ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μεταγλωτιστές

8ο εξάμηνο, Ακαδημαϊκή περίοδος 2012

Υλοποίηση της γλώσσας *Llama*

Κωνσταντίνος Αθανασίου 03108132

Νίκος Γιανναράκης 03108054

Ζωή Παρασκευοπούλου 03108152

05/11/2013

Περιεχόμενα

1 Εισαγωγή

Η *Llama* είναι μια γλώσσα η οποία συνδυάζει στοιχεία συναρτησιακού και προστακτικού προγραμματισμού και συντακτικά μοιάζει αρκετά με τη γλώσσα Caml. Η εργασία έχει υλοποιηθεί στη γλώσσα OCaml. Αναλυτικά οι προδιαγραφές τις γλώσσας βρίσκονται εδώ

2 Υλοποίση

Στην ενότητα αυτή θα παρουσιάσουμε συνοπτικά κάποια βασικά στοιχεία της υλοποίησης.

2.1 Συντακτικός και Γραμματικός Αναλυτής

Για τον συντακτικό αναλυτή χρησιμοποιήθηκε το εργαλείο Ocamllex και για τον γραμματικό αναλυτή το εργαλείο Ocamlyacc.

2.2 Σημασιολογική Ανάλυση

Ο σημασιολογικός αναλυτής της *Llama* αποτελείται κυρίως από την εξαγωγή τύπων καθώς και από τον έλεγχο κάποιων επιπλέον περιορισμών.

2.2.1 Εξαγωγή Τύπων

Η εξαγωγή τύπων βασίζεται στην παραγωγή περιορισμών, υπό τη μορφή εξισώσεων τύπων, κατά τη διάσχιση του ast, κάθε άγωστος τύπος αναπαριστάται από μια μοναδική μεταβλητή τύπου. Στο τέλος της διάσχισης καλείται η συνάρτηση unify η οποία λύνει τους περιορισμούς αναθέτει σε κάθε μεταβλητή τύπου έναν γνωστό τύπο. Σε περίπτωση όπου οι περιορισμοί δεν είναι επιλύσιμοι επιστρέφεται μήνυμα λάθους το οποίο επισημαίνει του δύο τύπους που δεν μπορούν να ενοποιηθούν.
Στην γλώσσα Llama οι τελεστές σύγκρισης υποστηρίζονται μόνο για τους τύπους int, float και char. Προκειμένου να ελέγξουμε στατικά αυτόν τον περιορισμό δημιουργούμε έναν καινούριο τύπο ord που αναπαριστά τους τύπους που υποστηρίζουν τελεστές διάταξης. Έτσι προκύπτει ο παρακάτω κανόνας τυποποίησης:

$$\frac{\Gamma \vdash e_1 : ord \quad \Gamma \vdash e_2 : ord}{\Gamma \vdash e_1 \diamond e_2 : bool, \diamond \in \{<, >, \leq, \geq\}}$$

Μετά την δημιουργία των αντιστοίχων constraints η συνάρτηση unify συσσωρεύει τούς τύπους που προκύπτει από τους περιορισμούς ότι πρέπει να υποστηρίζουν διάταξη και αφού λύσει όλους τους περιορισμούς και εφαρμόσει όλες τις αντικαταστάσεις που προκύπτουν για τις μεταβλητές τύπων, ελέγχει αν οι συσσωρευμένοι τύποι είναι από τους 3 τύπους που υποστηρίζουν διάταξη, διαφορετικά γυρνάει μήνυμα λάθους. Με ανάλογο τρόπο ελέγχεται ότι ο τύπος επιστροφής των συναρτήσεων δεν είναι τύπος συνάρτησης κάτι που απαγορεύεται στη *Llama* .

2.2.2 Έλεγχος επιπλέον περιορισμών

2.3 Πίνακες

Οι πίνακες σε σημασιολογικό επίπεδο αναπαρίστανται ως μία μεταβλητή με τύπο που αποτελείται τον τύπο των στοιχείων που περιλαμβάνει καθώς επίσης τον αριθμό των διαστάσεων του. Οι διαστάσεις του

πίνακα είναι γνωστές εξ αρχής, αντίθετα ο τύπος των στοιχείων του γίνεται γνωστός μέσω της εξαγωγής τύπων. Καθώς ο αριθμός των διαστάσεων του πίνακα αποτελούν μέρος του τύπου του πίνακα, αναφορά σε υπο-πίνακες (δηλαδή όχι σε συγκεκριμένα στοιχεία) δεν επιτρέπονται από το σύστημα τύπων. Οι πίνακες στον ενδιάμεσο κώδικα αναπαριστάνονται μέσω της τετράδας

$$Array, a, [i_1, i_2, \ldots, i_n], ptr$$

όπου a είναι ένας πίνακας, $[i_1, i_2, \ldots, i_n]$ είναι μία λίστα κάθε στοιχείο της οποίας είναι ένας ακέραιος που αντιστοιχεί σε μία θέση πίνακα σε κάθε διάσταση του και ptr είναι ένας δείκτης προς το στοιχείο του πίνακα στη θέση που προσδιορίζουν τα $i_1 \ldots i_n$. Αυτή η αναπαράσταση μας διευκολύνει στην παραγωγή τελικού κώδικα. Στον τελικό κώδικα ένας πίνακας αναπαρίσταται ως όπου d_i είναι το μέγεθος της i-οστής

d_1	d_2	$\ldots$	d_n	$a_{00\ldots0}$	$a_{00\ldots1}$	$\ldots$	$a_{10\ldots0}$	$\ldots$	$a_{(n-1)(n-1)\ldots(n-1)}$

διάστασης και $a_{ij\ldots k}$ είναι τα στοιχεία του πίνακα αποθηκευμένα κατά γραμμές. Έτσι το μέγεθος κάθε διάστασης είναι προσβάσιμο σε χρόνο $\mathcal{O}(1)$. Η πρόσβαση σε ένα στοιχείο του πίνακα είναι πιο ακριβή διαδικασία και εξαρτάται από τον αριθμό των διαστάσεων του πίνακα. Για την πρόσβαση σε ένα στοιχείο χρειάζεται να υπολογιστεί δυναμικά η παρακάτω έκφραση

$$i_1 \cdot (d_2 \cdot d_3 \cdot \ldots d_n) + i_2 \cdot (d_3 \cdot \ldots \cdot d_n) + \ldots i_n$$

όπου i_k είναι το index που αντιστοιχεί σε κάθε διάσταση και d_j είναι το μήκος κάθε διάστασης.

2.4 Συναρτήσεις Υψηλής Τάξης

Στη *Llama* οι συναρτήσεις μπορούν να περαστούν ως παράμετροι σε μία συνάρτηση ή να ανατεθούν σε κάποια μεταβλητή. Μία συνάρτηση ωστόσο δε μπορεί να επιστρέψει συνάρτηση ως αποτέλεσμα. Η αναπαράσταση μίας τέτοιας συνάρτησης στον τελικό κώδικα γίνεται ως ένα ζεύγος (closure) που αποτελείται από έναν δείκτη προς τον κώδικα της συνάρτησης και ένα δείκτη προς το περιβάλλον αυτής. Το περιβάλλον της συνάρτησης περιέχει τιμές για κάθε ελεύθερη μεταβλητή που εμφανίζεται στον κώδικα της. Έτσι σε κάθε συνάρτηση που παίρνει ως όρισμα μία άλλη συνάρτηση μπορούμε να θεωρήσουμε ότι περνάμε ένα όρισμα ακόμα που είναι το περιβάλλον της συνάρτησης-παραμέτρου.

code_ptr
env_ptr

Πίνακας 1: Αναπαράσταση closure

2.5 Τύποι Οριζόμενοι από τον Προγραμματιστή

2.5.1 Αναπαράσταση

2.5.2 Υλοποίηση Δομικής Ισότητας

2.6 Control Flow Graph

Για την υλοποίηση του γράφου ροής ελέγχου αρχικά χωρίσαμε της τετράδες του ενδιάμεσου κώδικα σε Blocks έτσι ώστε να υπάρχουν άλματα μόνο ως προς την πρώτη εντολή κάθε Block και οι εντολές άλματος

(γενικότερα οι εντολές αλλαγής ροής του προγράμματος) να βρίσκονται μόνο στο τέλος κάποιου Block. Σε κάθε Block πέρα από τις αριθμημένες τετράδες που το αποτελούν, μέσω ενός record αποθηκεύουμε και επιπλέον πληροφορίες που διευκολύνουν την ανάλυση του προγράμματος σε αυτή τη μορφή, όπως για παράδειγμα το αν ορίζεται κάποια συνάρτηση μέσα σε αυτό το Block ή σε ποια συνάρτηση ανήκουν οι τετράδες του κάθε Block και αν κάποιο Block περιλαμβάνει το τέλος μίας συνάρτησης (Endu).
Κάθε κόμβος του γράφου ροής ελέγχου είναι ένα Block με ένα παραπάνω στοιχείο, ένα set που περιλαμβάνει τον αριθμό των τετράδων που περιέχονται στο Block. Στην πραγματικότητα θα έπρεπε και αυτή η πληροφορία να περιλαμβάνεται στο παραπάνω record. Με βάση τις πληροφορίες που έχουμε κρατήσει σε κάθε κόμβο η κατασκευή του γράφου ροής ελέγχου είναι σχετικά απλή, αρκεί να τηρήσουμε τους παρακάτω κανόνες.

- Αν στο τέλος κάποιου Block υπάρχουν εντολές διακλάδωσης (if, match, jump) τότε προσθέτουμε μια ακμή προς το Block που περιέχει την εντολή προς την οποία γίνεται το άλμα. Το Block αυτό μπορεί να βρεθεί σε χρόνο $\mathcal{O}(m \cdot \log n)$ όπου m ο αριθμός των Blocks και n ο αριθμός των τετράδων σε κάθε Block.

- Αν στο τέλος κάποιου Block υπάρχει κλήση συνάρτησης τότε προσθέτουμε μία ακμή προς το Block που περιέχει τη δήλωση της συνάρτησης την οποία καλούμε και μία ακμή από το Block που περιέχει το τέλος της συνάρτησης που καλέσαμε προς το Block που περιέχει την αμέσως επόμενη εντολή από την κλήση της συνάρτησης. Η εύρεση των Block που περιλαμβάνουν τη δήλωση της συνάρτησης που καλείται και το τέλος αυτής αντίστοιχα γίνεται σε χρόνο $\mathcal{O}(m)$ όπου m ο αριθμός των Blocks και η εύρεση του Block με την επόμενη εντολή σε χρόνο $\mathcal{O}(m \cdot \log n)$ όπου m ο αριθμός των Blocks και n ο αριθμός των τετράδων σε κάθε Block.

- Αν δεν υπάρχει κάποια εντολή αλλαγής ροής όπως οι παραπάνω, τότε αρκεί να προσθέσουμε μία ακμή στο Block που περιλαμβάνει την επόμενη εντολή που μπορεί να βρεθεί σε χρόνο $\mathcal{O}(m \cdot \log n)$ όπου m ο αριθμός των Blocks και n ο αριθμός των τετράδων σε κάθε Block.

Θα πρέπει να σημειωθεί βέβαια ότι η παραπάνω διαδικασία ίσως να είναι πιο πολύπλοκη από ότι δείχνει, ανάλογα με τη γλώσσα την οποία στοχεύει. Για παράδειγμα στη *Llama* επιτρέπεται το πέρασμα μίας συνάρτησης ως παραμέτρου πράγμα που δυσκολεύει την κατασκευή ενός γράφου ροής ελέγχου στην περίπτωση που έχουμε κλήση αυτής της συνάρτησης-παραμέτρου. Μπορούν να γίνουν διαφόρων ειδών αναλύσεις ώστε να περιοριστεί το εύρος των τιμών που μπορεί να έχει μία τέτοια μεταβλητή/παράμετρος. Ωστόσο στην υλοποίηση μας διαλέξαμε την τετριμμένη λύση, κάθε τέτοια κλήση μπορεί να είναι προς οποιοδήποτε block που περιέχει μία συνάρτηση. Για την δημιουργία του γράφου χρησιμοποιήθηκε η βιβλιοθήκη Ocamlgraph. Αν θέλει κάποιος να εξετάσει τον γράφο ροής ελέγχου που παράγει ο Alpaca μπορεί να το κάνει δίνοντας το όρισμα -g. Αυτό θα εξάγει το γράφο ροής ελέγχου του δοθέντος αρχείου πηγαίου κώδικα σε μορφή .dot που μπορεί να μετατραπεί σε εικόνα με κάποιο λογισμικό όπως το Graphviz.

2.7 Βελτιστοποιήσεις

Στον Alpaca υπάρχουν δύο στάδια βελτιστοποιήσεων, ένα στην ενδιάμεση μορφή τετράδων και ένα στον τελικό κώδικα. Όλες οι βελτιστοποιήσεις είναι τοπικές ωστόσο καθώς δεν έχει γίνει ανάλυση ροής δεδομένων.

2.7.1 Απλοποίηση υπολογισμών στις τετράδες

Αν και η απλοποίηση αυτή θα μπορούσε να είχε γίνει στο στάδιο παραγωγής ενδιάμεσου κώδικα θα έκανε το στάδιο αυτό πιο πολύπλοκο και προτιμήθηκε να υλοποιηθεί σε κάποιο επόμενο στάδιο. Σκοπός είναι να

Generated	Optimized
+a,b,$1 :=,$1,-,c	+a,b,c

Πίνακας 2: Temp Reduction

μη χρησιμοποιούνται προσωρινές μεταβλητές για την εκτέλεση των διαφόρων πράξεων αλλά να γίνονται απευθείας πάνω στις μεταβλητές που πρέπει.

2.7.2 Βελτιστοποιήσεις βασισμένες στο γράφο ροής ελέγχου

Οι βελτιστοποιήσεις που βασίζονται στον γράφο ροής ελέγχου γίνονται κατά κύριο λόγο σε κάθε block ξεχωριστά.

Απαλοιφή κοινών ύπο-εκφράσεων Για την απαλοιφή κοινών ύπο-εκφράσεων σε κάθε block χρησιμοποιείται η μέθοδος *Value Numbering*. Κάθε έκφραση που υπολογίζεται αποθηκεύεται σε μία καινούργια προσωρινή μεταβλητή, έτσι σε περίπτωση που η τιμή αυτής της έκφρασης χρειαστεί σε κάποιον παρακάτω υπολογισμό - μέσα στο ίδιο Block - τότε χρησιμοποιείται η προσωρινή μεταβλητή που είχε δημιουργηθεί προηγουμένως. Για την αντιμετώπιση των πολλών νέων προσωρινών μεταβλητών και των μη απαραίτητων εντολών ανάθεσης χρησιμοποιούνται οι δύο παρακάτω βελτιστοποιήσεις.

Διάδοση αντιγράφων Όπου είναι εφικτό χρησιμοποιείται η αρχική μεταβλητή για την οποία υπολογίστηκε μία έκφραση και όχι η προσωρινή μεταβλητή που δημιουργήθηκε παραπάνω. Με αυτό τον τρόπο δίνεται η δυνατότητα να πραγματοποιήσουμε απαλοιφή νεκρού κώδικα σε τοπικό επίπεδο.

Απαλοιφή νεκρού κώδικα Όσες προσωρινές μεταβλητές δημιουργήθηκαν παραπάνω αλλά δεν χρησιμοποιούνται είτε επειδή αντικαταστάθηκαν κατά τη διάδοση αντιγράφων είτε επειδή δε χρειάστηκε, μπορούν να σβηστούν. Αυτό γίνεται με ένα πέρασμα κάθε Block ανάποδα, δηλαδή ξεκινώντας από την τελευταία εντολή προς την πρώτη, σημειώνοντας ποιες προσωρινές μεταβλητές χρειάζονται και σβήνοντας όσες κατά τη δήλωση τους δεν έχουν σημειωθεί ως χρήσιμες.

Απαλοιφή Απροσπέλαστου Κώδικα Από τις παραπάνω βελτιστοποιήσεις (και όχι μόνο) πιθανώς να προκύψουν σημεία του κώδικα που είναι απροσπέλαστα με βάση τη ροή ελέγχου. Ξεκινώντας από το αρχικό Block και εκτελώντας μία αναζήτηση κατά βάθος (DFS) πάνω στο γράφο ροής ελέγχου μπορούμε να συμπεράνουμε ποια blocks είναι απροσπέλαστα και να τα αφαιρέσουμε.

2.7.3 Tail Recursion Elimination

Στον Alpaca δεν βελτιστοποιούνται όλες οι κλήσεις ουράς αλλά μόνο οι κλήσεις ουράς προς την ίδια συνάρτηση, δηλαδή οι αναδρομικές. Η υλοποίηση της απαλοιφής κλήσης ουράς περιλαμβάνει 2 στάδια:

Εντοπισμός κλήσεων ουράς Ο εντοπισμός των σημείων κλήσης ουράς έγινε πάνω στο AST με βάση τους παρακάτω κανόνες:

$$\tau(let_{fun} = e_1, \_) \rightarrow e_1.tail \leftarrow true; \tau(e_1, true)$$

$$\tau(e_1; e_2, true) \rightarrow \tau(e_1, false); e_2.tail \leftarrow true; \tau(e_2, true)$$

$$\tau(begin; e_1; end, true) \rightarrow e_1.tail \leftarrow true; \tau(e_1, true)$$

$$\tau(if\ e_1\ then\ e_2\ else\ e_3, true) \rightarrow \tau(e_1, false); e_2.tail \leftarrow true; \tau(e_2, true); e_3.tail \leftarrow true; \tau(e_3, true)$$

$$\tau(if\ e_1\ then\ e_2, true) \rightarrow \tau(e_1, false); e_2.tail \leftarrow true; \tau(e_2, true)$$

$$\tau(match\ e_1\ with p_i \rightarrow e_i, true) \rightarrow \tau(e_1, false); e_i.tail \leftarrow true; \tau(e_i, true)$$

$$\tau(let\ x = e_1 in e_2, true) \rightarrow \tau(e_1, false); e_2.tail \leftarrow true; \tau(e_2, true)$$

Η διαδικασία τ υποθέτει ότι το πεδίο $tail$ κάθε έκφρασης είναι $false$. Στη συνέχεια θέτει την τιμή $true$ σε κάθε πεδίο $tail$ με βάση τους παραπάνω κανόνες.

Υλοποίηση κλήση ουράς Η ιδέα είναι ότι μπορούμε να χρησιμοποιήσουμε το υπάρχον εγγράφημα δραστηριοποίησης, αποφεύγοντας το κόστος δημιουργίας (κυρίως σε μνήμη) ενός καινούργιου. Για να γίνει κλήση ουράς αντί για κανονική κλήση αρκεί στις κλήσεις συνάρτησης που έχουν σημειωθεί ως $tail$ και επίσης αναφέρονται στην ίδια συνάρτηση με αυτή που εκτελείται, να αντικαταστήσουμε τις τιμές των παραμέτρων με τις νέες τιμές μέσω ανάθεσης και να κάνουμε $jump$ στην αρχή της συνάρτησης. Η διαδικασία αυτή υλοποιείται στον ενδιάμεσο κώδικα τετράδων και στη συνέχεια μεταφράζεται σε τελικό κώδικα χωρίς κάποια αλλαγή στην παραγωγή τελικού κώδικα.

2.7.4 Βελτιστοποιήσεις τελικού κώδικα

Peephole Οι βελτιστοποιήσεις που έγιναν στον τελικό κώδικα ακολουθούν τη λογική των λεγόμενων Peephole optimizations. Γίνονται πολλαπλά περάσματα στον τελικό κώδικα ψάχνοντας για γνωστά μοτίβα εντολών (1 έως και 3 εντολές) τα οποία στη συνέχεια μετατρέπονται σε πιο απλές και γρήγορες εντολές. Εκμεταλλευόμενοι το ισχυρό pattern-matching της OCaml η υλοποίηση αυτών των βελτιστοποιήσεων είναι αρκετά εύκολη σε σχέση με αυτές του γράφου ροής ελέγχου. Είναι ωστόσο και αρκετά αποτελεσματικές δεδομένου του απλοϊκού αλγορίθμου παραγωγής τελικού κώδικα που ακολουθήσαμε. Στον παρακάτω πίνακα παρουσιάζουμε πολλές από τις βελτιστοποιήσεις που υλοποιήσαμε. Επειδή γίνονται παραπάνω από ένα περάσματα απλοποιούνται και άλλοι συνδυασμοί αυτών.

Generated	Optimized
add target, 1	inc target
sub target, 1	dec target
add/sub target, 0	-
mov r1, 0 sub r1, val	mov r1,val neg r1
mov si, bp push si	push bp
mov r1, imm mov size ptr [r2], r1	mov size ptr [r2], imm
mov mem1, r1 mov r2, mem1	mov mem1, r1 mov r2, r1
mov mem1, r1 mov r1, mem1	mov mem1, r1
mov r1, val op r2, r1	op r2, val
jcond target1 jmp target2	jrevcond target2
mov r1, imm mov r2, size ptr [r3] mov size ptr [r2], r1	mov r2, size ptr [r3] mov size ptr [r2], imm

Πίνακας 3: Peephole Optimizations