά στην αποτύπωση της σημασιολογίας των δεδομένων αλλά πηγαίνει ένα βήμα παρακάτω αποτυπώνοντας τη σημασιολογία των λειτουργιών που γίνονται στον τομέα, ενεργοποιώντας τη διερμηνεία μεταβλητών. Πιο αναλυτικά, οι λειτουργίες της Haskell χτίζουν μια ενδιάμεση δομή δεδομένων που αντιστοιχεί στο δέντρο εκφράσεων του προγράμματος (Abstract Syntax Tree, AST). Για παράδειγμα, η Haskell έκφραση `a+b` μεταφράζεται στη Haskell δομή δεδομένων `Add (Var "a") (Var "b")`. Η δομή αυτή επιτρέπει να γίνουν διάφοροι μετασχηματισμοί, όπως βελτιστοποιήσεις, πριν από τη μετάφραση στην τελική γλώσσα. Το αποτέλεσμα ενός υπολογισμού που γίνεται σε περιβάλλον βαθιάς ενσωμάτωσης είναι μια δομή, όχι μία τιμή, κι αυτή η δομή μπορεί να χρησιμοποιηθεί για να υπολογιστεί μια τιμή ή ακόμη και να γίνει cross-compile πριν υπολογιστεί.

## 3.4 Ενσωμάτωση Γλωσσών Συγκεκριμένου Σκοπού στη Haskell

Στην ενότητα 3.1 αναφερθήκαμε στις γλώσσες συγκεκριμένου σκοπού (DSLs) και είδαμε πως αυτές χωρίζονται σε εξωτερικές και εσωτερικές/ενσωματωμένες. Στην τρέχουσα υποενότητα θα ασχοληθούμε κυρίως με τη δεύτερη κατηγορία, δηλαδή τις ενσωματωμένες γλώσσες συγκεκριμένου σκοπού (**Embedded DSLs, EDSLs**) και πιο συγκεκριμένα θα δούμε γιατί η Haskell μπορεί να είναι μια καλή επιλογή για host γλώσσα.

Κατά μία έννοια, οι EDSLs δεν είναι παρά βιβλιοθήκες της host γλώσσας που έχουν όμως μία δική τους μοναδική εμφάνιση και αισθητική, προσαρμοσμένη στο συγκεκριμένο τομέα. Επαναχρησιμοποιώντας τα εργαλεία και τις διευκολύνσεις της host γλώσσας, η επιλογή για ενσωμάτωση μειώνει σημαντικά το κόστος ανάπτυξης και διατήρησης της εκάστοτε DSL. Οι EDSLs ωστόσο είναι από το σχεδιασμό τους συνυφασμένες με τις δυνατότητες της host γλώσσας. Έτσι κληρονομούν τον μεταγλωττιστή και τα γενικού σκοπού γνωρίσματα της host γλώσσας δωρεάν, αφήνοντας όμως ταυτόχρονα την πολυπλοκότητα της host γλώσσας εκτεθειμένη και διαθέσιμη στον προγραμματιστή. Αυτό συχνά έχει σαν συνέπεια να δυσχεραίνει τη διαδικασία αποσφαλμάτωσης, αφού ένα bug στο επίπεδο της εφαρμογής θα χρειαστεί να ανιχνευτεί σε επίπεδο host γλώσσας. Επίσης ο debugger εκθέτει όλες τις ιδιωτικές δομές δεδομένων, γεγονός που μπορεί να δυσκολέψει ακόμη περισσότερο τους προγραμματιστές της εφαρμογής να συνδέσουν αυτό που βλέπουν στην οθόνη με τη λογική που έχει δομηθεί το πρόγραμμα.

Σε αυτά τα σημεία η Haskell, όντας η κορυφαία αμιγώς συναρτησιακή γλώσσα προγραμματισμού, έχει να προσφέρει μερικά συγκριτικά πλεονεκτήματα σε σχέση με άλλες γλώσσες. Η οκνηρή αποτίμηση, το ισχυρό σύστημα τύπων, ο ευέλικτος πολυμορφισμός είναι μερικά μόνο από τα χαρακτηριστικά που καθιστούν τη Haskell μια δημοφιλής επιλογή για host γλώσσα. Παρακάτω θα δώσουμε ένα απλό παράδειγμα ενσωμάτωσης σαν μια εισαγωγή σε μερικά από αυτά τα χαρακτηριστικά ώστε να είμαστε σε θέση να τα διακρίνουμε και στη συνέχεια στη δική μας υλοποίηση.

### 3.4.1 Χτίζοντας μια Απλή EDSL

Για να καταδείξουμε τον τρόπο που δομείται μια EDSL και τις διαφορές ανάμεσα σε ρηχή και βαθειά ενσωμάτωση θα προσπαθήσουμε να ενσωματώσουμε στη Haskell μια απλή γλώσσα εκφράσεων με πρόσθεση, αφαίρεση, πολλαπλασιασμό και σταθερές.

Εκφράζουμε τη γλώσσα μέσω της παρακάτω διεπαφής: χρησιμοποιούμε ένα συνώνυμο τύπου `Exp` για τα απλά `Int`s και τρεις διαφορετικές συναρτήσεις που αντιπροσωπεύουν τις σταθερές, την πρόσθεση, την αφαίρεση και τον πολλαπλασιασμό:

```
type Exp = Int

Lit :: Int → Exp
Add :: Exp → Exp → Exp
Sub :: Exp → Exp → Exp
Mul :: Exp → Exp → Exp
```

Με αυτόν τον τρόπο ενσωματώνουμε τα δεδομένα του πεδίου στη Haskell, παρέχουμε συναρτήσεις για την κατασκευή του μοντέλου κι έτσι μπορούμε εύκολα να αναπαραστήσουμε τον υπολογισμό μια έκφρασης όπως η $4 + 6 * 8$ με την παρακάτω γραμμή Haskell:

```
val = Lit 4 `Add` (Lit 6 `Mul` Lit 8)
```

Το πλεονέκτημα αυτής της ρηχής ενσωμάτωσης είναι ότι ο υπολογισμός της τιμής της έκφρασης γίνεται πολύ γρήγορα. Πέραν όμως της τελικής τιμής δεν μπορούμε να αποφασίσουμε τίποτα άλλο που αφορά την έκφρασή μας. Αυτή η κατάσταση μπορεί να γίνει ακόμη πιο προβληματική όταν προσθέτουμε μεταβλητές στη γλώσσα μας. Μπορούμε να διαφοροποιήσουμε τον τύπο μας να περιέχει binding πληροφορία και να προσθέσουμε δύο συναρτήσεις που αναπαριστούν την ανάθεση και τη χρήση μεταβλητών:

```
assign :: String → Int → Exp
var :: String → Exp
```

Τώρα θα μπορούσαμε αφελώς να παρασυρθούμε και να γράψουμε την έκφραση $x + 6 * 8$ ως εξής:

```
val = Var "x" `Add` (Lit 6 `Mul` Lit 8)
```

Η αποτίμηση της έκφρασης αυτής θα δημιουργούσε απρόβλεπτη συμπεριφορά, αφού δεν γνωρίζουμε την τιμή του x. Για να αποφύγουμε κάτι τέτοιο θα έπρεπε να την εισάγουμε προτού τη χρησιμοποιήσουμε:

```
val = let "x" 4 (Lit "x" `Add` (Lit 6 `Mul` Lit 8))
```

Αφού έχουμε αναθέσει τιμή στο x μπορούμε πλέον με ασφάλεια να τη χρησιμοποιήσουμε στην έκφρασή μας.

Αν είχαμε προτιμήσει να προβούμε σε βαθιά ενσωμάτωση θα είχαμε αποτρέψει τέτοιου είδους λάθη να συμβαίνουν ελέγχοντας πρώτα για κάθε μεταβλητή αν έχει ανατεθεί πριν χρησιμοποιηθεί. Για να επιτύχουμε το deep embedding εκφράζουμε τη γλώσσα μέσω της παρακάτω διεπαφής, δημιουργώντας ένα νέο GADT[1] με τέσσερις διαφορετικές συναρτήσεις που αντιπροσωπεύουν τις σταθερές και τις πράξεις:

```
data Exp where
  Lit :: Int → Exp
  Add :: Exp → Exp → Exp
  Sub :: Exp → Exp → Exp
  Mul :: Exp → Exp → Exp
  deriving Eq
```

Στη συνέχεια θα κάνουμε τον καινούριο τύπο δεδομένων στιγμιότυπο της Num ώστε να μπορούμε να τον χρησιμοποιούμε σε αριθμητική με ακεραίους:

```
instance Num Expr where
  fromInteger n = Lit n
  e1 + e2 = Add e1 e2
  e1 − e2 = Sub e1 e2
  e1 ∗ e2 = Mul e1 e2
```

Χτίζοντας εκφράσεις τύπου Expr αποκαλύπτουμε τη δομή του υπολογισμού:

```
GHCi> 1 + 2 ∗ 3 :: Expr
Add (Lit 1) (Mul (Lit 2) (Lit 3))
```

Αυτό είναι ένα μεγάλο όφελος αφού μας επιτρέπει να γράψουμε μια έκφραση και να εξάγουμε ένα δέντρο, μια αναπαράσταση του τι να κάνουμε, και όχι ένα άμεσο αποτέλεσμα. Με αυτόν τον τρόπου το deep embedding μας επιτρέπει να χρησιμοποιούμε τη σημασιολογία του μοντέλου μας ορίζοντας πολλαπλές ερμηνείες για την DSL. Το μειονέκτημα είναι ότι ο απλός υπολογισμός της αξίας της έκφρασης γίνεται πιο αργός αφού έχει προστεθεί το κόστος των κατασκευαστών.

Στην περίπτωση της βαθιάς ενσωμάτωσης λοιπόν είναι συνηθισμένο να γράφουμε μια συνάρτηση που τελικά υπολογίζει το αποτέλεσμα, για παράδειγμα:

```
run :: Expr → Integer
run (Lit n)   = n
run (Add a b) = run a + run b
run (Sub a b) = run a − run b
run (Mul a b) = run a ∗ run b
```

[1] Οι Generalized Algebraic Datatypes (GADts) επιτρέπουν στον προγραμματιστή να γράψει ρητά τους τύπους των κατασκευαστών.

### 3.4.2 Πλεονεκτήματα της Haskell ως Host Γλώσσας

Συγκεντρώνουμε εδώ μερικά από τα χαρακτηριστικά της Haskell που την καθιστούν μια ιδανική επιλογή για host σε περίπτωση ενσωμάτωσης μια γλώσσας συγκεκριμένου σκοπού:

- Η Haskell έχει πολύ καθαρό συντακτικό. Μία συνάρτηση από τα ορίσματά της χωρίζεται με κενά, όχι με παρενθέσεις και κόμματα — οι παρενθέσεις χρησιμοποιούνται μόνο για να κάνουμε ένθετες κλήσεις συναρτήσεων. Η εξαγωγή τύπων (type inference) που προσφέρει η Haskell μας επιτρέπει να παραλείπουμε τις δηλώσεις τύπων παρότι χρησιμοποιούμε μια γλώσσα με ισχυρό σύστημα τύπων. Ενώ το syntactic sugar που παρέχει για τα monads μας επιτρέπει να αποφεύγουμε να γράφουμε monadic τελεστές όπως $\gg=$ και $\ggg$ και αντί αυτού να κάνουμε bind monads μόνο με αλλαγή γραμμής.
- Το ισχυρό σύστημα τύπων μας επιτρέπει να εκφράζουμε πολύ εκλεπτυσμένους περιορισμούς που μπορούν να εφαρμοστούν κατά τη χρήση των EDSL συνιστωσών αλλά και για να εκφράσουν τη σχέση τους με άλλα μέρη της γλώσσας.

Επωφελούμενη από την περιεκτική σύνταξη της, η κοινότητα της Haskell — αλλά και γενικότερα η κοινότητα του συναρτησιακού προγραμματισμού — έχει αναπτύξει έναν μεγάλο αριθμό EDSLs που παρέχουν υψηλότερου επιπέδου διεπαφές και αφαιρέσεις για καλά οργανωμένα συστήματα.

## Κεφάλαιο 4

# Η Γλώσσα Ziria

## 4.1 Εισαγωγή

Η Ziria είναι μια νέα γλώσσα συγκεκριμένου σκοπού (DSL - 3.1) σχεδιασμένη για ραδιοεπικοινωνία ορισμένη από λογισμικό (SDR - 2.1). Τα εργαλεία που προσφέρει είναι κατάλληλα για ασύρματο προγραμματισμό στο φυσικό επίπεδο, λαμβάνοντας υπ'όψιν τον επαναπροσδιορισμό σωλήνωσης (pipeline reconfiguration) σαν βασικό συστατικό στοιχείο του PHY προγραμματισμού, ενώ επιτυγχάνει ταυτόχρονα τις απαραίτητες ταχύτητες γραμμής (line-rate) για SDR.

Ο μεταγλωττιστής της Ziria υλοποιεί ένα ευρύ σύνολο βελτιστοποιήσεων, όπως αύξηση του εύρους δεδομένων (bus width), διανυσματοποίηση (vectorization), δημιουργία αυτόματων πινάκων αντιστοίχισης (lookup table, LUT), απαλοιφή κλήσεων (inlining), μερική αποτίμηση, βελτιστοποίηση διαχείρισης μνήμης, σύντηξη αγωγού (pipeline fusion), auto-mapping — μία παραλλαγή στατικού προγραμματισμού — και άλλες βελτιστοποιήσεις θεμελιώδεις για να επιτευχθούν οι επιθυμητές ταχύτητες επεξεργασίας.

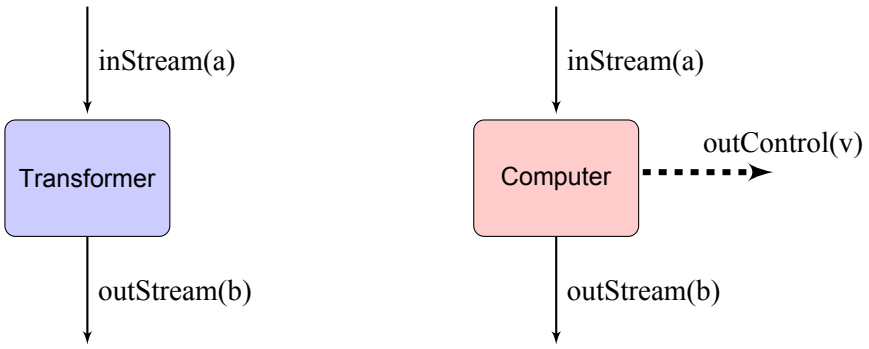
## 4.2 Προγραμματισμός στη Ziria

Σε αυτήν την ενότητα θα παρουσιάσουμε τις πιο βασικές προγραμματιστικές αφαιρέσεις της Ziria καθώς και τον τρόπο με τον οποίο αυτές μπορούν να βοηθήσουν τον προγραμματιστή να συνθέσει περίπλοκες σωληνώσεις όπως για παράδειγμα αυτών που χρειάζονται για το IEEE 802.11a/g PHY. Στη συνέχεια, θα καταδείξουμε πως οι προγραμματιστές που είναι εξοικειωμένοι με προστακτικές γλώσσες, όπως η C, μπορούν εύκολα να υλοποιήσουν τα βασικά μπλοκ αυτών των σωληνώσεων σαν προστακτικό κώδικα στην γλώσσα υπολογισμών της Ziria.

### 4.2.1 Προγραμματιστικές Αφαιρέσεις της Ziria για τις Ροές Δεδομένων

Τα προγράμματα Ziria επεξεργάζονται ροές δεδομένων (*streams*) τιμών: διαβάζουν τιμές από μία, πιθανώς άπειρη, ροή δεδομένων εισόδου (input stream) και γράφουν τιμές σε μια ροή δεδομένων εξόδου (output stream). Ένα πρακτικό παράδειγμα είναι αυτό ενός WiFi δέκτη, ο οποίος σε υψηλό επίπεδο δεν είναι παρά ένας υπολογισμός που διαβάζει μια ροή δεδομένων μιγαδικών αριθμών (I/Q δείγματα) από έναν A/D μετασχηματιστή και παράγει μια ροή δεδομένων από bytes που αντιστοιχούν σε MAC-layer πακέτα.

Η Ziria δίνει τη δυνατότητα παραγωγής και σύνθεσης υπολογισμών δύο διαφορετικών τύπων, Σχήμα 4.1:




**Σχήμα 4.1:** Τύποι υπολογισμών της Ziria: Transformer (αριστερά), Computer (δεξιά)

**Stream transformers**

Οι μετατροπείς ροών δεδομένων (stream transformers) είναι υπολογισμοί που προσομοιάζουν τις παραδοσιακές διατάξεις επεξεργασίας ροών δεδομένων που συναντούμε ήδη στις υπάρχουσες SDR πλατφόρμες, όπως το GnuRadio και η Sora. Εκτελούν επ' αόριστον, παίρνοντας τιμές από τη ροή δεδομένων εισόδου και διοχετεύοντας τιμές στη ροή δεδομένων εξόδου, κρατώντας πιθανώς κάποια εσωτερική κατάσταση. Ένα απλό παράδειγμα είναι αυτό μιας διάταξης scrambler.[1]

**Stream computers**

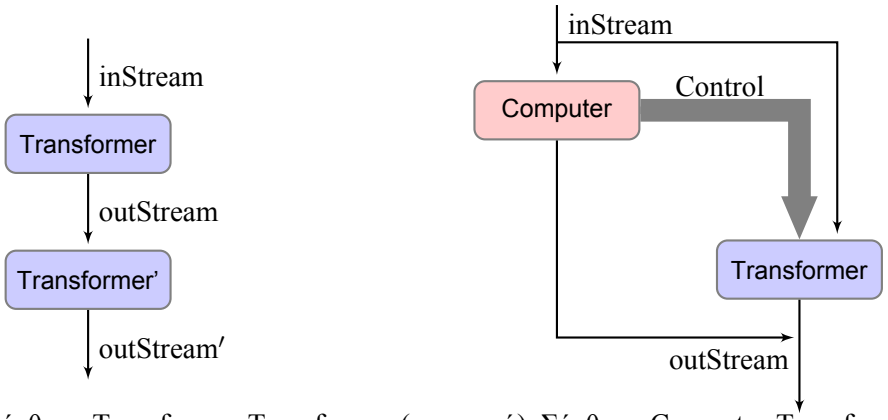
Οι υπολογιστές ροών δεδομένων (stream computers) είναι μια νέα πρωτοποριακή ιδέα στον SDR προγραμματισμό. Όπως οι stream transformers, παίρνουν τιμές από την ροή δεδομένων εισόδου και διοχετεύουν τιμές στη ροή δεδομένων εξόδου. Διαφοροποιούνται όμως από τους transformers, γιατί η εκτέλεσή τους γίνεται για ένα πεπερασμένο διάστημα και τελειώνει επιστρέφοντας μια επιπρόσθετη τιμή, στην οποία θα αναφερόμαστε ως τιμή ελέγχου (control value). Στη Ziria η τιμή ελέγχου χρησιμοποιείται για να επαναπροσδιοριστεί δυναμικά η υπόλοιπη πορεία της σωλήνωσης, σε μια διαδικασία που ονομάζεται *binding* και αντανακλά άμεσα την ροή ελέγχου πολλών PHY-layer πρωτοκόλλων. Για παράδειγμα, η αποκωδικοποίηση μιας επικεφαλίδας WiFi πακέτου αντιστοιχεί σε έναν stream computer: αποκωδικοποιεί την επικεφαλίδα ενός πακέτου και τελικά τερματίζει επιστρέφοντας μια τιμή ελέγχου που εμπεριέχει παραμέτρους για την κωδικοποίηση και τη διαμόρφωση, οι οποίες στη συνέχεια χρησιμοποιούνται για να αποκωδικοποιήσουν το ωφέλιμο φορτίο (playload) του πακέτου. Το γεγονός ότι οι υπολογιστές επεξεργάζονται συνεχώς δεδομένα εξόδου ενώ ταυτόχρονα υπολογίζουν την τελική τιμή επιστροφής επιτρέπει στις μεταγενέστερες συνιστώσες της σωλήνωσης να χρησιμοποιούν αυτές τις εξόδους χωρίς διακοπή, μειώνοντας έτσι τη συνολική καθυστέρηση του προγράμματος.

Το σύστημα τύπων της Ziria κάνει έναν σαφή διαχωρισμό μεταξύ αυτών των δύο αφαιρέσεων. Αναθέτει έναν τύπο `(Zr T a b)` στους Ziria transformers, οι οποίοι παίρνουν είσοδο τύπου `a` και παράγουν έξοδο τύπου `b` καθώς κι έναν τύπο `(Zr (C c) a b)` στους Ziria computers, οι οποίοι αντίστοιχα παίρνουν μια είσοδο τύπου `a` και παράγουν έξοδο τύπου `b`, ενώ τελικά επιστρέφουν μια τιμή ελέγχου τύπου `c`. Οι τύποι των τιμών `a`, `b`, `c` συνήθως περιλαμβάνουν bit, ακεραίους, μιγαδικούς διάφορων πλατών, δομές ή πίνακες με στατικό μέγεθος.

Η σύνθεση των stream transformers και computers απεικονίζεται στο Σχήμα 4.2. Όπως φαίνεται και στο σχήμα δεν επιτρέπεται η σύνθεση computer-computer. Στο δεξί σχήμα το έντονο βέλος "Control" που συνδέει τον "Computer" με τον "Transformer" αναπαριστά την τιμή επιστροφής του computer

[1] Ο scrambler χρησιμοποιείται για παράδειγμα σε ένα WiFi πομπό ώστε να κάνει XOR τα δεδομένα εισόδου με μία σειρά ψευδοτυχαίων αριθμών και τελικά να σχηματίσει το μεταδιδόμενο σήμα.

όταν τελειώσει τον υπολογισμό του και είναι αυτή που οδηγεί στον επαναπροσδιορισμό της σωλήνωσης. Αφού επιστραφεί η τιμή από τον computer η είσοδος που αρχικά κατευθυνόταν σε αυτόν θα ανακατευθύνει στο επόμενο στάδιο της σωλήνωσης, σε αυτήν την περίπτωση στον "Transformer" στην κάτω δεξιά γωνία του σχήματος. Και οι δύο συνιστώσες εξάγουν τις τιμές εξόδου στην ίδια ροή δεδομένων "Output Stream".




**Σχήμα 4.2:** Σύνθεση Transformer-Transformer (αριστερά), Σύνθεση Computer-Transformer (δεξιά)

### 4.2.2 Σύνθεση στο Μονοπάτι Ελέγχου

Οι τιμές ελέγχου που επιστρέφονται από τους streams computers είναι ο μοναδικός μηχανισμός που προσφέρει η Ziria για τον επαναπροσδιορισμό της διαδικασίας της σωλήνωσης. Στο παράδειγμα αποκωδικοποίησης που έχουμε ήδη αναφερθεί, σε χρόνο εκτέλεσης και κατά τη λήξη της αποκωδικοποίησης της επικεφαλίδας, η επιστρεφόμενη τιμή ελέγχου χρησιμοποιείται για να αρχικοποιηθεί και να καθοριστεί η αποκωδικοποίηση του ωφέλιμου φορτίου ώστε στη συνέχεια να μπορεί να δεχθεί το υπόλοιπο της εισόδου. Η προσέγγιση αυτή διαφέρει αισθητά από τις αντίστοιχες στις υπάρχουσες SDR πλατφόρμες, στις οποίες ο επαναπροσδιορισμός και η αρχικοποίηση διαφορετικών τμημάτων της σωλήνωσης επιτυγχάνεται μέσω μοιραζόμενων καθολικών μεταβλητών, πέρασμα ασύγχρονων μηνυμάτων, ή πρέπει να προγραμματιστεί χειροκίνητα σε χαμηλό επίπεδο με επιπλέον μηνύματα στο μονοπάτι δεδομένων. Όλες οι προαναφερθείσες τεχνικές όμως εμποδίζουν την εύκολη συντηρησιμότητα κι επαναχρησιμοποίηση του κώδικα.

Στη Ziria, αυτός ο τρόπος ελέγχου της ροής εκφράζεται χρησιμοποιώντας τον `seq`-uence combinator. Ως παράδειγμα παραθέτουμε ένα απόσπασμα από τον WiFi δέκτη που έχει υλοποιηθεί πλήρως σε Ziria:

```
seq { (h : HeaderInfo) ← DecodePLCP()
    ; Decode(h) }
```

Ο κώδικας αυτός τρέχει τον stream computer `DecodePLPC` μέχρι αυτός να παράξει μια τιμή ελέγχου `h` τύπου `HeaderInfo`, και στη συνέχεια μεταβαίνει στον stream computer `Decode`. Επειδή η ακολουθία αυτή εκφράζει τη ροή ελέγχου στο πρόγραμμα, ο ακολουθιακός τελεστής `seq` αναφέρεται ως σύνθεση στο μονοπάτι ελέγχου. Το σύστημα τύπων της Ziria εγγυάται ότι σε μια ακολουθία `seq {x ← c1; c2}`, το `c1` είναι πράγματι ένας stream computer και όχι transformer, ενώ το `c2` μπορεί να είναι είτε computer είτε transformer.

Ο απλοποιημένος κανόνας τύπων είναι:

$$\frac{\vdash c1 : Zr\ (C\ c)\ a\ b \qquad (x : c) \vdash c2 :\ Zr\ t\ a\ b}{\vdash \texttt{seq}\{\ x\ \leftarrow c1;\ c2\ \} : Zr\ t\ a\ b}$$

όπου το t είναι είτε `T` είτε `C d`. Αξίζει να σημειωθεί ότι και το `c1` και το `c2` παίρνουν τιμές του ίδιου τύπου `a` και παράγουν τιμές του ίδιου τύπου `b`, παρόλο που προκύπτουν σε διαφορετικές χρονικές στιγμές.

Ο δυναμικός επαναπροσδιορισμός που μας προσφέρεται μέσω του τελεστή `seq` αντικατοπτρίζει επιτυχημένα τη ροή ελέγχου σε πολλά πρωτόκολλα φυσικού στρώματος. Ένα ακόμη τυπικό παράδειγμα είναι αυτό της λήψης ενός WiFi πακέτου, κατά την οποία ο παραλήπτης στην αρχή παίρνει ένα προοίμιο για δείγμα ώστε να είναι σε θέση να εκτιμήσει τα χαρακτηριστικά του καναλιού επικοινωνίας και στη συνέχεια χρησιμοποιεί την εκτίμηση αυτή για να αντιστρέψει τις επιδράσεις του καναλιού και να αποκωδικοποιήσει σωστότερα το πακέτο.

### 4.2.3 Σύνθεση στο Μονοπάτι Δεδομένων

Η Ziria υποστηρίζει επίσης μια πιο συμβατική μορφή σύνθεσης, αυτή της σύνθεσης στο μονοπάτι δεδομένων, στην οποία η ροή δεδομένων εξόδου του ενός μπλοκ γίνεται ροή δεδομένων εισόδου του άλλου μπλοκ. Για παράδειγμα, σε έναν πομπό WiFi η ροή δεδομένων ενός CRC μπλοκ διοχετεύεται σε έναν scrambler, ακολουθούμενος από έναν κωδικοποιητή. Εκφράζουμε τη σύνθεση αυτή με τον τελεστή $\ggg$. Παραθέτουμε και πάλι ένα απόσπασμα από τον WiFi πομπό, για να δώσουμε ένα παράδειγμα σε κώδικα Ziria:

```
crc24(len) ⋙ scrambler() ⋙ encode12()
```

Ο γενικός κανόνας `c1` $\ggg$ `c2` επιτρέπει το πολύ ένα εκ των `c1` και `c2` να είναι stream computer. Αν υπάρχει ένας computer τότε και η συνολική σύνθεση `c1` $\ggg$ `c2` γίνεται computer, η οποία τερματίζει επιστρέφοντας την τιμή επιστροφής του αρχικού. Σε περίπτωση που κι οι δύο συνιστώσες είναι transformers, τότε το ίδιο θα ισχύει και για τη σύνθεσή τους.

Ο απλοποιημένος κανόνας τύπων είναι:

$$\frac{t = t1 \oplus t2 \quad \vdash c1 : Zr\ t1\ a\ b \quad \vdash c2 : Zr\ t2\ b\ c}{\vdash c1 \ggg c2 : Zr\ t\ a\ c}$$

όπου $T \oplus t = t \oplus T = t$. Επιπλέον μόνο ένα εκ των `c1`, `c2` μπορεί να έχει πρόσβαση ανάγνωσης-γραφής σε μοιραζόμενες μεταβλητές, ώστε να εγγυάται ελευθερία σε συνθήκες ανταγωνισμού κατά την παράλληλη εκτέλεση της $\ggg$ σύνθεσης.

### 4.2.4 Παράδειγμα: Σωλήνωση για τον WiFi Δέκτη

Στην υποενότητα αυτή, για να δούμε τον τρόπο με τον οποίο οι προγραμματιστικές αφαιρέσεις που περιγράψαμε συνενώνονται μεταξύ τους και δομούν περίπλοκα συστήματα, παρουσιάζουμε το βασικό μας παράδειγμα που είναι η υλοποίηση ενός WiFi 802.11a/g δέκτη στη Ziria, βασιζόμενη στην αντίστοιχη υλοποίηση σε Sora [6].

```
1  let comp Decode ( h : struct HeaderInfo ) =
2    DemapLimit (0) ⋙
3      if h. modulation ≡ M_BPSK then
4        DemapBPSK () ⋙ DeinterleaveBPSK ()
5      else if h . modulation ≡ M_QPSK then
6        DemapQPSK () ⋙ DeinterleaveQPSK ()
7      else ... — QAM16 , QAM64 cases
8    ⋙ Viterbi (h. coding , h. len ∗8 + 8)
9    ⋙ scrambler ()
10  let comp detectSTS () = removeDC () ⋙ cca ()
11  let comp receiveBits () =
```

```
12      seq { h ← DecodePLCP ()
13          ; Decode (h ) ⋙ check_crc (h. len )
14          }
15    let comp receiver () =
16       seq { det ← detectSTS ()
17           ; params ← LTS ( det . shift )
18           ; DataSymbol ( det . shift ) ⋙
19             FFT () ⋙
20             ChannelEqualization ( params ) ⋙
21             PilotTrack () ⋙
22             GetData () ⋙
23             receiveBits () }
```

Οι κύριες φάσεις της υλοποίησης είναι: η ανίχνευση καναλιού (γραμμή 16), η εκτίμηση καναλιού (γραμμή 17) και η αποδιαμόρφωση και αποκωδικοποίηση πακέτου (γραμμές 22 με 23). Οι γραμμές 18 με 22 κάνουν τη μετατροπή από το πεδίο του χρόνου στο πεδίο συχνότητας και αφαιρούν τις αλλοιώσεις καναλιού.

Η ανίχνευση καναλιού αποφασίζει αν συμβαίνει κάποια WiFi μετάδοση ψάχνοντας για ένα γνωστό προοίμιο στη ροή δεδομένων εισόδου. Το πρώτο μπλοκ, `detectSTS`, αποτελείται από ένα μπλοκ που αφαιρεί το DC μέρος (`removeDC`), ακολουθούμενο από τον κύριο αλγόριθμο ανίχνευσης (clear channel assesment, `cca`). Το κομμάτι αυτό της ανίχνευσης επιστρέφει τις απαραίτητες λειτουργίες χρονισμού στο `det`.

Το τμήμα εκτίμησης καναλιού εκτιμά την επίδραση των φυσικών φαινομένων στο μεταδιδόμενο σήμα (μπλοκ `LTS`). Η εκτίμηση του καναλιού επιστρέφεται σαν τιμή ελέγχου στο `params`.

Η αποκωδικοποίηση `OFDM` μεταρέπει τα ληφθέντα δεδομένα από το πεδίο του χρόνο στο πεδίο της συχνότητας (γραμμές 18 με 22), και ο `receiver` αφαιρεί τις επιδράσεις καναλιού, όπως αυτές εκτιμήθηκαν στην προηγούμενη φάση, με τη χρήση του `ChannelEquazitation` ακολουθούμενου από το `PilotTrack`. Τέλος περνάμε τα επεξεργασμένα σύμβολα στο `receiveBits` για αποδιαμόρφωση και αποκωδικοποίηση.

Μέσω του WiFi παραδείγματος που δώσαμε τονίζονται μερικά βασικά χαρακτηριστικά της υψηλού επιπέδου γλώσσας υπολογισμών της Ziria. Κατά πρώτον, η σύνθεση των επιμέρους συνιστωσών είναι ξεκάθαρη ενώ ο κώδικας είναι καλά δομημένος, σύντομος κι έχει εγγυημένα σωστούς τύπους μέσω της μεταγλώττισης. Κατά δεύτερον, οι παράμετροι διαμόρφωσης περνούν με σαφή τρόπο από και προς τις συνιστώσες μέσω της δομής `seq`.

### 4.2.5 Υλοποίηση των Μπλοκ Επεξεργασίας

Ο σχεδιασμός της Ziria διαχωρίζει δύο επίπεδα:

**Υψηλότερου επιπέδου γλώσσα υπολογισμών**
: Η βασική καινοτομία της Ziria έγκειται σε αυτό το επίπεδο όπου η **monadic** γλώσσα υπολογισμών (*computation language*) χρησιμοποιείται για να προσδιορίσει τη ροή ελέγχου ενός PHY προγράμματος, να συνθέσει σωληνώσεις για επεξεργασία ροών δεδομένων και να παράγει ένα αποδοτικό μοντέλο εκτέλεσης. Η υπογλώσσα αυτή αποτελείται από δύο διαφορετικούς τύπους υπολογισμών τους stream transformers και τους stream computers, οι οποίοι αναλύθηκαν παραπάνω (4.2.1).

**Χαμηλότερου επιπέδου προστακτική γλώσσα**
: Η Ziria παρέχει μία χαμηλότερου επιπέδου προστακτική γλώσσα εκφράσεων (*expression language*) που χρησιμοποιείται για την κωδικοποίηση βασικών προστακτικών υπολογισμών. Αυτό

την καθιστά κατάλληλη για την υλοποίηση αλγορίθμων ψηφιακής επεξεργασίας σήματος και τη διαχείριση δεδομένων μέσα στα υποσυστήματα της σωλήνωσης. Η υπογλώσσα αυτή προσομοιάζει γλώσσες που χρησιμοποιούνται ήδη από ειδικούς στον τομέα, όπως C και Matalab, ενώ ταυτόχρονα είναι μια ισχυρά τυποποιήσιμη (strongly typed) γλώσσα που υλοποιεί ένα σύνολο χαρακτηριστικών χαμηλού επιπέδου βελτιστοποιήσεων προσεκτικά επιλεγμένων ώστε να διατηρεί την εκφραστικότητά της και παράλληλα να εγγυάται αποδοτική μεταγλώττιση.

### 4.2.6 Συντακτικό της Ziria

Οι πιο σημαντικές συντακτικές μορφές της γλώσσας δίνονται παρακάτω στο Σχήμα 4.3.

**Stream computations and imperative expressions**

| | | |
|---|---|---|
| $c ::= x \leftarrow c;\ c$ | | Bind |
| $\mid c >\!\!>\!\!>\ c$ | | Arrow |
| $\mid$ letref $\overline{x{:}\tau := v}$ in $c$ | | Scoped shared state |
| $\mid$ letfun $g(\overline{x{:}\tau}) = m$ in $c$ | | Local function |
| $\mid$ letfunc $f(\overline{x{:}\tau}) = c_1$ in $c_2$ | | Local computation |
| $\mid c(\overline{e})$ | | Call computation function |
| $\mid$ take | | Move to control channel |
| $\mid$ emit $e$ | | Emit on data channel |
| $\mid$ return $m$ | | Return on control channel |
| $\mid$ repeat $c$ | | Repeat $c$ indefinitely |
| $\mid$ map$(f)$ | | Map over input stream |
| $\mid$ if $e$ then $c_1$ else $c_2$ | | Conditionals |
| $\mid \ldots$ | | |
| $e, m ::= x \mid v \mid x := e \mid m_1; m_2 \mid f(\overline{e}) \mid \ldots$ | | Expressions |
| $v ::=$ unit $\mid i \mid \ldots$ | | Values |

**Computation and expression types**

| | | |
|---|---|---|
| $\xi ::= \sigma \mid \overline{\tau} \to \sigma$ | | Computation types |
| $\sigma ::=$ ST $(\mathsf{C}\ \tau)\ \tau\ \tau \mid$ ST T $\tau\ \tau$ | | Computers/Transformers |
| $\theta ::= \tau \mid \overline{\tau} \to \tau$ | | Expression types |
| $a, b, \tau, \nu ::=$ unit $\mid$ int $\mid$ complex $\mid \ldots$ | | Base types |

**Σχήμα 4.3:** Συντακτικό της Ziria

Χρησιμοποιούμε το `c` για να δηλώσουμε computations και το `e, m` για να δηλώσουμε εκφράσεις από το προστακτικό τμήμα της γλώσσας. Έχουμε ως σύμβαση να χρησιμοποιούμε το `m` για τα statements (π.χ. `y:=y+1`) και το `e` για τις expressions (π.χ. `y+42`). Τυπικά τα statements είναι είναι απλά εκφράσεις που επιστρέφουν unit.

Οι υπολογισμοί (computations) `c` περιλαμβάνουν τους `seq` και $\ggg$ τελεστές, συνθήκες, bindings για mutable μεταβλητές, συναρτήσεις, υπολογισμούς και κλήσεις.

Επιπλέον, η Ziria παρέχει ένα σύνολο πρωταρχικών βασικών εντολών (primitives):

- `emit e`. Ένας stream *computer* που υπολογίζει το όρισμά του κι εκλύει την παραγόμενη τιμή στη ροή δεδομένων εξόδου. Είναι computer αφού τερματίζει μόλις εκλύσει την τιμή, επιστρέφοντας `unit` σαν τιμή ελέγχου.

- `take`. Ένας stream *computer* που παίρνει ένα στοιχείο από τη ροή δεδομένων εισόδου και το επιστρέφει σαν την τιμή ελέγχου του. Το παράδειγμα που ακολουθεί παίρνει μια τιμή x από τη ροή δεδομένων εισόδου και εκλύει την `x+1`:

```
seq { (x : int) ← take
    ; emit (x+1) }
```

- `do m` and `return e`. Ανάγουν μία `m` ή `e` έκφραση από το χαμηλού επιπέδου προστακτικό τμήμα στη γλώσσα υπολογισμών. Είναι και οι δύο computers που εκτελούν τα ορίσματά τους και επιστρέφουν το τελικό αποτέλεσμα σαν τιμή ελέγχου. Για παράδειγμα, δοθέντος μιας mutable μεταβλητής `y`, το παρακάτω τμήμα κώδικα, θα πάρει μια τιμή εισόδου, θα ανανεώσει το `y` και θα εκλύσει το `y+1`:

```
seq { (x : int) ← take
    ; do { y := y+x+1; }
    ; emit (y+1) }
```

  Οι βασικές εντολές `do` και `return` έχουν ίδια σημασιολογία, ωστόσο για διευκόλυνση του προγραμματιστή χρησιμοποιούμε το `do` για προστακτικό κώδικα που επιστρέφει `unit`, π.χ. `y:=y+1` και το `return` για εκφράσεις που θα επιστρέψουν μια τιμή, π.χ. `return (y+1)`.

- `repeat c`. Ένας stream *transformer* όπου εκτελεί τον stream *computer* `c`, όταν ο `c` τερματίσει, η `repeat` τον επαναρχικοποιεί και επανεκκινεί τη διαδικασία. Ουσιαστικά υλοποιεί αποδοτικά το `seq{c; c; ...}`. Δίνουμε για παράδειγμα ένα τμήμα κώδικα που φιλτράρει και διώχνει όλα τα στοιχεία μηδενικής τιμής από τη ροή δεδομένων εισόδου του:

```
repeat { (x : int) ← take
       ; if x ≡ 0 then return ()
         else emit x }
```

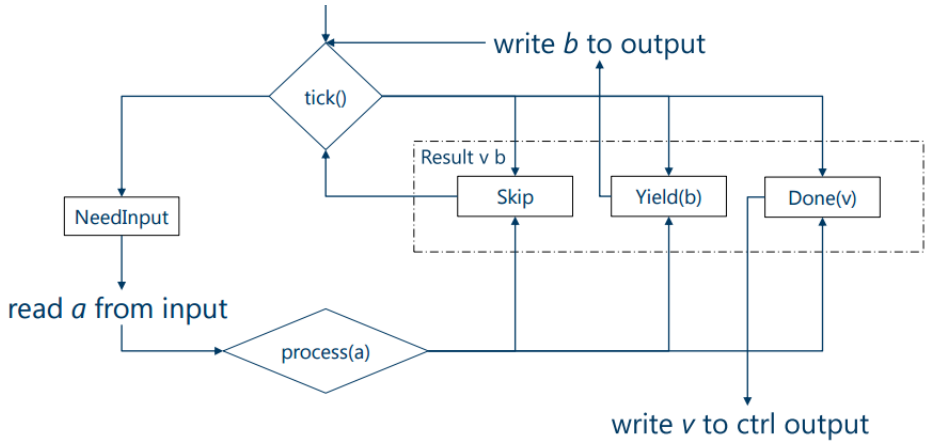
Η Ziria επίσης παρέχει μερικούς τελεστές για την επανάληψη υπολογισμών computer `n` φορές, για mapping συναρτήσεων πάνω στη ροή δεδομένων εισόδου κ.ά. Οι κανόνες τύπων για τις σημαντικές βασικές εντολές συνοψίζονται παρακάτω:

$$
\begin{array}{rll}
\vdash \mathsf{take} : & \forall ab.\ \mathsf{Zr}\ (\mathsf{C}\ a)\ a\ b & \\
\vdash \mathsf{emit}\ e : & \forall a.\ \mathsf{Zr}\ (\mathsf{C}\ \mathsf{unit})\ a\ \tau, & \text{if } \vdash e : \tau \\
\vdash \mathsf{return}\ e : & \forall ab.\ \mathsf{Zr}\ (\mathsf{C}\ \tau)\ a\ b, & \text{if } \vdash e : \tau \\
\vdash \mathsf{map}\ f : & \mathsf{Zr}\ \mathsf{T}\ \tau\ \sigma, & \text{if } \vdash f : \tau \to \sigma \\
\vdash \mathsf{repeat}\ c : & \mathsf{Zr}\ \mathsf{T}\ \tau\ \sigma, & \text{if } \vdash c : \mathsf{Zr}\ (\mathsf{C}\ \mathsf{unit})\ \tau\ \sigma
\end{array}
$$

### 4.2.7 Εκτέλεση Σωληνώσεων Ziria

Τα προγράμματα Ziria είναι προορισμένα να τρέχουν σε απλούς επεξεργαστές εμπορίου και για αυτό το λόγο η καλή επίδοση είναι πολύ σημαντικό κομμάτι του σχεδιασμού. Το *intrathread* μοντέλο εκτέλεσης της Ziria καλείται να καλύψει αυτήν την ανάγκη εκτελώντας την επεξεργασία με χαμηλή συνολική καθυστέρηση και αποφεύγοντας το buffering που εισάγεται από τη μεταγλώττιση ακόμη και για προγράμματα που χρησιμοποιούν $\ggg$.

Ένα βασικό χαρακτηριστικό του σχεδιασμού είναι ότι υπάρχουν υπολογισμοί που μπορούν αμέσως να παράγουν έξοδο και να την ωθήσουν (push) στη ροή εξόδου (`Yield b`) — με πιο απλό παράδειγμα αυτό του `emit e` — και υπολογισμοί που πρέπει πρώτα να τραβήξουν (pull) μια είσοδο ώστε να είναι σε θέση να εκτελέσουν (`NeedInput`) — με πιο απλό παράδειγμα αυτό του `take`. Σε περίπλοκα προγράμματα Ziria τα μπλοκ επεξεργασίας θα χρειαστεί και να ωθήσουν και να τραβήξουν δεδομένα




**Σχήμα 4.4:** Κύριος βρόχος επανάληψης μεταγλώττισης

κατά τη διάρκεια της εκτέλεσής τους. Συνεπώς, κάθε Ziria υπολογισμός μεταγλωττίζεται σε ένα ζεύγος τμημάτων κώδικα, που ονομάζονται `tick` και `proc`. Το `tick` τμήμα κώδικα αποφασίζει αν ένας υπολογισμός έχει ένα αποτέλεσμα άμεσα διαθέσιμο, και αν υπάρχει τότε άμεσα το δίνει στο `proc` τμήμα της μεταγενέστερης συνιστώσας της ροής επεξεργασίας. Από την άλλη πλευρά, αν χρειάζεται να τραβήξουμε κάποια είσοδο το `tick` μεταπηδά στο `tick` τμήμα κώδικα της προγενέστερης συνιστώσας. Ένα `proc` τμήμα κώδικα καταναλώνει μια δοσμένη είσοδο και μπορεί να δώσει έξοδο σε μεταγενέστερο `proc` τμήμα κώδικα. Για να δώσουμε μια πιο διαισθητική εικόνα του μοντέλου εκτέλεσης θεωρούμε ότι κάθε computer `Zr (C v) a b` μεταγλωττίζεται σε τρεις συναρτήσεις:

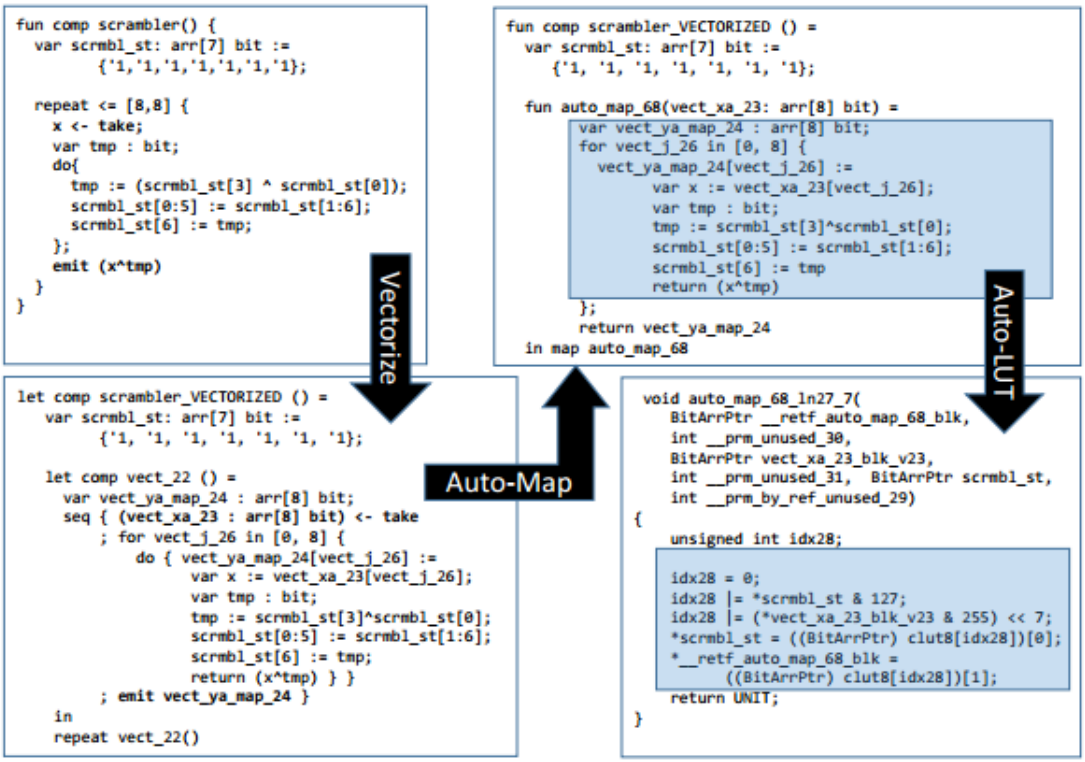
```
tick :: () → Zr (Result v b + NeedInput)
process :: a → Zr (Result v b)
init :: () → Zr ()

data Result v b ::= Skip | Yield b | Done v
```

Οπότε το κύριο έργο της μεταγλώττισης είναι να συνθέσει τις `tick()`, `process()`, `init()` από τα μικρότερα μπλοκ σε μία τελική κάθε είδους. Στο Σχήμα 4.4 παρουσιάζουμε τις δυνατές μεταβάσεις για ένα κύριο βρόχο επανάληψης της μεταγλώττισης:

Η παρακάτω εικόνα παρουσιάζει τον κύριο βρόχο επανάληψης μεταγλώττισης για έναν computer. Υπάρχουν δύο λεπτά σημεία που αξίζουν προσοχής:

- Τα `tick` και `proc` μπορούν να αποφασίσουν αν ο υπολογισμός, σε περίπτωση που μιλάμε για stream computers, πρέπει να τερματίσει. Αυτό θα συμβεί όταν ο `c1` τερματίσει σε έναν `seq { x ← c1; c2 }` υπολογισμό. Για αυτόν τον λόγο, ο `seq` μεταγλωττίζεται με ένα switchtable που επιλέγει ποιο από τα δύο `tick`/`proc` τμήματα είναι ενεργό σε οποιαδήποτε σημείο στο χρόνο. Ο τερματισμός του `c1` ενεργοποιεί τα `tick` /`proc` τμήματα που σχετίζονται με τον `c2`.
- Τα `tick` και `proc` για το `c1` ⋙ `c2` διαμορφώνονται ως `tick` για το `c2` και `proc` για το `c1`, αντίστοιχα. Αυτό μας δείχνει ότι η σωλήνωση απορροφά από τα δεξιά, συνεπώς δεν υπάρχει ανάγκη για ουρές με μεταβλητό μέγεθος μεταξύ του `c1` ⋙ `c2` κι επίσης ότι οι τιμές ωθούνται αμέσως μόλις γίνουν διαθέσιμες, μειώνοντας τη συνολική καθυστέρηση.

Ο μεταγλωττιστής της Ziria πέρα από αυτό το βασικό σχήμα μεταγλώττισης που περιγράψαμε κάνει επίσης στατικές αποφάσεις scheduling, όπως να εξαλείφει `tick` τμήματα κώδικα για συνιστώσες που ποτέ δεν θα μπορέσουν αμέσως να παράγουν έξοδο αλλά πάντα χρειάζονται είσοδο (π.χ. `map f` ή `take`). Με αυτή τη βελτιστοποίηση μειώνουμε την επιβάρυνση που προκαλεί το `tick` στο μονοπάτι δεδομένων.




**Σχήμα 4.5:** Σύνολο βελτιστοποιήσεων: αρχικός scrambler, auto-vectorized, auto-mapped, και παραγόμενος C κώδικας με LUT.

### 4.2.8 Παραλληλοποίηση Σωλήνωσης

Μία μεμονωμένη `c1` $\ggg$ ... $\ggg$ `cn` σύνθεση μπορεί εύκολα να κατευθυνθεί σε πολλούς πυρήνες εισάγοντας interthread ουρές και μεταγλωττίζοντας κάθε επιμέρους συνιστώσα σύμφωνα με το προηγούμενο intrathread μοντέλο. Μπορούμε να γενικεύσουμε αυτήν την παρατήρηση για προγράμματα όπου ο τελευταίος υπολογισμός σε μια ακολουθία `seq` είναι μια σύνθεση στο μονοπάτι δεδομένων:

```
seq { x ← c0
    ; c1 ⋙ ... ⋙ cn }
```

Στη Ziria υπάρχει δυνατότητα να αντικαταστήσουμε οποιοδήποτε από τους $\ggg$ τελεστές με μια παραλλαγή τους την $|\ggg|$, η οποία δείχνει ρητά το διαχωρισμό σε πολλούς πυρήνες.

## 4.3 Βελτιστοποιήσεις

Ο μεταγλωττιστής της Ziria υλοποιεί μια σειρά βελτιστοποιήσεων που στόχο έχουν να εξαλείψουν τους υπολογισμούς που υπάρχουν στις εκφράσεις, να μειώσουν την αντιγραφή μνήμης και να συνδυάσουν διάφορους υπολογισμούς ώστε τελικά να προκύψει μια πιο αποδοτική υλοποίηση. Οι βελτιστοποιήσεις αυτές βασίζονται κυρίως σε ήδη γνωστές τεχνικές, έχουν όμως καθοριστική σημασία για την παραγωγή αποδοτικού κώδικα. Αναφέρουμε παρακάτω τις πιο σημαντικές:

**Διανυσματοποίηση**

Μία κεντρική βελτιστοποίηση στον μεταγλωττιστή της Ziria είναι η *διανυσματοποίηση* (vectorization). Επαναγράφει τις συνιστώσες της σωλήνωσης έτσι ώστε αντί να παίρνουν τιμές τύπου

$a$ και να δίνουν τιμές τύπου $b$, να παίρνουν τιμές τύπου $array[d_{in}]$ $a$ και να δίνουν τιμές τύπου $array[d_{out}]$ $b$ για κατάλληλα $d_{in}$ και $d_{out}$. Με αυτό τον τρόπο επιτυγχάνουμε τη συστηματική μετατροπή των σωληνώσεων στο μονοπάτι δεδομένων ώστε να λειτουργούν σαν διανύσματα παρά σαν βαθμωτές τιμές. Είναι σημαντικό να τονιστεί ότι το σύνολο των δυνατών διανυσματοποιήσεων μιας συνιστώσας εξαρτάται από τη θέση της συνιστώσας μέσα στη σωλήνωση αλλά και από το πλάτος δεδομένων των γειτονικών τμημάτων. Για αυτό το λόγο, ο αποκεντρωμένος αλγόριθμος που χρησιμοποιείται χωρίζεται σε δύο φάσεις: Η πρώτη είναι μια top-down διαδικασία όπου θέτει, βάσει του συνολικού αριθμού των τιμών που κάνουν take και emit, όλες τις δυνατές διανυσματοποιήσεις για τους computers και τους transformers. Ενώ η δεύτερη είναι μια bottom-up διαδικασία όπου συνθέτει το σύνολο των δυνατών στοιχείων που διανυσματοποιούνται μαζί και δημιουργεί ξανά την assembly των στοιχείων αυτών για την τελική σωλήνωση. Η εναλλακτική λύση της χειροκίνητης διανυσματοποίησης πάνω στον κώδικα απορρίπτεται σαν μέθοδος αφού εκτός ότι παρεκκλίνει διαισθητικά από τις προδιαγραφές των πρωτοκόλλων και παρεμποδίζει την επαναχρησιμοποίηση κώδικα απαιτεί πολύ προσοχή από τον προγραμματιστή για να γίνει σωστά.

**Στατική Χρονοδρομολόγηση**

Ένα σύνολο βελτιστοποιήσεων εστιάζεται στη μετατροπή, όποτε αυτή είναι εφικτή, ακολουθίες υπολογισμών που περιλαμβάνουν `seq` και $\ggg$ σε προστακτικό κώδικα. Πιο συγκεκριμένα, μια πολύ χρήσιμη μορφή βελτιστοποίησης που ανήκει σε αυτήν την κατηγορία είναι το καλούμενο *auto-mapping*. Το σχήμα 4.5 παρουσιάζει ένα παράδειγμα auto-mapping: Δημιουργούμε μια ξεχωριστή τοπική συνάρτηση με τον βασικό κώδικα του scrambler, όπου έχει ήδη υποστεί διανυσματοποίηση, και αντικαθιστούμε το `repeat` μπλοκ με κλήση στη `map`. Για το σκοπό αυτό έχει σχεδιαστεί ένα ολόκληρο σύνολο βελτιστοποιήσεων, όπως το inlining, η αλλαγή θέσης σε let ορισμούς και συνθήκες και η αντικατάσταση βρόχων που βρίσκονται στο επίπεδο υπολογισμών με αντίστοιχους στο επίπεδο εκφράσεων. Το auto-mapping είναι ένα είδος στατικής χρονοδρομολόγησης αφού ο γεννήτορας κώδικα γνωρίζει ότι το `map` δεν μπορεί να προχωρήσει χωρίς να καταναλώσει είσοδο. Για παράδειγμα μια μακρυά ακολουθία από $\ggg$ συνθέσεις `map` θα ωθεί μόνο προς τα κάτω τιμές χωρίς να χρειαστεί να χρησιμοποιήσει τα `tick` μπλοκ.

**Παραγωγή Lookup Πινάκων**

Πολλές PHY λειτουργίες έχουν οριστεί να δουλεύουν σε bit επίπεδο ωστόσο μια άμεση υλοποίηση σε επεξεργαστές με πλατύ δίαυλο δεδομένων είναι μη αποδοτική. Μια συνηθισμένη βελτιστοποίηση είναι η αναγνώριση τέτοιων λειτουργιών και η επιτάχυνσή τους αποθηκεύοντας τα αποτελέσματά τους σαν lookup tables (LUTs), όπως για παράδειγμα φτιάχνουν χειροκίνητα η Sora και το GNURadio. Ωστόσο το γράψιμο συναρτήσεων που χρησιμοποιούν LUTs είναι επίπονο και έχει σαν αποτέλεσμα κώδικα όπου είναι δύσκολο να συντηρηθεί. Ο μεταγλωττιστής της Ziria λύνει το πρόβλημα αυτό ανιχνεύοντας αυτόματα τμήματα κώδικα που επιδέχονται υλοποίηση με LUT, για παράδειγμα εκφράσεις αρκετά περίπλοκες ώστε να αξίζουν να μετατρέψουν σε LUT αλλά που ταυτόχρονα δεν έχουν πολύ μεγάλο μέγεθος. Για παράδειγμα, η auto-mapped εκδοχή του scrambler όπως φαίνεται και στην εικόνα 4.5 μεταγλωττίζεται έτσι ώστε να χρησιμοποιεί ένα LUT με δείκτες που αποτελούνται από την τρέχουσα είσοδο (8 bits) και την κατάσταση του scrambler (7 bits), οπότε συνολικά έχουμε $2^{15}$ εγγραφές. Ωστόσο οι μετατροπές που γίνονται απαιτούν λεπτούς χειρισμούς για τη ρύθμιση της απόδοσης από το μεταγλωττιστή της Ziria.

Το σύνολο των βελτιστοποιήσεων αυτών επιτρέπουν στη Ziria να είναι η πρώτη υψηλού επιπέδου SDR πλατφόρμα με εξαιρετικά αποδοτικό μοντέλο εκτέλεσης.

## 4.4 Ziria και Συναρτησιακός Προγραμματισμός

Ο σχεδιασμός της Ziria και οι σημαντικοί τελεστές της αντλούν ιδέες από τα monads [16] και τα arrows [17]. Μία παραλλαγή του τελεστή `seq`, το επονομαζόμενο "switch", χρησιμοποιείται στη Yampa [18], ένα δημοφιλές functional reactive programming (FRP) framework, που στερείται όμως αποδοτικό μοντέλο εκτέλεσης. Η βιβλιοθήκη των Haskell Pipes περιλαμβάνει monad transformers, οι οποίοι παρέχουν υψηλού επιπέδου λειτουργικότητα παρόμοια με τους τελεστές για τις ροές δεδομένων της Ziria. Τέλος η σημασιολογία των `tick` και `process` προσομοιάζει το push-pull μοντέλο για ροές δεδομένων, δημοφιλές σε functional reactive programming (FRP) frameworks [19, 20].

# Κεφάλαιο 5

# Ενσωμάτωση της Ziria στην Haskell

## 5.1 Υλοποίηση Ενσωμάτωσης

### 5.1.1 Γενικά Στοιχεία

Σε αυτήν την ενότητα παρουσιάζουμε την προσέγγιση που έγινε για την υλοποίηση ενός διερμηνέα για τη Ziria στη Haskell. Για τη σημασιολογική ανάλυση της γλώσσας δημιουργούμε τον τύπο `Zir a b t`, όπου `a`, `b` ο τύπος της ροής δεδομένων εισόδου κι εξόδου αντίστοιχα, ενώ `t` είναι ο τύπος της τιμής επιστροφής. Παραθέτουμε παρακάτω τον ορισμό του τύπου `Zir a b t`, που αποτελεί τον βασικό τύπο υπολογισμών στην ενσωμάτωσή μας:

```
type M = IO
data F a b t = Yield t b | NeedInput (a → t)

-- Zir a b t ~= M (t    + (Zir a b t, b) + (a → Zir a b t))
--                 Done    Yield            NeedInput
type Zir a b t = FreeT (F a b) M t
```

**Κώδικας 5.1:** Κύριος Τύπος για την Ενσωμάτωση

Στην Ziria, όπως περιγράψαμε και στην υποενότητα 4.2.7, οι υπολογισμοί μπορούν να χωριστούν σε αυτούς που είναι σε θέση απευθείας να παράγουν έξοδο και να την ωθήσουν στη ροή δεδομένων εξόδου και σε αυτούς που χρειάζεται να τραβήξουν μια είσοδο από τη ροή δεδομένων εισόδου για να μπορούν να εκτελεστούν.

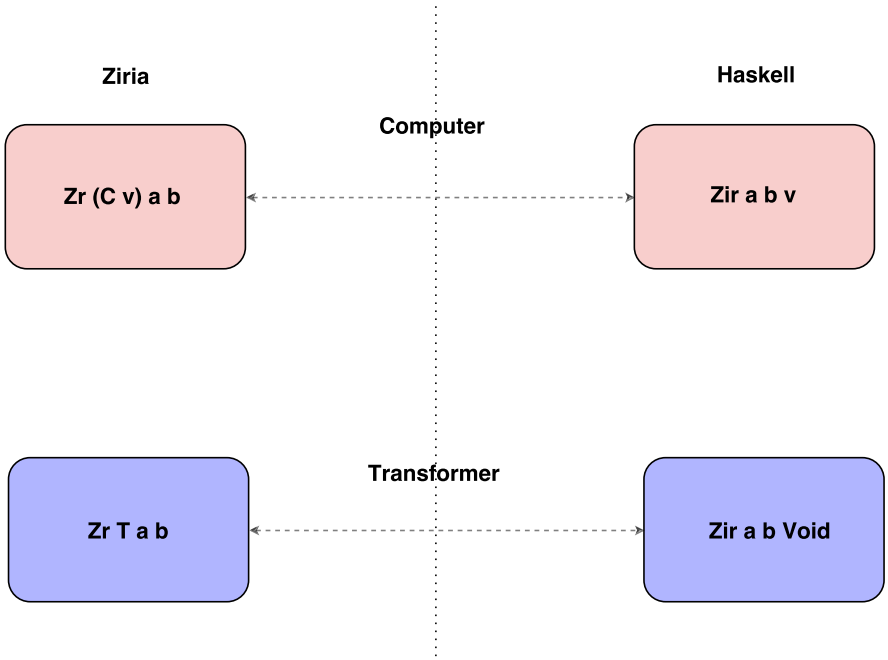
Ουσιαστικά επιτρέπουμε στον τύπο `Zir` να καταδεικνύει μία από τις δυνατές καταστάσεις που βρίσκεται ο εκάστοτε υπολογισμός:

- `Done`: ο υπολογισμός έχει τελειώσει και επιστρέφει `t`
- `Yield t b`: υπάρχει αποτέλεσμα στο `b` έτοιμο να ωθηθεί στη ροή δεδομένων εξόδου
- `NeedInput (a → t)`: ο υπολογισμός για να συνεχίσει χρειάζεται είσοδο στο `a`

Για την σύνθεση του τύπου αυτού χρησιμοποιείται ένας free monad transformer 2.2.4 με base functor το `F a b`, inner monad το IO και τύπο επιστροφής το `t`. Μέσω της χρήσης του free monad είμαστε σε θέση να δομήσουμε μια εμφωλευμένη σειρά από `Zir a b t` που αναπαριστά ουσιαστικά όλη τη δομή του προγράμματός μας μέσω των τριών καταστάσεων που περιγράψαμε παραπάνω, χωρίς όμως να πράττει κάποιου είδους υπολογισμό πάνω σε αυτές.

Σημειώνεται εδώ μια σημαντική υπόθεση εργασίας που έγινε για την ενσωμάτωση:

*Οι transformers θεωρούνται computers με τιμή επιστροφής `Void`*, Σχήμα 5.1.1. Ωστόσο επειδή ο διαχωρισμός γίνεται πιο εύκολα σαφής θα συνεχίσουμε να διατηρούμε λεκτικά τους όρους computer και transformer για το υπόλοιπο το κειμένου.




**Σχήμα 5.1:** Computers - Transformers σε Haskell

Οι τύποι για τις βασικές εντολές που αντιστοιχούν άμεσα στους τύπους που παραθέσαμε για τη Ziria 4.2.6 διαμορφώνονται ως εξής:

$$\begin{aligned} \texttt{ztake} &:: \texttt{Zir}\ a\ b\ a \\ \texttt{zemit} &:: b \to \texttt{Zir}\ a\ b\ () \\ \texttt{zmap} &:: (a \to b) \to \texttt{Zir}\ a\ b\ Void \\ \texttt{zrepeat} &:: \texttt{Zir}\ a\ b\ () \to \texttt{Zir}\ a\ b\ Void \end{aligned}$$

### 5.1.2 Σύνθεση στο Μονοπάτι Ελέγχου

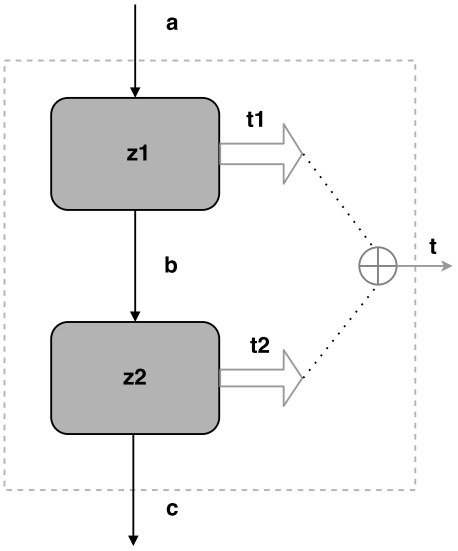
Για την υλοποίηση του τελεστή $\ggg$ που επιτελεί τη σύνθεση στο μονοπάτι ελέγχου χρειάστηκε να ορίσουμε μια καινούρια typeclass, την `Trivsum`, η οποία αναπαριστά έναν ιδιαίτερο τύπο trivial logical sum με τη λειτουργικότητα που περιγράψαμε στο 4.2.3. Όμως, όπως αναφέρθηκε παραπάνω έχουμε ουσιαστικά μόνο computers στην υλοποίησή μας, αυτό μας οδηγεί να επιλέξουμε τον `Either` τελεστή για τη σύνθεση δύο computer μεταξύ τους, όπου το `Void` λειτουργεί ως το μηδενικό στοιχείο του `TrivSum`, Σχήμα 5.1.2:

```
class TrivSum a b c | a b → c where
  squash :: Either a b → c

instance {-# OVERLAPPING #-} TrivSum Void Void Void where
  squash _ = undefined

instance TrivSum t Void t where
  squash (Left x) = x
  squash _ = undefined
```

```
instance TrivSum Void t t where
  squash (Right x) = x
  squash _ = undefined
```

**Κώδικας 5.2:** TrivSum a b c

Σε περίπτωση της `c1 ⋙ c2` σύνθεσης ο υπολογισμός οδηγείται - ορίζεται από το `c2`. Όταν δεν υπάρχει διαθέσιμη είσοδος `a` για τη συνολική σύνθεση ρωτάμε (poke) το `c2` να μας πει την κατάσταση που βρίσκεται. Σε περίπτωση που ο `c2` είναι computer κι έχει τερματίσει με τιμή επιστροφής `t2` τερματίζει ο υπολογισμός του arr επιστρέφοντας συνολικά `t2` (γρ. 10-11). Διαφορετικά αν το `c2` βρίσκεται σε `tick` (Nothing) κατάσταση, ελέγχουμε αν υπάρχει άμεσα διαθέσιμο αποτέλεσμα κι αν υπάρχει το ωθούμε στο `proc` μπλοκ κώδικα της παρακάτω συνιστώσας ενώ αν δεν υπάρχει σημαίνει ότι το `c2` χρειάζεται και προκαλεί `tick` στο `c1` (γρ. 13, γρ. 24). Αν το `c2` είναι σε `proc` σημαίνει ότι υπάρχει διαθέσιμη είσοδος `b`, την οποία καταναλώνει (γρ. 14-15).




**Σχήμα 5.2:** Σύνθεση στο Μονοπάτι Δεδομένων

Αρχικά εξετάζουμε τον `z2` και απαιτούμε από αυτόν το επόμενο στοιχείο (γρ. 11), αν ο `z2` είναι computer που έχει τερματίσει με τιμή επιστροφής `x`, τότε ολόκληρη η σύνθεση επιστρέφει με αυτήν την τιμή επιστροφής. Αν ο `z2` είναι σε θέση να παράγει έξοδο, χωρίς να χρειάζεται είσοδο (γρ. 12) συνεχίζουμε με τον υπολογισμό του `z2`. Διαφορετικά ο `z2` χρειάζεται να τραβήξει είσοδο από τον `z1` (γρ. 13) και ο έλεγχος του προγράμματος πηγαίνει upstream. Αντίστοιχα για τον `z1` έχουμε την περίπτωση που έχει τερματίσει (γρ. 20) και συμπεριφέρεται όπως ο `z2`, την περίπτωση όπου ο `z1` έχει έτοιμο να δώσει ένα output downstream (γρ.21), είσοδο δηλαδή για τον `z1` και τέλος την περίπτωση που ο `z1` χρειάζεται είσοδο για να συνεχίσει (γρ.22).

```
1 (⋙) :: TrivSum t1 t2 t ⇒ Zir a b t1 → Zir b c t2 → Zir a c t
2 z1 ⋙ z2 = az1 `zpipe` az2 >>= return . squash
3   where az1 = z1 >>= return . Left
4         az2 = z2 >>= return . Right
5
6 zpipe :: Zir a b t → Zir b c t → Zir a c t
7 {-# INLINE zpipe #-}
8 az1 `zpipe` az2 = pull az1 az2
```

```
  where pull :: Zir a b t → Zir b c t → Zir a c t
        pull z1 z2 = FreeT $ do
          v2 ← runFreeT z2
          case v2 of
            Pure x             → return $ Pure x
            Free (Yield z2' o) → return $ Free $ Yield (z1 `zpipe` z2') o
            Free (NeedInput g) → runFreeT $ push g z1     -- Go upstream!
        push :: (b → Zir b c t) → Zir a b t → Zir a c t
        push f z1 = FreeT $ do
          v1 ← runFreeT z1
          case v1 of
            Pure x             → return $ Pure x
            Free (Yield z1' m) → runFreeT $ pull z1' (f m) -- Push downstream!
            Free (NeedInput g) → return $ Free $ NeedInput (push f . g)
```

**Κώδικας 5.3:** Ενσωμάτωση του ⋙ Τελεστή

### 5.1.3 Διερμηνέας για τη Ziria

Ο τύπος `Sem a b t`, δημιουργήθηκε για τη διερμηνεία του προγράμματος, λαμβάνει μία λίστα εισόδου κι επιστρέφει μέσα σε ένα `IO Monad` μια τούπλα με πρώτο στοιχείο την τιμή του computation και δεύτερο μια λίστα με τη ροή δεδομένων εξόδου:

```
type Sem a b t = [a] → M (Maybe t, [b])
```

Η συνάρτηση `zrun`, είναι η συνάρτηση που καλούμε όταν θέλουμε να τρέξουμε ένα πρόγραμμα γραμμένο στην ενσωματωμένη Ziria, ο τύπος της είναι:

```
zrun :: Zir a b t → Sem a b t
-- actually
-- zrun :: Zir a b t → [a] → M (Maybe t, [b])
zrun = zwalk []
```

Οπότε η συνάρτηση καλείται με το πρόγραμμα και τη λίστα εισόδου π.χ. `zrun ex42 [0,1,1]`. Για την υλοποίησή της χρησιμοποιούμε μια βοηθητική συνάρτηση τη `zwalk`, την οποία στην αρχή καλούμε με κενό output stream:

```
zwalk :: [b] → Zir a b t → Sem a b t
{-# INLINE zwalk #-}
zwalk acc z inp = do
  v ← runFreeT z
  case v of
    Pure x             → return (Just x, reverse acc)
    Free (Yield az o)  → zwalk (o : acc) az inp
    Free (NeedInput g) → case inp of
                           (x : xs) → zwalk acc (g x) xs
                           []       → return (Nothing, reverse acc)
```

**Κώδικας 5.4:** Διερμηνέας για τη Ziria

Στο acc υπάρχει το output stream μέχρι τώρα (ανάποδα), στο z το υπόλοιπο πρόγραμμα και στο inp το υπόλοιπο input stream. Με το `runFreeT` μπαίνουμε στο εσωτερικό του `free monad`, αν μας επιστρέψει `Pure x` (γρ. 6), τότε το πρόγραμμά είναι computer με την αρχική έννοια, έχει φτάσει στο τέλος κι επιστρέφει με την τιμή επιστροφής `Just x`, την τιμή επιστροφής του computer και το συνολικό output stream αντεστραμμένο, γιατί κατά τη δημιουργία του βάζουμε μπροστά τις πιο πρόσφατες τιμές. Σε περίπτωση που επιστρέψει `Free (Yield az o)`, δίνεται μια τιμή `o` που προστίθεται στο output stream (γρ. 7). Διαφορετικά, χρειάζεται να καταναλώσουμε είσοδο για να συνεχίσουμε (γρ. 8), όπου υπάρχουν δύο περιπτώσεις:

- Αν υπάρχει διαθέσιμη είσοδος, την καταναλώνουμε και τη δίνουμε στο πρόγραμμα για να συνεχίσει `(g x)` (γρ. 9).
- Αν δεν υπάρχει είσοδος, το πρόγραμμα αναγκαστικά τερματίζει την εκτέλεση επιστρέφοντας την τιμή ελέγχου `Nothing` και το output stream που έχει διαμορφωθεί μέχρι τώρα στον accumulator.

### 5.1.4 Μοντέλο Δεδομένων για τη Ziria

Επιλέξαμε ένα απλό μοντέλο δεδομένων για τη Ziria, με primitive τύπους μόνο τους απλούς ακεραίους (`Numb`), τα bits (`Bit`), και τους αντίστοιχους πίνακες των τύπων αυτών. Η επιλογή αυτή καθορίστηκε από το γεγονός ότι ουσιαστικά η Ziria είναι γλώσσα για επεξεργασία ροών δεδομένων:

```
data Val where
  Numb :: Int → Val
  Bit :: Int → Val
  NumbArr :: [Int] → Val
  BitArr :: [Int] → Val
```

Στη συνέχεια για να ορίσουμε εύκολα χρήσιμους τελεστές πάνω στο `Val` χρειάστηκε να το ορίσουμε ως στιγμιότυπο διάφορων κλάσεων: `Num`, `Ord`, `Real`, `Enum`, `Integral`, `Bits`. Σημειώνεται εδώ ότι χρειάστηκε να οριστούν οι συναρτήσεις `bitsOfVal` και `valOfBits`, όπου κάνουν τη μετατροπή από `Val` σε απλούς τύπους της Haskell και το αντίστροφο, ωστόσο στην `bitsOfVal`, επειδή δεν θέλουμε να χάνουμε σαν πληροφορία από που έχει προέλθει ο εκάστοτε τύπος, επιστρέφουμε μια τούπλα με πρώτο στοιχείο να καταδεικνύει τον αρχικό τύπο (`ValAnnotation`) και δεύτερο την πραγματική τιμή:

```
bitsOfVal :: Val → (ValAnnotation, [Int])

valOfBits :: ValAnnotation → [Int] → Val
```

Μιας και η Ziria χρησιμοποιεί mutable μεταβλητές και πίνακες, για να ενσωματώσουμε αυτό το στοιχείο χρησιμοποιήσαμε μια υπάρχουσα mutable δομή της Haskell, το `IOVector`. Έτσι λοιπόν οι αναφορές υλοποιούνται απλά ως εξής:

```
type Vec a = V.IOVector a
type Ref = Vec Int
```

**Κώδικας 5.5:** Υλοποίηση Μεταβλητών

Υλοποιήσαμε επίσης δύο απλές συναρτήσεις, την `arrel` και την `arrsl`, που επιστρέφουν ένα στοιχείο ή ένα μέρος του πίνακα αντίστοιχα. Την πληροφορία που περιέχεται στα annotations που περιγράψαμε παραπάνω την χρησιμοποιούμε για να είμαστε σε θέση να γνωρίζουμε τι τύπου είναι αυτό που κάνουμε encode στο vector.

Τέλος υπάρχουν δύο συναρτήσεις που κάνουν cast, από τον έναν επιτρεπόμενο τύπο `Val` στον άλλο, τις `toBit` και `toNumb` αντίστοιχα.

Για να έχουμε μια άμεση εποπτεία της ενσωμάτωσης παραθέτουμε μερικά απλά παραδείγματα που χρησιμοποιήσαμε για να επαληθεύσουμε την ορθότητα της υλοποίησης:

```
ex1 :: Zir Char Int ()
ex1 = zemit 42

ex2 :: Zir Int Int Void
ex2 = zmap (\x → x∗(x+1))

ex3 :: Zir Int Int ()
ex3 = ztake >>= \x → zemit (x∗(x+1))
```

```
ex4 :: Zir Int Int Void
ex4 = zrepeat $ ztake >>= \x -> zemit (x*(x+1))

main = do
  assert  1 (ex1 'zrun' []) (Just (), [42])
  assert  2 (ex2 'zrun' [6, 3, 9]) (Nothing, [42, 12, 90])
  assert  3 (ex3 'zrun' [6]) (Just (), [42])
  assert  4 (ex4 'zrun' [6, 3, 9]) (Nothing, [42, 12, 90])
```

**Κώδικας 5.6:** Απλά Παραδείγματα Ενσωμάτωσης

## 5.2 Υλοποίηση του Κρυπτογραφικού Αλγορίθμου SHA-1

Για τον έλεγχο ορθότητας και συμπεριφοράς της ενσωμάτωσης υλοποιήσαμε σε Ziria αλλά και σε Haskell τον κρυπτογραφικό αλγόριθμο SHA-1. Ο SHA-1 επιλέχθηκε ως κατάλληλος αλγόριθμος για τον σκοπό αυτό, αφού από τη μία ταιριάζει με τον σχεδιασμό της Ziria, όντας ένας streaming αλγόριθμος που διαχωρίζει την είσοδο σε blocks, και από την άλλη είναι αρκετά πολύπλοκος ώστε να καλύπτει το μεγαλύτερο φάσμα χαρακτηριστικών της Ziria.

Στην κρυπτογραφία, ο SHA-1 αποτελεί μια κρυπτογραφική hash συνάρτηση που παράγει μια hash τιμή μήκους 160 bits (20 bytes). Σύμφωνα με αυτόν τον ορισμό η υλοποίησή μας, σε υψηλό επίπεδο, δέχεται για είσοδο ένα bit stream, αποτελούμενο από το μήκος του μηνύματος σε bytes για τα πρώτα 8 bits και στη συνέχεια ολόκληρο το προς κρυπτογράφηση μήνυμα ενώ εκπέμπει ως έξοδο ένα bit stream μήκους 160 με το κρυπτογραφημένο μήνυμα. Η υλοποίηση έχει στηριχθεί σε δύο βασικές συναρτήσεις:

`preprocess`
: Εδώ εκτελείται η απαραίτητη προεπεξεργασία του μηνύματος ώστε αυτό να έχει αναμενόμενο μήκος και δομή. Αυτό το επιτυγχάνει προσαρτώντας στην αρχή του μηνύματος το bit 1, όσα bit 0 χρειάζονται και το μήκος του μηνύματος σε 64-bit big endian ακέραιο ώστε το συνολικό νέο μήκος του μηνύματος να είναι πολλαπλάσιο των 512 bits.

`process`
: Η συνάρτηση αυτή αποτελεί τον κύριο κορμό της επεξεργασίας, σπάει το μήνυμα σε διαδοχικά κομμάτια των 512 bits όπου επεξεργάζεται ξεχωριστά. Κάθε κομμάτι σπάει σε δεκαέξι λέξεις των 32 bits, που στη συνέχεια επεκτείνονται με συνδυασμό αυτών σε ογδόντα λέξεις, από τις οποίες με bitwise πράξεις παράγεται το hash κάθε κομματιού και προστίθεται συσσωρευτικά σε πέντε μεταβλητές που κρατούν το μέχρις στιγμής αποτέλεσμα. Μετά το πέρας όλων των επαναλήψεων, οι πέντε μεταβλητές εκπέμπονται ακολουθιακά (στην δυαδική τους αναπαράσταση), σχηματίζοντας έτσι το τελικό bit stream μήκους 160.

Οι δύο αυτές συναρτήσεις συντίθενται κατά το μονοπάτι δεδομένων στην κύρια συνάρτηση του προγράμματος, η οποία σε Ziria διαμορφώνεται ως εξής:

```
let comp main =
    read [bit] >>> preprocess() >>> repeat process() >>> write [bit]
```

**Κώδικας 5.7:** SHA-1 main

Ενδεικτικά για να είμαστε σε θέση να έχουμε εύκολα μια εποπτεία της ενσωμάτωσης και να μπορούμε να συγκρίνουμε τις δύο προσεγγίσεις παραθέτουμε παρακάτω την υλοποίηση μιας απλής ενδιάμεσης συνάρτησης που χρειάστηκε να υλοποιήσουμε. Η συνάρτηση αυτή παίρνει σαν παράμετρο έναν απλό ακέραιο και κάνει `emit` την δυαδική του αναπαράσταση:

Native Ziria

```
fun comp emit_bin (n : int) {
    var n1 : int := n;
    var cnt : int := 32;

    times 32 {
      do {
        cnt := cnt − 1;
        n1 := n ≫ cnt;
      }
      emit bit (n1 & 1)
    }
}
```

Haskell Embedding

```
emit_bin n = do
  n1 ← new $ Numb 0
  cnt ← new $ Numb 31

  ztimes 32 $ do
    dcnt ← deref cnt
    cnt #= dcnt − 1
    n1 #= (n .≫. dcnt) .&. (Bit 1)
    dn1 ← deref n1
    zemit $ toBit dn1
```

**Σχήμα 5.3:** Emit Binary Representation σε Ziria και στην Ενσωμάτωση σε Haskell

Από τα παραπάνω παρατηρούμε ότι η αντιστοίχιση ανάμεσα στις δύο προσεγγίσεις είναι άμεση και απλή. Δύο διαφορές που αξίζει να σημειωθούν είναι οι εξής:

- Στην ενσωμάτωση δεν υφίσταται ο διαχωρισμός μεταξύ υψηλότερου επιπέδου γλώσσα υπολογισμών και χαμηλότερου επιπέδου γλώσσα εκφράσεων, αφού όλοι οι τελεστές ανήκουν στο ίδιο κεντρικό monad `Zir a b`.
- Στην ενσωμάτωση οι μεταβλητές έχουν υλοποιηθεί μέσω αναφορών (βλ. Κώδικας 5.5), οπότε εκτός από την αρχικοποίησή τους με `new`, πριν από κάθε assign πρέπει να γίνονται dereference `deref`.

## Κεφάλαιο 6

# Συμπεράσματα

## 6.1 Συνεισφορά

Σε αυτήν την διπλωματική εργασία καταφέραμε να επιτύχουμε τον αρχικό μας στόχο, δηλαδή να δώσουμε μια υλοποίηση της Ziria ως ενσωματωμένη γλώσσα συγκεκριμένου σκοπού στη Haskell.

Η ενσωμάτωση έγινε ουσιαστικά μέσω της υλοποίησης ενός διερμηνέα για τη Ziria στη Haskell. Για τον διερμηνέα αυτόν χρειάστηκε να ορίσουμε ένα νέο monad, το οποίο μας επιτρέπει να ορίσουμε εύκολα τους Ziria combinators στη Haskell. Η επιλογή της δομής του monad προκύπτει με άμεσο συσχετισμό από το μοντέλο εκτέλεσης μεταγλώττισης της Ziria. Συνεπώς και η τελική ενσωμάτωση είναι σχετικά απλή και σχετικά άμεση ως προς τη σημασιολογία της Ziria. Το στοιχείο αυτό καθιστά το module μας κατάλληλο για πειραματισμό και γρήγορη προτυποποίηση, αφού ο προγραμματιστής έχει στη διάθεσή του ολόκληρη την ισχύ της Haskell και είναι έτσι σε θέση να εκμεταλλευτεί χαρακτηριστικά που η ίδια η Ziria προς το παρόν δεν υποστηρίζει. Σε περίπτωση που αυτά τα χαρακτηριστικά αποδειχθούν χρήσιμα και βολικά θα μπορούσαν τελικά να κατέβουν στην Ziria, η οποία έχει ήδη άλλωστε επηρεαστεί στον σχεδιασμό της υψηλότερου επιπέδου γλώσσας υπολογισμών της από τη Haskell. Η επαναδιαμόρφωση της σημασιολογίας, δηλαδή το νέο monad που ορίσαμε, και το προγραμματιστικό όφελος που προκύπτει από αυτή είναι και το κύριο επιχείρημα υπέρ της συγκεκριμένης ενσωμάτωσης.

Η Ziria είναι μια γλώσσα για επεξεργασία ροών δεδομένων, έτσι με το να ορίσουμε τους βασικούς της τελεστές στο module μας δομούμε ουσιαστικά έναν εναλλακτικό τρόπο για να πραγματοποιούμε υπολογισμούς ροών δεδομένων μέσα στη Haskell. Ωστόσο η σύγκριση σε όρους απόδοσης με άλλα υπάρχοντα μοντέλα, όπως για παράδειγμα του stream fusion [21] που υποστηρίζει αυτή τη στιγμή ο GHC και που υλοποιεί εσωτερικά χαμηλού επιπέδου βελτιστοποιήσεις δεν είναι ιδιαίτερα δόκιμη.

Για τον ελέγχο ορθότητας και πληρότητας της ενσωμάτωσης αλλά και για να έχουμε μια καλύτερη εποπτεία του συντακτικού υλοποιήσαμε εξ'ολοκλήρου την κρυπτογραφική hash συνάρτηση SHA-1 σε Ziria, όπου μπορεί να χρησιμοποιηθεί από τους ερευνητές σαν βιβλιοθήκη σε άλλα μεγαλύτερα συστήματα, αλλά και φυσικά για σύγκριση και μέσω του module μας σε Haskell.

## 6.2 Μελλοντική Έρευνα

Η ραδιοεπικοινωνία ορισμένη από λογισμικό φέρνει την ευελιξία του λογισμικού στον σχεδιασμό ασύρματων πρωτοκόλλων, υποσχόμενη να λειτουργήσει σαν μια ιδανική πλατφόρμα που θα βοηθήσει στην ταχεία και καινοτόμα ανάπτυξη της ασύρματης δικτύωσης. Εντούτοις, η υλοποίηση των σύγχρονων ασύρματων πρωτοκόλλων σε υπάρχουσες πλατφόρμες χρειάζεται συνήθως λεπτούς χειρισμούς σε κώδικα χαμηλού επιπέδου, οι οποίοι τελικά υπονομεύουν τα ίδια τα πλεονεκτήματα του λογισμικού.

Η Ziria είναι μια νέα πλατφόρμα που προσπαθεί να λύσει αυτό το πρόβλημα. Αποτελείται από μια νέα, μη ενσωματωμένη γλώσσα προγραμματισμού συγκεκριμένου σκοπού για επεξεργασία ροών δεδομένων bit και πακέτων καθώς και από έναν μεταγλωττιστή που υλοποιεί πλήθος βελτιστοποιήσεων με στόχο την παραγωγή κώδικα που θα μπορεί να ανταποκριθεί στις υψηλές απαιτήσεις πραγματικού χρόνου που θέτει η ραδιοεπικοινωνία.

Όπως αναφέρθηκε στην παραπάνω ενότητα στην εργασία αυτή ενσωματώσαμε τη Ziria στην Haskell μέσω της υλοποίησης ενός διερμηνέα για αυτήν, με στόχο να έχουμε έναν εύκολο τρόπο να πειραματιζόμαστε και να συνεχίσουμε να εμπλουτίζουμε με χαρακτηριστικά της Haskell την ίδια τη Ziria. Ωστόσο, σαν δεύτερο βήμα θα θέλαμε να μελετήσουμε την υλοποίηση μιας καλύτερης ενσωμάτωσης, η οποία να μην απαιτεί να γίνονται serialize όλες οι monad λειτουργίες (`deref`/`assign`).

Επίσης στο απώτερο μέλλον, θα είχε αξία να προχωρήσουμε στην υλοποίηση ενός μεταγλωττιστή, αντί για διερμηνέα, ο οποίος θα παράγει ένα Ziria αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree, AST) από τον ενσωματωμένο κώδικα, το οποίο με τη σειρά του θα δίνεται στο υπάρχον C backend, όπου μέσω πολλαπλών βελτιστοποιήσεων εξασφαλίζει την απαραίτητη απόδοση για να μπορέσει τελικά η ενσωμάτωση να αποτελέσει μια λειτουργική Haskell βιβλιοθήκη για ραδιοεπικοινωνία και επεξεργασία ροών δεδομένων bit γενικότερα.

# Βιβλιογραφία


[1] S. Sen, R. Roy Choudhury, and S. Nelakuditi, "No time to countdown: Migrating backoff to the frequency domain," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, pp. 241–252, ACM, 2011.

[2] T. Li, M. K. Han, A. Bhartia, L. Qiu, E. Rozner, Y. Zhang, and B. Zarikoff, "CRMA: Collision-resistant multiple access," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, pp. 61–72, ACM, 2011.

[3] T. Bansal, B. Chen, P. Sinha, and K. Srinivasan, "Symphony: Cooperative packet recovery over the wired backbone in enterprise WLANs," in *Proceedings of the 19th Annual International Conference on Mobile Computing and Networking*, pp. 351–362, ACM, 2013.

[4] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: A wireless open-access research platform," in *Proceedings of the 14th European Signal Processing Conference*, pp. 1–5, IEEE, 2006.

[5] R. Sathappan, M. Dumas, and M. Uhm, "A new architecture for development platforms targeted to portable radio applications," *Lyrtech Technical Paper*, 2007.

[6] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "SORA: High-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, no. 1, pp. 99–107, 2011.

[7] M. Ettus, "Universal software radio peripheral (USRP)." Ettus Research LLC, http://www.ettus.com, 2008.

[8] E. Blossom, "GNURadio: tools for exploring the radio frequency spectrum," *Linux Journal*, vol. 2004, no. 122, p. 4, 2004.

[9] E. U. T. R. Access, "Physical layer procedures (release 8)," *Technical Specification, 3GPP TS*, vol. 36, p. V8, 2009.

[10] IEEE, "Wireless LAN medium access control (MAC) and physical layer (PHY) specifications: High-speed physical layer in the 5GHz band," 1999.

[11] M. Gowda, G. Stewart, G. Mainland, B. Radunovic, D. Vytiniotis, and D. Patterson, "Ziria: Language for rapid prototyping of wireless PHY," in *Procedings of the 20th Annual International Conference on Mobile Computing and Networking*, pp. 359–362, ACM, 2014.

[12] D. Vytiniotis, "Ziria: Wireless programming for hardware dummies," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance*, p. 1, ACM, 2014.

[13] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and D. Patterson, "Ziria: Language for rapid prototyping of wireless PHY," in *Proceedings of the ACM SIGCOMM Conference*, pp. 357–358, ACM, 2014.

[14] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agullo, “Ziria: A DSL for wireless systems programming,” in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 415–428, ACM, 2015.

[15] A. Gill, “Domain-specific languages and code synthesis using Haskell,” *Queue*, vol. 12, no. 4, p. 30, 2014.

[16] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991.

[17] J. Hughes, “Generalising monads to arrows,” *Science of Computer Programming*, vol. 37, no. 1-3, pp. 67–111, 2000.

[18] A. Courtney and C. Elliott, “Genuinely functional user interfaces,” in *Proceedings of the Haskell Workshop*, 2001.

[19] C. Elliott and P. Hudak, “Functional reactive animation,” *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 263–273, 1997.

[20] C. Elliott, “Push-pull functional reactive programming,” in *Proceedings of the Haskell Symposium*, 2009.

[21] D. Coutts, R. Leshchinskiy, and D. Stewart, “Stream fusion: From lists to streams to nothing at all,” *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 315–326, 2007.

## Κεφάλαιο 7

# Παράρτημα

Στα προηγούμενα κεφάλαια δείξαμε μόνο κάποια αποσπάσματα της υλοποίησης ή των παραδειγμάτων που γράψαμε. Ο αναγνώστης που θέλει να μάθει περισσότρες λεπτομέρειες μπορεί να δει ολόκληρες τις υλοποιήσεις στις επόμενες σελίδες.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}

module Ziria ( M
             , Zir
             , Void
             , ztake
             , zemit
             , zmap
             , zrepeat
             , ztimes
             , (>>>)
             , (.>>.)
             , (.<<.)
             , Val(..)
             , arrel
             , arrsl
             , new
             , deref
             , (:=)
             , zrun
             , toNumb
             , toBit
             ) where

import Control.Applicative
import Control.Monad
import Control.Monad.Trans.Free

import qualified Data.Vector.Mutable as V
import qualified Data.Bits as B

-- * Preliminaries

-- | A void type
-- | This will be used as the return type for Ziria's transformers.
-- | It will also be the zero element for 'TrivSum'.
data Void

```

```
instance Eq Void where
  (≡) = undefined

instance Show Void where
  showsPrec = undefined

-- * Ziria's squashed type to support arrow-like structures

class TrivSum a b c | a b → c where
  squash :: Either a b → c

instance {-# OVERLAPPING #-} TrivSum Void Void Void where
  squash _ = undefined

instance TrivSum t Void t where
  squash (Left x) = x
  squash _ = undefined

instance TrivSum Void t t where
  squash (Right x) = x
  squash _ = undefined

-- * Ziria specific operators

infix 4 #≤
infix 1 ⋙

-- * A simple data model for Ziria

data Val where
  Numb :: Int → Val
  Bit :: Int → Val
  NumbArr :: [Int] → Val
  BitArr :: [Int] → Val
 deriving (Eq, Show)

instance Num Val where
        (Numb x) + (Numb y) = Numb (x + y)
        (Numb x) - (Numb y) = Numb (x - y)
        (Numb x) * (Numb y) = Numb (x * y)
        negate (Numb x) = Numb (-x)
        abs (Numb x) = if x < 0 then Numb (-x)
                       else Numb x
        signum (Numb x) = if x < 0 then -1
                          else 1
        fromInteger n = Numb (fromIntegral n)

instance Ord Val where
        (Numb x) ≤ (Numb y) = compare x y /= GT
        (Numb x) <  (Numb y) =  compare x y ≡ LT
        (Numb x) ≥ (Numb y) =  compare x y /= LT
        (Numb x) >  (Numb y) =  compare x y ≡ GT

instance Real Val where
        toRational (Numb x) = toRational x

instance Enum Val where
        succ (Numb x) = Numb (x + 1)
```

```
        pred (Numb x) = Numb (x - 1)
        fromEnum (Numb x) = x
        fromEnum (Bit x) = x
        toEnum x = Numb x

instance Integral Val where
        (Numb x) `div` (Numb y) = Numb (x `div` y)
        (Numb x) `mod` (Numb y) = Numb (x `mod` y)
        toInteger (Numb x) = toInteger x
        (Numb x) `quotRem` (Numb y) = (Numb (x `div` y), Numb (x `mod` y))

instance B.Bits Val where
        -- (.&.) (Arr xs) (Arr ys) = Arr $ zipWith (B..&.) xs ys
        (.&.) x y = Numb ((fromEnum x) B..&. (fromEnum y))
        -- (.|.) (Arr xs) (Arr ys) = Arr $ zipWith (B..|.) xs ys
        (.|.) x y = Numb ((fromEnum x) B..|. (fromEnum y))
        -- xor (Arr xs) (Arr ys) = Arr $ zipWith (B.xor) xs ys
        xor x y = Numb ((fromEnum x) `B.xor` (fromEnum y))
        shiftL x y = Numb ((fromEnum x) `B.shiftL` y)
        shiftR x y = Numb ((fromEnum x) `B.shiftR` y)
        complement x = Numb (B.complement (fromEnum x))
        rotateL x y = Numb ((fromEnum x) `B.rotateL` y)
        rotateR x y = Numb ((fromEnum x) `B.rotateR` y)
        bitSizeMaybe x = (B.bitSizeMaybe (fromEnum x))
        bitSize x = (B.finiteBitSize (fromEnum x))
        isSigned x = (B.isSigned (fromEnum x))
        popCount x = (B.popCount (fromEnum x))
        testBit x i = (B.testBit (fromEnum x) i)
        bit i = Numb (1 `B.shiftL` i)

type Vec a = V.IOVector a

data ValAnnotation = BitAnnotation
                   | NumbAnnotation
                   | BitArrAnnotation
                   | NumbArrAnnotation
                  deriving (Eq, Show)
type Ref = Vec Int

arrel :: Enum n => (ValAnnotation, Ref) -> n -> (ValAnnotation, Ref)
arrel (annot, vecref) i =
    let annot' = if annot == BitArrAnnotation then
                    BitAnnotation
                else
                    NumbAnnotation
    in
        (annot', V.slice (fromEnum i) 1 vecref)


arrsl :: (Enum n1, Enum n2) => (ValAnnotation, Ref) -> (n1, n2) -> (ValAnnotation, Ref
    )
arrsl (annot, vecref) (from, upto) =
       (annot, V.slice (fromEnum from) ((fromEnum upto) - (fromEnum from) + 1) vecref)

(.>>.) :: Val -> Val -> Val
x .>>. y = Numb ((fromEnum x) `B.shiftR` (fromEnum y))

(.<<.) :: Val -> Val -> Val
```

```
x .<<. y = Numb ((fromEnum x) `B.shiftL` (fromEnum y))

toNumb :: Val → Val
toNumb (Numb x) = Numb x
toNumb (Bit x) = Numb x

toBit :: Val → Val
toBit (Numb x) = Bit x
toBit (Bit x) = Bit x

bitsOfVal :: Val → (ValAnnotation, [Int])
bitsOfVal (Bit x) = (BitAnnotation, [x])
bitsOfVal (Numb x) = (NumbAnnotation, [x])
bitsOfVal (BitArr xs) = (BitArrAnnotation, xs)
bitsOfVal (NumbArr xs) = (NumbArrAnnotation, xs)

valOfBits :: ValAnnotation → [Int] → Val
valOfBits BitAnnotation [x] = Bit x
valOfBits NumbAnnotation [x] = Numb x
valOfBits BitArrAnnotation x = BitArr x
valOfBits NumbArrAnnotation x = NumbArr x
valOfBits _ _ = error "valOfBits: corrupted reference"

-- * An interpreter for Ziria

type M = IO
data F a b t = Yield t b | NeedInput (a → t)

instance Functor (F a b) where
  fmap f (Yield x output) = Yield (f x) output
  fmap f (NeedInput g) = NeedInput (f . g)

-- Zir a b t ~= M (t  +  (Zir a b t, b)  +  (a → Zir a b t))
--                 Done  Yield              NeedInput
type Zir a b t = FreeT (F a b) M t

ztake :: Zir a b a
{-# INLINE ztake #-}
ztake = wrap $ NeedInput return

zemit :: b → Zir a b ()
{-# INLINE zemit #-}
zemit = wrap . Yield (return ())

(>>>) :: TrivSum t1 t2 t ⇒ Zir a b t1 → Zir b c t2 → Zir a c t
z1 >>> z2 = az1 `zpipe` az2 >>= return . squash
  where az1 = z1 >>= return . Left
        az2 = z2 >>= return . Right

zpipe :: Zir a b t → Zir b c t → Zir a c t
{-# INLINE zpipe #-}
az1 `zpipe` az2 = pull az1 az2
  where pull :: Zir a b t → Zir b c t → Zir a c t
        pull z1 z2 = FreeT $ do
          v2 ← runFreeT z2
          case v2 of
            Pure x             → return $ Pure x
            Free (Yield z2' o) → return $ Free $ Yield (z1 `zpipe` z2') o
```

```
            Free (NeedInput g) → runFreeT $ push g z1     -- Go upstream!
        push :: (b → Zir b c t) → Zir a b t → Zir a c t
        push f z1 = FreeT $ do
          v1 ← runFreeT z1
          case v1 of
            Pure x             → return $ Pure x
            Free (Yield z1' m) → runFreeT $ pull z1' (f m) -- Push downstream!
            Free (NeedInput g) → return $ Free $ NeedInput (push f . g)

zmap :: (a → b) → Zir a b Void
{-# INLINE zmap #-}
zmap f = wrap $ NeedInput $ wrap . Yield (zmap f) . f

zrepeat :: Zir a b () → Zir a b Void
{-# INLINE zrepeat #-}
zrepeat = forever

ztimes :: Enum n ⇒ n → Zir a b () → Zir a b ()
{-# INLINE ztimes #-}
ztimes x = replicateM_ $ fromEnum x

zfilter :: (a → Bool) → Zir a a Void
{-# INLINE zfilter #-}
zfilter f = wrap $ NeedInput $ \x → if f x then wrap $ Yield (zfilter f) x
                                           else zfilter f
new :: Val → Zir a b (ValAnnotation,Ref)
{-# INLINE new #-}
new v = FreeT $ do
  let (annot, bs) = bitsOfVal v
  let l = length bs
  r ← V.new l
  sequence_ $ zipWith (V.write r) [0..] bs
  return $ Pure (annot, r)

deref :: (ValAnnotation, Ref) → Zir a b Val
{-# INLINE deref #-}
deref (annot, r) = FreeT $ do
  let l = V.length r
  bs ← mapM (V.read r) [0..l − 1]
  return $ Pure $ valOfBits annot bs

(#=) :: (ValAnnotation, Ref) → Val → Zir a b ()
{-# INLINE (#=) #-}
(annot, r) #= v = FreeT $ do
  let (_, bs) = bitsOfVal v
  let l1 = V.length r
  let l2 = length bs
  if l1 /= l2
    then fail $ "assign " ++ show l2 ++ " bits to array of " ++ show l1
    else do sequence_ $ zipWith (V.write r) [0..] bs
            return $ Pure ()

type Sem a b t = [a] → M (Maybe t, [b])

zrun :: Zir a b t → Sem a b t
{-# INLINE zrun #-}
zrun = zwalk []
```

```
zwalk :: [b] → Zir a b t → Sem a b t
{-# INLINE zwalk #-}
zwalk acc z inp = do
  v ← runFreeT z
  case v of
    Pure x             → return (Just x, reverse acc)
    Free (Yield az o)  → zwalk (o : acc) az inp
    Free (NeedInput g) → case inp of
                           (x : xs) → zwalk acc (g x) xs
                           []       → return (Nothing, reverse acc)
```

**Κώδικας 7.1:** Ολόκληρη η ενσωμάτωση της Ziria σε Haskell

```
import Ziria
import Data.Bits

-- * The scrambler example from the paper

scrambler = do
  scrmbl_st <- new $ BitArr [1, 1, 1, 1, 1, 1, 1]
  tmp <- new $ Bit 0
  y <- new $ Bit 0
  zrepeat $ do
    x <- ztake
    tmp1 <- deref (scrmbl_st `arrel` 3)
    tmp2 <- deref (scrmbl_st `arrel` 0)
    tmp #= tmp1 `xor` tmp2
    tmp3 <- deref (scrmbl_st `arrsl` (1, 6))
    (scrmbl_st `arrsl` (0, 5)) #= tmp3
    tmp4 <- deref tmp
    (scrmbl_st `arrel` 6) #= tmp4
    tmp5 <- deref tmp
    y #= x `xor` tmp5
    tmp6 <- deref y
    zemit tmp6

input1 = replicate 3520 $ Bit 0
output1 = Prelude.take 3520 $ cycle $ map Bit [
  0,0,0,0,1,1,1,0,1,1,1,1,0,0,1,0,1,1,0,0,1,0,0,1,0,
  0,0,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,1,0,
  1,1,0,1,1,0,0,0,0,0,1,1,0,0,1,1,0,1,0,1,0,0,1,1,1,
  0,0,1,1,1,1,0,1,1,0,1,0,0,0,0,1,0,1,0,1,0,1,1,1,1,
  1,0,1,0,0,1,0,1,0,0,0,1,1,0,1,1,1,0,0,0,1,1,1,1,1,
  1,1]

-- * The main program, running the sequence of tests

assert :: Eq t => Int -> M t -> t -> IO ()
assert test m x = do
  putStr $ "Testing " ++ show test ++ "... "
  y <- m
  if x == y then putStrLn "done"
            else putStrLn "assertion failed!"

main = do
  assert 1 (scrambler `zrun` input1) (Nothing, output1)
```

**Κώδικας 7.2:** Scrambler στην ενσωματωμένη γλώσσα

```
import Ziria

import qualified Data.Vector.Mutable as V
import Data.Word
import Data.Int
import Data.List
import Data.Bits as B

import Prelude hiding (take, repeat, or, and)
import qualified Prelude (take, repeat, or, and)

-- * SHA1

emit_binary_representation n = do
  cnt ← new $ Numb 31
  temp ← new $ Numb 0

  ztimes 32 $ do
      tmp ← deref cnt
      cnt #= tmp - 1
      temp #= (n .>>. tmp) .&. (Bit 1)
      u ← deref temp
      zemit $ toBit u

left_rotate :: Val → Val → Val
left_rotate n b =
    ((n .<<. b) .|. (n .>>. (32 - b))) .&. 0xFFFFFFFF

sha1_get_msg_len msg_len = do
    let zeros_len = ((448 - ((msg_len + 1) `mod` 512) + 512) `mod` 512) in
        zeros_len + msg_len + 1 + 64

binToDec l = sum $ map (2^) $ findIndices (==1) $ reverse l

preprocess msg_len =  do
    cnt ← new $ Numb 63

    ztimes msg_len $ do
        x ← ztake
        zemit $ x
    zemit $ Bit 1 -- TODO: emits

    let zeros_len = ((448 - ((msg_len + 1) `mod` 512) + 512) `mod` 512) in
        ztimes zeros_len $ zemit $ Bit 0

    ztimes 64 $ do
        tmp ← deref cnt
        cnt #= tmp - 1
        zemit $ toBit $ (msg_len .>>. tmp) .&. 1

process msg_len = do
    h0 ← new $ Numb 0x67452301
    h1 ← new $ Numb 0xEFCDAB89
    h2 ← new $ Numb 0x98BADCFE
    h3 ← new $ Numb 0x10325476
    h4 ← new $ Numb 0xC3D2E1F0
    w  ← new $ NumbArr $ replicate 80 0
```

```
    chunk ← new $ BitArr $ replicate 512 0

    total_len ← new $ msg_len + (448 - ((msg_len + 1) `mod` 512) + 512) `mod` 512 +
64 + 1
    tmp_tl ← deref total_len
    times ← new $ tmp_tl `div` 512
    tmp_tm ← deref times

    ztimes tmp_tm $ do
        i ← new $ Numb 0

        -- chunk ← takes 512;
        ztimes 512 $ do
            tmp_i ← deref i
            s ← ztake
            chunk `arrel` tmp_i #= s
            i #= tmp_i + 1

        i #= 0
        ztimes 16 $ do
             tmp_i ← deref i
             -- do{w[i] := 0}
             w `arrel` tmp_i #= 0

             j ← new $ Numb 0
             ztimes 32 $ do
                 tmp_j ← deref j
                 tmp_ch ← deref (chunk `arrel` (tmp_i * 32 + tmp_j))
                 tmp_w ← deref (w `arrel` tmp_i)
                 if (tmp_ch == Bit 1)
                 then
                    w `arrel` tmp_i #= ((tmp_w .<<. 1) .|. 1)
                 else
                    w `arrel` tmp_i #= ((tmp_w .<<. 1) .|. 0)
                 j #= tmp_j + 1

             i #= tmp_i + 1

        i #= Numb 16
        ztimes 64 $ do
            tmp_i ← deref i
            tmp_w3 ← deref (w `arrel` (tmp_i - 3))
            tmp_w8 ← deref (w `arrel` (tmp_i - 8))
            tmp_w14 ← deref (w `arrel` (tmp_i - 14))
            tmp_w16 ← deref (w `arrel` (tmp_i - 16))
            w `arrel` tmp_i #= left_rotate (tmp_w3 `xor` tmp_w8 `xor` tmp_w14 `xor`
tmp_w16) 1
            i #= tmp_i + 1

        tmp_h0 ← deref h0
        tmp_h1 ← deref h1
        tmp_h2 ← deref h2
        tmp_h3 ← deref h3
        tmp_h4 ← deref h4
        a ← new $ tmp_h0
        b ← new $ tmp_h1
        c ← new $ tmp_h2
        d ← new $ tmp_h3
```

```
114         e ← new $ tmp_h4
115         f ← new $ Numb 0
116         k ← new $ Numb 0
117         temp ← new $ Numb 0
118
119         i #= 0
120         ztimes 80 $ do
121              tmp_i ← deref i
122              tmp_a ← deref a
123              tmp_b ← deref b
124              tmp_c ← deref c
125              tmp_d ← deref d
126              tmp_e ← deref e
127
128              if (0 ≤ tmp_i && tmp_i ≤ 19)
129              then do
130                 f #= tmp_d `xor` (tmp_b .&. (tmp_c `xor` tmp_d))
131                 k #= 0x5A827999
132              else if (20 ≤ tmp_i && tmp_i ≤ 39)
133              then do
134                 f #= tmp_b `xor` tmp_c `xor` tmp_d
135                 k #= 0x6ED9EBA1
136              else if (40 ≤ tmp_i && tmp_i ≤ 59)
137              then do
138                 f #= (tmp_b .&. tmp_c) .|. (tmp_b .&. tmp_d) .|. (tmp_c .&. tmp_d)
139                 k #= 0x8F1BBCDC
140              else do
141                 f #= tmp_b `xor` tmp_c `xor` tmp_d
142                 k #= 0xCA62C1D6
143
144              do
145                 tmp_k ← deref k
146                 tmp_f ← deref f
147                 tmp_wi ← deref (w `arrel` tmp_i)
148                 temp #= ((left_rotate tmp_a 5) + tmp_f + tmp_e + tmp_k + tmp_wi) .&.
    Numb 0xFFFFFFFF
149                 e #= tmp_d
150                 d #= tmp_c
151                 c #= left_rotate tmp_b 30
152                 b #= tmp_a
153                 tmp_temp ← deref temp
154                 a #= tmp_temp
155
156                 i #= tmp_i + 1
157
158         tmp_a ← deref a
159         tmp_b ← deref b
160         tmp_c ← deref c
161         tmp_d ← deref d
162         tmp_e ← deref e
163
164         tmp_h0 ← deref h0
165         tmp_h1 ← deref h1
166         tmp_h2 ← deref h2
167         tmp_h3 ← deref h3
168         tmp_h4 ← deref h4
169
170         h0 #= (tmp_h0 + tmp_a) .&. 0xFFFFFFFF
```

```
        h1 #= (tmp_h1 + tmp_b) .&. 0xFFFFFFFF
        h2 #= (tmp_h2 + tmp_c) .&. 0xFFFFFFFF
        h3 #= (tmp_h3 + tmp_d) .&. 0xFFFFFFFF
        h4 #= (tmp_h4 + tmp_e) .&. 0xFFFFFFFF

    do
        tmp_h0 ← deref h0
        emit_binary_representation tmp_h0
        tmp_h1 ← deref h1
        emit_binary_representation tmp_h1
        tmp_h2 ← deref h2
        emit_binary_representation tmp_h2
        tmp_h3 ← deref h3
        emit_binary_representation tmp_h3
        tmp_h4 ← deref h4
        emit_binary_representation tmp_h4

-- input for "ab" string
-- correct output for "ab"
sha1_input = map Bit [0,1,1,0,0,0,0,1,0,1,1,0,0,0,1,0]
sha1_output = map Bit [1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,1,
  0,1,1,0,0,0,0,1,0,1,0,0,1,1,1,0,0,0,0,0,0,1,0,0,
  1,0,0,0,1,1,0,1,0,0,1,1,0,1,0,0,0,0,0,1,1,0,1,0,1,
  1,1,1,1,0,0,0,1,1,1,1,0,1,1,1,1,0,1,0,0,0,1,1,0,1,
  1,1,1,0,1,1,0,1,0,1,0,1,1,0,1,0,1,1,1,0,0,1,0,0,1,
  1,1,0,0,0,1,0,0,0,1,1,1,0,1,0,0,1,0,1,1,0,0,0,1,1,
  0,0,1,0,0,0,0,0,1,0,0,1,1,0,1,1,1,0,0]

sha1_main msg_len =
  (preprocess msg_len) ⋙ zrepeat (process msg_len)

-- the message length must be set manually
-- the example we use is the "ab" string
-- with msg_len = 16
main = do
    y ← (zrun (sha1_main 16) sha1_input)
    print $ y
    print $ sha1_output
```

**Κώδικας 7.3:** SHA-1 στην ενσωματωμένη γλώσσα