NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Expressive and Efficient Streaming Libraries

Aggelos C. Biboudis

ATHENS

MARCH 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Εκφραστικές και Αποτελεσματικές Βιβλιοθήκες Ροών

Άγγελος Χ. Μπιμπούδης

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2017

PhD THESIS

Expressive and Efficient Streaming Libraries

Aggelos C. Biboudis

SUPERVISOR: Yannis Smaragdakis, Professor NKUA

THREE-MEMBER ADVISORY COMMITTEE:

Yannis Smaragdakis, Professor NKUA

Stathes Hadjiefthymiades, Associate Professor NKUA

Panagiotis Rondogiannis, Professor NKUA

SEVEN-MEMBER EXAMINATION COMMITTEE

Yannis Smaragdakis,
Professor NKUA

Stathes Hadjiefthymiades,
Associate Professor NKUA

Panagiotis Rondogiannis,
Professor NKUA

Alex Delis,
Professor NKUA

Nikolaos Papaspyrou,
Associate Professor NTUA

Konstantinos Sagonas,
Associate Professor NTUA

Dimitrios Vytiniotis,
Researcher Microsoft Research

Examination Date: March 7, 2017

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Εκφραστικές και Αποτελεσματικές Βιβλιοθήκες Ροών

Άγγελος Χ. Μπιμπούδης

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Γιάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Γιάννης Σμαραγδάκης, Καθηγητής ΕΚΠΑ

Ευστάθιος Χατζηευθυμιάδης, Αναπληρωτής Καθηγητής ΕΚΠΑ

Παναγιώτης Ροντογιάννης, Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Γιάννης Σμαραγδάκης,
Καθηγητής ΕΚΠΑ

Ευστάθιος Χατζηευθυμιάδης,
Αναπληρωτής Καθηγητής ΕΚΠΑ

Παναγιώτης Ροντογιάννης,
Καθηγητής ΕΚΠΑ

Αλέξης Δελής,
Καθηγητής ΕΚΠΑ

Νικόλαος Παπασπύρου,
Αναπληρωτής Καθηγητής ΕΜΠ

Κωνσταντίνος Σαγώνας,
Αναπληρωτής Καθηγητής ΕΜΠ

Δημήτριος Βυτινιώτης,
Ερευνητής Microsoft Research

Ημερομηνία Εξέτασης: 7 Μαρτίου 2017

ABSTRACT

Stream processing is mainstream (again): Widely-used stream libraries are now available for virtually all modern OO and functional languages, from Java to C# to Scala to OCaml to Haskell. Yet expressivity and performance are still lacking. This dissertation identifies the key high-level differences between various implementations, observes that future use cases are tied with past design decisions, and shows that simple abstraction mechanisms are not sufficient. Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance? We present a twofold modularization of streams. To begin with, we untangle streams from the definition of their syntax and semantics and afterwards we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.

Our contributions are preceded by a performance assessment of the current state-of-the-art of streaming libraries. Subsequently, we first propose a mechanism to enhance the maintainability of streams, supporting a high-level of extensibility. We treat streams as a domain-specific language and we design and implement **StreamAlg**, a library that has the ability to accept new operators and semantics á la carte. Next, we port the library design we used for streams to Java itself, with a lightweight tool named **Recaf**. We show how to create dialects in Java, override its semantics, support new syntactic elements and much more. Among many examples and case studies we build an extension of Java with a keyword that enables us to construct streams similar to C#. The culmination of our work is a library design, **Strymonas**, for very efficient streams while preserving their high-level nature. It explicitly avoids the reliance on black-box optimizers and "sufficiently-smart" compilers, offering highest, guaranteed and portable performance. Our approach relies on high-level concepts that are then readily mapped into an implementation.

SUBJECT AREA: Programming Languages, Performance

KEYWORDS: Code generation, domain-specific languages, multi-stage programming, optimization, stream fusion, streams

ΠΕΡΙΛΗΨΗ

Οι ροές επεξεργασίας δεδομένων είναι και πάλι στο προσκήνιο: ευρέως-διαδεδομένες βιβλιοθήκες για επεξεργασία ροών υπάρχουν σε κάθε μοντέρνα γλώσσα προγραμματισμού και οικοσύστημα, από την Java στην C# και από την Scala και OCaml στη Haskell. Ωστόσο, καίρια χαρακτηριστικά τους όπως η εκφραστικότητα και οι επιδόσεις έχουν σημαντικά περιθώρια βελτίωσης. Στην παρούσα διατριβή, εξακριβώνουμε το ποια είναι ακριβώς τα ποιοτικά χαρακτηριστικά κάθε βιβλιοθήκης και παρατηρούμε ότι παρούσες αλλά και μελλοντικές χρήσεις τους, είναι άρρηκτα συνδεδεμένες (και δεσμευτικές) με την εκάστοτε υλοποίηση. Απλοί μηχανισμοί αφαίρεσης δεν είναι επαρκείς. Είναι εφικτό να διαμορφώσουμε την υλοποίηση των βιβλιοθηκών ροών και να τις κάνουμε πιο επεκτάσιμες και με μεγαλύτερες επιδόσεις; Παρουσιάζουμε δυο λύσεις προς την επίτευξη της επεκτασιμότητάς τους. Κατά την πρώτη, διαχωρίζουμε τους ορισμούς του συντακτικού και της σημασιολογίας των βιβλιοθηκών και κατά τη δεύτερη λύση απελευθερώνουμε τις εν λόγω βιβλιοθήκες από την επιτακτική ανάγκη ύπαρξης ενός ικανού και έξυπνου μεταγλωττιστή. Ο στόχος της διατριβής είναι να δημιουργήσει επεκτάσιμες βιβλιοθήκες ροών, υψηλών επιδόσεων, διατηρώντας όμως την υψηλού επιπέδου δομή τους.

Η παρουσίαση των ερευνητικών μας αποτελεσμάτων έπεται αξιολόγησης επιδόσεων σε σύγχρονες βιβλιοθήκες που διεξήγαμε για διάφορες σύγχρονες γλώσσες προγραμματισμού. Στη συνέχεια προτείνουμε έναν μηχανισμό για την ενίσχυση της συντήρησης βιβλιοθηκών ροών διατηρώντας ενός υψηλού επιπέδου επεκτασιμότητα. Για να το επιτύχουμε αυτό, θεωρούμε τις εν λόγω βιβλιοθήκες ως γλώσσες προγραμματισμού ειδικού σκοπού, σχεδιάζοντας και υλοποιώντας την βιβλιοθήκη **StreamAlg**, μία βιβλιοθήκη που προσφέρει την δυνατότητα να δεχθεί επεκτάσεις με νέες μεθόδους και συμπεριφορά κατ' επιλογή. Στη συνέχεια, παρουσιάζουμε την εφαρμογή της προαναφερθείσας τεχνικής σε μία γλώσσα προγραμματισμού, την Java, για να επιτύχουμε την ίδια γενικότητα κατά την επέκταση της, με το εργαλείο που ονομάζουμε **Recaf**. Δείχνουμε πώς δημιουργούμε διαλέκτους στην Java για να υποστηρίξουμε νέο συντακτικό και νέα σημασιολογικά στοιχεία στην ίδια τη γλώσσα. Ως αποτέλεσμα, δημιουργήσαμε επεκτάσεις της Java, μία εκ των οποίων μας έδωσε τη δυνατότητα να δημιουργήσουμε βιβλιοθήκη ροών στη Java, όπως ακριβώς γίνεται και στην C#. Τέλος παρουσιάζουμε την πρότασή μας για βιβλιοθήκες ροών υψηλών επιδόσεων που υλοποιήσαμε σε δύο γλώσσες προγραμματισμού με τον ενιαίο όρο-ομπρέλα **Strymonas**. Η βιβλιοθήκη που παρέχουμε, χωρίς να βασίζεται σε απροσπέλαστους βελτιστοποιητές ή "ικανοποιητικά-ικανούς" μεταγλωττιστές, παρέχει, υψηλού επιπέδου, εγγυημένη και φορητή ταχύτητα ως προς την επεξεργασία δεδομένων. Η προσέγγισή μας βασίζεται σε έννοιες υψηλού επιπέδου οι οποίες στη συνέχεια υλοποιούνται.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Γλώσσες Προγραμματισμού, Επιδόσεις

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Γεννήτορες κώδικα, γλώσσες ειδικού σκοπού, προγραμματισμός πολλαπλών σταδίων, βελτιστοποίηση, σύντηξη ροών, ροές

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Η παρούσα διδακτορική διατριβή έχει τον τίτλο “Εκφραστικές και Αποτελεσματικές Βιβλιοθήκες Ροών (Expressive and Efficient Streaming Libraries)” και επικεντρώνεται στη θεματική περιοχή της μελέτης τεχνικών για υλοποίηση επεκτάσιμων ροών και ροών γρήγορων επιδόσεων. Σκοπός της συγκεκριμένης διατριβής είναι να μελετήσει την δομή των εγκαθιδρυμένων βιβλιοθηκών για επεξεργασία δεδομένων (stream libraries) και να προτείνει τεχνικές επέκτασης τους. Όλες οι γλώσσες προγραμματισμού γενικού σκοπού που εκτελούνται σε εικονικό περιβάλλον (VM-based), όπως είναι η C#, F#, Java και Scala έχουν τέτοιες βιβλιοθήκες, έτοιμες για αξιοποίηση από τους προγραμματιστές. Η χρήση τους αφορά την επεξεργασία δεδομένων που βρίσκονται στη μνήμη με τρόπο γρήγορο και αποδοτικό. Η επεξεργασία αυτή μπορεί να γίνει είτε σειριακά είτε παράλληλα. Οι ροές δηλώνονται σε μορφή κλήσεων μεθόδων των εν λόγω βιβλιοθηκών και η τελική σύνθεση που προκύπτει εφαρμόζεται σε δεδομένα που μπορεί να βρίσκονται σε οποιαδήποτε δομή δεδομένων, είτε σε συνεχόμενη μνήμη είτε όχι.

Η παρακάτω εικόνα παρουσιάζει τις βασικές μεθόδους με τις οποίες χειριζόμαστε τις εν λόγω ροές και διαμορφώνουμε τους υπολογισμούς πάνω σε αυτές.

Παραγωγή πεπερασμένης ροής

```
val of_arr   : 'a array → 'a stream
```

Παραγωγή πιθανώς άπειρης ροής

```
val unfold   : ('state  → ('a * 'state) option) → 'state → 'a stream
```

Μετασχηματισμός γραμμικής φύσεως

```
val map      : ('a → 'b) → 'a stream → 'b stream
```

Μετασχηματισμός μη γραμμικής φύσεως

```
val filter   : ('a → bool) → 'a stream → 'a stream
val take     : int → 'a stream → 'a stream
val flat_map : ('a → 'b stream) → 'a stream → 'b stream
```

Μετασχηματισμός παράλληλων ροών

```
val zip_with : ('a → 'b  → 'c) → 'a stream → 'b stream  → 'c stream
```

Καταναλωτές ροής

```
val fold     : ('state → 'a → 'state) → 'state → 'a stream → 'state
```

Τα χαρακτηριστικά αυτών των ροών είναι ότι δεν αποθηκεύουν ενδιάμεσα δεδομένα, ενεργούν μια φορά πάνω στα δεδομένα της μνήμης και η επεξεργασία ανά στοιχείο γίνεται μόνο όσο αυτό είναι απαραίτητο (on demand). Κοινός παρονομαστής όλων αυτών των

βιβλιοθηκών είναι ο συναρτησιακός προγραμματισμός που μέσω της χρήσης συναρτήσεων υψηλότερης τάξης επιτρέπουν στον προγραμματιστή να εκφραστεί καλύτερα ως προς την επεξεργασία δεδομένων που θέλει να δηλώσει, σε μορφή ροής.

Στην εν λόγω διατριβή μελετούμε τις βιβλιοθήκες ροών, προτείνουμε μηχανισμό για επεκτάσιμες βιβλιοθήκες ροών, γενικεύουμε τα ευρήματά μας από το μηχανισμό επεκτασιμότητας των βιβλιοθηκών, εφαρμόζοντάς τα στην ίδια τη γλώσσα προγραμματισμού Java και καταλήγοντας, προτείνουμε σχεδιασμό βιβλιοθήκης για ροές που εκτελούνται αποδοτικά για σειριακή εκτέλεση. Το περιεχόμενο της διατριβής είναι χωρισμένο σε οκτώ κεφάλαια.

Στο πρώτο κεφάλαιο γίνεται μία σύντομη εισαγωγή στην ανάγκη και χρήση ροών σωληνώσεων (stream pipelines) στις σύγχρονες εφαρμογές και παρουσιάζεται η ερευνητική συνεισφορά της διατριβής.

Στο δεύτερο κεφάλαιο αναλύεται το απαραίτητο υλικό για την κατανόηση των τεχνικών που εφαρμόστηκαν στα υπόλοιπα κεφάλαια.

Στο τρίτο κεφάλαιο μελετώνται τέσσερις γλώσσες προγραμματισμού C#, F#, Java και Scala σε εκτέλεση μικρών προγραμμάτων αναφοράς, αξιολογώντας την εκτέλεση των παρεχόμενων βιβλιοθηκών τους σε σειριακή, παράλληλη, βασισμένη σε Linux και Windows, εκτέλεση. Τα προγράμματα αναφοράς έχουν την μορφή του κώδικα που παραθέτουμε παρακάτω. Το συγκεκριμένο παράδειγμα υπολογίζει το άθροισμα των τετραγώνων όλων των στοιχείων του πίνακα `arr` και δείχνουμε την υλοποίηση σε Scala και Java.

Παρατηρούμε ότι και στις δυο γλώσσες, χρησιμοποιούνται οι αντίστοιχες βιβλιοθήκες ροών και πως και οι δυο έχουν την ίδια μορφή.

Αναλύονται οι σχεδιαστικές επιλογές πίσω από κάθε βιβλιοθήκη και παρουσιάζονται τα αποτελέσματα που αφορούν στις επιδόσεις των εν λόγω ροών.

Παράδειγμα σε Scala:

```
def sumOfSquares(arr : Array[Double]) : Double = {
  val sum : Double = arr.view
     .map(a_i => a_i * a_i)
     .sum
  sum
}
```

Το ίδιο παράδειγμα σε Java 8:

```
public double sumOfSquares(double[] arr) {
  double sum = DoubleStream.of(arr)
     .map(a_i → a_i * a_i)
     .sum();
  return sum;
}
```

Αν δούμε αφαιρετικά τις σχεδιαστικές επιλογές πίσω από τις δύο βιβλιοθήκες παρατηρούμε δύο στρατηγικές σχεδιασμού. Η πρώτη ονομάζεται *pull* και βρίσκεται πίσω από τις ροές της Scala, C# και F#. Η δεύτερη ονομάζεται *push* και βρίσκεται πίσω από τις ροές της Java. Στις παρακάτω εικόνες βλέπουμε το γενικό σχέδιο και των δύο. Στο pull μοντέλο, οι ροές αξιοποιούν το μοντέλο των iterators. Οπότε κάθε υλοποίηση μεθόδου χειρίζεται και έναν iterator. Η κατανάλωση της ροής γίνεται με το να εκτελέσει ο καταναλωτής (π.χ., η μέθοδος `fold`) τον iterator. Έτσι ο καταναλωτής χειρίζεται την ροή, *έλκοντας* τα στοιχεία από τον iterator, ένα-ένα.

```
Pull<T> of_arr(T[] arr) {
  return new Pull<T>() {
    boolean hasNext() {..}
    T next() {..}
  };
}

Pull<Integer> sIt =
  source(v).map(i→i*i);

while (sIt.hasNext()) {
  el = sIt.next();
  /* κατανάλωση στοιχείου */
}
```

Στη δεύτερη περίπτωση, στη λεγόμενη push, μπορούμε να θεωρήσουμε τις μεθόδους των ροών σαν να μετασχηματίζουν συναρτήσεις. Από την παραγωγή μιας τέτοιας ροής, επιστρέφεται μια μέθοδος υψηλότερης τάξης που κωδικοποιεί την επαναληπτική διαδικασία πάνω από τον πίνακα. Η κατανάλωση της ροής γίνεται με την κλήση της επιστρεφόμενης συνάρτησης περνώντας της ως παράμετρο μια συνάρτηση που ορίζει πώς θα καταναλωθεί το κάθε στοιχείο. Στην πραγματικότητα κάθε στοιχείο από τον πίνακα, *προωθείται* σε αυτές τις συναρτήσεις μετά τη σύνθεσή τους.

```
Push<T> of_arr(T[] arr) {
  return k → {
   for (int i = 0;
        i < arr.length; i++)
     k(arr[i]); };
}

Push<Integer> sFn =
  source(v).map(i→i*i);

sFn(el → /* κατανάλωση στοιχείου */);
```

Στο τέταρτο κεφάλαιο προτείνεται ένας σχεδιασμός βιβλιοθήκης ροών, που ονομάζεται **StreamAlg**, επεκτάσιμης και ως προς τις παρεχόμενες συναρτήσεις (operators όπως οι `map` και `sum`) αλλά και ως προς τη σημασιολογία/συμπεριφορά τους (semantics/behavior), χωρίς ο κώδικας να χρειάζεται μεταγλώττιση όταν επεκτείνεται. Το κίνητρο είναι να παρέχεται η δυνατότητα στον προγραμματιστή να εφαρμόζει τις τεχνοτροπίες και τις σχεδιαστικές επιλογές που είχαν κάνει άλλες υλοποιήσεις, κατά βούληση, χωρίς οι ροές να είναι μονολιθικά συνδεδεμένες με την υλοποίησή τους. Τα εν λόγω κίνητρα, δημιουργήθηκαν από τα αποτελέσματα της ανάλυσης από το πρώτο κεφάλαιο και, για την επίτευξη του σκοπού μας, θεωρούμε τις εν λόγω βιβλιοθήκες ως γλώσσες ειδικού σκοπού και εφαρμόζουμε τεχνική που έχει αναπτυχθεί στο ευρύτερο πεδίο των εν λόγω γλωσσών. Μια ροή στη βιβλιοθήκη που παρουσιάζουμε μοιάζει με το άθροισμα τετραγώνων που είδαμε νωρίτερα. Το σύμβολο `s` είναι μεταβλητή προγράμματος και καθορίζει τη σημασιολογία εκτέλεσης της ροής.

```
s.sum(s.map(x → x * x), s.source(arr));
```

Παρουσιάζουμε μέσα από προγράμματα αναφοράς ότι η ταχύτητα εκτέλεσης όχι μόνο δεν επηρεάζεται αρνητικά, από την εισαγωγή ενός καινούριου επιπέδου αφαίρεσης, αλλά μπορεί να βελτιωθεί και να ελέγχεται ποικιλοτρόπως.

Στα παρακάτω παραδείγματα βλέπουμε ότι μπορεί να έχουμε μια ποικιλία από εκτελέσεις, από pull και push, μέχρι και συνδυασμό συμπεριφορών ανάλογα το επιθυμητό σενάριο.

```
s = new Push();
s = new Pull();
s = new Log(new Push());
s = new WithFutures(new Push()).get();
s = new WithFutures(new Pull()).get();
```

Στο πέμπτο κεφάλαιο γίνεται γενίκευση της ίδιας τεχνικής, εφαρμόζοντάς την στην Java αυτή τη φορά και όχι σε μια άλλη γλώσσα ειδικού σκοπού (σαν τις προαναφερθείσες ροές). Ενώ άλλες γλώσσες παρέχουν δυνατότητες για επέκτασής τους εγγενώς, η Java δεν παρέχει κάτι αντίστοιχο. Στο κεφάλαιο αυτό προτείνεται ένα εργαλείο, που ονομάζεται **Recaf**, που προσφέρει τις εν λόγω δυνατότητες στην Java, χωρίς επέκταση του υπάρχοντος μεταγλωττιστή. Συγκεκριμένα, δίνεται η δυνατότητα στον προγραμματιστή να ορίσει καινούργια στοιχεία της γλώσσας, τη σημασιολογία τους αλλά και να επαναορίσει την ήδη υπάρχουσα, *με κώδικα Java*. Για παράδειγμα η Java δεν παρέχει τη δυνατότητα του συντακτικού στοιχείου `yield` που υπάρχει σε άλλες, όπως για παράδειγμα στην C#.

Για την επέκταση του συντακτικού, το Recaf δεν απαιτεί αλλαγές στον συντακτικό αναλυτή, αλλά βασίζεται σε ήδη υπάρχοντα συντακτικά πρότυπα της γλώσσας για να ανακαλύψει το είδος της επέκτασης.

Παρουσιάζουμε την επέκταση της γλώσσας Java με το `yield`. Η χρήση του είναι απλή και ακολουθεί τους κανόνες της C# και με αυτό μπορούμε να δημιουργήσουμε βιβλιοθήκη

ροών που ακολουθεί το ίδιο μοντέλο προγραμματισμού με τη C#.

```
recaf YieldImpl<Integer> s = new YieldImpl<Integer>();

recaf Iterable<Integer> filter(Iterable<Integer> iter,
                               Predicate<Integer> pred) {
  for (Integer t: iter) {
    if (pred.test(t)) {
       yield! t;
    }
  }
}
```

Στο εν λόγω παράδειγμα (filter operator) παρουσιάζεται η *μετάφραση* του κώδικα που χρησιμοποιεί το `yield`. Το yield, συντακτικά μοιάζει με το `return` οπότε ο μεταγλωττιστής του Recaf το αναγνωρίζει. Το Recaf, ανιχνεύει την ανάγκη για μετάφραση του κώδικα από τη χρήση της λέξης `recaf` πριν τις δηλώσεις και μετασχηματίζει τον κώδικα σε κάτι γενικό. Όπως βλέπουμε παρακάτω, η μετάφραση έχει απομακρύνει το `recaf` και έχει παράξει κώδικα Java:

```
YieldImpl<Integer> s = new YieldImpl<Integer>();

Iterable<Integer> filter(Iterable<Integer> iter,
                         Predicate<Integer> pred) {
  return s.Method(
           s.ForEach(() → iter,
                     (t) → s.If(() → pred.test(t),
                                s.Yield(() → t))));
}
```

Όπως και στο StreamAlg, υπάρχει ένα σύμβολο (επιλεγμένο να έχει το ίδιο όνομα, `s`, και στα δύο παραδείγματα) το οποίο είναι μεταβλητή και μπορεί να ορίσει τη σημασιολογία της γλώσσας. Στο StreamAlg, της γλώσσας ειδικού σκοπού των ροών και εδώ της γλώσσας γενικού σκοπού, Java.

Στο συγκεκριμένο παράδειγμα η υλοποίηση της σημασιολογίας γίνεται στο αντικείμενο `YieldImpl`.

Στο έκτο κεφάλαιο εξετάζεται το πρόβλημα της σύντηξης γρήγορων ροών και προτείνεται το σχέδιο βιβλιοθήκης **Strymonas** για ροές γρήγορης εκτέλεσης. Αυτό επιτυγχάνεται εξαλείφοντας παράγοντες που απομένουν κατά την σύνθεση σωληνώσεων από το χρήστη (στη δομή του κώδικα που εκτελείται), παράγοντας κώδικα που θα εκτελεστεί γρηγορότερα από άλλες βιβλιοθήκες γενικού σκοπού. Η τεχνολογία βασίζεται σε τεχνική μεταπρογραμματισμού που έδωσε την δυνατότητα να προγραμματιστεί *η ίδια*, η βιβλιοθήκη, ως γεννήτρια του γρήγορου κώδικα–χωρίς αλλαγή του μεταγλωττιστή γενικού σκοπού. Έτσι

ο προγραμματιστής μπορεί να γράψει δηλωτικά τις ροές του, επαναχρησιμοποιώντας ό,τι γνώση έχει ήδη για αυτές. Επιπλέον μπορεί να εκτελέσει κώδικα που θα του φέρει αποτελέσματα τόσο γρήγορα όσο αν είχε γράψει δια χειρός το πρόγραμμα με στοιχεία μιας γλώσσας χαμηλού επιπέδου (`while`, `for`-loops, state, κτλ). Η βιβλιοθήκη που προτείνουμε παράγει *η ίδια* το βέλτιστο κώδικα για γρήγορη εκτέλεση. Η χρήση γίνεται εφαρμόζοντας μια ροή σε ένα στοιχείο που περιέχει τα δεδομένα μας.

```
of_arr .⟨arr⟩.
  ▷ map (fun x → .⟨~x * ~x⟩.)
  ▷ sum
```

Με αυτόν τον τρόπο, η *βιβλιοθήκη* μας παράγει τον παρακάτω κώδικα, χωρίς να έχουμε γράψει καινούριο μεταγλωττιστή ή να έχουμε "προγραμματίσει" τον υπάρχοντα μεταγλωττιστή (της γλώσσας OCaml στο συγκεκριμένο παράδειγμα). Παρατηρούμε ότι ο παρακάτω κώδικας περιλαμβάνει state, `for`-loop, όλες οι συναρτήσεις υψηλότερης τάξης έχουν ενσωματωθεί καθώς επίσης και η αρχικοποίηση του πίνακα στον πίνακα με τα στοιχεία ένα έως τέσσερα.

```
let s_1 = ref 0 in
let arr_2 = [|0;1;2;3;4|] in
for i_3 = 0 to (Array.length arr_2) − 1 do
  let el_4 = arr_2.(i_3) in
  let t_5 = el_4 * el_4 in s_1 := !s_1 + t_5
done;
!s_1
```

Η τεχνική που χρησιμοποιούμε λέγεται *προγραμματισμός πολλαπλών σταδίων* (Multi-Stage Programming) και προέρχεται από την περιοχή της μερικής διατίμησης (Partial Evaluation). Μέσω αυτής μας δίνεται η δυνατότητα να ορίσουμε ποια μέρη του κώδικα θα περάσουν ως πρότυπο κώδικα, στην εκτέλεση του προγράμματος (αντί να έχουν αποτιμηθεί), και να τα χειριστούμε για να ελέγξουμε τη δομή του τελικού κώδικα που θα εκτελεστεί.

```
(* σύμβολα για τη δημιουργία προτύπων κώδικα με ασφαλή τρόπο *)
let c = .⟨ 1 + 2 ⟩.

(* δημιουργία κενών *)
let cf x = .⟨ ~x + ~x ⟩.

(* σύνθεση κώδικα *)
cf c ~> .⟨ (1 + 2) + (1 + 2) ⟩.
```

Στο παραπάνω παράδειγμα είδαμε τη χρήση των εν λόγω συμβόλων για να μεταφέρουμε τον κώδικα των συναρτήσεων υψηλότερης τάξης, ως κώδικα και όχι ως τιμές, στην φάση

της εκτέλεσης του προγράμματος (runtime). Ο Strymonas είχε έτσι τη δυνατότητα να διαχειριστεί αυτά τα κομμάτια κώδικα, να τα αναδιατάξει και να δημιουργήσει τον κώδικα που είδαμε παραπάνω.

Εκτός από απλά παραδείγματα σαν το προηγούμενο, προσφέρουμε σύντηξη ροών για πιο πολύπλοκες ροές σαν την παρακάτω που χρησιμοποιεί όλες τις μεθόδους ροών που παρουσιάσαμε παραπάνω (η μέθοδος `iota` δημιουργεί μία άπειρη ροή από φυσικούς αριθμούς).

```
zip_with
 (* παράλληλες ροές *)
 (fun e1 e2 → .⟨(∼e1,∼e2)⟩.)

 (* πεπερασμένη ροή *)
 (of_arr .⟨arr1⟩.
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ take .⟨12⟩.
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨∼x * ∼x⟩.))

 (* άπειρη ροή *)
 (iota .⟨1⟩.
   ▷ flat_map (fun x → iota .⟨∼x+ 1⟩. ▷ take .⟨3⟩.)
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.))

 ▷ fold (fun z a → .⟨∼a :: ∼z⟩.) .⟨[]⟩.
```

Στο έβδομο κεφάλαιο αναφέρονται πηγές σχετικές με την ερευνητική δουλειά και αντιπαραβάλλονται τα αποτελέσματα μας με εκείνες.

Συνοπτικά, οι χρήσεις των εργαλείων που αναπτύξαμε μπορούν είτε να αξιοποιηθούν ως έχουν είτε να δώσουν έναυσμα για περαιτέρω έρευνα και ανάπτυξη. Μερικές κατηγορίες χρήσης των αποτελεσμάτων μας είναι οι παρακάτω:

- **StreamAlg**
  - αποσπώμενες ροές
  - αποσπώμενοι βελτιστοποιητές
  - αποσπώμενες μηχανές βάσεων δεδομένων για επερωτήσεις
- **Recaf**
  - παραγωγή κώδικα / εκτέλεση κώδικα
  - πειραματισμός με σημασιολογία γλωσσών για εκπαίδευση
  - ενσωμάτωση βιβλιοθηκών με δικό τους συντακτικό

- **Strymonas**
  - γρήγορη βιβλιοθήκη ροών γενικού σκοπού για άμεση χρήση
  - δυνατότητες επέκτασης για σενάρια υπολογισμού υψηλών επιδόσεων

Στο όγδοο κεφάλαιο παρουσιάζονται μελλοντικές ερευνητικές κατευθύνσεις και τελική εκτίμηση της διατριβής.

Αποτελέσματα της παραπάνω έρευνας παρουσιάστηκαν στα πλέον έγκριτα διεθνή συνέδρια γλωσσών προγραμματισμού: ECOOP (2015), GPCE (2016) και POPL (2017). Το περιεχόμενο της διατριβής σχετίζεται συνολικά με τέσσερις δημοσιεύσεις εργασιών σε επιστημονικά συνέδρια [14–16, 91]. Το υλικό ανοιχτού κώδικα, που συνοδεύει τις δημοσιεύσεις, όποτε το συνέδριο παρείχε τη δυνατότητα (ECOOP (2015) και POPL (2017)), εξετάστηκε και πιστοποιήθηκε από ανεξάρτητη επιτροπή που έκρινε ότι ήταν συνεπές με το περιεχόμενο των δημοσιεύσεων. Η διατριβή χαρακτηρίζεται από πρωτοτυπία και πληρότητα.

To my beloved parents, Despoina and Christos,
for their immense encouragement and endless support

ACKNOWLEDGEMENTS

I want to express my sincere gratitude to my advisor Yannis Smaragdakis for his tremendous support on my PhD studies and research. He introduced me to the scientific field of programming languages design and the marvelous process of academic communication. His guidance has proven crucial, especially in the difficult times of thesis topic reassessments. I thank him for always encouraging me to publish, attend summer schools, being open to new research ideas, directions, collaborations, always willing to brainstorm.

I am grateful to Nick Palladinos (Nessos IT), a long-time friend, colleague and co-author, for our lengthy conversations, brainstorming and hacking sessions. Nick, with his passion in applied functional programming, has greatly shifted my interest to programming languages during the time we were colleagues at Nessos IT, several years before getting into the PhD program. This dissertation was shaped by our collaboration!

I thank Oleg Kiselyov, who over the course of one and a half year, towards the end of my PhD, changed my whole perspective about domain-specific optimizations, inspired me with his systematic approach in problem solving, development and paper authoring.

I thank my exceptional collaborators from Centrum Wiskunde & Informatica in Amsterdam (CWI), Tijs van der Storm and Pablo Inostroza, who made every day of my internship count with fruitful discussions, brainstorming and hacking sessions on our project.

I am indebted to the members of my dissertation committee: Stathes Hadjiefthymiades, Panos Rondogiannis, Alex Delis, Nikolaos Papaspyrou, Kostis Sagonas and Dimitrios Vytiniotis for their valuable comments on this dissertation and for guidance over the years.

During my journey as a PhD my research was influenced significantly by communication I had with: Brian Goetz (Oracle) for our interesting discussions about the Clash of the Lambdas performance assessment project, Aleksey Shipilëv (Red Hat, Oracle at that time) who, very patiently, provided very valuable feedback on my early steps in performance evaluation, Paul Sandoz (Oracle) for our Skype sessions, helping me with benchmarking orientation about streams, Vlad Ureche (EPFL) for the discussions we had about streams, performance and the work we have done together, Jan Dzik (Imperial College London) my long-time friend who helped me a lot with constructive comments and PhD guidance.

I would like to thank my colleagues for always being very enthusiastic about discussing technical and social aspects of PhD life! Konstantinos Ferles, George Balatsouras, George Fourtounis, George Kollias, George Kastrinis, Anastasis Antoniadis, Dimitris Galipos, Stamatis Kolovos and Efthymios Hadjimichael.

I would like to thank Christine for her moral support over the course of my PhD studies.

Last but not least, I am deeply grateful to my family and friends for their unparalleled support and encouragement to complete this dissertation.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Programming languages have started shifting away from the sequential programming model that the von Neumann architecture so vigorously imposed [8]. Instead of thinking in terms of *commands* and static *storage*, modern programming needs often encourage thinking in terms of processes and transformations over *streams* of data. That transition happened over several decades of research and development in programming languages, systems, and computer architectures. *Streaming* functionality is a prominent representative of this trend. Casually speaking, in computer science, a stream is a sequence of elements that can be piped through a series of transformation steps. A streaming library is a software library to manipulate streams. All streaming libraries seem to fulfill similar goals; however, their vastly different characteristics make them one of the most fascinating areas of software construction.

This dissertation investigates the modern design decisions behind the streaming libraries that are used in general-purpose programming. We identify the key high-level differences between various implementations and observe that future use cases are tied with past design decisions and simple abstraction mechanisms are not sufficient. *Is it possible to modularize the implementation of streams to enhance such libraries in terms of extensibility and performance?* We present a twofold modularization of streams. Firstly, we untangle streams from the definition of their syntax and semantics, and secondly, we liberate them from the need of a "sufficiently-smart" compiler. The utmost goal of this dissertation is to make streams extensible and performant, while maintaining their high level structure.

Nowadays, streaming libraries let us model algorithms as if data were in motion and not stationary: stock ticks, tweets, sales, products, inventory and real-time analytics are only some of the examples that generate petabytes of information available to data scientists. In terms of conceptual modeling, a stream corresponds to a pipe transporting gas or liquids over long distances. Materials are being processed in location A where an activity f takes place. After processing ends, each element is put in the pipe and is transferred to another location B, where f' takes place. The pipe represents the flow of data and the activities f and f' represent transformations on each element of the stream. We have only declared *what* activities take place and not *how* each transformation works. "Stream processing lets us model systems that have state without ever using assignment or mutable data" per the authors of *Structure and Interpretation of Computer Programs* [1]. Douglas McIlroy writes: "We should have some ways of connecting programs like garden hose–screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.". Douglas McIlroy[1] strongly expressed his vision and motivates the creation of Unix pipelines by Ken Thompson [143, 144]; a way to manipulate streams of I/O through pipes of commands.

The f/f' pipeline can be made a bit more concrete in the following code. The pipeline consists of a source (e.g., an array, or unbounded sequence of elements such as tweets)

[1]Advice from Doug McIlroy–`https://web.archive.org/web/20160601063841/https://www.bell-labs.com/usr/dmr/www/mdmpipe.html` .

channeled through three transformation steps; two intermediate and one terminal.

```
// conceptually
source → f → f′ → terminal
```

Figure 1.1: Stream pipelines

The code below demonstrates a stream pipeline in a concrete programming language (F# using the `Seq` module) where the channeling is realized by a pipe operator (a programming style familiar to us by way of Unix pipes). One should note that the pipe operator is nothing more than function application in reverse! Assuming that we have a stream of numbers, we can filter it and print the number of items that satisfy a predicate:

```
// in a real programming language (just as Unix pipes!)
stream ▷ Seq.filter (fun x → x > 2) ▷ Seq.length ▷ printfn "%d"

// nothing more than function calls!
printfn "%d" (Seq.length (Seq.filter (fun x → x > 2) stream))
```

Figure 1.2: Stream pipelines in programming languages

A streaming library is typically offered with a set of operators to create streams, transform and consume them into scalar or other kinds of data structures, as shown in Figure 1.3. Its distinguishing feature, in relation to simple collection APIs, is that intermediate transformations are performed *on-demand*, thus *they do not perform more computation than needed*. Producer operators can be either backed by an in-memory data structure or not. `of_arr` creates a stream out of an array and `unfold` builds a (possibly unbounded) stream from a seed value (it unfolds a whole stream from a single value). Next we discuss operators that transform a stream. A stream can be transformed either in a linear or a non-linear way. `map` applies a function f to each element of the input stream and returns a transformed stream. The number of elements on the input streams is equal to the number on the transformed stream. This is where linearity comes from. On the contrary, `filter` applies a predicate to the input stream, again element-wise and unless the predicate is satisfied, the element does not appear on the output stream. Other operators, like `take`, sub-range the input stream based on a counter value. `flat_map` applies a function to each element; the results of the function application are concatenated to form the output stream, which can be a stream of zero, one or more elements. `zip_with` merges two streams according to a zipping function, applied element-wise over two streams. As expected, `zip_with` can have variations on the number of input streams, as well as a default behavior, like zipping two elements into a pair (called simply `zip`). Finally, we have consumers, like `fold` which apply a binary function, combining all elements of the stream. `fold` is a standard recursion operator for processing lists and can be used to fold a stream (like folding a piece of

paper) into something else: `sum`, `product`, `max`, `min`, `count`, boolean operators like `or` and `and`, and `concat` are only some operators that can be implemented in terms of `fold`.[2]

```
Producers of finite
val of_arr   : 'a array → 'a stream

Producers of possibly infinite
val unfold   : ('state  → ('a * 'state) option) → 'state → 'a stream

Transformers of linear nature
val map      : ('a → 'b) → 'a stream → 'b stream

Transformers of non-linear nature
val filter   : ('a → bool) → 'a stream → 'a stream
val take     : int → 'a stream → 'a stream
val flat_map : ('a → 'b stream) → 'a stream → 'b stream

Transformers of parallel loops
val zip_with : ('a → 'b  → 'c) → 'a stream → 'b stream  → 'c stream

Consumers
val fold     : ('state → 'a → 'state) → 'state → 'a stream → 'state
```

Figure 1.3: Stream Operators

For example we could write a mini sentiment analysis over a dataset of tweets. Assuming that we use a library that performs sentiment analysis over a tweet, identifying the emotional state of the user, we can create a pipeline of tweets about PhD life, filter only positive tweets and get the profile information of five users in a list.

```
tweetsDataset ▷ filter(fun t → t.contains("#phdlife"))
  ▷ filter(fun t → SentimentAnalysis.detectSentiment(t) == POSITIVE)
  ▷ map(fun t → t.user)
  ▷ take 5
  ▷ fold(fun acc x → x :: acc) []
```

Figure 1.4: A sentiment analysis for tweets

[2] In fact, fold is highly powerful and standard operators like `map` and `filter` can also be implemented in terms of it.

1.1 Stream: A Ubiquitous Term

This dissertation is framed by the history and advancements of libraries that offer streaming functionality. “Stream” is an overloaded term in the field of programming languages and systems with a very long history and various semantics. It has been used to identify: 1) application level abstractions in programming languages for general-purpose stream programming, 2) programming paradigms (dataflow programming languages such as Lucid [179], functional reactive [44]), 3) concurrent instruction (or control) streams and data stream architectures such as SIMD, 4) frameworks for implementing device drivers and network protocols, 5) streaming abstractions for high-performance computing [171] targeting heterogeneous architectures such as combinations of FPGA and a multi-core architecture, 6) circuits for digital signal processing, 7) batch data processing platforms such as Apache Flink [28], Apache Storm [5], Apache Spark [193] 8) streaming algorithms [119] and many more. Stephens’ survey presents a detailed analysis of streams’ differences between dataflow, reactive and stream processing in hardware design [161] and the various directions are discussed. Johnston et al., in their 2011 survey [77], give an overview about advances in dataflow programming languages and Zhang et al. present advancements in parallel and distributed processing systems for big data [194]. Our work context is summarized by the first of the aforementioned domains; on streaming libraries for **general-purpose stream programming**.

1.2 Two Modern Needs: Extensibility & Performance

The rationale behind streams for general-purpose stream programming is that they can be used for data processing by providing a minimal and easy-to-use abstraction. Separating a large, iterative algorithm into small pieces of functionality increases understandability, or per the author of Stream Processing Functions [24]: “using expressions that denote objects of computation as opposed to using instructions that denote machine behavior”. However, the design decisions behind streaming libraries tie future use cases with their implementation. Consider the mainstream, VM-based, multi-paradigm programming languages C# (through the `System.Linq` namespace) and Java 8 (through the `java.util.stream` package), which offer vastly different designs for streams. While the first offers a `zip` operator, the second does not, sacrificing the functionality in favor of performance. Another example is that Java 8 streams, due to their internal structure (and the sophistication of the VM), significantly outperform C# in a number of occasions. On the flipside, C# guarantees laziness in more cases, often permitting higher memory efficiency.

Two key observations motivate our study. The first is that streams need not be tightly coupled to either their implementation or the range of operators they support. The user can freely change the underlying semantics for any reason. To achieve this we view the API of a streaming library as a domain specific language (DSL). By considering it as a DSL we can study both its syntactic and semantic elements. In order to modularize a streaming library on both, we isolate the functionally-inspired API that all stream APIs share, and we

propose an extensible design. This will give users the opportunity to use different flavors of streams at will. One flavor could boost performance, another could trace execution steps, yet another could be the combination of the two!

The second observation is that modern libraries rely either on extensible compilers or on a "sufficiently-smart" dynamic compiler to generate efficient machine-level code—as if the original source code had been loop-based, handwritten code with state, mutation and ... human intuition. In the first case, for example, Haskell provides rewrite rules on the `GHC.Base` and `GHC.List` modules to perform elimination of intermediate data structures. The rules are applied at compile time [58, 165]. Library authors following this strategy usually maintain two code bases (possibly in the same compilation unit; yet programming two different things): a) the library itself (following a certain pattern), and b) the optimizations in the form of rewriting rules. For the second case, of a "sufficiently-smart" dynamic compiler, the underlying VM technologies are exceptional pieces of engineering and tremendously complex, like the Java Hotspot Server Compiler [127]. However, sometimes it is difficult to predict their behavior. For example the point that a streams is used may fail to inline (unfold its body) so the quality of the expected loop can be very poor. [3] In this dissertation we view these optimizations as **domain–specific** entirely and implement them explicitly in the stream library itself.

1.3 Contributions

Our contributions are preceded by a performance assessment of the current state-of-the-art of stream libraries. Subsequently, we first propose a mechanism to enhance the maintainability of streams, supporting a high-level of extensibility. We treat streams as a domain-specific language and we apply a modern design pattern to achieve the ability to plug-in operators and semantics. We port the extensibility mechanism we used for streams to Java itself. We show how to create dialects in Java, override its semantics, support new syntactic elements and much more. For exposition, among many examples and case studies we build an extension of Java with a keyword that enables us to construct streams à la C#. The culmination of our work is a library design for very efficient streams while preserving their high-level nature.

The content of the thesis is divided into eight chapters.

Chapter 2 introduces the necessary background material to assist the reader through the techniques applied in the next chapters.

Chapter 3 discusses standard streaming libraries in four mainstream, VM-based programming languages C#, F#, Java and Scala. We assess their performance[4] in sequential

[3] A quote by John Rose discussing two design strategies for Java 8 Streams on the [hotspot-compiler-dev] mailing list: "HotSpot are less good at internal iterators. If the original point of the user request fails to inline all the way into the internal looping part of the algorithm (a hidden "for" loop), the quality of the loop will be very poor. "—`https://web.archive.org/web/20170322141224/http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html`

[4] `https://github.com/biboudis/clashofthelambdas`

and parallel executions, on both Linux and on Windows (based on Biboudis et al. [15]). We present the design choices and composition mechanisms behind every library and we discuss the performance implications.

In Chapter 4 we propose a new library design for streams,[5] **StreamAlg**, emphasizing modularity and re-usability (based on Biboudis et al.[16]). In this new library design we view streams as a domain-specific language and we apply a new solution (Object Algebras) to an old problem (the *Expression Problem*). New operators and new behaviors can be implemented without the need to recompile the code. The results of the third chapter frame the motivation behind this library. Instead of supporting a single encoding of stream semantics, with direct (and fixed) implications on performance, the programmers can select the best one that fits their needs. Streams are now considered *à la carte* and they are not coupled with their implementation in a monolithic way. In addition to extensibility on the axis of behavior existing implementations can also accept new operators, effectively evolving the API of the library. Experiments show that our abstraction is used merely for pipeline construction and performance is indeed controlled in a modular fashion.

Chapter 5 presents a generalization of this technique, applying it this time to Java[6] (based on Biboudis et al. [14]). While other languages support extensibility at the language level natively, Java does not. This chapter proposes a lightweight tool, **Recaf**, that offers these features to Java without extending the existing compiler. We use streams as a motivational device through our case studies, by creating new extensions at the language level, to support alternative stream designs.

Chapter 6 deals with the problem of creating fast streams and we propose a library design, **Strymonas**, for fast execution of pipelines[7] (based on Kiselyov et al. [91]). We apply a type-safe, meta-programming technique to eliminate constant factors from the composition of pipelines. Effectively we present a library that acts as a code generator itself. The generated code consists of fused code, simple control constructs such as `while` and `for`-loops as well as manipulation of state directly.

Chapter 7 mentions related work and we draw parallels among existing research directions.

In the final Chapter 8, we discuss improvement areas of our techniques and future research directions followed by a final assessment of the dissertation.

1.4 Credits

The contents of this doctoral dissertation are based on published papers that were written in collaboration with others. Specifically:

- *Clash of the Lambdas* [15]; joint research with Nick Palladinos and Yannis Smarag-

[5] `https://github.com/biboudis/streamalg`
[6] `https://github.com/cwi-swat/recaf`
[7] `https://github.com/strymonas`

dakis.

- *Streams à la carte* [16]; joint research with Nick Palladinos, George Fourtounis and Yannis Smaragdakis.
- *Recaf: Java Dialects As Libraries* [14]; work done while the author was affiliated with CWI; original design and implementation by the author, Pablo Inostroza and Tijs van der Storm; implementation of expression-level extensibility and corresponding applications by Pablo Inostroza.
- *Stream Fusion, to Completeness* [91]; original design by Oleg Kiselyov with help by the author on implementation and evaluation, jointly with Nick Palladinos and Yannis Smaragdakis.

2. BACKGROUND

Streams have a long lineage in computer science and we present a brief analysis of their background from an applicative (i.e., functional) perspective. Our purpose is to give a general overview and necessary background information on this dissertation's topic. We put terminology about streams in context, we present implementation strategies through history and what motivated them. Our analysis is not a complete survey on the history of streams but a glimpse through the lens of their functional programming roots, their extensibility and performance characteristics, all necessary ingredients for the reader to comprehend what follows. Additionally, we present the necessary background information on the design pattern we use for the Chapters 4 and 5. We conclude with the two metaprogramming technologies that we rely upon, for Chapters 5 and 6, namely the metaprogramming language Rascal, and Multi-Stage Programming languages (MetaOCaml and LMS), respectively.

2.1 Streams: an expressive tool for data processing

In order to understand the nature of modern streams we have to take a good look at the past. List processing has been a key functionality of functional programming languages and a certain feature called *laziness* will play a key role in our modern view of streams. However, modern, multi-paradigm programming languages like C#, F#, Scala and Java are not lazy. Since they do provide ways for stream processing, embedded into the language, how do they encode such a ubiquitous feature as laziness? We will discuss how these languages emulate laziness through coroutines. Finally, as we reach a good level of abstraction in our programs we pose a question: *how do we regain what we lost, namely, performance?* Lazy functional programming contributed significantly to modularity, separating consumers and producers of values. However, performance is regained through a process called *stream fusion* and in this section we will present why we need it. All of the aforementioned ecosystems provide implicit or explicit ways to retain the nice abstraction without regretting it.

2.1.1 Stream Origins: Peter Landin

The term *stream* was coined by Peter Landin [97] in 1965. He was the first to observe that a denotation of Algol-60's `for`-statements with `for`-lists leads to an interpretation of list processing with a different sequencing of evaluation. Intermediate results "never exist simultaneously" in this kind of list and calls this function a "stream". The roots of streams however, can be traced back two years earlier with a concept called *coroutine*. Landin recognized that functions that operate over streams can be considered as the functional analog of Conway's coroutines.

Conway introduced coroutines [37] to separate the internals of a COBOL compiler into

modules that communicate with each other directly, instead of having independent transformation passes. By generalizing routines and sub-routines, he introduces coroutines, which can yield execution explicitly or implicitly to other coroutines. Conway needed to create a one-pass compiler implementation and his motivating example demonstrates exactly that! He demonstrates coroutines through a state machine that processes a string of characters. The program should output the string but adjacent asterisks need to be replaced by another symbol. A one-pass operation is better and faster than two-pass in this problem! The key insight here is that a one-pass operation channels each element through the whole pipeline. On the contrary, a two-pass one performs an activity A, collects the results, and then channels the whole batch to activity B. Conway wanted to demonstrate that, with coroutines, *intermediate results* can be avoided.

W. H. Burge in 1975 [24] proposes a set of stream processing functions and transfers the benefits of coroutines to lists. Not only does he rely on Landin's stream definition but he also migrates his list-processing operators [23] to stream processing, laying the practical foundations for recursive programming techniques. Some operators are the following: `generate`, `map`, `zip`, `filter`, `append`, `concat`, `while`/`until`, `map2`, `sum`, `length`, `reverse`.

2.1.2 Lazy Evaluation: Data Processing Building Blocks

The need for laziness has played a key role in software construction for a wide range of reasons and it appears in many aspects of programming languages in various forms. In this section we examine the uniform support of functional programming languages for laziness and more specifically in Haskell. An attempt to understand the benefits of laziness as a language feature will give us better insight when we try to seek the same property in streams. However, laziness is orthogonal to performance. Later on, we discuss which directions are going to help us create lazy but efficient streaming libraries.

John Hughes in *Why Functional Programming Matters* [72] argues that modularity is of paramount importance for productivity. Modularity describes the ability to divide a problem into sub-problems, solve them and glue the solutions back together. As a consequence, functional programming promotes modularity by higher-order functions and laziness, but how? In short, higher-order functions and currying are extremely powerful tools to modularize an algorithm. Functional programmers can even extract recursion out of a particular problem, in a modular way.

```
data List a = Nil
            | Cons a (List a)
```

Figure 2.1: List data type

Consider the list data type for a moment, where one instance of a list can be `Cons 1 (Cons 2 Nil)` constructed with `1:2:[]` syntactically. This is the list of `[1, 2]`. Notice the recursive nature of the list datatype. The two data constructors are enumerated as the

cases of the type. `a` is the type parameter and the second constructor reuses the definition of the list. `Cons` has two arguments, an element and another list.

Previously, we talked about reusing that recursive nature modularly. If only we could replace ":" with a function application `f` and `Nil` with a value of type `a`. The action of replacing `Cons` and `Nil` could be performed inductively, by pattern matching over the structure of the list. Indeed, Huges [72] demonstrates that we can create list processing functions based on `foldr` by defining that particular `f` and a specific value of type `a`. For performing a sum operation, `f` is the function that performs addition, `(a,b) => a + b`, and 0 is the starting value.

```
-- short circuiting (e.g. in boolean expressions)
(&&) :: Bool -> Bool -> Bool
True  && x = x
False && x = False

-- circular data structures
circular = 0 : 1 : circular

take 5 circular -- [0,1,0,1,0]

-- infinite computations
iterate :: (a -> a) -> a -> [a]
iterate f x =  x : iterate f (f x)
```

Figure 2.2: Examples that benefit from laziness

The second ingredient, laziness, enables composed functions to consume values on demand, by propagating values from a producer to a consumer. Imagine that an algorithm generates a large set of answers, potentially infinite. Instead of handling the (potentially) infinite structure of an algorithm explicitly, or, even worse, attempting to store all elements in memory by spilling all kinds of technical details on our algorithm, laziness offers that exact abstraction universally. As an example, in a function composition $g \circ f$, we say that f will produce data (e.g., an infinite list of naturals) and g will consume until a condition is reached. As long as g needs data, by evaluating the next element of the list, f will continue to produce it. To give an example of an f let's consider `iterate` in Figure 2.2. It returns a list, such that each element is the result of successive applications of `f`, as if the answer is a "fully populated" infinite list of `[x, f x, f (f x), f (f (f x)), ...]`.

Laziness (discovered independently by researchers [53, 65]) is inherent in some programming languages (Haskell is the most notable representative) while others (*strict* by default) can emulate that behavior by wrapping values inside thunks. We must note that strictness is related to laziness but they are distinct concepts.[1]

[1] Lazy vs. Non-Strict: "Obviously there is a strong correspondence between a thunk and a partly-

So, what makes Haskell lazy? Two, separate strategies [132]. Understanding the process that makes laziness a built-in feature of Haskell will give us useful insights about laziness in general.

2.1.2.1 Reduction strategy: strictness

Strictness semantics define the direction of the reduction of the graph that represents a Haskell program to its normal form. Three concepts are involved in this informal definition. To begin with, reduction denotes the evaluation of an expression by replacing a subexpression. For example the function application `square 2` with definition `square x = x * x` can be reduced to `2 * 2` by replacing the argument in the body of the function. In Haskell, a program is translated into lambda expressions internally and it is represented by a graph. Normal form is when an expression does not have any subexpressions left to reduce. Strictness semantics define the direction of whether we reduce all arguments first (innermost reduction) and then the function application (strict), or apply the top function first (outermost reduction) and then the arguments (non-strict). Haskell has non-strict semantics and the arguments are reduced only when they are needed. This gives Haskell the capability of supporting infinite data structures and streams. Consider the implementation of the logical && in Haskell as shown in the first example of Figure 2.2. It specifies that the operator can yield a result when the first argument is false no matter what the second argument is. This means, that the code works, even when the second argument represents a computation that does not have a value (an error or an infinite loop can be denoted with bottom or $\perp$ in mathematical notation). As a result, the computation of the second argument was never needed, thus it was not performed and we can write our code in the most elegant form. That of a short-circuiting function definition. While other programming languages support short circuiting for boolean expressions, Haskell supports it for all functions. For completeness, in a strict language $f\perp = \perp$. Haskell's ubiquity comes from the fact that all structured data, such as lambda abstractions, data types and built-in functions participate in the aforementioned process. Streams in strict languages have to encode such laziness manually.

2.1.2.2 Evaluation strategy: call-by-need

The evaluation strategy is part of the operational semantics of a language. It determines when to evaluate the arguments, namely to share already computed results (of the same redex) or re-evaluate them (think about applying `3+3` to `square`). Since we explore the design space through the lens of laziness, one evaluation strategy for non-strict programming languages is *call-by-name*. Under this strategy, the argument `x` in `(fun x -> e2)`

evaluated expression. Hence in most cases the terms "lazy" and "non-strict" are synonyms. But not quite. For instance you could imagine an evaluation engine on highly parallel hardware that fires off sub-expression evaluation eagerly, but then throws away results that are not needed." –`https://web.archive.org/web/20170321161701/https://wiki.haskell.org/Lazy_vs._non-strict` .

when applied to `e1` is replaced with the unevaluated `e1` in `e2`. Another strategy is *call-by-need*, where subsequent evaluations of the same expression will return the same result and will have been computed only once (using thunks). After the result is computed, it is stored for subsequent evaluations (*memoized*). The Haskell language does not specify the order of evaluation. The only thing that drives the evaluation is the concept of *data dependency*.

Lists in Haskell are lazy and they can represent infinite lists. Since data constructors are lazy as well, `Cons` by having a non-strict, call-by-need framing, does not need to evaluate its second argument unless it is needed. That argument is also a list, so by pattern matching over that (`x:xs`) we can get the head of the list (effectively the first element of the list data type, leaving the rest again unevaluated.) Haskell is lazy by nature and since the topic of this thesis is streams, it is worth noting that by examining the `List` datatype we have already seen the first implementation of streams. Subsequent discussions that mention the term strictness for libraries will share a similar meaning to Haskell: non-strict will mean lazy in terms of getting elements on-demand, strict will mean that intermediate collections will be emitted.

2.1.3 Infinite Streams in Strict Languages

In Figure 2.3 we view the list definition again, but in a strict language. In order to support both finite and infinite streams, we have to introduce the same mechanism explicitly; thunks. Firstly, we show the transition from list to its lazy counterpart. `list_shape` represents the shape of streams in Chapter 6; the list with recursion factored out.

```
(* lists in OCaml *)
type 'a list =
  | Nil
  | Cons of 'a * 'a list

(* infinite lists in OCaml with thunks *)
type 'a lazy_list =
  | Nil
  | Cons of 'a * (unit → 'a lazy_list)

(* list shape *)
type ('a,'z) list_shape =
  | Nil
  | Cons of 'a * 'z
```

Figure 2.3: Implementing streams in a strict programming language

The two data types in Figure 2.4 demonstrate two forms of streams. The first one is inspired by Alphard [156] and needs to capture state and other variables to act as the step

function of an iterator. The second, and last one, represents the final form of streams, after closure-conversion has been applied as an optimization to the stepper function. The reason for optimizing the thunk is that we want to avoid the capturing of free variables and make them explicit instead. Finally, the reader will notice that the final definition uses the GADT syntax of OCaml which treats all types that do not appear on the type itself as existential. The iterator is a special form of unfold or anamorphism [68, 115].

```
(*  compact form of a stream *)
type 'a stream = Cons of 'a * (unit → ('a, 'a stream) list_shape)

(*  compact form of stream with closure conversion to
    explicitly pass the state to the stepper function *)
type 'a stream = { stepper : 's * ('s → ('a,'s) list_shape) → 'a stream }
```

Figure 2.4: Two forms of streams

The aforementioned data-type can be considered the foundation of our design presented later, in Chapter 6, implementing stream fusion.

2.1.4 Programming Languages: Towards Language Integration

Laziness consists of an operational semantics that supports lazy evaluation and a denotational semantics that supports non-strictness. These two combined provide a powerful tool as we have seen. However, when our programming language supports neither of those, we can emulate laziness with additional machinery. Using generators, for example, or other explicit forms of traversal i.e., iterators, we can blend functionally inspired programming paradigms that rely on laziness, in mainstream, multi-paradigm programming languages.

Lists have played a key role through the history of programming languages (LISP [109]) and the set-builder notation from mathematics inspires the creation of a new syntactic construct that represents the creation of a list in terms of other lists. Take for example the following set in set-builder notation: $S = \{ x \cdot x \mid x \in \mathbb{N}, x \mod 2 = 0\}$. It describes the set of all squared numbers, such that x is an even number. SETL [153], Smalltalk [60] and Miranda [174] all supported *list comprehensions*, a feature that also exists in modern languages like Haskell, JavaScript, Python, OCaml and Scala. Wadler introduced the term *list comprehensions* [182] in 1985. List comprehensions offer a convenient way to express mapping, filtering and cartesian product for lists (specification and transformation rules are presented in the Haskell 98 report [81, Section 3.11]).

Following the history of the pure functional programming world of Haskell [71], another advancement takes place. Researchers investigate ways to retrofit impurity (side-effects, state, exceptions, etc) in a pure language. Without going into much detail here, *monads*

introduced by Moggi to tame effects in denotational semantics [117], inspired Wadler to reuse them as an elegant abstraction to tame effectful computations in Haskell [180, 184].

Figure 2.5 provides the basic class for any type `m` that can be considered an instance of a monad [81, Section 6.3]. Note that `m` is a type constructor so `m a` is the type of computation that may produce a value of type `a`. In short, `return` injects a value in the monad (think about a container) and `>>=` (also known as "bind") will combine a monadic value of type `m a` with a function that applied to a type `a` will produce another monadic value of type `m b`.

```
class Monad m where
   return :: a -> m a
   (>>=) :: m a -> (a -> m b) -> m b
```

Figure 2.5: Monads in Haskell

Code can be written either explicitly or implicitly, by using the *do-notation* syntax, which gives an imperative look-and-feel to our functional program. In the code below, `m` represents a monadic computation and `v1` a value produced by that computation. If the monad has type `m a` then `v1` has type `a`. Finally, `f` is a function that depends on `v1`.

```
do v1 <- m
   action

-- translated to
-- m >>= (\v1 -> action)
```

Figure 2.6: Do-notation syntax in Haskell

As we have mentioned in the previous section, Haskell does not specify an ordering of evaluation. Monads, on the other hand, restore *some* of the ordering by enforcing a *data dependency* on the first parameter of `(>>=)`. This makes ordering of monadic actions explicit. For more information on ordering the reader can refer to Jones' lecture notes [131].[2]

Wadler [180] generalizes list comprehension syntax to any monad (giving rise to the aforementioned do-notation syntax, historically). The general form of comprehension syntax can be defined in terms of a `unit` method, an interpretation of the generator form $c \leftarrow x$ with a `map`-based translation, filters with an `if-then-else`-based translation and the comma-separated list of qualifiers into a `join`-based translation (different name for `concatMap` or `>>=`). Wadler shows that, with additional laws, this syntax is able to cover a wider range of computations, not only just sets and lists.

[2] IO inside–`https://web.archive.org/web/20170321162041/https://wiki.haskell.org/IO_inside`

Another way of viewing monad comprehensions is as a monadic domain-specific language, and this is exactly what the C# team members and Erik Meijer did with Language Integrated Query (LINQ) [110, 112, 114]–also `.NET`'s stream library! They reused ideas from monad comprehensions and provided a unified solution for manipulating database objects, in-memory objects, and XML documents.

The following figure contains two language-integrated queries. This is valid syntax that gets desugared at compile-time.

```
from p in Enumerable.Range(0, int.MaxValue)
where p % 2 == 0
select p

// and

from p1 in Enumerable.Range(0, 5)
from p2 in Enumerable.Range(0, 5)
select p1 + p2
sum()
```

Figure 2.7: Language Integrated Query

The C# compiler generates the code in Figure 2.8, for the code in Figure 2.7.

```
Enumerable.Range(0, int.MaxValue).Where<int>((Func<int, bool>) (p => p % 2 == 0))

// and

IEnumerable<int> source = Enumerable.Range(0, 5);
Func<int, IEnumerable<int>> func = p1 => Enumerable.Range(0, 5);
Func<int, IEnumerable<int>> collectionSelector;
source.SelectMany<int, int, int>(collectionSelector, ((p1, p2) => p1 + p2));
```

Figure 2.8: Language Integrated Query, desugared

In the examples above, we see the parallels with monad comprehensions. `Where` implements filtering, two consecutive `from` expressions are desugared into a `SelectMany`-based form (`SelectMany` is `.NET`'s `concatMap`). C# is a strict language and LINQ a library that supports deferred execution. Next, we present the basic set of interfaces (`IEnumerable` and `IEnumerator`) that the reader may have noticed in the previous example. We use the underlying concept of *iterator blocks* in C#, which provide support for generators.

2.1.5 Generators and Laziness

C# 2.0 introduced an implementation of coroutines as described in [43, Chapter 26] and later revised in [76]. In C#, iterators are statement blocks that can *yield* values in an ordered fashion. When a function returns an `IEnumerable` object, invoking it does not involve any traversal execution but an `IEnumerator` is returned instead.

```
class Fibonacci(var state : Int) {
  var fn, fn1, fn2 : Int = 0

  def next() : Int = {
    state match {
      case 1 => {
        fn = 0
        fn1 = fn
        state = 2
        fn // yield fn
      }
      case 2 => {
        fn = 1
        fn2 = fn1
        fn1 = fn
        state = 3
        fn // yield fn
      }
      case 3 => {
        fn = fn1 + fn2
        fn2 = fn1
        fn1 = fn
        fn // yield fn
      }
    }
  }
}
val f = new Fibonacci(1)
for (i <— 1 to 10) println(f.next())
```

Figure 2.9: Fibonacci simulated with coroutines, in Scala

`IEnumerator` encapsulates a state machine of four possible states: *before*, *running*, *suspended*, and *after*. The state machine models suspension and resumption of enumeration, using a compiler-generated enumerator class. That class, informally, keeps tabs on the execution state of the iteration. When a caller iterates through the `IEnumerable` by invoking `IEnumerator`'s `MoveNext()` method and the `T Current` property, the enumerator per-

forms an execution step and the calculated value is made available through the property. The body of the method that returns an `IEnumerable` or `IEnumerator` can include one or more `yield` statements. Each statement yields the execution back to the caller. The execution of the suspended function is resumed when the caller invokes the `MoveNext()` function again.

Iterators have deep links to generators (IPL-V [121]) and can be considered to be coroutines, modeling resumption and yielding of execution to other coroutines. In Simula [39], when an object, A, maintains a procedure-like, coroutine-based relationship with another object, B, it is considered a semi-coroutine. Coroutines, in general, are able to return a value during execution, while also yielding the execution to another coroutine (either implicitly or explicitly). Immediately after yielding, execution is transferred and the coroutine is suspended. All local data and the location (continuation point), where the execution was yielded, are saved. Historically, iterators in modern programming languages like C# adopt a generator facility such that of Alphard's [156] and CLU's [103]. These programming languages provided support for generators solely for the generalization of iteration. There are two kinds of coroutines [108]: 1) semi-coroutines can yield execution implicitly only to their caller (asymmetrical) and 2) full-coroutines can activate explicitly other coroutines (symmetrical).

In modern generator implementations we notice the basic elements of the semi-coroutine relationship. In Simula that relationship is established by a `call(X)`/`detach` mechanism [39]. Modern iterators (e.g., in C#) offer the same pair, in the form of `X.MoveNext()` and `yield`.

The `Fibonacci` example in Figure 2.9 (more analysis in Buhr [22], in which the same example is examined) is presented using Scala and demonstrates that Fibonacci is a class that contains one method, `next()`. When the method is invoked, it calculates the next result, saves the current data and the current state. State emulates the preservation of the continuation point among executions. This particular demonstration gives the necessary intuition regarding the program transformation of `yield` that happens in C# and other languages. In this example, we notice that it is the client that drives the execution of the semi-coroutine. The last observation is the cornerstone of what we are going to call *pull-based* streams in the following chapters of this dissertation (discussed in the next section, 2.1.6). To conclude, we presented generators and their relationship to iterator blocks in a strict programming language that offers monad comprehensions.

2.1.6 Two styles for On-Demand Processing

This dissertation discusses the two distinct design choices (push vs pull) behind streaming libraries several times. In Chapter 3 we identify them in mainstream, multi-paradigm, VM-based ecosystems, in Chapters 4 and 5 we investigate extensibility mechanisms of streams (and programming languages in general) and in Chapter 6 we offer a unified design, partially evaluated. Although we will revisit their definition four times, we present in Figure 2.10 their high level structure in Scala. Both figures present three operators. One

```
class Stream[T]
(val sIter: Iterator[T]) {
  def map[R](f: T => R) = {
    val new_sIter = new Iterator[R] {
      def hasNext = sIter.hasNext
      def next() = f(sIter.next)
    }
    new Stream(new_sIter)
  }

  def fold[A]
      (a: A)
      (op: (A, T) => A): A = {
    var acc = a
    while (sIter.hasNext) {
      acc = op(acc, sIter.next())
    }
    acc
  }
}

object Stream {
  def of[T](xs: Array[T]) = {
    val new_sIter =
      new Iterator[T] {
        override val size = xs.length
        var i = −1
        def hasNext = {
          i+=1 ; i != size
        }
        def next() = {
          if (i >= size)
            throw new Exception()
          xs(i)
        }
      }
    new Stream(new_sIter)
  }
}
```

```
class Stream[T]
(val sF:
   (T => Unit) => Unit) {
  def map[R](f: T => R) = {
    val new_sF =
        (iterf: R => Unit) =>
      sF(v => iterf(f(v)))
    new Stream(new_sF)
  }

  def fold[A]
      (a: A)
      (op: (A, T) => A): A = {
    var acc = a
    sF(v => {
      acc = op(acc, v)
    })
    acc
  }
}

object Stream {
  def of[T](xs: Array[T]) = {
    val new_sF =
       (iterf: T => Unit) => {
      var i = 0
      val size = xs.length
      while (i < size) {
        iterf(xs(i))
        i += 1
      }
    }
    new Stream(new_sF)
  }
}
```

Figure 2.10: a) Pull-based design (on the left) and b) Push-based design (on the right)

to produce a stream from an array (`of`), one to transform streams (`map`) and one consumer (`fold`).

The design on the left is a pull-based design. A pull-based stream is essentially an iterator. The producer creates an iterator that drives the traversal of an array on-demand. The transformer transforms the passed iterator, by applying the transformation function before returning the next element. The consumer uses the passed iterator to perform the iteration and, for this reason, we say that the stream is *lazy on the consumer*.

A push-based stream is essentially a higher-order function of type `(T => Unit) => Unit`. A stream holds this function, which represents the entire iteration over the elements of an array, with another function applied to each one. The return value of the parameter serves to propagate information down to the next consumer in the operator chain (in this small example it is `Unit` as it is ignored).

The iteration itself is encoded as a loop (hereafter: the `iterf` function). The producer creates such a stream with the initial looping function passed as the `iterf` function. The transformer creates a new stream function. It applies a new iteration function on the iteration function of the previous operator via function application. The consumer will evaluate the stream function, transforming again the iteration function if needed. In this case `fold` holds the state in a separate accumulator, initializes it according to the passed `state`, and returns it. Other terminal operators, such as `sum`, `count`, and other eager operations, such as `toArray`, can be implemented in terms of `fold`. The producer encodes the iteration from the beginning in a suspended computation and we say that the stream is *lazy on the producer*.

In the following chapters we investigate the performance of both designs through the lens of four standard implementations from various programming language ecosystems that follow one or the other strategy. Additionally, we introduce mechanisms to make streams extensible and performant along the axes of these designs.

2.1.7 Lowering the Abstraction

If we were to cite one more phrase from the famous *Structure and Interpretation of Computer Programs book* [1], it would be that: “programs must be written for people to read, and only incidentally for machines to execute”. Abelson H. and Sussman G. believe in keeping the code elegant and showed the way to many generations of computer scientists and programmers. While abstraction improves understandability, code evolution, modularity and testability, performance takes a significant hit. As we use simpler and simpler structures to design sensible programs, we get bigger and bigger performance bottlenecks by their in-between interactions. Burstall R. M. and Darlington J. [26], acknowledged the problem, in the functional programming setting, in 1977. They presented a system to systematically replace recursion with iteration as well as perform other kinds of transformations. Run-time costs come in many forms and without referring explicitly to particular technologies or design choices, we mention some of these costs below:

- excessive construction and deconstruction of values (e.g., composing and decomposing in pattern matching);
- recursive calls when an environment does not support specific optimizations (e.g., tail-call optimization);
- heap-allocated closures (e.g., lambdas capturing free variables) when the program is higher-order;
- multiple iterations over one sequence of data that can be traversed once (e.g., a map of squares pipeline over a stream);
- iterations over multiple sequences of data that can be traversed at once (e.g., zipping a stream); and
- highly polymorphic call-sites (e.g., `MoveNext()` in iterators), also known as "megamorphic", which incur dynamic dispatch ("virtual method") overhead.

Figures 2.11 and 2.12 contain two semantically equivalent, stream-style computations encoding the sum of a cartesian product: $\sum_{i=1,j=1}^{n} x_i \cdot y_i$. In Figure 2.11 we use some streaming library named `Stream`. Regardless of the design decisions that the library embodies and the smartness of the VM to optimize away constant-factors, we are immediately aware that the API is parameterized by higher-order functions which entail virtual method invocation costs. The library may use some internal structure to encode iteration that would only add more to costs of the aforementioned nature. Additionally, the lambda passed to `map` captures the variable `d`, which occurs free in the body of the mapping function. Hidden design choices may reveal additional costs!

```
def cart (x : Array[Int], y : Array[Int]) : Int = {
  Stream(x)
   .flatMap(d => Stream(y).map (dp => dp * d))
   .sum
}
```

Figure 2.11: Cartesian Product

The snippet of code in Figure 2.12 performs the aforementioned computation using structured programming with explicit iteration and state. We drive the traversals explicitly and we are able to hoist the second traversal under the first without extra allocations per element. We inlined all three lambdas: 1) the mapping function of `map`, 2) the mapping function of `flatMap` that contributes the second iteration and the summation that the `sum` operator declares internally. All state variables are grouped together prior to traversals and are explicitly set (e.g., indices are incremented). In conclusion, by code inspection we are absolutely sure that no intermediate arrays are allocated.

```
def cart (x : Array[Int], y : Array[Int]) : Int = {
   var d, dp=0
   var sum=0
   while (d < x) {
      dp = 0
      while (dp < y.length) {
         sum += x(d) * y(dp)
         dp +=1
      }
      d += 1
   }
   sum
}
```

Figure 2.12: Cartesian Product using low level constructs

We say that Figure 2.12 shows a *fused* form of the same stream. In Chapter 3 we investigate the performance characteristics of the stream libraries of four mainstream, VM-based programming languages. These libraries rely only on the cooperation of the underlying VM to exploit the structure of the libraries and execute performant code.

In stream fusion (as both a research area and an engineering problem), the mechanisms proposed over the years vary on the level of involvement of the library author, while the goal remains the same. The goal is for people to write highly abstract programs and the machine to execute fast, loop-based code. Stream fusion is an optimization that, ideally, the compiler could perform.

In Chapter 6 we propose a new system to generate low-level, loop-based and fused code for a wide range of operators. In our research we do not rely on fusion rules, declared by the library author and applied by the compiler. Instead, we rely on a type safe, meta-programming technique called *multi-stage programming*. Our library generates the fused, loop-based form of stream pipelines by *itself* eliminating constant factors from the code right away. As a result, the stock compiler obtains relatively *simple* code to execute.

2.2 Embedding DSLs with Object Algebras

Object Algebras [124] is a design pattern that we use in this work. It has already been applied in numerous cases in the literature [73, 125] and solves the Expression Problem (EP) [185]. EP, as Philip Wadler put it on the Java Generics mailing list in 1998, “is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)”. Wadler’s formulation reheated the discussions on the expressiveness of programming languages and re-

searchers responded by providing solutions of the EP. This problem had an appeal in both the OO and the functional world, justifying its popularity. Many solutions were presented over the years that made heavy use of generics, subtyping and F-bounds [173]. These solutions, however, continued to inspire subsequent attempts. Most of the time either the lack of true *self* types through F-bounds or the lack of advanced typing mechanisms, such as multi-methods and virtual classes, made the solutions not portable or cumbersome to encode.

```
trait ExpAlg[T] {
  def lit(n : Int) : T
  def add(x: T, y : T) : T
}
```

Figure 2.13: Object Algebra for simple expressions

The Object Algebras design pattern, relies only on simple generics. Instead of defining a language's abstract syntax using concrete data structures, it defines it using generic factories. A generic interface declares generic methods for each variant ("cases"). Implementations of such interfaces define a specific semantics by creating semantic objects representing operations like pretty printing, evaluation, and so on ("functions").

```
trait Eval {  def eval() : Int }

trait ExpEval extends ExpAlg[Eval] {
  def lit(n : Int) : Eval = new Eval {
    def eval() : Int = n
  }
  def add(x : Eval, y : Eval) : Eval = new Eval {
    def eval() : Int = x.eval() + y.eval()
  }
}

def test1[T](a : ExpAlg[T]) : T =
  a.add(a.add(a.lit(1), a.lit(2)), a.add(a.lit(3), a.lit(4)))

val t1_1 : Eval = test1(new ExpEval{})

// 10
println(t1_1.eval())
```

Figure 2.14: Evaluating simple expressions

In Figure 2.13 (using Scala) we present the signature of our DSL with two constructs: literals and addition. The usual way of representing that DSL is with a recursive algebraic

data type (ADT) of sums and products. On the contrary, the object algebras design pattern encodes this without an ADT by relying only on method application and abstraction. `ExpAlg` presents the *signature* of our DSL and is parameterized by the type, *sort*, of the expression involved (traits in Scala are like interfaces in Java 8 with default methods).

The first thing on our todo-list now, is to define an interpretation of the programs written in this DSL. Therefore, we define our first "function" over the datatype, performing evaluation (Figure 2.14). Similarly (without recompiling), we can define another "function", a trait `View`, that prints instead of evaluating our program (Figure 2.15).

```
trait View {
  def show() : String
}

trait ExpView extends ExpAlg[View] {
  def lit(n : Int) : View = new View {
    def show() : String = s"${n}"
  }
  def add(x : View, y : View) : View = new View {
    def show() : String = s"${x.show()} + ${y.show()}"
  }
}
```

Figure 2.15: Printing simple expressions

Next, we proceed with the addition of a new "case" to the datatype, multiplication. In the code below, we use subtyping to extend the previous datatype. We now implement the printing "function" for the new "case" (extending also `ExpView`).

```
trait ExpWithMul[T] extends ExpAlg[T] {
  def mul(x: T, y : T) : T
}

trait ExpViewWithMul extends ExpWithMul[View] with ExpView {
  def mul(x : View, y : View) : View = new View {
    def show() : String = s"${x.show()} * ${y.show()}"
  }
}
```

Figure 2.16: Extending simple expressions

Note, in Figure 2.17, that `test1` still works and that `t2_1` prints the new program (`test2` with the extended algebra, `ExpViewWithMul`).

```
// test1 works with both
// 1 + 2 + 3 + 4
val t1_2 : View = test1(new ExpView{})
val t1_3 : View = test1(new ExpViewWithMul{})
println(t1_2.show())
println(t1_3.show())

def test2[T](a : ExpWithMul[T]) : T =
   a.mul(a.add(a.lit(1), a.lit(2)), a.add(a.lit(3), a.lit(4)))

// 1 + 2 * 3 + 4
val t2_1 : View = test2(new ExpViewWithMul{})
println(t2_1.show())
```

Figure 2.17: Using the extended algebra

Independently discovered, Carette et al. [29] built the first family of tagless interpreters for a higher-order typed object language in a typed metalanguage without advanced language features. The usual approach to represent a DSL in a host language is to perform semantic analysis through the use of an Abstract Syntax Tree. Interpreters for that DSL traverse recursively an algebraic data type by pattern matching. This is called the *initial* encoding. On the contrary, tagless interpreters adopt the *final* encoding approach. Terms of the DSL are represented as expressions build through operators. These operators are regular methods and the returned values essentially map the syntactic domain to the semantic as denotations. This method is related to the Böhm-Berarducci encoding of an algebraic data type [18]. The motivation of this solution was the same as object algebras, extensibility. However the authors are also motivated by the need to use the static type checking mechanism of the host language, to type-check the embedded DSLs as well. Additionally, they provide various transformations of the interpretations e.g., from De-Bruijn–based to HOAS-based semantics, from a direct style to a call-by-value CPS one. Finally, the authors take advantage of MetaOCaml and Template Haskell to stage tagless interpreters by creating DSL compilers without abstraction penalties.

The work presented in this thesis uses the encoding of Object Algebras / Tagless Interpreters two times. Firstly in the context of streams' extensibility (Chapter 4) in Java, and secondly in the context of extending Java itself (Chapter 5).

2.3 Code generation through Metaprogramming

Through this work we have used two different kinds of metaprogramming facilities. The first one, the Rascal Metaprogramming Language [92] is a framework that is used to program code transformations. The second one follows a metaprogramming technique called

Multi-Stage Programming (MSP) and provides language constructs to compile programs at run-time. These two systems have very different background. The first belongs in the family of Language Workbenches [45] and the second comes from the partial evaluation community.

2.3.1 Rascal Metaprogramming Language: A Metaprogramming DSL

Rascal [92] is a metaprogramming language that is used to create source-code analyzers and transformers [12, 67, 163]. Rascal supports explorative programming at a very high level, providing an interactive environment to develop and experiment with construction of such metaprograms (a source-code transformer is considered a metaprogram). It is considered a functional programming language, supporting closures, higher-order functions, comprehensions, algebraic data types, generators and pattern matching. Additionally, Rascal comes with language support for more advanced features such as pattern-directed invocation for transforming syntax trees. Among the usual datatypes for general purpose programming, Rascal also offers source locations as first-class citizen values.

During parsing, the source information is extracted from source code directly either in an unstructured manner, based on regular expressions, or structured, using syntactic definitions. In the structured case, Rascal uses a context-free grammar of a programming language or DSL to create a parse tree for further analysis. *Abstract*- and *Concrete*-based are two kinds of pattern matching over a parse tree. Although Rascal provides pattern matching over algebraic data structures, a concrete patterns give a highly expressive programming model. Concrete pattern matching consists of a quoted syntax fragment that may contain variable patterns (typed or not).

```
data Exp
  = nat(int nat)
  | add(Exp lhs, Exp rhs)
  | mul(Exp lhs, Exp rhs);

Exp add(nat(x), nat(x)) = mul(nat(x), nat(2));
```

Figure 2.18: Transformation through Abstract Syntax

Figure 2.18 demonstrates a single rewrite rule for an algebraic data type that represents addition and multiplication expressions. The second equality defines a rewrite rule for the addition of two equal numbers, to a multiplication node of the number with two.

However with Rascal we are able to program code transformations with concrete syntax in the following, and more intuitive, manner. In Figure 2.19, `lexical` defines a non-terminal for numerical nodes, `syntax` a context-free non-terminal for expressions and `transform` is a function that performs top-down pattern matching. In the single case `plus`, we use the *PatternWithAction* syntax that replaces the current subtree of addition between two

```
lexical Nat = [0—9]+;

syntax Exp
  = nat:Nat
  | bracket "(" Exp ")"
  > left plus: Exp "+" Exp
  > left mul: Exp "*" Exp;

public Exp transform(Exp e) {
  return top—down visit(e) {
    case plus(n:nat(_),n) => [Exp]"<n>*2"
  }
}
```

Figure 2.19: Transformation through Concrete Syntax

terms of the same value by a multiplication of the term and the number two. The result expression, `"<n>*2"`, uses concrete syntax which also includes a reference to the captured variable `n`.

Chapter 5 uses Rascal to program a source-to-source translator that enables the creation of Java dialects.

2.3.2 Staging: Manipulate Program Fragments Type-Safely

A simple way to ensure the static safety of generated programs is to map them one-to-one to fragments of the generator's code. As a result, the generator and the generated program can be viewed as one, are type-checked by the same type system, and some parts of the program are merely evaluated later (i.e., generated). This approach is commonly called *staging* [83] and the program is called *multi-stage* or *staged*. Multi-Stage Programming (MSP) is the general paradigm for writing staged programs.

MSP views program generation as the addition of extra stages to regular programs. Instead of a plain execution stage (possibly internally broken into multiple stages, such as compilation and object execution) we have at least a generation stage and an execution phase. Programmers make use of a set of language constructs that introduce more stages for explicitly annotated segments of their code, so that these segments are evaluated at different times. At a later time, the previous (partially) evaluated, in earlier stages, parts of the program can be replaced by simpler constructs, such as constant values and statements with linear control flow.

The origins of staging constructs are found in languages like *MetaML* [170]: a statically-typed, MSP language, as an extension of Standard ML/NJ [6]. MetaML introduced four language constructs:

- the meta-brackets that delay a computation e.g., `<40+2>`. Evaluation cannot happen and the computation is considered *frozen*. These are *future-stage* computations, and can be thought of as generated code.

- the *escape* operator, `~x` for some variable `x`, can be used only inside meta-brackets—e.g., `<40+~x>`. This operator permits calculations at the current stage and splices the result inside the delayed expression for later use. This allows evaluation steps to take place during program generation, i.e., to vary the generated code.

- `run x` forces the evaluation of a meta-bracket expression. Essentially, it compiles the computation at run-time and runs it to produce the result.

- `lift x` allows the conversion of a value—the result of the evaluation of an expression that does not contain a function—into code.

Consider a function whose body contains a mix of staged and unstaged parts. What happens when we evaluate that method with an argument list consisting of some known values and some to-be-supplied at a later stage? MSP evaluates the unstaged and escaped parts of the program, utilizing information that is available at the current stage. Then it produces a residual program that is going to be evaluated at a subsequent stage, when the rest of the parameters are available. This is similar to a *partial evaluation* (PE) of the program [36, 79], where the unstaged and escaped parts are evaluated, with staged parts left for later evaluation. (Staging and partial evaluation are closely related concepts [78, 83]: staging can be seen as instructing a partial evaluator as to what parts of the program to partially evaluate. Conversely, automatic partial evaluation can be seen as computing the staging annotations automatically—a step called *binding-time analysis* in the PE literature [80].)

A very powerful capability for modern MSP languages is *cross-stage persistence* (CSP). CSP means that a value, expression or identifier bound at a stage n, is available in stages n and later. Practically this means that staged code can have the ability to stage computations that are open in the same way as lambdas can have free-variables. However lambdas are not used for program generation but staged computations are. To get a good grasp of this capability: first, imagine that your staged computation uses a function from the standard library, consumes a record or performs arithmetic with primitives. Afterwards, pose a question: your algorithm is now data. Definitions for types, bindings for variables from previous stages must be disconnected from stage to stage. CSP is a capability that ensures safety for such patterns in staged code.

Applications of MSP include the implementation of domain-specific languages [66], building compilers from interpreters [169] and the “finally tagless” approach to building efficient interpreters [29].

We next review staging in more detail via two modern staging implementations.

2.3.2.1 BER MetaOCaml for OCaml

MetaOCaml [27] is a bytecode MSP compiler for OCaml and BER MetaOCaml [90] is its continuation: a heavily re-factored version of the MetaOCaml compiler that is more extensible and easier to integrate with releases of the regular OCaml compiler.[3]

We illustrate staging via the folklore example of a simple power function, which has been used for demonstrating partial evaluation (and staging) since at least 1977 [47]. The power function is defined recursively using the basic method of exponentiation by squaring. If the exponent is even we square the result of raising x to half the given power. Otherwise, we reduce the exponent by one and we multiply the result by x.

```
let even n = (n mod 2) = 0;;
let square x = x * x;;
let rec power n x =
  if n = 0 then 1
  else if even n then square (power (n/2) x)
  else x * (power (n-1) x);;
```

Figure 2.20: Power function (in OCaml)

We can stage the above function in the MetaOCaml code below to produce power functions specialized for a certain n—e.g., 5. The staged version of the function is identical to the original, with the mere addition of staging annotations/constructs. BER MetaOCaml has three of the four MetaML constructs mentioned earlier: meta-brackets, escape and run. In this example, although n is statically known, x remains a variable: its value will only be known at a later evaluation stage.

```
open Runcode;;
let even n = (n mod 2) = 0;;
let square x = x * x;;
let rec powerS n x =
  if n = 0 then .⟨1⟩.
  else if even n
       then .⟨square ~(powerS (n/2) x)⟩.
       else .⟨~x *  ~(powerS (n-1) x)⟩.;;

let power5 = !. .⟨fun x → ~(powerS 5 .⟨x⟩.)⟩.;;
```

Figure 2.21: Power function, staged (in MetaOCaml)

[3] More historical details about the evolution path of MetaOCaml can be found at https://web.archive.org/web/20170322141305/http://okmij.org/ftp/ML/MetaOCaml.html .

Note the structure of the above code. The return value of the `powerS` function is a staged computation, but it includes a part that can be evaluated, which is the recursive application. The result (i.e., the code/AST of the result) is spliced back into the staged computation of the final value, `power5`: a specialized function, to be available to later stages.

The operator `!.` is aliased with `Runcode.run`—the "run" functionality of MetaOCaml. This function compiles the staged lambda function, and links it back as the (specialized) code to be executed in the body of the `power5` function. Simply put, `!.` transfers our code from the world of representations of functions, `(int -> int) code`, to the world of functions, `(int -> int)`.

BER MetaOCaml lets us inspect the code that is generated from our staged algorithm, using the `print_code` function. The result looks like the following snippet. As the reader observes, the recursive applications are performed at compile-time, partially evaluating the function. The result is the residue program below:

```
fun x → x * (square (square (x * 1)))
```

Figure 2.22: Result of staging in the power function

In the example, the `square` function is referred from a future-stage computation using an identifier (`square`) bound at the present stage. This function is characterized as *cross-stage persistent*.

2.3.2.2 Lightweight Modular Staging for Scala

Rompf et al. [145] introduced MSP support in Scala with *Lightweight Modular Staging* (LMS). In LMS, the programmer has just one staging construct available, in the form of a user-level type. To indicate that, e.g., an expression does not have a current-stage integer value but a future-stage integer value, the user changes the declared type of the expression from `Int` to `Rep[Int]`. The unary abstract type constructor `Rep[_]` indicates future-stage values. Types for other values, as well as the exact version of (overloaded, current or future stage) operators are inferred.

LMS follows a library-based approach, relying on a special and extensible version of the Scala compiler called *Scala-Virtualized* [147]. In Scala-Virtualized, a Scala program is represented in terms of function calls, e.g., the control-flow construct `do b while (c)` is represented by the `__doWhile(b, c)` function call. In this way, all interesting program statements are mapped to function calls. Even method calls are represented as infix functions—e.g., `x.a(y)` as `infix_a(x,y)`. This technique permits operations to be added to the type `Rep[T]` which also supports every method of a bare `T`.

In the example below the input that needs to be designated as *future-stage* is the base `x`. This is the dynamic part of this snippet. All else is static input and will participate in compile-time evaluation. The type constructor `Rep`, in e.g., `Rep[T]`, has the property that

all operations on `T` are applicable to `Rep[T]` as well and operations on it will be generated later.

```
def even (n: Int) = n % 2 == 0
def square (x: Rep[Int]) = x * x
def powerS (n : Int, x : Rep[Int]) : Rep[Int] = {
  if (n == 0) 1
  else if (even(n)) square(powerS(n/2, x))
  else x * powerS(n−1, x)
}
def powerTest(x : Rep[Int]) : Rep[Int] = powerS(5, x)
```

Figure 2.23: Power function, staged (in Scala/LMS)

The core difference of LMS from other staging approaches is that *binding times (i.e., stages) are distinguished only by types*. The simplicity of the type-based approach to staging has been a significant boost for LMS, and owes much to the power of the Scala type system. In LMS applications, regular computations are routinely switched to staged computations with small, local changes to declared types.

In both technologies the user has many ways to generate code. For example, in Scala, CUDA code can be generated instead of Scala, and in BER MetaOCaml the user can compile directly to native code instead of the bytecode-generating, `Runcode.run` function.

We use MetaOCaml and LMS in Chapter 6 to implement a library design for streams. The disciplined programming model offered by MSP facilitated the implementation process of this library. So by using MSP we are able to develop essentially a partial evaluator for general-purpose streams, using a type-safe metaprogramming facility.

3. STREAMS FOR BULK DATA PROCESSING

Java 8 has introduced lambdas with the explicit purpose of enabling streaming abstractions. Such abstractions present an accessible, natural path to multicore parallelism—perhaps the highest-value domain in current computing. Other languages, such as Scala, C#, and F#, have supported lambda abstractions and streaming APIs, making them a central theme of their approach to parallelism. Although the specifics of each API differ, there is a core of common features and near-identical best-practices for users of these APIs in different languages.

Streaming APIs allow the high-level manipulation of value streams (with each language employing slightly different terminology) with functional-inspired operators, such as `filter`, or `map`. Such operators take user-defined functions as input, specified via local functions (lambdas). The Java example fragment below shows a "sum of even squares" computation, where the even numbers in a sequence are squared and summed. The input to `map` is a lambda, taking an argument and returning its square. This particular lambda application is *non-capturing*: the bodies of the lambda expressions below use only their argument values, and no values from the environment.

```
public int sumOfSquaresEvenSeq(int[] v) {
  int sum = IntStream.of(v)
    .filter(x → x % 2 == 0)
    .map(x → x * x)
    .sum();
  return sum;
}
```

Figure 3.1: Sum of Even Squares

The above computation can be trivially parallelized with the addition of a `.parallel()` operator before the call to `filter`. This ability showcases the simplicity benefits of streaming abstractions for parallel operations.

This chapter presents a comparative study of the lambda+streams APIs of four multi-paradigm, virtual machine-based languages, Java, Scala, C#, and F#, with an emphasis on implementation and performance comparison, across mainstream platforms (JVM on Linux and Windows, .NET CLR for Windows, Mono for Linux). We perform micro-benchmarking[1] and aim to get a high-level understanding of the costs and causes. Our goal is the usual goal of microbenchmarking: to minimize most threats-to-validity by controlling external factors. (The inherent drawback of microbenchmarking, which we do not attempt to address, is the threat that benchmarks are not representative of real uses.) In order to control external factors, we attempt to select equivalent abstractions in all set-

[1] Code in https://github.com/biboudis/clashofthelambdas .

tings, isolate dependencies, employ best-practice benchmarking techniques, and repeatedly consult experts on the different platforms.

Since lambdas+stream operators have arisen independently in so many contexts and have been central in parallel programming strategies, one would expect them to be well-understood: a mainstream, high-value feature is expected to have fairly uniform implementation techniques and trade-offs. Instead we find interesting variation, even in the compilation to intermediate code (per platform, e.g., across Java and Scala, which are both JVM languages). Furthermore, we find that JIT optimization inside the VM does not always interact predictably with the code produced for lambdas. This was a minor surprise, given the maturity of the respective facilities.[2]

A second aspect of declarative streaming operations is that they enable aggressive optimization [118]. Optimization frameworks, such as LinqOptimizer [128] and ScalaBlitz [137, 138], recognize common patterns of streaming operations and optimize them, by inlining, performing loop fusion, and more.

In all, we find that Java offers high performance for lambdas and streaming operations, primarily due to optimizing for non-capturing lambdas. At the same time, Java suffers from the lack of an optimizing framework—LinqOptimizer and ScalaBlitz give a significant boost to C#/F# and Scala implementations, respectively, when optimizations are applicable.

3.1 Implementation Techniques for Lambdas and Streaming

As part of our investigation, we examined current APIs and implementation techniques for lambdas and streaming abstractions in the languages and libraries under study. We detail such elements next, so that we can refer to them directly in our experimental results.

3.1.1 Programming Languages

We begin with the API and implementation description for the languages of our study: Java, Scala, and C#/F# (the latter are similar enough that are best discussed together, although they exhibit non-negligible performance differences).

3.1.1.1 Java

Java is probably the best reference point for our study, although it is also the relative newcomer among the lambdas+streaming facilities. We already saw examples of the Java API for streaming in the Introduction. In terms of implementation, the Java language team has chosen a translation scheme for lambdas that is highly optimized and fairly unique among statically typed languages.

[2] Although Java lambdas are standard only as of version 8, their arrival had been forthcoming since at least 2006.

In the Java 8 declarative stream processing API, operators fall into two categories: intermediate (*always lazy*—e.g., map and filter) and terminal (which can produce a value or perform side-effects—e.g., sum and reduce). For concreteness, let us consider the pipeline below. The following expression (serving as a running example in this section) calculates the sum of all values in a double array.

```
public double sumOfSquaresSeq(double[] v) {
  double sum = DoubleStream.of(v)
    .map(d → d * d)
    .sum();
  return sum;
}
```

Figure 3.2: Sum of squares, in Java 8 Streams

The code first creates a sequential, ordered `Stream` of `double`s from an array that holds all values. (`DoubleStream` represents a primitive specialization of `Stream`–one of three specialized `Streams`, together with `IntStream` and `LongStream`.) The calls `map` and `sum` are an intermediate and a terminal operation respectively. The first operation returns a `Stream` and it is lazy. It simply declares the transformation that will occur when the stream will be traversed. This transformation is a stateless operation and is declared using a (non-capturing) lambda function. The second operation needs all the stream processed up to this point, in order to produce a value; this operation is `eager` and it is effectively the same as reducing the stream with the lambda `(x,y) -> x+y`.

Implementation-wise, the (stateless or stateful) operations on a stream are represented by objects chained together sequentially. A terminal operation triggers the evaluation of the chain. In our example, *if no optimization were to take place*, the `sum` operator would retrieve data from the stream produced by `map`, with the latter being supplied the necessary lambda expression. This traversing of the elements of a stream is realized through `Spliterators`.

The `Spliterator` interface offers an API for traversing and partitioning elements of a source and it can operate either sequentially or in parallel. This interface is also equipped with more advanced functionality–e.g., it can detect whether a pipeline interferes (modifies) the data source while processing. The definition of a stream and operations on it are usually described declaratively and the user does not need to invoke operations on a `Spliterator` (although a controlled traversal is possible). The `Spliterator` interface is shown on Figure 3.3.

Normally, for the general case of standard stream processing, the implementation of the interface above will have a `forEachRemaining` method that internally calls methods `hasNext` and `next` of the `java.util.Iterator` interface to traverse a collection, as well as `accept` to apply an operation to the current element. The `Consumer<T>` argument is a functional interface (with a single method named `accept`) that takes as a parameter an

```
public interface Spliterator<T> {
  boolean tryAdvance(Consumer<? super T> action);
  void forEachRemaining(Consumer<? super T> action);
  Spliterator<T> trySplit();
  long estimateSize();
  long getExactSizeIfKnown();
  int characteristics();
  boolean hasCharacteristics(int characteristics);
  Comparator<? super T> getComparator();
}
```

Figure 3.3: The `Spliterator` interface of Java 8 Streams

item of type `T` and returns void. Thus, three virtual calls per element will occur.

```
while (hasNext()){ action.accept(next()); }
```

Figure 3.4: The `forEachRemaining` implementation for the general case

However, stream pipelines, such as the one in our example, can be optimized. For the array-based `Spliterator`, the `forEachRemaining` method performs an indexed-based, do-while loop. The entire traversal is then transformed: instead of `sum` requesting the next element from `map`, the pipeline operates in the inverse order: `map` pushes elements through the `accept` method of its downstream `Consumer` object, which implements the `sum` functionality. In this way, the implementation eliminates two virtual calls per step of iteration and effectively uses internal iteration, instead of external. We call this a "push-based" design.

The following snippet of code is a simplified excerpt from the `Spliterators.java` source file of the Java 8 library and demonstrates this special handling, where `a` holds the source array and `i` indexes over its length:

```
do { consumer.accept(a[i]); } while (++i < hi);
```

Figure 3.5: Specialized implementation for array traversal

The internal iteration can be seen in this code. Each of the operators applicable to a stream needs to support this inverted pattern by supplying an `accept` operation. That operation, in turn, will call `accept` on whichever `Consumer<T>` may be downstream. For instance, the fragment of the `map` implementation below shows the `accept` call on the passed consumer (essentially the next operator) which is the `sum` in our example. The code also shows the call to `apply`, invoking the passed lambda.

```
<T, R> Stream<R> map(Stream<T> source, Function<T, R> mapper) {
  return new MapperStream<T, R>(source) {
    Consumer<T> wrap(Consumer<R> consumer) {
      return new Consumer<T>() {
        void accept(T v) { consumer.accept(mapper.apply(v)); }
      };
    }
  };
}
```

Figure 3.6: Part of the implementation of the `map` operator

Having seen the implementation of streams, we now turn our attention to lambdas. There could be several potential translations for lambdas, such as inner-classes (for both capturing or non-capturing, lambdas), translation based on `MethodHandle`s—the dynamic and strongly typed component that was introduced in JSR-292—and more. Each option has some advantages and disadvantages. For the translation of lambdas in Java 8, the compiler incorporates a technique based on JSR-292 [152] and more specifically on the new `invokedynamic` command [100, Chapter 6] and `MethodHandle`s [59].

When the compiler encounters a lambda function, it desugars it to a method declaration and emits an `invokedynamic` instruction at that point. For instance, our `sumOfSquaresSeq` example compiles to the bytecode in Figure 3.7.

```
...                          // v on the stack
invokestatic  #7             // DoubleStream.of
invokedynamic #10,  0        // applyAsDouble
invokeinterface #11,  2      // DoubleStream.map
invokeinterface #8,  1       // DoubleStream.sum
dstore_1
dload_1
dreturn
```

Figure 3.7: Bytecode of sum of squares

Note the `invokedynamic` instruction on line 3, used to return an object that represents a lambda closure. The method invoked is `LambdaMetafactory.metafactory` and is implemented as part of the Java standard library. The fully dynamic nature of the call is due to having a single implementation for retrieving objects for any given method signature. This process involves three phases: *Linkage*, *Capture* and *Invocation*. When `invokedynamic` is met for the first time it must link this site with a method. For the lambda translation case, an instance of `CallSite` is generated whose target knows how to create function objects. This target (`LambdaMetafactory.metafactory`) is a factory for function objects.

The Capture phase may involve allocation of a new object that may capture parameters or will always return the same object (if no parameters are captured). The third phase is the actual invocation. The advantage of this translation scheme is that, for lambdas that do not capture any free variables, a single instance for all usages is enough. Furthermore, the call site is linked only once for successive invocations of the lambda and, after that, the JVM inlines the retrieved method's invocation at the dynamic call site. Additionally, there is no performance burden for loading a class from disk, as there would be in the case of a fully static translation.

The `invokedynamic` instruction is parameterized by the method's name and descriptor. In this example, the information is found in entry #10 of the constant pool:

```
#10 = InvokeDynamic #0:#172
```

Figure 3.8: Entry in constant pool

What Java gains with this translation scheme is that the linking of the call site has been deferred at runtime and the (JIT) compiler uses a recipe to find the target type. This command, that was introduced to improve the support of dynamic languages on the JVM, is used as part of the translation strategy of a statically typed language.

The name for this *dynamic call site* is called a *lambda factory* for that particular lambda. In short, `invokedynamic` will return an instance of the functional interface `DoubleUnaryOperator` which contains the specialized method `applyAsDouble`. In this example, the 10th runtime constant pool entry refers to a bootstrap method specifier (that is stored in a different attribute in the class file named `BootstrapMethods`) which in this case is stored in the first method of that attribute array.

The metafactory's signature is shown at the bootstrap methods' attribute. The parameters in the same order are the following:

- `caller`: the method handle factory with specified access checking. This object is responsible for looking up method at the caller.
- `invokedName`: the name of the method.
- `invokedType`: the `CallSite` signature.
- `samMethodType`: the implemented method type.
- `implMethod`: handle to the actual implementation. The type signature of the actual method may differ from the functional interface and many adaptations may employed via the supplied combinators [148].
- `instantiatedMethodType`: it is the same with `samMethodType` or a specialization of it. This is what will be enforced dynamically.

The first three, are stacked automatically by the VM, the rest are pushed explicitly. The reader can notice that both `samMethodType` and `instantiatedMethodType` describe the same method type in our case, a method (lambda) that has one formal parameter and a return type, both of type `Double`. The method doesn't refer to `this`, `super`, or other members of the enclosing instance so it is translated as a static method, thus the `MethodHandle` refers to the static method `lambda$sumOfSquaresSeq$3`.

```
0: #169 invokestatic java/lang/invoke/LambdaMetafactory.metafactory:
 (Ljava/lang/invoke/MethodHandles$Lookup;
  Ljava/lang/String;
  Ljava/lang/invoke/MethodType;
  Ljava/lang/invoke/MethodType;
  Ljava/lang/invoke/MethodHandle;s
  Ljava/lang/invoke/MethodType;)
     Ljava/lang/invoke/CallSite;
Method arguments:
 #170 (D)D
 #171 invokestatic ClashOfLambdas.lambda$sumOfSquaresSeq$3:(D)D
 #170 (D)D
```

Figure 3.9: Signature of `LambdaMetafactory.metafactory`

3.1.1.2 Scala

Scala is an object-functional programming language for the JVM. Scala has a rich object system offering traits and mixin composition. As a functional language, it has support for higher-order functions, pattern matching, algebraic data types, and more. Since version 2.8, Scala comes with a rich collections library offering a wide range of collection types, together with common functional operators, such as `map`, `filter`, `flatMap`, etc. There are two Scala alternatives for our purposes. One is lazy transformations of collections: an approach semantically equivalent to that of other languages, which also avoids the creation of intermediate, allocated results. The other alternative is to use strict collections, which are better supported in the Scala libraries, yet not equivalent to other implementations in our set and suffering from increased memory consumption.

To achieve lazy processing, one has to use the `view` method on a collection.[3] This method wraps a collection into a `SeqView`. The following example illustrates the use of `view` for performing such transformations lazily:

[3] Scala has more APIs for lazy collections (e.g., "Streams"), but the views API we employed is the exact counterpart, in spirit and functionality, to the machinery in the other languages under study.

```
def sumOfSquareSeq (v : Array[Double]) : Double = {
  val sum : Double = v.view
    .map(d => d * d)
    .sum
  sum
}
```

Figure 3.10: Sum of squares, in Scala Views

Ultimately, `SeqView` extends `Iterable[A]`, which acts as a factory for iterators. As an example, we can demonstrate the common `map` function by mapping the transformation function to the source's `Iterable` iterator:

```
def map[T, U](source: Iterable[T], f: T => U) = new Iterable[U] {
  def iterator = source.iterator map f
}
```

Figure 3.11: Implementation of `map`, in Scala Views

The `Iterator`'s `map` function can then be implemented by delegation to the source iterator:

```
def map[T, U](source: Iterator[T], f: T => U): Iterator[U] = new Iterator[U] {
  def hasNext = source.hasNext
  def next() = f(source.next())
}
```

Figure 3.12: Implementation of `map` function on the `Iterator` type, in Scala Views

Note that we have 3 virtual calls (`next`, `hasNext`, `f`) per element pointed by the iterator. The iteration takes place in the expected, unoptimized order, i.e., each operator has to "request" elements from the one supplying its input, rather than having a "push" pattern, with the producer calling the consumer directly. We call this a "pull-based" design.

The Scala translation is based on synthetic classes that are generated at compile time. For lambdas, Scala generates a class that extends `scala.runtime.AbstractFunction`. For lambdas with free variables (captured from the environment), the generated class includes private member fields that get initialized at instantiation time.

The strict processing of Scala collections is similar to the above lazy idioms from the end-user standpoint: only the `view` call is omitted in our `sumOfSquareSeq` code example. Operators such as `map` are overloaded to also process strict collections.

3.1.1.3 C#/F#

C# is a modern object-oriented programming language targeting the .NET framework. An important milestone for the language was the introduction of several new major features in C# 3.0 in order to enable a more functional style of programming. These new features, under the umbrella of LINQ [112, 114], can be summarized as support for lambda expressions and function closures, extension methods, anonymous types and special syntax for query comprehensions. All of these new language features enable the creation of new functional-style APIs for the manipulation of collections.

F# is a modern .NET functional-first programming language based on OCaml, with support for object-oriented programming, based on the .NET object system.

In C# we have two ways of programming with data streams: 1) as fluent-style method calls (Figure 3.13) or 2) with the equivalent query comprehension syntactic sugar (Figure 3.14).

```
nums.Where(x => x % 2 == 0).Select(x => x * x).Sum();
```

Figure 3.13: Sum of Squares, in C# LINQ (fluent API)

```
(from x in nums
 where x % 2 == 0
 select x * x).Sum();
```

Figure 3.14: Sum of Squares, in C# LINQ (syntactic sugar)

In F#, programming with data is just as simple as a direct pipeline of various operators (note that the definition of the pipe operator `(|>) a f = f a`).

```
nums  ▷ Seq.filter (fun x → x % 2 = 0)
      ▷ Seq.map (fun x → x * x)
      ▷ Seq.sum
```

Figure 3.15: Sum of Squares, in F#

For the purposes of this discussion, we can consider that both C# and F# have identical operational behaviors and both C# methods (`Select`, `Where`, etc.) and F# operators (`Seq.map`, `Seq.filter`, etc.) operate on `IEnumerable<T>` objects and return `IEnumerable<T>`.

The `IEnumerable<T>` interface can be thought of as a factory for creating `IEnumerator<T>` objects:

```
interface IEnumerable<T> {
  IEnumerator<T> GetEnumerator();
}
```

Figure 3.16: `IEnumerable` **interface**

and `IEnumerator<T>` is an iterator for an on demand consumption of values:

```
interface IEnumerator<T> {
  // Return current position element
  T Current {
    get;
  }

  // Move to next element and return false if no more elements remain
  bool MoveNext();
}
```

Figure 3.17: `IEnumerator` **interface**

Each of these methods/operators implement a pair of interfaces called `IEnumerable<T>` / `IEnumerator<T>` and through the composition of these methods a call graph of iterators is chained together. The lazy nature of the iterators allows the composition of an arbitrary number of operators without worrying about intermediate materialization of collections between each call. Instead, each operator call is interleaved with each other. As an example we can present an implementation of the `Select` method.

```
static IEnumerable<R> Select<T, R>(IEnumerable<T> source, Func<T, R> f) {
  return new SelectEnumerable<T, R>(source, f);
}
```

Figure 3.18: Implementation of `Select` **operator**

The `SelectEnumerable` has a simple factory-style implementation:

```
class SelectEnumerable<T, R> : IEnumerable<R> {
  private readonly IEnumerable<T> inner;
  private readonly Func<T, R> func;

  public SelectEnumerable(IEnumerable<T> inner,
                          Func<T, R> func) {
    this.inner = inner;
    this.func = func;
  }

  IEnumerator<R> GetEnumerator() {
    return new SelectEnumerator(inner.GetEnumerator(), func);
  }
}
```

Figure 3.19: Implementation of `SelectEnumerable` factory

`SelectEnumerator` implements the `IEnumerator<R>` interface and it holds a reference to the inner iterator. It delegates to this iterator the `MoveNext` and `Current` calls.

```
class SelectEnumerator<T, R> : IEnumerator<R> {
  private readonly IEnumerator<T> inner;
  private readonly Func<T, R> func;

  public SelectEnumerator(IEnumerator<T> inner,
                          Func<T, R> func) {
    this.inner = inner;
    this.func = func;
  }

  bool MoveNext() {
    return inner.MoveNext();
  }

  R Current { get {
    return func(inner.Current);
  }}
}
```

Figure 3.20: Implementation of `SelectEnumerator` type

For programmer convenience, both C# and F# offer support for automatically creating

the elaborate scaffolding of the `IEnumerable<T>` / `IEnumerator<T>` interfaces, but for our discussion it is not crucial to understand the mechanisms.

From a performance point of view, it is not difficult to see that there is a lot of virtual call indirection between the chained enumerators. We have 3 virtual calls (`MoveNext`, `Current`, `func`) per element per iterator. Iteration is similar to Scala or to the generic, unoptimized Java iteration: it is an external iteration, with each consumer asking the producer for the next element.

In terms of lambda translation, C# lambdas are always assigned to delegates, which can be thought of as type-safe function pointers, and in F# lambdas are represented as compiler generated class types that inherit `FSharpFunc`.

```
abstract class FSharpFunc<T, R> {
  abstract R Invoke(T arg);
}
```

Figure 3.21: Lambdas representation in F#

In both cases, if a lambda captures free variables, these variables are represented as member fields in a compiler-generated class type.

3.1.2 Optimizing Frameworks

We next examine two optimizing frameworks for streaming operations: ScalaBlitz and LinqOptimizer.

3.1.2.1 ScalaBlitz

ScalaBlitz is an open source framework that optimizes Scala collections by applying optimizations for both sequential and parallel computations. It eliminates boxing, performs lambda inlining, loopfusion and specializations to particular data-structures. ScalaBlitz performs optimizations at compile-time based on Scala macros [25]. By enclosing a functional pipeline into an `optimize` block, ScalaBlitz expands in place an optimized version of it (Figure 3.22). This can be achieved because this library is implemented as a def macro, with signature shown in Figure 3.23.

The `optimize` block is a function that starts with the `macro` keyword. When the compiler encounters an application of the macro `optimize(expression)`, it will expand that application by invoking `optimize_impl`, with the abstract-syntax tree of the functional pipeline expression as argument. The result of the macro implementation is an expanded abstract syntax tree. This tree will be replaced at the call site and will be type-checked.

```
def sumOfSquareSeqBlitz (v : Array[Double]) : Double = {
  optimize {
    val sum : Double = v.map(d => d * d).sum
    sum
  }
}
```

Figure 3.22: Sum of squares, in Scala Blitz

```
def optimize[T](exp: T): Any = macro optimize_impl[T]
```

Figure 3.23: Def macro implementation

3.1.2.2 LinqOptimizer

LinqOptimizer is an open source optimizer for LINQ queries which compiles declarative queries into fast loop-based imperative code, eliminating virtual calls and temporary heap allocations. It is a run-time compiler based on LINQ Expression trees, which enable a form of metaprogramming, based on type-directed quotations. In the following example a lambda expression is assigned to a variable with type `Expression<Func<…>>`.

```
Expression<Func<int, int>> exprf = x => x + 1;

// compile to IL
Func<int, int> f = exprf.Compile();

// 2
Console.WriteLine(f(1));
```

Figure 3.24: Run-time compiler of LINQ Expression trees

At compile time, the compiler emits code to build an expression tree that represents the lambda expression. LINQ offers library support for runtime manipulation of expression trees (through visitors) and also support for run-time compilation to IL. Using such features, LinqOptimizer lifts queries into the world of expression trees and performs the following optimizations: 1) inlines lambdas and performs loop fusion: (Figure 3.25) and 2) for queries with nested structure (`SelectMany`, `flatMap`) applies nested loop generation (Figure 3.26).

```
var sum = (from num in nums.AsQueryExpr() // lift
           where num % 2 == 0
           select num * num).Sum();
// effectively optimizes to
int sum = 0;
for (int index = 0; index < nums.Length; index++) {
   int num = nums[index];
   if (num % 2 == 0)
      sum += num * num;
}
```

Figure 3.25: Lambda inlining and loop fusion

```
var sum = (from num in nums.AsQueryExpr() // lift
           from _num in _nums
           where num % 2 == 0
           select num * _num).Sum();
// effectively optimizes to
int sum = 0;
for (int index = 0; index < nums.Length; index++) {
   for (int _index = 0; _index < _nums.Length; _index++) {
      int num = nums[index];
      int _num = _nums[_index];
      if (num % 2 == 0)
        sum += num * _num;
}
```

Figure 3.26: Nested loop generation

3.2 Results

We next discuss our benchmarks and experimental results.

3.2.1 Microbenchmarks

We use four main microbenchmarks which come from the sets used by Murray et al. [118]. We focus our efforts on measuring iteration throughput and lambda invocation costs. In all of our benchmarks we produce scalar values as the result of a terminal operation (e.g., instead of producing a transformed list of values), as we do not want to cause memory management effects (e.g., garbage collection). Furthermore, we did not employ sorting or grouping operators, in order to avoid interfering with algorithmic details of library imple-

mentations (e.g., mergesort vs quicksort, hash tables vs balanced trees, etc.).

We measure the performance of:

- **sum** iteration speed with no lambdas, just a single iteration.
- **sumOfSquares** a small pipeline with one map operation (i.e., one lambda).
- **sumOfSquaresEven** a bigger pipeline with a filter and map chain of two lambdas.
- **cart** a nested pipeline with a `flatMap` and an inner operation, again with a `flatMap` (capturing a variable), to encode a Cartesian product.

We developed this set for all four languages, Java, Scala, C# and F#, for both sequential and parallel execution. For the latter three we have also included optimized versions using ScalaBlitz and LinqOptimizer. For Scala we also include alternate implementations, which employ more idiomatic strict collections (without the views API). Arguably this approach is better supported in the Scala libraries. Therefore we present separate measurements for Scala-views and Scala-strict tests. In our following analysis, when we do not refer to a Scala-strict test explicitly, Scala-views are implied. Additionally, we include a baseline suite of benchmarks for the sequential cases.

We have run these benchmarks on both Windows and Linux, although Windows is the more universal reference platform for our comparison: it allows us to perform the C#/F# tests on the industrial-strength implementation of the Microsoft CLR virtual machine.

The purpose of baseline benchmarks is to assess the performance difference between functional pipelines and the corresponding imperative, hand-written equivalents. The imperative examples make use of indexed-based loop iterations in the form of `for`-loops (except for the Scala case in which the while-loop is the analogue of imperative iteration).

Input: All tests were run with the same input set. For the **sum**, **sumOfSquares** and **sumOfSquaresEven** we used an array of $N = 10,000,000$ long integers, produced by N integers with a `range` function. The **cart** test iterates over two arrays. An outer one of $1,000,000$ long integers and an inner one of 10.

The Scala, C# and F# tests were compiled with optimization flags enabled and for Java/Scala tiered compilation was left disabled (C2 JIT compiler only). Additionally, we fixed the heap size to 3GB for the JVM to avoid heap resizing effects during execution.

3.2.2 Experimental Setup

Systems: We performed both Windows (Figure 3.29) and Linux (Figure 3.30) tests natively on the same system.

	Windows	Ubuntu Linux
Version	8.1	13.10/3.11.0-24
Architecture	x64	x64
CPU	Intel Core i5-3360M vPro 2.8GHz	
Cores	2 physical x 2 logical	
Memory	4GB	

Figure 3.27: Experimental Setup (Hardware)

	Windows	Ubuntu Linux
Java	Java 8 (b132)/JVM 1.8	
Scala	2.10.4/JVM 1.8	
C#	C#5 /CLR v4.0	C# mono 3.4.0.0/mono 3.4.0
F#	F#3.1/CLR v4.0	F# open-source 3.0/mono 3.4.0

Figure 3.28: Experimental Setup (Software)

Microbenchmarking automation: For Java and Scala benchmarks we used the Java Microbenchmark Harness (JMH) [157] benchmarking tool. JMH is an annotation-based tool and takes care of all intrinsic details of the execution process. Its goal is to produce as objective results as possible. The JVM performs JIT compilation (we use the C2 JIT compiler) so the benchmark author must measure execution time after a certain warm-up period to wait for transient responses to settle down. JMH offers an easy API to achieve that. In our benchmarks we employed 10 warm-up iterations and 10 proper iterations. We also force garbage collection before benchmark execution. Regarding the CLR, warm-up effects take an infinitesimal amount of time compared to the JVM [158]. The CLR JIT compiler compiles methods exactly once and subsequent method calls invoke directly the JITted version. Code is never recompiled (nor interpreted at any point). For the purpose of benchmarking C#/F# programs, as there is not any widely-used, state-of-the-art tool for microbenchmarking, we created the *LambdaMicrobenchmarking* utility[4] written in C#, according to the common microbenchmarking practices described in [154]. It calculates the average execution of method invocations using the `TimeSpan.TotalMilliseconds` property of the `TimeSpan` structure that converts ticks to whole and fractional milliseconds. Our utility uses the Student-T distribution for statistical inference; mean error and standard deviation. The same distribution is employed in JMH as well. Our utility forces garbage collection between runs. For all tests, we do not measure the time needed to initialize data-structures (filling arrays), and neither the run-time compilation cost of the optimized queries in the LinqOptimizer case nor the compile-time overhead of macro expansion in the ScalaBlitz case.

[4] https://github.com/biboudis/LambdaMicrobenchmarking .

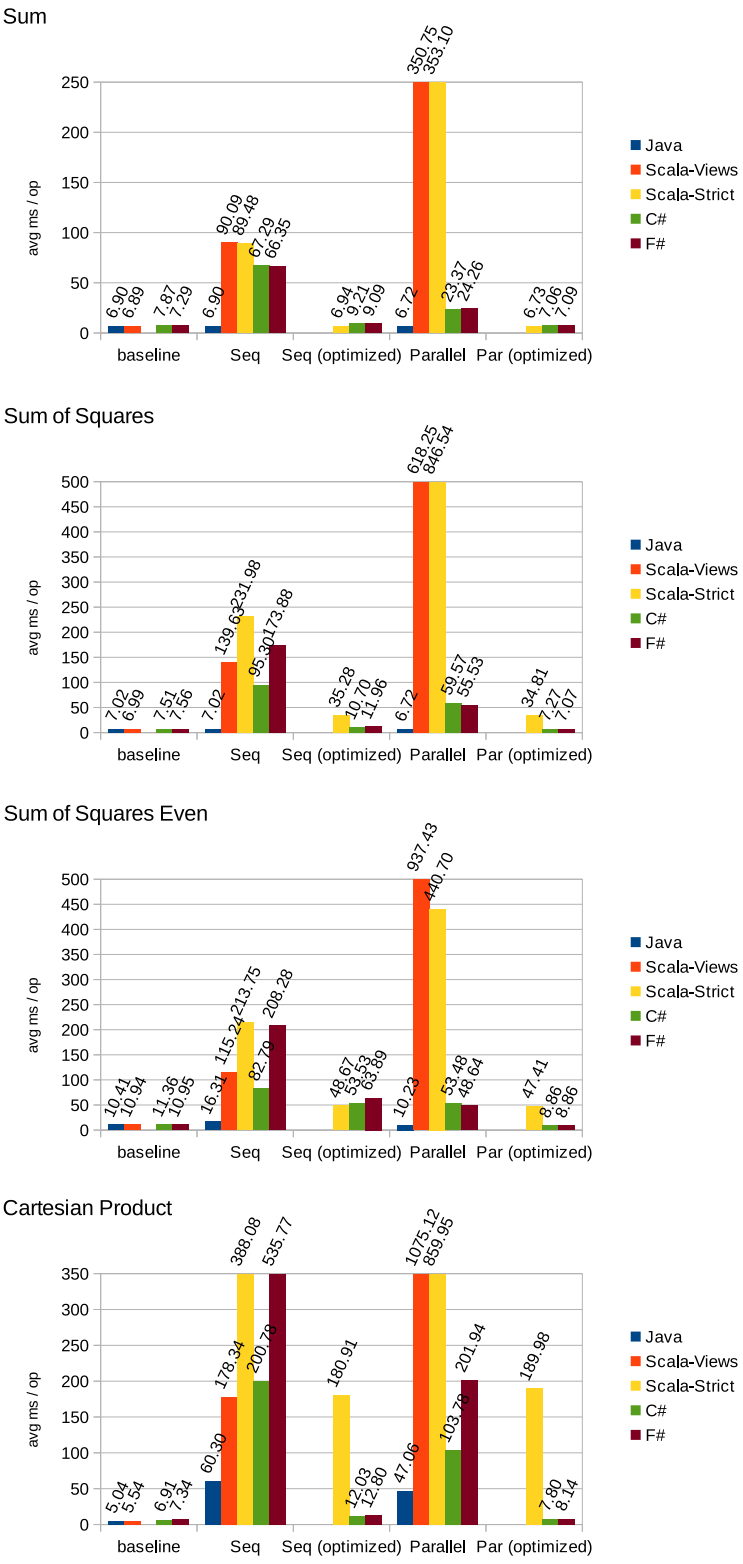


Figure 3.29: Microbenchmark Results on Windows (CLR/JVM) in milliseconds / operation (average of 10). Y-axis truncated for readability.

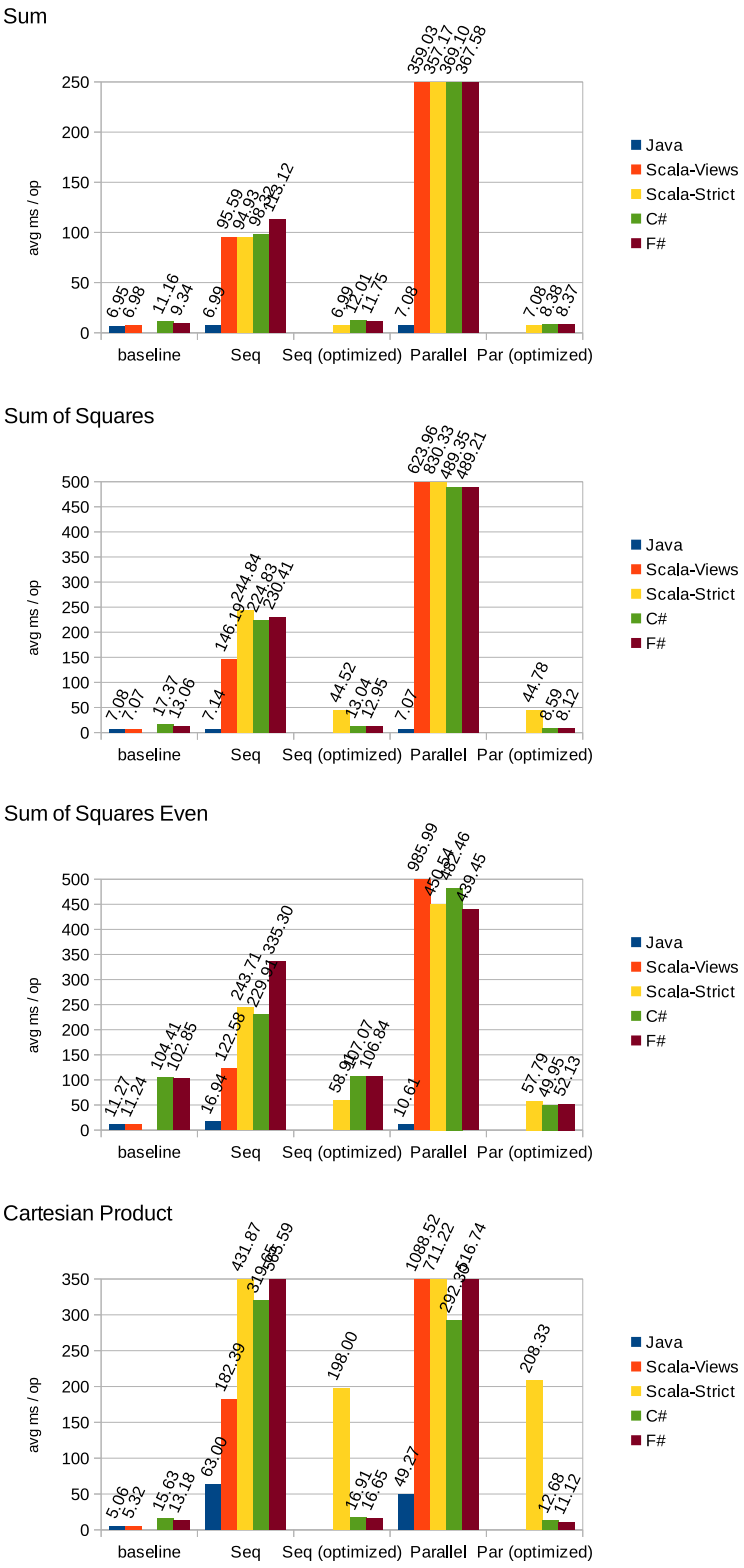


Figure 3.30: Microbenchmark Results on Linux (mono/JVM) in milliseconds / operation (average of 10). Y-axis truncated for readability.

	Windows					Linux				
Benchmark	Java	Scala-Views	Scala-Strict	C#	F#	Java	Scala-Views	Scala-Strict	C#	F#
sumBaseline	0.011	0.015		1.214	0.168	0.054	0.011		0.552	0.818
sumSeq	0.015	0.607	0.277	2.407	0.525	0.014	0.449	0.475	0.359	1.015
sumSeqOpt		0.010		0.536	0.212		0.022		0.248	0.730
sumPar	0.035	2.348	2.622	0.895	4.371	0.009	3.653	1.827	106.800	117.358
sumParOpt		0.017		0.075	0.196		0.026		1.400	2.010
sumOfSquaresBaseline	0.008	0.016		0.129	0.202	0.023	0.013		0.799	1.072
sumOfSquaresSeq	0.009	1.049	2.052	0.763	3.755	0.019	1.331	0.895	1.193	1.116
sumOfSquaresSeqOpt		1.104		0.215	0.292		0.238		0.583	0.171
sumOfSquaresPar	0.008	3.691	9.355	2.745	0.162	0.017	2.807	6.347	23.856	40.342
sumOfSquaresParOpt		0.036		0.433	0.094		0.136		0.782	0.485
sumOfSquaresEvenBaseline	0.044	0.085		0.204	0.393	0.059	0.035		0.906	1.270
sumOfSquaresEvenSeq	0.121	1.157	1.510	3.789	4.838	0.096	1.159	1.042	0.895	1.680
sumOfSquaresEvenSeqOpt		0.550		2.052	5.351		0.162		0.847	0.522
sumOfSquaresEvenPar	0.025	5.184	8.207	5.943	2.556	0.027	4.905	16.252	46.739	21.465
sumOfSquaresEvenParOpt		0.502		0.115	0.128		0.483		1.737	4.390
cartBaseline	0.060	0.041		0.015	1.007	0.010	0.010		0.040	0.113
cartSeq	0.749	6.195	3.939	4.284	5.840	0.510	2.437	5.486	0.954	2.791
cartSeqOpt		0.666		0.148	0.232		0.763		0.751	0.307
cartPar	0.131	13.167	13.165	4.954	7.855	0.243	7.641	7.484	10.963	7.546
cartParOpt		2.694		0.904	1.371		2.642		1.810	1.310
refBaseline	0.069	0.259		0.159	0.360	0.152	0.288		1.740	1.566
refSeq	0.221	1.077	0.719	1.267	3.415	0.237	0.438	0.353	1.269	0.639
refSeqOpt		0.284		2.082	1.437		0.235		2.409	1.643
refPar	0.119	5.123	0.853	8.548	2.556	0.271	6.904	0.765	44.879	27.644
refParOpt		0.247		0.782	0.187		0.112		1.445	2.592

Figure 3.31: Standard deviations for 10 runs of each benchmark.

3.2.3 Performance Evaluation

Languages: Among the languages[5] of our study, Java exhibits by far the best performance, in both sequential and parallel tests, due to its advanced translation scheme. Notably, Java results show not only that three out of four of our tests are very close to baseline measurements but also that the parallel versions scale well. Regarding the parallel versions, all microbenchmarks reveal that even in cases where Java was very close to the baseline, performance increases further achieving parallel speedups of 1.1x-1.6x. For the **cart** benchmark, although Java has the best performance among all streaming implementations, it still pays a considerable cost for inner closures, as can be seen in comparison to the baseline benchmark for the sequential case. Scala Parallel Collections using the lazy, `view`, API seem to suffer in the parallel tests quite significantly over all other implementations (note that the Y-axis is truncated) due to boxing/unboxing, iterator, and function object abstraction penalties. (For a more detailed analysis, see Section 3.3.) The strict Scala API (which, although non-equivalent to other implementations is arguably more idiomatic) performed significantly better. Although we present results for a 3GB heap space, we have also conducted the same tests under various constrained heap spaces. In practice, Scala-strict benchmarks ran with about 4x more heap space than their Java counterparts, which is unsurprising given that all strict operators need to generate and process intermediate collections. Still, the parallel Scala/Scala-strict benchmarks were almost always the slowest among all implementations on both Windows and Linux.

In the sequential tests of C# and F# we observe a constant difference in favor of C# for

[5]Although it is easy to categorize benchmarks per language, and we refer to languages throughout, it is important to keep in mind that the comparison concerns primarily the standard libraries of these languages and only secondarily the language translation techniques for lambdas.

sumOfSquares, **sumOfSquaresEven** and a significant difference of 2.7x for the **cart** benchmark on Windows. As `seq<'T>` is a type alias for .NET's `IEnumerable<T>` we conclude that the difference is attributed to different implementations of operators. In the parallel benchmarks, as F# relies on the standard library for .NET, it is driven by its performance. Thus, all parallel benchmarks (Windows and Linux) show these two languages at the same level.

In all cases, the parallel benchmarks of LINQ on mono scaled poorly, revealing poor scaling decisions in the implementation. Additionally, comparing the Windows and Linux charts for the respective baseline benchmarks, mono seems to have generated slower code for the **sumOfSquaresEven** benchmark, in which the modulo operation is applied. This indicates that JIT compilation optimizations can be improved significantly, especially in cases such as the handwritten fused loop-if operation of the **sumOfSquaresEven** situation.

Among all standard parallel libraries, F# achieved the best scaling of 2.6x-4.3x.

Optimizing frameworks: When streaming pipelines are amenable to optimization, the improvement can be dramatic.

ScalaBlitz improved Scala in virtually all cases. Especially in the **sum** benchmark, Scala was significantly improved, achieving an execution time close to that of the Java/Scala baseline tests. Notable are the 52x speed-up in relation to Scala Parallel Collections for the **sum** benchmark on Windows, as well as 50x on Linux. Additionally, ScalaBlitz achieved a 17x improvement for **sumOfSquares** and 19x for **sumOfSquaresEven** (again for the parallel benchmarks) on Windows. ScalaBlitz did not demonstrate improved performance in the case of nested loops (sequential **cart**) but presented a 5.7x speedup in the parallel version on Windows (and 5.2x on Linux). Apart from the elimination of abstraction penalties, ScalaBlitz offers additional performance improvement in the parallel optimized versions due to its iterators that allow fine-grained and efficient work-stealing [138].

LinqOptimizer improved in all cases the performance of the C# and F# benchmarks. The result of LinqOptimizer universally demonstrates the smallest performance gap with the baseline benchmarks, in absolute values. Especially in the **cart** benchmark, LinqOptimizer achieved a speed-up of 17x(sequential) and 13x(parallel) for C# and 42x and 25x respectively for F#. Among the two .NET languages, F# is the one that benefits more by LinqOptimizer in the sequential **sumOfSquares** and **sumOfSquaresEven** benchmarks. F# gets 14x and 3x improvements for these benchmarks, respectively, while C# gets 9x and 1.5x for the sequential tests. In the case of **cart**, LinqOptimizer has employed the nested loop optimization, which brings execution near the baseline level.

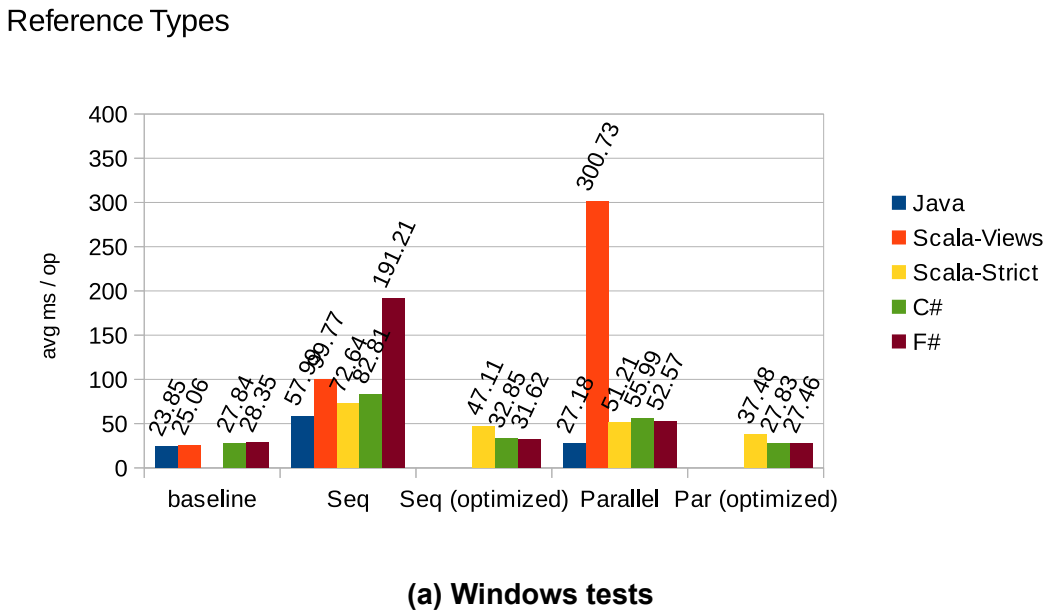
In table 3.31 we present the standard deviation of all microbenchmarks. Among all measurements, the parallel collections of Scala and C#/ F# on mono/Linux presented the highest deviations. Java demonstrates the highest stability. The strict version of Scala for the parallel **sumOfSquares** benchmark exhibit a relatively higher standard deviation, possibly because of memory effects.

3.3 Discussion

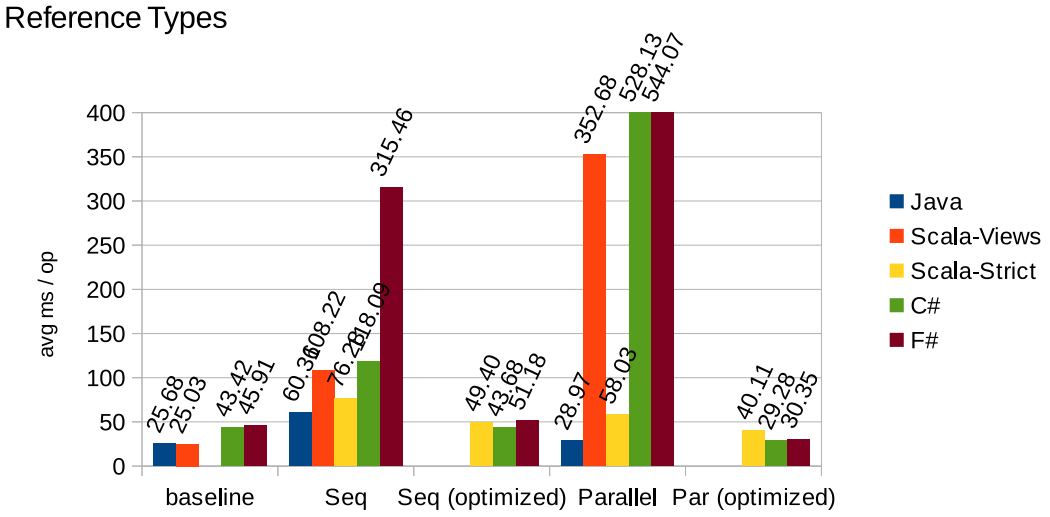
Our microbenchmarks paint a fairly clear picture of the current status of lambdas combined with streaming implementations, as well as their future improvement prospects. Java employs the most aggressive implementation technique that does not perform invasive optimization. Other languages could benefit from the same translation approach. At the same time, Java does not have an optimization framework along the lines of ScalaBlitz or LinqOptimizer. The **cart** microbenchmark showcases the need for such optimizations: C#/F# are 7x faster in parallel performance than Java. For more realistic programs, such benefits may arise more often. Hence, identifying cases in which Java can benefit from a Stream API optimizing framework (as in the closed-over variables of **cart**) is a promising direction. Scala is an outlier in most of our measurements. We found that its performance, in both the strict and the non-strict case, is subject to memory management effects. We first examined whether such effects can be alleviated with the use of VM flags, without intrusive changes to the benchmarks' source code. Our microbenchmark runs employ the default JVM setup of a parallel garbage collector (GC) with *GC ergonomics* enabled by default. GC ergonomics is an adaptive mechanism that tries to meet (in order) three goals: 1) minimize pause time, 2) maximize throughput, 3) minimize footprint. Leaving GC ergonomics enabled is not always beneficial for Scala. We conducted the same tests without the use of adaptive sizing (-XX:-UseAdaptiveSizePolicy) and no explicit sizing of generations (on Linux). For both strict and non-strict (not optimized) parallel tests, we observed an improvement of 1.1x-2.9x, with the parallel version of **sumOfSquaresPar** exhibiting the maximum increase. However, removing adaptive sizing of the heap also causes a performance degradation of about 10%-15% in the majority of sequential tests. In limited exploration (also based on suggestions by Scala experts) we found no other flag setup that significantly affects performance.

The main problem with Scala performance is that the Scala Collections are not specialized for primitive types. Therefore, Scala suffers significant boxing and unboxing overheads for primitive values, as well as memory pressure due to the creation of intermediate (boxed) objects. Prokopec et al. [138] explain such issues, along with the effects of indirections and iterator performance. Method-level specialization for primitive types can currently be effected in two ways. One is the Scala `@specialized` annotation, which specializes chains of annotated generic call sites [41], while the other is *Miniboxing* [175]. Use of the `@specialized` annotation causes the injection of specialized method calls while preserving compatibility with generic code. The use of `@specialized` preserves separate compilation by generating all variants of specialized methods, hence leading to bytecode explosion. Partly due to such considerations, Scala Collections do not employ the `@specialized` annotation. Miniboxing presents a promising alternative that minimizes bytecode size and defers transformations to load time. Currently Miniboxing is offered as a Scala compiler plugin. Having specialized collections in the Scala standard library could greatly improve performance in our benchmarks.

To demonstrate the above points, in Figure 3.32 we present an additional benchmark (**refs**), which executes a pipeline with reference types and avoids automatic boxing of our



(a) Windows tests



(b) Linux tests

Figure 3.32: Microbenchmark with manual boxing. Y-axis in milliseconds / operation (average of 10), truncated for readability.

input data. The benchmark operates on an array of $10,000,000$ instances of a class, `Ref`, employs two filter operators, and finally returns the size of the resulting collection. This benchmark effectively performs boxing manually, for all languages. In this benchmark, Java outperforms other streaming libraries but the difference is quite small. Scala is now directly comparable to all other implementations, since it performs no extraneous boxing compared to other languages. Both sequential and parallel tests for Java didn't invoke the GC. However, Scala in the `Filtered` trait, which is defined in the `GenSeqViewLike` implementation trait, causes internal boxing for the size operator. The `length` definition in `Filtered`, which delegates to the lazy value of `index`, and the array allocation inside that lazy value are responsible for this effect. In the Scala-strict parallel test, nearly 100% of the allocated memory (originating both from the main thread and from the Fork/Join workers) comes from the intermediate arrays, but the ample heap space combined with the almost perfect inlining of the main internal transformer (`filter2combiner_quick` of `ParArrayIterator`) makes the Scala version highly competitive.

Figure 3.32 exhibits a desirable property: if we consider the implementations that remove the incidental overheads that we identified (and which otherwise dominate computation costs), all language versions exhibit parallel scaling. Observe the parallel speedups in the case of Java, Scala-strict, F#, and C# on Windows.

One final remark is on the choice of using the C2 JIT compiler only (by using the -XX:-TieredCompilation flag). In both Scala and Java tests, using tiered compilation degraded the performance in the majority of our benchmarks. Concretely, for the Scala tests, tiered compilation had only a minor positive effect on the **sum** tests and an approximately 10% performance degradation in all other cases. Regarding the Java cases, all tests, apart from the sequential and parallel versions of the **refs** benchmark, presented performance degradation.

3.4 Summary of Mainstream Stream APIs

In this chapter, we evaluated the combined cost of lambdas and stream APIs in four different multiparadigm languages running on two different runtime platforms. We used benchmarks expressed with the closest comparable datatypes that each language offers in order to preserve semantic equivalence. Our benchmarks constitute a fine-grained set. Each benchmark builds upon the previous one in terms of complexity. Additionally we run all benchmarks on both Windows and Linux. Our results clearly show the benefit of advanced implementation techniques in Java, but also the performance advantage of optimizing frameworks that can radically transform streaming pipelines.

One key observation about the libraries we study is that streaming operators introduce a separate (domain-specific) sub-language that is interpreted during program run-time. This observation is inspired by the architecture of the Java 8 streams library, which aggressively manipulates the streaming pipeline, as if the library calls were syntax nodes of an interpreted program. A pipeline of the form "`of(...).filter(...).map(...).sum()`" is formed with `sum` being at the outermost layer, i.e., right-to-left as far as surrounding

code is concerned. However, when the terminal operator (`sum`) is reached, it starts evaluation over the stream data by eventually invoking an iteration method in operator `of`. It is this method that drives iteration and calls the operators left-to-right. The result of such manipulation is significant performance gains.

We discussed two major designs of streaming libraries, push and pull. Although we study two distinct designs, the set of operators supported remains the same. The design decisions adopted by the library-author of a streaming library (normally) do not affect (at least syntactically) the domain-specific sub-language for stream–oriented programming. In the following chapter we will present a new library design that focuses on the ability to have pluggable designs (semantics) and an extensible set of operators.

4. PLUGGABLE SEMANTICS FOR STREAMS

The problem with existing library designs is that there is no way to alter the semantics of a streaming pipeline without changing the library itself. This is detrimental to library extensibility. For instance, a user may want to extend the library in any of the ways below:

- Create push-vs-pull versions of all operators.
- Create a logging interpretation of a pipeline, which logs actions and some intermediate results.
- Create an interpretation computing asynchronous versions of an evaluation (futures-of-values instead of values).
- Create an optimizing interpretation that fuses together operators, such as neighboring `filter`s or `map`s.

Additionally, the current architecture of streaming libraries prevents the introduction of new operators, precisely because of the inflexible way that evaluation is performed. As discussed above, Java streams introduce push-style iteration by default. This approach would yield semantic differences from pull-style iteration if more operators, such as `zip`, were added to the library. Furthermore, in some languages the addition of new operators requires editing the library code or using advanced facilities: in Java such addition is only possible by changing the library itself, while in C# one needs to use extension methods, and in Scala one needs to use implicits.

4.1 Introducing the StreamAlg design

In this chapter, we propose a new design and architecture for streaming libraries *à la carte* to maximize extensibility. Our approach requires no language changes, and only leverages features found across all languages examined—i.e., standard parametric polymorphism (generics). We argue for the benefits of this design in terms of extensibility and low adoption barrier (i.e., use of only standard language features), all without sacrificing performance. Additionally, we demonstrate extensibility and provide several alternative semantics for streaming pipelines, all in an actual, publicly available implementation. Finally, we provide an example of the use of object algebras in a real-world, performance-critical setting.

Underlying our architecture is the object algebra construction of Oliveira and Cook [124] and Oliveira et al. [125]. This is combined with a library design that dissociates the push or pull nature of iteration from the operators themselves, analogously to the recent “defunctionalization of push arrays” approach in the context of Haskell [166].

Based on this architecture, we have implemented an alternative stream library for Java: *StreamAlg*. In StreamAlg, the pipeline shown earlier gets inverted and parameterized

by an `alg` object, which designates the intended semantics. For instance, a plain Java-streams-like evaluation would be written as in Figure 4.1.

```
PushFactory alg = new PushFactory();
int sum = Id.prj(
            alg.sum(
              alg.map(x → x * x,
                alg.filter(x → x % 2 == 0,
                  alg.source(v))))).value;
```

Figure 4.1: Example pipeline with push-based semantics

(The `Id.prj` and `value` elements in Figure 4.1 are part of a standard pattern for simulating higher-kinded polymorphism with plain generics. They can be ignored for the purposes of understanding our architecture. We discuss the pattern in detail in Section 4.3.)

Although the code in Figure 4.1 is slightly longer than pipelines we show in Chapter 3, its elements are highly stylized. The user can adapt the code to other pipelines with trivial effort, comparable to that of the original code fragment in Java 8 streams. Most importantly, if the user desired a different interpretation of the pipeline, the only necessary change is to the first line of the example. An interpretation that has pull semantics and fuses operators together only requires a new definition of `alg`:

```
FusedPullFactory alg = new FusedPullFactory();
... // same as earlier
```

Figure 4.2: Declaration of an interpretation

Such new semantics can be defined externally to the library itself. Adding `FusedPullFactory` requires no changes to the original library code, allowing for semantics that the library designer had not foreseen.

This highly extensible design comes at no cost to performance. The new architecture introduces no extra indirection and does not prevent the JIT compiler from performing any optimization. This is remarkable, since current Java 8 streams are designed with performance in mind (cf. the earlier push-style semantics). As we show, StreamAlg matches or exceeds the performance of Java 8 streams.

4.2 Stream Algebras = Streams + Object Algebras

We next describe the architecture of StreamAlg and its design elements, including separate push and pull semantics, enhanced interpretations of a pipeline, optimizations and more.

4.2.1 Motivation

The goal of our work is to offer extensible streaming libraries. The main axis of extensibility that is not well-supported in past designs is that of pluggable semantics. In existing streaming libraries there is no way to change the evaluation behavior of a pipeline so that it performs, e.g., lazy evaluation, augmented behavior (e.g., logging), operator fusing, etc. Currently, the semantics of a stream pipeline evaluation is hard-coded in the definition of operators supplied by the library. The user has no way to intervene.

The original motivation for our work was to decouple the pull- vs. pull-style iteration semantics from the library operators. As discussed in Chapter 3, Java 8 streams are push-style by default, while Scala, C#, and F# streams are pull-style. A recent approach in the context of Haskell [166] performs a similar decoupling of push- vs. pull-style semantics through defunctionalization of the interface, yet affords no other extensibility.

Although Java 8 streams allow some pull-style iteration, they do not support fully pluggable pull-style semantics. The current pull-style functionality is via the `iterator()` operator. This operator is a terminal operator and adapts a push-style pipeline into an iterator that can be used via the `hasNext`/`next` methods. This is subtly different from changing the semantics of an entire pipeline into pull-style iteration.

For instance, the `flatMap` operator takes as input a function that produces streams, applies it to each element of a stream, and concatenates the resulting streams. In a true pull-style iteration, it is not a problem if any of the intermediate streams happen to be infinite (or merely large): their elements are consumed as needed. This is not the case when a Java 8 `flatMap` pipeline is made pull-style with a terminal `iterator` call. Figure 4.3 shows a simple example. Stream `s` is infinite: it starts with zero and its step function keeps adding 2 to the previous element. The `flatMap` application produces modified copies of the infinite stream `s`, with each element multiplied by those of a finite array, `v`. Evaluation does not end until an out-of-memory exception is raised.[1]

StreamAlg's design removes such issues, allowing pipelines with pluggable semantics. Although the separation of pull- and push-style semantics was our original motivation, it soon became evident that an extensible architecture offers a lot more options for semantic extensibility of a stream pipeline. We discuss next the new architecture and several semantic additions that it enables.

[1] This is a known issue, which we have discussed with Java 8 streams implementors, and does not seem to have an easy solution. The underlying cause is that the type signatures of operators (e.g., `of` or `flatMap`) encode termination conditions as return values from downstream operators. For `flatMap` to avoid confusing the conditions from its parameter stream (result of `map` in this example) and its downstream (`iterator` in the example) it needs to evaluate one more element of the parameter stream than strictly needed, and that element happens to be infinite in the example.

```
Stream<Long> s = Stream.iterate(0L, i → i + 2);

Iterator<Long> iterator = Stream
  .of(v)
  .flatMap(x → s.map(y → x * y))
  .iterator();

iterator.hasNext();
```

Figure 4.3: Infinite streams and flatMap

4.2.2 Stream as Multi-Sorted Algebras

StreamAlg's extensible, pluggable-semantics design is implemented using an architecture based on object algebras. Object algebras were introduced by Oliveira and Cook [124] as a solution to the *expression problem* [185]: the need to have fully extensible data abstraction while preserving the modularity of past code and maintaining type safety. The need for extensibility arises in two forms: adding new data variants and adding new operations. Intuitively, an object algebra is an interface that describes method signatures for creating syntax nodes (data variants). An implementation of the algebra offers semantics to such syntax nodes. Thus, new data variants (syntax nodes) are added by extending the algebra, while new operations (semantics) correspond to different implementations of the algebra.

We next present the elements of the object algebra approach directly in our streaming domain.

In our setting, the set of variants to extend are the different operators: `map`, `take` (called `limit` in Java), `filter`, `flatMap`, etc. These are the different cases that a semantics of stream evaluation needs to handle. The "operations" on those variants declare the manipulation/transformation that will be employed for all produced data items. We will use the term "behavior" for such operations.

Our abstraction for streams is a multi-sorted algebra. The two sorts that can be evolved as a *family* are the type of the stream, which can hold some type of values, and the type of the value produced by terminal operations. The signature of the former is called `StreamAlg` while the latter is `ExecStreamAlg`. The `Exec*` prefix is used to denote that this is the algebra for the types that perform execution. The algebras are expressed as generic interfaces and classes implementing these interfaces are *factories*. In our multi-sorted algebra these two distinct parts are connected with the subtyping relation and classes that implement the two interfaces can evolve independently, to form various combinations.

Intermediate Operators. Our base interface, `StreamAlg`, is shown in Figure 4.4.

```
interface StreamAlg<C<_>> {
    <T>    C<T> source(T[] array);
    <T, R> C<R> map(Function<T, R> f, C<T> s);
    <T, R> C<R> flatMap(Function<T, C<R>> f, C<T> s);
    <T>    C<T> filter(Predicate<T> f, C<T> s);
}
```

Figure 4.4: Basic interface of StreamAlg

As can be seen, `StreamAlg` is parameterized by a unary type constructor that we denote by the `C<_>` syntax. This is a device used for exposition. That is, for the purposes of our presentation we assume *type-constructor* polymorphism (a.k.a. higher-kinded polymorphism): the ability to be polymorphic on type constructors. This feature is not available in Java (although it is in, e.g., Scala).[2] In our actual implementation, type-constructor polymorphism is emulated via a standard stylized construction, which we explain in Section 4.3.

Every operator of streams is also a constructor of the corresponding algebra; it returns (*creates*) values of the abstract set. Each constructor of the algebra creates a new intermediate node of the stream pipeline and, in addition to the value of the previous node (parameter `s`) that it will operate upon, it takes a *functional interface*. (A functional interface has exactly one abstract method and is the type of a lambda in Java 8.)

Terminal Operators. The `ExecStreamAlg` interface describes terminal operators, which trigger execution/evaluation of the pipeline. These operators are also parametric. They can return a scalar value or a value of some container type (possibly parameterized by some other type). For instance, `count` can return `Long`, hence having blocking (synchronous) semantics, or it can return `Future<Long>`, to offer asynchronous execution.

```
interface ExecStreamAlg<E<_>, C<_>> extends StreamAlg<C> {
    <T> E<Long> count(C<T> s);
    <T> E<T>    reduce(T identity, BinaryOperator<T> acc, C<T> s);
}
```

Figure 4.5: Extending basic interface with terminal operators

Once again, this algebra is parameterized by unary type constructors and it also carries as a parameter the abstract stream type that it will pass to its super type, `StreamAlg`.

[2] The original object algebras work of Oliveira and Cook [124] did not require type-constructor polymorphism for its examples. Later work by Oliveira et al. [125] used type-constructor polymorphism in the context of Scala.

4.2.3 Adding New Behavior for Intermediate Operators

We next discuss the extensibility that our design affords, with several examples of different interpretation semantics.

Push Factory. The first implementation in StreamAlg is that of a push-style interpretation of a streaming pipeline, yielding behavior equivalent to the default Java 8 stream library.

Push-style streams implement the `StreamAlg<Push>` interface (where `Push` is the *container* or *carrier type* of the algebra). All operators return a value of some type `Push<T>` for some `T`, i.e., a type expression derived from the concrete constructor `Push`. Our `PushFactory` implementation, restricted to operators `source` and `map`, is shown below.

```
class PushFactory implements StreamAlg<Push> {
  public <T> Push<T> source(T[] array) {
    return k → {
      for(int i=0 ; i < array.length ; i++){
        k.accept(array[i]);
      }
    };
  }

  public <T, R> Push<R> map(Function<T, R> mapper, Push<T> s) {
    return k → s.invoke(i → k.accept(mapper.apply(i)));
  }
}
```

Figure 4.6: Example of a PushFactory

A `Push<...>` type is the embodiment of a push-style evaluation of a stream. It carries a function, which can be composed with others in a *push-y* manner. In the context of Java, we want to be able to assign lambdas to a `Push<...>` reference. Therefore we declare `Push<X>` as a functional interface, with a single method, `void invoke(Consumer<T>)`. This consumer can be thought of as the *continuation* of the evaluation (hence the conventional name, `k`). The entire stream is evaluated as a loop, as shown in the implementation of the `source` operator, above. `source` returns a lambda that takes as a parameter a `Consumer<T>`, iterates over the elements of a source, `s`, and passes elements one-by-one to the consumer.

Similarly, the `map` operator returns a push-stream embodiment of type `Push<...>`. This stream takes as argument another stream, `s`, such as the one produced by `source`, and invokes it, passing it as argument a lambda that represents the `map` semantics: it calls its continuation, `k`, with the argument (i.e., the element of the stream) as transformed by the mapping function. This pattern follows a similar continuation-passing-style convention

as in the original Java 8 streams library. (As discussed in Chapter 3, this reversal of the pipeline flow enables significant VM optimizations and results in faster code.)

The next operator, whichever it is, will consume the transformed elements of type `R`. The implementation of other operators, such as `filter` and `flatMap`, follows a similar structure.

Pull Factory. As discussed earlier, Java 8 streams do not have a full pull-style iteration capability. They have to fully realize intermediate streams, since the pull semantics is implemented as a terminal operator and only affects the external behavior of an entire pipeline. (As we will see in our experiments of Section 4.5, this is also a source of inefficiency in practice.) Therefore, the first semantic addition in StreamAlg is pull-style streams.

Pull-style streams implement the `StreamAlg<Pull>` interface. In this case `Pull<T>` is an interface that represents iterators, by extending the `Iterator<T>` interface. For pull semantics, each operator returns an anonymous class—one that implements this interface by providing definitions for the `hasNext` and `next` methods. In Figure 4.7 we demonstrate the implementation of the `source` and `map` operators, which are representative of others.

We follow the Java semantics of iterators (the effect happens in `hasNext`). Each element that is returned by the `next` method of the `map` implementation is the transformed one, after applying the needed mapper lambda to each element that is retrieved. The retrieval is realized by referring to the `s` object, which carries the iterator of the previous pipeline step.

Note how dissimilar the `Push` and `Pull` interfaces are (a lambda vs. an iterator with `next` and `hasNext`). Our algebra, `StreamAlg<C<_>>` is fully agnostic regarding `C`, i.e., whether it is `Push` or `Pull`.

Log Factory. With a pluggable semantics framework in place, we can offer several alternative interpretations of the same streaming pipeline. One such is a logging implementation. The log factory expresses a cross-cutting concern, one that interleaves logging capabilities with the actual execution of the pipeline. Although the functionality is simple, it is interesting in that it takes a mixin form: it can be merged with other semantics, such as push or pull factories. The code for the `LogFactory`, restricted to the `map` and `count` operators, is shown in Figure 4.8.

The code employs a delegation-based structure, one that combines an implementation of an execution algebra (of any behavior for intermediates and orthogonally of any behavior for terminal operators) with a logger. We parameterize `LogFactory` with an `ExecStreamAlg` and then via delegation we pass the intercepted lambda as the mapping lambda of the internal algebra. For example, if the developer has authored a pipeline `alg.reduce(0L, Long::sum, alg.map(x -> x + 2, alg.source(v)))`, then, instead of using an `ExecPushFactory` that will perform push-style streaming, she can pass a `LogFactory<>(new ExecPushFactory())` effectively mixing a push factory with a log factory.

```
class PullFactory implements StreamAlg<Pull> {
  public <T> Pull<T> source(T[] array) {
    return new Pull<T>() {
      final int size = array.length;
      int cursor = 0;

      public boolean hasNext() {
        return cursor != size;
      }

      public T next() {
        if (cursor >= size)
          throw new NoSuchElementException();
        return array[cursor++];
      }
    };
  }

  public <T, R> Pull<R> map(Function<T, R> mapper, Pull<T> s) {
    return new Pull<R>() {
      R next = null;

      public boolean hasNext() {
        while (s.hasNext()) {
          T current = s.next();
          next = mapper.apply(current);
          return true;
        }
        return false;
      }

      public R next() {
        if (next != null || this.hasNext()) {
          R temp = this.next;
          this.next = null;
          return temp;
        } else
        throw new NoSuchElementException();
      }
    };
  }
}
```

Figure 4.7: Example of PullFactory functionality

```
class LogFactory<E<_>, C<_>> implements ExecStreamAlg<E, C> {
  ExecStreamAlg<E, C> alg;

  <T, R> C<R> map(Function<T, R> mapper, C<T> s) {
    return alg.map(i → {
      System.out.print("map: " + i.toString());
      R result = mapper.apply(i);
      System.out.println(" → " + result.toString());
      return result;
    }, s);
  }

  public <T> E<Long> count(C<T> s) {
    return alg.count(s);
  }
}
```

Figure 4.8: Example of LogFactory functionality

Fused Factory. An interpretation can also apply optimizations over a pipeline. The optimization is applied automatically, as long as the user chooses an evaluation semantics that enables it. This is effected with an extension of a `PullAlgebra` that performs fusion of adjacent operations. Using a `FusedPullFactory` the user can transparently enable fusion for multiple `filter` and multiple `map` operations. In this factory, the two operators are redefined and, instead of creating values of an anonymous class of type `Pull`, they create values of a refined version of the `Pull` type. This gives introspection capabilities to the `map` and `filter` operators. They can inspect the dynamic type of the stream that they are applied to. If they operate on a fusible version of `map` or on a fusible version of `filter` then they proceed with the creation of values for these extended types with the composed operators. We elide the definition of the factory, since it is lengthy.

In Figure 4.9 we demonstrate the re-definition of the map operator and the specific subtype of `Pull` that is used `FusibleMapPull`. This interface extends `Pull` and defines a factory method to create a new iterator with composed functions and the original source as parameters.

4.2.4 Adding New Operators

StreamAlg also allows adding new operators without changing the library code. In case we want to add a new operator, we first have to decide in which algebra it belongs. For instance, we have added a `take` operator without disturbing the original algebra definitions. A `take` operator has signature `C<T> take(int n)` so it clearly belongs in `StreamAlg`. We

```
class ExecFusedPullFactory extends ExecPullFactory
                           implements ExecStreamAlg<Id.t, Pull.t> {
  public <T, R> Pull<R> map(Function<T, R> mapper, Pull<T> s) {
    if (self instanceof FuseMapPull) {
      return ((FuseMapPull<T>) s).compose(mapper);
    } else {
      return new FusibleMapPull<>(s, mapper);
    }
  }

  interface FuseMapPull<T> extends Pull<T> {
    <S> Pull<S> compose(Function<T, S> other);
  }

  class FusibleMapPull<T, R> implements FuseMapPull<R> {
    private final Pull<T> source;
    private final Function<T, R> mapper;

    public FusibleMapPull(Pull<T> source, Function<T, R> mapper) { ... }

    public <S> FuseMapPull<S> compose(Function<R, S> other) {
      return new FusibleMapPull<>(source, other.compose(mapper));
    }

    public boolean hasNext() { ... }
    public R next() { ... }
  }
}
```

Figure 4.9: Composition of operators in ExecFusedPullFactory

have to implement the operator for both push and pull streams, but we want to allow the possibility of using `take` with any `ExecStreamAlg`. Our approach again uses delegation, much like the `LogFactory`, shown earlier in Figure 4.8. We create a generic `TakeStreamAlg<E, C>` interface and orthogonally we create an interface `ExecTakeStreamAlg<E, C>` that extends `TakeStreamAlg<C>` and `ExecStreamAlg<E, C>`. In the case of push streams, `ExecPushWithTakeFactory<E>` implements the interface we created, where `C = Push`, by defining the `take` operator. All other operators for the push case are inherited from the `PushFactory` supertype. The `ExecPushWithTakeFactory<E>` factory is parameterized by `ExecStreamAlg<E, Push> alg`. Generally, the factory can accept as parameter any algebra for terminal operators.

```
class ExecFutureFactory<C<_>> implements ExecStreamAlg<Future, C> {
  private final ExecStreamAlg<Id, C> execAlg;
  public <T> Future<Long> count(C<T> s) {
    Future<Long> future = new Future<>(() → {
      return execAlg.count(s).value;
    });
    future.run();
    return future;
  }
  public <T> Future<T> reduce(T identity,
                              BinaryOperator<T> accumulator,
                              C<T> s) {
    Future<T> future = new Future<>(() → {
      return execAlg.reduce(identity, accumulator, s).value;
    });
    future.run();
    return future;
  }
}
```

Figure 4.10: Count and reduce operators in FutureFactory

4.2.5 Adding New Behavior for Terminal Operators

Future Factory. StreamAlg also enables adding new behavior for terminal operators. The most interesting example in our current library components is that of `FutureFactory`: an interpretation of the pipeline that triggers an asynchronous computation. Instead of returning scalar values, a `FutureFactory` (shown in Figure 4.10) parameterizes `ExecStreamAlg` with a concrete type constructor, `Future<X>`.[3] (This is in much the same way as, e.g.,

[3]That is, `Future` is our own class, which extends the Java library class `FutureTask`, and not to be confused with the Java library `java.util.concurrent.Future` interface.

a `PushFactory` parameterizes `StreamAlg` with type constructor `Push`, in Figure 4.6.) `Future` is a type that provides methods to start and cancel a computation, query the state of the computation, and retrieve its result.

`FutureFactory` defines terminal operators `count` and `reduce`, to return `Future<Long>` and `Future<T>` respectively. Intermediate operators are defined similarly to the terminal ones, but are omitted from the figure.

4.3 Emulating Type-Constructor Polymorphism

As noted earlier, our presentation so far was in terms of type-constructor polymorphism, although this is not available in Java. For our implementation, we simulate type-constructor polymorphism via a common technique. The same encoding has been used in the implementation of object-oriented libraries—e.g., in type classes for Java [63] and in finally tagless interpreters for C# [105]. The technique was also recently presented formally by Yallop and White [192], and used to represent higher-kinded polymorphism in OCaml.

In this encoding, for an unknown type constructor `C<_>`, the application `C<T>` is represented as `App<t, T>`, where `T` is a Java class and `t` is a marker class that identifies the type constructor `C`. For example, our stream algebra shown in Section 4.2.2 is written in plain Java as follows:

```
public interface App<C, T> { }

public interface StreamAlg<C> {
    <T> App<C, T> source(T[] array);
    <T, R> App<C, R> map(Function<T, R> f, App<C, T> app);
    <T, R> App<C, R> flatMap(Function<T, App<C, R>> f, App<C, T> app);
    <T> App<C, T> filter(Predicate<T> f, App<C, T> app);
}
```

Figure 4.11: Type constructor encoding

A subtle point arises in this encoding: given `C`, how is `t` generated? This class is called the “brand”, as it tags the application so that it cannot be confused with applications of other type constructors; this brand should be extensible for new types that may be added later to the codebase. This means that (a) `t` should be a fresh class name, created when `C` is declared; and (b) there should be a protocol to ensure that the representation is used safely.

Brand freshness. The freshness of the brand name is addressed by declaring `t` as a nested class inside the class of the new type constructor. Since `t` exists at a unique

point in the class hierarchy, no other class may declare a brand that clashes with it, and its declaration happens at the same time as `C` is declared. In the following, we see the encoding of the type constructor `Pull<T>`, with its `t` brand:

```
public interface Pull<T> extends App<Pull.t, T>, Iterator<T> {
    static class t { }
    static <A> Pull<A> prj(App<Pull.t, A> app) { return (Pull<A>) app; }
}
```

Figure 4.12: A type constructor with its brand

We see that the encoding above has an extra method `prj`, which does a downcast of its argument. The OCaml encoding of Yallop and White needs two methods `inj` and `prj` (for "inject" and "project") that cast between the concrete type and the instantiation of the type application. In Java, we define `prj`, which takes the representation of the type application and returns the actual `Push<T>` instantiation. In contrast to OCaml, Java has subtyping, so `inj` functions are not needed: a `Pull<T>` object can always be used as being of type `App<Pull.t, T>`. The `Iterator` interface in the declaration above is not related to the encoding, but is part of the semantics of pull-style streams.

Safely using the encodings. This encoding technique has a single unchecked cast, in the `prj` function. The idea is that the cast will be safe if the only way to get a value of type `App<Pull.t, X>` (for any `X`) is if it is really a value of the subtype, `Pull<X>`. This property clearly holds if values of type `App<Pull.t, X>` (or values of any type involving `Pull.t`) are never constructed. In the Yallop and White technique for OCaml, this is ensured syntactically by the "freshness" of the brand, `t`, which is private to the type constructor. In Java, the property is ensured by convention: every subtype `S` of `App` has a locally defined brand `t` and no subtype of `App<S.t, X>` other than `S` exists.

Type expressions without type-constructor polymorphism. Another detail of the encoding is the representation of type expressions that are not parametric according to a type constructor; for those we need an identity type application, `Id`.

By using the type `Id` (Figure 4.13), a scalar value of type `Integer` can be represented as `Id<Integer>`, and a non blocking one as `Future<Integer>`.

4.4 Using Streams

With the encoding of type-constructor polymorphism, our description of the library features is complete. A user can employ all operators to build pipelines, and can flexibly choose the semantics of these pipelines.

```
public class Id<T> implements App<Id.t, T> {
    public final T value;
    public Id(T value) { this.value = value; }
    public static class t { }
    public static <A> Id<A> prj(App<Id.t, A> app) { return (Id<A>) app; }
}
```

Figure 4.13: Identity type application

The following example declares a pipeline that filters long integers and then counts them. The expression assumes an implementation, `alg`, of a stream algebra. Note that the prefix, `Id.prj`, and suffix, `value`, of the pipeline expression are only needed for our type-constructor polymorphism simulation.

```
Long result = Id.prj(alg.count(
                      alg.filter(x → x % 2L == 0,
                        alg.source(v)))).value;
```

Figure 4.14: Sum of squares, in StreamAlg

Similarly, `cart` in the example below constructs a sum of the cartesian product pipeline between two arrays. The factory object (implementing the algebras) is factored out and becomes a parameter of the method `cart`.

```
<E, C> App<E, Long> cart(ExecStreamAlg<E, C> alg) {
  return alg.reduce(0L, Long::sum,
           alg.flatMap(
             x → alg.map(y → x * y,
                    alg.source(v2)),
             alg.source(v1)));
}
```

Figure 4.15: Cartesian product, in StreamAlg

The above can be used with any of the various semantics factories presented in Section 4.2, depending on the kind of evaluation the user wants to perform. We present a summary of all the combinations of factories that can be used. The first five expressions return a scalar value `Long` and the last two a `Future<Long>`.

```
Id.prj(cart(new ExecPushFactory())).value;

Id.prj(cart(new ExecPullFactory())).value;

Id.prj(cart(new ExecFusedPullFactory())).value;

Id.prj(cart(new LogFactory<>(new ExecPushFactory()))).value;

Id.prj(cart(new LogFactory<>(new ExecPullFactory()))).value

Future.prj(cart(new ExecFutureFactory<>(new ExecPushFactory())));

Future.prj(cart(new ExecFutureFactory<>(new ExecPullFactory())));
```

Figure 4.16: Various interpretations for the same pipeline

4.5 Performance

It is interesting to assess the performance of our approach, compared to the highly optimized Java 8 streams. Since our techniques add an extra layer of abstraction, one may suspect they introduce inefficiency. However, there are excellent reasons why our design can yield high performance:

- Object algebras are used merely for pipeline construction and not for execution. Once the data processing loop starts, it should be as efficient as in standard Java streams.
- Our design offers fully pluggable semantics. This is advantageous for performance. We can leverage fusion of operators, proper pull-style iteration without materialization of full intermediate results, and more.

Our benchmarks aim to showcase these two aspects. In this sense, some of the benchmarks are unfair to Java 8 streams: they explicitly target cases for which we can optimize better. We point out when this is the case.

We use a set of microbenchmarks offering various combinations of streaming pipelines: [4]

- **reduce**: a sum operation.
- **filter/reduce**: a filter-sum pipeline.

[4]The benchmark programs are adapted from Chapter 3 were we saw that Java 8 streams typically significantly outperform other implementations. Still, performance of streaming libraries lags behind hand-optimized code. This is to be expected, since hand-written code can remove most overhead of lazy evaluation, by fusing the consumer of data with the producer. JDK developers have shown significant interest in future VM optimizations that will allow Java streams to approach the performance of hand-written code [151].

- **filter/map/reduce**: a filter-map-sum pipeline.
- **cart/reduce**: a nested pipeline with a `flatMap` and an inner operation, with a `map` (capturing a variable), to encode the sum of a Cartesian product.
- **fused filters**: 8 consecutive filter operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for StreamAlg.
- **fused maps**: 8 consecutive map operations and a count terminal operation. The implementation is push-style for Java 8, push-style, pull-style & fused for StreamAlg.
- **count**: a count operation (pull-style).
- **filter/count**: a filter-count pipeline (pull-style).
- **filter/map/count**: a filter-map-count pipeline (pull-style).
- **cart/take/count**: a nested pipeline with a `flatMap` and an inner operation, with a `map`, to encode taking the first few elements of a Cartesian product and then counting them (pull-style).

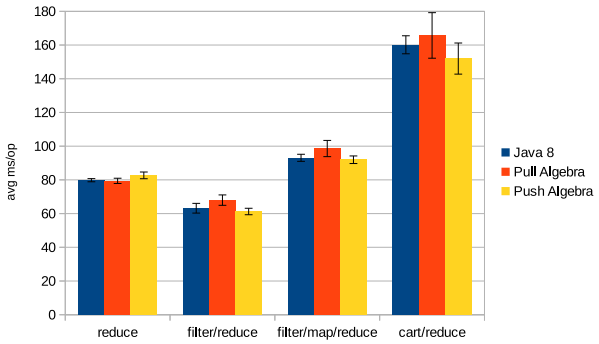
Although StreamAlg is not yet full-featured, it faithfully (relative to Java 8 streams) implements the facilities tested in these benchmarks.

Input. All tests were run with the same input set. For all benchmarks except **cart/reduce** and **cart/take/count** we used an array of $N = 10,000,000$ `Long` integers (boxed),[5] produced by N integers with a `range` function that fills the arrays procedurally. The **cart/reduce** test iterates over two arrays. An outer one of $10,000,000$ long integers and an inner one of 10. For the **cart/take/count** test, the sizes of the inner and outer arrays are reversed and the `take` operator draws only the first $n = 100,000$ elements. Fusion operations use $1,000,000$ long integers.

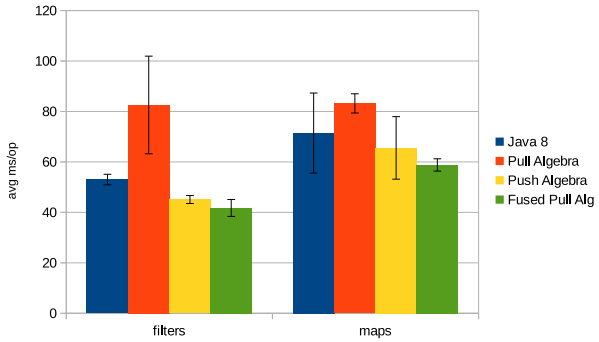
Setup. We use a Fedora Linux x64 operating system (version 3.17.4-200.fc20) that runs natively on an Intel Core i5-3360M vPro 2.8GHz CPU (2 physical x 2 logical cores). The total memory of the system is 4GB. We use version 1.8.0.25-4.b18 of the Open JDK.

Automation. We used the Java Microbenchmark Harness (JMH) [157]: a benchmarking tool for JVM-based languages that is part of the OpenJDK. JMH is an annotation-based tool and takes care of all intrinsic details of the execution process, in order to remove common experimental biases. The JVM performs JIT compilation so the benchmark author must measure execution time after a certain warm-up period to wait for transient responses to settle down. JMH offers an easy API to achieve that. In our benchmarks we employed

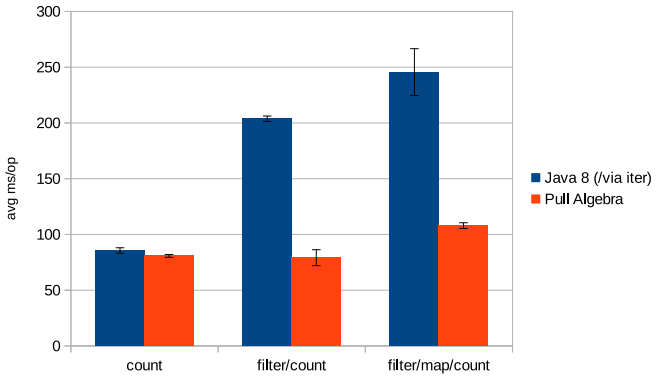
[5] Specialized pipelines for primitive types are not supported in StreamAlg, but should be a valuable future engineering addition.



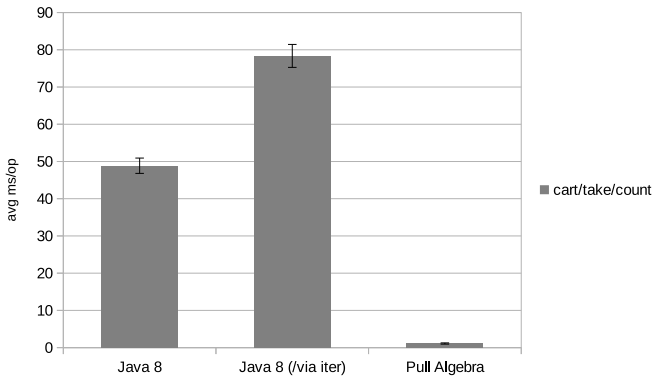
(a) Basic pipelines



(b) Fusible pipelines



(c) Pull based reduction



(d) Pull & Push based flatMap/take

Figure 4.17: Microbenchmarks on JVM in milliseconds / operation (average of 10).

10 warm-up iterations and 10 proper iterations. We also force garbage collection before benchmark execution. Additionally, we used 2 VM-forks for all tests, to measure potential run-to-run variance. We have fixed the heap size to 3GB for the JVM to avoid heap resizing effects during execution.

Results. The benchmarks are cleanly grouped in 4 sets:

- In Figure 4.17a we present the results of the first 4 benchmarks: **reduce**, **filter/reduce**, **filter/map/reduce**, and **cart/reduce**. These are "fair" comparisons, of completely equivalent functionality in the two libraries. As can be seen, the performance of our push algebra implementation matches or exceeds that of Java 8, validating the claim that our approach does not incur undue overheads.

- Figure 4.17b presents the results for the next two benchmarks: **fused filters** and **fused maps**. These benchmarks are intended to demonstrate the improvement from our fusing semantics. The Java 8 implementation compared is push-style. Still, our fused pull-style semantics yield a successful optimization, outperforming even the efficient, push-style iteration. Due to our design, this optimization is achieved modularly and independently of the rest of the stream implementation.

- Figure 4.17c includes the next 3 benchmarks: **count**, **filter/count**, and **filter/map/count**. These are benchmarks of semantically uneven implementations. Java 8 streams support pull-style functionality by transforming the stream into an iterator, but this is not equivalent to full pull-style iteration. As can be seen, a true pull semantics for all operators can be much faster.

- Finally in Fig 4.17d we show the **cart/take/count** benchmark. This benchmark contains a pipeline that is pathological for the Java pull-style streams, in much the same way as the infinite evaluation of Section 4.2.1. (Push-style streams have the same pathology, by definition: they cannot exploit the fact that only a small number of results are needed in the final operator.) Instead of an infinite stream we reproduce the nested `flatMap`/`map` pipeline but with the larger array being the nested one. As `flatMap` needs to materialize nested arrays (effectively applying the inner `map` function to create the inner stream) it suffers from the effect of inner allocations. Our proper pull-style pipeline does not present this behavior. The result is spectacular in favor of our pull algebra implementation, because of the small number of elements actually needed by the `take` operator.

4.6 Discussion

We next present observations related to the design of StreamAlg and its constituent elements.

4.6.1 Fluent API

Object Algebras drive the "interpretation" of streams in our work, so a nested, reversed pattern occurs when declaring the operators of a pipeline: instead of chained methods such as "`of(...).filter(...).count()`", our pipeline is declared by means of operators applied to values "`alg.count(alg.filter(..., alg.source(...)))`". This reversed pattern follows the declaration order of the operators, contradicting the natural ordering of a fluent API. A fluent API (or fluent interface) promotes code readability and is usually implemented by using method chaining [52].

We created an experimental fluent API in Java using static methods in the interface of the object algebra,[6] but the result was cumbersome. In contrast, we created skeletal C# and Scala implementations of a fluent API for our library design, for demonstration purposes. (Our object algebra streaming architecture applies to any modern OO language with generics, so C# and Scala libraries based on the same principles can be developed in the future.)

In C# the user can create a fluent API through the use of extension methods. Extension methods enable the user to "add" methods to *existing types* without creating a new derived type, recompiling, or modifying the original type. Extension methods are simply a compiler shorthand that enables static methods to be called with instance syntax. Using that feature, the user can create a static class enclosing extension methods that capture the reversed flow.

Figure 4.18 shows the relevant C# code snippet. Extension methods are defined for the function type, `Func<F, App<C, T>>`. The user wraps all methods of an algebra with methods that, instead of returning `C<T>`, return a function that takes as parameter an algebra object. The algebra object is, thus, hidden from the original code and introduced implicitly in calls to such returned functions.

The above technique enables fluent ordering, as shown in the `Example` method of the figure. The fluent API also has immediate side benefits in current programming tools: the user is able to retrieve the list of operators via the intelligent code completion feature of the IDE.

We also retrieve fluency in Scala using a similar technique, enabled by the feature of implicit classes [126] as shown in Figure 4.19.

4.7 Summary of an Extensible Design of Streams

We presented an alternative design for streaming libraries, based on an object algebras architecture. Our design requires only standard features of generics and is, thus, widely applicable to modern OO languages, such as Java, Scala, and C#. We implemented a Java streaming library based on these principles and showed its significant benefits, in

[6] Static methods in interfaces is a feature introduced in Java 8.

```
static class Stream {
  public static
  Func<F, App<C, T>> OfArray<F, C, T>(T[] array) where F : IStreamAlg<C> {
    return alg => alg.OfArray(array);
  }

  public static
  Func<F, App<C, T>> Filter<F, C, T>(this Func<F, App<C, T>> streamF,
                                     Func<T, bool> predicate)
    where F : IStreamAlg<C> {
    return alg => alg.Filter(streamF(alg), predicate);
  }

  public static
  Func<F, App<E, int>> Count<F, E, C, T>(this Func<F, App<C, T>> streamF)
    where F : IExecStreamAlg<E, C> {
    return alg => alg.Count(streamF(alg));
  }

  public static
  App<E, int> Example<E, C, F>(int[] data, F alg)
    where F : IExecStreamAlgebra<E, C> {
    Func<F, App<E, int>> streamF =
          Stream.OfArray(data)
                .Filter(x => x % 2 == 0)
                .Count();
    return streamF(alg);
  }
}
```

Figure 4.18: Example of Fluent API creation in C#

```
object Stream {
  trait StreamAlg[C[_]] {
    def ofArray[T](array: Array[T]) : C[T]
    def filter[T](f : T => Boolean, app : C[T]) : C[T]
  }

  trait ExecStreamAlg[C[_], E[_]] extends StreamAlg[C] {
    def count[T](app : C[T]) : E[Long]
  }

  def ofArray[T, C[_], E[_], F <: ExecStreamAlg[C, E]]
             (array: Array[T]) : F => C[T] = {
    alg => alg.ofArray(array)
  }

  trait Push[T]
  trait Pull[T]
  type  Id[A] = A

  implicit class RichReader[T, C[_], E[_], F <% ExecStreamAlg[C, E]]
                           (func : F => C[T]) {
    def filter(p : T => Boolean) : F => C[T] = {
      alg => alg.filter(p, func(alg))
    }
    def count() : F => E[Long] = {
      alg => alg.count(func(alg))
    }
  }

  def example[T,  C[_], E[_]](array: Array[Int])
                             (alg : ExecStreamAlg[C, E]) : E[Long] = {
    Stream.ofArray[Int, C, E, ExecStreamAlg[C, E]](array)
          .filter((x:Int) => x%2==0)
          .count()(alg)
  }
}
```

Figure 4.19: Example of Fluent API creation in Scala

terms of transparent semantic extensibility, without sacrificing performance. Given our extensible library design, there are several avenues for further work. The clearest path is towards enriching the current library implementation with shared-memory parallel evaluation semantics, cloud evaluation semantics, distributed pipeline parallelism, GPU processing, and more. Since we expose the streaming pipeline, such additions should be transparent to current evaluation semantics, and can even be performed by third-party programmers.

Inspired by the level of flexibility that streams à la carte demonstrate, we apply the same technique to a full-blown programming language, that lacks of such mechanism. Java's syntax and semantics are not extensible, while other programming languages offer such facilities implemented in the general-purpose compiler of each language. It is apparent that when a certain programming style does not fit naturally into Java's idiomatic use, a programmer relies on library encodings of the desired style. For instance C# offers an implementation of coroutines via the use of `yield`. The standard streaming facility of .NET is based on the use of `yield` making the implementation of collections there much cleaner. Java does not have such keyword. Extending Java with `yield` (and defining its semantics) is like extending a DSL, as we saw in streams, with a new operator. Our goal is to design and implement a lightweight tool that can be introduced in the compilation pipeline of Java codebases, unlocking that potential for Java developers. We discuss this topic in the next chapter.

5. GENERALIZE EXTENSIBILITY FOR JAVA

Most programming languages provide some level of extensibility to their users. For example, C++ offers operator overloading, enabling the user to extend libraries and give syntactic sugar for operators between complex types. Programming languages with advanced type systems provide monads, as in Haskell and computation expressions, as in F#. Others, provide more pervasive mechanisms leaning towards metaprogramming facilities like macros. Nowadays, mainstream languages support both functional and object-based abstractions, but it is well-known that some programming patterns or idioms are hard to encapsulate using standard mechanisms. Examples include complex control-flow features, asynchronous programming, or embedded DSLs. It is possible to support various programming styles using library-based encodings without the aforementioned mechanisms, but this often leads to code that is verbose and tedious to maintain.

Consider the example of a simple extension for automatically closing a resource in Java:[1]

```
using (File f: IO.open(path)) { ... }
```

Figure 5.1: C#'s `using` construct

Such a language feature abstracts away the boilerplate needed to correctly close an IO resource, by automatically closing the `f` resource whenever the execution falls out of the scope of the block.

Unfortunately, in languages like Java, defining such constructs as a library is limited by inflexible statement-oriented syntax, and semantic aspects of (non-local) control-flow. For instance, one could try to simulate `using` by a method receiving a closure, such as:

```
void using(Closeable r,  Consumer<Closeable> block)
```

Figure 5.2: Attempt to implement an extension without any language support

However, the programmer can now no longer use non-local control-flow statements (e.g., `break`) within the closure block, and all variables will become effectively final as per the Java 8 closure semantics. Furthermore, encodings like this disrupt the conventional flow of Java syntax, and lead to an atypical, inverted code structure. More sophisticated idioms lead to even more disruption of the code style. For example, event-driven and interactive applications adopt an asynchronous model of execution to maintain interactivity with their surrounding components. As a consequence, a sequential program must be converted to one that is build around event handlers, also known as callbacks. "Callback hell" is the well-known nomenclature for describing the complicated situation when programming

[1] similar to C#'s `using` construct, or Java's try-with-resources

asynchronous code. Functional reactive programming, promises and futures [9, 10, 44, 84, 111, 113]

In this chapter we present Recaf,[2] a lightweight tool[3] to extend Java with custom dialects. Extension writers do not have to alter Java's parser or write any transformation rules. The Recaf compiler generically transforms an extended version of Java into code that builds up the desired semantics. Hence, Recaf is lightweight: the programmer can define a dialect without stepping outside the host language.

Recaf is based on two key ingredients:

- Syntactic extension: Java's surface syntax is liberated to allow the definition of new language constructs. Recaf *specifies* a set of syntactic patterns relying on existing control-flow or declaration constructs. The author of an extension uses a new keyword straight away, following one of these specified patterns. The Recaf compiler is responsible for *detecting* the pattern and translating the use of the keyword accordingly, as will see later in this chapter. For instance, the `using` construct follows the pattern of Java's `for`-each.
- Semantics extension: a single, syntax-directed transformation maps method bodies (with or without language extensions) to method calls on polymorphic factories which are responsible for constructing objects encapsulating the user-defined or customized semantics. The factories are developed within Java following the Object Algebras [124] design pattern.

The author of an extension can think of Recaf as a source-to-source translator and runtime library for developing extensions by defining denotations of the semantic domain in Java (not by extending the AST and evolving a family of non-extensible visitors).

The combination of the two aforementioned key points enables a range of application scenarios. For instance, Recaf can be used to: extend Java with new syntactic constructs (like `using`), modify or instrument the semantics of Java (e.g., to implement aspects like tracing, memoization), replace the standard Java semantics with a completely different semantics (e.g., translate Java methods to Javascript source code), embed DSLs into Java (e.g., grammars, or GUI construction), define semantics-parametric methods which support multiple interpretations, and combine any of the above in a modular fashion (e.g., combine `using` with a tracing aspect). Developers can define the semantics of new and existing constructs and create DSLs for their daily programming needs. Language designers can experiment with new features, by quickly translating their denotations in plain Java. In other words, Recaf brings the vision of "languages as libraries" to mainstream, object-oriented programming languages.

This chapter presents a transformation of Java statements and we show how the semantics can be defined as Object Algebras. We generalize the transformation to virtualize Java expressions, widening the scope of user defined semantics and we describe the

[2] As in *recaffeinating coffee* which describes the process of enhancing its caffeine content [34].
[3] The code is available at `https://github.com/cwi-swat/recaf` .

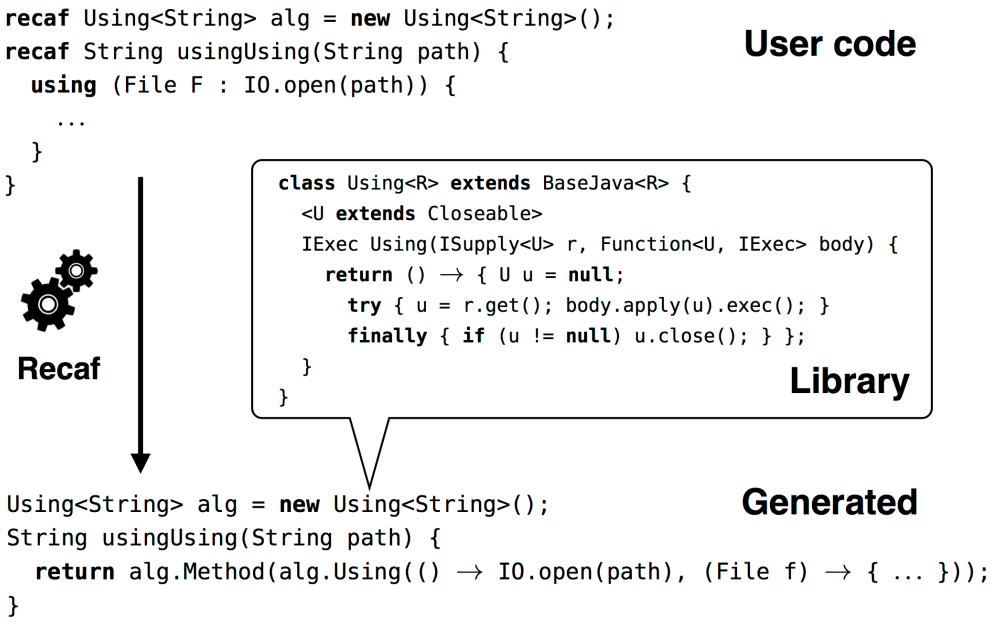


Figure 5.3: High level overview of Recaf

implementation of Recaf, and how it deals with certain intricacies of the Java language. We evaluate the expressiveness of Recaf with three case studies: i) providing language support for generators and asynchronous computations, ii) a minimal implementation of pull-based streams using the generator extension from the first case study and iii) creating a DSL for parser combinators. The results of the case studies and directions for future work are discussed in Section 5.6.

5.1 Introducing the Recaf compiler

Figure 5.3 gives a bird's eye overview of Recaf. It shows how the `using` extension is used and implemented with Recaf. The top shows a snippet of code illustrating how the programmer would use a Recaf extension, in this case consisting of the `using` construct. The programmer writes an ordinary method, decorated with the `recaf` modifier to trigger the source-to-source transformation. To provide the custom semantics, the user also declares a `recaf` variable, in scope of the `recaf` method. In this case, an object with static type of `Using<String>` is defined (`alg` in this example).

The downward arrow indicates Recaf's source-to-source transformation. Recaf **detects** that the new keyword relies on the `for`-each statement syntactically. An enhanced `for`-loop, in vanilla Java, omits explicit looping variables by operating over objects of type `*Iterable`. The new keyword, `using`, relies on the same pattern and the two uses are shown below for comparison (the highlighted parts in Figure 5.4 show the fragments over the concrete syntax, that the Recaf compiler matches to detect the pattern).

```
for    ( Customer c : List          ) { ... }

using  ( File     f : IO.open(path) ) { ... }
```

Figure 5.4: Recaf matching fragments over the concrete syntax

Recaf, after detecting the concrete syntax of the pattern, virtualizes the compilation unit at the method level by transforming the code fragment that includes the extension to the plain Java code at the bottom.

Each statement in the user code is transformed into calls on the `alg` object. The `using` construct itself is mapped to the `Using` method. The `Using` class, shown in the call-out, defines the semantics for `using`. It takes two parameters: one of type `ISupply`, a lambda that takes no parameters and supplies a value (the value on the right of the semicolon) and one function of type `Function<U, IExec>` that represents the code inside the block of `using`, as a function, parameterized by a value of type `U` (one the left of the semicolon and of type `File` in this example). It extends a class (`BaseJava`) capturing the ordinary semantics of Java, and defines a single method, also called `Using`. This particular `Using` method defines the semantics of the `using` construct as an interpreter of type `IExec`.

Recaf and StreamAlg from Chapter 4 both rely on object algebras. Recall that streams were considered a DSL and their API defined the syntactic modality of the language in a straightforward manner. The user writes pipelines in that style directly and new operators (syntactic extensions) are defined directly in an object algebra. However, for a general purpose language like Java, Recaf must communicate a new extension in a Java-like syntax to a definition in the object algebra. This is the rationale behind the design choice of relying on existing patterns as the `for`-each statement for this example.

In addition to using a `recaf` variable to specify the semantics of a `recaf` method, it is also possible to decorate a formal parameter of a method with the `recaf` modifier. This allows binding of the semantics at the call site of the method itself. Thus, Recaf supports three different binding times for the semantics of a method: static (using a static field), at object construction time (using an instance field), and late binding (method parameter).

```
List<T> method(recaf Semantics<T> alg)
```

Figure 5.5: Enabling Recaf transformation at the method level (parameter position)

Recaf further makes the distinction between statement-only virtualization and expression virtualization. In the latter case, statements are virtualized too. This mode is enabled by using the `recaff` keyword, instead of `recaf`. Section 5.3 provides all the details regarding the difference.

```
interface MuJavaMethod<R, S> {
  R Method(S s);
}
```

```
interface MuJava<R, S> {
  S Exp(Supplier<Void> e);
  S If(Supplier<Boolean> c, S s1, S s2);
  <T> S For(Supplier<Iterable<T>> e, Function<T, S> s);
  <T> S Decl(Supplier<T> e, Function<T, S> s);
  S Seq(S s1, S s2);
  S Return(Supplier<R> e);
  S Empty();
}
```

Figure 5.6: Object Algebra interfaces defining the abstract syntax of μJava method bodies and statements.

5.2 Statement Virtualization

In this section we describe the first level of semantic and syntactic polymorphism offered by Recaf, which restricts virtualization and syntax extension to statement-like constructs.

5.2.1 μJava

μJava is a simplified variant of Java used for exposition in this chapter. In μJava all variables are assumed to be final, there is no support for primitive types nor `void` methods, and all variable declarations have initializers. Figure 5.6, shows the abstract syntax of μJava statements and method bodies in the form of Object Algebra interfaces.

Both interfaces are parametric in two generic types, `R` and `S`. `R` represents the return type of the method, and `S` the semantic type of statements. The method `Method` in `MuJavaMethod` mediates between the denotation of statements (`S`) and the return type `R` of the virtualized method. The programmer of a Recaf method needs to ensure that `R` returned by `Method` corresponds to the actual return type declared in the method. Note that `R` does not have to be bound to the same concrete type in both `MuJavaMethod` and `MuJava`. This means that the return type of a virtualized method can be different than the type of expressions expected by `Return`.

The `MuJava` interface assumes that expressions are represented using the standard Java `Supplier` type, which represents thunks. Java expressions may perform arbitrary side-effects; the thunks ensure that evaluation is delayed until after the semantic objects are created.

$$
\begin{array}{rcl}
\mathcal{M}_a[\![S]\!] & = & \texttt{return}\ a\texttt{.Method(}\mathcal{S}_a[\![S]\!]\texttt{);} \\
 & & \\
\mathcal{S}_a[\![e\texttt{;}]\!] & = & a\texttt{.Exp(() } \rightarrow \texttt{ \{}e\texttt{; return null;\})} \\
\mathcal{S}_a[\![\texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2\,]\!] & = & a\texttt{.If(() } \rightarrow e\texttt{, }\mathcal{S}_a[\![S_1]\!]\texttt{,}\mathcal{S}_a[\![S_2]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{for(}T\ x\texttt{: }e\texttt{)}\ S]\!] & = & a\texttt{.For(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![T\ x\ \texttt{=}\ e\texttt{;}\ S]\!] & = & a\texttt{.Decl(() } \rightarrow e\texttt{, (}T\ x\texttt{) } \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![S_1\texttt{;}\ S_2]\!] & = & a\texttt{.Seq(}\mathcal{S}_a[\![S_1]\!]\texttt{, }\mathcal{S}_a[\![S_2]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{return}\ e\texttt{;}]\!] & = & a\texttt{.Return(() } \rightarrow e\texttt{)} \\
\mathcal{S}_a[\![\texttt{;}]\!] & = & a\texttt{.Empty()} \\
\mathcal{S}_a[\![\texttt{\{}\ S\ \texttt{\}}\,]\!] & = & \mathcal{S}_a[\![S]\!]
\end{array}
$$

Figure 5.7: Virtualizing method statements into statement algebras

```
for (Integer x: l)
  if (x % 2 == 0)
    return x;
  else
    ;
return null;
```

```
return a.Method(
  a.Seq(
    a.For(() → l, (Integer x) →
      a.If(() → x % 2 == 0,
           a.Return(() → x),
           a.Empty())),
    a.Return(() → null)));
```

Figure 5.8: Example method body (left) and its transformation into algebra `a` (right).

The constructs `For` and `Decl` employ higher-order abstract syntax (HOAS [134]) to introduce local variables. As a result, the bodies of declarations (i.e., the statements following the declaration, within the same scope) and `for`-each loops are represented as functions from some generic type `T` to the denotation `S`.

5.2.2 Transforming Statements

The transformation for μJava is shown in Figure 5.7, and consists of two transformation functions $\mathcal{M}$ and $\mathcal{S}$, respectively transforming method bodies, and statements. The transformation folds over the syntactic structure of μJava, compositionally mapping each construct to its virtualized representation. Both functions are subscripted by the expression a, which represents the actual algebra that is used to construct the semantics. The value of a is determined by the `recaf` modifier on a field or formal parameter.

As an example, consider the code shown on the left of Figure 5.8. The equivalent code after the Recaf transformation is shown on the right. The semantics of the code is now virtualized via the algebra object `a`. The algebra `a` may implement the same semantics as ordinary Java, but it can also customize or completely redefine it.

$$
\begin{array}{rcll}
S & ::= & \texttt{x!}\ e\ \texttt{;} & \text{[Return-Like]} \\
 & | & \texttt{x}\ (T\ x\texttt{:}\ e)\ S & \text{[For-Each-Like]} \\
 & | & \texttt{x}\ (e)\ \{S\} & \text{[While-Like]} \\
 & | & \texttt{x}\ \{S\} & \text{[Try-Like]} \\
 & | & \texttt{x}\ T\ x = e\texttt{;} & \text{[Declaration-Like]}
\end{array}
$$

Figure 5.9: Syntax extensions of statements (S) for μJava.

$$
\begin{array}{rcl}
\mathcal{S}_a[\![x\ \texttt{!}\ e\texttt{;}]\!] & = & a\texttt{.}x\texttt{(()} \rightarrow e\texttt{)} \\
\mathcal{S}_a[\![x\ \texttt{(}T\ y\texttt{:}\ e\texttt{)}\ S]\!] & = & a\texttt{.}x\texttt{(()} \rightarrow e\texttt{,}\ \texttt{(}T\ y\texttt{)} \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![x\ T\ y\ \texttt{=}\ e\texttt{;}\ S]\!] & = & a\texttt{.}x\texttt{(()} \rightarrow e\texttt{,}\ \texttt{(}T\ y\texttt{)} \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![x\ \texttt{(}e\texttt{)}\ S]\!] & = & a\texttt{.}x\texttt{(()} \rightarrow e\texttt{,}\ \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![x\ \texttt{\{}\ S\ \texttt{\}}]\!] & = & a\texttt{.}x\texttt{(}\mathcal{S}_a[\![S]\!]\texttt{)}
\end{array}
$$

Figure 5.10: Transforming syntax extensions to algebra method calls

5.2.3 Statement Syntax

Statement syntax is based on generalizing the existing control-flow statement syntax of Java. Informally speaking, wherever Java requires a keyword (e.g., `for`, `while` etc.), Recaf allows the use of an identifier. This identifier will then, by convention, correspond to a particular method with the same name in the semantic algebra.

The grammar presented in Figure 5.9 defines a potentially infinite family of new language constructs, by using identifiers (x) instead of keywords. Each production is a generalization of existing syntax. For instance, the first production follows the syntax of `return` e, with the difference that an exclamation mark is needed after the identifier x to avoid ambiguity. The second production is like `for`–each, the third like `while`, and the fourth follows the pattern of `if` without `else`. Finally, the last production supports custom declarations, where the first identifier x represents the keyword.

Transforming the extension into an algebra simply uses the keyword identifier `x` as a method name, but follows the same transformation rules as for the corresponding, non-extended constructs. The transformation rules are shown in Figure 5.10.

5.2.4 Direct Style Semantics

The direct style interpreter for μJava is defined as the interface `MuJavaBase`, implementing by using default methods. The type parameter `R` represents the return type of the method. `S` is bound to the interface `IExec`, which represents thunks (closures). Both are shown in Figure 5.11.

The algebra `MuJavaBase` thus maps μJava statement constructs to semantic objects of type `IExec`. Most of the statements in μJava have a straightforward implementation. Non-local control-flow (i.e., `return`), however, is implemented using exception handling.

```
interface MuJavaBase<R> extends MuJava<R, IExec> { ... }

interface IExec { void exec(); }
```

Figure 5.11: Basic interfaces for a direct style interpreter

The method `Method` ties it all together and mediates between the evaluation of the semantic objects, returned by the algebra and to the actual return type of the method (Figure 5.12). Since the mapping between the statement denotation and the actual return type of a method is configurable, it is not part of `MuJavaBase` itself. In this way, `MuJavaBase` can be reused with different `Method` implementations.

```
default R Method(IExec s) {
  try {
    s.exec();
  }
  catch (Return r) {
    return (R) r.value;
  }
  catch (Throwable e) {
    throw new RuntimeException(e);
  }
  return null;
}
```

Figure 5.12: Method-level extensibility

The full code of this interpreter is included as Appendix A.1.

Example: Maybe. As a simple example, similar to the `using` extension introduced earlier, consider a `maybe` construct (Figure 5.13), to unwrap a value of type `java.util.Optional`. In a sense, `maybe` literally overrides the semicolon, similar to the bind operator of Haskell. Syntactically, the `maybe` operator follows the declaration-like syntax.

The `Maybe` method returns an `IExec` closure that evaluates the expression, of type `Optional`, and if the optional is not empty, executes the body of `maybe`.

5.2.5 Continuation-Passing Style Semantics

The direct style base interpreter can be used for many extensions like `using` or `maybe`. However, language constructs that require non-local control-flow semantics require a

```
interface Maybe<R> extends MuJavaBase<R> {
  default <T> IExec Maybe(Supplier<Optional<T>> x, Function<T, IExec> s) {
    return () → {
       Optional<T> opt = x.get();
       if (opt.isPresent()) s.apply(opt.get()).exec();
   };
  }
}
```

Figure 5.13: Implementation of a `Maybe` extension

continuation-passing style (CPS) interpreter. This base interpreter can be used instead of the direct style interpreter for extensions like coroutines, backtracking, call/cc etc. It also shows how Object Algebras enables the definition of two different semantics for the same syntactic interface.

The CPS-style interpreter is defined as the interface `MuJavaCPS`, similarly to `MuJavaBase` from Section 5.2.4. The `MuJavaCPS` algebra maps μJava abstract syntax to CPS denotations (`SD`). `SD<R>` is a functional interface that takes as parameters a return and a success continuation. The return continuation `r` is of type `K<R>` (a consumer of `R`) and contains the callback in case a statement is the return statement. The success continuation is of type `K0` (a thunk) and contains the callback in case the execution falls off without returning. Both interfaces are shown in Figure 5.14.

```
interface MuJavaCPS<R> extends MuJava<R, SD<R>> { … }
interface SD<R> { void accept(K<R> r, K0 s); }
```

Figure 5.14: Basic interfaces for a CPS-style interpreter

To illustrate the CPS interpreter, consider the following code, in Figure 5.15, that defines the semantics of the `if-else` statement. Based on the truth value of the condition, either the then branch or the else branch is executed, with the same continuations as received by the if-then-else statement.

```
default SD<R> If(Supplier<Boolean> c, SD<R> s1, SD<R> s2) {
  return (r, s) → {
    if (c.get()) s1.accept(r, s);
    else s2.accept(r, s);
  };
}
```

Figure 5.15: `if-else` statement in the CPS-style interpreter

The full code of the μJava CPS interpreter, `MuJavaCPS`, is included in Appendix A.2.

Example: Backtracking. The CPS interpreter serves as a base implementation for language extensions, requiring complex control flow. We demonstrate the backtracking extension that uses Wadler's list of successes technique [182] and introduces the `choose` keyword. As an example, consider the method in Figure 5.16 which finds all combinations of integers out of two lists that sum to 8.

```
List<Pair> solve(recaf Backtrack<Pair> alg) {
  choose Integer x = asList(1, 2, 3);

  choose Integer y = asList(4, 5, 6);

  if (x + y == 8) {
    return new Pair(x, y);
  }
}
```

Figure 5.16: Usage of the Backtracking Extension

In Figure 5.17 we present the extension for μJava. Note that `Method` has a generic parameter type in the `MuJavaMethod` interface. In this case we change the return type of method to `List<T>` instead of just `T`. The result is the list of successes, so the return continuation should add the calculated item in the return list, instead of just passing it to the continuation. `Choose` simply invokes its success continuation for every different value, effectively replaying the execution for every element of the set of values.

5.3 Expression Virtualization

The previous section discussed virtualization of declaration and control-flow statements. We widen the scope of Recaf for virtualizing expressions as well. Instead of relying on opaque functions that will yield a value when evaluated (of the functional interface/type `Supplier` in Java), we enable expression virtualization by translating expressions into method invocations as well. The `MuExpJava` interface specifies the semantic objects for μJava expressions. While `MuJava` relied on `Supplier` function types, a new interface `MuStmJava` unlocks expression extensibility for all statements of μJava. The two interfaces are shown in Figure 5.18. To support expression virtualization, the transformation of statements and expressions follows a new set of rules included in Appendix A.3.

Consider, for example, how expression virtualization desugars the μJava code fragment, in Figure 5.19, of a `for` statement: `for (Integer x: y) println(x + 1);`.

```
interface Backtrack<R>
extends MuJavaCPS<R>, MuJavaMethod<List<R>, SD<R>> {

   default List<R> Method(SD<R> body) {
      List<R> result = new ArrayList<>();
      body.accept(ret → { result.add(ret); }, () → {});
      return result;
   }

    default <T> SD<R> Choose(Supplier<Iterable<T>> e,
                             Function<T, SD<R>> s) {
      return (r, s0) → {
         for (T t: e.get())
           s.apply(t).accept(r, s0);
      };
   }
}
```

Figure 5.17: Backtracking Extension

```
interface MuStmJava<S, E> {
  S   Exp(E x);
  <T> S Decl(E x,Function<T,S> s);
  <T> S For(E x,Function<T,S> s);
  S   If(E c, S s1, S s2);
  S   Return(E x);
  S   Seq(S s1, S s2);
  S   Empty();
}
```

```
interface MuExpJava<E> {
  E Lit(Object x);
  E This(Object x);
  E Field(E x, String f);
  E New(Class<?> c, E...es);
  E Invoke(E x, String m, E...es);
  E Lambda(Object f);
  E Var(String x, Object it);
}
```

Figure 5.18: Generic interfaces for the full abstract syntax of μJava

```
  a.For(a.Var("y", y), Integer x →
    a.Exp(a.Invoke(a.This(this), "println",
                  a.Add(a.Var("x", x), a.Lit(1)))));
```

Figure 5.19: Expression Virtualization example

5.3.1 An Interpreter for μJava Expressions

Just like statements, the base semantics of μJava expressions is represented by an interpreter, this time conforming to the interface `MuExpJava`, shown in Figure 5.18. This interpreter binds `E` to the closure type `IEval`.

```
interface IEval { Object eval(); }
```

Figure 5.20: Evaluation of expressions

As `IEval` is not polymorphic, we do not make any assumptions about the type of the returned object. The interpreter is fully dynamically typed, because Java's type system is not expressive enough to accurately represent the type of expression denotations, even if the Recaf transformation would have had access to the types of variables and method signatures.

The evaluation of expressions is straightforward. Field access, object creation (`new`), and method invocation, however are implemented using reflection. The full interpreter code, `MuExpJavaBase`, is included in Appendix A.4.

```
recaf Solve alg = new Solve();
recaff Iterable<Map<String,Integer>> example() {
  var 0, 5, IntVar x;
  var 0, 5, IntVar y;
  solve! x + y < 5;
}
```

Figure 5.21: Embedding for a constraint solver

Example: Library embedding. Figure 5.21 presents a library embedding of a simple constraint solving language, *Choco* [139], a Java library for constraint programming. Choco's programming model is heavily based on factories for nearly all of its components, from variable creation, constraint declaration over variables, to search strategies.

We have developed a Recaf embedding which translates a subset of Java expressions to the internal constraints of Choco, which can then be solved. The `Solve` algebra defines the `var` extension to declare constraint variables. The `solve!` statement posts constraints to Choco's solver. This embedding illustrates how expression virtualization allows the extension developer to completely redefine (a subset of) Java's expression syntax.

5.4 Implementation of Recaf

All Recaf syntactic support is provided by Rascal [92], a language for source code analysis and transformation. Rascal features built-in primitives for defining grammars, tree traversal and concrete syntax pattern matching. Furthermore, Rascal's language workbench features [45] allow language developers to define editor services, such as syntax highlighting or error marking, for their languages. Section 2.3.1 provides background material on Rascal.

5.4.1 Generically Extensible Syntax for Java

Section 5.2.3 introduced generic syntax extensions in the context of μJava, illustrating how the base syntax could be augmented by adding arbitrary keywords, as long as they conform to a number of patterns, e.g. `while`- or declaration-like. We implemented these patterns and a few additional ones for full Java using Rascal's declarative syntax rules.

These rules modularly extend the Java grammar, defined in Rascal's standard library, as can be seen in Figure 5.22.

The first rule defines the "keyword identifier" `KId` which corresponds to the new keywords introduced by the extensions. By explicitly creating this syntactic category in the extended grammar, we avoid potential ambiguities with the already defined keywords from the original Java grammar. The productions for `Expr` and `Stm` are labeled to indicate the (combination of) patterns that inspired the new syntax. For instance, the first `Stm` production (labeled `returnLike`), captures the return-like syntax using the exclamation mark. Grammar symbols of the form $\{S\ s\}\oplus$ (where $\oplus \in \{*, +\}$) capture lists of elements of type S separated by separator s. The post-fix operator `!` disallows specific, labeled productions from a nonterminal. For instance, the body of a `whileLike` statement cannot be the `empty` statement, because this would be ambiguous with statements consisting of a method call. Note that the productions of Figure 5.22 seem ambiguous with the ordinary Java statement syntax. This is not the case however, because the base grammar explicitly excludes keywords like `while`, `for` etc. from the `Id` nonterminal.

5.4.2 Transforming Methods

Recaf's source code transformation, transforms any method that has the `recaf` or `recaff` modifier. If the modifier is attached to the method declaration, the algebra is expected to

```
syntax KId = @category="MetaKeyword" Id;

syntax Expr
  = methodLike: "#" KId "(" {Expr ","}* ")"
  | closureLike: "#" KId Block;

syntax Stm
  = returnLike: KId "!" Expr ";"
  | returnLike2: KId "!" ";"
  | forLike: KId "(" FormalParam ":" Expr ")" Stm
  | forMany: KId "(" {FormalParam ","}+ ")" Stm
  | whileLike: KId "(" {Expr ","}+ ")" Stm!empty
  | whileFor: KId "(" {Expr ","}+ "," FormalParam ":" Expr ")" Stm
  | switchLike: KId "(" {Expr ","}+ ")" "{" Item+ "}"
  | switchFor: KId "(" FormalParam ":" Expr ")" "{" Item+ "}"
  | tryLike: KId Stm!exprStm!empty
  | trySwitch: KId "{" Item+ "}"
  | declLike: KId FormalParam ";"
  | declLikeInit: KId FormalParam "=" Expr ";"
  | declFor: KId FormalParam "=" "(" Expr ")" Stm!empty
  | declWhile: KId {Expr ","}+ "," FormalParam ";"
  | declWhile2: KId {Expr ","}+ "," FormalParam "=" Expr ";"
  | declReturn: FormalParam "=" KId "!" Expr ";";

syntax Item
  = caseLike: KId {Expr ","}+ ":" BlockStm+
  | caseFor: KId FormalParam ":" BlockStm+;
```

Figure 5.22: Full Java 8 grammar extension for Recaf in Rascal.

be declared as a field in the enclosing scope (class or interface). If the modifier is attached to a method's formal parameter, that parameter itself is used instead. Furthermore, if the `recaff` modifier is used, expressions are transformed as well.

The transformation is defined as Rascal rewrite rules that match on a Java statement or expression using concrete-syntax pattern matching. This means that the matching occurs on the concrete syntax tree directly, having the advantage of preserving comments and indentation from the Recaf source file. As an example, the rule in Figure 5.23 defines the transformation of the `while`-statement.

```
Expr stm2alg((Stm)`while (<Expr c>) <Stm s>`,Id a,Names ns)
  = (Expr)`<Id a>.While(<Expr c2>, <Expr s2>)`
  when
    Expr c2 := injectExpr(c, a, ns),
    Expr s2 := stm2alg(s, a, ns);
```

Figure 5.23: Transformation of a statement for `while`

This rewrite rule uses the actual syntax of the Java `while` statement as the matching pattern and returns an expression that calls the `While` method on the algebra `a`. The condition `c` and the body `s` are transformed in the `when`-clause (where `:=` indicates binding through matching). The function `injectExpr` either transforms the expression `c`, in the case of transformations annotated with the `recaff` keyword, or creates closures of type `Supplier` otherwise. The body `s` is transformed by calling `stm2alg` recursively.

The `ns` parameter represents the declared names that are in scope at this point in the code and is the result of a local name analysis needed to correctly handle mutable variables. Local variables introduced by declarations and for-loops are mutable variables in Java, unless they are explicitly declared as `final`. This poses a problem for the HOAS encoding we use for binders: the local variables become parameters of closures, but if these parameters are captured inside another closure, they have to be (effectively) final. To correctly deal with this situation, variables introduced by declarations or for-loops are wrapped in explicit reference objects, and the name is added to the `Names` set. Whenever such a variable is referenced in an expression, it is unwrapped. For extensions that introduce variables it is unknown whether they should be mutable or not, so the transformation assumes they are final, as a default. In total, the complete Recaf transformation consists of 790 SLOC.

5.4.3 IDE Support

Recaf is supported by custom editor services through Rascal's hooks into the Eclipse IDE[4]. First, Recaf editors feature syntax highlighting, which is automatically derived from

[4]Appendix A.5 includes a screenshot.

the Recaf grammar definition. For instance, the `@category` annotation in the grammar definition of Figure 5.22 informs the IDE that the `KId` identifiers should be highlighted as keywords.

Recaf editors also support error marking. This feature is based on a bridge to the Eclipse JDT [11]. After the Recaf code is transformed, it is type checked using JDT. In the case of errors, their associated locations are mapped back to the locations in the original Recaf code. The key mechanism behind this technology is origin tracking on parse trees when performing source-to-source transformations [74].

5.4.4 Recaf Runtime

The Recaf runtime library comes with two base interpreters of Java statements, similar to `MuJavaBase` and `MuJavaCPS`, and an interpreter for Java Expressions. In addition to `return`, the interpreters support the full non-local control-flow features of Java, including (labeled) `break`, `continue` and `throw`. The CPS interpreter represents each of those as explicit continuations in the statement denotation (`SD`), whereas the direct style interpreter uses exceptions.

The main difference between `MuExpBase` and the full expression interpreter is in the handling of assignments. We model mutable variables by the interface `IRef`, which defines a setter and getter to update the value of the single field that it contains. The `IRef` interface is implemented once for local variables, and once for fields. The latter uses reflection to update the actual object when the setter is called. In addition to the `Var(String,Object)` constructor, the full interpreter features the constructor `Ref(String,IRef<?>)` to model mutable variables. The expression transformation uses the local name analysis (see above) to determine whether to insert `Var` or `Ref` calls.

Since the Recaf transformation is syntax-driven, some Java expressions are not supported. For instance, since the expression interpreter uses reflection to call methods, statically overloaded methods are currently unsupported (because it is only possible to dispatch on the runtime type of arguments). Another limitation is that Recaf does not support static method calls, fields references or package qualified names. These three kinds of references all use the same dot-notation syntax as ordinary method calls and field references. However, the transformation cannot distinguish these different kinds, and interprets any dot-notation as field access or method invocation with an explicit receiver. We consider a type-driven transformation for Recaf as an important direction for future work.

5.5 Case Studies

We evaluate the expressiveness of Recaf with three case studies. The first provides language support for generators and asynchronous computations, the second develops a minimal implementation of pull-based streams using the generator extension from the first case study and the third develops a DSL for parser combinators.

5.5.1 Spicing up Java with Side-Effects

The Dart programming language recently introduced `sync*`, `async` and `async*` methods, to define generators, asynchronous computations and asynchronous streams [116] without the typical stateful boilerplate or inversion of control flow. Using Recaf, we have implemented these three language features for Java, based on the CPS interpreter, closely following the semantics presented in [116]. In the following section 5.5.2 we will develop a pull-based stream library based on the generator syntax.

Generators. The extension for generators is defined in the `Iter` class. The `Iter` class defines `Method` to return a plain Java `Iterable<R>`. When the `iterator()` is requested, the statement denotations start executing. The actual implementation of the iterator is defined in the client code using two new constructs. The first is `yield!`, which produces a single value in the iterator. Its signature is `SD<R> Yield(ISupply<R>)`.[5] Internally, `yield!` throws a special `Yield` exception to communicate the yielded element to a main iterator effectively pausing the generator. The `Yield` exception contains both the element, as well as the current continuation, which is stored in the iterator. When the next value of the iterator is requested, the saved continuation is invoked to resume the generator process. The second construct is `yieldFrom!` which flattens another iterable into the current one. Its signature is `SD<R> YieldFrom(ISupply<Iterable<R>> x)` and it is implemented by calling `ForEach(x, e → Yield(() -> e))`. In the code snippet below, we present a recursive implementation of a `range` operator, using both `yield!` and `yieldFrom!`:

```
recaf Iterable<Integer> range(int s, int n) {
  if (n > 0) {
    yield! s;
    yieldFrom! range(s + 1, n — 1);
  }
}
```

Figure 5.24: Dart's `yield` and `yieldFrom` in Java

Async. The implementation of `async` methods also defines `Method`, this time returning a `Future<R>` object. The only syntactic extension is the `await` statement. Its signature is `<T> Await(Supplier<CompletableFuture<T>>, Function<T, SD<R>>)`, following the syntactic template of for-each. The `await` statement blocks until the argument future completes. If the future completes normally, the argument block is executed with the value returned from the future. If there is an exception, the exception continuation is invoked instead. `Await` literally passes the success continuation to the future's `whenComplete` method. The Async extension supports programming with asynchronous computations

[5] `ISupply` is a thunk which has a `throws` clause.

without having to resort to call-backs. For instance, the following method computes the string length of a web page, asynchronously fetched from the web.

```
recaf Future<Integer> task(String url)
  await String html = fetchAsync(url);
  return html.length();
}
```

Figure 5.25: Dart's `async` in Java

Async*. Asynchronous streams (or reactive streams) support a straightforward programming style on observables, as popularized by the Reactive Extensions [111, 113] framework. The syntax extensions to support this style are similar to `yield!` and `yieldFrom!` constructs for defining generators. Unlike the `yield!` for generators, however, `yield!` now produces a new element asynchronously. Similarly, the `yieldFrom!` statement is used to splice on asynchronous stream into another. Its signature reflects this by accepting an `Observable` object (defined by the Java variant of Reactive Extensions, RxJava[6]): `SD<R> YieldFrom(ISupply<Observable<R>>)`. Reactive streams offer one more construct, `awaitFor!`, which is similar to the ordinary for-each loop. Whereas the semantics of a for-each loop describe synchronous iteration, the new extension "iterates" asynchronously over a stream of observable events with the signature of Figure 5.26.

Whenever a new element becomes available on the stream, the body of the `awaitFor!` is executed again. An `async*` method will return an `Observable`. Figure 5.27 shows a simple method that prints out intermediate results arriving asynchronously on a stream. After the result is printed, the original value is yielded, in a fully reactive fashion.

```
<T> SD<R> AwaitFor(ISupply<Observable<T>>, Function<T, SD<R>>)}
```

Figure 5.26: `await for` implementation in Recaf

```
recaf <X> Observable<X> print(Observable<X> src) {
  awaitFor (X x: src) {
    System.out.println(x);
    yield! x;
  }
}
```

Figure 5.27: Dart's `await for` in Java

[6] `https://github.com/ReactiveX/RxJava`

5.5.2 Streams (Pull-based)

To demonstrate the use of extensions in libraries, in this section we present a prototypical implementation of a stream library The implementation follows the design of pull-based streams as in C#, F# and Scala 3 and examined more closely as a semantic object in Chapter 4. In this section we adopt the C#-way that makes use of the `yield` statement. The implementation of the yield extension comes from the 5.5.1 case study above. In the current case study we model that kind of execution model with exceptions (for exposition), so semantics are reasonably sufficient to expose the idea of code reusability of user-defined language extensions. The snippet below shows a simple usage that implements a pipeline of mapping, filtering and summing a range of ten integers starting from 0.

```
public static void main(String args[]) {
  Integer result = PStream
    .range(10)
    .map(i → i ^ 2)
   .sum();
}
```

Figure 5.28: Pipeline using `PStream` (implemented with Recaf)

The name of our library is `PStream` and consists of three operators: a producer `range`, a transformer `map` and a terminal `sum`. The first one creates a range of values (wrapping Java 8's `IntStream.range`) and `map` is implemented using iterator blocks with `yield` as in C#. These two methods are annotated with the `recaf` keyword, as needed by our framework, and their implementation can take advantage of the `yield!` keyword. Note that, in the case of mapping, the yielded value is applied to the mapping function. In the filtering case, a predicate accepts the value and if satisfied is yielded. Figure 5.29 presents the implementation of the library and the translation of Recaf is presented in the Appendix A.6.

The `PStream` library is compiled with Recaf and the Java stock compiler, in that order. The resulting code can be packaged, distributed and imported as a regular Java library.

5.5.3 Parsing Expression Grammars (PEGs)

To demonstrate language embedding and aspect-oriented language customization we have defined a DSL for Parsing Expression Grammars (PEGs) [51]. The abstract syntax of this language is shown in Figure 5.30. The `lit!` construct parses an atomic string, and ignores the result. `let` is used to bind intermediate parsing results. For terminal symbols, the `regexp` construct can be used. The language overloads the standard sequencing and `return` constructs of Java to encode sequential composition, and the result of a parsing process. The constructs `choice`, `opt`, `star`, and `plus` correspond to the usual regular

```
public class PStream {
  public PStream(Iterable<Integer> source) {
      this.source = source;
  }
  Iterable<Integer> source;

  private recaf Iter<Integer> alg = new Iter<Integer>();

  public static PStream range(int n) {
      return new PStream(rangeIter(n));
  }
  private static Iterable<Integer> rangeIter(int n) {
   return new Iterable<Integer>() {
         public Iterator<Integer> iterator() {
             return IntStream.range(0, n).iterator();
      }
    };
  }

  public PStream map(Function<Integer, Integer> f) {
    return new PStream(mapIter(this.source, f));
  }
  recaf Iterable<Integer> mapIter(
       Iterable<Integer> source,
       Function<Integer, Integer> f) {
    for (Integer t: source) {
       yield! f.apply(t);
    }
  }

  public Integer sum() {
      return sumIter(this.source);
  }
  private Integer sumIter(Iterable<Integer> source) {
    Integer acc = 0;
    for (Integer t: source) {
      acc+=t;
    }
    return acc;
  }
}
```

Figure 5.29: Pull-based stream implementation using semi-coroutines

EBNF operators. The `choice` combinator accepts a list of alternatives (`alt`). The Kleene operators bind a variable x to the result of parsing the argument statement S, where the provided expression e represents the result if parsing of S fails.

$$
\begin{array}{rcll}
S & ::= & \texttt{lit!}\ e\texttt{;} & [\textsc{Literals}] \\
 & | & \texttt{let}\ T\ x\ \texttt{=}\ e\ \texttt{;} & [\textsc{Binding}] \\
 & | & \texttt{regexp String}\ x\ \texttt{=}\ e\ \texttt{;} & [\textsc{Terminals}] \\
 & | & S\ \texttt{;}\ S & [\textsc{Sequence}] \\
 & | & \texttt{return}\ e\ \texttt{;} & [\textsc{Result}] \\
 & | & \texttt{choice\{}\ C+\texttt{\}} & [\textsc{Alternative}] \\
 & | & \texttt{opt}\ T\ x\ \texttt{=}\ \texttt{(}e\texttt{)}\ S & [\textsc{Zero or one}] \\
 & | & \texttt{star}\ T\ x\ \texttt{=}\ \texttt{(}e\texttt{)}\ S & [\textsc{One or more}] \\
 & | & \texttt{plus}\ T\ x\ \texttt{=}\ \texttt{(}e\texttt{)}\ S & [\textsc{Zero or one}] \\
C & ::= & \texttt{alt}\ l\texttt{:}\ S+ & [\textsc{Alternative}](\texttt{l=}\ label)
\end{array}
$$

Figure 5.30: Abstract syntax of embedded PEGs.

```
recaf Parser<Exp> primary() {
  choice {
    alt "value":
      regexp String n = "[0-9]+";
      return new Int(n);
    alt "bracket":
      lit! "("; let Exp e = addSub(); lit! ")";
      return e;
  }
}
```

Figure 5.31: Parsing primaries using Recaf PEGs.

The PEG language can be used by considering methods as nonterminals. A PEG method returns a parser object which returns a certain semantic value type. A simple example of parsing primary expressions is shown in Figure 5.31. The method `primary` returns an object of type `Parser` which produces an expression `Exp`. Primaries have two alternatives: constant values and other expressions enclosed in parentheses. In the first branch of the `choice` operator, the `regexp` construct attempts to parse a numeric value, the result of which, if successful, is used in the `return` statement, returning an `Int` object representing the number. The second branch first parses an open parenthesis, then binds the result of parsing an additive expression (implemented in a method called `addSub`) to `e`, and finally parses the closing parenthesis. When all three parses are successful, the `e` expression is returned. Note that the `return` statements return expressions, but the result of the method is a parser object.

Standard PEGs do not support left-recursive productions, so nested expressions are typically implemented using loops. For instance, an additive expression could be defined as

```
recaf Parser<Exp> addSub() {
  let Exp l = mulDiv();
  star Exp e = (l) {
    regexp String o = "[+\\−]";
    let Exp r = mulDiv();
    return new Bin(o, e, r);
  }
  return e;
}
```

Figure 5.32: Methods as nonterminals.

`addSub ::= mulDiv (("+"|"-") mulDiv)*`. The `addSub` method includes the definition of this grammar using the PEG embedding (Figure 5.32).

The first statement parses a multiplicative expression. The `star` construct creates zero or more binary expressions, from the operator (`o`), the left-hand side (`e`) and the right-hand side (`r`). If the body of the `star` fails to recognize a $+$ or $-$ sign, the `e` will be bound to the initial seed value `l`. The constructed binary expression will be fed back into the loop as `e` through every subsequent iteration.

The (partial) PEG for expressions shown in Figure 5.31 and in Figure 5.32, support any kind of whitespace between elements of an expression. Changing the PEG definitions manually to parse intermediate layout, however, would be very tedious and error-prone. Exploiting the Object Algebra-based architecture, we add the layout handling as a modular aspect, by extending the PEG algebra and overriding the methods that construct the parsers.

For instance, to insert layout between sequences, the PEG subclass for layout overrides the `Seq` as follows:

```
<T, U> Parser<U> Seq(Parser<T> p1, Parser<U> p2) {
   return PEG.super.Seq(p1, PEG.super.Seq(layout, p2));
}
```

Figure 5.33: Layout between sequences

Another concern with standard PEGs is exponential worst-case runtime performance. The solution is to implement PEGs as packrat parsers [50], which run in linear time by memoizing intermediate parsing results. Again, the base PEG language can be modularly instrumented to turn the returned parsers into memoizing parsers.

5.6 Discussion

Static Type Safety The Recaf source-to-source transformation assumes certain type signatures on the algebras that define the semantics. For instance, the transformation of binding constructs (declarations, for-each, etc.) expects `Function` types in certain positions of the factory methods. If a method of a certain signature is not present on the algebra, the developer of a Recaf method will get a static error during compilation of the generated code.

The architecture based on Object Algebras provides type-safe, modular extensibility of algebras. Thus, the developer of semantics may enjoy full type-safety in the development of extensions. The method signatures of most of the examples and case-studies accurately describe the expected types and do not require any casts.

On the other hand, the statement evaluators represent expressions as opaque closures, which are typed in the expected result such as `Supplier<Boolean>` for the `if-else` statement. At the expression level, however, safety guarantees depend on the denotation types themselves. More general semantics, as in the Java base expression interpreter, however, are defined in terms of closures returning `Object`. The reason is that Java's type system is not expressive enough to represent them otherwise (lacking features such as higher-kinded types and implicits). As a result, potentially malformed expressions are not detected at compile-time.

Another consequence of this limitation is that the `Supply`-based statement interpreters described in Section 5.2 cannot be combined out-of-the-box with expression interpreters in the context of Expression Virtualization, as both interpreters must be defined in terms of generic expressions. Fortunately, the `Supply`-based statement interpreters can be reused by applying the Adapter pattern [55]. In the runtime library, we provide an adapter that maps a `Supplier`-based algebra to one that is generic in the expression type. As we have discussed earlier, this is unsafe by definition. Thus, although we can effectively integrate statement and expression interpreters, we lose static type guarantees for the expressions.

To conclude, Recaf programs are type-correct when using Statement Virtualization, as long as they generate type-correct Java code. However, in the context of expression virtualization, compile-time guarantees are overridden as the expressions are fully generic, and, therefore, no static assumptions on the expressions can be made.

Runtime Performance. Runtime performance depends on the implementation of the semantics. The base interpreters are admittedly naive, but for the purpose of Recaf they illustrate the modularity and reusability enabled by Recaf for building language extensions on top of Java. The Dart-like extensions reuse the CPS interpreter. As such they are too slow for production use. Closure creation to represent the program increases heap allocations. But these examples illustrate the expressiveness of Recaf's embedding method: a very regular syntactic interface (the algebra) may be implemented by an interpreter that

completely redefines control-flow evaluation. On the other hand, the constraint embedding example only uses the restricted method syntax to build up constraint objects for a solver. Solving the constraints does not incur any additional overhead. The DSL is used merely for construction of the constraint objects.

5.7 Summary of Recaf

This chapter presented Recaf, a lightweight tool to extend both the syntax and the semantics of Java methods just by writing Java code. Recaf is based on two techniques. First, the Java syntax is generalized to allow custom language constructs that follow the pattern of the regular control-flow statements of Java. Second, a generic source-to-source transformation translates the source code of methods into calls to factory objects that represent the desired semantics. Furthermore, formulating these semantic factories as Object Algebras enables powerful patterns for composing semantic definitions and language extensions.

In the next chapter, we depart from the extensibility of either domain-specific or general-purpose programming languages and we concentrate solely on the performance aspect of streams. We study sequential streams for data that fit into memory (no parallel or distributed context, yet). Our goal is to design a stream library, again with familiar API (not in the style of object algebras), but with the difference that complex pipelines will perform the same as hand-written, loop-based ones. Additionally, we offer guarantees that no lists, buffers or other variable-size data structures are created. Last but not least, all optimization choices we make are tightly coupled with the library itself, without altering the back-end of the compiler. We avoid pulling domain knowledge in the general-purpose compiler, staying completely at the library level.

6. STREAM FUSION, TO COMPLETENESS

Functional streaming libraries let us easily build various pipelines, by composing sequences of simple transformers, such as `map` or `filter`, with producers (backed by an array, a file, or a generating function) and consumers (reducers). The purely applicative approach of building a complex pipeline from simple immutable pieces simplifies programming and reasoning: the assembled pipeline is an executable specification. To be practical, however, a library has to be efficient: at the very least, it should avoid creating intermediate structures (files, lists, etc.) whose size grows with the length of the stream.

Most modern programming languages—Java, Scala, C#, F#, OCaml, Haskell, Clojure, to name a few—currently offer functional stream libraries. They all provide basic mapping and filtering. Handling of infinite, nested or parallel (zipping) streams is rare—especially all in the same library. Although all mature libraries avoid unbounded intermediate structures, they all suffer, in various degrees, from the overhead of abstraction and compositionality: extra function calls, the creation of closures, objects and other bounded intermediate structures.

An excellent example is the Java 8 Streams, often taken as the standard of stream libraries. Java 8 Streams stress performance: e.g., streaming from a known source, such as an array, amounts to an ordinary loop, well-optimized by a Java JIT compiler (Chapter 3). However, Java 8 Streams are still much slower than hand-optimized loops for non-trivial pipelines (e.g., over 10x slower on the standard cartesian product benchmark (Chapter 3). Furthermore, the library cannot handle ('zip') several streams in parallel[1] and cannot deal with nesting of infinite streams (Chapter 4). These are not mere omissions: infinite nested streams demand a different iteration model, which is hard to efficiently implement with a simple loop.

In the rest of this dissertation we will investigate the structure of streams and propose a streaming library design that offers both high expressiveness and *guaranteed*, high performance. Before we proceed with the new design, the following section presents an experiment. We port the essential form of the design from the fastest streaming library we have seen so far—the Java 8 Stream API—in a programming language with the fastest, whole-program optimizing compiler.

6.1 MLton: a first experiment to achieve raw performance

MLton [188], the heavy artillery of whole-program optimizing compilers, targets Standard ML. Its utmost goal is raw performance (execution speed and memory footprint) of ML programs. MLton achieves this goal by eliminating intricacies of ML programs, highly higher-order in nature, resulting in very fast native code. With whole-program optimization, MLton reconstructs a very accurate control-flow representation of the program, without letting

[1] One could emulate `zip` using `iterator` from push-streams—at significant drop in performance.

higher-orderness, functors and polymorphic types get in the way. In short, MLton performs a $0CFA$-whole program, control flow analysis after it defunctorizes and monomorphizes the program, resulting in a simply-typed higher-order representation (IL). Using that representation MLton performs a series of optimizations passes over the aforementioned IL, performing 22 SSA–to–SSA transformations in an effort to eliminate call overhead by aggressively inlining both direct- and continuation-based calls.

In `sml-streams`[2] we investigate the simplest form of the Java 8 Stream API design in ML (the push-based design described in the Background Section 2.1.6 and the Push Factory we described in Section 4.2.3) using MLton.

The goal of this library is to quickly evaluate whether MLton is able to fuse the operators, producing efficient, loop-based code without any special input from us. The benchmark set we used is the same one we used in Chapter 3 and its sole purpose was to help us investigate the inner workings of MLton through the lens of streams (full, formal benchmarking is not the purpose of this experiment). This design is in CPS and the main loop, per the push-based technique, is explicitely coded in `ofArray`. Hence, our hypothesis was that MLton can aggressively inline our library.

After conducting the experiment, we observed that consecutive uses of `map`s and `filter`s were fused in the generated code. However, the cartesian product did not exhibit nested loop fusion:

```
v1 ▷ Stream.flatMap(fn x => v2 ▷ Stream.map (fn y => x * y))
   ▷ Stream.sum
```

Figure 6.1: Cartesian product, in Standard ML

After closely examining the SSA control–flow graph[3] we observe that there is just one looping call but its execution follows two paths, depending on two cases. The first case concerns `flatMap` iteration and the second one concerns `map`, using the defunctionalized data structure containing the code and captured variables of the execution (`v2` and `x` respectively).

[2] `sml-streams` (the code, available at `https://github.com/biboudis/sml-streams`) is a port of the early design of `Nessos Streams` led by Nick Palladinos–`https://web.archive.org/web/20170327003549/https://nessos.github.io/Streams/`.
We would also like to thank Henry Cejtin for the very fruitful discussions we had about the elegant incorporation of short-circuiting operators in `sml-streams`

[3] A recent quote by Matthew Fluet whom we thank, on MLton-User, provided useful insights about the compilation process (and the CFG representation) in that particular example of the `cart` execution: “While we wait for the compiler to learn that optimization, we can verify that things would be better if we didn’t conflate the driving of the ”Stream.map” on the v2 and the ”Stream.flatMap” on the v1. To do so, make a code-clone of ”Stream.ofArray” and use one for v1 and the other for v2”—`https://sourceforge.net/p/mlton/mailman/message/33030638/` .

6.2 Introducing the Strymonas library

We next present Strymonas: a streaming library design that offers both high expressiveness and *guaranteed*, highest performance. First, we support the full range of streaming operators (a.k.a. stream *transformers*, *combinators*) from past libraries: not just `map` and `filter` but also sub-ranging (`take`), nesting (`flat_map`—a.k.a. `concatMap`) and parallel (`zip_with`) stream processing. All operators are freely composable: e.g., `zip_with` and `flat_map` can be used together, repeatedly, with finite or infinite streams. Our novel stream representation captures the essence of stream processing for virtually all operators examined in past literature.

Second, our stream representation allows eliminating the abstraction overhead altogether, for the full set of stream operators. We perform *stream fusion* (Section 6.4) and other aggressive optimizations. The generated code contains no extra heap allocations in the main loop (Thm.1). By not generating tuples or other objects, we avoid the overhead of dynamic object construction and pattern-matching, and also the hidden, often significant overhead of memory pressure and boxing of primitive types as in Java 8 (using the generic types and not the hand-specialized) and in Scala. The result not merely approaches but attains the performance of hand-optimized code, from the simplest to the most complex cases, up to *well over* the complexity point where hand-written code becomes infeasible. Although the library operators are purely functional and freely composable, the actual running stream code is loop-based, highly tangled and imperative.

Our technique relies on staging (Section 2.3.2), a form of metaprogramming, to achieve guaranteed stream fusion. This is in contrast to past use of source-to-source transformations of functional languages [86], of AST run-time rewriting [118, 128], compile-time macros [137] or Haskell GHC Rules [38, 133] to express domain-specific streaming optimizations.

Rather than relying on an optimizer to eliminate artifacts of stream composition, we do not introduce the artifacts in the first place. Our library transforms highly abstract stream pipelines to code fragments that use the most suitable imperative features of the host language. The appeal of staging is its certainty and guarantees. Unlike the aforementioned techniques, staging also ensures that the generated code is well-typed and well-scoped, by construction. We discuss the trade-offs of staging in Section 6.9.

Our work describes a general approach, and not just a single library design. To demonstrate the generality of the principles, we implemented two library versions,[4] in diverse settings. The first is an OCaml library, staged with BER MetaOCaml [90]. The second is a Scala library (also usable by client code in Java and other JVM languages), staged with Lightweight Modular Staging (LMS) [145].

We evaluate Strymonas on a suite of benchmarks (Section 6.8), comparing with hand-written code as well as with other stream libraries (including Java 8 Streams). Our staged implementation is up to more than two orders-of-magnitude faster than standard Java/S-

[4] `https://github.com/strymonas/`.

cala/OCaml stream libraries, matching the performance of hand-optimized loops. (Indeed, we occasionally had to improve hand-written baseline code, because it was slower than the library.)

Thus, the contributions of this chapter are: (i) the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit the full elimination of abstraction overhead. The main principle is a novel representation of streams that captures rate properties of stream transformers and the form of termination conditions, while separating and abstracting components of the entire stream state. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration. (ii) The implementation of the design in terms of two distinct library versions for different languages and staging methods: OCaml/MetaOCaml and Scala/JVM/LMS.

6.3 Overview: A Taste of the Library

We first give an overview of our approach, presenting the client code (i.e., how the library is used) alongside the generated code (i.e., what our approach achieves). Although we have implemented two separate library versions, one for OCaml and one for Scala/JVM languages, for simplicity, all examples in this chapter will be in (Meta)OCaml, which was also our original implementation.

For the sake of exposition, we take a few liberties with the OCaml notation, simplifying the syntax of the universal and existential quantification and of sum data types with record components. (The latter simplification—inline records—is supported in the latest, 4.03, version of OCaml.) This chapter is accompanied by the complete code for the Strymonas library, also including our examples, tests, and benchmarks.

MetaOCaml is a dialect of OCaml with staging annotations $.\langle \mathtt{e} \rangle.$ and $\sim \mathtt{e}$, and the `code` type [90, 169]. In the Scala version of our library, staging annotations are implicit: they are determined by inferred types. Staging annotations are optimization directives, guiding the partial evaluation of library expressions. Thus, staging annotations are not crucial to understanding what our library can express, only how it is optimized. On first read, staging annotations may be simply disregarded. We get back to them, in detail, in Section 2.3.2.

The (Meta)OCaml library interface is given in Figure 6.2. The library includes stream producers (one generic—`unfold`, and one specifically for arrays—`of_arr`), the generic stream consumer (or stream reducer) `fold`, and a number of stream transformers. Ignoring `code` annotations, the signatures are standard. For instance, the generic `unfold` operator takes a function from a state, `'z`, to a value `'a` and a new state (or nothing at all), and, given an initial state `'z`, produces an opaque stream of `'a`s.

The first example is summing the squares of elements of an array `arr`—in mathematical notation, $\sum a_i^2$. We have used this example as a running throughout this dissertation. Figure 6.3 presents this example using Strymonas and is not far from the mathematical notation. Here, $\triangleright$, like the similar operator in F#, is the inverse function application: argument

```
Stream representation (abstract)
type 'a stream

Producers
val of_arr   : 'a array code → 'a stream
val unfold   : ('z code → ('a * 'z) option code) → 'z code → 'a stream

Consumer
val fold     : ('z code → 'a code → 'z code) → 'z code → 'a stream → 'z code

Transformers
val map      : ('a code   →   'b code) → 'a stream → 'b stream
val filter   : ('a code   → bool code) → 'a stream → 'a stream
val take     : int code   → 'a stream  → 'a stream
val flat_map : ('a code   → 'b stream) → 'a stream → 'b stream
val zip_with : ('a code   →   'b code  → 'c code)  →
               ('a stream → 'b stream  → 'c stream)
```

Figure 6.2: The library interface

to the left, function to the right. The stream components are first-class and hence may be passed around, bound to identifiers and shared; in short, we can build libraries of more complex components. Strymonas generates the code in Figure 6.4 and the highlighted parts identify all the user-supplied values from the original pipeline in Figure 6.3:

```
let sum = fold (fun z a → .<.~a + .~z>.) .<0>.

of_arr .⟨arr⟩.
  ▷ map (fun x → .<.~x * .~x>. )
  ▷ sum
```

Figure 6.3: Sum of squares, in Strymonas

```
let s_1 = ref 0 in
let arr_2 = arr in
 for i_3 = 0 to Array.length arr_2 −1 do
   let el_4 = arr_2.(i_3) in
   let t_5 = el_4 * el_4 in
   s_1 := t_5 + !s_1
 done;
!s_1
```

Figure 6.4: Generated sum of squares by Strymonas

It is relatively easy to see which part of the code came from which part of the pipeline

“specification”. The generated code has no closures, tuples or other heap-allocated structures: it looks as if it were hand-written by a competent OCaml programmer. The iteration is driven by the source operator, `of_arr`, of the pipeline. This is precisely the iteration pattern that Java 8 streams optimize. As we will see in later examples, this is but one of the optimal iteration patterns arising in stream pipelines.

The next example sums only some elements:

```
let ex = of_arr .⟨arr⟩.
   ▷ map (fun x → .⟨~x * ~x⟩.)

ex ▷ filter (fun x → .⟨~x mod 17 > 7⟩.)
   ▷ sum
```

Figure 6.5: Sum of squares filtered, in Strymonas

We have abstracted out the mapped stream as `ex`. The earlier example is, hence, `ex ▷ sum`. The current example applies `ex` to the more complex summator that first filters out elements before summing the rest. The next example limits the number of summed elements to a user-specified value `n`:

```
ex ▷ filter (fun x → .⟨~x mod 17 > 7⟩.)
   ▷ take .⟨n⟩.
   ▷ sum
```

Figure 6.6: Sum of squares filtered and short-ranged, in Strymonas

We stress that the limit is applied to the filtered stream, not to the original input; writing this example in mathematical notation would be cumbersome.

The generated code (Figure 6.7) again looks as if it were handwritten, by a competent programmer. However, compared to the first example, the code is more tangled; for example, the `take .⟨n⟩.` part of the pipeline contributes to three separate places in the code: where the `nr_4` reference cell is created, tested and mutated. The iteration pattern is more complex. Instead of a `for` loop there is a `while`, whose termination conditions come from two different pipeline operators: `take` and `of_arr`.

```
let s_1 = ref 0 in
let arr_2 = arr in
let i_3 = ref 0 in
let nr_4 = ref n in
while !nr_4 > 0 && !i_3 ≤ Array.length arr_2 −1 do
   let el_5 = arr_2.(!i_3) in
   let t_6 = el_5 * el_5 in
   incr i_3;
   if t_6 mod 17 > 7
   then (decr nr_4; s_1 := t_6+ !s_1)
done; ! s_1
```

Figure 6.7: Generated short-ranged example by Strymonas

The dot-product of two arrays `arr1` and `arr2` looks just as simple:

```
zip_with (fun e1 e2 → .⟨∼e1 * ∼e2⟩.)
         (of_arr .⟨arr1⟩.)
         (of_arr .⟨arr2⟩.) ▷ sum
```

Figure 6.8: Dot-product, in Strymonas

It shows off the zipping of two streams, with the straightforward, again hand-written quality, generated code:

```
let s_17 = ref 0 in
let arr_18 = arr1 in
let arr_19 = arr2 in
 for i_20 = 0 to min (Array.length arr_18 −1) (Array.length arr_19 −1) do
   let el_21 = arr_18.(i_20) in
   let el_22 = arr_19.(i_20) in
   s_17 := el_21 * el_22 + !s_17
 done; ! s_17
```

Figure 6.9: Generated dot-product example, in Strymonas

The optimal iteration pattern is different still (though simple): the loop condition as well as the loop body are equally influenced by two `of_arr` operators.

In the final, complex example (Figure 6.10) we zip two complicated streams. The first is a finite stream from an array, mapped, subranged, filtered and mapped again. The second

is an infinite stream of natural numbers from 1, with a filtered flattened nested substream. After zipping, we fold everything into a list of tuples.

We did not show any types, but they exist (and have been inferred). Therefore, an attempt to use an invalid operation on stream elements (like concatenating integers or applying an ill-fitting stream component) will be immediately rejected by the type-checker.

Although this pipeline is purely functional, modular and rather compact, the generated code (shown in Appendix B.1) is large, entangled and highly imperative. Writing such code correctly by hand is clearly challenging.

```
zip_with
 (* zipping function *)
 (fun e1 e2 → .⟨(∼e1,∼e2)⟩.)

 (* 1st stream *)
 (of_arr .⟨arr1⟩.
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ take .⟨12⟩.
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨∼x * ∼x⟩.))

 (* 2nd stream *)
 (iota .⟨1⟩.
   ▷ flat_map (fun x → iota .⟨∼x+ 1⟩. ▷ take .⟨3⟩.)
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.))

▷ fold (fun z a → .⟨∼a :: ∼z⟩.) .⟨[]⟩.
```

Figure 6.10: Complex pipeline

6.4 Stream Fusion Problem

The key to an expressive and performant stream library is a representation of streams that fully captures the generality of streaming pipelines and allows desired optimizations. To understand how the representation affects implementation and optimization choices, we review past approaches. We revisit the two styles of stream libraries for on-demand processing (push and pull from Section 2.1.6) and we place them under a common framing (streams in strict languages and `stream_shape` as discussed in Section 2.1.3).

We see that, although some of them take care of the egregious overhead, none manage to eliminate all of it: the assembled stream pipeline remains slower than hand-written code.

The most straightforward representation of streams is a linked list, or a file, of elements.

It is also the least performing. The first example in Section 6.3, of summing squares, will entail: (1) creating a stream from an array by copying all elements into it; (2) traversing the list creating another stream, with squared elements; (3) traversing the result, summing the elements. We end up creating three intermediate lists. Although the whole processing still takes time linear in the size of the stream, it requires repeated traversals and the production of linear-size intermediate structures. Also, this straightforward representation cannot cope with sources that are always ready with an element: "infinite streams".

The problem, thus, is deforestation [181]: eliminating intermediate, working data structures. For streams, in particular, deforestation is typically called "stream fusion". One can discern two main groups of stream representations that let us avoid building intermediate data structures of unbounded size.

Push Streams. The first, heavily algebraic approach, represents a stream by its reducer (the fold operation) [115]. If we introduce the "shape functor" (first discussed in Section 2.1.3) for a stream with elements of type `'a` as:

```
type ('a,'z) stream_shape =
   | Nil
   | Cons of 'a * 'z
```

Figure 6.11: Shape of streams

then the stream is formally defined as:[5]

```
type 'a stream = ∀'w. (('a,'w) stream_shape → 'w) → 'w
```

Figure 6.12: Push-based stream definition

A stream of `'a`s is hence a function with the ability to turn any generic "folder" (i.e., a function from `('a,'w) stream_shape` to `'w`) to a single `'w`. The "folder" function is formally called an `F`-algebra for the `('a,-) stream_shape` functor.

For instance, an array is easily representable as such a fold:

[5]Strictly speaking, `stream` should be a record type: in OCaml, only record or object components may have the type with explicitly quantified type variables. For the sake of clarity we lift this restriction in this chapter.

```
let of_arr : 'a array → 'a stream =
   fun arr → fun folder →
   let s = ref (folder Nil) in
   for i=0 to Array.length arr − 1 do
     s := folder (Cons (arr.(i),!s))
   done; !s
```

Figure 6.13: `of_arr` **definition (push)**

Reducing a stream with the reducing function `f` and the initial value `z` is especially straightforward in this representation:

```
let fold : ('z → 'a → 'z) → 'z → 'a stream → 'z =
  fun f z str → str (
    function Nil → z
           | Cons (a,x) → f x a)
```

Figure 6.14: `fold` **definition (push)**

More germane to our discussion is that mapping over the stream (as well as `filter`-ing and `flat_map`-ing) are also easily expressible, without creating any variable-size intermediate data structures:

```
let map : ('a → 'b) → 'a stream → 'b stream =
 fun f str →
  fun folder → str (fun x → match x with
    | Nil        → folder Nil
    | Cons (a,x) → folder (Cons (f a,x)))
```

Figure 6.15: `map` **definition (push)**

A stream element `a` is transformed “on the fly” without collecting in working buffers. Our sample squaring-accumulating pipeline runs in constant memory now. Deforestation, or stream fusion, has been accomplished. The simplicity of this so-called “push stream” approach makes it popular: it is used, for example, in the reducers of Clojure as well as in the OCaml “batteries” library. It is also the basis of Java 8 Streams, under an object-oriented reformulation of the same concepts.

In push streams, it is the stream producer, e.g., `of_arr`, that drives the optimal execution of the stream. Implementing `take` and other such operators that restrict the processing to a prefix of the stream requires extending the representation with some sort of a “feedback”

mechanism (often implemented via exceptions). Where push streams stumble is the zipping of two streams, i.e., the processing of two streams in parallel. This simply cannot be done with constant per-element processing cost. Zipping becomes especially complicated (as we shall see in Section 6.7.3) when the two pipelines contain nested streams and hence produce elements at generally different rates.[6]

Pull Streams. An alternative representation of streams, pull streams, has a long pedigree, all the way from the generators of Alphard [156] in the '70s. These are objects that implement two methods: `init` to initialize the state and obtain the first element, and `next` to advance the stream to the next element, if any. Such a "generator" (or IEnumerator, as it has come to be popularly known) can also be understood algebraically—or rather, co-algebraically. Whereas push streams represent a stream as a fold, pull streams, dually, are the expression of an *unfold* [56, 115]:[7]

```
type 'a stream = ∃'s. 's * ('s → ('a,'s) stream_shape)
```

Figure 6.16: Pull-based stream definition

The stream is, hence, a pair of the current state and the so-called "step" function that, given a state, reports the end-of-stream condition `Nil`, or the current element and the next state. (Formally, the step function is the F-co-algebra for the `('a,-) stream_shape` functor.) The existential quantification over the state keeps it private: the only permissible operation is to pass it to the step function.

When an array is represented as a pull stream, the state is the tuple of the array and the current index:

```
let of_arr : 'a array → 'a stream =
  let step (i,arr) =
    if i < Array.length arr
    then Cons (arr.(i), (i+1,arr))
    else Nil
  in fun arr → ((0,arr),step)
```

Figure 6.17: `of_arr` definition (pull)

[6]The Reactive Extensions (Rx) framework [111] gives a real-life example of the complexities of implementing `zip`. Rx is push-based and supports `zip` at the cost of maintaining an unbounded intermediate queue. This deals with the "backpressure in Zip" issue, extensively-discussed in the Rx github repo. Furthermore, Rx seems to have abandoned blocking zip implementations since 2014.

[7]For the sake of explanation, we took another liberty with the OCaml notation, avoiding the GADT syntax for the existential.

The step function—a pure operator rather than a closure—dereferences the current element and advances the index. Reducing the pull stream now requires an iteration, of repeatedly calling `step` until it reports the end-of-stream. (Although the types of `of_arr`, `fold`, and `map`, etc. nominally remain the same, the meaning of `'a stream` has changed.)

```
let fold : ('z → 'a → 'z) → 'z → 'a stream → 'z =
 fun f z (s,step) →
  let rec loop z s = match step s with
    | Nil        → z
    | Cons (a,t) → loop (f z a) t
  in loop z s
```

Figure 6.18: `fold` **definition (pull)**

With pull streams, it is the reducer, i.e., the stream consumer, that drives the processing. Mapping over the stream (Figure 6.19) merely transforms its step function: `new_step` calls the old step and `map`s the returned current element, passing it immediately to the consumer, with no buffering. That is, like push streams, pull streams also accomplish fusion.

Befitting their co-algebraic nature, pull streams represent both finite and infinite streams. Stream operators, like `take`, that cut evaluation short are also easy. On the other hand, skipping elements (filtering) and nested streaming is more complex with pull streams, requiring the generalization of the `stream_shape`, as we shall see in Section 6.7. The main advantage of pull streams over push streams is in expressiveness: pull streams have the ability to process streams in parallel, enabling `zip_with` as well as more complex stream merging. Therefore, we take pull streams as the basis of our library.

```
let map : ('a → 'b) → 'a stream → 'b stream =
  fun f (s,step) →
    let new_step = fun s → match step s with
      | Nil        → Nil
      | Cons (a,t) → Cons (f a, t)
    in (s,new_step)
```

Figure 6.19: `map` **definition (pull)**

Imperfect Deforestation. Both push and pull streams eliminate the intermediate lists (variable-size buffers) that plague a naive implementation of the stream library. Yet they do not eliminate all the abstraction overhead. For example, the `map` stream operator transforms the current stream element by passing it to some function `f` received as an argument of `map`. A hand-written implementation would have no other function calls. However, the pull-stream `map` operator introduces a closure: `new_step`, which receives a `stream_shape`

value from the old `step`, pattern-matches on it and constructs the new `stream_shape`. The push-stream `map` has the same problem: The step function of `of_arr` unpacks the current state and then packs the array and the new index again into the tuple. This repeated deconstruction and construction of tuples and co-products is the abstraction overhead, which a complete deforestation should eliminate, but pull and push streams, as commonly implemented, do not. Such "constant" factors make library-assembled stream processing much slower than the hand-written version (by up to two orders of magnitude—see Section 6.8).

6.5 Staging Streams

A well-known way of eliminating abstraction overhead and delivering "abstraction without guilt" is program generation: compiling a high-level abstraction into efficient code. In fact, the original deforestation algorithm in the literature [181] is closely related to partial evaluation [159]. This section introduces staging: one particular, manual technique of partial evaluation. It lets us achieve our goal of eliminating all abstraction overhead from the stream library. Perfect stream fusion with staging is hard: Section 6.5.1 shows that straightforward staging (or automated partial evaluation) does not achieve full deforestation. We have to re-think general stream processing (Section 6.6).

6.5.1 Simple Staging of Streams

We can turn a library into, effectively, a compiler of efficient code by adding staging annotations. This is not a simple matter of annotating one of the standard definitions (either pull- or push-style) of `'a stream`, however. To see this, we next consider staging a set of pull-stream operators. Staging helps with performance, but the abstraction overhead still remains.

The first step in using staging is the so-called "binding-time analysis": finding out which values can be known only at run-time ("dynamically") and what is known already at code-generation time, ("statically") and hence can be pre-computed. Partial evaluators perform binding-time analysis, with various degrees of sophistication and success, automatically and opaquely. In staging, binding-time analysis is manual and explicit.

We start with the pull streams `map` operator. Its first argument, the mapping function `f`, takes the current stream element, which is clearly not known until the processing pipeline is run. The result is likewise dynamic. However, the mapping operation itself can be known statically. Hence the staged `f` may be given the type `'a code` $\rightarrow$ `'b code`: given code to compute `'a`s, the mapping function, `f`, is a static way to produce code to compute `'b`s.

The second argument of `map` is the pull stream, a tuple of the current state (`'s`) and the step function. The state is not known statically. The result of the step function depends on the current state and, hence, is fully dynamic. The step function itself, however, can be statically known. Hence we arrive at the following type of the staged stream:

```
type 'a st_stream =
  ∃'s. 's code * ('s code → ('a,'s) stream_shape code)
```

Figure 6.20: Type of (simply) staged streams

Having done such binding-time analysis for the arguments of the `map` operator, it is straightforward to write the staged `map`, by annotating—i.e., placing brackets and escapes on—the original `map` code according to the decided binding-times:

```
let map : ('a code → 'b code) → 'a st_stream → 'b st_stream =
  fun f (s,step) →
    let new_step = fun s → .⟨match ~(step s) with
      | Nil        → Nil
      | Cons (a,t) → Cons (~(f .⟨a⟩.), t)⟩.
    in (s,new_step)
```

Figure 6.21: `map` in (simply) staged streams

The operators `of_arr` and `fold` are staged analogously. We use the method of [75] to prove the correctness, which easily applies to this case, given that `map` is non-recursive. The sample processing pipeline (the first example from Section 6.3 in Figure 6.3), applied to the array `.⟨[0;1;2;3;4]⟩.`, produces the code below:

```
– : int code = .⟨
let rec loop_1 z_2 s_3 =
  match match match s_3 with
        | (i_4,arr_5) →
          if i_4 < (Array.length arr_5)
          then Cons ((arr_5.(i_4)),
                     ((i_4 + 1), arr_5))
          else Nil
    with
    | Nil  → Nil
    | Cons (a_6,t_7) → Cons ((a_6 * a_6), t_7)
  with
  | Nil  → z_2
  | Cons (a_8,t_9) → loop_1 (z_2 + a_8) t_9 in
loop_1 0 (0, [|0;1;2;3;4|])⟩.
```

Figure 6.22: Generation by (simply) staged streams

As expected, no lists, buffers or other variable-size data structures are created. Some constant overhead is gone too: the squaring operation of `map` is inlined. However, the triple-nested `match` betrays the remaining overhead of constructing and deconstructing `stream_shape` values. Intuitively, the clean abstraction of streams (encoded in the pull streams type of `'a stream`) isolates each operator from others. The result does not take advantage of the property that, for this pipeline (and others of the same style), the looping of all three operators (`of_arr`, `map`, and `fold`) will synchronize, with all of them processing elements until the same last one. Eliminating the overhead requires a different computation model for streams.

6.6 Eliminating All Abstraction Overhead in Three Steps

We next describe how to purge all of the stream library abstraction overhead and generate code of hand-written quality and performance. We will be continuing the simple running example of the earlier sections, of summing up squared elements of an array. (Section 6.7 will later lift the same insights to more complex pipelines.) As in Section 6.5.1, we will be relying on staging to generate well-formed and well-typed code. The key to eliminating abstraction overhead from the generated code is to move it to a generator, by making the generator take better advantage of the available static knowledge. This is easier said than done: we have to use increasingly more sophisticated transformations of the stream representation to expose more static information and make it exploitable. The three transformations we show next require more-and-more creativity and domain knowledge, and cannot be performed by a simple tool, such as an automated partial evaluator. In the process, we will identify three interesting concepts in stream processing: the structure of iteration (Section 6.6.1), the state kept (Section 6.6.2), and the optimal kind of loop construct and its contributors (Section 6.6.3).

6.6.1 Fusing the Stepper

Modularity is the cause of the abstraction overhead we observed in Section 6.5.1: structuring the library as a collection of composable components forces them to conform to a single interface. For example, each component has to use the uniform stepper function interface (see the `st_stream` type) to report the next stream element or the end of the stream. Hence, each component has to generate code to examine (deconstruct) and construct the `stream_shape` data type.

At first glance, nothing can be done about this: the result of the step function, whether it is `Nil` or a `Cons`, depends on the current state, which is surely not known until the stream processing pipeline is run. We do know however that the step function invariably returns either `Nil` or a `Cons`, and the caller must be ready to handle both alternatives. We should exploit this static knowledge.

To statically (at code generation-time) make sure that the caller of the step function handles

both alternatives of its result, we have to change the function to accept a pair of handlers: one for a `Nil` result and one for a `Cons`. In other words, we have to change the result's representation, from the sum `stream_shape` to a product of eliminators. Such a replacement effectively removes the need to construct the `stream_shape` data type at run-time in the first place. Essentially, we change `step` to be in continuation-passing style, i.e., to accept the continuation for its result. The `stream_shape` data type nominally remains, but it becomes the argument to the continuation and we mark its variants as statically known (with no need to construct it at run-time). All in all, we arrive at the following type for the staged stream:

```
type 'a st_stream =
 ∃'s. 's code *
  (∀'w. 's code → (('a code,'s code) stream_shape → 'w code) → 'w code)
```

Figure 6.23: Type of staged streams after fusing the stepper

That is, a stream is again a pair of a hidden state, `'s` (only known dynamically, i.e., `'s code`), and a step function, but the step function does not return `stream_shape` values (of dynamic `'a`s and `'s`s) but accepts an extra argument (the continuation) to pass such values to. The step function returns whatever (generic type `'w`, only known dynamically) the continuation returns.

The variants of the `stream_shape` are now known when `step` calls its continuation, which happens at code-generation time. The `map` operator becomes:

```
let map : ('a code → 'b code) → 'a st_stream → 'b st_stream =
  fun f (s,step) →
    let new_step s k = step s @@ function
    | Nil        → k Nil
    | Cons (a,t) → .⟨let a' = ∼(f a) in
                    ∼(k @@ Cons (.⟨a'⟩., t))⟩.
    in (s,new_step)
```

Figure 6.24: `map` **after fusing the stepper**

taking into account that `step`, instead of returning the result, calls a continuation on it. Although the data-type `stream_shape` remains, its construction and pattern-matching now happen at code-generation time, i.e., statically. As another example, the `fold` operator becomes:

```
let fold : ('z code → 'a code → 'z code) → 'z code → 'a st_stream → 'z code
    =
  fun f z (s,step) →
    .⟨let rec loop z s = ~(step .⟨s⟩. @@ function
        | Nil        → .⟨z⟩.
        | Cons (a,t) → .⟨loop ~(f .⟨z⟩. a) ~t⟩.)
      in loop ~z ~s⟩.
```

Figure 6.25: `fold` **after fusing the stepper**

Our running example pipeline, summing the squares of all elements of a sample array, now generates the following code

```
val c : int code = .⟨
  let rec loop_1 z_2 s_3 =
    match s_3 with
    | (i_4,arr_5) →
        if i_4 < (Array.length arr_5)
        then
          let el_6 = arr_5.(i_4) in
          let a'_7 = el_6 * el_6 in
          loop_1 (z_2 + a'_7) ((i_4 + 1), arr_5)
        else z_2 in
  loop_1 0 (0, [|0;1;2;3;4|])⟩.
```

Figure 6.26: Generated code, after fusing the stepper

In stark contrast with the naive staging of Section 6.5.1, the generated code has no traces of the `stream_shape` data type. Although the data type is still constructed and deconstructed, the corresponding overhead is shifted from the generated code to the code-generator. Generating code may take a bit longer but the result is more efficient. For full fusion, we will need to shift overhead to the generator two more times.

6.6.2 Fusing the Stream State

Although we have removed the most noticeable repeated construction and deconstruction of the `stream_shape` data type, the abstraction overhead still remains. The main loop in the generated code pattern-matches on the current state, which is the pair of the index and the array. The recursive invocation of the loop packs the index and the array back into a pair. Our task is to deforest the pair away. This seems rather difficult, however: the state is being updated on every iteration of the loop, and the loop structure (e.g., number

```
type 'a st_stream =
  ∃'s. (∀'w. ('s → 'w code) → 'w code) *
       (∀'w. 's → (('a code,unit) stream_shape → 'w code) → 'w code)
```

Figure 6.27: Type of staged streams after fusing the state

of iterations) is generally not statically known. Although it is the (statically known) `step` function that computes the updated state, the state has to be threaded through the fold's `loop`, which treats it as a black-box piece of code. The fact it is a pair cannot be exploited and, hence, the overhead cannot be shifted to the generator. There is a way out, however. It requires a non-trivial step: The threading of the state through the loop can be eliminated if the state is mutable.

The step function no longer has to return (strictly speaking: pass to its continuation) the updated state: the update happens in place. Therefore, the state no longer has to be annotated as dynamic—its structure can be known to the generator. Finally, in order to have the appropriate operator allocate the reference cell for the array index, we need to employ the let-insertion technique [19], by also using continuation-passing style for the initial state. The definition of the stream type (`'a st_stream`) now fuses the state as shown in Figure 6.27.

That is, a stream is a pair of an `init` function and a `step` function. The `init` function implicitly hides a state: it knows how to call a continuation (that accepts a static state and returns a generic dynamic value, `'w`) and returns the result of the continuation. The `step` function is much like before, but operating on a statically-known state (or more correctly, a hidden state with a statically-known structure).

```
let of_arr : 'a array code → 'a st_stream =
  let init arr k =
    .⟨let i = ref 0 and
          arr = ~arr in ~(k (.⟨i⟩.,.⟨arr⟩.))⟩.
  and step (i,arr) k =
   .⟨if !(~i) < Array.length ~arr
     then
      let el = (~arr).(!(~i)) in
      incr ~i;
      ~(k @@ Cons (.⟨el⟩., ()))
     else ~(k Nil)⟩.
  in
  fun arr → (init arr,step)
```

Figure 6.28: `of_arr`, after fusing the state

The new `of_arr` operator (Figure 6.28) demonstrates the let-insertion (the allocation of

```
let fold : ('z code → 'a code → 'z code) →
            'z code → 'a st_stream → 'z code =
  fun f z (init,step) →
    init @@ fun s →
       .⟨let rec loop z = ~(step s @@ function
           | Nil        → .⟨z⟩.
           | Cons (a,_) → .⟨loop ~(f .⟨z⟩. a)⟩.)
   in loop ~z⟩.
```

Figure 6.29: `fold`**, after fusing the state**

```
val c : int code = .⟨
  let i_8 = ref 0
  and arr_9 = [|0;1;2;3;4|] in
  let rec loop_10 z_11 =
    if ! i_8 < Array.length arr_9
    then
      let el_12 = arr_9.(! i_8) in
      incr i_8;
      let a'_13 = el_12 * el_12 in
      loop_10 (z_11+ a'_13)
    else z_11 in
  loop_10 0⟩.
```

Figure 6.30: Generated code after fusing the state

the reference cell for the current array index) in `init`, and the in-place update of the state (the `incr` operation). Once again, until now the state of the `of_arr` stream had the type `(int * 'a array) code`. It has become `int ref code * 'a array code`, the statically known pair of two code values. The construction and deconstruction of that pair now happens at code-generation time.

The earlier `map` operator did not even look at the current state (nor could it), therefore its code remains unaffected by the change in the state representation.

The `fold` operator no longer has to thread the state through its loop as shown in Figure 6.29. It obtains the state from the initializer and passes it to the step function, which knows its structure.

The generated code for the running-example stream-processing pipeline is shown in Figure 6.30. The resulting code shows the absence of any overhead. All intermediate data structures have been eliminated. The code is what we could expect to get from a competent OCaml programmer.

6.6.3 Generating Imperative Loops

It seems we have achieved our goal. The library (extended for filtering, zipping, and nested streams) can be used in (Meta)OCaml practice. It relies, however, on tail-recursive function calls. These may be a good fit for OCaml,[8] but not for Java or Scala. (In Scala, tail-recursion is only supported with significant run-time overhead.) The fastest way to iterate is to use the native while-loops, especially in Java or Scala. Also, the dummy `('a code,unit) stream_shape` in the `'a st_stream` type looks odd: the `stream_shape` data type has become artificial. Although `unit` has no effect on generated code, it is less than pleasing aesthetically to need a placeholder type in our signature. For these reasons, we embark on one last transformation.

The last step of stream staging is driven by several insights. First of all, most languages provide two sorts of imperative loops: a general while-loop and the more specific, and often more efficient (at least in OCaml) for-loops. We would like to be able to generate for-loops if possible, for instance, in our running example. However, with added subranging or zipping (described in detail in Section 6.7, below) the pipeline can no longer be represented as an OCaml for-loop, which cannot accommodate extra termination tests. Therefore, the stream producer should not commit to any particular loop representation. Rather, it has to collect all the needed information for loop generation, but leave the actual generation to the stream consumer, when the entire pipeline is known. Thus the stream representation type becomes as follows:

```
type ('a,'s) producer_t =
   | For    of
      {upb:   's → int code;
       index: 's → int code → ('a → unit code) →  unit code}
   | Unfold of
      {term: 's → bool code;
       step: 's → ('a → unit code) → unit code}
 and 'a st_stream =
   ∃'s. (∀'w. ('s → 'w code) → 'w code) *
        ('a,'s) producer_t
 and 'a stream =
   'a code st_stream
```

Figure 6.31: Type of staged streams after modularizing loop structure

That is, a stream type is a pair of an `init` function (which, as before, has the ability to call a continuation with a hidden state) and an encoding of a producer. We distinguish two sorts of producers: a producer that can be driven by a for-loop or a general “unfold” producer. Each of them supports two functions. A for-loop producer carries the exact upper bound, `upb`, for the loop index variable and the `index` function that returns the stream element

[8] Actually, our benchmarking reveals that for- and while-loops are currently faster even in OCaml.

given an index. For a general producer, we refactor (with an eye for the while-loop) the earlier representation

```
(('a code,unit) stream_shape → 'w code) → 'w code
```

Figure 6.32: Generalizing the representation of streams

into two components: the termination test, `term`, producing a dynamic `bool` value (if the test yields `false` for the current state, the loop is finished) and the `step` function, to produce a new stream element and advance the state. We also used another insight: the imperative-loop–style of the processing pipeline makes it unnecessary (moreover, difficult) to be passing around the consumer (`fold`) state from one iteration to another. It is easier to accumulate the state in a mutable cell. Therefore, the answer type of the `step` and `index` functions can be `unit code` rather than `'w code`.

There is one more difference from the earlier staged stream, which is a bit harder to see. Previously, the stream value was annotated as dynamic: we really cannot know before running the pipeline what the current element is. Now, the value produced by the `step` or `index` functions has the type `'a` without any `code` annotations, meaning that it is statically known! Although the value of the current stream element is determined only when the pipeline is run, its structure can be known earlier. For example, the new type lets the producer yield a pair of values: even though the values themselves are annotated as dynamic (of a `code` type) the fact that it is a pair can be known statically. We use this extra flexibility of the more general stream value type extensively in Section 6.7.2.

We can now see the new design in action. The stream producer `of_arr` is surely the for-loop-style producer:

```
let of_arr : 'a array code → 'a stream = fun arr →
  let init k = .⟨let arr = ~arr in ~(k .⟨arr⟩.)⟩.
  and upb arr = .⟨Array.length ~arr − 1⟩.
  and index arr i k =
      .⟨let el = (~arr).(~i) in ~(k .⟨el⟩.)⟩.
  in (init, For {upb;index})
```

Figure 6.33: `of_arr`, for imperative loop generation

In contrast, the unfold operator is an `Unfold` producer.

```
let unfold : ('z code → ('a * 'z) option code) → 'z code → 'a stream = ...
```

Figure 6.34: `unfold`, for imperative loop generation

Importantly, a producer that starts as a for-loop may later be converted to a more general while-loop producer, (so as to tack on extra termination tests—see `take` in Section 6.7.2). Therefore, we need the conversion function, in Figure 6.35, used internally within the library.

```
let for_unfold : 'a st_stream → 'a st_stream =  function
  | (init,For {upb;index}) →
     let init k = init @@ fun s0 →
             .⟨let i = ref 0 in ∼(k (.⟨i⟩.,s0))⟩.
     and term (i,s0)   = .⟨!(∼i) ≤ ∼(upb s0)⟩.
     and step (i,s0) k =
       index s0 .⟨!(∼i)⟩. @@
             fun a → .⟨(incr ∼i; ∼(k a))⟩.
     in (init, Unfold {term;step})
  | x → x
```

Figure 6.35: Code combinator transforming the loop structure for `unfold`

The stream mapping operation composes the mapping function with the `index` or `step`: transforming, as before, the produced value “in-flight”, so to speak.

```
let rec map_raw: ('a → ('b → unit code) → unit code) → 'a st_stream → 'b
    st_stream =
 fun tr → function
   | (init,For ({index;_} as g)) →
       let index s i k = index s i @@ fun e → tr e k in
       (init, For {g with index})
   | (init,Unfold ({step;_} as g)) →
       let step s k = step s @@ fun e → tr e k in
       (init, Unfold {g with step})
```

Figure 6.36: Code combinator for `map`

We have defined `map_raw` with the general type (to be used later, e.g., in Section 6.7.2); the familiar `map` is a special case:

```
let map : ('a code → 'b code) → 'a stream → 'b stream =
  fun f str → map_raw (fun a k → .⟨let t = ∼(f a) in ∼(k .⟨t⟩.)⟩.) str
```

Figure 6.37: `map` as a special case of `map_raw`

The mapper `tr` in `map_raw` is in the continuation-passing style with the `unit code` answer-type. This allows us to perform let-insertion [19], binding the mapped value to a variable, and hence avoiding the potential duplication of the mapping operation.

As behooves pull-style streams, the consumer at the end of the pipeline generates the loop to drive the iteration. Yet we do manage to generate for-loops, characteristic of push-streams, see Section 6.4.

```
let rec fold_raw : ('a → unit code) → 'a st_stream → unit code =
 fun consumer → function
   | (init,For {upb;index}) →
         init @@ fun sp →
           .⟨for i = 0 to ~(upb sp) do
              ~(index sp .⟨i⟩. @@ consumer)
             done⟩.
   | (init,Unfold {term;step}) →
         init @@ fun sp →
           .⟨while ~(term sp) do
             ~(step sp @@ consumer)
           done⟩.
```

Figure 6.38: Feeding the produced stream to the imperative `consumer`

It is simpler (especially when we add nesting later) to implement a more general `fold_raw` (Figure 6.38, which feeds the eventually produced stream element to the given imperative `consumer`. The ordinary `fold` is a wrapper that provides such a consumer (Figure 6.39), accumulating the result in a mutable cell and extracting it at the end.

```
let fold : ('z code → 'a code → 'z code) → 'z code → 'a stream → 'z code =
  fun f z str →
    .⟨let s = ref ~z in
      (~(fold_raw (fun a → .⟨s := ~(f .⟨!s⟩. a)⟩.) str);
      !s)⟩.
```

Figure 6.39: `fold` as a special case of `fold_raw`

The generated code for our running example (Figure 6.40) is what we expect an OCaml programmer to write, and, furthermore, such code performs ultimately well in Scala, Java and other languages. We have achieved our goal—for simple pipelines, at least.

```
val c : int code = .⟨
  let s_1 = ref 0 in
  let arr_2 = [|0;1;2;3;4|] in
   for i_3 = 0 to (Array.length arr_2) − 1 do
     let el_4 = arr_2.(i_3) in
     let t_5 = el_4 * el_4 in s_1 := !s_1 + t_5
   done;
  ! s_1⟩.
```

Figure 6.40: Generated code

6.7 Full Library

The previous section presented our approach of eliminating all abstraction overhead of a stream library through the creative use of staging—generating code of hand-written quality and efficiency. However, a full stream library has more operators than we have dealt with so far. This section describes the remaining facilities: filtering, sub-ranging, nested streams and parallel streams (zipping). Consistently achieving deforestation and high performance in the presence of all these features is a challenge. We identify three concepts of stream processing that drive our effort: the rate of production and consumption of stream elements (*linearity* and filtering—Section 6.7.1), size-limiting a stream (Section 6.7.2), and processing multiple streams in tandem (zipping—Section 6.7.3). We conclude our core discussion with a theorem of eliminating all overhead.

6.7.1 Filtered and Nested Streams

Our library is primarily based on the design presented at the end of Section 6.6. Filtering and nested streams (`flat_map`) require an extension, however, which lets us treat filtering and flat-mapping uniformly.

Let us look back at this design. It centers on two operations, `term` and `step`: forgetting for a moment the staging annotations, `term s` decides whether the stream still continues, while `step s` produces the current element and advances the state. Exactly one stream element is produced per advance in state. We call such streams *linear*. They have many useful algebraic properties, especially when it comes to zipping. We will exploit them in Section 6.7.3.

Clearly the `of_arr` stream producer and the more general `unfold` producers build linear streams. The `map` operation preserves the linearity. What destroys it is filtering and nesting. In the filtered stream `prod ▷ filter p`, the advancement of the `prod` state is no longer always accompanied by the production of the stream element: if the filter predicate `p` rejects the element, the pipeline will yield nothing for that iteration. Likewise, in the nested stream `prod ▷ flat_map (fun x → inner_prod x)`, the advancement of the `prod`

state may lead to zero, one, or many stream elements given to the pipeline consumer.

Given the importance of linearity (to be seen in full in Section 6.7.3) we keep track of it in the stream representation. We represent a non-linear stream as a composition of an always-linear producer with a non-linear transformer. The final data type is shown in Figure 6.41.

The difference from the earlier representation in Section 6.6 is the addition of a sum data type with variants `Linear` and `Nested`, for linear and nested streams. We also added a cardinality marker to the general producer, noting if it generates possibly many elements or at most one.

```
type card_t = AtMost1
            | Many

type ('a,'s) producer_t =
  | For    of
     {upb:   's → int code;
      index: 's → int code → ('a → unit code) → unit code}
  | Unfold of
     {term: 's → bool code;
      card: card_t;
      step: 's → ('a → unit code) → unit code}

 and 'a producer =
   ∃'s. (∀'w. ('s → 'w code) → 'w code) *
              ('a,'s) producer_t
 and 'a st_stream =
  | Linear of 'a producer
  | Nested of ∃'b. 'b producer * ('b → 'a st_stream)

 and 'a stream = 'a code st_stream
```

Figure 6.41: Final data type of staged streams

The `flat_map` operator (Figure 6.42) adds a non-linear transformer to the stream (recursively descending into the already nested stream) and the `filter` operator (Figure 6.43) becomes just a particular case of flat-mapping: nesting of a stream that produces at most one element.

The addition of recursively `Nested` streams requires an adjustment of the earlier, Section 6.6, `map_raw` and `fold` definitions to recursively descend down the nesting. The adjustment is straightforward; please see the accompanying source code for details. The adjusted `fold` will generate nested loops for nested streams.

```
let rec flat_map_raw : ('a → 'b st_stream) → 'a st_stream → 'b st_stream =
  fun tr → function
    | Linear prod         → Nested (prod,tr)
    | Nested (prod,nestf) → Nested (prod,fun a → flat_map_raw tr @@ nestf a)

let flat_map : ('a code → 'b stream) → 'a stream → 'b stream =
  flat_map_raw
```

Figure 6.42: `flat_map_raw` **handling nested and linear streams**

```
let filter : ('a code → bool code) →
             'a stream → 'a stream = fun f →
  let filter_stream a =
  ((fun k → k a), Unfold {card = AtMost1; term = f;
                          step = fun a k → k a})
  in flat_map_raw (fun x → Linear (filter_stream x))
```

Figure 6.43: `filter` **as a special case of** `flat_map_raw`

6.7.2 Sub-Ranging and Infinite Streams

The stream operator `take` limits the size of the stream:

```
val take : int code → 'a stream → 'a stream
```

Figure 6.44: `take`**, in Strymonas**

For example, `take .⟨10⟩. str` is a stream of the first 10 elements of `str`, if there are that many. It is the `take` operator that lets us handle conceptually infinite streams. Such infinite streams are easily created with `unfold`: for example, `iota n`, the stream of all natural numbers from `n` up:

```
let iota n = unfold (fun n → .⟨Some (~n,~n+ 1)⟩.) n
```

Figure 6.45: `iota` **operator for infinite streams**

The implementation of `take` demonstrates and justifies design decisions that might have seemed arbitrary earlier. For example, distinguishing linear streams and indexed, for-loop–style producers in the representation type pays off. In a linear stream pipeline, the number of elements at the end of the pipeline is the same as the number of produced

elements. Therefore, for a linear stream, `take` can impose the limit close to the production. The for-loop-style producer is particularly easy to limit in size: we merely need to adjust the upper bound:

```
let take = fun n → function
 | Linear (init, For {upb;index}) →
    let upb s = .⟨min (~n−1) ~(upb s)⟩. in
    Linear (init, For {upb;index})
 ...
```

Figure 6.46: `take` **operator handling Linear cases**

Limiting the size of a non-linear stream is slightly more complicated:

```
let take = fun n → function
 ...
 | Nested (p,nestf) →
    Nested (add_nr n (for_unfold p),
     fun (nr,a) →
      map_raw (fun a k → .⟨(decr ~nr; ~(k a))⟩.) @@
      more_termination .⟨! ~nr > 0⟩. (nestf a))
```

Figure 6.47: `take` **operator handling Nested cases**

The idea is straightforward: allocate a reference cell `nr` with the remaining element count (initially `n`), add the check `!nr > 0` to the termination condition of the stream producer, and arrange to decrement the `nr` count at the end of the stream. Recall, for a non-linear stream—a composition of several producers—the count of eventually produced elements may differ arbitrarily from the count of the elements emitted by the first producer. A moment of thought shows that the range check `!nr > 0` has to be added not only to the first producer but to the producers of all nested substreams: this is the role of function `more_termination` (see the accompanying code for its definition) in the fragment above. The operation `add_nr` allocates cell `nr` and adds the termination condition to the first producer. Recall that, since for-loops in OCaml cannot take extra termination conditions, a for-loop-style producer has to be first converted to a general unfold-style producer, using `for_unfold`, which we defined in Section 6.6. The operation `add_nr` (definition not shown) also adds `nr` to the produced value: The result of `add_nr n (for_unfold p)` is of type `(int ref code,'a code) st_stream`. Adding the operation to decrement `nr` is conveniently done with `map_raw` from Section 6.6. We, thus, now see the use for the more general (`'a` and not just `'a code`) stream type and the general stream mapping function.

6.7.3 zip: Fusing Parallel Streams

This section describes the most complex operation: handling two streams in tandem, i.e., zipping in Figure 6.48.

```
val zip_with : ('a code → 'b code → 'c code) →
               ('a stream → 'b stream → 'c stream)
```

Figure 6.48: `zip_with` **operator**

Many stream libraries lack this operation: first, because zipping is practically impossible with push streams, due to inherent complexity, as we shall see shortly. Linear streams and the general `map_raw` operation turn out to be important abstractions that make the problem tractable.

One cause of the complexity of `zip_with` is the need to consider many special cases, so as to generate code of hand-written quality. All cases share the operation of combining the elements of two streams to obtain the element of the zipped stream. It is convenient to factor out this operation in auxiliary `zip_raw` (Figure 6.49). The `zip_raw` operator builds a stream of pairs—statically known pairs of dynamic values. Therefore, the overhead of constructing and deconstructing the pairs is incurred only once, in the generator. There is no tupling in the generated code.

```
val zip_raw: 'a st_stream → 'b st_stream → ('a * 'b) st_stream

let zip_with f str1 str2 =
   map_raw (fun (x,y) k → k (f x y)) @@
   zip_raw str1 str2
```

Figure 6.49: `zip_with` **operator as a composition of** `map_raw` **and** `zip_raw`

The `zip_raw` function (Figure 6.50) is a dispatcher for various special cases. The simplest case is zipping two linear streams. Recall, a linear stream produces exactly one element when advancing the state. Zipped linear streams, hence, yield a linear stream that produces a pair of elements by advancing the state of both argument streams exactly once. The pairing of the stream advancement is especially efficient for `for`-loop–style streams, which share a common state, the index. `zip_producer` implements exactly that behavior when both streams are linear (Figure 6.51). The step functions are called in an alternate fashion (e.g., step1 step2 step1 ...) but the order of calling the two step functions is not specified. `zip_producer` handles three cases: first, when both streams are `for`-based, second, when both are `unfold`-based and a third, general, case that calls `for_unfold`. In Figure 6.51 we show the first case.

```
let rec zip_raw str1 str2 = match (str1,str2) with
  | (Linear prod1, Linear prod2) →
      Linear (zip_producer prod1 prod2)

  | (Linear prod1, Nested (prod2,nestf2)) →
      push_linear (for_unfold prod1)
                  (for_unfold prod2,nestf2)

  | (Nested (prod1,nestf1), Linear prod2) →
      map_raw (fun (y,x) k → k (x,y)) @@
      push_linear (for_unfold prod2)
                  (for_unfold prod1,nestf1)

  | (str1,str2) →
      zip_raw (Linear (make_linear str1)) str2
```

Figure 6.50: `zip_raw` **handling four combinations of linear and nested cases**

```
let rec zip_producer: 'a producer → 'b producer → ('a * 'b) producer =
 fun p1 p2 → match (p1,p2) with
 | (i1,For f1), (i2,For f2) →
     let init k =
       i1.init @@ fun s1 →
       i2.init @@ fun s2 → k (s1,s2)

     and upb (s1,s2) =
       .⟨min ~(f1.upb s1) ~(f2.upb s2)⟩.)

     and index fun (s1,s2) i k =
       f1.index s1 i @@ fun e1 →
       f2.index s2 i @@ fun e2 → k (e1,e2)
     in (init, For {upb;index})

  | (* elided *)
```

Figure 6.51: Zipping linear streams

In the general case, `zip_raw str1 str2` has to determine how to advance the state of `str1` and `str2` to produce one element of the zipped stream: the pair of the current elements of `str1` and `str2`. Informally, we have to reason all the way from the production of an element to the advancement of the state. For linear streams, the relation between the

current element and the state is one-to-one. In general, the state of the two components of the zipped stream advance at different paces. Consider the following sample streams:

```
let stre = of_arr arr1
           ▷ filter (fun x → .⟨~x mod 2 = 0⟩.)

let strq = of_arr arr2
           ▷ map (fun x → .⟨~x * ~x⟩.)

let str2 = of_arr arr1
           ▷ flat_map (fun _ → of_arr .⟨[|1;2]⟩.)

let str3 = of_arr arr1
           ▷ flat_map (fun _ → of_arr .⟨[|1;2;3]⟩.)
```

Figure 6.52: Different paces of streams

To produce one element of `zip_raw stre strq`, the state of `stre` has to be advanced a statically-unknown number of times. Zipping nested streams is even harder such as `zip_raw str2 str3`, where the states advance in complex patterns and the end of the inner stream of `str2` does not align with the end of the inner stream in `str3`.

Zipping simplifies if one of the streams is linear, as in `zip_raw stre strq`. The key insight is to advance the linear stream `strq` after we are sure to have obtained the element of the non-linear stream `stre`. This idea is elegantly realized as mapping of the step function of `strq` over `stre` (the latter, is, recall, `int stream`, which is `int code st_stream`), obtaining the desired zipped `(int code, int code) st_stream`:

```
map_raw (fun e1 k → strq.step sq (fun e2 → k (e1,e2))) stre
```

Figure 6.53: Propagating the termination condition between two streams

The above code is an outline: we have to initialize `strq` to obtain its state `sq`, and we need to push the termination condition of `strq` into `stre`. Function `push_linear` in the accompanying code takes care of all these details.

The last and most complex case is zipping two non-linear streams. Our solution is to convert one of them to a linear stream, and then use the approach just described. Turning a non-linear stream to a producer involves “reifying” a stream: converting an `'a stream` data type to essentially a `(unit → 'a option) code` function, which, when called, reports the new element or the end of the stream. We have to create a closure and generate and deconstruct the intermediate data type `'a option`. There is no way around this: in one form or another, we have to capture the non-linear stream’s continuation. The human

programmer will have to do the same—this is precisely what makes zipping so difficult in practice. Our library reifies only one of the two zipped streams, without relying on tail-call optimization, for maximum portability.

6.7.4 Elimination of All Overhead, Formally

Sections 6.3, above, and 6.8, below, demonstrate the elimination of abstraction overhead on selected examples and benchmarks. We now state how and why the overhead is eliminated in all cases.

We call the higher-order arguments of `map`, `filter`, `zip_with`, etc. "user-generators": they are specified by the library user and provide per-element stream processing.

Theorem 1 *Any well-typed pipeline generator–built by composing a stream producer (Figure 6.2) with an arbitrary combination of transformers followed by a reducer–terminates, provided the user-generators do. The resulting code–with the sole exception of pipelines zipping two flat-mapped streams–constructs no data structures beyond those constructed by the user-generators.*

Therefore, if the user generators proceed without construction and allocation, the entire pipeline, after the initial set-up, runs without allocations. The only exception is the zipping of two streams that are both made by flattening inner streams. In this case, the rate-adjusting allocation is inevitable, even in hand-written code, and is not considered overhead.

Proof sketch: The proof is simple, thanks to the explicitness of staging and treating the generated code as an opaque value that cannot be deconstructed and examined. Therefore, the only tuple construction operations in the generated code are those that we have explicitly generated. Hence, to prove our theorem, we only have to inspect the brackets that appear in our library implementation, checking for tuples or other objects.

6.8 Experiments

We evaluated our approach on several benchmarks from previous chapters measuring the iteration throughput. Specifically we rely on a combination of benchmarks from Chapter 3 (consequently, from Murray et al. [118] as well) and Chapter 4. Our benchmarks also come Coutts et al. [38], to which we added more complex combinations (the last three on the list above). (The Murray and Coutts sets also contain a few more simple operator combinations, which we omit for conciseness, as they share the performance characteristics of other benchmarks.)

- **sum**: the simplest `of_arr arr ▷ sum` pipeline, summing the elements of an array;

- **sumOfSquares**: our running example from Section 6.5.1 on;
- **sumOfSquaresEven**: the sumOfSquares benchmark with added filter, summing the squares of only the even array elements;
- **cart**: $\sum x_i y_j$, using `flat_map` to build the outer-product stream;
- **maps**: consecutive map operations with integer multiplication;
- **filters**: consecutive filter operations using integer comparison;
- **dotProduct**: compute dot product of two arrays using `zip_with`;
- **flatMap_after_zipWith**: compute $\sum (x_i + x_i) y_j$, like cart above, doubling the `x` array via `zip_with (+)` with itself;
- **zipWith_after_flatMap**: `zip_with` of two streams one of which is the result of `flat_map`;
- **flat_map_take**: `flat_map` followed by `take`.

The source code of all benchmarks is available at the project's repository and the OCaml versions are also listed in Appendix B.4.

The staged code was generated using our library (Strymonas), with MetaOCaml on the OCaml platform and LMS on Scala, as detailed below. As one basis of comparison, we have implemented all benchmarks using the streams libraries available on each platform:[9] Batteries[10] in OCaml and the standard Java 8 and Scala streams. As there is not a unifying module that implements all the operators we employ, we use data type conversions where possible. Java 8 does not support a `zip` operator, hence some benchmarks are missing for that setup.[11]

As the baseline and the other basis of comparison, we have hand-coded all the benchmarks, using high-performance, imperative code, with `while` or index-based `for`-loops, as applicable. In Scala we use only `while`-loops as they are the analogue of imperative iterations; `for`-loops in Scala operate over `Range`s and have worse performance. In fact, in one case we had to re-code the hand-optimized loop upon discovering that it was not as optimal as we thought: the library-generated code significantly outperformed it!

[9] We restrict our attention to the closest feature-rich apples-to-apples comparables: the industry-standard libraries for OCaml+JVM languages. We also report qualitative comparisons in Section 7.3.

[10] Batteries is the widely used "extended standard" library in OCaml `http://batteries.forge.ocamlcore.org/`.

[11] One could emulate `zip` using `iterator` from Java 8 push-streams—at significant drop in performance. This encoding also markedly differs from the structure of our other stream implementations.

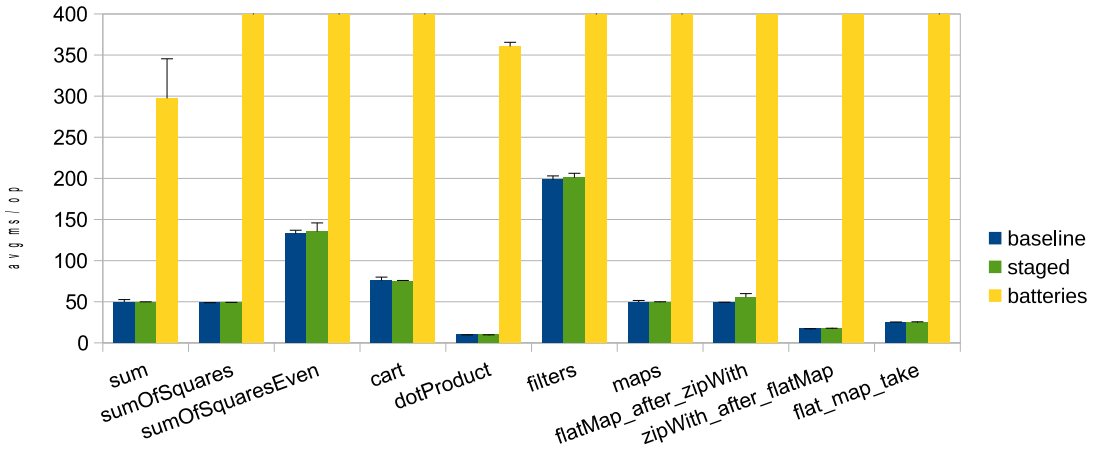


Figure 6.54: OCaml microbenchmarks in milliseconds/operation (avg. of 30, with mean-error bars shown). "Staged" is our library (Strymonas). The figure is truncated: OCaml batteries take more than 60sec (per iteration!) for some complex benchmarks.

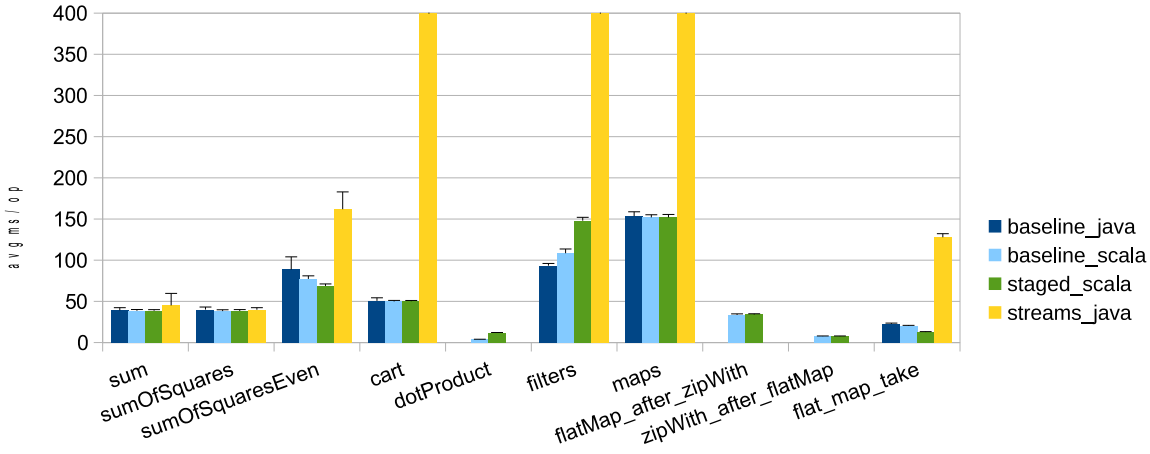


Figure 6.55: JVM microbenchmarks (both Java and Scala) in milliseconds/operation (avg. of 30, with mean-error bars shown). "Staged_scala" is our library (Strymonas). The figure is truncated.

Input. All tests were run with the same input set. For the **sum**, **sumOfSquares**, **sumOfSquaresEven**, **maps**, **filters** we used an array of $N = 100,000,000$ small integers: $x_i = i \bmod 10$. The **cart** test iterates over two arrays. An outer one of $10,000,000$ integers and an inner one of 10. For the **dotProduct** we used $10,000,000$ integers, for the **flatMap_after_zipWith** $10,000$, for the **zipWith_after_flatMap** $10,000,000$ and for the **flat_map_take** N numbers sub-sized by 20% of N.

Setup. The system we use runs an x64 OSX El Capitan 10.11.4 operating system on bare metal. It is equipped with a 2.7 GHz Intel Core i5 CPU (I5-5257U) having 2 physical and 2 logical cores. The total memory of the system is 8 GB of type 1867 MHz DDR3. We use version build 1.8.0_65-b17 of the Open JDK. The compiler versions of our setup are presented in the figure below:

Language	Compiler	Staging
Java	Java 8 (1.8.0_65)	—
Scala	2.11.2	LMS 0.9.0
OCaml	4.02.1	BER MetaOCaml N102

Figure 6.56: Experimental setup

Automation. For Java and Scala benchmarks we used the Java Microbenchmark Harness (JMH) [157] tool: a benchmarking tool for JVM-based languages that is part of the OpenJDK. JMH is an annotation-based tool and takes care of all intrinsic details of the execution process. Its goal is to produce as objective results as possible. The JVM performs JIT compilation (we use the C2 JIT compiler) so the benchmark author must measure execution time after a certain warm-up period to wait for transient responses to settle down. JMH offers an easy API to achieve that. In our benchmarks we employed 30 warm-up iterations and 30 proper iterations. We also force garbage collection before benchmark execution and between runs. All OCaml code was compiled with `ocamlopt` into machine code. In particular, the MetaOCaml-generated code was saved into a file, compiled, and then benchmarked in isolation. The test harness invokes the compiled executable via `Sys.command`, which is not included in the results. The harness calculates the average execution time, computing the mean error and standard deviation using the Student-T distribution. The same method is employed in JMH. For all tests, we do not measure the time needed to initialize data-structures (filling arrays), nor the run-time compilation cost of staging. These costs are constant (i.e., they become proportionally insignificant for larger inputs or more iterations) and they were small, between 5 and 10ms, for all our runs.

Results. In Figures 6.54 and 6.55 we present the results of our experiments divided into two categories: a) the OCaml microbenchmarks of baseline, staged and batteries experiments and b) the JVM microbenchmarks. The JVM diagram contains the baselines

for both Java and Scala. Shorter bars are better. Recall that all “baseline” implementations are carefully hand-optimized code.

As can be seen, our staged library achieves extremely high performance, matching hand-written code (in either OCaml, Java, or Scala) and outperforming other library options by orders of magnitude. Notably, the highly-optimized Java 8 streams are more than 10x slower for perfectly realistic benchmarks, when those do not conform to the optimal pattern (linear loop) of push streams.

6.9 Discussion: Why Staging?

Our approach relies on staging. This may impose a barrier to the practical use of the library: staging annotations are unfamiliar to many programmers. Furthermore, it is natural to ask whether our approach could be implemented as a compiler optimization pass.

Complexity of staging. How much burden staging really imposes on a programmer is an empirical question. As our library becomes known and more-used we hope to collect data to answer this. In the meantime, we note that staging can be effectively hidden in code combinators. The first code example of Section 6.3 (summing the squares of elements of an array) can be written without the use of staging annotations as:

```
let sum = fold (fun z a → add a z) zero

of_arr arr
  ▷ map (fun x → mul x x)
  ▷ sum
```

Figure 6.57: Comparing the Strymonas API to unstaged APIs

In this form, the functions that handle stream elements are written using a small operator library, with operations `add`, `mul`, etc. that hide all staging. The operations are defined simply as:

```
let add x y = .⟨~x + ~y⟩. and mul x y = .⟨~x * ~y⟩.

let zero = .⟨0⟩.
```

Figure 6.58: Adding staging annotations

Furthermore, our Scala implementation has no explicit staging annotations, only `Rep` types (which are arguably less intrusive). For instance, a simple pipeline is shown below:

```
def test (xs : Rep[Array[Int]]) : Rep[Int] =
  Stream[Int](xs)
    .filter(d ⇒ d % 2 == 0)
    .sum
```

Figure 6.59: No staging annotations in Scala implementation

Staging vs. compiler optimization. Our approach can certainly be cast as an optimization pass. The current staging formulation is an excellent blueprint for such a compiler rewrite. However, staging is both less intrusive and more disciplined—with high-level type safety guarantees—than changing the compiler. Furthermore, optimization is guaranteed only with full control of the compiler. Such control is possible in a domain-specific language, but not in a general-purpose language, such as the ones we target. Relying on a general-purpose compiler for library optimization is slippery. Although compiler analyses and transformations are (usually) sound, they are almost never complete: a compiler generally offers no guarantee that any optimization will be successfully applied.[12] There are several instances when an innocuous change to a program makes it much slower. The compiler is a black box, with the programmer forced into constantly reorganizing the program in unintuitive ways in order to achieve the desired performance.

6.10 Summary of Stream Fusion, to Completeness

We have presented the principles and the design of stream libraries that support the widest set of operations from past libraries and also permit elimination of the entire abstraction overhead. The design has been implemented as the Strymonas library, for OCaml and for Scala/JVM. As confirmed experimentally, our library indeed offers the highest, guaranteed, and portable performance. Underlying the library is a representation of streams that captures the essence of iteration in streaming pipelines. It recognizes which operators drive the iteration, which contribute to filtering conditions, whether parts of the stream have linearity properties, and more. This decomposition of the essence of stream iteration is what allows us to perform very aggressive optimization, via staging, regardless of the streaming pipeline configuration.

[12] A recent quote by Ben Lippmeier, discussing RePa [85] on Haskell-Cafe, captures well the frustrations of advanced library writers: “The compilation method [...] depends on the GHC simplifier acting in a certain way—yet there is no specification of exactly what the simplifier should do, and no easy way to check that it did what was expected other than eyeballing the intermediate code. We really need a different approach to program optimisation [...] The [current approach] is fine for general purpose code optimisation but not ‘compile by transformation’ where we really depend on the transformations doing what they’re supposed to.”—`https://web.archive.org/web/20170322141138/http://mail.haskell.org/pipermail/haskell-cafe/2016-July/124324.html`

7. RELATED WORK

This chapter includes related work of previous chapters. More specifically, Section 7.1 contains related work for Chapters 3 and 4. Section 7.2 contains related work for Chapter 5 and Section 7.3 for the previous chapter, Chapter 6.

7.1 Extensibility of Streams

In the rest of this section, we discuss related work both from dedicated streaming systems, and from streaming APIs built on top of general-purpose programming languages.

7.1.1 Streaming DSLs and interpreters

From the implementor's point of view, stream algebras in Chapter 4 expose a DSL for stream programming. The DSL is embedded in Java and takes advantage of the optimizing nature of the underlying JIT-based implementation. In a similar fashion, the StreamIt language, which needed a special implementation with stream-specific analyses and optimizations [171], was implemented atop the Java platform, as StreamJIT [20]. However, while StreamJIT required a full implementation effort, our technique is more lightweight, being available as a Java library, with an API extending that of native Java streams. StreamIt generally has a very different focus from our work: although it offers flexible, declarative operators, its domain of applicability is multimedia streams of very large data, as opposed to general functional programming over data collections.

The DirectFlow DSL of Lin and Black also supported push and pull configurations for information-flow systems with extensible operators [99]. Compared to our design, it uses a compiler, exposes the internal "pipes" that connect different stream operators, and requires the management (instantiation and connection) of objects for these pieces of the flow graph.

The operator fusion semantics that we showed is only one example of stream-based optimizations. Other optimizations that can be unlocked by our technique are operator reordering, redundancy elimination in the pipeline, and batching [70].

Our DSL representation follows a shallow embedding with Church-style encodings [124]; as Gibbons and Wu have demonstrated, this makes it natural and simple to implement the interpreter in recursive style [57].

Our design permits not only the definition of completely new operators, but also composite operators that reuse existing ones. This expressiveness enables modularity in our design in a manner similar to the modularity offered by the composite operators of high-level streaming languages such as SPL [69].

7.1.2 Collections and big data (including Java streams)

Su et al. showed how Java streams can support different compute engines in the same pipeline [164], for the domain of distributed data sets. Unlike our design, there is no infrastructure for the change of evaluation engine without affecting the library code.

Our approach can process the pipeline, so in that respect is similar to the "application DSL" of ScalaPipe [191]. ScalaPipe operates as a program generator. It generates an AutoPipe application that produces C, OpenCL etc. The high level program is written in Scala. The target program is C.

StreamFlex offers high-throughput, low-latency streaming capabilities in Java, taking advantage of ownership types [160]. StreamFlex is an extension of Java (due to type system additions), and, furthermore, comes with an altered JVM to support real time execution. It focuses on event processing and especially on the scheduling of filters (so that priority is given to the threads that handle the stream without GC pressuring, etc.).

Compared to the above, our work is smoothly integrated in the language (as an embedded, internal DSL—effectively a plain library). We discover the DSL hidden inside the Java Streams API and show how its implementation can improve, with pluggable and modular semantics, via object algebras.

Svensson and Svenningsson demonstrated how pull-style and push-style array semantics can be combined in a single API using a defunctionalized representation and a shallow embedding for their DSL [166]. However, they propose a new API and a separate DSL layer that passes through a compiler, while we remain compatible with existing Java-like stream APIs. Furthermore, our approach enables full semantic extensibility, beyond just changing the pull vs. pull style of iteration.

7.2 Extensibility of Java

Syntactic and semantic extensibility of programming languages has received a lot of attention in literature, historically going back to Landin's, "Next 700 Programming Languages" [98]. In this section we focus on work that is related to Recaf from the perspective of semantic language virtualization, languages as libraries, and semantic language customization.

7.2.1 Language Virtualization

Language virtualization allows the programmer to redefine the meaning of existing constructs and define new ones for a programming language. Well-known examples include LINQ [114]) that offers query syntax over custom data types, Haskell's `do`-notation for defining custom monadic evaluation sequences, and Scala's `for`-comprehensions. In terms of language extensibility and DSL embeddings we also gain inspiration from Carette et al. [29]. This work demonstrates that programs can be represented using ordinary

functions rather than data constructors and this technique that has its roots in Reynold's work [142].

Scala Virtualized [147] and Lightweight Modular Staging (LMS) [145] are frameworks to redefine the meaning of almost the complete Scala language. However, these frameworks rely on the advanced typing mechanisms of Scala (higher-kinded types and implicits) to implement concrete implementations of DSL embeddings. Additionally, compared to Scala, Java does not have support for delimited continuations so we rely on a CPS interpretation to mitigate that. Recaf scopes virtualization to methods, a choice motivated by the statement-oriented flavor of the Java syntax, and inspired by how the `async`, `sync*` and `async*` modifiers are scoped in Dart [116] and `async` in C# [17].

Another related approach is the work on F#'s computation expressions [130] which allow the embedding of various computational idioms via the definition of concrete computation builder objects, similar to our Object Algebras. The F# compiler desugars ordinary F# expressions to calls into the factory, in an analogous way to the transformation employed by Recaf. Note that the semantic virtualization capabilities offered by computation expressions are scoped to the expression level. Both, Recaf and F# support custom operators, however in F# they are not supported in control flow statements [168]. Carette et al. [29] construct CPS interpreters among others. In Recaf we use the same approach to enable control-flow manipulation extensions.

7.2.2 Languages as Libraries

Recaf is a framework for library-based language extension. The core idea of "languages as libraries" is that embedded languages or language extensions exist at the same level as ordinary user code. This is different, for instance, from extensible compilers (e.g., [122]) where language extensions are defined at the meta level.

The SugarJ system [46] supports language extension as a library, where new syntactic constructs are transformed to plain Java code by specifying custom rewrite rules. The Racket system supports a similar style of defining library-based languages by transformation, leveraging a powerful macro facility and module system [172]. A significant difference from Recaf is that, in both SugarJ and Racket, the extension developer writes the transformations herself, whereas in Recaf the transformation is generic and provided by the framework.

7.2.3 Language Customization

Language extension is only one of the use cases supported by Recaf. Recaf can also be used to instrument or modify the base semantics of Java. Consequently, Recaf can be seen as a specific kind of meta object protocol [88], where the programmer can customize the dynamic semantics of a programming language, from within the host language itself. OpenC++ [32] introduced such a feature for C++, allowing the customization of member

access, method invocation and object creation.

7.3 Stream Fusion

The literature on stream library designs is rich. Our approach is the first to offer full generality while eliminating processing overhead. We discuss individual related work in more detail next.

7.3.1 Stream Fusion as an optimization

Burstall R. M. and Darlington J. [26] propose a semi-automatic mechanism. Their algorithm consists of two key transformations, the first one being the *unfolding* of a function definition for their definitions and the second to perform a *folding* of the definition back to its definition after a number of transformation passes. The strategy is considered semi-automatic as it was up to the user to introduce auxiliary function definitions to guide the transformation steps. That particular work has been a key point in the history of program transformations and for stream optimizations in general.

Haskell has lazy lists, which seem to offer incremental processing by design. Lazy lists cannot express pipelines that require side-effects such as reading or writing files[1]. The all-too-common memory leaks point out that lazy lists do not offer, again by design, stream fusion. Overcoming the drawbacks of lazy lists, coroutine-like iteratees [89] and many of their reimplementations support incremental processing even in the presence of effects, for nested streams and for several consumers and producers. Although iteratees avoid intermediate streams they still suffer large overheads for captured continuations, closures, and coroutine calls.

The work of Burstall R. M. and Darlington J. inspired Wadler's *listless transformer* [183] that eliminates intermediate data structures of a small set of operators for list processing in a lazy language. Wadler's deforestation algorithm [181] was followed by Gill et al. who developed shortcut fusion [58] which performs local optimizations in the form of *fusion rules*. That system combined with GHC's RULES [133] gave to library authors the ability to identify optimization opportunities in the form of algebraic equations e.g., two consecutive filters `filter p . filter q` can be replaced by `filter (\x -> q x && p x)`, fusing essentially the two lambdas together.

Shortcut fusion also specifies an implementation strategy for list producers, intermediate operators and consumers. All known operators are written using `foldr` and producers are written using `build` in the effort to help the compiler discover pairs of a production and consumption with a single fusion rule called *foldr/build* avoiding the creation of an intermediate lists. Svenningsson a few years later proposes a system [165] not based on `foldr` but on its dual, the function `unfoldr` covering a wider range of optimization opportunities,

[1] We disregard the lazy IO misfeature [89].

namely of pipelines containing `zip` and accumulating parameters. That system comes with its own fusion rule, implementing the so called `unbuild/unfoldr` fusion. Coutts et al. proposes an improvement over `unbuild/unfoldr` called *Stream Fusion* (the approach that has become associated with this fixed term) [38] that also fuses pipelines with `filter`. The approach relies on the rewrite GHC Rules. Its notable contribution is the support for stream filtering. In that approach there is no specific treatment of linearity. The Coutts et al. stream fusion supports zipping, but only in simple cases (no zipping of nested, subranged streams). Finally, the Coutts et al. approach does not fully fuse pipelines that contain nested streams (`concatMap`). The reason is that the stream created by the transformation of `concatMap` uses an internal function that cannot by optimized by GHC by employing simple case reduction. The problem is presented very concisely by Farmer et al. in the *Hermit in the Stream* work [49].

The application of HERMIT [49] to streams [48] fixes the shortcomings of the Coutts et al. Stream Fusion [38] for `concatMap`. As the authors and Coutts say, `concatMap` is complicated because its mapping function may create any stream whose size is not statically known. The authors implement Coutts's idea of transforming `concatMap` to `flatten`; the latter supports fusion for a constant inner stream. Using HERMIT instead of GHC Rules, Farmer et al. present as motivating examples two cases. Our approach handles the *non-constant inner stream case* without any additional action.

```
concatMapS (\x → case even x of
   True → enumFromToS 1 x
   False → enumFromToS 1 (x + 1))
```

Figure 7.1: Multiple inner streams

The second case is about *multiple inner streams* (of the same state type). Farmer et al. eliminate some overhead yet do not produce fully fused code. E.g., pipelines such as the one in Figure 7.1 (in Haskell), are not fully fused (Farmer et al. raise the question of how often such cases arise in a real program.)

```
flat_map_cps (fun x k →
  .⟨ if (even ∼x)
     then ∼(k (enumFromToS ...))
     else ∼(k (enumFromToS ...)) ⟩.)
```

Figure 7.2: `flat_map_cps` as an alternative

Our library internally places no restrictions on inner streams; it may well be that the flat-mapping function produces streams of different structure for each element of the outer stream. On the other hand, the `flat_map` interface only supports nested streams of a fixed structure—hence with the applicative rather than monadic interface. We can provide

a more general `flat_map` with the continuation-passing interface for the mapping function, which then implements a `flat_map_cps` as in Figure 7.2.

We have refrained from offering this more general interface since there does not seem to be a practical need.

GHC Rules [133], extensively used in Stream Fusion, are applied to typed code but by themselves are not typed and are not guaranteed type-preserving. To write GHC rules, one has to have a very good understanding of GHC optimization passes, to ensure that the RULE matches and has any effect at all. Rules by themselves offer no guarantee, even the guarantee that the re-written code is well-typed. Multi-stage programming ensures that all staging transformations are type-correct.

Repa by Keller et al. [85] is a Haskell library that supports multi-dimensional arrays types and fast, parallel operations over them. Repa supports shape polymorphism over the representation of arrays (shape is the dimensionality of an array in the form of a type) so pipelines expressed using the well known functional operators (such as `map`, `fold`) take under consideration that information at the type level.

Lippmeier et al. [102] present a line of work based on SERIES. They aim to transform first-order, non-recursive, synchronous, finite data-flow programs into fused pipelines. They derive inspiration from traditional data-flow languages like Lustre [64] and Lucid Synchrone [136]. In contrast, our library supports a greater range of fusible operators, but for bulk data processing.

Generalized Stream Fusion [106] puts forward the idea of *bundled* stream representations. Each representation is designed to fit a particular stream consumer following the documented cost model. Although this design does not present a concrete range of optimizations to fuse operators and generate loop-based code directly, it presents a generalized model that can “host” any number of specialized stream representations. Conceptually, this framework could be used to implement our optimizations. However, it relies on the black-box GHC optimizer—which is the opposite of our approach of full transparency and portability.

Svensson et al.[166] unify pull- and push-arrays into a single library by defunctionalizing push arrays, concisely explaining why pull and push must co-exist under a unified library. They use a *compile* monad to interpret their embedded language into an imperative target one. In our work we get that for free from staging. Similarly, the representation of arrays in memory, with their `CMMem` data type, corresponds to staged arrays (of type `'a array code`) in our work. The library they derive from the defunctionalization of `Push` streams is called `PushT` and the authors provide evidence that indexing a push array can, indeed, be efficient (as opposed to simple push-based streams). The paper does not seem to handle more challenging operators like `concatMap` and `take` and does not efficiently handle the combinations of infinite and finite sources. Still, we share the same goal: to unify both styles of streams under one roof. Finally, Svensson et al. target arrays for embedded languages, while we target arrays natively in the language. Fusion is achieved by our library without relying on a compiler to intelligently handle all corner cases.

7.3.2 Stream Fusion and Code Generation

Chapter 6 defended the thesis that in order to achieve complete stream fusion we can drive domain-specific transformations at the library level. Active libraries [178] have pioneered this shift, to program the optimizations in application-specific program generators rather than in the compiler itself. ATLAS [190] was the first project to abstract Basic Linear Algebra Subroutines (BLAS) using the Automated Empirical Optimization of Software (AEOS) technique. In AEOS, the system adapts to the underlying architecture by iteratively performing critical operations and tries to find the most optimal implementation. FFTW follows a similar philosophy for FFT calculations [54]. Spiral [140] introduces a LISP-based DSL, called SPL that decouples the mathematical notation of DSP programming from the actual running code. Additionally, Spiral offers the user the ability to control code generation through the use of various compiler options, compiler directives and last but not least, templates.

Beckmann et al. [13] rely on run-time code generation using partial evaluation techniques for numerical and image processing applications. It is interesting to see that the TaskGraph library described in that paper follows a semi-type-safe way to do multi-stage programming in C++. That particular line of work inspires Cohen et al. [35] to perform loop transformations in a generative manner with proper multi-stage programming in MetaOCaml. The heavy use of code combinators in Cohen et al. resembles the implementation of the operators of Strymonas. Ofenbeck et al. also implement a subset of Spiral in Scala/LMS [123]. The last two pieces of related work demonstrate the trend to experiment with more type-safe styles of partial evaluation and more specifically with multi-stage programming.

MetaOCaml treats code as data in an opaque form, thus the generated code is treated as a black box. As a result, it cannot be examined and optimized further in MetaOCaml (a realization also pointed out in both Cohen et al. [35] and Spiral in Scala/LMS [123]). In the latter, the authors create a multi-level DSL by relying on the extensible IR feature of Scala/LMS. By having an extensible IR at our dispose the code is not a black box any more: the programmer can perform a number of flexible transformations such as: common subexpression elimination, dead-code elimination and various forms of code motion. Since MetaOCaml does not support an extensible IR, we handle explicitly all code manipulations with combinators. This puts an extra responsibility on the library author, but it makes our library portable on both MetaOCaml and Scala LMS.

One of the earliest stream libraries that rely on staging, is Common Lisp's SERIES [186, 187], which extensively uses Lisp macros to interpret a subset of Lisp code as a stream EDSL. It builds a data flow graph and then compiles it into a single loop. It can handle filtering, multiple producers and consumers, but not nested streams. The (over)reliance on macros may lead to surprises since the programmer might not be aware that what looks like CL code is actually a DSL, with a slightly different semantics and syntax. An experimental Pipes package [87] attempts to re-implement and extend SERIES, using, this time, a proper EDSL. Pipes extends SERIES by allowing nesting, but restricts zipping to simple cases. It was posited that "arbitrary outputs per input, multiple consumers, multiple producers: choose two" [87]. Pipes "almost manages" (according to its author) to implement

all three features. Our library demonstrates the conjecture is false by supporting all three facilities in full generality and with high performance.

Jonnalagedda et al. present a library using only CPS encodings (fold-based) [82]. It uses the Gill et al. foldr/build technique [58] to get staged streams in Scala. Like foldr/build, it does not support operators with multiple inputs such as `zip`.

In our work, we employ the traditional MSP model to implement a performant streaming library. Rompf et al. [146] demonstrate a loop fusion and deforestation algorithm for data parallel loops and traversals. They use staging as a compiler transformation pass and apply it to query processing for in-memory objects. That technique lacks the rich range of fused operators over finite or infinite sources that we support, but seems adequate for the case studies presented in that work. Porting our technique from the staged-library level to the compiler-transformation level may be applicable in the context of Scala/LMS.

Ziria [162], a language for wireless systems' programming, compiles high-level reconfigurable data-flow programs to vectorized, fused C-code. Ziria's `tick` and `process` (pull and push respectively) demonstrate the benefits of having both processing styles in the same library. It would be interesting to combine our general-purpose stream library with Ziria's generation of vectorized C code.

7.3.3 Query Engine Optimizations

Streaming libraries and query engines in database systems share many ideas. Both areas offer a high level API, describing a low-level operation over data that needs to be executed efficiently. Abstractly, a query expression is typically translated by a query engine into a physical plan of execution. These plans consist of (physical) operators that consume data stored in tables, in a streaming fashion, and produce other streams (streams of tuples in the DBMS nomenclature). A database query optimizer performs a cost analysis to calculate the most efficient physical scan. As a result, a physical plan is the output of the optimizer and it shares the same goals as a pipeline in streaming libraries, namely traversing data efficiently without producing intermediate values.

The Volcano model [61, 62] implements the iterator model (pull-based), which prevailed among the proposed solutions at the time. Operators are equipped with a `next` function, which is repeatedly called for traversal. Thomas Neumann [120] identified three problems with the iterator model. First, the `next` function in iterator-based models, is called for every tuple. Second, each call raises a performance hit, of one virtual call per element, usually accompanied by branch prediction degradation. Finally, this model promotes poor code locality. Neumann asserts that modern solutions are able to produce more than one tuple during each `next` call (vectorized processing). Neumann, also identifies the problem of an operator producing more data per application ("this pure pipelining usually cannot be used any more, as the tuples have to be materialized somewhere to be accessible") and we believe that this is closely related to `flatMap` in push-based streams (demonstrated by the cartesian product benchmark in Chapters 3 and 4).

Neumann presents HyPer, which departs from in-memory operations and relies on compiling a query plan to imperative code, following a generative approach. This shift is also evident in streaming libraries as we have seen in this dissertation. We have also employed it in Strymonas and it is on the rise in query engine implementations [94–96].

Arumugam et al. present a pure-push approach in Datapath [7]. Early on, in their paper, we can observe the push-based strategy they have adopted and the departure of query engines from iterator-based models. This is a similar quality that Java 8 Streams seek to reach with bulk-data processing, as we have seen in Chapter 3 (recall the `forEachRemaining` method in Java 8 Streams).

A very promising direction presented by Klonatos et al. [93] bridges the generative approach of building query engines with metaprogramming languages–in particular, Scala and LMS. Interestingly, in the Klonatos et al. work, there exists a mechanical way to obtain a push-based engine from a pull-based one. The two models are investigated, through the lens of query engines, by Shaikhha et al. [155]. It is not surprising the push-model in query engines suffers from inefficiencies with short-ranging and parallel traversals, namely of `limit` and `merge-join` operators. For this reason, since the design of Strymonas of Chapter 6 overcomes the aforementioned limitations, we believe that it is applicable to the domain of query engine compilation.

Finally, Spark [4, 193] an open-source cluster-computing framework, in its second major iteration, improves upon the query engine for SQL and DataFrames (an API for the processing of distributed collections of data) by implementing “whole stage code generation” [2]. Whole-stage code generation is inspired by HyPer [120]. While that shift is related to modern stream fusion techniques and our Strymonas library (Chapter 6), StreamAlg’s extensible library of Chapter 4 could also be highly applicable in the context of either DBMS or distributed frameworks, offering a degree of freedom in the choice of query engines.

8. CONCLUSIONS

Having presented the results of this dissertation, this final chapter assesses our initial thesis, concludes and gives pointers for future work.

StreamAlg, presented in Chapter 4, demonstrates a design that enhances streams with pluggable semantics and operators. Chapter 3 helped us prepare the ground for StreamAlg by giving us a better understanding of the various stream semantics. For example, StreamAlg can support a push-based or a pull-based pipeline construction. Furthermore, pluggable optimizations, tracing, logging, blocking or non-blocking terminal operators are only some of the interpretations that can be plugged in without the need to recompile the code. StreamAlg decouples the semantics from the syntax of the domain-specific language of streams, effectively performing the first step towards modularization. The proposed design has been implemented in Java for exposition but it is easily ported in other programming languages. In this dissertation it is evident that there are cases in streams that benefit from modular internals, most importantly from push-vs-pull pluggability, since that was our motivating case study.

Object Algebras have proven a powerful design pattern in two cases through this dissertation. StreamAlg has been the first case and Recaf, presented later in Chapter 5, the second. Recaf demonstrates the general applicability of the pattern, virtualizing Java in a lightweight manner. We were able to extend streams at the language level by creating a library using a newly introduced keyword, `yield`. It is one of the many examples and case studies we present in Chapter 5.

Finally, Chapter 6 presented a new library design for fused streams called Strymonas. We described a library design for streams with the fundamental set of operators that exist in nearly any mainstream library. Our library defers the compilation of a pipeline until runtime when all necessary information exists, such as all parameters and the actual sequence of operators. Strymonas removes all constant overheads from the code (e.g., eliminates the creation of tuples, fuses loops that are possibly nested, nonlinear, or parallel, inlines lambdas and more) and generates code that runs faster than mainstream VM-based Stream APIs on a single-core. The whole philosophy of Strymonas is based on MSP, a modern method of doing partial evaluation in a type-safe and controlled manner. However we discovered in the process that MSP is not merely a sprinkling of `code` or `Rep` types to do simple manual binding-time analysis, as has been advertised through the literature. Instead, this library has been carefully developed with multi-stage programming in mind, from the beginning. As a result we reused methodologies from the partial evaluation community, developed long before MSP, in every iteration of the development process.

Ward Cunningham refers to “sufficiently-smart” (static or runtime) compilers as a myth [1]. Indeed, trying to align an implementation of a streaming library with the generality of the Hotspot compiler(s) is a highly specialized and technically demanding task and it binds the

[1] Sufficiently Smart Compiler–`https://web.archive.org/web/20170322011429/http://wiki.c2.com/?SufficientlySmartCompiler`

implementations with the JVM itself. Furthermore, the domain semantics are impossible to be expressed in the generic virtual machine (e.g., reified loops and code that must be inlined) and the domain-expert knows more about streams than the compiler (e.g., pipelines are going to be executed many times, a zip can encode reified loops and a cartesian product must perform nested loop fusion with stack-allocated data).

In some other occasions we have direct access to the compilation process where rewriting rules are used to remove abstraction artifacts and rewrite the programs in a semantically preserving way. As a result, as in the case of GHC, the design of a library is tightly coupled with the declared rules and vice versa. Instead of relying on a series of interventions, by the library author to the library indirectly, we propose a solution to keep the design and implementation only at the library side.

Summarizing, we improve streams in terms of extensibility and performance and with the mechanisms we proposed we enhance them without breaking their high level structure. In this dissertation we treat interpretations and optimizations as pluggable components and we advocate that domain-specific optimizations must be developed in "active" Stream APIs instead of "sufficiently-smart compilers".

8.1 Limitations & Future Work

This last section summarizes the limitations and future work directions from our work.

8.1.1 Improving Recaf

Recaf offers a lightweight, yet disciplined, approach in Java to create dialects as libraries. Additionally it has been successful in capturing a lot of extensibility scenarios through small examples and case studies: shallow embedding, deep embedding, aspect-oriented scenarios and more. However, examples such as the streaming library which uses a new extension `yield!`, expose the interpretive overhead that occurs when the program is executed—especially in CPS semantics with the use of exceptions for control flow. One direction to improve runtime execution of transformed programs would be to investigate "compiler algebras", which generate byte code (or even native code) at runtime. Frameworks such as ASM [21] and Javassist [33] could be used to dynamically generate byte-code, which could then be executed by the `Method` method.

In addition, we can use Recaf's expression virtualization as a domain specific language for code generation. Let's take Javassist [33] for example; the snippet of code in Figure 8.1 creates a method `specialM` in a specialized class (with the same name as shown below), assuming that the method exists in the superclass. All `CT*` types are part of Javassist's API. In this example, we are implementing a particular specialization according to the characteristics of some method `m` (its definition is not shown in the code).

Note that the body of the specialized method is created as a string. By using Recaf's

```
String body = "{
  System.out.println(\"Logging\");
  return super.${m}($$);
}";

CtMethod specialM = CtNewMethod.make(
    m.getReturnType(), m.getName(), m.getParameterTypes(), m.getExceptionTypes(),
    body.replaceAll(Pattern.quote("${m}"), m.getName()), specialized);

specialized.addMethod(specialM);
```

Figure 8.1: Use Recaf as a DSL for Javassist

support, and different types of annotations that capture a creation pattern, we get a full-blown DSL for programming generators.

A practical limitation of Recaf is that it does not let the extension author redefine the class and interface definitions. Further development to support virtualization at the level of classes will complete the slate of use cases for the project and provide fine-grained intercession. We can consult the design decisions behind JavaScript 6 [42] based on the work of Cutsem et al [177] on Proxies and provide a complete meta-object protocol for Java classes. Ureche et al. [176] present an automated and composable mechanism that allows programmers to change the data layout in delimited scopes at the level of both expressions and class definitions. For example a complex number can be represented as a class with two fields or a class with one long integer. However the decision to change the layout of our data transparently is orthogonal to the design of an algorithm (GCD). By extending Recaf on the class/interface declaration level we could support data layout transformations. As a consequence, we can explore richer metaprogramming patterns in general, as in the *The Art of Metaobject Protocol* [88].

In this dissertation we have used two popular MSP facilities for OCaml and Scala. Mint implemented an MSP facility for Java with strong type safety guarantees in the presence of effects and other OOP features [189]. Although it follows a conservative view on quoted code, limiting expressivity, it eliminates all the dangers of scope-extrusion (the problem where code containing variables is used out of its binder's scope) . Since Recaf provides a virtualized view of Java at the method level for both statements and expressions, it can lay the necessary foundations for an MSP facility in Java along the lines of LMS and even enable type safety guarantees for the generated code.

Additionally, since non-determinism or asynchrony does not strictly need CPS in the general case, we can investigate whether Recaf can be used with alternative forms of extensibility such as free monads or others [135, 167].

We strongly believe that Recaf can be a viable platform for Java language extensibility. This will require a series of improvements on the performance of the Recaf generator

itself, on coverage of Java semantics and on debugging support. On top of these improvements, package management support can simplify the development process and will make importing of language extensions a mere use of a command, e.g. `recaf install backtracking coroutines`.

8.1.2 Enriching the API of Strymonas

In Strymonas we have implemented the most common set of stream operators. We can easily add `dropWhile` (`takeWhile` is already expressible through the generalized `take`). We leave for future work operators that involve the production of two or more streams, such as `partition`, `split` and `unzip`–as well as `groupBy` The latter requires not only stream fusion but also high-performance hash-maps.

8.1.3 IO/side effects inside unfold

Supporting value reuse in streams is orthogonal to the use of staging, fusion and optimizations. We have adopted the same philosophy (in terms of target uses) as Java 8 streams. In the following code, `chars` is an infinite stream of characters read from the command line. The user enters the numbers $1-4$ (and presses enter two times). This effectful operation makes clear why the stream is not reused; the `zip_with` operation accepts the `same` stream twice. As a result, the resulting tuples `(("1", "2"), ("3", "4"), ...)` pair (successive) values from the same stream and not from a forked stream (for each consumer) as the programmer might desire `(("1", "1"), ("2", "2"), ...)`.[2]

```
let chars = unfold (fun n →
  .⟨ if (∼(n) = "0")
     then None
     else Some (∼(n), ∼(.⟨read_line()⟩.)) ⟩.) .⟨""⟩.;;

let test = zip_with (fun e1 e2 → .⟨(∼e1, ∼e2)⟩.) chars chars
           ▷ fold (fun z a → .⟨∼a :: ∼z⟩.) .⟨[]⟩.
           ▷ (fun l → .⟨List.rev ∼l⟩.);;
```

Figure 8.2: Reusing streams (forking)

The way to support that use case would be to cache the producer with a `.cache` operator as F# offers, or use some other buffering mechanism.

This particular case originates historically from data-flow programming and nowadays is met usually in asynchronous or reactive scenarios at different rates of consumption. Lipp-

[2] Amos Robinson, UNSW (Sydney, Australia),"Stream fusion, to completeness: using a stream multiple times", e-mail to author, 6 Jan 2017.

meier et al. [101] present the Repa Flow Haskell library evolved from the ”Data Flow Fusion with Series Expressions in Haskell” paper [102] which targets data-flow use cases inspired by pipelines that involve delays, clocks and rates. Although *streams* and *flows* are two different representations, we can attempt to draw parallels with the aforementioned use case. In Lippmeier et al. [101] a `dup` operator takes two argument sink endpoints and returns one sink that will push an element to the two argument sinks, effectively branching the stream. Since we encode pushing of values using a pull-based basis we could implement a `dup` operator as well with reasonable effort.

8.1.4 Exploring Link-Time Optimizations for Streams

With StreamAlg we were able to control the performance of streams, manually, by plugging-in the most appropriate implementation of an object algebra. What if we could exploit use-site knowledge of stream pipelines, automatically, and discover which implementation of the semantics is the most appropriate to use? An optimizing compiler with link-time optimization capabilities may be suitable for such an exploration. Renucci [141] automatically selects the most suitable collection implementation, exploiting use-site knowledge in Scala, using Dotty [40] and call-graph construction [129]. Dotty embodies many intuitions gained by the DOT calculus [3], a minimal formal foundation for Scala. Finally, the same system could be used to implement key concepts developed in Strymonas, as a set of link-time optimizations in Scala.

8.1.5 Towards High-Performance Computing

Strymonas has lowered the abstraction overheads by promoting expressivity, it establishes a first step towards high performance computing. The techniques of Strymonas revolve around single-processor/sequential optimizations by eliminating constant factors that slow down stream pipelines. As a next step, Strymonas can attempt to utilize SIMD instruction sets for data parallel streams, again through staging. Cohen et al. [35] implement loop-based transformations such as interchange, fusion, fusion and shifting through MetaOCaml. Consequently it has been demonstrated that low-level optimizations are feasible with multi-stage programming. Our goal will be to explore the vectorization opportunities that functional patterns expose, for both single-dimensional but also multi-dimensional arrays (again in OCaml and Scala/JVM), efficiently utilizing cache lines for computations.

This research activity can enhance Strymonas with multi-dimensional arrays and can apply techniques used to write fast numerical code as in Spiral [140], ArrayFire [107] for CPU, GPU, FPGA accelerators, Ziria [162] (a DSL for programming software defined radios), Accelerate [30, 104] (a deeply embedded DSL for HPC implemented in Haskell). The designers of the JVM are also leaning towards this direction; bringing the JVM closer to the bare metal and native code [150]. Most importantly, emphasis is given to `Vector` shapes that will form the bases of data types providing implicit access to SIMD utilization [149].

Finally, Strymonas can be used to create distributed, data-parallel processing pipelines

as in FlumeJava [31]. FlumeJava is an API for distributed streams, built on top of MapReduce, with emphasis in fusion. FlumeJava's `ParallelDo` *producer-consumer fusion* is one interesting direction that would require the addition of higher-order abstract syntax to our stream language, to designate sharing in a type safe manner.

We hope to explore some of these directions in our future work, or to inspire others to do so.

ABBREVIATIONS - ACRONYMS

ABRV	DESCRIPTION
ADT	Algebraic Data Type
AEOS	Automated Empirical Optimization of Software
API	Application Programming Interface
AST	Abstract Syntax Tree
BLAS	Basic Linear Algebra Subroutines
CPS	Continuation Passing Style
CSP	Cross-Stage Persistence
DBMS	Data Base Management System
DSL	Domain Specific Language
EP	Expression Problem
FPGA	Field-Programmable Gate Array
GADT	Generalized Algebraic Data Type
GC	Garbage Collector
GHC	Glasgow Haskell Compiler
HOAS	Higher-Order Abstract Syntax
HPC	High Performance Computing
IR	Intermediate Representation
JIT	Just-In Time
JMH	Java Microbenchmark Harness
JVM	Java Virtual Machine
LINQ	Language Integrated Query
LMS	Lightweight Modular Staging
MSP	Multi-Stage Programming
SIMD	Single Instruction Multiple Data
SSA	Static Single Assignment

APPENDIX A. APPENDIX TO CHAPTER 5

A.1 Direct Style Interpreter for μJava

```
public interface MuJavaBase<R> extends MuJava<R, IExec> {
  default IExec Exp(Supplier<Void> e) {
   return () → { e.get(); };
  }

  default <T> IExec Decl(Supplier<T> e, Function<T, IExec> s) {
   return () → s.apply(e.get()).exec();
  }

  default <T> IExec For(Supplier<Iterable<T>> e, Function<T, IExec> s) {
    return () → { for (T t: e.get()) s.apply(t).exec(); };
  }

  default IExec If(Supplier<Boolean> c, IExec s1, IExec s2) {
   return () → {
      if (c.get()) s1.exec();
      else s2.exec(); };
  }

  default IExec Return(Supplier<R> e) {
   return () → { throw new Return(e.get()); };
  }

  default IExec Seq(IExec s1, IExec s2) {
   return () → { s1.exec(); s2.exec(); };
  }

  default IExec Empty() {
   return () → {};
  }
}
```

A.2 CPS Interpreter for μJava

```
public interface MuJavaCPS<R> extends MuJava<R, SD<R>> {
  default SD<R> Exp(Supplier<Void> e) {
```

```
    return (r, s) → { e.get(); s.call(); };
  }

  default <T> SD<R> Decl(Supplier<T> e, Function<T, SD<R>> s) {
    return (r, s0) → s.apply(e.get()).accept(r, s0);
  }

  default <T> SD<R> For(Supplier<Iterable<T>> e, Function<T, SD<R>> s) {
    return (r, s0) → {
      Iterator<T> iter = e.get().iterator();
        new K0() {
          public void call() {
            if (iter.hasNext())
              s.apply(iter.next()).accept(r, () → call());
            else
              s0.call();
        }
      }.call();
     };
  }

  default SD<R> If(Supplier<Boolean> c, SD<R> s1, SD<R> s2) {
      return (r, s) → {
        if (c.get()) s1.accept(r, s);
        else s2.accept(r, s);};
  }

  default SD<R> Return(Supplier<R> e) {
    return (r, s) → r.accept(e.get());
  }

  default SD<R> Seq(SD<R> s1, SD<R> s2) {
    return (r, s) → s1.accept(r, () → s2.accept(r, s));
  }

  default SD<R> Empty() {
   return (r, s) → s.call();
  }
}
```

A.3 Translation Rules to Support Expression Virtualization

$$
\begin{array}{rcl}
\mathcal{S}_a[\![\texttt{;}]\!] & = & a\texttt{.Exp(}\mathcal{E}_a[\![e]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{if (}e\texttt{)}\ S_1\ \texttt{else}\ S_2\ ]\!] & = & a\texttt{.If(}\mathcal{E}_a[\![e]\!]\texttt{,}\ \mathcal{S}_a[\![S_1]\!]\texttt{,}\mathcal{S}_a[\![S_2]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{for(}T\ x\texttt{:}\ e\texttt{)}\ S]\!] & = & a\texttt{.For(}\mathcal{E}_a[\![e]\!]\texttt{,(}T\ x\texttt{)}\ \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![T\ x\ \texttt{=}\ e\texttt{;}\ S]\!] & = & a\texttt{.Decl(}\mathcal{E}_a[\![e]\!]\texttt{,(}T\ x\texttt{)}\ \rightarrow \mathcal{S}_a[\![S]\!]\texttt{)} \\
\mathcal{S}_a[\![\texttt{return}\ e\texttt{;}]\!] & = & a\texttt{.Return(}\mathcal{E}_a[\![e]\!]\texttt{)} \\
 & & \\
\mathcal{E}_a[\![v]\!] & = & a\texttt{.Lit(}v\texttt{)} \\
\mathcal{E}_a[\![x]\!] & = & a\texttt{.Var("}x\texttt{",}x\texttt{)} \\
\mathcal{E}_a[\![\texttt{this}]\!] & = & a\texttt{.This(this)} \\
\mathcal{E}_a[\![e\texttt{.}x]\!] & = & a\texttt{.Field(}\mathcal{E}_a[\![e]\!]\texttt{,"}x\texttt{")} \\
\mathcal{E}_a[\![e\texttt{.}x\texttt{(}e_1,...,e_n\texttt{)}]\!] & = & a\texttt{.Invoke(}\mathcal{E}_a[\![e]\!]\texttt{,"}x\texttt{",}\mathcal{E}_a[\![e_1]\!],...,\mathcal{E}_a[\![e_n]\!]\texttt{)} \\
\mathcal{E}_a[\![\texttt{new}\ T\texttt{(}e_1,...,e_n\texttt{)}]\!] & = & a\texttt{.New(}T\texttt{.class,}\mathcal{E}_a[\![e_1]\!],...,\mathcal{E}_a[\![e_n]\!]\texttt{)} \\
\mathcal{E}_a[\![\texttt{(}x_1,...,x_n\texttt{)}\rightarrow S]\!] & = & a\texttt{.Lambda((}x_1,...,x_n\texttt{)}\rightarrow\mathcal{S}_a[\![S]\!]\texttt{)}
\end{array}
$$

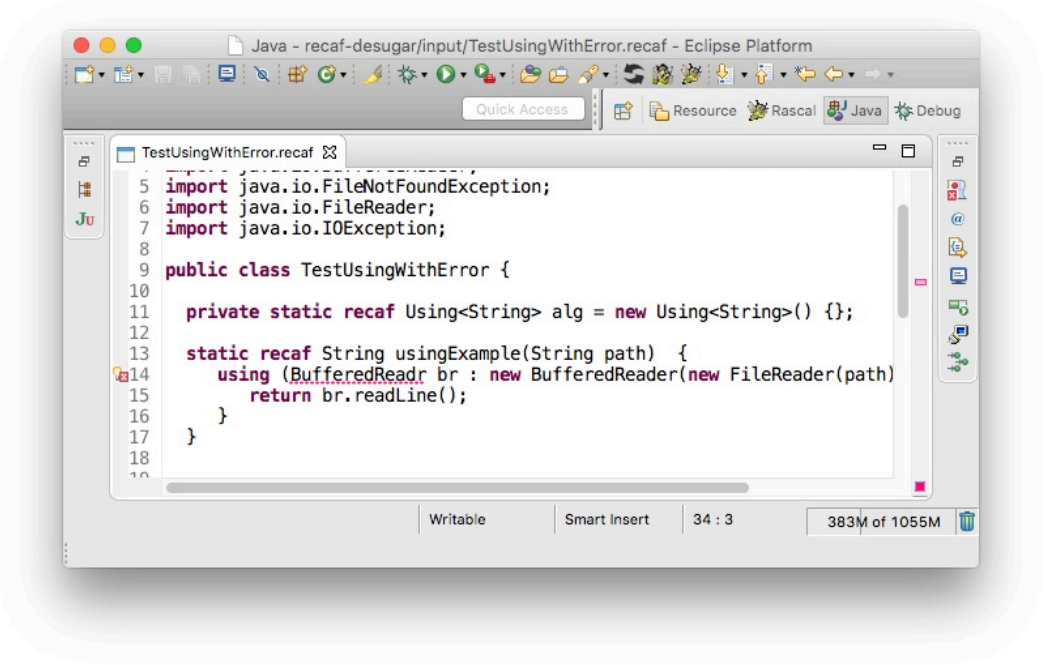
A.4 Interpreter for μJava Expressions

```
public interface MuExpJavaBase extends MuExpJava<IEval> {
  @Override
  default IEval Lit(Object x) {
    return () → x;
  }

  @Override
  default IEval This(Object x) {
    return () → x;
  }

  @Override
  default IEval Field(IEval x, String f) {
    return () → {
     Object o = x.eval();
     Class<?> clazz = o.getClass();
     try {
       return clazz.getField(f).get(o);
     } catch (Exception e) {
       throw new RuntimeException(e);
     }
   };
  }

  static Object[] evalArgs(IEval[] es)  {
    Object[] args = new Object[es.length];
```

```
      for (int i = 0; i < es.length; i++)
        args[i] = es[i].eval();
        return args;
  }

  @Override
  default IEval New(Class<?> c, IEval... es) {
    return () → {
     Object[] args = evalArgs(es);
      for (Constructor<?> cons : c.getDeclaredConstructors())
       try {
        return cons.newInstance(args);
      } catch (Exception e) {
          continue;
      }
      throw new RuntimeException("no constructor");
   };
  }

  @Override
  default IEval Invoke(IEval x, String m, IEval... es) {
    return () → {
      Object recv = x.eval();
      Object[] args = evalArgs(es);
      for (Method method: recv.getClass().getMethods())
       if (m.equals(method.getName()))
        try {
            return method.invoke(recv, args);
          } catch (Exception e) {
            continue;
          }
          throw new RuntimeException("no such method " + m);
    };
  }

  @Override
  default IEval Lambda(Object f) {
    return () → f;
  }

  @Override
  default IEval Var(String x, Object it) {
    return () → it;
  }
```

```
}
```

A.5 Screenshot of Recaf IDE



A.6 Generated code for Pull Streams case study of Chapter 5.5.2

```
package generated;

import recaf.demo.cps.Iter;
import java.util.stream.IntStream;
import java.util.Iterator;
import java.util.function.Predicate;
import java.util.function.*;
import java.util.NoSuchElementException;

public class PStream {
  public PStream(Iterable<Integer> source) {
    this.source = source;
  }

  Iterable<Integer> source;

  public static PStream range(int n) {
```

```
    return new PStream(rangeIter(n));
  }

  private static Iterable<Integer> rangeIter(int n) {
      return new Iterable<Integer>() {
      public Iterator<Integer> iterator() {
        return IntStream.range(0, n).iterator();
       }
    };
  }

  private Iter<Integer> alg = new Iter<Integer>();

  public PStream map(Function<Integer, Integer> f) {
    return new PStream(mapIter(this.source, f));
  }

  Iterable<Integer> mapIter(Iterable<Integer> source,
                            Function<Integer, Integer> f) {
    recaf.core.Ref<Iterable<Integer>> $source =
      new recaf.core.Ref<Iterable<Integer>>(source);
    recaf.core.Ref<Function<Integer, Integer>> $f =
      new recaf.core.Ref<Function<Integer, Integer>>(f);
    return (Iterable<Integer>)
      alg.Method(
        alg.<Integer>ForEach(
          () → $source.value,
          (recaf.core.Ref<Integer> t) →
             alg.Yield(() → $f.value.apply(t.value))));
  }

  public Integer sum() {
    return sumIter(this.source);
  }

  private Integer sumIter(Iterable<Integer> source) {
    Integer acc = 0;
      for (Integer t: source) {
       acc+=t;
     }
     return acc;
  }
}
```

APPENDIX B. APPENDIX TO CHAPTER 6

B.1 Generated code for the Complex example of Chapter 6

We show the generated code for the last example of Section §6.3, repeated below for reference:

```
(* Zipping function *)
zip_with (fun e1 e2 → .⟨(∼e1,∼e2)⟩.)
 (* First stream to zip *)
 (of_arr .⟨arr1⟩.
   ▷ map (fun x → .⟨∼x * ∼x⟩.)
   ▷ take .⟨12⟩.
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨∼x * ∼x⟩.))
 (* Second stream to zip *)
 (iota .⟨1⟩.
   ▷ flat_map (fun x → iota .⟨∼x+ 1⟩. ▷ take .⟨3⟩.)
   ▷ filter (fun x → .⟨∼x mod 2 = 0⟩.))
 ▷ fold (fun z a → .⟨∼a :: ∼z⟩.) .⟨[]⟩.
```

The generated code is:

```
let s_23 = ref [] in
let arr_24 = arr1 in
 let i_25 = ref 0 in
 let curr_26 = ref None in
 let nadv_27 = ref None in
 let adv_32 () =
   curr_26 := None;
   while
     ((! curr_26) = None) &&
       ((! nadv_27 ≠  None) ||
          (! i_25 ≤ (min (12 − 1) (Array.length arr_24 − 1))))
     do
     match ! nadv_27 with
      | Some adv_28 → adv_28 ()
      | None  →
          let el_29 = arr_24.(! i_25) in
          let t_30 = el_29 * el_29 in
          incr i_25;
          if (t_30 mod 2) = 0
          then let t_31 = t_30 * t_30 in
```

```
                curr_26 := Some t_31
     done in
 adv_32 ();
 let s_33 = ref (Some (1, (1 + 1))) in
 let term1r_34 = ref (! curr_26 ≠ None) in
 while ! term1r_34 && ! s_33 ≠ None do
    match ! s_33 with
    | Some (el_35,s'_36) →
        s_33 := (Some (s'_36, (s'_36 + 1)));
        let s_37 =
           ref (Some (el_35 + 1, (el_35 + 1) + 1)) in
        let nr_38 = ref 3 in
        while (! term1r_34) &&
              (((! nr_38) > 0) && ((! s_37) ≠ None)) do
          match ! s_37 with
          | Some (el_39,s'_40) →
              s_37 := Some (s'_40, (s'_40 + 1));
              decr nr_38;
              if el_39 mod 2 = 0
              then
                (match ! curr_26 with
                 | Some el_41 →
                     adv_32 ();
                     term1r_34 := !curr_26 ≠ None;
                     s_23 := (el_41, el_39) :: ! s_23)
          done
    done;
! s_23
```

B.2 Cartesian Product

```
let cart = fun (arr1, arr2) →
  ofArr arr1
  ▷ flat_map (fun x →
      ofArr arr2 ▷ map (fun y → .⟨ ~x * ~y⟩.))
  ▷ fold (fun z a → .⟨~z + ~a⟩.) .⟨0⟩.;;
```

B.3 Generated code for Cartesian Product

```
let x = Array.init 1000 (fun i_1  → i_1) in
let y = Array.init 10   (fun i_2  → i_2) in
let arr_1 = x in
let size_1 = Array.length arr_1 in
let iarr_1 = ref 0 in
let rec loop_1 acc_1 =
  if (!iarr_1) ≥ size_1
  then acc_1
  else
    (let el_1 = arr_1.(!iarr_1) in
     incr iarr_117;
     (let acc1_tmp =
        let arr_2 = y in
        let size_2 = Array.length arr_2 in
        let iarr_2 = ref 0 in
        let rec loop_2 acc_2 =
          if (!iarr_2) ≥ size_2
          then acc_2
          else
            (let el_2 = arr_2.(!iarr_2) in
             incr iarr_2;
             (let acc2_tmp =
                acc_2 + (el_1  * el_2) in
              loop_2 acc2_tmp)) in
        loop_2 acc_1 in
      loop_1 acc1_tmp)) in
loop_1 0
```

B.4 Streams and baseline benchmarks

```
let sumS
= fun arr →
   of_arr arr
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let sumShand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 − 1 do
      sum := !sum + (∼arr1).(counter1);
   done;
```

```
    !sum ⟩.;;

let sumOfSquaresS
= fun arr →
    of_arr arr
    ▷ map (fun x → .⟨∼x * ∼x⟩.)
    ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let sumOfSquaresShand
= fun arr1 → .⟨
    let sum = ref 0 in
    for counter1 = 0 to Array.length ∼arr1 − 1 do
     let item1 = (∼arr1).(counter1) in
     sum := !sum + item1*item1;
    done;
    !sum⟩.;;

let mapsS
= fun arr →
    of_arr arr
    ▷ map (fun x → .⟨∼x * 1⟩.)
    ▷ map (fun x → .⟨∼x * 2⟩.)
    ▷ map (fun x → .⟨∼x * 3⟩.)
    ▷ map (fun x → .⟨∼x * 4⟩.)
    ▷ map (fun x → .⟨∼x * 5⟩.)
    ▷ map (fun x → .⟨∼x * 6⟩.)
    ▷ map (fun x → .⟨∼x * 7⟩.)
    ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let maps_hand
= fun arr1 → .⟨
    let sum = ref 0 in
    for counter1 = 0 to Array.length ∼arr1 − 1 do
    let item1 = (∼arr1).(counter1) in
     sum := !sum + item1*1*2*3*4*5*6*7;
    done;
    !sum⟩.;;

let filtersS
= fun arr →
    of_arr arr
    ▷ filter (fun x → .⟨∼x > 1⟩.)
    ▷ filter (fun x → .⟨∼x > 2⟩.)
    ▷ filter (fun x → .⟨∼x > 3⟩.)
```

```
   ▷ filter (fun x → .⟨~x > 4⟩.)
   ▷ filter (fun x → .⟨~x > 5⟩.)
   ▷ filter (fun x → .⟨~x > 6⟩.)
   ▷ filter (fun x → .⟨~x > 7⟩.)
   ▷ fold (fun z a → .⟨~z + ~a⟩.) .⟨0⟩.;;

let filters_hand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ~arr1 - 1 do
      let item1 = (~arr1).(counter1) in
      if (item1 > 1 && item1 > 2 && item1 > 3 &&
        item1 > 4 && item1 > 5 && item1 > 6 &&
        item1 > 7) then
      begin
      sum := !sum + item1;
      end;
   done;
   !sum⟩.;;

let sumOfSquaresEvenS
= fun arr →
   of_arr arr
   ▷ filter (fun x → .⟨~x mod 2 = 0⟩.)
   ▷ map (fun x → .⟨~x * ~x⟩.)
   ▷ fold (fun z a → .⟨~z + ~a⟩.) .⟨0⟩.;;

let sumOfSquaresEvenShand
= fun arr1 → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ~arr1 - 1 do
   let item1 = (~arr1).(counter1) in
   if item1 mod 2 = 0 then
   begin
     sum := !sum + item1*item1
   end;
   done;
   !sum⟩.;;

let cartS
= fun (arr1, arr2) →
   of_arr arr1
   ▷ flat_map (fun x →
      of_arr arr2 ▷ map (fun y → .⟨ ~x * ~y⟩.))
```

```
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let cartShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 − 1 do
      let item1 = (∼arr1).(counter1) in
      for counter2 = 0 to Array.length ∼arr2 − 1 do
         let item2 = (∼arr2).(counter2) in
         sum := !sum + item1 * item2;
      done;
   done;
   !sum ⟩.;;

let dotProductS
= fun (arr1, arr2) →
   zip_with (fun e1 e2 → .⟨∼e1 * ∼e2⟩.)
            (of_arr arr1) (of_arr arr2)
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let dotProductShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter = 0 to
         min (Array.length ∼arr1)
             (Array.length ∼arr2) − 1 do
      let item1 = (∼arr1).(counter) in
      let item2 = (∼arr2).(counter) in
      sum := !sum + item1 * item2;
   done;
  !sum⟩.;;

let flatMap_after_zipWithS
= fun (arr1, arr2) →
   zip_with (fun e1 e2 → .⟨∼e1 + ∼e2⟩.)
            (of_arr arr1) (of_arr arr1)
   ▷ flat_map (fun x → of_arr arr2
         ▷ map (fun el → .⟨∼el + ∼x⟩.))
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let flatMap_after_zipWithShand
= fun (arr1, arr2) → .⟨
   let sum = ref 0 in
   for counter1 = 0 to Array.length ∼arr1 − 1 do
```

```
      let x = (~arr1).(counter1)
              + (~arr1).(counter1) in
      for counter2 = 0 to Array.length ~arr2 - 1 do
      let item2 = (~arr2).(counter2) in
        sum := !sum + item2 + x;
      done;
    done;
    !sum⟩.;;

let zipWith_after_flatMapS
= fun (arr1, arr2) →
    of_arr arr1
    ▷ flat_map (fun x →
       of_arr arr2 ▷ map (fun y → .⟨~y + ~x⟩.))
    ▷ zip_with (fun e1 e2 → .⟨~e1 + ~e2⟩.)
                (of_arr arr1)
    ▷ fold (fun z a → .⟨~z + ~a⟩.) .⟨0⟩.;;

let zipWith_after_flatMapShand
= fun (arr1, arr2) → .⟨
    let sum = ref 0 in
    let i1 = ref 0 in
    let i2 = ref 0 in
    let flag1 = ref ((!i1) ≤ ((Array.length ~arr1) - 1)) in
    while (!flag1) && ((!i2) ≤ ((Array.length ~arr2) - 1)) do
       let el2 = (~arr2).(!i2) in
       incr i2;
       (let i_zip = ref 0 in
        while (!flag1) &&
              ((!i_zip) ≤ ((Array.length ~arr1) - 1)) do
          let el1 = (~arr1).(!i_zip) in
          incr i_zip;
          let elz = (~arr1).(!i1) in
          incr i1;
          flag1 := ((!i1) ≤ ((Array.length ~arr1) - 1));
          sum := ((!sum) + (elz + el1 + el2))
          done)
       done;
    !sum⟩.;;

let flat_map_takeS
= fun (arr1, arr2) →
    of_arr arr1
    ▷ flat_map (fun x → of_arr arr2
```

```
      ▷ map (fun y → .⟨ ∼x * ∼y⟩.))
   ▷ take .⟨20000000⟩.
   ▷ fold (fun z a → .⟨∼z + ∼a⟩.) .⟨0⟩.;;

let flat_map_takeShand
= fun (arr1, arr2) → .⟨
   let counter1 = ref 0 in
   let counter2 = ref 0 in
   let sum = ref 0 in
   let n = ref 0 in
   let flag = ref true in
   let size1 = Array.length ∼arr1 in
   let size2 = Array.length ∼arr2 in
   while !counter1 < size1 && !flag do
      let item1 = (∼arr1).(!counter1) in
      while !counter2 < size2 && !flag do
         let item2 = (∼arr2).(!counter2) in
         sum := !sum + item1 * item2;
         counter2 := !counter2 + 1;
         n := !n + 1;
         if !n = 20000000 then
         flag := false
      done;
      counter2 := 0;
      counter1 := !counter1 + 1;
   done;
   !sum ⟩.;;
```

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. Ed. by MIT Press. 1985.

[2] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. https://web.archive.org/web/20170322191830/https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html. May 2016.

[3] Nada Amin et al. "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science. Edinburgh, UK: Springer International Publishing, 2016, pp. 249–272. url: http://wadlerfest.namin.net/.

[4] *Apache Spark*. https://web.archive.org/web/20170325002442/https://spark.apache.org/.

[5] *Apache Storm*. https://web.archive.org/web/20170225062710/http://storm.apache.org/.

[6] Andrew Appel and David MacQueen. "Standard ML of New Jersey". In: *Programming Language Implementation and Logic Programming*. Springer, 1991, pp. 1–13.

[7] Subi Arumugam et al. "The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses". In: *Proc. of the ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. 2010, pp. 519–530.

[8] John Backus. "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641.

[9] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013), 52:1–52:34.

[10] Henry C. Baker Jr. and Carl Hewitt. "The Incremental Garbage Collection of Processes". In: *Proc. of the 1977 Symposium on Artificial Intelligence and Programming Languages*. ACM, 1977, pp. 55–59.

[11] Bas Basten et al. "M3: A general model for code analytics in Rascal". In: *Proc. of the IEEE 1st International Workshop on Software Analytics*. SWAN '15. IEEE. 2015, pp. 25–28.

[12] Bas Basten et al. "Modular Language Implementation in Rascal - Experience Report". In: *Sci. Comput. Program.* 114.C (Dec. 2015), pp. 7–19.

[13] Olav Beckmann et al. “Runtime Code Generation in C++ as a Foundation for Domain-Specific Optimisation”. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Springer Berlin Heidelber, 2004, pp. 291–306.

[14] Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. “Recaf: Java Dialects As Libraries”. In: *Proc. of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE ’16. Amsterdam, Netherlands: ACM, 2016, pp. 2–13.

[15] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. “Clash of the Lambdas”. In: *Proc. 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’14. 2014. arXiv: 1406.6631 [cs.PL].

[16] Aggelos Biboudis et al. “Streams à la carte: Extensible Pipelines with Object Algebras”. In: *Proc. of the 29th European Conference on Object-Oriented Programming*. ECOOP ’15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 591–613.

[17] Gavin Bierman et al. “Pause ’N’ Play: Formalizing Asynchronous C#”. In: *Proc. of the 26th European Conference on Object-Oriented Programming*. ECOOP ’12. Beijing, China: Springer-Verlag, 2012, pp. 233–257.

[18] Corrado Böhm and Alessandro Berarducci. “Automatic synthesis of typed Λ-programs on term algebras”. In: *Theoretical Computer Science* 39 (1985), pp. 135–154.

[19] Anders Bondorf. “Improving Binding Times Without Explicit CPS-conversion”. In: *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*. LFP ’92. San Francisco, California, USA: ACM, 1992, pp. 1–10.

[20] Jeffrey Bosboom et al. “StreamJIT: A Commensal Compiler for High-performance Stream Programming”. In: *Proc. of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’14. Portland, Oregon, USA: ACM, 2014, pp. 177–195.

[21] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. “ASM: a code manipulation tool to implement adaptable systems”. In: *Adaptable and Extensible Component Systems* 30 (2002), p. 19.

[22] Peter A Buhr. *Understanding Control Flow*. Springer International Publishing, 2016.

[23] W. H. Burge. “Combinatory Programming and Combinatorial Analysis”. In: *IBM J. Res. Dev.* 16.5 (Sept. 1972), pp. 450–461.

[24] W. H. Burge. “Stream Processing Functions”. In: *IBM J. Res. Dev.* 19.1 (Jan. 1975), pp. 12–25.

[25] Eugene Burmako. “Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming”. In: *Proc. of the 4th Workshop on Scala*. SCALA ’13. ACM, 2013, p. 3.

[26] R. M. Burstall and John Darlington. “A Transformation System for Developing Recursive Programs”. In: *J. ACM* 24.1 (Jan. 1977), pp. 44–67.

[27] Cristiano Calcagno et al. “Implementing Multistage Languages Using ASTs, Gensym, and Reflection”. In: *Proc. of the 2nd International Conference on Generative Programming and Component Engineering*. GPCE ’03. Springer, 2003, pp. 57–76.

[28] Paris Carbone et al. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[29] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages”. In: *J. Funct. Program.* 19.5 (Sept. 2009), pp. 509–543.

[30] Manuel M.T. Chakravarty et al. “Accelerating Haskell Array Codes with Multicore GPUs”. In: *Proc. of the 6th Workshop on Declarative Aspects of Multicore Programming*. DAMP ’11. Austin, Texas, USA: ACM, 2011, pp. 3–14.

[31] Craig Chambers et al. “FlumeJava: Easy, Efficient Data-parallel Pipelines”. In: *Proc. of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. Toronto, Ontario, Canada: ACM, 2010, pp. 363–375.

[32] Shigeru Chiba. “A Metaobject Protocol for C++”. In: *Proc. of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’95. Austin, Texas, USA: ACM, 1995, pp. 285–299.

[33] Shigeru Chiba. “Javassist–a reflection-based programming wizard for Java”. In: *Proc. of the OOPSLA’98 Workshop on Reflective Programming in C++ and Java*. Vol. 174. 1998.

[34] Ronald Clarke and OG Vitzthum. *Coffee: Recent Developments*. John Wiley & Sons, 2008.

[35] Albert Cohen et al. “In Search of a Program Generator to Implement Generic Transformations for High-performance Computing”. In: *Sci. Comput. Program.* 62.1 (Sept. 2006), pp. 25–46.

[36] Charles Consel and Olivier Danvy. “Tutorial Notes on Partial Evaluation”. In: *Proc. of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’93. ACM, 1993, pp. 493–501.

[37] Melvin E. Conway. “Design of a Separable Transition-diagram Compiler”. In: *Commun. ACM* 6.7 (July 1963), pp. 396–408.

[38] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. “Stream Fusion: From Lists to Streams to Nothing at All”. In: *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’07. Freiburg, Germany: ACM, 2007, pp. 315–326.

[39] Ole-Johan Dahl and C. A. R. Hoare. “Chapter III: Hierarchical Program Structures”. In: *Structured Programming*. Ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press Ltd., 1972, pp. 175–220.

[40] *Dotty Compiler: A Next Generation Compiler for Scala*. `https://web.archive.org/web/20170325001401/http://dotty.epfl.ch/`.

[41] Iulian Dragos. “Compiling Scala for Performance”. eng. PhD thesis. IC, 2010.

[42] ECMA International. *Standard ECMA-262: ECMAScript 2015 Language Specification*. Sixth. June 2015.

[43] ECMA International. *Standard ECMA-334: C# Language Specification*. Forth. June 2006.

[44] Conal Elliott and Paul Hudak. “Functional Reactive Animation”. In: *Proc. of the 2nd ACM SIGPLAN International Conference on Functional Programming*. ICFP’97. Amsterdam, The Netherlands: ACM, 1997, pp. 263–273.

[45] Sebastian Erdweg et al. “Evaluating and Comparing Language Workbenches: Existing Results and Benchmarks for the Future”. In: *Computer Languages, Systems & Structures* 44, Part A (2015). Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014), pp. 24 – 47.

[46] Sebastian Erdweg et al. “SugarJ: Library-based Syntactic Language Extensibility”. In: *Proc. of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’11. Portland, Oregon, USA: ACM, 2011, pp. 391–406.

[47] Andrei P. Ershov. “On the partial computation principle”. In: *Information processing letters* 6.2 (1977), pp. 38–41.

[48] Andrew Farmer, Christian Hoener zu Siederdissen, and Andy Gill. “The HERMIT in the Stream: Fusing Stream Fusion’s concatMap”. In: *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. PEPM ’14. ACM, 2014, pp. 97–108.

[49] Andrew Farmer et al. “The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs”. In: *Proc. of the 2012 Haskell Symposium*. Haskell ’12. ACM, 2012, pp. 1–12.

[50] Bryan Ford. “Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl”. In: *Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’02. Pittsburgh, PA, USA: ACM, 2002, pp. 36–47.

[51] Bryan Ford. “Parsing Expression Grammars: A Recognition-based Syntactic Foundation”. In: *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’04. Venice, Italy: ACM, 2004, pp. 111–122.

[52] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[53] Daniel P. Friedman and David S. Wise. “CONS should not evaluate its arguments”. In: *Automata, Languages and Programming* (1976), pp. 257–284.

[54] Matteo Frigo and Steven G Johnson. “FFTW: An adaptive software architecture for the FFT”. In: *Proc. of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*. Vol. 3. IEEE. 1998, pp. 1381–1384.

[55] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[56] Jeremy Gibbons and Geraint Jones. "The Under-appreciated Unfold". In: *Proc. of the 3rd ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: ACM, 1998, pp. 273–279.

[57] Jeremy Gibbons and Nicolas Wu. "Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl)". In: *Proc. of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: ACM, 2014, pp. 339–347.

[58] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. "A Short Cut to Deforestation". In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: ACM, 1993, pp. 223–232.

[59] Brian Goetz. *Translation of Lambda Expressions*. `https://web.archive.org/web/20170322185227/http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`. Apr. 2012. (Visited on 03/22/2017).

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

[61] Goetz Graefe. "Volcano–An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135.

[62] Goetz Graefe and William J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search". In: *Proc. of the 9th International Conference on Data Engineering*. ICDE '93. IEEE Computer Society, 1993, pp. 209–218.

[63] Daniel Gronau. *HighJ - Haskell-style type classes in Java*. `https://github.com/DanielGronau/highj`. 2011.

[64] Nicholas Halbwachs et al. "The synchronous data flow programming language LUSTRE". In: *Proc. of the IEEE* 79.9 (1991), pp. 1305–1320.

[65] Peter Henderson and James H. Morris Jr. "A Lazy Evaluator". In: *Proc. of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. POPL '76. Atlanta, Georgia: ACM, 1976, pp. 95–103.

[66] Christoph A. Herrmann and Tobias Langhammer. "Combining Partial Evaluation and Staged Interpretation in the Implementation of Domain-Specific Languages". In: *Science of Computer Programming* 62.1 (2006). Special Issue on the First MetaOCaml Workshop 2004, pp. 47 –65.

[67] M. Hills and P. Klint. "PHP AiR: Analyzing PHP systems with Rascal". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Feb. 2014, pp. 454–457.

[68] Ralf Hinze. "Concrete Stream Calculus: An Extended Study". In: *J. Funct. Program.* 20.5-6 (Nov. 2010), pp. 463–535.

[69] M. Hirzel et al. “IBM Streams Processing Language: Analyzing Big Data in Motion”. In: *IBM Journal of Research and Development* 57.3-4 (May 2013), 7:1–7:11.

[70] Martin Hirzel et al. “A Catalog of Stream Processing Optimizations”. In: *ACM Comput. Surv.* 46.4 (Mar. 2014), 46:1–46:34.

[71] Paul Hudak et al. “A History of Haskell: Being Lazy with Class”. In: *Proc. of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pp. 12–1–12–55.

[72] J. Hughes. “Why Functional Programming Matters”. In: *Comput. J.* 32.2 (Apr. 1989), pp. 98–107.

[73] Pablo Inostroza and Tijs van der Storm. “Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras”. In: *Proc. of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE ’15. Pittsburgh, PA, USA: ACM, 2015, pp. 171–180.

[74] Pablo Inostroza, Tijs van der Storm, and Sebastian Erdweg. “Tracing Program Transformations with String Origins”. In: *Proc. of the International Conference on Theory and Practice of Model Transformations*. ICMT ’14. Springer. Springer, Cham, 2014, pp. 154–169.

[75] Jun Inoue and Walid Taha. “Reasoning about Multi-Stage Programs”. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Vol. 7211. ESOP ’12. Springer, 2012, pp. 357–376.

[76] Bart Jacobs et al. “Iterators revisited: Proof rules and implementation”. In: *Proc. of the 7th ECOOP Workshop on Formal Techniques for Java-like Programs*. FTfJP ’05. 2005, pp. 1–21.

[77] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. “Advances in Dataflow Programming Languages”. In: *ACM Comput. Surv.* 36.1 (Mar. 2004), pp. 1–34.

[78] Neil D. Jones. “An Introduction to Partial Evaluation”. In: *ACM Computing Surveys* 28.3 (Sept. 1996), pp. 480–503.

[79] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. “An Experiment in Partial Evaluation: The Generation of a Compiler Generator”. In: *SIGPLAN Notices* 20.8 (Aug. 1985), pp. 82–87.

[80] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. “Mix: A self-applicable partial evaluator for experiments in compiler generation”. English. In: *LISP and Symbolic Computation* 2.1 (1989), pp. 9–50.

[81] Simon Peyton Jones, ed. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[82] Manohar Jonnalagedda and Sandro Stucki. “Fold-based Fusion As a Library: A Generative Programming Pearl”. In: *Proc. of the 6th ACM SIGPLAN Symposium on Scala*. SCALA ’15. ACM, 2015, pp. 41–50.

[83] Ulrik Jørring and William L. Scherlis. “Compilers and Staging Transformations”. In: *Proc. of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’86. ACM, 1986, pp. 86–96.

[84] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. “An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications”. In: *Proc. of the 7th Workshop on Dynamic Languages and Applications*. DYLA ’13. Montpellier, France: ACM, 2013, 3:1–3:9.

[85] Gabriele Keller et al. “Regular, Shape-polymorphic, Parallel Arrays in Haskell”. In: *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 261–272.

[86] R. Kelsey and P. Hudak. “Realistic Compilation by Program Transformation (Detailed Summary)”. In: *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: ACM, 1989, pp. 281–292.

[87] Paul Khuong. *Introducing Pipes, a lightweight stream fusion EDSL*. `https://web.archive.org/web/20110822025633/http://www.pvk.ca/Blog/Lisp/Pipes/introducing_pipes.html`. 2011.

[88] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[89] Oleg Kiselyov. “Iteratees”. In: *Proc. of the 11th International Symposium on Functional and Logic Programming*. Ed. by Tom Schrijvers and Peter Thiemann. FLOPS ’12. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 166–181.

[90] Oleg Kiselyov. “The Design and Implementation of BER MetaOCaml”. In: *Proc. of the 12th International Symposium on Functional and Logic Programming*. FLOPS ’14. Springer, 2014, pp. 86–102.

[91] Oleg Kiselyov et al. “Stream Fusion, to Completeness”. In: *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: ACM, 2017, pp. 285–299.

[92] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation”. In: *Proc. of the 2009 9th IEEE International Working Conference on Source Code Analysis and Manipulation*. SCAM ’09. IEEE Computer Society, 2009, pp. 168–177.

[93] Yannis Klonatos et al. “Building Efficient Query Engines in a High-level Language”. In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 853–864.

[94] Christoph Koch. “Abstraction Without Regret in Database Systems Building: a Manifesto”. In: *IEEE Data Engineering Bulletin* 37.1 (2014). Preprint.

[95] Christoph Koch. “Incremental Query Evaluation in a Ring of Databases”. In: *Proc. of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 87–98.

[96] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. “Generating Code for Holistic Query Evaluation”. In: *Proc. of the 26th International Conference on Data Engineering*. ICDE’10. 2010, pp. 613–624.

[97] Peter Landin. “Correspondence Between ALGOL 60 and Church’s Lambda-notation: Part I”. In: *Commun. ACM* 8.2 (Feb. 1965), pp. 89–101.

[98] Peter Landin. “The Next 700 Programming Languages”. In: *Commun. ACM* 9.3 (Mar. 1966), pp. 157–166.

[99] Chuan-kai Lin and Andrew P. Black. “DirectFlow: A Domain-Specific Language for Information-Flow Systems”. In: *Proc. of the 21st European Conference on Object-Oriented Programming*. Vol. 4609. ECOOP ’07. Berlin, Germany: Springer Berlin Heidelberg, 2007, pp. 299–322.

[100] Tim Lindholm et al. *The Java® Virtual Machine Specification : Java SE 8 Edition*. `https://web.archive.org/web/20170322185325/http://docs.oracle.com/javase/specs/jvms/se8/html/index.html`. Java SE 8 Edition. Mar. 2014.

[101] Ben Lippmeier, Fil Mackay, and Amos Robinson. “Polarized Data Parallel Data Flow”. In: *Proc. of the 5th International Workshop on Functional High-Performance Computing*. FHPC ’16. Nara, Japan: ACM, 2016, pp. 52–57.

[102] Ben Lippmeier et al. “Data Flow Fusion with Series Expressions in Haskell”. In: *Proc. of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 93–104.

[103] Barbara Liskov et al. “Abstraction Mechanisms in CLU”. In: *Commun. ACM* 20.8 (Aug. 1977), pp. 564–576.

[104] Frederik M. Madsen et al. “Functional Array Streams”. In: *Proc. of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*. FHPC ’15. Vancouver, BC, Canada: ACM, 2015, pp. 23–34.

[105] Sandro Magi. *Abstracting over Type Constructors using Dynamics in C#*. `https://web.archive.org/web/20170322142156/http://higherlogics.blogspot.ca/2009/10/abstracting-over-type-constructors.html`. Oct. 2009. (Visited on 03/22/2017).

[106] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. “Exploiting Vector Instructions with Generalized Stream Fusion”. In: *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 37–48.

[107] James Malcolm et al. “ArrayFire: a GPU Acceleration Platform”. In: *Modeling and Simulation for Defense Systems and Applications VII*. Vol. 8403. May 2012, 84030A.

[108] Christopher D Marlin. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. 95. Springer Science & Business Media, 1980.

[109] John McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4 (1960), pp. 184–195.

[110] Erik Meijer. “Confessions of a Used Programming Language Salesman”. In: *Proc. of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 677–694.

[111] Erik Meijer. “Reactive Extensions (Rx): Curing your Asynchronous Programming Blues”. In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP ’10. ACM. 2010, p. 11.

[112] Erik Meijer. “The World According to LINQ”. In: *Communications of the ACM* 54.10 (Oct. 2011), pp. 45–51.

[113] Erik Meijer. “Your Mouse is a Database”. In: *Commun. ACM* 55.5 (May 2012), pp. 66–73.

[114] Erik Meijer, Brian Beckman, and Gavin Bierman. “LINQ: Reconciling Object, Relations and XML in the .NET Framework”. In: *Proc. of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: ACM, 2006, pp. 706–706.

[115] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire”. In: *Proc. of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. London, UK, UK: Springer-Verlag, 1991, pp. 124–144.

[116] Erik Meijer, Kevin Millikin, and Gilad Bracha. “Spicing Up Dart with Side Effects”. In: *Queue* 13.3 (Mar. 2015), 40:40–40:59.

[117] Eugenio Moggi. “Notions of Computation and Monads”. In: *Inf. Comput.* 93.1 (July 1991), pp. 55–92.

[118] Derek Gordon Murray, Michael Isard, and Yuan Yu. “Steno: Automatic Optimization of Declarative Queries”. In: *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 121–131.

[119] S. Muthukrishnan. “Data Streams: Algorithms and Applications”. In: *Found. Trends Theor. Comput. Sci.* 1.2 (Aug. 2005), pp. 117–236.

[120] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550.

[121] Allen Newell. “Documentation of IPL-V”. In: *Commun. ACM* 6.3 (Mar. 1963), pp. 86–89.

[122] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. “Polyglot: An Extensible Compiler Framework for Java”. In: *Proc. of the 12th International Conference on Compiler Construction*. CC’ 03. Warsaw, Poland: Springer-Verlag, 2003, pp. 138–152.

[123] Georg Ofenbeck et al. “Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries”. In: *Proc. of the 12th International Conference on Generative Programming: Concepts and Experiences*. GPCE ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 125–134.

[124] Bruno C. d. S. Oliveira and William R. Cook. “Extensibility for the Masses: Practical Extensibility with Object Algebras”. In: *Proc. of the 26th European Conference on Object-Oriented Programming*. Vol. 7313. ECOOP ’12. Beijing, China: Springer Berlin Heidelberg, 2012, pp. 2–27.

[125] Bruno C. d. S. Oliveira et al. “Feature–Oriented Programming with Object Algebras”. In: *Proc. of the 27th European Conference on Object-Oriented Programming*. ECOOP ’13. Montpellier, France: Springer-Verlag, 2013, pp. 27–51.

[126] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. “Type Classes As Objects and Implicits”. In: *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. Reno/-Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.

[127] Michael Paleczny, Christopher Vick, and Cliff Click. “The Java hotspot™ Server Compiler”. In: *Proc. of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM’01. Monterey, California: USENIX Association, 2001, pp. 1–1.

[128] Nick Palladinos and Kostas Rontogiannis. *LinqOptimizer*. `https://web.archive.org/web/20170327002925/https://nessos.github.io/LinqOptimizer/`. 2013.

[129] Dmitry Petrashko et al. “Call Graphs for Languages with Parametric Polymorphism”. In: *Proc. of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’16. Amsterdam, Netherlands: ACM, 2016, pp. 394–409.

[130] Tomas Petricek and Don Syme. “The F# Computation Expression Zoo”. In: *Proc. of the 16th International Symposium on Practical Aspects of Declarative Languages*. PADL ’14. San Diego, CA, USA: Springer-Verlag New York, Inc., 2014, pp. 33–48.

[131] Simon Peyton Jones. “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell”. In: *Engineering theories of software construction*. Press, 2001, pp. 47–96.

[132] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., 1987.

[133] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. “Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC”. In: *Haskell workshop*. Vol. 1. 2001, pp. 203–233.

[134] F. Pfenning and C. Elliott. “Higher-Order Abstract Syntax”. In: *Proc. of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: ACM, 1988, pp. 199–208.

[135] Atze van der Ploeg and Oleg Kiselyov. “Reflection Without Remorse: Revealing a Hidden Sequence to Speed up Monadic Reflection”. In: *Proc. of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Haskell ’14. 2014, pp. 133–144.

[136] Marc Pouzet. “Lucid synchrone, version 3”. In: *Tutorial and reference manual. Université Paris-Sud, LRI* (2006).

[137] Aleksandar Prokopec and Dmitry Petrashko. *ScalaBlitz: Lightning-Fast Scala collections framework*. `http://scala-blitz.github.io/`. 2013.

[138] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. “Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections”. In: *Proc. of 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. PDP ’15. 2015, pp. 248–252.

[139] Charles Prud’homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. 2015.

[140] Markus Püschel et al. “Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms”. In: *Int. J. High Perform. Comput. Appl.* 18.1 (Feb. 2004), pp. 21–45.

[141] Allan Vincent Roger Renucci and Dmytro Petrashko. *Auto-Collections for Scala*. Tech. rep. 2016.

[142] John C. Reynolds. “User-Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction”. In: *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*. Ed. by David Gries. Springer New York, 1978, pp. 309–317.

[143] Dennis M. Ritchie. “The evolution of the unix time-sharing system”. In: *Language Design and Programming Methodology: Proceedings of a Symposium Held in Sydney, Australia, 10–11 September, 1979*. Ed. by Jeffrey M. Tobias. Springer Berlin Heidelberg, 1980, pp. 25–35.

[144] OM Ritchie and Ken Thompson. “The UNIX time-sharing system”. In: *The Bell System Technical Journal* 57.6 (1978), pp. 1905–1929.

[145] Tiark Rompf and Martin Odersky. “Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs”. In: *Proc. of the 9th International Conference on Generative Programming and Component Engineering*. GPCE ’10. Eindhoven, The Netherlands: ACM, 2010, pp. 127–136.

[146] Tiark Rompf et al. “Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging”. In: *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’13. Rome, Italy: ACM, 2013, pp. 497–510.

[147] Tiark Rompf et al. “Scala-Virtualized: Linguistic Reuse for Deep Embeddings”. In: *Higher Order Symbol. Comput.* 25.1 (Mar. 2012), pp. 165–207.

[148] John Rose. “Bytecodes meet Combinators: invokedynamic on the JVM”. In: *Proc. of the 3rd Workshop on Virtual Machines and Intermediate Languages*. ACM, 2009, p. 2.

[149] John Rose. *implementation of Vector API and associated lambdas*. `https://web.archive.org/web/20170228002819/http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html`. Jan. 2016. (Visited on 02/28/2017).

[150] John Rose. *OpenJDK Project Panama: Interconnecting JVM and native code*. `https://web.archive.org/web/20170131035746/http://openjdk.java.net/projects/panama/`.

[151] John Rose. *perspectives on streams performance*. `https://web.archive.org/web/20170228002819/http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html`. Mar. 2015. (Visited on 02/28/2017).

[152] John Rose et al. *JSR 292: Supporting dynamically typed languages on the Java platform*. `https://web.archive.org/web/20170322185141/https://jcp.org/en/jsr/detail?id=292`. 2011.

[153] Jack Schwartz. “Set theory as a language for program specification and programming”. In: *Courant Institute of Mathematical Sciences, New York University* 12 (1970), pp. 193–208.

[154] Peter Sestoft. *Microbenchmarks in Java and C#*. 2013.

[155] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. “Push vs. Pull-Based Loop Fusion in Query Engines”. In: *arXiv preprint arXiv:1610.09166* (2016).

[156] Mary Shaw, William A. Wulf, and Ralph L. London. “Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators”. In: *Commun. ACM* 20.8 (Aug. 1977), pp. 553–564.

[157] Aleksey Shipilev et al. *OpenJDK Code Tools: jmh*. `https://web.archive.org/web/20161211145153/http://openjdk.java.net/projects/code-tools/jmh/`.

[158] Jeremy Singer. “JVM versus CLR: a comparative study”. In: *Proc. of the 2nd International Conference on Principles and Practice of Programming in Java*. PPPJ ’03. Computer Science Press, Inc., 2003, pp. 167–169.

[159] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. “Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC”. In: *Programming Languages and Systems — ESOP ’94: 5th European Symposium on Programming Edinburg, U.K., April 11–13, 1994 Proceedings*. Ed. by Donald Sannella. ESOP ’94. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 485–500.

[160] Jesper H. Spring et al. “StreamFlex: High-throughput Stream Programming in Java”. In: *Proc. of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*. OOPSLA ’07. Montreal, Quebec, Canada: ACM, 2007, pp. 211–228.

[161] Robert Stephens. “A Survey of Stream Processing”. In: *Acta Informatica* 34.7 (1997), pp. 491–541.

[162] Gordon Stewart et al. “Ziria: A DSL for Wireless Systems Programming”. In: *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. Istanbul, Turkey: ACM, 2015, pp. 415–428.

[163] Jouke Stoel et al. “Solving the Bank with Rebel: On the Design of the Rebel Specification Language and Its Application Inside a Bank”. In: *Proc. of the 1st Industry Track on Software Language Engineering*. ITSLE ’16. Amsterdam, Netherlands: ACM, 2016, pp. 13–20.

[164] Xueyuan Su et al. “Changing Engines in Midstream: A Java Stream Computational Model for Big Data Processing”. In: *Proc. VLDB Endow.* 7.13 (Aug. 2014), pp. 1343–1354.

[165] Josef Svenningsson. “Shortcut Fusion for Accumulating Parameters & Zip-like Functions”. In: *Proc. of the 7th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’02. Pittsburgh, PA, USA: ACM, 2002, pp. 124–132.

[166] Bo Joel Svensson and Josef Svenningsson. “Defunctionalizing Push Arrays”. In: *Proc. of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*. FHPC ’14. ACM, 2014, pp. 43–52.

[167] Wouter Swierstra. “Data Types à La Carte”. In: *J. Funct. Program.* 18.4 (July 2008), pp. 423–436.

[168] Don Syme. *The F# 3.0 Language Specification*. `https://web.archive.org/web/20170325225238/http://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf`. Sept. 2012.

[169] Walid Taha. “A Gentle Introduction to Multi-stage Programming”. en. In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by Christian Lengauer et al. 3016. Springer Berlin Heidelberg, 2004, pp. 30–50.

[170] Walid Taha and Tim Sheard. “Multi-stage Programming with Explicit Annotations”. In: *Proc. of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics based Program Manipulation*. PEPM ’97. ACM, 1997, pp. 203–217.

[171] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Proc. of the 11th International Conference on Compiler Construction*. CC ’02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196.

[172] Sam Tobin-Hochstadt et al. “Languages As Libraries”. In: *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. ACM, 2011, pp. 132–141.

[173] Mads Torgersen. “The Expression Problem Revisited”. In: *Proc. of 18th European Conference on Object-Oriented Programming*. ECOOP ’04. Springer Berlin Heidelberg, 2004, pp. 123–146.

[174] D. A. Turner. “Miranda: A Non-strict Functional Language with Polymorphic Types”. In: *Proc. of the Functional Programming Languages and Computer Architecture*. FPCA ’85. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 1–16.

[175] Vlad Ureche, Cristian Talau, and Martin Odersky. “Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations”. In: *Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’13. Indianapolis, Indiana, USA: ACM, 2013, pp. 73–92.

[176] Vlad Ureche et al. “Automating Ad Hoc Data Representation Transformations”. In: *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA ’15. Pittsburgh, PA, USA: ACM, 2015, pp. 801–820.

[177] Tom Van Cutsem and Mark S. Miller. “Proxies: Design Principles for Robust Object-oriented Intercession APIs”. In: *Proc. of the 6th Symposium on Dynamic Languages*. DLS ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 59–72.

[178] Todd L. Veldhuizen and Dennis Gannon. “Active Libraries: Rethinking the roles of compilers and libraries”. In: *CoRR* math.NA/9810022 (1998).

[179] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., 1985.

[180] Philip Wadler. “Comprehending Monads”. In: *Proc. of the 1990 ACM Conference on LISP and Functional Programming*. LFP ’90. Nice, France: ACM, 1990, pp. 61–78.

[181] Philip Wadler. “Deforestation: Transforming Programs to Eliminate Trees”. In: *Theor. Comput. Sci.* 73.2 (Jan. 1988), pp. 231–248.

[182] Philip Wadler. “How to Replace Failure by a List of Successes”. In: *Proc. of the Conference on Functional Programming Languages and Computer Architecture*. FPCA ’85. Nancy, France: Springer-Verlag New York, Inc., 1985, pp. 113–128.

[183] Philip Wadler. “Listlessness is Better Than Laziness: Lazy Evaluation and Garbage Collection at Compile-time”. In: *Proc. of the 1984 ACM Symposium on LISP and Functional Programming*. LFP ’84. Austin, Texas, USA: ACM, 1984, pp. 45–52.

[184] Philip Wadler. “Monads for Functional Programming”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, 1995, pp. 24–52.

[185] Philip Wadler. *The Expression Problem*. `https://web.archive.org/web/20170322142231/http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`. Dec. 1998. (Visited on 03/22/2017).

[186] Richard C. Waters. “Automatic Transformation of Series Expressions into Loops”. In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 52–98.

[187] Richard C. Waters. *User Manual for the Series Macro Package*. 1989.

[188] Stephen Weeks. “Whole-program Compilation in MLton”. In: *Proc. of the 2006 Workshop on ML*. ML ’06. Portland, Oregon, USA: ACM, 2006, pp. 1–1.

[189] Edwin Westbrook et al. “Mint: Java Multi-stage Programming Using Weak Separability”. In: *Proc. of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’10. Toronto, Canada: ACM, 2010, pp. 400–411.

[190] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. “Automated Empirical Optimization of Software and the ATLAS Project”. In: *Parallel Computing* 27.1–2 (2001). New Trends in High Performance Computing, pp. 3 –35.

[191] J. G. Wingbermuehle, R. D. Chamberlain, and R. K. Cytron. “ScalaPipe: A Streaming Application Generator”. In: *Proc. of the 2012 Symposium on Application Accelerators in High Performance Computing*. SAAHPC ’12. July 2012, pp. 44–53.

[192] Jeremy Yallop and Leo White. “Lightweight Higher-Kinded Polymorphism”. In: *Proc. of the 12th International Symposium on Functional and Logic Programming*. FLOPS ’14. Springer, 2014, pp. 119–135.

[193] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65.

[194] Y. Zhang et al. “Parallel Processing Systems for Big Data: A Survey”. In: *Proc. of the IEEE* 104.11 (Nov. 2016), pp. 2114–2136.