Υποβολή εργασίας Μεταγλωττιστών

Παπαδόπουλος Κωνσταντίνος-Νικόλαος 03118220

Ιανουάριος 2024

Έχω υλοποιήσει τον μεταγλωτιστή για τη γλώσσα Grace σε OCaml με χρήση των εργαλείων ocamllex για την λεκτική ανάλυση, menhir για τη συντακτική ανάλυση και LLVM για τη παραγωγή κώδικα. Έχω επίσης υλοποιήσει τα προαιρετικά τμήματα με βελτιστοποίηση τόσο στο παραγώμενο AST όσο και στον ενδιάμεσο/τελικό κώδικα, και την runtime library, γραμμένη σε C.

Εγκατάσταση και χρήση

Για την εγκατάσταση πρέπει να βρίσκονται προεγκαταστημένα τα εξής πακέτα:
- ocaml
- clang
- ar
- llvm και llvm-dev
- opam και dune

Επίσης χρειάζεται να εγκατασταθούν opam packages εκτελώντας `opam install dune menhir core core_unix llvm`

Για να χτίσετε τον μεταγλωτιστή, κατευθυνθείτε μέσα στον υποφάκελο grace_compiler και εκτελέστε '`make`' (αν δεν έχετε το opam στο bash environment εκτελέστε πρώτα '`eval $(opam config env)`'). Μαζί με τον μεταγλωττιστή θα γίνει compile και η runtime library που βρίσκεται στον φάκελο runtime_library. Υπάρχουν επίσης οι επιλογές '`make clean`' και '`make disclean`'.
Μετά το χτίσιμο του μεταγλωτιστή, παράγεται το εκτελέσιμο '`grace_compiler/gracec.exe`' (που δέχεται τα γνωστά arguments και flags).

Λεπτομέρειες σχεδίασης και υλοποίησης

Παρακάτω θα περιγράφω μόνο τα σημεία σε κάθε στάδιο του μεταγλωττιστή που θεωρώ πως αξίζει να κάνω κάποιο σχόλιο ή που έχω κάνει κάποια δική μου παραδοχή.

ASTs

Κάθε στάδιο του μεταγλωττιστή από τη συντακτική ανάλυση και έπειτα (semantic analysis, ast optimizations, ir code genration) δέχεται και επιστρέφει ένα AST. Ήθελα να μπορώ κάθε variant του AST να αποθηκεύει διαφορετικά metadata για τα nodes (πχ AST με πληροφορίες για τη θέση κάθε κόμβου στο πρόγραμμα για τη συντακτική ανάλυση, typed AST για τη σημασιολογική ανάλυση, etc) αλλά όλα να μοιράζονται την ίδια δομή. Λόγω αυτού υλοποιώ τη δομή του AST ως ένα functor που δέχεται ένα parameterized type και παράγει ένα AST με το type αυτό ως metadata. Έτσι κάθε AST variant είναι ένα module που προκύπτει ως εφαρμογή αυτού του functor με διαφορετικό metadata type.

Σημασιολογική ανάλυση

Χρησιμοποιώ ένα (mutable) symbol table που αποθηκεύω τύπους συναρτήσεων, παραμέτρων και μεταβλητών κατά τη δήλωσή τους και τα οποία ανακαλώ/αναζητώ κατά το type checking.

Επιπλέον από όσα αναφέρονται στην εκφώνηση, απαγορεύω τη δεικτιοδότηση σταθερής συμβολοσειράς (eg `"hi"[1]`), τη δεικτιοδότηση πίνακα με χαρακτήρα (eg `a['a']`) και δεν ελέγχω ότι οι δεικτιοδοτήσεις σε πίνακα που μπορούν να γίνουν evaluate σε compile time βρίσκονται εντός ορίων του πίνακα. Επίσης, απαιτώ οι πραγματικές παράμετροι που είναι πίνακες να έχουν το ίδιο μέγεθος με τις τυπικές. Εξαίρεση αποτελεί η περίπτωση που ένας πίνακας ως πραγματική παράμετρος εμφανίζεται incomplete στο declaration αλλά complete στο definition όπου εκεί επιτρέπω στην τυπική παράμετρο να έχει και διαφορετικό μέγεθος.

Παραγωγή LLVM IR

Στην Grace η top-level συνάρτηση του προγράμματος μπορεί να έχει οποιοδήποτε όνομα (όχι αναγκαστικά main), επομένως την τυλίγω σε μια συνάρτηση που ονομάζω main η οποία καλεί την top-level και επιστρέφει 0. Επειδή στο LLVM απαιτούνται μοναδικά ονόματα συναρτήσεων, μετονομάζω όλες τις συναρτήσεις με μοναδικό όνομα κατά την διάρκεια των AST optimizations (βλ. παρακάτω).

Οι μεταβλητές γίνονται allocate στο stack και έτσι επιτυγχάνεται το mutability αυτών. Στους ορισμούς συναρτήσεων, οι τυπικές παράμετροι αντιγράφονται επίσης στο stack (τη στιγμή που γίνονται reference για πρώτη φορά) για να μπορούν να μεταβληθούν.

Χρησιμοποιείται και σε αυτό το στάδιο ένα symbol table για να αποθηκεύει τις LLVM μονάδες συναρτήσεων (χρειάζεται να ανακτηθούν στα function calls) και τις θέσεις μνήμης των μεταβλητών.

Οι by reference παράμετροι υλοποιούνται ως pointers και οι πίνακες ως pointers σε (llvm) πίνακα (ομοίως και οι πίνακες πολλών διαστάσεων). Τα incomplete arrays υλοποιούνται ως pointers στο πρώτο στοιχείο τους.

Επειδή κάθε LLVM basic block πρέπει να τερματίζει, προσθέτω ένα return statement στο τέλος κάθε συνάρτησης που δεν τερματίζει με return (αυτό μπορεί να συμβεί όταν μια συνάρτηση επιστρέφει σε όλα τα cases κάποιου if και άρα το προστιθέμενο return θα είναι dead code, αλλά χρειάζεται για τον λόγο που εξήγησα). Αν όντως μια συνάρτηση επιστρέφει μέσα σε προγενέστερο if, για να μην υπάρχουν ταυτόχρονα return και branch στο basic block του if, κάνω evaluate τα conditions σε ξεχωριστό basic block.

Βελτιστοποιήσεις

Υλοποιώ lambda lifting στο AST πριν την παραγωγή κώδικα και εξαλείφω τις nested functions μετατρέποντάς τις σε global, περνόντας όλες τις ελεύθερες μεταβλητές αυτών (ή των nested ορισμένων συναρτήσεών τους) ως by reference parameters. Αυτό εκτιμώ οτι αν συνδυαστεί με τα reg2mem και constant folding passes του LLVM δημιουργεί περαιτέρω ευκαιρίες για βελτιστοποίηση, αλλά εξυπηρετεί και έναν πιο πρακτικό σκοπό καθώς στο LLVM δεν γίνεται να έχουμε πρόσβαση στα activation records των parent συναρτήσεων και ως εκ τούτου δεν μπορούμε να υλοποιήσουμε διαφορετικά την πρόσβαση μιας εμφωλευμένης συνάρτησης σε μεταβλητή μιας εξωτερικής.

Σε επίπεδο τελικού κώδικα, χρησιμοποιούμε τα παρακάτω LLVM optimization passes:

- memory_to_register_promotion (πολύ χρήσιμο δεδομένου πως υλοποιούμε τις μεταβλητές)
- constant_propagation (ομοίως)
- reassociation (facilitates better constant propagation)
- global value numbering (καθώς επίσης εξαλείφει περιτά loads)
- dead code elimination
- function_inlining