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

Βελτιστοποίηση κλήσεων αναδρομής ουράς σε συναρτησιακές γλώσσες με πολλαπλά είδη αποτίμησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΜΠΟΥΓΟΥΛΙΑΣ

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

Αθήνα, Αύγουστος 2019

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

Βελτιστοποίηση κλήσεων αναδρομής ουράς σε συναρτησιακές γλώσσες με πολλαπλά είδη αποτίμησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΜΠΟΥΓΟΥΛΙΑΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Αυγούστου 2019.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Αύγουστος 2019

Παναγιώτης Μπουγουλιάς

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

Περίληψη

Οι συναρτησιακές γλώσσες προγραμματισμού είναι διάσημες για τη χρήση αναδρομής αντί για «βρόχους», κάτι που υπάρχει και χαρακτηρίζει τις προστακτικές γλώσσες προγραμματισμού. Ενώ η αναδρομή είναι καλύτερη από τους βρόχους σε ό,τι αφορά την καθαρότητα του κώδικα, καθώς είναι άμεσα συνδεδεμένη με το μαθηματικό ορισμό των συναρτήσεων, έχει ένα βασικό ελάττωμα το οποίο συνδέεται με τη χρήση της μνήμης για την εκτέλεση των προγραμμάτων. Το πρόβλημα αυτό είναι ότι ένας αλγόριθμος που χρησιμοποιεί σταθερό χώρο μνήμης για την εκτέλεση του σε μια προστακτική γλώσσα προγραμματισμού, μπορεί να χρησιμοποιήσει γραμμικό χώρο σε συναρτησιακή, εξαιτίας της αναδρομής.

Η λύση αυτού του προβλήματος, που δόθηκε στη δεκαετία του ’70 για τη γλώσσα Scheme, ήταν η χρήση της βελτιστοποίησης κλήσεων ουράς, μια πολύ απλή αλλά, όπως αποδείχθηκε, πολύ αποτελεσματική τεχνική για την εξάλειψη του παραπάνω προβλήματος. Εκτός από τη Scheme, η βελτιστοποίηση αυτή περιλαμβάνεται στις υλοποιήσεις των περισσότερων συναρτησιακών γλωσσών με αυστηρή σημασιολογία, όπως η SML/NJ και η OCaml, καθώς και σε υλοποιήσεις προστακτικών γλωσσών, όπως ο ”Clang” μεταγλωττιστής για τη C/C++, κάτι το οποίο φανερώνει την αξία της συγκεκριμένης βελτιστοποίησης για τη εξάλειψη του κόστους δέσμευσης μνήμης εξαιτίας της αναδρομής.

Ο σκοπός αυτής της διπλωματικής διατριβής είναι να ενσωματώσει αυτή την ευρέως γνωστή βελτιστοποίηση μεταγλωττιστή (βελτιστοποίηση κλήσεων ουράς) για συναρτησιακές γλώσσες προγραμματισμού με αυστηρή αποτίμηση σε συναρτησιακή γλώσσα προγραμματισμού με πολλαπλά είδη αποτίμησης (κλήση κατά αξία, κλήση κατ’ όνομα και κλήση κατ’ ανάγκη). Συγκεκριμένα στην περίπτωση της κλήσης κατ’ ανάγκης, οι τιμές που υπακούουν σε αυτή τη σημασιολογία ”δραπετεύουν” από τη εμβέλεια του ονόματος τους πιο εύκολα απ’ ότι στην κλήση κατ’ αξία, με αποτέλεσμα να είναι αρκετά πιο δύσκολο να βρεθούν οι κλήσεις ουράς. Προσθέτοντας επισημειώσεις αυστηρότητας στη γλώσσα, μπλέκοντας δηλαδή την αυστηρή με την οκνηρή σημασιολογία, με παρόμοιο τρόπο όπως τα BangPatterns της Haskell, πραγματοποιούμε μια στατική ανάλυση η οποία εντοπίζει τις βελτιστοποιήσιμες κλήσεις ουράς σε μια τέτοια γλώσσα. Στη συνέχεια, υλοποιούμε τη βελτιστοποίηση κατά το χρόνο εκτέλεσης. , μετασχηματίζοντας με κατάλληλο τρόπο το πλαίσιο που αντιπροσωπεύει την αντίστοιχη κλήση συνάρτησης. Με την υλοποίηση αυτή, από την εκτέλεση προγραμμάτων ως αναφορά, παρατηρούμε ότι η βελτιστοποίηση μας είτε βελτιώνει το χρόνο εκτέλεσης είτε δεν τον αλλάζει, με αρκετές περιπτώσεις να προσεγγίζουν το σταθερό χώρο μνήμης που προσφέρει η βελτιστοποίηση κλήσης ουράς στις συναρτησιακές γλώσσες προγραμματισμού με αυστηρή σημασιολογία.

Λέξεις κλειδιά

Βελτιστοποίηση κλήσης ουράς, οκνηρή αποτίμηση, στατική ανάλυση, διερμηνέας.

Abstract

Functional programming languages favour recursion over loops while the latter are a key feature of imperative programming languages. Although recursion may be better than loops in terms of code purity, as it is directly linked to the mathematical definition of functions, it has a major flaw associated with the memory overhead to execute programs. This problem is that an algorithm uses a fixed memory space to run it in a imperative programming language, it can use linear space in a functional language because of recursion.

The solution to this problem, given in the 1970s for the programming language Scheme, was tail-call optimisation, a very simple but, as it turned out, very effective in order to eliminate the above problem. In addition to Scheme, this optimisation is included in most implementations of functional programming languages with strict semantics, such as SML / NJ and OCaml, as well as in imperative programming languages, such as the ”Clang” compiler for C/C++, which demonstrates the value of this particular optimisation to eliminate memory overhead due to recursion.

The purpose of this thesis is to integrate this well-known compiler optimisation (tail call optimisation) for strict functional programming languages in functional programming languages with multiple evaluation order choices (call-by-value, call-by-name and call-by-need). Specifically in the presence of call-by-need semantics, the program values under this semantics ”escape” their context more easily than in call-by-value, making tail-calls much more difficult to reveal. By adding strictness to the language, in the form of strictness annotations similar to Haskell’s BangPatterns, we are able to confront with the problem. We perform a static analysis, which spots the tail calls that are actually optimisable in such a language. Next, we optimise the tail calls retrieved by the analysis into the runtime system, in the form of an interpreter which allocates and measures frames, where we mutate the frame that represents the optimisable function call.

Our results from the evaluation of microbenchmarks show that our technique either improves the runtime, or does not change it. Some microbenchmarks, though, approach the constant memory space in the same way as in strict functional programming languages.

Key words

Tail-call optimisation, lazy evaluation, static analysis, interpreter.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Νίκο Παπασπύρου, για τη συνεχή καθοδήγηση και εμπιστοσύνη του. Θέλω να ευχαριστήσω ακόμα το Γιώργο Φουρτούνη, ο οποίος με βοήθησε σε διάφορα στάδια αυτής της εργασίας. Θα ήθελα τέλος να ευχαριστήσω την οικογένειά μου και κυρίως τους γονείς μου, οι οποίοι με υποστήριξαν και έκαναν δυνατή την απερίσπαστη ενασχόλησή μου τόσο με την εκπόνηση της διπλωματικής μου, όσο και συνολικά με τις σπουδές μου.

Παναγιώτης Μπουγουλιάς,

Αθήνα, 28η Αυγούστου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-19, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Αύγουστος 2019.

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

Περιεχόμενα

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός της εργασίας είναι η ενσωμάτωση της βελτιστοποίησης κλήσης ουράς σε συναρτησιακές γλώσσες προγραμματισμού με την παρουσία πολλαπλών επιλογών σειράς αποτίμησης (κλήση κατ' αξία, κλήση κατ' όνομα, κλήση κατ' ανάγκη). Η βελτιστοποίησή μας μπορεί να συνδυαστεί με το μετασχηματισμό defunctionalization, έτσι ώστε η βελτιστοποίηση απόδοσης που προκύπτει να υποστηρίζει επίσης συναρτησιακές γλώσσες υψηλότερης τάξης.

1.2 Κίνητρο

Η βελτιστοποίηση κλήσεων ουράς για συναρτησιακές γλώσσες με αυστηρή αποτίμηση είναι ένα καλά μελετημένο θέμα, αλλά αποτυγχάνει στην παρουσία οκνηρής αποτίμησης (lazy): ο υπολογισμός ενός οκνηρού όρου μπορεί να συμβεί αυθαίρετα κατά τη διάρκεια της εκτέλεσης του προγράμματος και έτσι οι οκνηροί όροι δραπετεύουν από το περιβάλλον τους πιο εύκολα από τους αυστηρούς. Από τη μία πλευρά, η οκνηρότητα δίνει στους προγραμματιστές μεγάλες επιλογές: να χρησιμοποιούν άπειρες δομές δεδομένων, να ορίσουν τη ροή ελέγχου (δομές) ως αφαιρέσεις (abstaractions), να αυξήσουν την απόδοση αποφεύγοντας τους περιττούς υπολογισμούς και αποφεύγοντας τις περιπτώσεις σφάλματος κατά την αξιολόγηση σύνθετων εκφράσεων. Αλλά αυτή η ευελιξία έρχεται με κόστος: τα πολύ οκνηρά προγράμματα μπορεί να οδηγήσουν σε κακή απόδοση, ακόμη και σε διαρροές μνήμης. Από την άλλη πλευρά, η αυστηρότητα επιτρέπει στους προγραμματιστές να μετρούν την επίδοση του προγράμματος πιο εύκολα, αλλά τα προγράμματα μπορούν εμπίπτουν σε ανεπιθύμητη συμπεριφορά, όπως η απόκλιση.

Σε αυτή τη διατριβή, παίρνουμε το καλύτερο και των δύο κόσμων: τα προγράμματά μας δεν χρειάζεται να είναι πολύ οκνηρά ούτε πολύ αυστηρά. Χρησιμοποιώντας επισημειώσεις σειράς αποτίμησης, υποστηρίζουμε τη βελτιστοποίηση κλήσεων ουράς σε οκνηρές συναρτησιακές γλώσσες προγραμματισμού, εξαλείφοντας έτσι την δέσμευση επιπλέον μνήμης που σχετίζεται με την αναδρομή, όπως και στις αυστηρές συναρτησιακές γλώσσες.

1.3 Σύνοψη της διπλωματικής

Σε αυτή τη διατριβή, ενσωματώνουμε την βελτιστοποίηση κλήσεων ουράς σε συναρτησιακή γλώσσα με την παρουσία πολλαπλών σειρών αποτίμησης (κλήση κατ' αξία, κλήση κατ' όνομα, κλήση κατ' ανάγκη). Η βελτιστοποίησή μας υποστηρίζει τύπους δεδομένων που έχουν οριστεί από τον χρήστη με αντιστοίχιση μοτίβων (pattern matching) και συνεπώς μπορούν να συνδυαστούν με το μετασχηματισμό defunctionalization, έτσι ώστε να υποστηρίζονται επίσης λειτουργικές γλώσσες ανώτερης τάξης. Η γλώσσα πηγής, παρόμοια με τη Haskell, μετασχηματίζεται (μέσω defunctionalization) σε μια χαμηλού επιπέδου, ελάχιστη, πρώτης τάξης συναρτησιακή γλώσσα με μη αυστηρή σημασιολογία, οκνηρή αποτίμηση και οκνηρά δομημένα δεδομένα καθώς και επισημειώσεις αυστηρότητας. Για να βρούμε ευκαιρίες για βελτιστοποίηση, κάνουμε μια στατική ανάλυση για τη συναρτησιακή γλώσσα χαμηλού επιπέδου, για τον εντοπισμό θέσεων κλήσης ουράς. Το δύσκολο κομμάτι, σε σύγκριση με τις

γλώσσες με αυστηρή σημασιολογία, είναι ότι η οκνηρή σημασιολογία κάνει τις τιμές του προγράμματος να ξεφύγουν από την εμβέλεια του ονόματός τους και, συνεπώς, η εύρεση θέσεων κλήσης ουράς δεν είναι τόσο εύκολη. Αυτή η βελτιστοποίηση αξιολογήθηκε σε έναν διερμηνέα της γλώσσας που δεσμεύει ρητά και μετράει πλαίσια, έτσι ώστε στις θέσεις που βρέθηκαν από την ανάλυση, να μπορούμε να αντικαταστήσουμε σωστά τις περιττές παραμέτρους πλαισίου με αυτές που απαιτούνται από το πλαίσιο που αντιπροσωπεύει την επόμενη αναδρομική κλήση. Η βελτιστοποίηση είτε βελτιώνει την απόδοση του προγράμματος είτε δεν την αλλάζει. Επίσης, στην περίπτωση προγραμμάτων με αυστηρή σημασιολογία, η βελτιστοποίησή μας είναι ισοδύναμη με την κλασική βελτιστοποίηση κλήσεων ουράς. Συμπερασματικά, σε μια μη αυστηρή, οκνηρή συναρτησιακή γλώσσα με οκνηρούς κατασκευαστές δεδομένων και επισημειώσεις αυστηρότητας, για όλους τα προγράμματα που χρησιμοποιούμε ως αναφορά, υπάρχει πάντα βελτιστοποίηση μνήμης, και για την πλειοψηφία από αυτά υπάρχει σημαντική βελτιστοποίηση μνήμης με ενίσχυση της απόδοσης.

1.4 Συνεισφορά

Οι σημαντικότερες συνεισφορές μας, που παρουσιάζονται σε αυτή τη διατριβή, είναι:

- Παρουσιάζουμε πώς η βελτιστοποίηση της κλήσης ουράς μπορεί να εφαρμοστεί σε μια συναρτησιακή γλώσσα με μεικτή σειρά αποτίμησης. Βασική συνεισφορά μας είναι πώς μπορεί να εφαρμοστεί υπό την παρουσία της οκνηρής αποτίμησης, μετατρέποντας έτσι αυτή τη βελτιστοποίηση σε επέκταση της βελτιστοποίησης κλήσεων ουράς για γλώσσες με κλήση κατ' αξία.

- Μια πρωτότυπη υλοποίηση ενός διερμηνέα με ρητή δέσμευση πλαισίων, καθώς και μηχανισμό μέτρησης πλαισίων, για μια συναρτησιακή γλώσσα με κλήση κατ' αξία, κλήση κατ' όνομα και κλήση κατ' ανάγκη, οκνηρούς κατασκευαστές δεδομένων και pattern matching.

- Ένας αλγόριθμος στατικής ανάλυσης για την εύρεση πραγματικών θέσεων κλήσης ουράς και την υλοποίηση του συστήματος χρόνου εκτέλεσης για αυτές τις βελτιστοποιήσεις που είναι ενσωματωμένες στον διερμηνέα.

- Η αξιολόγηση αυτής της βελτιστοποίησης σε προγράμματα αναφοράς δείχνει ότι είτε βελτιώνει τη μνήμη ή δεν την αλλάζει. Η αξιολόγηση φαίνεται να προσεγγίζει σε πολλές περιπτώσεις τον αριθμό της δέσμευσης πλαισίων σε γλώσσες που χρησιμοποιούν την κλήση κατ' αξία, που είναι το καλύτερο αποτέλεσμα που μπορούμε να έχουμε.

Κεφάλαιο 2

Η γλώσσα

Σε αυτό το κεφάλαιο, θα περιγράψουμε τη γλώσσα που μελετήσαμε σε αυτή τη διατριβή. Η σημασιολογία της γλώσσας θα γίνει σαφής στον αναγνώστη, καθώς και η ειδική αντιμετώπιση των πολλαπλών ειδών αποτίμησης στον τελικό πυρήνα της γλώσσας, η οποία χρησιμοποιήθηκε για τη στατική ανάλυση και για τις βελτιστοποιήσεις. Εν συντομία, υπάρχει μια περιεκτική περιγραφή της διαδρομής από μια συναρτησιακή γλώσσα ανώτερης τάξης σε μια συναρτησιακή γλώσσα πρώτης τάξης με τις επισημειωμένες των σειρών αποτίμησης.

2.1 Επισκόπηση της γλώσσας

Η γλώσσα που μελετήσαμε είναι μια λειτουργική γλώσσα με πολλαπλές σειρές αποτίμησης (κλήση κατ' ανάγκη ή οκνηρή, κλήση κατ' όνομα και κλήση κατ' αξία ή αυστηρή). Θεωρούμε την οκνηρότητα ως κάτι που συμβαίνει φυσικά στη γλώσσα, ενώ οι άλλες δύο σειρές αποτίμησης χρησιμοποιούνται ως επεκτάσεις γλώσσας, χρησιμοποιώντας κατάλληλες επισημειώσεις.

Οι επισημειώσεις αποτίμησης μπορούν να εμφανιστούν συντακτικά (δηλ. ο προγραμματιστής μπορεί να τις χρησιμοποιήσει στον κώδικα στις στις τυπικές παράμετρους στον ορισμό της συνάρτησης) ή μετά την εφαρμογή των μετασχηματισμών, που περιγράφονται στα τμήματα 3.2 έως 3.4, στη γλώσσα πηγής. Το τελευταίο είναι μόνο η περίπτωση αυστηρών όρων που αποκαλύπτονται από την ανάλυση αυστηρότητας.

2.2 Μετασχηματισμός defunctionalization

Πρόκειται για μια τεχνική μετασχηματισμού χρόνου μεταγλώτισσης που καταργεί τις εφαρμογές υψηλότερης τάξης αντικαθιστώντας τις με μία μόνο συνάρτηση πρώτης τάξης, που παρουσιάστηκε από τον John Reynolds. Η παρατήρηση του Reynolds ήταν ότι ένα δεδομένο πρόγραμμα περιέχει μόνο πεπερασμένα πολλές αφαιρέσεις λειτουργιών, έτσι ώστε το καθένα μπορεί να ανατεθεί σε (και να αντικατασταθεί από) ένα μοναδικό αναγνωριστικό. Στη συνέχεια αντικαθίσταται κάθε εφαρμογή συνάρτησης μέσα στο πρόγραμμα από μια κλήση προς τη λειτουργία εφαρμογής με το αναγνωριστικό λειτουργίας ως το πρώτο επιχείρημα. Η μόνη δουλειά της εφαρμογής είναι η συνάρτηση να εφαρμόζεται σε αυτό το πρώτο επιχείρημα και, στη συνέχεια, να εκτελείται τις εντολές που αντιστοιχούν με το αναγνωριστικό συνάρτησης στην εντολή με τα υπόλοιπες παραμέτρους.

Ο μετασχηματισμός αποκοπής χρησιμοποιήθηκε στο MLton, ένας βελτιστοποιημένος μεταγλωττιστής για ML. Η γλώσσα πυρήνα πρώτης τάξης δημιούργησε ευκαιρίες για ανάλυση ολόκληρου του προγράμματος και οδήγησε σε εξαιρετική απόδοση.

Ως παράδειγμα του μετασχηματισμού, ακολουθεί η συνάρτηση map του Prelude.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

inc :: Int -> Int
```

```
inc a = a + 1

result = map inc [1,2,3]
```

Μετά την εκτέλεση του defunctionalization, το ‘map ’είναι (instantiated για ακεραίους):

```
data Func = Inc

apply :: Func -> b -> b
apply f x =
  case f of
    Inc -> inc x

map :: Func -> [a] -> [b]
map f l =
case l of
  []     -> acc
  x : xs -> apply f x : map f xs

inc :: Int -> Int
inc a = a + 1

result :: [Int]
result = map Inc [1, 2, 3] []
```

Οι διαφορές μεταξύ των δύο εκδόσεων είναι:

- Η υψηλής τάξης συνάρτηση, το πρώτο όρισμα της ‘map ’(f :: a -> b) μετασχηματίζεται σε *μοναδικό* αναγνωριστικό, το οποίο είναι ο κατασκευαστής δεδομένων ‘Inc’.

- Για την εφαρμογή μέσα στο σώμα της συνάρτησης ‘result’, έχουμε την συνάρτηση πρώτης τάξης *apply* η οποία αντιστοιχεί τα μοναδικά αναγνωριστικά που εφαρμόζονται στη ‘map’.

Δεν δίνονται περισσότερες λεπτομέρειες και φορμαλισμός του μετασχηματισμού defunctionalization εδώ. Ο αναγνώστης μπορεί να αναφερθεί στη δημοσίευση του Reynolds για περισσότερες πληροφορίες σχετικά με αυτό το μετασχηματισμό.

2.3 Ανάλυση αυστηρότητας

Η κλήση κατ’ ανάγκη μόνο όρους σε τιμές όταν χρειάζεται. Αυτό παρέχει την ευκαιρία για άπειρες δομές δεδομένων και προγράμματα που δεν αποκλίνουν, όπως αυτά σε αυστηρές γλώσσες προγραμματισμού. Η αξιολόγηση μπορεί να συμβεί αυθαίρετα και ως εκ τούτου οι οκνηροί όροι μπορούν να ξεφύγουν από το περιβάλλον τους.

Αλλά ο Mycroft παρατήρησε ότι ορισμένοι οκνηροί όροι είναι στην πραγματικότητα αυστηροί υπό τις κατάλληλες συνθήκες. Για παράδειγμα, το ‘case’ αποτιμά κάποιον όρο (*scrutinee* [Jone95]), κάνοντάς τον αυστηρό, παρόμοια με την αυστηρότητα των BangPatterns.

Και είναι κάτι παραπάνω από αυτό. Ας δούμε ξανά το πρόγραμμα sum:

```
sum :: [Int] -> Int -> Int
sum [] acc = acc
sum (x: xs) acc = sum xs (x + acc)
```

Εδώ, ο οκνηρός όρος ‘acc’, ο οποίος είναι σε μορφή που δεν έχει υπολογιστεί ακόμα (*thunk* [Blos88]), σε κάθε κλήση λειτουργίας αποτιμάται μία φορά, όταν το πρόγραμμα κατά τη ροή εκτέλεσης του φτάνει στη βασική περίπτωση. Αυτή η αυθαίρετη αποτίμηση των οκνηρών όρων φαίνεται να μην είναι τόσο αυθαίρετη. Γνωρίζουμε πότε πρόκειται να αξιολογηθεί το ’acc’.

Λοιπόν, το ερώτημα είναι, τι είναι αυτό που κάνει το ‘acc ’τόσο προβλέψιμο; Από το παραπάνω παράδειγμα, έχουμε τις ακόλουθες πληροφορίες για το ‘acc’:

- ‘acc’ είναι ένας ακέραιος αριθμός, που πάντα προστίθεται σε έναν άλλο ακέραιο αριθμό.
- ‘acc’ είναι το αποτέλεσμα της συνάρτησης.

Από αυτές τις παρατηρήσεις μπορούμε να συμπεράνουμε ότι το acc πρόκειται να αξιολογηθεί (δεύτερη παρατήρηση) και όλες οι ενδιάμεσες τιμές είναι απαραίτητες για την αποτίμησή του, καθώς το ‘acc ’είναι ένας ακέραιος αριθμός (πρώτη παρατήρηση), οπότε και μπορούν να αποτιμώνται κάθε φορά.

Αν και σε αυτή τη διατριβή χρησιμοποιήσαμε ακέραιους αριθμούς και λίστες, η ανάλυση αυστηρότητας μπορεί να εφαρμοστεί εκτός από ακεραίους και σε οποιοδήποτε άλλες τιμές βασικού τύπου. Η ανάλυση αυστηρότητας εκτελείται στη γλώσσα της ανώτερης τάξης μετά το defunctionalization για να εντοπίσει τούς αυστηρούς όρους.

2.4 Άλλοι μετασχηματισμοί

Μετά από το μετασχηματισμό defunctionalization και την ανάλυση αυστηρότητας, έχουμε επίσης εκτελέσει μερικούς μετασχηματισμούς στη γλώσσα, προκειμένου η ενδιάμεση γλώσσα να φτάσει στην τελική μορφή, η οποία είναι η είσοδος της ανάλυσης, που παρουσιάζεται στην επόμενη ενότητα.

- **Άλφα-μετονομασία:** Αυτός ο μετασχηματισμός (*alpha-renaming* ή *alpha-conversion* [Bare92]) μετονομάζει όλες τις μεταβλητές που δεσμεύονται από την αντιστοίχιση προτύπων. Κάθε έκφραση ‘case’ ανοίγει ένα νέο πεδίο εφαρμογής και τα ονόματα μεταβλητών σε αυτό το πεδίο δεσμεύονται στο πλησιέστερο «περίπτωση». Αυτή η μεταμόρφωση επίσης χειρίζεται τη σκίαση ονόματος.

- **Μετατροπή του if σε case:** Όταν η γλώσσα υποστηρίζει αντιστοίχιση προτύπων στα δεδομένα (κατασκευαστές), η έκφραση ‘if’ είναι άχρηστη. Μπορεί να μετατραπεί σε αντιστοίχιση προτύπου το μηδενικό boolean κατασκευαστή. Ο Haskell δεν έχει έκφραση ‘if ’ως ενσωματωμένη (builtin). Επιτρέπει μόνο συντακτική χρήση (ως συντακτική ζάχαρη για μοτίβα boolean).

- **Προβολές κατασκευαστών (constructor projections):** Κάθε μεταβλητή που δεσμεύεται από ένα μοτίβο ‘case ’πρέπει να είναι μετασχηματισμένο σε έναν ειδικό συντακτικό κόμβο στη γλώσσα του πυρήνα, ο οποίος περιέχει το μοναδικό αναγνωριστικό της περίπτωσης και η θέση μέσα στη λίστα έκφρασης στο πρότυπο κατασκευαστή [Four13]. Το αναγνωριστικό ”case” είναι μοναδικό μέσα σε έναν ορισμό συνάρτησης (τοπικά).

2.5 Σύνταξη

Σε αυτή την ενότητα δίνεται η σύνταξη της βασικής μας γλώσσας. Αυτό είναι το αποτέλεσμα της ανάλυσης αυστηρότητας και μετασχηματισμού καθυστέρησης, καθώς και των άλλων μετασχηματισμών της προηγούμενης Ενότητα. Εκτελούμε τον αλγόριθμο ανάλυσης για να εντοπίσουμε θέσεις κλήσης ουράς με είσοδο τη γλώσσα από αυτήν την ενότητα. Ο διερμηνέας είναι επίσης χτισμένος για τη γλώσσα αυτή. ο αξιολογητής για οι βελτιστοποιήσεις εφαρμόζονται επίσης στο εσωτερικό του διερμηνέα για αυτή τη γλώσσα.

Στο σχήμα 2.1, υπάρχει η αφηρημένη σύνταξη της ενδιάμεσης γλώσσα πρώτης τάξης. Πρέπει να επισημάνουμε τα ακόλουθα σημεία, πριν εμβαθύνουμε βαθύτερα στη γλώσσα:

$\langle p \rangle ::= \langle fdef \rangle^+$	Program
$\langle fdef \rangle ::= f\ v_1 \ldots v_n = \langle expression \rangle$	Function Definition
$\langle expr \rangle ::= v$	Variable
$\mid \langle integer \rangle$	Integer
$\mid f\ e_1 \ldots e_n$	Function application
$\mid c\ e_1 \ldots e_n$	Constructor application
$\mid$ `case` e_0 `of` $patt_1 \rightarrow e_1 \mid \ldots \mid patt_n \rightarrow e_n$	Case expression
$\langle patt \rangle ::= c\ v_1 \ldots v_n$	Constructor pattern
$\mid \langle integer \rangle$	Integer pattern
$\langle integer \rangle ::= 1, 2, \ldots$	Integer domain

Σχήμα 2.1: Η γραμματική της γλώσσας

- Στη γλώσσα προέλευσης υπάρχει μια έκφραση 'if 'συντακτικά. Στο συντακτικό σχήμα δεν υπάρχει. Αυτό οφείλεται στο ότι η 'case' είναι πολύ πιο ισχυρή από το 'if 'και το' if' μπορεί να μετατραπεί σε 'case'. Στην πραγματικότητα έχουμε (boolean είναι κατασκευασμένα δεδομένα επίσης) pattern matching σε booleans:

```
if cond then e1 else e2 => case cond of {True -> e1; False -> e2}
```

- Δεν υπάρχουν μερικές εφαρμογές, ούτε σε συναρτήσεις ούτε σε εφαρμογές κατασκευαστών. Αυτό σημαίνει ότι τα επιχειρήματα σε μια κλήση λειτουργίας είναι ίδια στον αριθμό με τον ορισμό της λειτουργίας.

- Δεν υπάρχει έκφραση 'let' στην γλώσσα. Αντ 'αυτού υποθέτουμε ότι η γλώσσα μας εξαρτάται πλήρως από τον ανυψωτή λάμδα (lambda lifter). Ο lambda lifter του Johnson *πλήρους οκνηρότητας* θα έχει το επιθυμητό αποτέλεσμα [John85].

- Τα μοτίβα των περιπτώσεων είναι απλά και δεν επιτρέπουν τα μοτίβα μπαλαντέρ (wildcard). Μετατρέποντας ένα πολύπλοκο μοτίβο σε ένα απλό είναι ένα καλά μελετημένο θέμα.

Η σύνταξη στο σχήμα είναι μια σύνταξη μιας συναρτησιακής γλώσσας πρώτης τάξης. Η σύνταξή μας έχει δύο κύριες διαφορές από αυτές που φαίνονται στο σχήμα:

- Η συγκεκριμένη σύνταξη της βασικής γλώσσας έχει σημειώσεις τάξης αξιολόγησης προκειμένου να να διακρίνει μεταξύ της σημασιολογίας κατά τη διέλευση των παραμέτρων. Έχουμε ήδη αναφέρει το τα σχόλια αυστηρότητας που προστίθενται από την ανάλυση αυστηρότητας. Πιο συγκεκριμένα, ο προγραμματιστής μπορεί συγκεκριμένα να σχολιάσει τις μορφές στον ορισμό της λειτουργίας με:
  - '!' για μια αυστηρή τυπική παράμετρο,
  - '# 'για τυπική παράμετρο κλήσης με όνομα, ενώ
  - Οι οκνηρές τυπικές παράμετροι δεν σχολιάζονται, επειδή θεωρούμε την οκνηρότητα ως κάτι που φυσικά συμβαίνει στη γλώσσα.

  Όσον αφορά την αφηρημένη σύνταξη, οι τυπικές παράμετροι ενός ορισμού συνάρτησης είναι:

```
type Formal = (VN, (EvalOrder, Type))
type VN = String
type EvalOrder = CBV | CBN | Lazy
```

```
data Type = TInt | TCons Type -- an value of type integer (TInt) or
                              -- a list of values with type Type
```

όπου ο κατασκευαστής τύπου μιας τυπικής παραμέτρου έχει τις πληροφορίες σχετικά με το όνομα της παραμέτρου και τη στατική πληροφορία της εντολής αξιολόγησης και τον τύπο της. Υποθέτουμε τους τύπους βάσης ως ακέραιους αριθμούς και αργότερα το τμήμα ανάλυσης, θα εξηγήσουμε γιατί αυτό αρκεί.

- Υπάρχει ένας ακόμα κόμβος στο συντακτικό δέντρο: οι προβολές κατασκευαστών (*constructor projections*). Ο μετασχηματισμός γι 'αυτό, ο συγκεκριμένος κόμβος σύνταξης έχει εξηγηθεί νωρίτερα σε προηγούμενη ενότητα και η σύνταξη σε όρους Haskell είναι:

  ```
  data Expr = ... | CProj CaseID CPos
  ```

  Σε μια προβολή κατασκευαστή, το 'CaseID' δείχνει τη θέση του 'case' όπου ανήκει ο όρος, μέσα στο σώμα της συνάρτησης, και το δεύτερο όρισμα ('CPos') είναι η θέση της μεταβλητής στη λίστα εκφράσεων του μοτίβου κατασκευαστή σε διακλάδωση του 'case'.

  Ο σκοπός των προβολών του κατασκευαστή είναι να 'δέσει' τις μεταβλητές στη δεξιά πλευρά της διακλάδωσης με τις μεταβλητές στην αριστερή πλευρά. Με τον τρόπο αυτό, οι μεταβλητές αυτές διακρίνονται από τις μεταβλητές ανώτερου επιπέδου, δηλ. από τις τυπικές παραμέτρους στον ορισμό της συνάρτησης και προσφέρουν τη δυνατότητα να γνωρίζουμε τη θέση τους στο πλαίσιο, ένας λόγος που θα γίνει σαφέστερος στο επόμενο κεφάλαιο, όπου θα δώσουμε τις λεπτομέρειες του μοντέλου εκτέλεσης.

Κεφάλαιο 3

Μοντέλο εκτέλεσης

Σε αυτό το κεφάλαιο, θα παρουσιάσουμε τις τεχνικές λεπτομέρειες του μοντέλου εκτέλεσης για τη γλώσσα που μελετήσαμε σε αυτή τη διατριβή καθώς και τις τεχνικές λεπτομέρειες σχετικά με την υλοποιήση και τη λειτουργία τον διερμηνέα, στον οποίο αξιολογούμε την τεχνική μας για την βελτιστοποίηση της κλήσης ουράς.

Στην επόμενη ενότητα υπάρχει μια περιγραφή υψηλού επιπέδου του μοντέλου. Στη συνέχεια παρουσιάζονται οι δομές δεδομένων που χρησιμοποιούνται κατά το χρόνο εκτέλεσης και στη συνέχεια η λειτουργία του διερμηνέα παρουσιάζεται με τη μορφή λειτουργικής σημασιολογίας (operational semantics) για τη γλώσσα.

3.1 Περιγραφή υψηλού επιπέδου του μοντέλου εκτέλεσης

In this section, we present a high-level description for the execution model; this will be helpful for the reader to follow the technical details from the next sections. Also, it will be easier to compare to other existing abstract machines and interpreters that exist for a lazy functional language.

Το μοντέλο μας, βασισμένο στον GIC [Four14a], includes the following high-level properties:

- Υλοποιούμε μια αφηρημένη μηχανή πρώτης τάξης. Πρώτης τάξης, διότι δεν υποστηρίζουμε πέρασμα συναρτήσεων υψηλής τάξης ως παραμέτρους σε συνάρτηση , καθώς και μερική αποτίμηση. Η οκνηρότητα είναι η προεπιλεγμένη σημασιολογία για το μοντέλο μας.
- Έχουμε ένα μοντέλο βασισμένο στη δέσμευση πλαισίων για τη γλώσσα μας, αλλά τα πλαίσια δεσμεύονται στο σωρό αντί για στοίβα (heap allocated frames). Ο λόγος γι' αυτό είναι ότι η οκνηρή αποτίμηση δε συμβαδίζει με την ακολουθιακή ροή της εκτέλεσης του προγράμματος, κάτι το οποίοπ καθιστά αδύνατη τη δέσμευση/αποδέσμευση πλαισίων με τις λειτουργίες push/pop.
- Ο διερμηνέας μας υποστηρίζει τη ρητή δέσμευση πλαισίων, η οποία χρειάζεται για το μετασχηματισμό των πλαισίων στη βελτιστοποίηση καθώς και την αξιολόγηση της τεχνικής μας, όπως επίσης και ένα μηχανισμό καταμέτρησης των πλαισίων.
- Η γλώσσα μας υποστηρίζει πολλαπλά είδη αποτίμησης και έτσι ο διερμηνέας μας τα χειρίζεται και αυτά (call-by-value, call-by-name, call-by-need) με κατάλληλο τρόπο.
- Τα δεδομένα μας κατασκευάζονται από οκνηρούς κατασκευαστές δεδομένων, αλλά μπορούμε επίσης να υποστηρίξουμε ML-λίστες (δεδομένα με βαθιά αποτίμηση, όπως στη γλώσσα ML).
- Στις κλήσεις συναρτήσεων ακολουθούμε την πολιτική push/enter. Πριν εισέλθουμε στο σώμα της συνάρτησης, δεσμεύουμε το πλαίσιο και αποτιμούμε όλες τις πραγματικές παραμέτρους, σεβόμενοι τη σημασιολογία τους.
- Το μοντέλο μας υποστηρίζει pattern matching (σε δεδομένα και τιμές βασικού τύπου) με παρόμοιο τρόπο, όπως στη Haskell. Αυτό αποτελεί το πιο σύνθετο γνώρισμα μιας οκνηρής συναρτησιακής γλώσσας, ειδικά αν αναλογιστούμε ότι οι πρώτες αφηρημένες μηχανές γι' αυτές τις γλώσσες δεν παρείχαν υποστήριξη για το συγκεκριμένο γνώρισμα.

3.2 Σύστημα χρόνου εκτέλεσης

Αυτή η ενότητα περιέχει μια λεπτομερή εξήγηση για την πρωτότυπη εφαρμογή του διερμηνέα μας που χρησιμοποιείται στη γλώσσα. Πρώτον, θα παρουσιάσουμε τις δομές δεδομένων που χρησιμοποιούνται κατά τη διάρκεια εκτέλεσης. Έπειτα, παρουσιάζουμε τα λειτουργικά χαρακτηριστικά του διερμηνέα μας.

3.3 Δομές δεδομένων κατά το χρόνο εκτέλεσης

Εδώ υπάρχουν ορισμοί των δομών χρόνου εκτέλεσης, όπως χρησιμοποιούνται στην υλοποίηση. Αν και η εφαρμογή είναι σε Haskell, ο κώδικας θα είναι απλός, έτσι ώστε κάθε αναγνώστης με ένα βασικό υπόβαθρο συναρτησιακού προγραμματισμού να μπορεί να το καταλάβει.

Στις επόμενες ενότητες που θα περιγράψουμε τις δομές χρόνου εκτέλεσης, θα χρησιμοποιήσουμε την ακόλουθη σύμβαση. Στην αρχή, ένας ορισμός θα εμφανιστεί, ενώ στξ συνέχια πιθανώς θα συνοδεύεται από έναν ή περισσότερους ορισμούς. Αν περιέχει πιο πολύπλοκα δεδομένα στο σώμα του, θα περιγράφεται αργότερα στην ενότητα ή θα υπάρχει ένας δείκτης σε μια άλλη ενότητα που θα περιέχει λεπτομέρειες γι' αυτόν τον ορισμό.

3.3.1 Μνήμη

Σε αυτή την ενότητα παρέχουμε τον ορισμό της δομής της μνήμης(*memory*), ένα *mutable* χώρο αποθήκευσης, όπου αποθηκεύονται τα πλαίσια.

Για τη δομή *memory*, έχουμε τα παρακάτω:

```
type FrameId = Int
data Mem = Mem {
  memFrames :: Map FrameId Frame, lastFrameId :: FrameId
}
```

Ο αναγνώστης μπορεί να σκεφτεί τη δομή αυτή ως μνήμη σε έναν υπολογιστή: κάθε κύτταρο μνήμης έχει μια διεύθυνση (*FrameId*) και η διεύθυνση έχει ένα περιεχόμενο (*frame*). Μια παρόμοια αναπαράσταση με το mutable σωρό του Launchbury για οκνηρή αποτίμηση. Εδώ να σημειώσουμε ότι όλα τα πλαίσια δεν έχουν την ίδια χωρητικότητα. Λεπτομέρειες θα δοθούν σε επόμενη ενότητα για ένα πλαίσιο.

Αυτή η προαναφερθείσα δομή δεδομένων συνοδεύεται από τις τρεις ακόλουθες λειτουργίες, που δίνονται αμέσως μετά.

- Προσθήκη ενός πλαισίου στη μνήμη με τη λειτουργία *push*, στο σχήμα 3.1.
- Ανάγνωση του περιεχόμενου ενός πλαισίου από τη μνήμη, αν δοθεί το αναγνωριστικό του (*getFrame*) , η οποία δίνεται στο σχήμα 3.2.
- Ενημέρωση του περιεχομένου ενός πλαισίου στη μνήμη, δοθέντος του αναγνωριστικού του (*updFrame*), η οποία δίνεται στο σχήμα 3.3.

Αργότερα θα χρησιμοποιήσουμε τα ονόματα των λειτουργιών *push*, *getFrame* και *updFrame* για να αναφερθούμε σε αυτές χωρίς περαιτέρω επεξήγηση.

3.3.2 Προσωρινά σταματημένη εκτέλεση από δεδομένα (suspensions)

Ο ορισμός για τα suspensions είναι:

```
data Susp = Susp (CN, [Expr]) FrameId
```

$$
\begin{aligned}
&push :: Mem \times Frame \rightarrow Mem \\
&push\ (mem, frame) = mem' \\
&mem' = Mem\ \{\ memFrames = frames', lastFrameId = lastId\ \} \\
&frames' = frame : (memFrames\ mem) \\
&lastId = lastFrameId\ mem + 1
\end{aligned}
$$

Σχήμα 3.1: Push λειτουργία για τη μνήμη

$$
\begin{aligned}
&getFrame :: Mem \times FrameId \rightarrow Frame \\
&getFrame\ (mem, id) = frame \\
&frame = lookup\ id\ (memFrames\ mem)
\end{aligned}
$$

Σχήμα 3.2: GetFrame λειτουργία

$$
\begin{aligned}
&updFrame :: Mem \times FrameId \times Frame \rightarrow Mem \\
&upFrame\ (mem, frameId, frame) = mem\ \{memFrames = insert\ frameId\ frame\ (memFrames\ mem)\ \}
\end{aligned}
$$

Σχήμα 3.3: UpdateFrame λειτουργία

Ένα *Susp* περιέχει:

- Το όνομα του κατασκευαστή *CN* μαζί με τα ορίσματα του σε μια λίστα εκφράσεων.
- Ένα δείκτη με τη μορφή αναγνωριστικού (*FrameId*) για το πλαίσιο που είναι το περιβάλλον του suspension. Αυτό είναι το περιβάλλον μέχρι τη στιγμή που αυτό το suspension δημιουργήθηκε. Όταν η εκτέλεση του προγράμματος επιβάλλει την αποτίμησή του, αυτό θα είναι το περιβάλλον που θα χρησιμοποιήσει.

3.3.3 Τιμές

Μια τιμή μπορεί να είναι ένας ακέραιος (ή γενικά μια τιμή βασικού τύπου) ή ένα suspension.

```
data Value =
  VI Integer
| VC Susp
```

Από τον παραπάνω ορισμό, έχουμε ότι μια τιμή μπορεί να είναι είτε ένας ακέραιος (ή γενικά ένας τύπος βάσης) ή αναστολή κατασκευασμένων δεδομένων. Ενώ η πρώτη είναι προφανής, ειδικά για τους περισσότερους προγραμματιστές που χρησιμοποιούν αυστηρές γλώσσες, το τελευταίο είναι κάτι που συμβαίνει σε οκνηρές γλώσσες προγραμματισμού. Οι υπολογισμοί μπορεί να μην αποτιμώνται βαθιά και έτσι μπορούν να δημιουργήσουν thunks ή αφήνουν thunks που μπορεί να αποτιμηθούν αργότερα.

3.3.4 Πλαίσια

Η βασική μονάδα από τη μηχανή εκτέλεσης είναι το πλαίσιο. Ένα πλαίσιο περιέχει όλες τις απαραίτητες πληροφορίες για μια κλήση συνάρτησης. Δεσμεύεται κάθε φορά που πραγματοποιείται μια νέα κλήση συνάρτησης. Τότε, μπορεί να ενημερωθεί σε περίπτωση που περιέχει μια οκνηρή παράμετρο.

Ακολουθεί ο ορισμός του πλαισίου:

```
type FN      = String
type CaseID  = Int
type FrameId = Int
```

```
data Frame   = Frame {
  fName  :: FN,                  -- Function Name
  fArgs  :: [FrameArg],          -- Bindings of formals with actuals
  fSusps :: [(CaseID, Susp)],    -- Data deconstruction forced by 'case'
  fPrev  :: FrameId              -- pointer to previous stack frame
}
```

Ένα πλαίσιο έχει τις ακόλουθες πληροφορίες κατά τη διάρκεια της ζωής του:

- Το όνομα της συνάρτησης (*FN*). Αυτή η πληροφορία είναι σημαντική για την αναζήτηση του ορισμού της συνάρτησης στη δομή *functions map*, μια δομή που θα εξηγηθεί αργότερα, για να προχωρήσει η εκτέλεση του προγράμματος.
- Οι πραγματικές παράμετροι που συνδέονται με τις τυπικές παραμέτρους του ορισμού της συνάρτησης *'FN'*.
- Πληροφορίες για τα suspensions, δημιουργία thunk και αποτίμηση που επιβάλλεται από την 'case'. Κάθε *'CaseID'*, είναι μοναδικό στο σώμα της συνάρτησης και περιέχει δείκτες στα δικά του thunks.
- Ένα δείκτη με τη μορφή αναγνωριστικού στο προηγούμενο πλαίσιο (*fPrev*). Αυτό είναι το *περιβάλλον* στην υλοποίηση του διερμηνέα, το οποίο υπονοείται και δεν υπάρχει ρητά στο *state* του.

Ένα όρισμα που ζει στα 'fArg :: [FrameArg]', έχει τον ακόλουθο ορισμό:

```
data FrameArg =
  StrictArg { val :: Value }
| ByNameArg { expr :: Expr }
| LazyArg   { expr :: Expr, isEvaluated :: Bool, cachedVal :: Maybe Value }
```

Ένα όρισμα μπορεί να είναι:

- Αυστηρό και να περιέχει μόνο μια *τιμή*.
- Σε σημασιολογία της κλήσης κατ΄ όνομα και έτσι να περιέχει μόνο μια έκφραση, ένα thunk χωρίς επιλογή για απομνημόνευση.
- Lazy και έτσι να περιέχει μια έκφραση, με τον ίδιο τρόπο όπως τα ορίσματα call-by-name, αλλά επίσης και μια *σημαία*(flag) για το αν είναι αποτιμημένο ή όχι (εξασφαλίζοντας έτσι την ιδιότητα της αποτίμησης άπαξ, ή στα αγγλικά *single evaluation property*) και επίσης χώρο για την αποθηκευμλενη του τιμή (*cached value*).

3.4 Διερμηνέας

Ο διερμηνέας είναι μια συνάρτηση που λαμβάνει ένα *expression*, έναν *στατικό* χώρο μνήμης ενός προγράμματος που περιέχει πληροφορίες σχετικά με τους ορισμούς συναρτήσεων και ένα *state*, και φτάνει σε ένα *τελικό state* και *επιστρέφει μια τιμή*, όπου είναι μια έκφραση από τις εκφράσεις της σύνταξης που παρουσιάζεται σε προηγούμενη ενότητα. Ο ορισμός του *state* ακολουθεί αργότερα στην ενότητα.

Από εδώ και πέρα, η συνάρτηση *eval* αντιστοιχεί στη λειτουργία του διερμηνέα. Η δήλωση του *eval* η λειτουργία εμφανίζεται στο σχήμα 3.4:

Το *State* της *eval* φαίνεται στο σχήμα 3.5, όπου οι ορισμοί για *Mem*, *FrameId* δίνονται νωρίτερα στο κεφάλαιο. Το πεδίο *NRFrames* δίνει τον αριθμό από τα πλαίσια που δεσμεύτηκαν έως τώρα.

Η δομή *FunctionsMap* είναι μια δομή που δημιουργείται κατά το χρόνο μεταγλώττισης και παραμένει ζωντανή στο χρόνο εκτέλεσης. Δεν αλλάζει κατά την εκτέλεση του προγράμματος και αποτελείται από ζευγάρια κλειδιού-τιμής, όπου:

$eval :: Expr \times FunctionsMap \times State \rightarrow State \times Value$

Σχήμα 3.4: Δήλωση της συνάρτησης *eval*

$State :: (Mem, FrameId, NRFrames)$

Σχήμα 3.5: Κατάσταση (state) του διερμηνέα

- Τα κλειδιά είναι τα ονόματα από τις top-level συναρτήσεις.
- Οι τιμές είναι ένα ζεύγος τυπικών παραμέτρων με τις στατικές τους πληροφορίες (είδος αποτίμησης, τύπος) της συνάρτησης καθώς και το σώματ της συνάρτησης.

Ο ορισμός δίνεται στο σχήμα 3.6:

```
type FN = String
data FunctionsMap = Map FN ([Formal], Expr)
```

Σχήμα 3.6: FunctionsMap

Το αποτέλεσμα του διερμηνέα είναι το αποτέλεσμα της εκτέλεσης του σώματος της συνάρτησης κορυφαίου επιπέδου *main*. Αυτή η συνάρτηση είναι μια ειδική περίπτωση συνάρτησης και θεωρείται ότι δεν έχει κανένα όρισμα. Έτσι, η αρχική έκφραση *expr0* είναι:

```
(_, expr0) = Map.lookup "main" functionsMap
```

Το *αρχικό state* του διερμηνέα είναι:

$State0 = (mem0, frameId0, nr\_frames0)$

Σχήμα 3.7: Αρχικό state του διερμηνέα

Στο αρχικό state, που φαίνεται στο σχήμα 3.7, έχουμε:

- Το αρχικό state της μνήμης *mem0*, είναι:

```
-- cTOPFRAMED: last frame id available when execution starts
mem0   = push (Mem Map.empty 0) frame0
frame0 = Frame "main" [] [] cTOPFRAMEID
```

- Το αρχικό id από το πρώτο frame στη μνήμη που είναι ελεύθερο:

```
frameId0 = lastFrameId mem0
```

- Ο αρχικός αριθμός πλαισίων που έχουν δεσμευτεί, πριν ξεκινήσει η εκτέλεση της eval, είναι:

```
nr_frames = 1
```

καθώς έχουμε *μια* δέσμευση πλαισίου για τη συνάρτηση "main".

Σε αυτό το σημείο, έχουμε τα πάντα έτοιμα. Δηλώσαμε τη συνάρτηση του διερμηνέα *eval* και έχουμε μια αρχική να ξεκινήσει η εκτίμησή μας με. Τώρα, ας δούμε την εκτέλεση κάθε έκφρασης του δένδρου σύνταξης της γλώσσας μας που παρουσιάζεται στην ενότητα της σύνταξης. Οι μικρές λεπτομέρειες εφαρμογής παραλείπονται για λόγους σαφήνειας. Αρχικά υποθέτουμε παντού την οκνηρότητα, ενώ αργότερα θα υπάρχει μια επισκόπηση του τρόπου με τον οποίο ο διερμηνέας εκτελείται παρουσία λιστών ML.

Η δομή *FunctionsMap* επίσης παραλείπεται ως όρισμα της eval. Υποθέτουμε ότι ψάχνουμε αυτό για τα τυπικές παραμέτρους και όταν θέλουμε να μεταβούμε στο σώμα μιας συνάρτησης, π.χ. στην κλήση συναρήσεων.

Σε κάθε βήμα εκτέλεσης η *eval* αναζητά την τρέχουσα κατάσταση. Όπως έχουμε ήδη αναφέρει το state είναι ένα πεδίο με:

```
State = (Mem, FrameId, NRFrames)
-- FrameId: id of the current frame
```

, και έτσι κοιτάζοντας το frame του τρέχοντος state, παίρνουμε το τρέχον περιβάλλον, έχουμε:

```
thisFrame = getFrame mem frameId
Frame caller funArgs susps prevFrameId = thisFrame
```

Με αυτόν τον τρόπο, έχουμε πληροφορίες για:

- *caller*: το όνομα της συνάρτησης που καλεί και είμαστε τώρα μέσα στο σώμα της,
- *funArgs*: τα τρέχοντα ορίσματα της τρέχουσας συνάρτησης που είναι κατασκευασμένα στο πλαίσιο (δείτε αργότερα πώς τα χτίζουμε όταν γίνεται κλήση συνάρτησης),
- *susps*: προσωρινά σταματημένες εκτέλεσεις για δεδομένα που η εκτέλεσή τους επιβάλλεται από το pattern matching του case,
- *prevFrameId*: Το αναγνωριστικό από το προηγούμενο πλαίσιο, σαν να είχαμε ένα δείκτη στο προηγούμενο πλαίσιο.

Αφού αποκτηθούν οι απαραίτητες πληροφορίες από την τρέχουσα κατάσταση, ο διερμηνέας χειρίζεται κάθε μια από τις εκφράσεις με τον ακόλουθο τρόπο, που παρουσιάζεται στις επόμενες ενότητες.

3.4.1 Αποτίμηση μεταβλητής

Υπενθυμίζουμε σε αυτό το σημείο ότι μια μεταβλητή δεσμευμένη στην αντιστοίχιση μοτίβων δεν αξιολογείται εδώ λόγω του μετασχηματισμού της σε constructor projection που περιγράψαμε προηγουμένως, και επομένως σε αυτό το τμήμα μελετάμε την αποτίμηση μεταβλητών που είναι δεμένες με τις τυπικές παραμέτρους μιας συνάρτησης.

Μια μεταβλητή ανώτατου επιπέδου υπάρχει στις τυπικές παραμέτρους ενός ορισμού συνάρτησης. Στους όρους *πλαισίου*, ψάχνουμε για το σωστό *FrameArg* στο πεδίο *funArgs*, που αναφέρθηκε προηγουμένως όταν εξηγήσαμε την *τρέχουσα κατάσταση* του διερμηνέα.

Η διαδικασία είναι η ακόλουθη:

- Πρώτον, βρίσκουμε τη θέση της μεταβλητής στο τρέχον frame.
  - Ψάχνουμε τη συνάρτηση *που καλεί* στη δομή *FunctionsMap* για να βρούμε την υπογραφή (*signature*) της συνάρτησης.
  - Άπαξ και βρούμε την υπογραφή, στη συνέχεια βρίσκουμε τη θέση της μεταβλητής στις τυπικές παραμέτρους της συνάρτησης, με χρήση του ονόματός της, δηλ. τον *δείκτη (i)* του ορίσματος.
- Δεδομένου του δείκτη που υπολογίσαμε νωρίτερα, βρίσκουμε το κατάλληλο όρισμα στο τρέχον πλαίσιο.

Στη συνέχια έχουμε τρεις περιπτώσεις όσα και τα είδη αποτίμησης.
Σε αυτό το σημείο, δίνουμε τα v' and s' για: $eval(v,s) = (v',s')$

- cbv: Αποτίμηση μεταβλητής για call-by-value μεταβλητές φαίνονται στο σχήμα 3.8.

$$(v', s') = (val, s),$$
$$where$$
$$v = StrictArg\ val$$

Σχήμα 3.8: Αποτίμηση μεταβλητής για call-by-value

$$(v', s') = (val, s),\ where$$
$$v = ByNameArg\ expr,$$
$$(val, s'') = eval\ expr\ (mem, prevFrameId, nr\_frames)$$

Σχήμα 3.9: Αποτίμηση μεταβλητής για call-by-name

$$v = LazyArg\ e\ b\ val$$

– b is *true*:

$$(v', s') = (val, s)$$

– b is *false*:

$$(v', s') = (val', (updFrame\ mem'\ frameId\ frame', frameId, nr\_frames')), where$$
$$(val', (mem', frameId', nr\_frames')) = eval\ e\ (mem, prevFrameId, nr\_frames)$$
$$frame'[funArgs] = (funArgs[i] = LazyArg\ e\ true\ val')$$

Σχήμα 3.10: Αποτίμηση μεταβλητής για call-by-need

- cbn e.o.: Αποτίμηση μεταβλητής για call-by-name φαίνεται στο σχήμα 3.9.
- lazy e.o.: Στο σχήμα 3.10) έχουμε *δυο* υποπεριπτώσεις. Αυτές είναι αν το b είναι *true* ή *false*, δηλ. ο οκνηρός όρος είναι αποτιμημένος και αποθηκευμένος ή μη αποτιμημένος. Στην τελευταία περίπτωση, ο διερμηνέας χρειάζεται να αποτιμήσει τη μεταβλητή και να ενημερώσει το πλαίσιο, όπως φαίνεται και στο σχήμα.

  Σημειώνουμε εδώ ότι όταν έχουμε αλλαγή στην κατάσταση της μνήμης, και πιθανή αποτίμηση της έκφρασης π[ροκαλεί τη δημιουργία νέου πλαισίου ενημερώνουμε την κατάσταση της μνήμης και το μετρητή των πλαισίων.

3.4.2 Κλήση συνάρτησης

Σε αυτή την ενότητα, περιγράφεται η λειτουργία του διερμηνέα για κλήση συνάρτησης, όπου η έκφραση είναι: $Expr = Call\ callee\ actuals$, όπου έχουμε τις πληροφορίες για το όνομα της συνάρτησης *που καλείται* και τις πραγματικές παραμέτρους της κλήσης.

$$makeArgs :: [Actual] \times [Formal] \times State \rightarrow [FrameArg] \times State,$$

Σχήμα 3.11: Δήλωση της συνάρτησης κατασκευής ορισμάτων

$$(v, state') = eval\ actual\ state$$
$$frameArg = StrictArg\ v$$

Σχήμα 3.12: Κατασκευή call-by-value ορίσματος στο πλαίσιο

$$frameArg = ByNameArg\ actual$$

Σχήμα 3.13: Κατασκευή call-by-name ορίσματος στο πλαίσιο

$$frameArg = LazyArg\ actual\ false\ empty$$

Σχήμα 3.14: Κατασκευή οκνηρού ορίσματος στο πλαίσιο

$$eval\ (Call\ callee\ actuals, s) = (val, s'''),$$
where
$$(formals, funBody) = lookup\ callee\ FunctionsMap$$
$$(frameArgs, (mem', \_, nr\_frames')) = makeArgs\ actuals\ formals\ state$$
$$newFrame = Frame\ callee\ frameArgs\ []\ frameId$$
$$mem'' = push\ mem'\ newFrame\ (nr\_frames + 1)\ frameId$$
$$s' = (mem'', lastFrameId\ mem'', nr\_frames + 1)$$
$$(val, s'') = eval\ funBody\ s'$$
$$(s'' = (mem''',, nr\_frames''))$$
$$s''' = (mem''', frameId, nr\_frames'')$$

Σχήμα 3.15: Λειτουργική σημασιολογία για την κλήση συνάρτησης

Πρώτον, θα παρουσιάσουμε μια συνάρτηση για την κατασκευή των επιχειρημάτων, λαμβάνοντας υπόψη τις πραγματικές παραμέτρους της κλήσης και τις τυπικές παραμέτρους του ορισμού της συνάρτησης που βρίσκεται στη δομή *FunctionsMap*.

Η συνάρτηση, που ονομάζεται *makeArgs* έχει τη μορφή, που φαίνεται στο σχήμα 3.11, όπου οι πραγματικοί και οι τυπικοί παράμετροι εξηγούνται παραπάνω και το *State* αναπαριστά την τρέχουσα κατάσταση του διερμηνέα.

Για κάθε πραγματική που αντιστοιχεί σε κάθε επίσημη παράμετρο, έχουμε τα ακόλουθα, ανάλογα αν η παράμετρος είναι σε cbv, cbn ή lazy:

- cbv: Όπως φαίνεται στο σχήμα 3.12 η συνάρτηση επιστρέφει *(frameArg, state')*, όπου το *frameArg* προστίθεται στο *[FrameArg]* και το *state'* είναι η είσοδος στο όρισμα state της συνάρτησης *makeArgs*.
- cbn: Όπως φαίνεται στο σχήμα 3.13, το state παραμένει αμετάβλητη και έτσι το *(frameArg, state)* επιστρέφεται.
- lazy: Όπως φαίνεται στο σχήμα 3.14 το state δεν αλλάζει, με αποτέλεσμα να επιστρέφεται το *(frameArg, state)*.

Όπως φαίνεται παραπάνω, μόνο τα επιχειρήματα cbv εκτελούνται σε αυτό το σημείο και η αλλαγή μνήμης που προκαλούν, επιστρέφεται στην κατάσταση της συνάρτησης *eval*.

Τώρα, είναι καιρός να ξαναρχίσουμε την παρουσίαση σχετικά με την κλήση λειτουργίας στον διερμηνέα. Η λειτουργική σημασία μιας κλήσης λειτουργίας εμφανίζεται στο σχήμα 3.15. Κάθε φορά που έχουμε μια κλήση συνάρτησης, ένα νέο πλαίσιο δεσμεύεται στη μνήμη. Τότε, ο διερμηνέας αποτιμά το σώμα της συνάρτησης και τελικά επαναρυθμίζουμε το περιβάλλον για την επόμενη λειτουργία που χειρίζεται ο διερμηνέας.

3.4.3 Ταίριασμα προτύπων

Όπως έχει ήδη αναφερθεί, διαχωρίζουμε μεταξύ pattern matching σε ακεραίους και pattern matching σε δεδομένα. Μια διακλάδωση (branch) στο pattern matching είναι:

```
type Branch  = (Pattern, Expr)
data Pattern = CPat { tag :: CN, vars :: [VN] } -- pattern matching on constructors
             | IPat { pattVal :: Int }          -- pattern matching on integers
```

$$caseExpr = Case\ caseId\ e\ branches$$

Σχήμα 3.16: Η έκφραση case

$$\begin{aligned} & eval\ (e, s) = (e', s'), \\ & where \\ & \quad s' = (mem', savedFrameId, n),\ and \\ & \quad e' = VI\ i,\ integer\ pattern\ matching, or \\ & \quad e' = VC\ c,\ constructor\ pattern\ matching \\ & \quad c = Susp\ (cn, \_)\ \_ \end{aligned}$$

Σχήμα 3.17: Αποτίμηση του όρου που εξετάζεται

$$\begin{aligned} & eval\ (caseExpr, s) = (eval\ e''\ s', s'), \\ & where \\ & \quad e'' = lookup\ (IPat\ i)\ cases \\ & \quad s'\ is\ the\ state\ after\ scrutinee's\ evaluation. \end{aligned}$$

Σχήμα 3.18: Λειτουργική σημασιολογία για pattern matching σε ακεραίους

Η σύνταξη της έκφρασης 'case' φαίνεται στο σχήμα 3.16. Για την αποτίμηση της έκφρασης αυτής, ο διερμηνέας κάνει τα επόμενα βήματα:

- Πρώτα, αποτιμά τον όρο που εξετάζεται (scrutinee), η αποτίμηση του οποίου φαίνεται στο σχήμα 3.17.
- Στη συνέχεια, στο σχήμα 3.18, φαίνεται πώς συμπεριφέρεται ο διερμηνέας για pattern matching πάνω σε ακεραίους.

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{caseExpr}, s) = (\mathit{eval}\ e''\ s'', s''),\\
&\mathit{where}\\
&\quad \mathit{pattIndex} = \mathit{indexOfPattern}\ cn\ \mathit{patterns}\\
&\quad (\_, e'') = \mathit{branches}\ [\mathit{pattIndex}]\\
&\quad \mathit{susps}' = (\mathit{caseId}, c)\ :\ \mathit{susps}\\
&\quad \mathit{frame}' = \mathit{Frame}\ \mathit{caller}\ \mathit{funArgs}\ \mathit{susps}'\ \mathit{prevFrameId}\\
&\quad s'' = (\mathit{updFrame}\ \mathit{mem}'\ \mathit{frameId}\ \mathit{frame}',\ \mathit{frameId},\ n)
\end{aligned}
$$

Σχήμα 3.19: Λειτουργική σημασιολογία για pattern matching σε κατασκευαστές δεδομένων

- Τέλος, το pattern matching πάνω σε κατασκευαστές δεδομένων φαίνεται στο σχήμα 3.19.

Στους παραπάνω ορισμούς, η *indexOfPatterns* δίνει σαν έξοδο το δείκτη στη λίστα των διακλαδώσεων, για να γνωρίζει ο διερμηνέας ποια θα είναι η επόμενη έκφραση που θα αποτιμήσει.

Σημειώνουμε, σε αυτό το σημείο ότι, παρόλο που η υλοποίηση μας δουλεύει με λίστες, αντίστοιχα η λειτουργία μπορεί να γενικευτεί για οποιοδήποτε κατασκευαστή δεδομένων, καθώς αυτοί είναι επί της ουσίας συναρτήσεις. Το τελευταίο είναι προφανές αν αναλογιστούμε ότι δουλεύουμε σε μια συναρτησιακή γλώσσα, όπου οι συναρτήσεις είναι πολίτες πρώτης κατηγορίας.

3.4.4 Εφαρμογή κατασκευαστή δεδομένων

Στην περίπτωση που έχει εφαρμογή ενός *lazy κατασκευαστή δεδομένων* (lazy data constructor), αυτό είναι ισοδύναμο με την δέσμευση μνήμης ενός thunk. Στην περίπτωσή μας, αυτό είναι μια δέσμευση ενός suspension, όπως εξηγήθηκε προηγουμένως. Πιο συγκεκριμένα, η *eval* για μια εφαρμογή όρου σε lazy data constructor φαίνεται στο σχήμα 3.20.

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{ConstrF}\ \mathit{tag}\ \mathit{exprs}, s) = (\mathit{VC}\ (\mathit{Susp}\ (\mathit{tag}, \mathit{exprs})\ \mathit{frameId}), s)\\
&, \mathit{where}\\
&\quad \mathit{frameId}:\ \mathit{the\ id\ of\ the\ current\ frame\ existing\ in\ the\ interpeter's\ state},\\
&\quad \mathit{tag}:\ \mathit{the\ name\ of\ the\ constructor}.
\end{aligned}
$$

Σχήμα 3.20: Λειτουργική σημασιολογία για εφαρμογή οκνηρών κατασκευαστών

3.4.5 Προβολές κατασκευαστών

Σε αυτήν την περίπτωση, ο διερμηνέας έχει συναντήσει μια μεταβλητή που δεσμεύεται από pattern matching. Όπως αναφέρθηκε προηγουμένως, η αποτίμηση ενός όρου αναγκάζεται, όταν πρέπει να αποτιμηθούν τέτοιες μεταβλητές. Η λειτουργία της *eval* για την *προβολή κατασκευαστή* εμφανίζεται στο σχήμα 3.21.

Πρώτον, βρίσκουμε το κατάλληλο *susp* μέσα στο πεδίο *susps* του τρέχοντος πλαισίου. Στη συνέχεια, από τη λίστα εκφράσεων *el* βρίσκουμε τον κατάλληλοο κατασκευαστή στη θέση *cpos* . Μετά την αποτίμηση της έκφρασης έχουμε την τιμή-αποτέλεσμα και το επόμενο state του διερμηνέα μας.

$$
\begin{aligned}
&\quad eval\ (CProj\ caseId\ cpos, s) = (val, s'),\\
&where\\
&\quad s' = mem'\ frameId\ nr\_frames'\\
&\quad susp = lookup\ cid\ susps\\
&\quad Susp\ (\_, el)\ savedFrameId = susp\\
&\quad e' = el[cpos]\\
&\quad (val, (mem', \_, nr\_frames')) = eval\ (e', s)
\end{aligned}
$$

Σχήμα 3.21: Λειτουργική σημασιολογία για την προβολή κατασκευαστή

Κεφάλαιο 4

Ανάλυση: Σε αναζήτηση ευκαιριών βελτιστοποίησης

Σε αυτό το κεφάλαιο, παρουσιάζουμε την ανάλυση που εκτελείται προκειμένου να αποκαλυφθούν οι πραγματικές θέσεις κλήσης ουράς. Πρώτον, κάνουμε ορισμένους σημαντικούς ορισμούς υψηλού επιπέδου για να μπορέσει ο αναγνώστης να παρακολουθήσει τις επόμενες ενότητες.

Η είσοδος της στατικής ανάλυσης μας είναι η γλώσσα πυρήνα πρώτης τάξης, που έχει δοθεί στο αντίστοιχο κεφάλαιο, μαζί με τη λειτουργική σημασιολογία που παρουσιάστηκε στο προηγούμενο κεφάλαιο. Η έξοδος είναι κατάλληλα επισημειωμένες κλήσεις συναρτήσεων, όπου ο εμπλουτισμένος διερμηνέας του επόμενου κεφαλαίου θα διαχειριστεί κατάλληλα. Η ανάλυση που παρουσιάζεται σε αυτό το κεφάλαιο αποκαλύπτει που θα εφαρμοστεί η βελτιστοποίηση, ενώ στο επόμενο κεφάλαιο θα παρουσιαστεί η λειτουργία κατά το χρόνο εκτέλεσης.

4.1 Κλασική βελτιστοποίηση κλήσεων ουράς

Στο σημείο αυτό, αρχικά δίνουμε δυο ορισμούς: οφείλουμε να διευκρινήσουμε ότι ο όρος κλήσεις ουράς δεν είναι ο ίδιος με τον όρο βελτιστοποιήσιμες κλήσεις ουράς. Ενώ οι δυο αυτοί όροι ταυτίζονται σε μια γλώσσα προγραμματισμού με αυστηρή σημασιολογία, αυτό δεν συμβαίνει σε μια γλώσσα υπό την παρουσία της οκνηρής αποτίμησης.

4.1.1 Ορισμοί

Ορισμός 4.1 Μια κλήση συνάρτησης είναι σε *θέση κλήσης ουράς*, ή είναι *κλήση ουράς*, *αν και μόνο αν* η εκτέλεσή της πραγματοποιείται ακριβώς πριν επιστρέψει η καλούσα συνάρτηση.

Ορισμός 4.2 Μια κλήση ουράς είναι *βελτιστοποιήσιμη*, ή είναι *πραγματική κλήση ουράς*, *αν και μόνο αν* αυτή είναι σε θέση κλήσης ουράς (ορισμός 4.1) και ικανοποιεί τους κανόνες που παρουσιάζονται στην ενότητα 4.1.3.

Στις επόμενες ενότητες, παρουσιάζουμε τη διαδρομή από τις *κλήσεις συναρτήσεων*, που υπάρχουν στη γλώσσα πυρήνα μας, στις *θέσεις κλήσεων ουράς*, και από εκεί στις *βελτιστοποιήσιμες κλήσεις ουράς*. Οι τελευταίες επισημειώνονται από την ανάλυσή μας στη γλώσσα πυρήνα, και συγκεκριμένα με την επισημείωση *TailCall*. Όταν ο διερμηνέας μας συναντά αυτές τις εκφράσεις, πραγματοποιεί τη βελτιστοποίηση κατά το χρόνο εκτέλεσης.

4.1.2 Αναλυση ροής ελέγχου προγράμματος: Εντοπίζοντας τις κλήσεις ουράς

Για μια γλώσσα προγραμματισμού με αυστηρή σημασιολογία, αυτό το βήμα είναι ακριβώς ό,τι χρειάζεται: φανερώνει τις κλήσεις ουράς, οι οποίες είναι όλες βελτιστοποιήσιμες. Αντίθετα, σε μια γλώσσα με πολλαπλά είδη αποτιμήσεων, και ειδικά με την παρουσία οκνηρής αποτίμησης, αυτό δεν είναι αρκετό, καθώς η αποτίμηση των οκνηρών όρων δε γνωρίζουμε πότε θα αποτιμηθεί, με αποτέλεσμα να δραπετεύουν από τη στατική τους εμβέλεια.

Πρόκειται για μια τοπική ανάλυση, η οπία πραγματοποιείται στο σώμα μιας συνάρτησης. Η ανάλυση συνοψίζεται στους παρακάτω κανόνες:

- Το σώμα μιας συνάρτησης είναι κλήση ουράς.
- Όταν μια if/case έκφραση είναι κλήση ουράς, τότε οι διακλαδώσεις τους είναι επίσης κλήσεις ουράς. Στην περίπτωση ενός 'if' για παράδειγμα, τα then/else είναι σε θέσεις κλήσεων ουράς.
- Τίποτα άλλο δεν είναι σε θέση κλήσεων ουράς.

Για να γίνει πιο σαφές το παραπάνω, χρησιμοποιούμε το εξής παράδειγμα:

```
foo n =
  if n > 0
    then n * foo (n-1)
    else 1
```

Το σώμα της συνάρτησης 'foo' είναι tail call. Αυτό σημαίνει ότι το 'if', που είναι η πιο πάνω έκφραση στο σώμα της συνάρτησης, είναι σε θέση κλήσης ουράς. Έτσι, τα then/else είναι κλήσεις ουράς. Στη 'then' πρόταση όμως, όπου πραγματοποιείται ένας πολλαπλασιασμός, η κλήση στη 'foo' δεν είναι σε θέση κλ'ησης ουράς, σύμφωνα με τον τρίτο κανόνα.

Αλλά αν είχαμε το επόμενο πρόγραμμα:

```
foo n res =
  if n > 0
    then foo (n - 1) (n * res)
    else res
```

τότε η κλήση της συνάρτησης στη then πρόταση είναι σε θέση κλήσης ουράς, καθώς το then ειναι σε θέση κλήσης ουράς.

4.1.3 Ανάλυση οδηγούμενη από τα δεδομένα: Βελτιστοποιήσιμες κλήσεις ουράς

Μετά την αποκάλυψη των θέσεων κλήσης ουράς, που προκύπτει από τη ροή ελέγχου του προγράμματος, η οποία παρουσιάζεται στην προηγούμενη ενότητα, τώρα εφαρμόζουμε ορισμένους κανόνες για να αποκαλυφθούν οι πραγματικά βελτιστοποιήσιμες κλήσεις ουράς, ανάλογα με το είδος αποτίμησης που έχουν οι όροι στην καλούσα και καλόυμενη συνάρτηση αντίστοιχα. Μετά από όλα, βελτιστοποίηση κλήσης ουράς πρόκειται για το πέρασμα του ελέγχου από τον καλούντα στον καλούμενο. Αυτό το βήμα παραλείπεται στις γλώσσες που χρησιμοποιούν την κλήση κατ' αξία, όπου υπάρχει μόνο ένα είδος αποτίμησης, το οπίο επιτρέπει τη βελτιστοποίηση εντελώς απροβλημάτιστα.

Παρόλο που φαίνεται ότι οι κανόνες εφαρμόζονται σε ένα δεύτερο πέρασμα ανάλυσης, αυτό γίνεται για λόγους σαφήνειας στο κείμενο. Στην πραγματικότητα μπορούμε να το κάνουμε αυτό σε ένα πέρασμα, σε μια ενιαία ανάλυση, ακριβώς μόλις η ανάλυση ροής ελέγχου αποφασίσει, αν η κλήση που μελετάται είναι κλήση ουράς.

Οι κανόνες για ένα πιθανό tail-call, που αποκαλύφθηκε από την ανάλυση της προηγούμενης υποενότητας, για να είναι βελτιστοποιήσιμο tail-call είναι:

- Οι πραγματικές παράμετροι στην κλήση δεν εξαρτώνται από τις τυπικές παραμέτρους του της καλούσας συνάρτησης. Για παράδειγμα, αυτές οι πραγματικές παράμετροι μπορεί να είναι σταθερές.
- Οι πραγματικές παράμετροι στην κλήση είναι μεταβλητές. Εάν είναι μεταβλητές μπορούμε προβλέψουμε στατικά αν και πότε θα είναι δυνατή η μετάλλαξη του πλαισίου κατά τη διάρκεια εκτέλεσης. Ο αναγνώστης μπορεί επίσης να αναφερθεί στο σχήμα 5.5 για τις πιθανές περιπτώσεις. Πραγματοποιούμε τη βελτιστοποίηση σχεδόν σε όλες τις περιπτώσεις, εκτός από δυο υποπεριπτώσεις.
- Η πραγματική παράμετρος είναι μια έκφραση, αλλά είναι σε θέση κλήσης κατ' αξία. Σε αυτή την περίπτωση, υπάρχουν δύο υποπεριπτώσεις:

– Η έκφραση έχει βασικό τύπο (π.χ. ακέραιο). Αυτή είναι η ευκολότερη υποπερίπτωση, καθώς αυτές οι εκφράσεις, όταν αποτιμώνται, αποτιμώνται απευθείας σε τιμή βασικού τύπου, χωρίς να αφήνουν thunks που δεν έχουν αποτιμηθεί. Σε αυτήν την περίπτωση, θα μπορούσαμε επίσης να έχουμε δεδομένα όπως οι λίστες της ML, τα οποία επίσης αποτιμώνται σε βάθος.
– Η έκφραση είναι κατασκευασμένα δεδομένα. Σε αυτή την περίπτωση, βρίσκουμε όλες τις μεταβλητές μέσα στην έκφραση. Εάν αυτές οι μεταβλητές αποτιμώνται απευθείας σε τιμή, χωρίς να αφήνουν thunks, (τιμές με τύπο βάσης ή βαθιά αποτιμημένα δεδομένα), τότε αυτή η κλήση συνάρτησης είναι βελτιστοποιήσιμη κλήση ουράς.
– Οι κλήσεις ουράς που δεν υπόκεινται στους παραπάνω κανόνες δεν είναι βελτιστοποιήσιμες.

Κεφάλαιο 5

Υλοποίηση βελτιστοποίησης κλήσεων ουράς στο χρόνο εκτέλεσης

In this chapter, we will present how the optimisations are implemented in the interpreter. More specifically, we will present how *eval* operates, when a tail call has come up (figure 5.1).

$eval\ (e, s) = (e', s'),$
$\quad where\ e = TailCall\ callee\ actuals$

Σχήμα 5.1: Επισημειωμένες εκφράσεις

5.1 Λειτουργία για την κλασική βελτιστοποίηση κλήσεων ουράς

Classic tail call evaluation contains one major *operation*, which makes the optimisation feasible: this is *frame mutation*. This operation includes the replacement of the current frame's arguments with the arguments necessary for the construction of the frame for the next function call. For this operation, it is mandatory that the caller's arguments do not escape and for those that escape we correctly passed them to the new frame.

Η κλασική βελτιστοποίηση κλήσης ουράς περιέχει μια κύρια *λειτουργία*, η οποία κάνει τη βελτιστοποίηση εφικτή: αυτή είναι η *μετάλλαξη πλαισίου*(frame mutation). Αυτή η ενέργεια περιλαμβάνει την αντικατάσταση των ορισμάτων τρέχοντος πλαισίου με τα απαραίτητα ορίσματα για την κατασκευή του πλαισίου που αντιπροσωπεύει την επόμενη κλήση λειτουργίας. Για αυτή τη λειτουργία, είναι υποχρεωτικό τα ορίσματα του καλούντος να μην 'δραπετεύσουν' και γι 'αυτά που 'δραπετεύουν' να τα μεταβιβάσουμε σωστά στο νέο πλαίσιο.

Η *eval* για το *TailCall* ακολουθεί στο σχήμα 5.2; Η λειτουργικότητα του *checkMutate* θα εξηγηθεί αργότερα στην ενότητα. Οι υπόλοιπες μεταβλητές που φαίνονται στο σχήμα είναι:

- *formals* είναι οι τυπικές παράμετροι του ορισμού της κληθείσας συνάρτησης,
- *actuals* είναι οι πραγματικές παράμετροι της κληθείσας συνάρτησης,
- *funArgs* είναι τα ορίσματα μέσα στον τρέχον πλαίσιο του καλούντα,
- and *s* είναι το τρέχον state του διερμηνέα.

Σε αυτό το σημείο, παρουσιάζουμε τον ορισμό της συνάρτησης *checkMutate*, η οποία είναι μια συνάρτηση της οποίας αν της δοθεί το τρέχον state του διερμηνέα εκτελεί το frame mutation και παράγει το επόμενο state του διερμηνέα. Η δήλωση της συνάρτησης δίνεται στο σχήμα 5.3 και η λειτουργική σημασιολογία της συνάρτησης δίνεται στο σχήμα 5.4.

Ακολουθεί η περιγραφή της βοηθητικής συνάρτησης *mutate*, η λειτουργική σημασιολογία της οποίας φαίνεται στο σχήμα 5.5. Από το σχήμα φαίνεται τα παρακάτω για την είσοδο της συνάρτησης. Η συνάρτηση λοιπόν πάιρνει:

- Τις επίσημες παραμέτρους της καλούσας,

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{TailCall\ callee\ actuals}, s) = (e', s'),\\
&\textit{where}\\
&\quad (\mathit{formals}, \mathit{funBody}) = \mathit{lookup\ FunctionsMap\ callee}\\
&\quad e' = \mathit{eval}\ (\mathit{funBody}, s')\\
&\quad s' = \mathit{checkMutate\ formals\ actuals\ funArgs\ s}
\end{aligned}
$$

Σχήμα 5.2: Λειτουργική σημασιολογία για βελτιστοποίηση κλήσεων ουράς

$$\mathit{checkMutate} :: [\mathit{Actual}] \times [\mathit{Formal}] \times [\mathit{FrameArg}] \times \mathit{State} \to \mathit{State}$$

Σχήμα 5.3: Δήλωση της λειτουργίας *checkMutate*

$$
\begin{aligned}
&\mathit{checkMutate}\ (\mathit{actuals}, \mathit{formals}, \mathit{funArgs}, s) = s',\\
&\textit{where}\\
&\quad (\mathit{mem}, \mathit{frameId}, \mathit{nr\_frames}) = s\\
&\quad (\mathit{callerFormals}, \_) = \mathit{lookup\ callee\ FunctionsMap}\\
&\quad (\mathit{funArgs}', s'') = \mathit{mutate\ callerFormals\ formals\ funArgs\ actuals\ s}\\
&\quad (\mathit{mem}', \mathit{frameId}', \mathit{nr\_frames}') = s''\\
&\quad \mathit{frame}'' = \mathit{Frame\ callee\ funArgs}'\ []\ \mathit{prevFrameId}\\
&\quad s' = (\mathit{updFrame\ mem}'\ \mathit{frameId\ frame}'', \mathit{frameId}, \mathit{nr\_frames}')
\end{aligned}
$$

Σχήμα 5.4: Λειτουργική σημασιολογία για τη λειτουργία *checkMutate*

- Τις επίσημες παραμέτρους της κληθείσας,
- τα ορίσματα του τρέχοντος πλαισίου,
- τις πραγματικές παραμέτρους της κλήσης,
- την τρέχουσα κατάσταση του διερμηνέα,

και παράγει:

- Τα ορίσματα πλαισίου για το νέο πλαίσιο. Σημειώνουμε εδώ ότι θέλουμε να μετατρέψουμε τις πραγματικές παραμέτρους της κλήσης της λειτουργίας στις κατάλληλες παραμέτρους πλαισίου, χρησιμοποιώντας τις υπογραφές λειτουργίας του καλούντος και του καλούμενου. Συγκεκριμένα, μας ενδιαφέρει η μορφή των ορισμάτων και επομένως πρέπει να γνωρίζουμε το είδος αποτίμησης των πραγματικών παραμέτρων.
- Η επόμενη κατάσταση για τον διερμηνέα. Επειδή η αποτίμηση *μπορεί* να πραγματοποιηθεί, ενώ αυτή η λειτουργία εκτελείται (για παράδειγμα, πρέπει να μετατρέψουμε ένα οκνηρό όρισμα σε αυστηρό και έτσι να κάνουμε την αποτίμηση πριν μεταβούμε στο σώμα της συνάρτησης), πρέπει να αλλάξουμε την κατάσταση μνήμης και να μετρήσουμε σωστά τα πλαίσια για το χειρισμό της βελτιστοποίησης που ακολουθεί στο επόμενο κεφάλαιο.

Καθώς η γλώσσα είναι πρώτης τάξης, δεν υποστηρίζει μερικές εφαρμογές όρων, και επομένως ο αριθμός των παραμέτρων στην λίστα πραγματικών παραμέτρων είναι ίση με τον αριθμό των τυπικών

της συνάρτησης του καλούντος. Η συνάρτηση *mutate* τερματίζεται όταν οι δύο αυτές λίστες είναι κενές στην ίδια επανάληψη, διαφορετικά ‘ρίχνει’ μια εξαίρεση. Η εκτέλεσή της αρχίζει και συνεχίζει μέχρι να τερματίσει επεξεργάζοντας κάθε πραγματική παράμετρο της κλήσης. Το αποτέλεσμα είναι μια λίστα που συγκεντρώνει το αποτέλεσμα κάθε επανάληψης.

$$
\begin{aligned}
&\mathit{getIndex} :: \mathit{VN} \times \mathit{EvaluationOrder} \times \mathit{Type} \times [\mathit{Formal}] \to \mathit{Index}\\
&\mathit{getIndex}\ (\mathit{vn},\ \mathit{eo},\ \mathit{type})\ \mathit{fs} = \mathit{elemIndex}\ (\mathit{vn}, (\mathit{eo}, \mathit{type}))\ \mathit{fs}\\
&\\
&\mathit{getEvaluationOrder} :: \mathit{Formal} \to \mathit{EvaluationOrder}\\
&\mathit{getEvaluationOrder} :: (\_, (\mathit{eo}, \_)) = \mathit{eo}\\
&\\
&\mathit{signOfVar} :: [\mathit{Formal}] \times \mathit{VN} \to \mathit{EvalutionOrder} \times \mathit{Type}\\
&\mathit{signOfVar\ formals\ vn} = \mathit{lookup\ vn\ formals}\\
&\\
&\mathit{transform} :: \mathit{CallerEO} \times \mathit{CalleeEO} \times \mathit{FrameArg} \times \mathit{State} \to \mathit{FrameArg} \times \mathit{State}\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{CBV}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBN}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{Lazy}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{CBN}, \mathit{ByNameArg}\ e, s) = (\mathit{StrictArg}\ v, s'),\\
&\ \ \mathit{where}\ (v, s') = \mathit{eval}\ (e, s)\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{Lazy}, \mathit{LazyArg}\ e\ b\ v, s) = \mathit{if}\ b\ \mathit{then}\ (\mathit{StrictArg}\ v, s)\ \mathit{else}\ (\mathit{val}, s'),\\
&\ \ \mathit{where}\ (\mathit{val}, s') = \mathit{eval}\ (e, s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBV}, \mathit{StrictArg}\ (\mathit{VI}\ v), s) = (\mathit{ByNameArg}\ (\mathit{EInt}\ v), s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBV}, \mathit{StrictArg}\ (\mathit{VC}\ c), s) = \mathit{error}\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{true}\ (\mathit{VI}\ v), s) = (\mathit{ByNameArg}\ (\mathit{EInt}\ v), s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{true}\ (\mathit{VC}\ c), s) = \mathit{error}\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{false}\ v, s) = (\mathit{ByNameArg}\ e, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{CBV}, \mathit{StrictArg}\ v, s) = (\mathit{LazyArg}\ e\ \mathit{true}\ v, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{CBN}, \mathit{ByNameArg}\ e, s) = (\mathit{LazyArg}\ e\ \mathit{false}\ \mathit{null}, s)\\
&\\
&\mathit{mutate} :: [\mathit{Formal}] \times [\mathit{Formal}] \times [\mathit{FrameArg}] \times [\mathit{Actual}] \times \mathit{Index} \times \mathit{State} \to [\mathit{FrameArg}] \times \mathit{State}\\
&\mathit{mutate}\ (\_,\ [], \_, [], \_,\ \mathit{acc},\ s) = (\mathit{acc},\ s)\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (\mathit{EVar}\ v\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}'\ :\ \mathit{acc}),\ s')\\
&\ \ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad \mathit{sign@}(\mathit{callerEO}, \mathit{callerType}) = \mathit{signOfVar}\ (\mathit{vn}, \mathit{callerfs})\\
&\qquad \mathit{callerIx} = \mathit{getIndex}\ (\mathit{vn}, \mathit{sign})\ \mathit{callerfs}\\
&\qquad \mathit{arg} = \mathit{args}[\mathit{callerIx}]\\
&\qquad (\mathit{arg}', s') = \mathit{mutate}\ (\mathit{calleeEO}, \mathit{callerEO}, \mathit{arg}, s)\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (\mathit{a@}(\mathit{EInt}\ n)\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}\ :\ \mathit{acc}),\ s)\\
&\ \ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad \mathit{if}\ \mathit{calleeEO} = \mathit{CBV}\ \mathit{then}\ \mathit{StrictArg}\ (\mathit{VI}\ n)\ \mathit{else\ if}\ \mathit{calleeEO} = \mathit{CBN}\ \mathit{then}\ \mathit{ByNameArg}\ a\\
&\qquad\qquad\qquad\qquad \mathit{else}\ \mathit{LazyArg}\ a\ \mathit{false}\ \mathit{null}\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (a\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{if}\ \mathit{calleeEO} = \mathit{CBV}\ \mathit{then}\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}'\ :\ \mathit{acc}),\ s')\\
&\ \ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad (v, s') = \mathit{eval}\ (a, s)\\
&\qquad \mathit{arg}' = \mathit{StrictArg}\ v
\end{aligned}
$$

Σχήμα 5.5: Λειτουργική σημασιολογία για τη λειτουργία *mutate*

Κεφάλαιο 6

Επίλογος

Αυτή η εργασία δείχνει πώς να επεκτείνουμε την κλασική βελτιστοποίηση κλήσης ουράς σε γλώσσες με οκνηρή αποτίμηση που υποστηρίζουν πολλαπλά είδη αποτίμησης (όπως π.χ. κλήση κατ' αξία και κλήση κατ' όνομα). Η εφαρμογή μας είναι ένας διερμηνέα που χρησιμοποιεί πληροφορίες από μια τοπική στατική ανάλυση, προκειμένου να ανιχνευθούν ευκαιρίες βελτιστοποίησης στις θέσεις κλήσεων ουράς.

Η τεχνική μας μπορεί επίσης να υλοποιηθεί σε ενα μεταγλωττιστή για οκνηρές γλώσσες που χρησιμοποιεί defunctionalization, όπως ο GIC [Four14a] ή ο GRIN [Boqu96, Podl19]. Αυτή η ενσωμάτωση χρειάζεται ένα μικρό γεννήτορα κώδικα για μετάλλαξη των πλαισίων για τις τις κλήσεις ουράς, οδηγούμενη από τις αποφάσεις του διερμηνέα μας. Ένα τέτοιο βήμα είναι το προφανές για την υλοποίηση μας.

Ενσωμάτωση με ένα πιο σύνθετο μεταγλωττιστή θα μπορούσε επίσης να βοηθήσει για να αξιολογήσουμε την τεχνική μας με πιο ρεαλιστικά προγράμματα ως αναφορά (benchmarks), όπως η σουίτα δοκιμαστικών προγραμμάτων *nofib* [Part93].

Κείμενο στα αγγλικά

Chapter 1

Introduction

1.1 Purpose of the thesis

The purpose of the thesis is to integrate tail-call optimisation in functional programming languages in the presence of multiple evaluation order choices (call-by-value, call-by-name, call-by-need). Our optimisation can be combined with the defunctionalization transformation [Reyn72], so that the resulting performance optimisation also supports higher-order functional languages.

1.2 Motivation

While programs in imperative programming lanaguages can run in constant space using “loops”, this does not happen in functional programming languages (which favour recursion over “loops”), because of the memory overhead due to recursion. On the one side, functional programming offers programmers and software engineers features that other programming styles do not [Hugh89], and on the other side, memory overhead due to recursion can be eliminated using tail-call optimisation.

Tail-call optimisation for strict functional languages is a well-studied topic, but it fails in the presence of laziness: the computation of a lazy argument can happen arbitrarily during the execution of the program and thus lazy arguments escape their context more easily than strict ones. On the one hand, laziness gives programmers great options: to use infinite data structures [Abel13], to define control flow (structures) as abstaractions instead of primitives, to increase performance by avoiding needless calculations, and to avoid error conditions when evaluating compound expressions. But this flexibility comes at a cost: too lazy programs can lead to poor performance and even memory leaks. On the other hand, strictness enables programmers to count performance more easily, but programs can fall into undesired behavior, such as divergence.

In this thesis, we get the best of both worlds: our programs need not be too lazy nor too strict, with evaluation order annotations supporting tail-call optimisation in lazy evaluated functional programming languages, thus eliminating memory overhead associated with recursion in the same way as in strict functional languages.

1.3 Overview of the thesis

In this thesis, we integrate tail-call optimisation in a functional language in the presence of multiple evaluation order choices (strict, lazy, call-by-name semantics). Our optimisation supports user-defined data types with pattern matching and thus can be combined with the defunctionalization transformation, so that higher-order functional languages are also supported. The source language, similar to Haskell, is transformed (via defunctionalization) to a low-level, minimal, first-order functional language with non-strict semantics, lazy evaluation and lazy structured data as well as strictness annotations. To find opportunities for optimisation, we perform a static analysis on the low-level functional language, to spot tail-call positions. The difficult part, compared with languages with strict semantics, is that lazy semantics makes program values escape their context and thus finding tail-call positions is not trivial. This optimisation was evaluated on an interpreter of the language that explicitly allocates

and measures frames, so that on tail-call positions found by the analysis, we can properly replace the unnecessary current frame's arguments with the arguments needed by the frame that represents the next function call. Our optimisation either improves program run-time, or does not change it. Also, in the case of strict programs, our optimisation is equivalent to classic tail-call optimisation. In conclusion, in a non-strict, lazy-evaluated functional language with lazy data constructors and strictness annotations, for all benchmarks we use, there is always memory optimisation, and for the majority of them there is a significant memory optimisation with performance boost.

1.4 Contributions

Our major contributions, presented in this thesis, are:

- We show how tail-call optimisation and tail recursion modulo cons can be applied to a functional programming language with mixed evaluation order. Our key contribution is how it can be applied in the presence of *lazy* evaluation, and thus turning the optimisation to an extension of tail-call optimisation for call-by-value languages.
- A prototype implementation of an interpreter with explicit frame allocation, as well as frame counting mechanism, for a functional language with call-by-value, call-by-name and call-by-need semantics, lazy data constructors and pattern matching.
- A static analysis algorithm to find *true* tail call positions and the implementation of the runtime system for these optimisations embedded in the interpreter.
- The evaluation of this optimisation on micro benchmarks shows that it either improves the runtime or does not change it. The evaluation seems to approach in many cases the number of frame allocations in call-by-value languages, which is the best result we can have.

An overview of the contents of each chapter follows:

- In *Chapter 2*, we provide the necessary background for the reader in order to follow the next chapters. We give a comprehensive explanation of classic tail-call optimisation (section 2.1), an overview of the evaluation order choices we study in this thesis (section 2.2), an overview of static analysis (section 2.3). Finally, in section 2.4 we give an overview of abstact machines and interpreters, focusing on lazy functional languages.
- In *Chapter 3*, we give an overview of the thesis; we provide the intuition for our key ideas presented in the rest of the thesis.
- In *Chapter 4*, we present the language we studied. Briefly, we describe how from a higher-order functional language and applying the appropriate transformations (sections 4.2-4.4) we can reach the first-order functional language we used for our analysis and evaluation of the optimisation.
- In *Chapter 5*, we describe the execution model of our language. We present operational properties as well as implementation details.
- In *Chapter 6*, we give the static analysis algorithm that we use in order to spot and thus properly annotate true tail-call positions. These annotated function calls are later handled by the interpreter.
- In *Chapter 7*, how we operationally treat inside the interpreter the tail call optimisable function calls.
- In *Chapter 8*, we give an evaluation of our optimisations on microbenchmarks.
- *Chapter 9* and *Chapter'10* are related work and conclude the thesis respectively.

Chapter 2

Background

In this chapter, we provide the reader the necessary background for this thesis. The optimisations presented in this thesis are explained. Anyone familiar with these ideas, can skip this section and continue to the next chapter, where we give the intuition behind the main idea of this thesis.

2.1 Tail-call optimisation

The main optimisation this thesis discusses is tail-call optimisation (TCO). This optimisation, first found in Scheme [Suss75, Stee76] and generally usually found in strict functional languages, allows functional programming languages to have constant space similar to 'for' loops from imperative programming languages [Clin98]. Without this, recursion would require *linear* memory space; i.e. one frame allocation for each function call.

Implementations of programming languages use a stack for function calls, where stack frames (or *activation records*) are pushed for every function invocation: this permits recursion and has good performance. Stack allocation for function calls has been an old technique, dating back at least to ALGOL [Dijk60, Naur81] and LISP [McCa60, Stoy79].

Tail calls are function calls that happen last inside a function and they can be implemented with a "go to" instruction. In the special case that the caller's frame will not be needed again, *tail-call optimization* can reuse it to construct the frame of the callee, saving space, and making some programs run in constant space. Tail calls can be further optimized by the instruction-scheduling stage of a compiler [Torc12, §12.4.3]. Tail calls have been used to transform recursive functions to iterative (tail-recursive) ones [McCa62, §9][Barr68]. Tail-call optimization is an old idea [Gill65, p. 7][Knut74, p. 21], which is an established part of modern compilers.

Some languages and implementations often offer features that are not easy to compose with tail-call optimization. Frames may not be resized in-place for languages that support coroutines [Wait84, p. 60]. Languages that support first-class continuations [Sper10], higher-order return values [Appe92, p. 103][Stee78], backtracking [Bobr73], or non-strict evaluation, and implementations in continuation-passing style [Appe92, p. 103], may allocate frames on the heap, because the lifetime of a frames does not follow a simple stack (push/pop) usage.

Tail-call optimization is also important for implementations of logic languages [Bigo99], functional programming languages running on the Java virtual machine [Mads18], imperative languages such as C [Baue03, Prob01], or even low-level compiler-targeted languages such as LLVM [Pand15].

What exactly is *tail call optimisation*? Tail calls are function calls in specific locations; specifically they are function calls performed as the final action of a function. Tail-call optimisation is actually passing the control from the caller to the callee; the runtime does not allocate a new frame for the callee function. Instead, it reuses the current frame from the caller function. This leads to a constant memory usage for the whole procedure.

In the example below, we define a 'factorial' function in OCaml:

```
let rec fact n =
  if n > 0
```

```
    then n * fact (n-1)
    else 1
```

Inside the function body, there is a function call to `fact`. This is not a tail-call, because this call is not the last call performed in the function body. In this example, the multiplication is the last operation performed, just before the function returns.

Let's assume that we have a call to the function:

```
let main = fact 3
```

The calling stack in this execution follows:

```
fact 3 = 3 * fact 2
fact 2 = 2 * fact 1
fact 1 = 1 * fact 0
fact 0 = 1
```

From the calling stack above, it seems that `fact 3` requires the result of `fact 2`, which requires the result of `fact 1`, which requires the result of `fact 0`, in order to produce the final result. A stack frame is allocated for each function call and it is finally free, when the result is returned to the caller. The last operation that happens in every call is the multiplication.

Let's take a look at a second example, again for the 'factorial' function:

```
let rec fact' n acc =
  if n > 0
    then fact' (n-1) (n * acc)
    else acc
```

These two examples have the same result. The *path* they follow in order to produce it though, is totally different. At this point, let's imagine a call to the function of the second example:

```
let main = fact' 3 1
```

The calling stack for `fact'` is the following:

```
fact' 3 1 = fact' 2 3
fact' 2 3 = fact' 1 6
fact' 1 6 = fact' 0 6
fact' 0 6 = 6
```

As we observe, the call to `fact'` is the last call the function does before it returns. Also, the return result 'acc' of the function is evaluated at every function call. This is the reason why `fact'` can run using only one stack frame, and thus we have *constant* stack space for the execution. Constant space is also what happens if we wrote the 'factorial' function with a 'for' loop, in imperative programming style.

Tail-call optimisation is great for both having functional programming, and thus recursion and more clear code, and also not the memory overhead often required by recursion.

2.2 Evaluation order choices

In this section, we will describe the evaluation order[1] choices we used in this thesis. In the setting of our first-order language, evaluation order is closely tied to how parameters passed and evaluated at a function call. Evaluation order controls *how* the caller and the callee function will interact, when the former calls the latter.

The evaluation order choices we studied are:

[1] Also known as *evaluation strategy*.

- call-by-value or *strict* arguments (e.g. OCaml, Scala, Scheme),
- call-by-name arguments (e.g. λ-calculus, Scala), and
- call-by-need or *lazy* arguments (e.g. Haskell, Scala).

Call-by-name and call-by-need semantics is similar to each other; they always produce the same result. Their key difference is that call-by-need computes a value when it needs it, memoizes the result after the computation and does not evaluate it again, while call-by-name re-evaluates the value, in case it's needed again for a computation. This makes call-by-name less practical. Call-by-need can also be considered as a *practical* implementation of call-by-name, first appeared as a real-world implementation in the *Miranda* programming language [Turn85], later in *Clean* [Plas99], and has become more popular with implementations of the Haskell programming language such as the Glasgow Haskell Compiler (GHC).

2.2.1 Call-by-value (CBV) or *strict* evaluation

Call by value is the most commonly used technique for parameter passing. It' s used in the most functional programming languages (such as Scheme and OCaml) and also in imperative and object oriented programming languages (C/C++, Java etc).

Call by value principles briefly are:

- Evaluate *fully* the actual parameters at the call inside the caller function body.
- *Bind* the *fully evaluated* values with callee's formal parameters *locally* inside the callee.

Let's become more clear using a simple first example:

```
length :: [Int] -> Int -> Int
length l@[]     acc = acc
length l@(x:xs) acc = length xs (acc + 1)

-- Make the call to length
main = length [1,2,3] 0
```

Let's suppose strictness in the example above, i.e.:

- 'l' is a strict list, exactly the same as a regular list in SML.
- 'acc' is also a strict parameter.

During the execution we have the following memory snapshots:

```
length [1,2,3] 0 = length [2,3] 1 -- At this point, 'acc' is
                                  -- evaluated to 0+1 = 1.
length [2,3]   1 = length [3]   2
length [3]     2 = length []    3
length []      3 = 3
```

This is the tail-recursive form of the function, known from the previous section, that counts the number of elements inside a list. As it seems above, the accumulator parameter 'acc' is evaluated in every function call, before the control of the call passes to the callee.

Now, let's take a look at a second example, where the accumulator is a list. The purpose for this example is to highlight the difference between strict data constructors a la ML and lazy data constructors. It will become obvious in later section, when we will talk about laziness (section 2.2.3).

```
makelist n acc =
  if n > 0
    then makelist (n-1) (n : acc)
    else acc
main = makelist 3 []
```

The calling stack of this program follows

```
makelist 3 []      = makelist 2 [3]
-- Again, 'acc' is evaluated before
-- the call on the right-hand side is triggered.
makelist 2 [3]     = makelist 1 [2,3]
makelist 1 [2,3]   = makelist 0 [1,2,3]
makelist 0 [1,2,3] = [1,2,3]
```

Until this point, we examined CBV semantics, the most common semantics that a programmer uses. From this point on, we will dive into the two remaining kinds of semantics we used and we will show to the reader the relation between them and their key differences, especially for datatypes. It is important the reader to understand the key differences from now, even though we will have a comprehensive explanation throughout the thesis.

2.2.2 Call-by-name (CBN) evaluation

Call by name is an evaluation strategy where the arguments to a function are not evaluated before the function is called. They are substituted directly into the function body (using capture-avoiding substitution) and then left to be evaluated whenever they appear in the function. If an argument is not used in the function body, the argument is never evaluated; if it is used several times, it is re-evaluated each time it appears.

Call-by-name evaluation is occasionally preferable to call-by-value evaluation. If a function's argument is not used in the function, call by name will save time by not evaluating the argument, whereas call by value will evaluate it regardless. If the argument is a non-terminating computation, the advantage is enormous. However, when the function argument is used, call by name is often slower, requiring a mechanism called a *thunk*, first appeared in ALGOL60 [Naur81]. More recent works works prove a relation between call-by-name and call-by-value [Wadl03], supporting our interest to investigate call-by-name as an evaluation order present in our core language.

A first example in order to showcase how a programmer can put CBN semantics to good use, is:

```
loop x = loop x

head []     = error "Empty list"
head (x:xs) = x

main = head [42, loop 42]
```

While in CBV, this program would cause a memory overflow, because it would try to evaluate [42, loop 42], while 'loop' is an ifinite loop and it would diverge, and thus the above would be an incorrect program, in CBN it will return 42, which is actually the first element of the list.

In a second example though, the drawback of CBN becomes obvious; it re-evaluates already evaluated values and thus turns the algorithm complexity from linear to more than quadratic. The example follows:

```
fact n acc =
  if n > 0
```

```
    then fact (n-1) (n*acc)
    else acc

main = fact 3
```

In this example, the successive order of calls would be:

```
-- 1
fact 3 1 =
  if n > 0                  -- Here becomes n = 3
    then fact (n-1) (n*acc) -- where n = 3  and acc = 1*3
-- 2
if n > 0                    -- Evaluates n-1=3-1=2 and n = 2
  then fact (n-1) (n*acc)   -- again n-1=2-1=1 and n*acc=1*1
-- This will follow until the end of execution,
-- and this leads to call-by-name with memoization (call-by-need).
```

2.2.3 Call-by-need or *lazy* evaluation

Call by need is a memoized variant of call by name where, if the function argument is evaluated, that value is stored for subsequent uses. If the argument is side-effect free, this produces the same results as call by name, saving the cost of recomputing the argument.

Haskell is a well known language that uses lazy evaluation. Because evaluation of expressions may happen arbitrarily far into a computation, Haskell only supports side-effects (such as mutation) via the use of monads. This eliminates any unexpected behavior from variables whose values change prior to their delayed evaluation.

Lazy evaluation is roughly guarded by two major priciples:

- *call-by-name* semantics, and
- *single-evaluation* property, which requires some form of memoization in the case of reused values.

At this point, let's take a closer look at this program, also presented in the previous section about CBN evaluation, but now in the presence of laziness:

2.2.4 Lazy data constructors

Assume the following program:

```
loop x = loop x -- function that diverges

head l =
  case l of
    []     -> error "Empty list"
    (x:xs) -> x

main = head [42, loop 42]
```

When we call `head` with argument `[42, loop 42]`, the following happens:

- The program allocates a frame with an unevaluated *thunk* [Blos88], which is the expression `[42, loop 42]`, for the function call to `head`. This means that the formal parameter `l` contains a pointer to the memory cell, which contains the aforementioned actual parameter.

- Then, `head` is evaluated. Imagine `case` as an operation that "opens" the data, i.e. it forces evaluation *one* more step, until it finds which constructor is used. The first expression to be evaluated is the list `l`, which is *needed* in order to pick a branch. More specifically the evaluation of `l` produces `x1:x2`, where `x1` contains a pointer to `42` and `x2` contains a pointer to `loop 42`. It also binds `x`, `xs` to `x1`, `x2` respectively.
- At this point, `case` knows which branch to pick from the previous step. As `l` is not an empty list, it picks the second branch.
- On the right-hand side of the second branch, there is variable `x`. Now, the value of `x` is needed, and thus it evaluates `x1`, that is evaluates the content of the pointer and finally it returns 42.
- An important notice is that because pattern matching variable `xs` does not exist on the right-hand side of the branch, it does not evaluate `xs` and thus the program does not diverge.

2.2.5 BangPatterns: Haskell with strictness

Haskell's de facto compiler, the Glasgow Haskell Compiler (GHC), contains a language extension, "bang patterns", available both in the interactive environment (`ghci`) with:

```
ghci -XBangPatterns
```

and also in the compiler as a flag, or as a language extension annotation inside a Haskell's source file as:

```
{-# LANGUAGE BangPatterns #-}
```

which allow the explicit use of strictness in Haskell. We should also mention the existence of `seq`, a function found in Haskell's Prelude:

```
seq :: a -> b -> b
```

which forces the evaluation of the first argument and returns the second argument as a result.

The example below showcases the use:

```
{-# LANGUAGE BangPatterns #-}

loop x = loop x -- function that diverges

-- !l: l is a strict parameter
head !l =
  case l of
    []     -> error "Empty list"
    (x:xs) -> x

main = head [42, loop 42]
```

The same annotation in our language's source files is used; it will be explained in detail in later section of Chapter 4. For now, we provide some explanation about how the above program will run, in order to highlight the difference with `head` with the lazy parameter, Haskell's builtin, which was given earlier.

The program above has the following behaviour when executed:

- Again, `head` is called. But now the frame for `head` has an evaluated *thunk*, because `!` moves the evaluation one more step. So, in contrast to the earlier example, `l` has a pointer to `x1:x2`, where `x1` contains a pointer to `42` and `x2` contains a pointer to `loop 42`. The *thunk* is also marked as *evaluated*.

- Then, `head` is evaluated. At this point, when `case` needs to evaluate `l`, which is marked as evaluated, `case` knows which branch to follow, which is the same as earlier, and does not do anything else.
- After the branch is picked, again the right-hand side of the branch is evaluated and `42` is returned as a result.
- An important notice here is that `!` does not force *deep* evaluation of the data structure. It just moves the value from weak-head normal form (WHNF) to head normal form. For data, this means that the outermost value is evaluated.

The idea about extending Haskell with strictness was used in StrictCore, an intermediate language which aims to improve GHC's Core language by having thunk/value distinction at the type level and multi-arity functions and multiple value returns, inspired by [Boli09].

2.2.6 Deepseq: ML-lists in Haskell

The above functionality of 'BangPatterns' shows that even with these strictness annotations, data constructors (lists in our example) are not the same as ML lists. But Haskell also has the 'deepseq' module, which cause deep evaluation of the data, and in this way we can have the option for *head normal data*, which are the same as the data that exist in languages like ML.

An explanation of 'deepseq' functionality follows. BangPatterns are used in data constructor definitions.

```
data List a = Empty | Cons !a !(List a)
```

In the above data definition, `Cons` contains:

- A strict head, annotated with a '!'.
- A strict tail, annotated also with a '!'.

In case that this list was used for `head`, we finally have ML lists and the program will diverge as it happens in a strict functional language like OCaml. 'Deepseq' automatically can turn a data structure defined in Haskell into a ML-like data structure, by automatic instance derivation.

In case the programmer needs more customization, they can use BangPatterns in every possible combination. The definitions below are also used in this thesis' language.

```
-- 1. Haskell builtin lists.
data List a = Empty | Cons a (List a)
-- 2. Lists with strict head.
data List a = Empty | Cons !a (List a)
-- 3. Lists with strict tail.
data List a = Empty | Cons a !(List a)
-- 4. Deeply evaluated lists.
data List a = Empty | Cons !a !(List a)
```

2.3 Static analysis

Static program analysis attempts to predict all possible runtime behaviour of a program [Niel10]. This analysis happens without running the program, using just the program text or some other equivalent intermediate representation. This permits the analysis to run before the program runs and can thus be used to guide decisions such as compile-time optimizations.

In practice, all realistic programming language implementations incorporate a set of static analysis that help apply optimizations, check type annotations, detect errors, or otherwise catch program behavior before it happens.

Of particular interest to this thesis is strictness analysis [Peyt91, Hold10], an analysis especially useful for lazy languages, where laziness can be overriden by a "strictness" annotation (such as the functionality described in section 2.2.5). This analysis runs on a lazy program and determines which expressions can be evaluated efficiently using call-by-value, instead of lazy evaluation, without changing the meaning of the program. More details about this analysis will be given in section 4.3.

In Chapter 6, we will also give our own static analysis that will be able to detect the program points where tail-call optimization can be applied.

2.4 Abstract machines and Interpreters

In this section, we provide some background about abstract machines used for functional programming languages implementations. We do not dive deep into every abstract machine; our purpose is to provide keywords and references to the reader and, where appropriate, to compare with our approach. For more details on how our model differs from other non-strict models, consult the work of Fourtounis *et al.* [Four13, Four14a].

2.4.1 Stack Environment Control Dump machine (SECD)

The SECD machine is a highly influential virtual machine and abstract machine intended as a target for functional programming language compilers, introduced in 1964 [Land64], and explained in detail by Danvy [Danv05]. The letters stand for Stack, Environment, Control, Dump, the internal registers of the machine. The registers Stack, Control, and Dump point to (some realisations of) stacks, and Environment points to (some realisation of) an associative array.

2.4.2 Three-instruction machine (TIM)

The Three Instruction Machine (TIM) [Argo89] represents a closure by a pair of a code pointer and a pointer to a heap-allocated frame [Fair87]. The frame, which is a vector of code-pointer/frame-pointer pairs, gives the values of the free variables of the closure, and may be shared between many closures. These pairs of code pointer and frame pointer need to be handled very carefully in a lazy system, because they cannot be duplicated without the risk of duplicating work, and thus violating the single evaluation property.

In our approach, we also use heap allocated frames, but we have a uniform representation of our runtime structs. Our interpreter allocates frames both for function and constructor applications, and updates the frames appropriately when the evaluation of a lazy argument is forced.

2.4.3 Abstract machines for Haskell

The G-machine was an abstract machine originally designed for Lazy ML [Augu84], a version of ML with lazy evaluation. It is based on compiling supercombinators, functions with no free variables (or top level functions); the lambda lifter transforms lambda calculus into supercombinators. Then, the supercombinators are compiled into G-code, the abstract machine code, whose execution model is graph reduction [Turn79]. The G-machine handled tail calls in a similar way to ours (by replacing the arguments of the caller's frame with the arguments of the callee function), but it suffered from space leaks [Jone90]. Our technique's uniform representation of runtime structures and full function and constructors applications (as it is first order), prevents these space leaks. Although the G-machine is not built for Haskell, it is the predecessor of the Spineless Tagless G-machine, Haskell's abstract machine until today.

The Spineless Tagless G-machine (or STG) [Jone92], in contrast to all other the abstract machines mentioned above, for which the abstract machine code for a program consists of a sequence of abstract machine instructions, is itself a small functional language. The machine is built for Haskell, a non-strict higher-order functional programming language with lazy evaluation. Each instruction is given a precise operational semantics using a state transition system. This machine is dealing updates following the self-updating model; instead of leaving the responsibility for frame updating to the code that updates the thunk, it passes the responsibility to the thunk itself. This can be useful if the machine is used for a language that supports distributed or parallel computations, where the thunk that needs to updated is not accessible from the code that updates it, at least in an affordable way. STG also has a uniform representation for heap allocated objects (a head normal form or a thunk); by a code pointer together with zero or more fields which give the values of the free variables of the code. It also supports "let" bindings natively, while the function's arguments are either variables or constants. This means that all actual parameters are transformed to "let" bindings, where "let" is a heap builder operationally.

2.4.4 Push/enter vs. eval/apply

In the functional programming world there are two basic approaches for evaluation of function calls: push/enter and eval/apply.

- The push/enter model, briefly proceeds as following: the function (callee), which statically knows its own arity, examines the stack to figure out how many arguments it has been passed, and where they are. This is also our own approach, where at a function we look up the signature of the function, evaluate its arguments and jump to the function body.
- The eval/apply model briefly proceeds as following: the caller, which statically knows what the arguments are, examines the function closure, extracts its arity, and makes an exact call to the function.

Marlow & Jones find that eval/apply is better than push/enter for higher-order implementations of lazy evaluation [Marl06]. However, we implement an interpreter for a first-order functional language, where no partial applications take place and thus the first approach (eval/apply) is not directly applicable. We follow a push/enter approach, since we know in advance how many are the arguments and we are able to construct the appropriate runtime data structure from the callee's point of view.

Chapter 3

Overview

In this chapter, we present to the reader some examples providing the intuition of the main idea described in the following chapters. The description is informal and the reader has to know only the details of laziness described in the background in order to follow. We do not refer to details of the analysis and the execution model that may confuse the reader, while leaving that part for later explanation, in the appropriate sections.

More specifically, in section 3.1 we provide examples for integers (values always in WHNF) and for almost all the cases of handling lists ('lazy' data constructors): consuming ('sum'), constructing ('makelist'), consuming and constructing ('map'). In section 3.2, an example for tail recursion modulo cons is presented. It is an extension of classic tail-call optimisation, first appeared in an implementation of Prolog.

3.1 Classic tail-call optimisation

3.1.1 Example 1: Consuming lists

The first example with constructed data is a function that *consumes* the input data in order to produce a result. The function traverses the input once and then computes the result. As we already mentioned, all arguments here are *lazy*.

```
sum :: [Int] -> Int -> Int
sum []     acc = acc
sum (x:xs) acc = sum xs (x + acc)
```

In *strict* languages like SML/NJ or Scheme, this is subject to tail-call optimisation. As we know from section 2.2.1, `acc` is *fully* evaluated in every execution step, while in the presence of laziness `acc` is evaluated *once*, when the input list is empty and the function *needs* it as a result.

The intuition here is that lazy argument `acc` has to be strict, as in the example below, because it is a variable that has always a value (in the case the value of the variable `x`) added to it, and finally it is returned as a result. This is known in the world from strictness analysis, as we will describe later in section 4.3.

```
sum :: [Int] -> Int -> Int
sum l !acc =
  case l of
    []     -> acc
    x : xs -> sum xs (x + acc)
```

At the latter example, `acc` (an integer or WHNF) is strict. This means that `acc` is fully evaluated in each execution step, and thus it can be tail-call optimised. This, of course, is not possible if `acc` was, for example, a list, as we will illustrate with the following examples.

3.1.2 Example 2: Constructing lists

In this example, we have a function that constructs a list with successive integers up to the input number ‘n’ (‘0’ excluded, ‘n’ included). The function contains a tail-recursive call in its body to itself (`makelist (n-1) (n : acc)`).

```
makelist :: Int -> [Int] -> [Int]
makelist !n !acc =
  if n > 0
    then makelist (n-1) (n : acc)
    else acc
```

Although the example above seems a perfect candidate for tail-call optimisation, it is not, at least in the traditional point of view of tail-call optimisation. Here, lists are lazy data constructors and are not fully evaluated in every execution step. The bang (`!`) does not force deep evaluation of the data, but it forces the evaluation one more step. That means that the strict `acc` constructs its *backbone* (a memory thunk) in memory, along with a *recipe* (`n-1` and `n:acc` cells) for further evaluation.

With a more case-aware analysis and additional work in the evaluator, we were actually able to tail-call optimise the example above. The details will follow in sections 6.1 and 7.1 for the analysis and the evaluator respectively.

3.1.3 Example 3: Constructing and consuming lists

An example of constructing and consuming, or better *processing*, lists is `map` or `foldl`/`foldr`. In the example below, we present a defunctionalized version of `map`, similar to Prelude’s `map`, instantiated for the integer domain.

```
data Func = Add Int
apply f x =
 case f of
   Add a0 -> add a0 x

map :: Func -> [a] -> [b] -> [b]
map f l acc =
 case l of
   []     -> acc
   x : xs -> map f xs (apply f x : acc)

inc a = a + 1

result = map (Add 1) [1, 2, 3] []
```

In the example presented above, the tail-recursive call to `map` can be subject to tail-call optimisation, following the same principles as in the examples like `sum` and `makelist`. The classic non tail-recursive version of `map` (`apply f x : map f xs`) is actually subject to tail recursion modulo cons, which is presented in the following section.

3.2 Tail recursion modulo cons

Tail recursion modulo cons is a generalization of classic tail-call optimisation. First found in an implementation of Prolog, it can also be applied in functional programming languages, especially those with lazy evaluation [Wadl84].

Tail recursion modulo cons can also be found in bibliography, as *guarded* recursion, a recursive call guarded by a constructor. This can become more clear, if we consider lazy data constructors, described in section 2.2.3, which use a mechanism called *thunk* for their evaluation.

Let's illustrate this idea with a simple example presented in the next section.

3.2.1 Motivating example

In Example 3 (section 3.1.3), `makelist` was in the tail-recursive form. In this section, we have a different version of the function. Programmers who constantly use strict functional programming languages like OCaml will argue that the program below is bad.

In Haskell, this is not the case. Actually, in Haskell's Prelude all functions are written in the form below, where these programs are subject to many optimisations triggered by Haskell's *rewriting machine*, and performed in the runtime (especially this program is subject to *fusion* [Cout07]).

At this point, we showcase the use of tail recursion *modulo cons*, instead of other optimisations used in GHC.

```
makelist x =
  if x > 0
    then x : makelist (x-1)
    else []
```

In the example above, the `then` clause contains a constructor application:

@ (@ (:) x) (@ makelist (x-1))

This recursive call is guarded by the constructor `(:)`. From section 2.2.4, where we described lazy data constructors, we know that this constructor application will be suspended when the program is running. This means that it will create a memory thunk with two cells, which will contain `x` and `makelist (x-1)`. The function call to makelist is actually a call that can be transformed to a tail-recursive call to `makelist`, if the `(:)` thunk is *updated* with the value of the variable `x`.

Chapter 4

The language

In this chapter, we will describe the language that we studied in this thesis. The semantics of the language will become clear to the reader, as well as the special treatment of the evaluation order in the final core language, which was used for the static analysis and for the optimisations. Briefly, a comprehensive description of the path from a higher-order functional language to a first-order functional language with the evaluation-order annotations will follow.

4.1 Overview of the language

The language we studied is a functional language with multiple evaluation orders (call-by-need or *lazy*, call-by-name and call-by-value or *strict*). We consider laziness as something that happens naturally in the language, while the other two evaluation orders are used as language extensions, using proper annotations. Our language is based on the language used by Fourtounis *et al.* [Four13, Four14a].

The annotations can appear syntactically (i.e. the programmer can annotate the formal parameters in the function definition) or after applying the transformations, described in sections 3.2 to 3.4, to the source language. The latter is only the case for strict arguments revealed by strictness analysis.

4.2 Defunctionalization transformation

Defunctionalization is a compile time tranformation technique which eliminates higher order functions, replacing them by a single first-order *apply* function, introduced by John Reynolds [Reyn72]. Reynolds' observation was that a given program contains only finitely many function abstractions, so that each can be assigned (and replaced by) a unique identifier. Every function application within the program is then replaced by a call to the apply function with the function identifier as the first argument. The apply function's only job is to dispatch on this first argument, and then perform the instructions denoted by the function identifier on the remaining arguments.

One complication to this basic idea is that function abstractions may reference free variables. In such situations, defunctionalization must be preceded by lambda lifting [John85], so that any free variables of a function abstraction are passed as extra arguments to apply. In addition, if closures are supported as first-class values, it becomes necessary to represent these captured bindings by creating data structures.

The defunctionalization transformation was used in MLton, an optimising compiler for ML. The first-order core language created oportunities for whole-program analysis and led to great performance [Cejt00]. Defunctionalization has also been used in implementations of lazy programming languages, such as GRIN [Boqu96, Podl19] and GIC [Four14a, Four14b].

As an example of defunctionalization, Prelude's `map` follows. The transformation leads to the non tail-recursive version of `map`.

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

```
inc :: Int -> Int
inc a = a + 1

result = map inc [1,2,3]
```

After performing defunctionalization `map` is (instantiated for integers):

```
data Func = Inc

apply :: Func -> b -> b
apply f x =
  case f of
    Inc -> inc x

map :: Func -> [a] -> [b]
map f l =
case l of
  []     -> acc
  x : xs -> apply f x : map f xs

inc :: Int -> Int
inc a = a + 1

result :: [Int]
result = map Inc [1, 2, 3] []
```

The differences between the two versions are:

- The higher-order function, which is the first argument of `map`, (`f`, having type `(a -> b)`), is transformed to a *unique* identifier, which is the data constructor `Inc`.
- For the application inside the body of `result` function, we define the first-order `apply` function, which pattern-matches on the unique identifiers that are applied to `map`.

Further details and formalization of defunctionalization transformation are not given here. The reader can refer to paper by Reynolds [Reyn72] for more information about this transformation.

4.3 Strictness analysis

Lazy evaluation only evaluates terms to values when needed; this provides the opportunity for infinite data structures and programs that do not diverge, as those in strict programming languages. But everything comes at a cost: evaluation can happen arbitrarily and thus lazy arguments can escape their context.

But Mycroft noticed that some lazy terms are actually strict under the right circumstances [Mycr80]. For instance, the `case` construct forces the evaluation of an expression, called the *scrutinee* [Jone95], causing the scrutinee to actually become strict, similar to strictness presented in section 2.2.5 about BangPatterns.

And it's more than that. Let's look again the sum program of section 3.1.1:

```
sum :: [Int] -> Int -> Int
sum []     acc = acc
sum (x:xs) acc = sum xs (x + acc)
```

Here, lazy argument `acc`, which is a thunk in each function call is evaluated once, when the program's execution flow reaches the base case. This *arbitrary* evaluation of lazy arguments seems not to be so arbitrary after all. We do know when `acc` is going to be evaluated.

The question is: what makes `acc` so predictable? From the example above, we have the following information for `acc`:

- `acc` is an integer, always added to another integer.
- `acc` is the result of the function.

From these observations we can deduct that `acc` is going to be evaluated (second observation) and all intermediate values are needed for its evaluation, as `acc` is an integer (first observation).

Although in this thesis we used integers and lists, strictness analysis can be applied except for integer domain to any other domain. In practice, strictness analysis is a core analysis of Haskell implementations [Peyt91, Burn96, Hold10].

In our model, strictness analysis may be performed in the defunctionalized higher-order language in order to properly annotate strict arguments. Strictness annotations are also available in the language syntax, and thus the programmer can manually give such annotations. The origin of the strictness annotations (analysic vs. manual input) is thus not imporant for the aims of this thesis; we only assume that such annotations are available.

4.4 Other transformations

After defunctionalization transformation and strictness analysis are performed, we also perform some more transformations to the source language in order the intermediate language to reach the final form, which is the input of the analysis, presented in later section.

- **Alpha-renaming:** This transformation (also known as *alpha-conversion* [Bare92]) renames all variables bound by pattern matching. Every 'case' expression opens a new scope and variable names in this scope are bound to the closest 'case' pattern. This transformation also handles name shadowing.
- **If-to-case transform:** When the language supports pattern matching on data (constructors), 'if' expression is not needed. It can be transformed to pattern matching on the nullary boolean constructor (`True` vs. `False`). Haskell does not have an 'if' expression as builtin; it only allows syntactic use (as syntactic sugar for boolean patterns).
- **Constructor projections:** Every variable bound by a 'case' pattern has to be transformed to a special syntactic node in the core language, which contains the case unique identifier and the position inside the expression list in the constructor pattern [Four13]. The 'case' id is unique inside a function definition (locally).

4.5 Syntax

In this section, the syntax of our core language is given. This is the output of the strictness analysis and defunctionalization transformation as well as the other transformations of the previous section. We run the analysis algorithm to spot tail-call positions with input the language from this section. The interpreter is also built for that language; the evaluator for the optimisations are also implemented inside the interpreter for this language.

Figure 4.1 shows the abstract syntax of the first order intermediate language. We need to highlight the following points, before we dive deeper into the language:

$\langle p \rangle ::= \langle fdef \rangle^+$	Program
$\langle fdef \rangle ::= f\ v_1 \ldots v_n = \langle expression \rangle$	Function Definition
$\langle expr \rangle ::= v$	Variable
$\mid \langle integer \rangle$	Integer
$\mid f\ e_1 \ldots e_n$	Function application
$\mid c\ e_1 \ldots e_n$	Constructor application
$\mid$ `case` e_0 `of` $patt_1 \rightarrow e_1 \mid \ldots \mid patt_n \rightarrow e_n$	Case expression
$\langle patt \rangle ::= c\ v_1 \ldots v_n$	Constructor pattern
$\mid \langle integer \rangle$	Integer pattern
$\langle integer \rangle ::= 1, 2, \ldots$	Integer domain

Figure 4.1: My grammar

- The source language contains an 'if' expression syntactically. In the syntax figure there is no such construct. This is because 'case' is much more powerful than 'if' and an 'if' can be transformed to a 'case'. Actually, we have (boolean are constructed data as well):

```
if cond then e1 else e2 => case cond of { True -> e1; False -> e2 }
```

- There are no partial applications, neither in function nor in constructor applications. This means that the arguments in a function call are the same in number as in the function definition.
- There is no 'let' expression in the core language. Instead we assume that our language fully depends upon a lambda lifter; a Johnson-style *full laziness* lambda lifter [John85] will get the work done.
- Case patterns are simple and do not allow wildcard patterns. Turning a complex case pattern into a simple case pattern is a well studied topic [Augu85, Wadl87].
- The scrutinee of a case construct is not a case expression. We assume that case-of-case transform [Peyt98] is performed.

The syntax in the figure is a syntax of a first order functional language. Our syntax has two main differences from the one shown in the figure:

- The concrete syntax of the core language has evaluation order annotations in order to distinguish between the semantics during the parameter passing. We have already mentioned the strictness annotations added by the strictness analysis. More specifically, the programmer can concretely annotate the formals in the function definition with:
  - '!' for a strict formal parameter,
  - '#' for a call-by-name formal parameter, while
  - lazy arguments are not annotated, because we consider laziness as something that naturally happens in the language.

  As far as the abstract syntax is concerned, the formal parameters of a function definition are:

```
type Formal = (VN, (EvalOrder, Type))
type VN = String
type EvalOrder = CBV | CBN | Lazy
```

```
data Type = TInt | TCons Type -- an value of type integer (TInt) or
                              -- a list of values with type Type
```

where the type constructor of a formal parameter has the information about the parameter's name and the static info of its evaluation order and its type. We assume the base types as integers and later in the analysis part, we will explain why this is enough.

- There is one more node in the syntax tree: *constructor projections*. The transformation for this particular syntax node has been explained earlier (section 4.4).

  ```
  data Expr = ... | CProj CaseID CPos
  ```

  In a constructor projection, the first argument 'CaseID' shows the position of case where the argument belongs and the second argument the position of the variable in the left hand side of a constructor pattern in a 'case' branch.

  The purpose of constructor projections is to bind the variables on the right-hand side of the branch with the variables on the left hand side of the branch. In this way, these variables are distinguished from the top level variables, i.e. the formal parameters in the function definition and they offer the possibility to know the position in the frame, a reason that will become more clear in the next chapter, where we are going to give the details of the execution model.

Chapter 5

Execution model

In this chapter, we will present technical details the operational semantics for the language we studied in this thesis as well as the technical details about the implementation of the interpreter, on which we evaluate our technique for the tail-call optimisation and tail recursion modulo cons.

Section 5.1 gives a high-level description of the machine; then in section 5.3 the runtime data structures and the design decisions will become clear to the reader.

5.1 Overview of the execution model

In this section, we present a high-level description for the execution model; this will be helpful for the reader to follow the technical details from the next sections. Also, it will be easier to compare to other existing abstract machines and interpreters that exist for a lazy functional language.

Our model, based on GIC [Four14a, Theo13], includes the following high-level properties:

- We implement a first-order lazy abstract machine. It is first-order as we do not support function arguments cannot be functions and partial evaluation. Laziness is the default semantics for our model.
- We have a frame-based execution model for our language, but the frames are *heap-allocated frames*, instead of stack-allocated. The reason for this is that lazy evaluation breaks the sequential order of execution, because of frame updates, and thus we are not able to allocate and deallocate frames with push/pop operations.
- Our interpreter supports explicit frame allocation, needed for frame mutation and evaluation of our technique, as well as a frame counting mechanism of frames.
- Our language supports multiple evaluation orders and thus our interpreter also handles them (call-by-value, call-by-name, call-by-need).
- Our data are lazy data constructors, but we can also support ML-lists (data with deep evaluation presented in section 2.2.6).
- At function calls we follow the push/enter function call policy. Before jumping into the function body, we allocate the frame and evaluate all the actual parameters, with the respect to the semantics.
- Our model supports pattern matching (constructors and base type values) similar to Haskell. This is the most complicated feature in a lazy functional language, if we consider that initial abstract machine did not provide support for that (section 2.4.3).

5.2 Runtime system

This section contains a detailed explanation of the prototype implementation of the interpreter we used for the language. First, we are going to present the data structures used in the runtime; then, we present the operational features of our interpreter.

5.3 Runtime data structures

Here, there are the definitions of the runtime structures, as they are used in the implementation. The implementation is in Haskell, but the code will be simple, so that every reader with a basic background of a functional language can understand.

In the next sections that will describe the runtime structures, we will use the following convention. At first, a definition will appear; probably accompanied with one or more definitions. If it contains more complex data in its body, it will be described later in the section or there will be a pointer to another section.

5.3.1 Memory: The global frame container

In this section, we provide the definition of *memory*: a *mutable* memory space, where frames (explained in section 5.3.4) are stored.

We have the following definition for *memory*:

```
type FrameId = Int
data Mem = Mem {
  memFrames :: Map FrameId Frame, lastFrameId :: FrameId
}
```

The data definition has the following fields:

- A *map* data structure in which a unique *frame* identifier actually corresponds to a frame in the memory (*memFrames*).
- The frame identifier for the last frame that was allocated (*lastFrameId*).

The reader can think of memory as the memory in a computer: every memory cell has an address (*FrameId*) and the address has a content (*frame*). A similar representation to Launchbury's mutable heap for lazy evaluation [Laun93]. Here, not all frames have the same capacity; the details will be given in later section for a frame.

This aforementioned data structure is accompanied with three operations in order to handle it:

- Add a frame to the memory with *push* operation (figure 5.1).
- Get a frame from the memory given its unique identifier (*getFrame*) (figure 5.2).
- Update the content of a memory's frame, given its unique identifier (*updFrame*) (figure 5.3).

$$
\begin{aligned}
&push :: Mem \times Frame \rightarrow Mem \\
&push\ (mem, frame) = mem' \\
&mem' = Mem\ \{\ memFrames = frames', lastFrameId = lastId\ \} \\
&frames' = frame : (memFrames\ mem) \\
&lastId = lastFrameId\ mem + 1
\end{aligned}
$$

Figure 5.1: Push frame operation

$$
\begin{aligned}
&getFrame :: Mem \times FrameId \rightarrow Frame \\
&getFrame\ (mem, id) = frame \\
&frame = lookup\ id\ (memFrames\ mem)
\end{aligned}
$$

Figure 5.2: Get frame operation

Later on, we will use the terms *push*, *getFrame*, and *updFrame* in order to refer to operations performed in memory.

$$updFrame :: Mem \times FrameId \times Frame \rightarrow Mem$$
$$upFrame\ (mem, frameId, frame) = mem\ \{memFrames = insert\ frameId\ frame\ (memFrames\ mem)\ \}$$

Figure 5.3: Update frame operation

5.3.2 Suspended execution of constructed data

The data definition of suspensions (for short or suspended execution of constructed data) is:

```
data Susp = Susp (CN, [Expr]) FrameId
```

A *Susp* data construction contains:

- The constructor's name *CN* along with the arguments applied to it in an expression list, as the constructors in our language are *lazy*.
- A pointer to the frame' s identifier (*FrameId*) , which is the environment for this suspension. This is the environment, until this suspension has been created. When the execution of the program forces the constructor' s execution again, it will use this environment for its evaluation.

5.3.3 Values

A value can either an integer value or a suspension.

```
data Value =
  VI Integer
| VC Susp
```

From the above definition, we have that a value can be either an integer (or generally a base type value) or a suspension of constructed data (see section 5.3.2). While the former is obvious, especially for most programmers who use strict languages, the latter is something that happens in programming languages using lazy evaluation. Computations may not be deeply evaluated and thus they can create thunks or leave unevaluated thunks that can be evaluated later.

5.3.4 Frames

The basic *unit* of the execution machine is a *frame*. A frame contains all necessary information for a function call. It is allocated every time a new function call takes place; then, it can be mutated in case it contains a lazy parameter.

Here is the definition of a *frame*:

```
type FN      = String
type CaseID  = Int
type FrameId = Int

data Frame   = Frame {
  fName  :: FN,                 -- Function Name
  fArgs  :: [FrameArg],         -- Bindings of formals with actuals
  fSusps :: [(CaseID, Susp)],   -- Data deconstruction forced by 'case'
  fPrev  :: FrameId             -- pointer to previous stack frame
}
```

A frame has the following information during its lifetime:

- The function's name (*FN*). This information is important to lookup the function definition in *functions map*, a structure that will be explained later, and the execution to proceed.

- The actual parameters that are binded with the formal parameters of the function definition of *'FN'*.

- Information about suspended execution, thunk creation and evaluation forced by 'case'. Each *'CaseID'*, unique in the body of the function contains pointers to its own thunks.

- A pointer to the previous frame's unique identifier (*fPrev*). This is the *environment* in the interpreter's implementation, implied and not explicitely existing in the *state* of the interpreter.

An argument that lives in 'fArg :: [FrameArg]' has the following definition:

```
data FrameArg =
  StrictArg { val :: Value }
| ByNameArg { expr :: Expr }
| LazyArg   { expr :: Expr, isEvaluated :: Bool, cachedVal :: Maybe Value }
```

An argument can be either:

- Strict and thus containing a *value* (more about values in section 5.3.3).

- Call-by-name and thus containing only an expression, an unevaluated thunk with no option for memoization.

- Lazy and thus containing an expression, in the same way as call-by-name arguments, but also a *flag* whether it is evaluated or not (preserving in this way the *single evaluation property* if it is already evaluated) and also space for the *cached value*.

5.4 Interpreter

The interpreter is a function that takes a program's *expression*, a *static* memory space which contains information about top-level function definitions and a *state*, reaches a *final state* and *returns a value*, where an expression is one of the expressions of the syntax tree presented in section 4.5. The definition of the *state* follows later in the section.

From now on, the function *eval* corresponds to the interpreter function.The declaration of the *eval* function is shown in figure 5.4:

$$eval :: Expr \times FunctionsMap \times State \rightarrow State \times Value$$

Figure 5.4: Declaration of *eval*

The *State* of the *eval* is shown in figure 5.5: where the definitions for *Mem*, *FrameId* are given earlier in this chapter. The field *NRFrames* gives the number of frames allocated so far. Remember at this point that the interpreter includes *explicit frame allocation* and this number is the *goal number* for the tail-call optimisation, that will be presented in a later section.

The structure *FunctionsMap* is a stucture that is created at compile time and it remains alive in the runtime as well. It does not change during the execution of the program and is a *map* of key-value pair, where:

- Keys are the names of top-level functions.

- Values are a pair of formal parameters with their static information (evaluation order, type) of the function and the body of the function.

$$State :: (Mem, FrameId, NRFrames)$$

Figure 5.5: State of the interpreter

```
type FN = String
data FunctionsMap = Map FN ([Formal], Expr)
```

Figure 5.6: FunctionsMap

The data definition of the aforementioned structure is shown in figure 5.6: The result of the interpreter is the result of the execution of the body of the top-level function *main*. This function is a special case of a function and is assumed not to have any arguments. So, the initial expression *expr0* is:

```
(_, expr0) = Map.lookup "main" functionsMap
```

The *initial state* for the interpreter's execution to start with, is:

$$State0 = (mem0, frameId0, nr\_frames0)$$

where:

- The initial state of memory, called *mem0*, is:

  ```
  -- cTOPFRAMED: last frame id available when execution starts
  mem0   = push (Mem Map.empty 0) frame0
  frame0 = Frame "main" [] [] cTOPFRAMEID
  ```

- The initial id of the last frame id that lives in the memory is given by:

  ```
  frameId0 = lastFrameId mem0
  ```

- The initial number of frames allocated, before eval starts execution are:

  ```
  nr_frames = 1
  ```

  as we have *one* frame allocation for "main" function.

At this point, we have everything set up. We declared the interpreter's function *eval* and we have an initial state to start our eval with. Now, let's dive into the execution of each expression of the syntax tree of our language presented in section 4.5. Minor implementation details are omitted for clarity; at first we assume laziness everywhere, while later there will be an overview of how the interpreter runs in the presence of ML-lists.

We finally begin our pattern matching on the expressions. The *FunctionsMap* structure is also omitted as eval's argument; we assume that we lookup this for formals and we want to jump to a function body, e.g. in the function call expression.

In every execution step our *eval* is looking up the current state. As we have already mentioned the state is a record of:

```
State = (Mem, FrameId, NRFrames)
-- FrameId: id of the current frame
```

and thus by looking up the current's state frame, we can obtain the environment, as following:

```
thisFrame = getFrame mem frameId
Frame caller funArgs susps prevFrameId = thisFrame
```

$$(v', s') = (val, s),$$
$$\textit{where}$$
$$v = \textit{StrictArg val}$$

Figure 5.7: Variable lookup for call-by-value

In this way, we have information about:

- *caller*: the name of the caller function we are currently in,
- *funArgs*: the current function' s actual arguments that are built for the frame (see later how we build these arguments when a function call is invoked),
- *susps*: suspended executions for constructors forced by pattern matching,
- *prevFrameId*: The id of the previous frame, just like having a pointer to the previous frame.

After the necessary information from the current state is obtained, the intepreter handles each one of the expressions in the following way, presented in the following sections.

5.4.1 Variable lookup

We remind at this point that a variable bound at case pattern matching is not evaluated here because of the constructor projection transformation we described earlier in section 4.4, and thus in this section we are concerned about evaluation of top-level variables.

A top-level variable exists in the formal parameters of a function definition. In *frame's terms*, we are looking for the proper *FrameArg* in the *funArgs* field, mentioned earlier when we explained the interpreter's *current state*.

The procedure is the following (this is equivalent to lookup the variable inside the environment):

- First, we find the position of the variable in the current frame.
  - Lookup the *caller* function in the *FunctionsMap* structure to find the function's *signature*.
  - Once the signature is found, then we find the position in formal parameters of the function definition, i.e. the *index (i)* of the argument.
- Given the index that we found earlier, we find the proper frame argument in the current frame.

Once we have found the proper frame argument, we have *three* cases, depending whether the variable is in cbv, cbn or lazy evaluation order (we use the abbrevation e.o. which stands for evaluation order).

Later, we give the v' and s' for: $eval(v, s) = (v', s')$

- cbv e.o.: Variable lookup operation for call-by-value variables is shown in figure 5.7.
- cbn e.o.: Variable lookup operation for call-by-value variables is shown in figure 5.8.

  We note that whatever is the new interpreter's state *s''*, we do not memoize it, as call-by-name semantics does not include any change of the memory's state.
- lazy e.o.: In this case (figure 5.9) we have *two* subcases. The subcases are whether *b* is *true* or *false*, i.e. the lazy argument is evaluated and cached or not evaluated. In the latter case, the interpreter needs to evaluate the variable and update the frame, as shown below, in the appropriate case:

$$(v', s') = (val, s),\ where$$
$$v = ByNameArg\ expr,$$
$$(val, s'') = eval\ expr\ (mem, prevFrameId, nr\_frames)$$

Figure 5.8: Variable lookup for call-by-name

$$v = LazyArg\ e\ b\ val$$

– b is *true*:

$$(v', s') = (val, s)$$

– b is *false*:

$$(v', s') = (val', (updFrame\ mem'\ frameId\ frame', frameId, nr\_frames')), where$$
$$(val', (mem', frameId', nr\_frames')) = eval\ e\ (mem, prevFrameId, nr\_frames)$$
$$frame'[funArgs] = (funArgs[i] = LazyArg\ e\ true\ val')$$

Figure 5.9: Variable lookup for call-by-need

We note that, at this point, where memory change happens, and probably evaluation of the expression provokes the creation of new frames, we update the memory state as well as the frame counter.

5.4.2 Function call

In this section, we describe the interpreter for a function call. The call is of the form:

$$Expr = Call\ callee\ actuals,$$

where we have the information about *callee*'s name and the actual parameters from the function call.

$$makeArgs :: [Actual] \times [Formal] \times State \to [FrameArg] \times State$$

Figure 5.10: Construction of frame arguments

$$(v, state') = eval\ actual\ state$$
$$frameArg = StrictArg\ v$$

Figure 5.11: Construction of call-by-value frame argument

$$frameArg = ByNameArg\ actual$$

Figure 5.12: Construction of call-by-name argument

$$frameArg = LazyArg\ actual\ false\ empty$$

Figure 5.13: Construction of lazy argument

$$
\begin{aligned}
&eval\ (Call\ callee\ actuals, s) = (val, s''')\\
&where\\
&\quad (formals, funBody) = lookup\ callee\ FunctionsMap\\
&\quad (frameArgs, (mem', \_, nr\_frames')) = makeArgs\ actuals\ formals\ state\\
&\quad newFrame = Frame\ callee\ frameArgs\ []\ frameId\\
&\quad mem'' = push\ mem'\ newFrame\ (nr\_frames + 1)\ frameId\\
&\quad s' = (mem'', lastFrameId\ mem'', nr\_frames + 1)\\
&\quad (val, s'') = eval\ funBody\ s'\\
&\quad (s'' = (mem''',, nr\_frames''))\\
&\quad s''' = (mem''', frameId, nr\_frames'')
\end{aligned}
$$

Figure 5.14: Operational semantics for function call

First, we are going to present a function about constructing the function's arguments, given the actual parameters of the call and the formal parameters from the function's signature, lying in the function definition in *FunctionsMap*. So, let's pause for a while for the definition of this function, and then resume later for function call handling inside the interpreter.

The function, called *makeArgs* has the form, that is shown in figure 5.10, where the actuals and the formals are explained above and the *State* is the interpreter's current state.For every actual corresponding to every formal, we have the following, depending whether the parameter is in cbv, cbn or lazy evaluation order(abbr. e.o):

- cbv e.o.: As shown in figure 5.11 the function returns *(frameArg, state')*, where *frameArg* is added to *[FrameArg]* and *state'* is the next input state for *makeArgs*.
- cbn e.o.: As shown in figure 5.12, the state remains unchanged and thus *(frameArg, state)* is returned.
- lazy e.o.: As shown in figure 5.13 the state does not change, and thus *(frameArg, state)* is returned.

As it seems above, only cbv arguments are executed at this point, and the memory change they provoke, is returned in the state of the *eval* function.

Now, it' s time to resume the presentation about the function call in the interpreter. The operational meaning of a function call is shown in figure 5.14. Every time we have a function call a new frame is pushed to the memory. Then, the interpreter evaluates the body of the function, and finally we reset the environment for the next operation handled by the interpreter.

5.4.3 Pattern matching

As we have shown in section 4.5, we distinguish between pattern matching on integers and pattern matching on data. More specifically a branch for a pattern matching is:

```
type Branch  = (Pattern, Expr)
data Pattern = CPat { tag :: CN, vars :: [VN] } -- pattern matching on constructors
             | IPat { pattVal :: Int }          -- pattern matching on integers
```

$$caseExpr = Case\ caseId\ e\ branches$$

Figure 5.15: Case expression

$$\begin{aligned}
&eval\ (e, s) = (e', s'),\\
&where\\
&\quad s' = (mem', savedFrameId, n),\ and\\
&\quad e' = VI\ i,\ integer\ pattern\ matching, or\\
&\quad e' = VC\ c,\ constructor\ pattern\ matching\\
&\quad c = Susp\ (cn, _)\ _
\end{aligned}$$

Figure 5.16: Evaluation of the scrutinee

$$\begin{aligned}
&eval\ (caseExpr, s) = (eval\ e''\ s', s'),\\
&where\\
&\quad e'' = lookup\ (IPat\ i)\ cases\\
&\quad s\text{'} \textit{ is the state after scrutinee}'\textit{s evaluation}.
\end{aligned}$$

Figure 5.17: Integer pattern matching operational semantics

$$\begin{aligned}
&eval\ (caseExpr, s) = (eval\ e''\ s'', s''),\\
&where\\
&\quad pattIndex = indexOfPattern\ cn\ patterns\\
&\quad (_, e'') = branches\ [pattIndex]\\
&\quad susps' = (caseId, c)\ :\ susps\\
&\quad frame' = Frame\ caller\ funArgs\ susps'\ prevFrameId\\
&\quad s'' = (updFrame\ mem'\ frameId\ frame',\ frameId,\ n)
\end{aligned}$$

Figure 5.18: Pattern matching on constructors operational semantics

The case expression is shown in the figure 5.15. For its evaluation, the interpreter does the following:

- First, we evaluate the scrutinized expression, shown in figure 5.16.
- Next, in figure 5.17 we show how the interpreter works for integer pattern matching.
- Finally, the pattern matching on constructors is shown in figure 5.18.

In the above definitions, *indexOfPatterns* gives as output the index of the pattern in the *patterns* list, and this list is retrieved by the *branches* list by ignoring the second value.

Integer pattern matching cannot be exhaustive, because integer domain consists of infinite elements, and thus we assume that *IPat i* is a pattern that exists in the patterns list. Otherwise, the interpreter will throw an exception.

We note that even though our interpreter works for lists, this is applied to every data constructor, when 'case' works in the same way as Haskell's builtin 'case'. The 'constructor' representation would be the same for every constructor definition, as constructors are also functions themselves. The latter is obvious taking into consideration that we work on a functional language; a programming language with *functions as first-class citizens*.

5.4.4 Constructor application

In the case of having a constructor application of a *lazy data constructor*, this is equivalent to a memory allocation of a thunk. In our case, this is a suspended execution of lazy data constructor, the one explained in section 5.3.2. More specifically, the *eval* for a lazy constructor application is shown in figure 5.19.

$$eval\ (ConstrF\ tag\ exprs, s) = (VC\ (Susp\ (tag, exprs)\ frameId), s)$$

, *where*

$frameId$: *the id of the current frame existing in the interpeter's state*,

tag : *the name of the constructor*.

Figure 5.19: Lazy list constructor application operational semantics

5.4.5 Constructor projection

In this case, the interpreter has come across a variable bound by pattern matching. As we mentioned earlier (section 2.2.4), the evaluation of a term is forced, when such variables are needed to be evaluated. The *eval* function of *constructor projection* is shown in figure 5.20. First, we find the *susp*

$$eval\ (CProj\ caseId\ cpos, s) = (val, s'),$$

where

$$s' = mem'\ frameId\ nr\_frames'$$
$$susp = lookup\ cid\ susps$$
$$Susp\ (\_, el)\ savedFrameId = susp$$
$$e' = el[cpos]$$
$$(val, (mem', \_, nr\_frames')) = eval\ (e', s)$$

Figure 5.20: Constructor projection operational semantics

inside the *susps* in the current frame. Then, from the expression list *el* we find the right constructor in the *cpos* position. After evaluating this expression, we have the result-value and the next state for our interpreter.

Chapter 6

Analysis: Searching for optimisation opportunities

In this chapter, we present the analysis performed in order to reveal true tail-call positions. First, we make some important high-level definitions for the reader to be able to follow up. The presentation of the algorithms for each one of the optimisations will follow; section 6.1 contains the algorithm for classic tail-call optimisation and section 6.2 contains the algorithm for tail recursion modulo cons.

The input of the analysis is the core level language described in section 4.5, along with the operational semantics provided in the previous chapter. The output is appropriately *annotated* function calls which the enriched evaluator of the next chapter is going to handle. The analysis in this chapter reveals *where* to apply the optimisations, while in the next chapter we present *how* to actually perform the optimisations in the runtime system.

6.1 Classic tail-call optimisation

At this point, we shall make clear that the terms tail-call or tail-call position are not the same as tail-call optimisable, providing the definitions shown below.

6.1.1 Tail-call position (tail-call) vs. Tail-call optimisable (true tail-call)

Definition 6.1 A function call is in *tail call position*, or *tail call*, *if and only if* its execution is the last action performed before the function returns.

Definition 6.2 A function call is *tail call optimisable*, or *true tail call*, *if and only if* it is in tail call position and it satisfies the rules shown in section 6.1.3.

The same definitions also stand for tail recursion modulo cons, if we substitute *tail-call* with *tail recursion modulo cons*; an equivalent section is omitted for section 6.2.

In the next sections, we present the path from *function calls* in the core language to *tail-call positons*, and from there to *true tail-calls*; that is function calls that are actually optimisable. The latter is annotated as a new expression in the core language, called *TailCall*. When the interpreter comes across this expression, it performs the optimisation, while the program is executing.

6.1.2 Control-flow analysis: Spotting tail-call positions

For a call-by-value language, this step reveals each and every true tail-call position. For a language with mixed evaluation order, and specifically with call-by-need semantics this is not enough, as it has been already sketched in section 3.1.2, as lazy arguments are arbitrarily evaluated and thus they escape their context.

This is a local analysis, performed locally in a function body. The analysis is summarized in the following rules:

- The body of a lambda (or a function) is a tail call.

- When an if/case expression is in tail call position, then the right-hand side of the branches is in tail-call position. In the case of an ‘if’ expression, this means that then/else clause is in tail call position.
- Nothing else is in tail call position.

To illustrate the above, let’s consider the following, very simple example:

```
foo n =
  if n > 0
    then n * foo (n-1)
    else 1
```

The body of ‘foo’ function is in tail call position. This means that ‘if’, which is the body, is in tail call position. Thus, then/else are tail calls. Now in the ‘then’ clause, where the multiplication takes place, the call to ‘foo’ is not in tail call position, according to the third rule.

But if we had the following example:

```
foo n res =
  if n > 0
    then foo (n - 1) (n * res)
    else res
```

then the call is in tail-call position as the ‘then’ clause is in tail-call position.

6.1.3 Data-driven analysis: Revealing true tail-calls

After revealing tail-call position, deriving from the control flow of the program shown in the previous section, now we apply some rules in order to reveal true tail call positions, depending by the evaluation order of caller and callee function. After all, tail-call optimisation is all about passing the control from the caller to the callee. This step is omitted in call-by-value languages as there is only one evaluation order and the semantics allow the optimisation quite naturally.

Although it seems that the rules are applied in a second analysis pass, this is done for clarity purposes; in the implementation we can do that in-place, in a single analysis pass just after a tail call (and thus potentially a true tail call) is revealed.

The rules for a potential tail call (revealed by control flow analysis from section 6.1.2) to be a true tail call are:

- Actual parameters in the function call are not dependent by the formal parameters of the caller function. For example, these actual parameters can be constants.
- Actual parameters in the function call are variables. If they are variables we can statically predict when the runtime mutation is possible. The reader can also refer to figure 7.5 for possible cases. We perform the mutation for all the nine combinations, except for a subcase in two cases.
- Actual parameter is an expression, but it is in call-by-value position. In this case, we have two subcases:
  - The expression is a base type expression. This is easier as such expressions, when they are evaluated, they reduce to a base type value and thus they do not leave thunks. In this subcase, we could also have data like ML-lists, which also reduce to a value.
  - The expression is constructed data. In this case, we find all variables inside the expression. If these variables are reduced to value (base type values or deeply evaluated data), then the function call is tail-call optimisable.

Function calls that are not subject to the rules above are not tail call optimisable.

6.2 Tail recursion modulo cons

Tail recursion modulo cons can be applied to any constructor application, that is guarded by a lazy constructor. In our language this happens to (:), which is lazy and is the constructor for lists. It could also be applied to arithmetic operations, for example in Haskell, but in our language all arithmetic operations are strict.

The control flow of the program also reveals the tail recursion modulo cons in the same way as in classic tail calls. The only difference is that constructor application must be a tail call. Thus the rules for tail recursion modulo cons have the following form:

- The body of a lambda (or a function) is a tail call and it is also a lazy constructor application (instead of a function application).
- When an if/case expression is in tail call position, then the right-hand side of the branches is in tail-call position. In the case of an 'if' expression, this means that then/else clause is in tail call position. The right-hand side of a branch must also be a lazy constructor application.
- Nothing else is in tail recursion modulo cons position.

The additional rule for tail recursion modulo cons is that if you have a tail call of the form:

```
x : y
```

then `x` must not contain any recursive calls and `y` must be a function call that is subject to the rules presented in section 6.1.3; in this way frame mutation will become possible. If the last rule is applied, then we can have (`x :` *tail recursive call to the function*).

Chapter 7

Runtime evaluator for optimisations

In this chapter, we will present how the optimisations are implemented in the interpreter. More specifically, in section 7.1 we have the evaluation of tail-call optimisation, the expression shown in figure 7.1.

$$eval\ (e, s) = (e', s'), where$$
$$e = TailCall\ callee\ actuals$$

Figure 7.1: Annotated expressions

It is the classic tail-call optimisation. There will be a comprehensive explanation of how we mutated the frames and how we counted them for our evaluation, that follows in the next chapter.

7.1 Evaluator for classic tail-call optimisation

Classic tail call evaluation contains one major *operation*, which makes the optimisation feasible: this is *frame mutation*. This operation includes the replacement of the current frame's arguments with the arguments necessary for the construction of the frame for the next function call. For this operation, it is mandatory that the caller's arguments do not escape and for those that escape we correctly passed them to the new frame.

The *eval* for the *TailCall* follows in figure 7.2; we remind that this is actually optimisable as the analysis revealed, although we make few, inexpensive extra checks for some cases. The functionality of *checkMutate* will be explained later on in this section. The other variables, shown in figure 7.2 are:

- *formals* are the formal parameters of the function definition of the callee function,
- *actuals* are the actual parameter of this tail-call optimisable function call,
- *funArgs* are the arguments inside the current frame,
- and s is the current state of the interpreter.

$$\begin{aligned} &eval\ (TailCall\ callee\ actuals, s) = (e', s'), \\ &where \\ &\quad (formals, funBody) = lookup\ FunctionsMap\ callee \\ &\quad e' = eval\ (funBody, s') \\ &\quad s' = checkMutate\ formals\ actuals\ funArgs\ s \end{aligned}$$

Figure 7.2: Tail-call optimisation operational semantics

At this point, we are ready to provide the definition for *checkMutate*, which is a function that given the current interpreter's state performs the runtime mutation of the frame and produces the next state of the interpreter. Its declaration is given in the figure 7.3 and the operational meaning of the function in the figure 7.4.

$$checkMutate :: [Actual] \times [Formal] \times [FrameArg] \times State \rightarrow State$$

Figure 7.3: Declaration of *checkMutate* operation

$$\begin{aligned}
&checkMutate\ (actuals, formals, funArgs, s) = s',\\
&where\\
&\quad (mem, frameId, nr_frames) = s\\
&\quad (callerFormals, _) = lookup\ callee\ FunctionsMap\\
&\quad (funArgs', s'') = mutate\ callerFormals\ formals\ funArgs\ actuals\ s\\
&\quad (mem', frameId', nr_frames') = s''\\
&\quad frame'' = Frame\ callee\ funArgs'\ []\ prevFrameId\\
&\quad s' = (updFrame\ mem'\ frameId\ frame'', frameId, nr_frames')
\end{aligned}$$

Figure 7.4: Operational semantics for *checkMutate*

Now, we are going to describe an auxiliary function used in order to constructor the function's arguments of the mutated frame; specifically in the definition of *funArgs'* the function called *mutate*. The operational semantics for *mutate* operation is shown in figure 7.5. From the figure, we have that *mutate* is a function that takes as input:

- The formal parameters of the caller function,
- the formal parameters of the callee function,
- the frame arguments that exist in the current frame (the one we want to mutate),
- the actual parameter of the function call, which is tail-call optimisable,
- the current state of the interpreter,

and produces:

- The frame arguments for the new frame. We note here that we want to turn the actual parameters of the function call into the appropriate frame arguments, using the function signatures of the caller and the callee. Specifically, we are interested in the format of the arguments, and thus we need to know the evaluation order of the functions.
- The next state for the interpreter. Because evaluation *may* take place, while this function is executing (for example we need to turn a lazy frame argument into a strict and thus we perform the evaluation before we jump into the callee function body), we need to change the memory state and count the frames correctly for our evaluation that follows in the next chapter.

As the language is first-order, it does not support partial applications, and thus the number of arguments in the actual parameters list is equal to the number of formals of the callee function. The *mutate* function terminates when these two lists are empty at the same iteration, or else it throws an exception. Its execution starts and continues until it terminates by processing each and every actual parameter of the function call. The result is a list that accumulates the result of every iteration.

$$
\begin{aligned}
&\mathit{getIndex} :: \mathit{VN} \times \mathit{EvaluationOrder} \times \mathit{Type} \times [\mathit{Formal}] \to \mathit{Index}\\
&\mathit{getIndex}\ (\mathit{vn},\ \mathit{eo},\ \mathit{type})\ \mathit{fs} = \mathit{elemIndex}\ (\mathit{vn}, (\mathit{eo}, \mathit{type}))\ \mathit{fs}\\
&\\
&\mathit{getEvaluationOrder} :: \mathit{Formal} \to \mathit{EvaluationOrder}\\
&\mathit{getEvaluationOrder} :: (\_, (\mathit{eo}, \_)) = \mathit{eo}\\
&\\
&\mathit{signOfVar} :: [\mathit{Formal}] \times \mathit{VN} \to \mathit{EvalutionOrder} \times \mathit{Type}\\
&\mathit{signOfVar}\ \mathit{formals}\ \mathit{vn} = \mathit{lookup}\ \mathit{vn}\ \mathit{formals}\\
&\\
&\mathit{transform} :: \mathit{CallerEO} \times \mathit{CalleeEO} \times \mathit{FrameArg} \times \mathit{State} \to \mathit{FrameArg} \times \mathit{State}\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{CBV}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBN}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{Lazy}, \mathit{arg}, s) = (\mathit{arg}, s)\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{CBN}, \mathit{ByNameArg}\ e, s) = (\mathit{StrictArg}\ v, s'),\\
&\quad \mathit{where}\ (v, s') = \mathit{eval}\ (e, s)\\
&\mathit{transform}\ (\mathit{CBV}, \mathit{Lazy}, \mathit{LazyArg}\ e\ b\ v, s) = \mathit{if}\ b\ \mathit{then}\ (\mathit{StrictArg}\ v, s)\ \mathit{else}\ (\mathit{val}, s'),\\
&\ \mathit{where}\ (\mathit{val}, s') = \mathit{eval}\ (e, s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBV}, \mathit{StrictArg}\ (\mathit{VI}\ v), s) = (\mathit{ByNameArg}\ (\mathit{EInt}\ v), s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{CBV}, \mathit{StrictArg}\ (\mathit{VC}\ c), s) = \mathit{error}\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{true}\ (\mathit{VI}\ v), s) = (\mathit{ByNameArg}\ (\mathit{EInt}\ v), s)\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{true}\ (\mathit{VC}\ c), s) = \mathit{error}\\
&\mathit{transform}\ (\mathit{CBN}, \mathit{Lazy}, \mathit{LazyArg}\ e\ \mathit{false}\ v, s) = (\mathit{ByNameArg}\ e, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{CBV}, \mathit{StrictArg}\ v, s) = (\mathit{LazyArg}\ e\ \mathit{true}\ v, s)\\
&\mathit{transform}\ (\mathit{Lazy}, \mathit{CBN}, \mathit{ByNameArg}\ e, s) = (\mathit{LazyArg}\ e\ \mathit{false}\ \mathit{null}, s)\\
&\\
&\mathit{mutate} :: [\mathit{Formal}] \times [\mathit{Formal}] \times [\mathit{FrameArg}] \times [\mathit{Actual}] \times \mathit{Index} \times \mathit{State} \to [\mathit{FrameArg}] \times \mathit{State}\\
&\mathit{mutate}\ (\_,\ [], \_, [], \_,\ \mathit{acc},\ s) = (\mathit{acc},\ s)\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (\mathit{EVar}\ v\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}'\ :\ \mathit{acc}),\ s')\\
&\ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad \mathit{sign@}(\mathit{callerEO}, \mathit{callerType}) = \mathit{signOfVar}\ (\mathit{vn}, \mathit{callerfs})\\
&\qquad \mathit{callerIx} = \mathit{getIndex}\ (\mathit{vn}, \mathit{sign})\ \mathit{callerfs}\\
&\qquad \mathit{arg} = \mathit{args}[\mathit{callerIx}]\\
&\qquad (\mathit{arg}', s') = \mathit{transform}\ (\mathit{calleeEO}, \mathit{callerEO}, \mathit{arg}, s)\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (\mathit{a@}(\mathit{EInt}\ n)\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}\ :\ \mathit{acc}),\ s)\\
&\ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad \mathit{if}\ \mathit{calleeEO} = \mathit{CBV}\ \mathit{then}\ \mathit{StrictArg}\ (\mathit{VI}\ n)\ \mathit{else}\ \mathit{if}\ \mathit{calleeEO} = \mathit{CBN}\ \mathit{then}\ \mathit{ByNameArg}\ a\ \mathit{else}\ \mathit{LazyArg}\ a\ \mathit{false}\ \mathit{nu}\\
&\mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ (a\ :\ \mathit{as}),\ \mathit{ix},\ \mathit{acc},\ s)\\
&=\ \mathit{if}\ \mathit{calleeEO} = \mathit{CBV}\ \mathit{then}\ \mathit{mutate}\ (\mathit{callerfs},\ \mathit{calleefs},\ \mathit{args},\ \mathit{as},\ (\mathit{ix}+1), (\mathit{arg}'\ :\ \mathit{acc}),\ s')\\
&\ \mathit{where}\ \mathit{calleeEO} = \mathit{getEvaluationOrder}\ \mathit{calleefs}[\mathit{ix}]\\
&\qquad (v, s') = \mathit{eval}\ (a, s)\\
&\qquad \mathit{arg}' = \mathit{StrictArg}\ v
\end{aligned}
$$

Figure 7.5: Operational semantics for *mutate* operation

Chapter 8

Evaluation of the optimisations

In figure 8.1, we have the results from the evaluation on microbenchmarks in our interpreter.[1] The code of the microbenchmarks is given in appendix A.

- Our first example, `fact` is an example that uses integers with strictness annotations. We note here that if `fact` is not annotated, when strictness analysis is applied is transformed in the form presented in the appendix, and thus it is tail call optimisable.
- The example 'sum' is an example presented also in the overview chapter, and we were able to optimise it, in the same way as in a strict functional language. The same stands also for 'length'.
- The example 'average' is an example that traverses the list twice, but because of laziness it computes it once. Here as well, we observe a significant optimisation.
- The example 'makelist' is also one of the examples presented in the thesis, and showcases how tail call optimisation works when the function produces data.
- The example 'getNthFib' contains a function that potentially produces an infinite list. This example produces the n-th fibonacci number. The optimisation shows how we can have infinite data structures, while tail call optimisation can also work here, reducing memory overhead associated with recursion. The example 'ones' also works with infinite lists and with a much better optimisation than the 'fibs' example.
- The example 'last' is the most classic example of a program, that forces the evaluation of the input deeply; it pattern matches on every element of the list, causing its evaluation to finally find the last element. The use of tail call optimisation, as it seems in the figure, can make such pathological programs to run with much more efficient.

Programs/Mode	Without TCO	With TCO
fact	1002	1
sum	2003	1002
average	9605	4804
getNthFib	439204	439178
ones	2003	1002
fast-reverse	3007	3004
last	3001	2001

Figure 8.1: Evaluation on microbenchmarks

[1] Available in: `https://www.github.com/pbougou/diploma`

From the results of the evaluation presented in this section, we saw that our runtime in some cases approaches the tail call optimisation in strict functional languages and also in none of the cases it did not get worse. This makes us optimistic in order to embed our technique into a compiler for a lazy language.

Chapter 9

Related work

GHC, Haskell's de facto implementation, used Cmm and later LLVM to support tail calls [Tere10]. When GHC is targeting Cmm, it uses an optimisation called loopification [Woś10], which turns recursive calls into loops in the code generator, exploiting Haskell Core' s join points. Our approach bridges the gap between implementations for non-strict functional languages and the frame-based approaches for strict functional languages, with some differences our interpreter (section 5.1). Having explicitely control flow inside our core language is not something that could improve our approach, as it is based on data-driven analysis of the arguments of the caller and the callee.

Short cut fusion [Gill93, Pard16], an optimisation for functional programming languages, is based on work initially started with the listlessness optimization [Wadl84]. Listlessness is closely related to tail recursion modulo cons: "programs for the listless machine are related to programs compiled using an optimization called tail recursion, particularly tail recursion modulo cons" [Wadl84].

Chapter 10

Conclusion

This thesis shows how to extend classic tail-call optimization to lazy languages supporting alternative evaluation order choices (such as call-by-value and call-by-name). Our prototype implementation is an interpreter that uses information from a local static analysis, in order to detect opportunities for tail-call optimization at tail-call positions.

Our technique can also be implemented in a purely compile-time setting, integrated with a defunctionalizing compiler for lazy functional languages, such as GIC [Four14a] or GRIN [Boqu96, Podl19]. This integration needs a small code generator for stack frame mutation for tail calls, driven by the decisions currently made by our interpreter. Such an extension should be an obvious next step for our implementation.

Integration with a more complex compiler should also help with evaluating our technique with more realistic benchmarks, such as the *nofib* benchmark suite [Part93].

Moreover, Kerneis and Chroboczek proved that tail calls and lambda lifting are compatible in a call-by-value setting [Kern11]. Since our technique can be optionally combined with a lambda lifter (section 4.5), another future step in the context of this thesis would be to formalize the interaction between lambda lifting, tail calls, and our richer support for evaluation order choices.

In addition to Haskell, our approach can also be used in other languages that combine strictness and non-strictness in the same language, such as PLT Scheme [Barz05] and Scala [Wamp09].

Finally, although we described the analysis that spots optimisable tail recursion modulo cons positions, we leave the implementation of the runtime system for this optimisation for later work.

Βιβλιογραφία

[Abel13] Andreas Abel, Brigitte Pientka, David Thibodeau and Anton Setzer, "Copatterns: Programming Infinite Structures by Observations", in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pp. 27–38, New York, NY, USA, 2013, ACM.

[Appe92] Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, New York, NY, USA, 1992.

[Argo89] Guy Argo, "Improving the Three Instruction Machine", in *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pp. 100–115, New York, NY, USA, 1989, ACM.

[Augu84] Lennart Augustsson, "A Compiler for Lazy ML", in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pp. 218–227, New York, NY, USA, 1984, ACM.

[Augu85] Lennart Augustsson, "Compiling pattern matching", in Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, vol. 201 of *Lecture Notes in Computer Science*, pp. 368–381, Springer Berlin / Heidelberg, 1985. 10.1007/3-540-15975-4_48.

[Bare92] H. P. Barendregt, "Lambda Calculi with Types", in S. Abramsky, Dov M. Gabbay and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pp. 117–309, Oxford University Press, Inc., New York, NY, USA, 1992.

[Barr68] D. W. Barron, *Recursive techniques in programming*, American Elsevier, 1968.

[Barz05] Eli Barzilay and John Clements, "Laziness Without All the Hard Work: Combining Lazy and Strict Languages for Teaching", in *Proceedings of the 2005 Workshop on Functional and Declarative Programming in Education*, FDPE '05, pp. 9–13, New York, NY, USA, 2005, ACM.

[Baue03] Andreas Bauer, "Compilation of Functional Programming Languages using GCC—Tail Calls", Master's thesis, Institut für Informatik, Technische Universität München, 2003.

[Bigo99] Peter A. Bigot and Saumya Debray, "Return value placement and tail call optimization in high level languages", *The Journal of Logic Programming*, vol. 38, no. 1, pp. 1 – 29, 1999.

[Blos88] Adrienne Bloss, Paul Hudak and Jonathan Young, "Code optimizations for lazy evaluation", *LISP and Symbolic Computation*, vol. 1, no. 2, pp. 147–164, Sep 1988.

[Bobr73] Daniel G. Bobrow and Ben Wegbreit, "A Model and Stack Implementation of Multiple Environments", *Communications of the ACM*, vol. 16, no. 10, pp. 591–603, October 1973.

[Boli09] Maximilian C. Bolingbroke and Simon L. Peyton Jones, "Types Are Calling Conventions", in *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pp. 1–12, New York, NY, USA, 2009, ACM.

[Boqu96] Urban Boquist and Thomas Johnsson, "The GRIN Project: A Highly Optimising Back End for Lazy Functional Languages", in *Proceedings of IFL '96, volume 1268 of LNCS*, pp. 58–84, Springer-Verlag, 1996.

[Burn96] Geoffrey Burn and Daniel Le Métayer, "Proving the correctness of compiler optimisations based on a global analysis: a study of strictness analysis", *Journal of Functional Programming*, vol. 6, no. 1, p. 75–109, 1996.

[Cejt00] Henry Cejtin, Suresh Jagannathan and Stephen Weeks, "Flow-Directed Closure Conversion for Typed Languages", in Gert Smolka, editor, *Programming Languages and Systems*, vol. 1782 of *Lecture Notes in Computer Science*, pp. 56–71, Springer Berlin Heidelberg, 2000.

[Clin98] William D. Clinger, "Proper Tail Recursion and Space Efficiency", in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pp. 174–185, New York, NY, USA, 1998, ACM.

[Cout07] Duncan Coutts, Roman Leshchinskiy and Don Stewart, "Stream Fusion. From Lists to Streams to Nothing at All", in *ICFP'07*, 2007.

[Danv05] Olivier Danvy, "A Rational Deconstruction of Landin's SECD Machine", in *Proceedings of the 16th International Conference on Implementation and Application of Functional Languages*, IFL'04, pp. 52–71, Berlin, Heidelberg, 2005, Springer-Verlag.

[Dijk60] Edsger W. Dijkstra, "Recursive Programming", *Numerische Mathematik*, vol. 2, pp. 312–318, 1960.

[Fair87] John Fairbairn and Stuart Wray, "TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators", in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pp. 34–45, Berlin, Heidelberg, 1987, Springer-Verlag.

[Four13] Georgios Fourtounis, Nikolaos Papaspyrou and Panos Rondogiannis, "The Generalized Intensional Transformation for Implementing Lazy Functional Languages", in *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages - Volume 7752*, PADL 2013, pp. 157–172, Berlin, Heidelberg, 2013, Springer-Verlag.

[Four14a] Georgios Fourtounis, *The Intensional Transformation for Implementing Functional Programming Languages*, Ph.D. thesis, National Technical University of Athens, 2014.

[Four14b] Georgios Fourtounis, Nikolaos Papaspyrou and Panagiotis Theofilopoulos, "Modular Polymorphic Defunctionalization", *Computer Science and Information Systems*, vol. 11, no. 4, 2014.

[Gill65] Stanley Gill, "Automatic computing: its problems and prizes", *The Computer Journal*, vol. 8, no. 3, pp. 177–189, 1965.

[Gill93] Andrew Gill, John Launchbury and Simon L. Peyton Jones, "A Short Cut to Deforestation", in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pp. 223–232, New York, NY, USA, 1993, ACM.

[Hold10] Stefan Holdermans and Jurriaan Hage, "Making "Stricterness" More Relevant", *Higher Order Symbol. Comput.*, vol. 23, no. 3, pp. 315–335, September 2010.

[Hugh89] J. Hughes, "Why Functional Programming Matters", *Comput. J.*, vol. 32, no. 2, pp. 98–107, April 1989.

[John85] Thomas Johnsson, "Lambda Lifting: Transforming Programs to Recursive Equations", in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pp. 190–203, New York, NY, USA, 1985, Springer-Verlag New York, Inc.

[Jone90] Richard Jones, "Tail Recursion Without Space Leaks", Technical Report 72*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, March 1990.

[Jone92] Simon L. Peyton Jones, "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine", *Journal of Functional Programming*, vol. 2, pp. 127–202, 1992.

[Jone95] Simon Peyton Jones and André Santos, "Compilation by Transformation in the Glasgow Haskell Compiler", in Kevin Hammond, David N. Turner and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1994*, pp. 184–204, London, 1995, Springer London.

[Kern11] Gabriel Kerneis and Juliusz Chroboczek, "Continuation-Passing C, compiling threads to events through continuations", *Higher-Order and Symbolic Computation*, vol. 24, no. 3, pp. 239–279, Sep 2011.

[Knut74] Donald E. Knuth, "Structured Programming with Go to Statements", *ACM Computing Surveys*, vol. 6, no. 4, pp. 261–301, December 1974.

[Land64] Peter J. Landin, "The Mechanical Evaluation of Expressions", *The Computer Journal*, vol. 6, no. 4, pp. 308–320, 1964.

[Laun93] John Launchbury, "A natural semantics for lazy evaluation", in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pp. 144–154, New York, NY, USA, 1993, ACM.

[Mads18] Magnus Madsen, Ramin Zarifi and Ondřej Lhoták, "Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine", in *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pp. 139–150, New York, NY, USA, 2018, ACM.

[Marl06] Simon Marlow and Simon Peyton Jones, "Making a fast curry: push/enter vs. eval/apply for higher-order languages", *Journal of Functional Programming*, vol. 16, no. 4-5, pp. 415–449, 2006.

[McCa60] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, April 1960.

[McCa62] John McCarthy, "Towards a Mathematical Science of Computation", in *IFIP Congress*, pp. 21–28, 1962.

[Mycr80] Alan Mycroft, "The Theory and Practice of Transforming Call-by-need into Call-by-value", in *Proceedings of the Fourth 'Colloque International Sur La Programmation' on International Symposium on Programming*, pp. 269–281, Berlin, Heidelberg, 1980, Springer-Verlag.

[Naur81] Peter Naur, "The European Side of the Last Phase of the Development of ALGOL 60", in Richard L. Wexelblat, editor, *History of Programming Languages I*, pp. 92–139, ACM, New York, NY, USA, 1981.

[Niel10] Flemming Nielson, Hanne R. Nielson and Chris Hankin, *Principles of Program Analysis*, Springer Publishing Company, Incorporated, 2010.

[Pand15] Mayur Pandey and Suyog Sarda, *LLVM Cookbook*, Packt Publishing, 2015.

[Pard16] Alberto Pardo, João Paulo Fernandes and João Saraiva, "Multiple intermediate structure deforestation by shortcut fusion", *Science of Computer Programming*, vol. 132, pp. 77 – 95, 2016. Selected and extended papers from SBLP 2013.

[Part93] Will Partain, "The nofib Benchmark Suite of Haskell Programs", in John Launchbury and Patrick Sansom, editors, *Functional Programming, Glasgow 1992*, pp. 195–202, London, 1993, Springer London.

[Peyt91] Simon L. Peyton Jones and John Launchbury, "Unboxed Values As First Class Citizens in a Non-Strict Functional Language", in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 636–666, Berlin, Heidelberg, 1991, Springer-Verlag.

[Peyt98] Simon L. Peyton Jones and André L. M. Santos, "A Transformation-based Optimiser for Haskell", *Sci. Comput. Program.*, vol. 32, no. 1-3, pp. 3–47, September 1998.

[Plas99] Rinus Plasmeijer and Marko van Eekelen, "Keep It Clean: A Unique Approach to Functional Programming.", *SIGPLAN Not.*, vol. 34, no. 6, pp. 23–31, June 1999.

[Podl19] Péter Dávid Podlovics, Csaba Hruska and Andor Pénzes, "A modern look at GRIN, an optimizing functional language back end", in *Conference on Software Technology and Cyber Security (STCS 2019)*, 2019.

[Prob01] Mark Probst, "Proper Tail Recursion in C", Diplomarbeit, Technische Universität Wien, 2001.

[Reyn72] John C. Reynolds, "Definitional interpreters for higher-order programming languages", in *Reprinted from the proceedings of the 25th ACM National Conference*, pp. 717–740, ACM, 1972.

[Sper10] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler and Jacob Matthews, *Revised [6] Report on the Algorithmic Language Scheme*, Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[Stee76] Guy L Steele and Gerald J Sussman, "Lambda: The Ultimate Imperative", Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[Stee78] Guy L. Steele, Jr., "Rabbit: A Compiler for Scheme", Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[Stoy79] Herbert Stoyan, "LISP History", *Lisp Bulletin*, no. 3, pp. 42–53, December 1979.

[Suss75] Gerald J. Sussman and Guy L. Steele, Jr., "An Interpreter for Extended Lambda Calculus", Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[Tere10] David A. Terei and Manuel M.T. Chakravarty, "An LLVM Backend for GHC", in *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pp. 109–120, New York, NY, USA, 2010, ACM.

[Theo13] Panagiotis Theofilopoulos, "An efficient implementation of lazy functional programming languages based on the generalized intensional transformation", Master's thesis, Department of Mathematics, National and Kapodistrian University of Athens, 2013.

[Torc12] Linda Torczon and Keith Cooper, *Engineering A Compiler*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012.

[Turn79] D. A. Turner, "A new implementation technique for applicative languages", *Software: Practice and Experience*, vol. 9, no. 1, pp. 31–49, 1979.

[Turn85] D. A. Turner, "Miranda: A Non-strict Functional Language with Polymorphic Types", in *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pp. 1–16, New York, NY, USA, 1985, Springer-Verlag New York, Inc.

[Wadl84] Philip Wadler, "Listlessness is Better Than Laziness: Lazy Evaluation and Garbage Collection at Compile-time", in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pp. 45–52, New York, NY, USA, 1984, ACM.

[Wadl87] P. Wadler, "Efficient Compilation of Pattern-Matching", in S. L. Peyton-Jones, editor, *The Implementation of Functional Programming Languages*, pp. 78–103, Prentice-Hall International, 1987.

[Wadl03] Philip Wadler, "Call-by-value is Dual to Call-by-name", *SIGPLAN Not.*, vol. 38, no. 9, pp. 189–201, August 2003.

[Wait84] William M. Waite and Gerhard Goos, "Properties of Real and Abstract Machines", in *Compiler Construction*, Texts and Monographs in Computer Science, pp. 46–84, Springer New York, 1984.

[Wamp09] Dean Wampler and Alex Payne, *Programming Scala: Scalability = Functional Programming + Objects*, O'Reilly Media, Inc., 1st edition, 2009.

[Woś10] Krzysztof Woś, "Low-level code optimisations in the Glasgow Haskell Compiler", May 2010. Computer Science Tripos, St John's College.

Παράρτημα Α

Benchmarks

A.1 fact

```
  fact n acc = if n > 0 then fact (n-1) (n*acc) else acc

  main = fact 1000 1
```

A.2 average

```
fun sum l1 !acc  =
    case l1 of
      h : t -> sum t (h + acc)
      Nil   -> acc

fun length l !acc  =
    case l of
      h : t -> length t (acc + 1)
      []    -> acc

makeList n  = if n > 0 then n : (makeList (n - 1)) else []

average l  = (sum l 0) / (length l 0)

main  = average (makeList 2400)
```

A.3 fibs

```
  fun next !l =
      case l of
          [] -> []
          a : t ->
            case t of
              [] -> []
              b : d -> (a + b) : (next t)

  fun fibs = 0 : 1 : next fibs

  fun main = idx 25 fibs

  fun idx !n !l =
      case l of
```

```
        [] -> -2
        a : b ->
            if n > 0 then idx (n-1) b else a
```

A.4 ones

```
ones = 1 : ones

fun main = take 1000 ones

fun take !n !l =
    case l of
        [] -> -2
        a : b ->
            if n > 0 then take (n-1) b else a
```

A.5 last

```
fun last !l  =
    case l of
      [] ->  -1
      x : xs ->
        case xs of
          [] -> x
          y : ys -> last xs

fun createCons !n = if n > 0 then n : (createCons (n - 1)) else []

fun main  = last (createCons 1000)
```