Διπλωματική Εργασία

Μελέτη και Αξιοποίηση Τεχνικών Ανάλυσης Μερικής Διαφυγής και Αντικατάστασης Βαθμωτών για Στατική Βελτιστοποίηση στον Μεταγλωττιστή PyPy

Author:
Γεώργιος Παπανικολάου

Supervisors:
Ιωάννης Γαροφαλάκης
Αθανάσιος Νικολακόπουλος

12 Οκτωβρίου 2016

Nanos Gigantum Humeris Insidentes

– Bernard of Chartres

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

Περίληψη

Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Δίπλωμα Μηχανικού

Μελέτη και Αξιοποίηση Τεχνικών Ανάλυσης Μερικής Διαφυγής και Αντικατάστασης Βαθμωτών για Στατική Βελτιστοποίηση στον Μεταγλωττιστή PyPy

Γεώργιος Παπανικολάου

Αυτή η εργασία κατ'αρχάς συνοδεύει την απόπειρα βελτίωσης του υποσυστήματος βελτιστοποίησης του μεταγλωττιστή PyPy μέσω της ανάλυσης διαφυγής. Αποσκοπεύει επίπλεον στην πληροφόρηση του αναγνώστη σχετικά με την περίπλοκη "τέχνη" που ακούει στο όνομα *βελτιστοποίηση κώδικα* και τα προβλήματα που αντιμετωπίζουν οι προγραμματιστές κατά την διαδικασία σχεδιασμού των μεταγλωττιστών και κατά την υλοποίησή τους. Θα δώσουμε λεπτομέρειες σχετικά με γενικά προβλήματα για ανάπτυξη δυναμικών γλωσσών καθώς επίσης και συγκεκριμένα για την Python και το σύστημα PyPy. Έπειτα θα αναλύσουμε τις θεωρητικές λεπτομέρειες για σχεδιασμό μεταγλωττιστών και συγκεκριμένα για στατική ανάλυση κώδικα βάσει γραφημάτων, μερική ανάλυση διαφυγής και αντικατάσταση βαθμωτών. Κύριος σκοπός της εργασίας αυτής είναι η σχεδίαση ένος backend module για το σύστημα PyPy που θα υλοποιεί την μερική ανάλυση διαφυγής. Η δουλεία είναι βασισμένη σε ένα προηγούμενο paper που αποτελεί υλοποίηση και μελέτη για την γλώσσα Java. Το module θα αποτελέσει παράδειγμα για την θεωρία που θα αναλύσουμε αλλά θα προσπαθίσουμε επίσης να το είσαγουμε στο όλο codebase του project έτσι ώστε να συμβάλουμε στην βελτίωση του μεταγλωττιστή. Τέλος η εργασία θα περιλαμβάνει φυσικά μετρήσεις και benchmarks.

UNIVERSITY OF PATRAS

Abstract

Computer Engineer & informatics Department

Engineer's Degree

Study and Implementation of Partial Escape Analysis and Scalar Replacement Methods for Static Optimization in the PyPy Compiler Framework

Georgios Papanikolaou

This document, first and foremost, is the companion of an attempt to improve the escape analysis optimization of the PyPy interpreter. However, it also aims to shine light at the peculiar craft of compiler optimization and will try to inform the reader of the nuisances and problems that the engineers face throughout the designing process. We will elaborate on general problems based on dynamic language design, as well as problems that we experienced specifically with Python and with the PyPy framework. Furthermore we will expand on the details of compiler optimization with intricate details on the static analysis of graphs, partial escape analysis and scalar replacement. The main goal is the design and the implementation of a backend optimization module for PyPy that performs partial escape analysis. It is based on a previous treatise of the same subject – an implementation for Java. It will serve as an example of the said theory, and we will also try to fully integrate it into the whole PyPy project, in order to improve the overall speed of the interpreter. Last, but not least, we will accompany our implementation with benchmark and measurements.

Acknowledgements – Ευχαριστίες

I'd like to thank Carl Friedrich Bolz for his valuable and eye-opening help with the theory that this text entails and his aid with the debugging, my official supervisors for their help, acceptance and tolerance of my quirks and last but not least my parents for their financial and moral help. Thank you guys.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Γενικά

Σε αυτό το κεφάλαιο αποσκοπούμε να ενημερώσουμε τον αναγνώστη γενικά περί δυναμικών γλωσσών προγραμματισμού και πιο συγκεκριμένα για την διαδικασία μεταγλώττισης τέτοιων γλωσσών και τα προβλήματα που αντιμετωπίζει κανείς. Θεωρούμε ότι ο αναγνώστης ήδη κατέχει μια σχετικά καλή ιδέα για προγραμματισμό για τις κάποιες από τις λεπτομέρειες που διέπουν τους μεταφραστές τους.

1.2 Δυναμικές Γλώσσες

Ο όρος είναι λίγο ασαφής αλλά γενικά ως δυναμική γλώσσα εννοούμε μια γλώσσα προγραμματισμού πολύ υψηλού επιπέδου, που παρουσιάζει συμπεριφορές υψηλής αφαιρετικότητας κατά της εκτέλεση του προγράμματος, σε αντίθεση με άλλες γλώσσες στις οποίες αυτό λαμβάνει χώρα κατά την μετάφραση του κώδικα σε κώδικα μηχανής. Συνήθως τα προγράμματα αυτών των γλωσσών δεν μεταφράζονται απευθείας, αλλά ένα ειδικό πρόγραμμα – το οποίο καλείται μεταγλωττιστής (interpreter) αναλαμβάνει να τα “τρέξει”, με την όλη διαδικασία της μετατροπής (του υψηλού επιπέδου κώδικα σε κώδικα μηχανής) να λαμβάνει χώρα κατά το runtime· δηλαδή κατά την διάρκεια που ο χρήστης τρέχει το πρόγραμμα και όχι κατά την μεταγλώττιση όπως συμβαίνει με άλλες γλώσσες εξίσου υψηλού επιπέδου (βλ. Rust).

Τα χαρακτηριστικά και οι υποκατηγορίες των δυναμικών γλωσσών βρίθουν και η ολοκληρωμένη λεπτομερής απαρίθμησή τους είναι εκτός των σκοπών αυτής της εργασίας. Το σημαντικότερο κοινό χαρακτηριστικό είναι η χρήση του μεταγλωττιστή και το “τρέξιμο” του προγράμματος στο περιβάλλον που δημιουργεί αυτό. Η λειτουργία δηλαδή αυτή είναι σαν μια εικονική μηχανή και αυτό μας δίνει απευθείας την δυνατότητα για ένα ακόμα επίπεδο αφαιρετικότητας στα design patterns του προγραμματισμού μας. Έτσι έχουμε πράγματα όπως metaprogramming και φυσικά δυναμικούς τύπους.

Οι δυναμικοί τύποι είναι το σημαντικότερο χαρακτηριστικό από την μεριά του χρήστη, φυσικά για την ευκολία που δίνει σε αυτόν η εκάστοτε γλώσσα. Ο χρήστης δεν χρειάζεται να δηλώσει ρητά τον τύπο μιας μεταβλητής. Αυτος συμπεραίνεται από την αρχικοποίηση ή τα “συμφραζόμενα” της μεταβλητής μέσα στο πρόγραμμα. Επίσης

σημαντικό είναι ότι σε μερικές από αυτές τις γλώσσες μπορεί να αλλάξει κατά την διάρκεια της εκτέλεσης.

Άλλο ένα τέτοιο σύγχρονο χαρακτηριστικό υψηλού επιπέδου είναι το just-in-time compilation αλλά δεν θα ασχοληθούμε καθόλου με αυτό.

Οι δυναμικές γλώσσες ήταν πάντα δημοφιλείς, αλλά στις μέρες μας οι καινούργιες συνθήκες, η ολοένα αυξημένη υπολογιστική ισχύ, και οι μεγάλες ομάδες (με πληθώρα αναγκών) πίσω από τον σχεδιασμό των γλωσσών, έχουν οδηγήσει σε νέες πτυχές στον κόσμο των γλωσσών και του προγραμματισμού. Πολύ συχνά "αναδύονται" καινούργια χαρακτηριστικά για συγκεκριμένες ανάγκες ή κάποιο είδος αφαίρετικότητας τα καταστεί πιο εύκολα. Η ώθηση αυτή, που διέπει αυτά τα communities των δυναμικών γλωσσών, είναι μια ισορροπία μεταξύ πρακτικότητας και κομψότητας. Τα χαρακτηριστικά των γλωσσών αυτών τείνουν να μεγαλώνουν (με εξαίρεση την Python), στην οποία ακόμα διατηρείται ένα μινιμαλιστικό mindset. Σχεδόν σε όλες, αντί για κάποιο καινούργιο abstraction (το οποίο θα προερχόταν ή θα οδηγούσε σε κάποια καινοτομία), προτιμάται ένα μεγάλο πλήθος μικρών μικρών βελτιωτικών χαρακτηριστικών, καθώς στοχεύουν να είναι εύκολες στην καθημερινή χρήση και από τον πιο ανειδίκευτο προγραμματιστή. Θα μπορούσε κανείς να πει, ότι αυτές οι γλώσσες είναι περισσότερο βιβλιοθήκες (libraries) πάνω σε μια απλή γλώσσα (core language). Τέλος, είναι προφανές, ότι το λιγότερο σημαντικό χαρακτηριστικό για αυτές τις γλώσσες είναι οι επιδόσεις. Πολλές φορές γίνονται επιλογές (κατά τον σχεδιασμό τους) υπέρ της ευκολίας χρήσης αντί των επιδόσεων. Όμως ακόμα και σε άλλη περίπτωση, λόγω του μεγάλου αριθμού constructions στις γλώσσες, η στατική ανάλυση, το inference και η βελτιστοποίηση έχουν καταστεί εξαιρετικά δύσκολες.

1.2.1 Python

Γενικά - Ιστορία

Συγκεκριμένα η γλώσσα, με την οποία θα ασχοληθούμε και στην οποία θα υλοποιήσουμε το module, είναι η *Python*[15]. Η Python είναι μια γενικού σκοπού δυναμική, "πολύ-παραδειγματική", υψηλού επιπέδου γλώσσα η οποία είναι εξαιρετικά δημοφιλής εδώ και πολλά χρόνια. Η φιλοσοφία της δίνει βάση στην καλή αναγνωσιμότητα του κώδικάς της και στην ευκολία της χρήσης. Υπάρχουν υλοποιήσεις σχεδόν σε όλες τις πλατφόρμες[1] και πολλές διαφορετικές εκδόσεις[2] της και πειράματα[3]. Επιπλέον λόγω της δημοτικότητάς της έχει επηρεάσει πολλές άλλες γλώσσες όλων των ειδών και έχει ουσιαστικά συμβάλει στην σημερινή εικόνα του κόσμου των υπολογιστών.

Βασίζεται στην φιλοσοφία των πρωταρχικών ώριμων γλωσσών (όπως C, Java), δηλαδή το κυρίως *προγραμματιστικό παράδειγμά* της είναι ο "Προστακτικός- Διαδικαστικός Προγραμματισμός" (imperative/declarative programming) με πολλά στοιχεία – και ένα καλό σύστημα αντικειμένων (object-oriented programming). Επίσης λέμε ότι "ξεχωρίζει" τις έννοιες δεδομένων και κώδικα – δηλαδή ακολουθεί το παράδειγμα

[1] π.χ. σε C, C#, Java κλπ, με την πιο δημοφιλής (και το reference για τις άλλες) να είναι η λεγόμενη CPython σε C

[2] βλ. Stackless Python

[3] βλ. PyPy

της C και όχι της lisp. Έχει παρόλα αυτά και πολλές επιρροές και χαρακτηριστικά από συναρτησιακό προγραμματισμό (functional programming).

Σε αντίθεση με άλλες παρόμοιες γλώσσες, η Python προτιμά τον μινιμαλισμό. Οι τελικές – σχετικά αυστηρές – αποφάσεις στον σχεδιασμό της λαμβάνονται από τον Guido van Rossum, ο οποίος έχει περιπαικτικά τον τίτλο του "Benevolent Dictator For Life".[9]

Χαρακτηριστικά - Ιδιαιτερότητες

Από την άλλη, όπως οι περισσότερες γλώσσες, βασικό χαρακτηριστικό που τις διέπει είναι η παντελής έλλειψη δηλώσεων (*no declaration notion*). Κάθε πρόγραμμα χτίζεται με εντολές. Υπάρχουν κάποιες εκφράσεις που μπορεί να μοιάζουν με δηλώσεις, όπως οι "δήλωση" συνάρτησης. Στην πραγματικότητα δεν είναι, και απλώς δημιουργείται ένα αντικείμενο (runtime object) που δρα ως συνάρτηση. Όμοιως και στην περίπτωση των κλάσεων, και των module[4], τα οποία αποτελούν βασικό κομμάτι του οικοσυστήματος της γλώσσας. Πράγματι, *τα πάντα* είναι ένα runtime αντικείμενο για την Python. Άπαξ και δημιουργηθεί ένα αντικείμενο, μια αναφορά (reference) σε αυτό, θα αποθηκευτεί στην τοπική λίστα ονομάτων (*namespace*), η οποία στην Python λέγεται *module*. Αυτή είναι και η "μονάδα" του προγράμματος στην Python. Το κατώτερο module στην ιεραρχία λέγεται φυσικά main module και είναι η αντίστοιχη main συνάρτηση του κάθε προγράμματος.

Άλλα χαρακτηριστικά της γλώσσας περιλαμβάνουν εξαιρετικά καλό σύστημα γεννητόρων (*generators*) και επαναληπτών (*iterators*) που διευκολύνουν σε μεγάλο βαθμό κάποιες συγκεκριμένες περιπτώσεις, σύστημα metaprogramming βασισμένο σε μετακλάσεις, σύστημα σχολίων χτισμένο μέσα στη γλώσσα (docstrings), κ.α.

Ακολουθεί ένα μικρό παράδειγμα για μια πρώτη γεύση με την γλώσσα. Είναι κομμάτι της build-in βιβλιοθήκης και υλοποιεί έναν μετρητή με την μορφή iterator. Σημαντική είναι η χρήση whitespace για το identation. Αυτό οδηγεί σε κώδικα με ακόμα μεγαλύτερη αναγνωσιμότητα.

```
1 def count(n=0):
2     while True:
3         yield n
4         n += 1
```

example.py

1.3 Μεταγλώττιση Δυναμικών Γλωσσών

Ο λεγόμενος μεταγλωττιστής (interpreter) είναι ο αντίστοιχος του μεταφραστής (compiler) στις "στατικές" γλώσσες. Αντί να μεταφράζει εξ ολοκλήρου το πρόγραμμα σε κώδικα μηχανής (και να εκτελεί όλες τις διαδικασίες για τις οποίες είναι προγραμματισμένος) κατά την διάρκεια του compile-time (την ώρα που τρέχει ο compiler για

[4] βλ. `import` statement

να “παράγει” το πρόγραμμα), το κάνει όταν αυτό χρειαστεί - και αν χρειαστεί - κατά την διάρκεια του runtime. Όπως είδαμε στην προηγούμενη ενότητα, δεν υπάρχει μια σταθερή δομή δηλώσεων (η οποία να μπορεί να αναλυθεί), οπότε η μεταγλώττιση και ειδικά η βελτιστοποίηση είναι πολύ δύσκολες. Θεωρητικά η γλώσσα θα μπορούσε να δημιουργήσει μια κλάση με εκατοντάδες τελείως διαφορετικούς τρόπους βάσει αποτελεσμάτων από NP-complete υπολογισμούς και εξωτερικούς παράγοντες.[1]

1.3.1 PyPy

Τι είναι το PyPy

Το framework που θα χρησιμοποιήσουμε λέγεται PyPy[14]. Το PyPy ξεκίνησε το 2007 ως μια απόπειρα για έναν interpreter της Python γραμμένο στην ίδια την γλώσσα[5]. Από τότε έχει εξελιχθεί σε ένα ολοκληρωμένο *framework* με πολλές μοντέρνες δυνατότητες όπως *JIT compilation*. Αποτελείται από 2 μεγάλα subprojects:

- **RPython framework**

  Το πρώτο είναι ένα compiler framework. Είναι γραμμένο σε κανονική Python, και ουσιαστικά είναι μια βάση – ένα πρόγραμμα το οποίο μπορεί να παράγει compilers. Μπορεί να “διαβάσει” μια ειδική περιορισμένη έκδοση της Python (που λέγεται RPython). Οι διαφορές που έχει είναι ότι στερείται κάποιων κατασκευών υψηλής αφαιρετικότητας. Έχει παρόλα αυτά πολλά από τα γνωστά χαρακτηριστικά της Python.

  Σε αυτό το framework έχουν γραφτεί πλέον μεταφραστές και σε άλλες γλώσσες πέραν μόνο της αρχικής Python, που “περιλαμβάνεται” στο project. Το πιο γνωστό παράδειγμα είναι το Topaz[19].

- **PyPy**

  Το δεύτερο πρότζεκτ είναι το PyPy. Ένας compiler της Python γραμμένος βάσει του προηγούμενου framework, σε RPython. Έχει πλέον ξεπεράσει σε ταχύτητα και αποδοτικότητα τον CPython, τον reference compiler της γλώσσας. Οι λόγοι που δεν ανακηρύσσεται αυτός reference compiler είναι “διοικητικοί”[6]. Βέβαια υπάρχουν και μερικά προβλήματα με παλιό κώδικα (legacy code) και library support. Οι λόγοι μάλλον που ο compiler δεν είναι drop-in replacement βέβαια, είναι λόγοι εμπιστοσύνης.

Φυσικά όπως όλα τα μεγάλα project έχει modular δομή κώδικα για πιο εύκολη διαχείριση του τεράστιου πλέον όγκου του. Εμείς θα υλοποιήσουμε ένα τέτοιο module στο υποσύστημα του backend optimization.

[5] βλ. bootstrapping
[6] βλ. Guido van Rossum

Ανάλυση προγραμμάτων

Το PyPy, όπως και όλοι οι μεταγλωττιστές δυναμικών γλωσσών, αναλύει το κάθε πρόγραμμα "ζωντανά" (live program analysis), δηλαδή στην μνήμη και όχι ως "νεκρά"[7] αρχεία. Αυτό σημαίνει ότι το πρόγραμμα φορτώνεται (ίσως και ολόκληρο) αρχικά στην μνήμη και προχωρά η διαδικασία της μεταγλώττισης. Όταν φτάσει σε ένα αρκετά καλό σημείο, θα μειωθεί η "δυναμικότητά" του και θα αναλυθούν τα αντικείμενα που έχουν προκύψει στην μνήμη. Ουστιαστικά η RPython είναι το υποσύνολο της Python που περιλαμβάνει μόνο όσα χαρακτηριστικά υποστηρίζονται από αυτό το σύστημα ανάλυσης.

Στην περίπτωση που η ανάλυση γινόταν βάσει των στατικών αρχείων, τότε ουσιαστικά θα ήταν σαν να "ακυρώναμε" την έννοια της δυναμικότητας της γλώσσας. Από την άλλη, και η ανάλυση μιας *σταθερής* εικόνας του προγράμματος στην μνήμη, θα ήταν το ίδιο. Για αυτό και το σύστημα του PyPy είναι εξαιρετικά δυναμικό. Μπορεί να χειριστεί ακόμα και τελείως δυναμικά κομμάτια κώδικα εφόσον η είσοδος της ροής σε αυτά είναι οριοθετημένη (bounded)[1]. Είναι σημαντικό να ξανά-αναφέρουμε ότι και το ίδιο το PyPy είναι γραμμένο σε RPython, και αυτό σημαίνει ότι και το ίδιο υφίσταται την ίδια δυναμική ανάλυση και απολαμβάνει τα ίδια πλεονεκτήματα αυτής.

[7] στατικά

Κεφάλαιο 2

Βελτιστοποίηση δυναμικού κώδικα

2.1 Γενικά

Η βελτιστοποίηση είναι η διαδικασία κατά την οποία ένα κομμάτι κώδικα τροποποιείται έτσι ώστε να καταστεί πιο αποτελεσματικό - είτε από άποψη χώρου είτε από άποψη χρόνου - χωρίς να αλλάξουν τα αποτελέσματα που δίνει ή τα side- effects που προκαλεί. Ο χρήστης δεν θα πρέπει να αντιληφθεί την αλλαγή και απλά να δει το πρόγραμμα του να τρέχει πιο γρήγορα και/ή να απαιτεί μικρότερη μνήμη. Η βελτιστοποίηση κώδικα είναι το κεντρικό θέμα αυτής της εργασίας. Πάνω σε αυτό το θέμα γίνεται σήμερα το μεγαλύτερο κομμάτι ακαδημαϊκής έρευνας από τα πανεπιστήμια του κόσμου, καθώς τα υπόλοιπα κομμάτια ενός μεταφραστή/μεταγλωττιστή θεωρούνται τετριμμένα. Η θεωρία των parsers και των semitics analyzers για παράδειγμα έχει φτάσει σε ένα τυπικό σημείο κορεσμού, και η αντίληψη που έχουμε για αυτά είναι σχεδόν ολοκληρωμένη. Από την άλλη η βελτιστοποίηση κώδικα, παρά το γεγονός ότι μελετάται από τις απαρχές της εποχής των ηλεκτρονικών υπολογιστών, είναι ακόμα σε πρώιμα στάδια. Είναι δεδομένο ότι οι compilers για τις ”ώριμες” γλώσσες θα παράγουν σωστό κώδικα αλλά θα κριθούν σε σχέση με τις άλλες για το πόσο καλό βελτιστοποιημένο κώδικα παράγουν.

Δηλαδή ο κάθε οργανισμός/πανεπιστήμιο/μεταγλωττιστής εκτελεί την βελτιστοποίηση διαφορετικά. Αυτό γιατί τα περισσότερα προβλήματα βελτιστοποίησης είναι NP- complete οπότε η πλειοψηφία των αλγόριθμων θα πρέπει να βασιστεί σε ευρετικά και σε προσεγγίσεις. Είναι πολλές φορές δυνατόν ο ένας αλγόριθμος να παράγει σωστό και γρήγορο κώδικα σε μια περίπτωση προβλήματος ενώ σε μια παρόμοια, όχι απλά να μην υπάρχει βελτίωση, αλλά να υπάρχει και απότομη αύξηση χρόνου και/ή μνήμης. Παρόλα αυτά οι περισσότεροι αλγόριθμοι τείνουν να δουλεύουν καλά για την πλειοψηφία των προβλημάτων.

Φυσικά η καλύτερη - και έξυπνη - βελτιστοποίηση θα έρθει από την μεριά του προγραμματιστή. Κανένας αλγόριθμος δεν είναι τόσο εξύπνος ακόμα ώστε να αντικαταστήσει τον άνθρωπο π.χ. στην επιλογή του κατάλληλου αλγορίθμου για ταξινόμηση. Με άλλα λόγια, όσων αφορά την ανάλυση πολυπλοκότητας σε big-O (τον καλύτερο ”κριτή” αλγορίθμων που έχουμε), οι αλγόριθμοι βελτιστοποίησης μπορούν να κάνουν αλλαγές μόνο σε τάξη μεγέθους σταθεράς. Βέβαια αν ο αλγόριθμος είναι ιδανικός από την πλευρά του ”έξυπνου” χρήστη, τότε τέτοιες μικρές βελτιώσεις μπορούν να κάνουν την διαφορά.

Από την άλλη ο προγραμματιστής θα πρέπει ιδανικά να περιορίζεται στις έξυπνες επιλογές αλγορίθμου κλπ. και να μην επιχειρεί να βελτιώσει το πρόγραμμα πρόωρα· κατά την πρώτη δηλαδή φάση της σχεδίασης και υλοποίησης, γιατί όποια τυχόν βελτιστοποίηση από μέρους του μεταφραστή θα γίνει βάσει ιδιοματικής σύνταξης της γλώσσας. Με άλλα λόγια, από ένα απλό loop αρχικοποίσης θα παραχθεί πολύ πιο ποιοτικός κώδικας μηχανής εκ μέρους του μεταφραστή, σε σχέση με κάποιο απόκρυφα δύσκολο τρόπο αρχικοποίησης, που ο προγραμματιστής επέλεξε επειδή θεώρησε ότι θα είναι πιο γρήγορος. Στα λόγια του Donald Knuth: *"Premature optimization is the root of all evil in programming"*.[13]

Η βελτιστοποίηση κώδικα δε θα πρέπει φυσικά σε καμία περίπτωση να πειράζει την ορθότητα του προγράμματος. Θα πρέπει το παραγόμενο βελτιστοποιημένο πρόγραμμα να αποδίδει τα ίδια αποτελέσματα στον τελικό χρήστη και να παράγει τις ίδιες "παρενέργειες" (side-effects) σε όλες τις περιπτώσεις και για όλες τις εισόδους. Σαφώς το πρόγραμμα θα πρέπει να είναι σωστό εκ μέρους του προγραμματιστή για να μπορεί ο μεταφραστής να εγγυηθεί το παραπάνω.

Σημαντικός είναι ο καθορισμός του πότε είναι δυνατός και πότε απαιτείται η βελτιστοποίηση. Φυσικά τον τελευταίο λόγο και σε αυτή την περίπτωση τον έχει ο προγραμματιστής. Όλοι οι μοντέρνοι μεταφραστές έχουν την επιλογή πλήρους απενεργοποίησης των διαδικασιών βελτιστοποιήσης ενώ πολλοί προσφέρουν και δυνατότητες επιλογής συγκεκριμένων διαδικασιών (π.χ. μόνο αντικατάσταση βαθμωτών).

Αξίζει να αναφέρουμε ότι η βελτιστοποίσης βασίζεται σε πολύ μεγάλο βαθμό στην αρχιτεκτονική του κάθε υπολογιστή και το αποτέλεσμα μπορεί να είναι δραστικά διαφορετικό όταν μεταφράσουμε για κάποια άλλη αρχιτεκτονική. Οι ώριμοι μεταφραστές είναι ικανοί να μεταφράσουν για όλες τις σύγχρονες αρχιτεκτονικές.

Χονδρικά η διαδικασία μεταγλώττισης έχει ως εξής: λεξική ανάλυση (lexical analysis), συντακτική ανάλυση (syntactic analysis), σημαντική ανάλυση (semantics) όπου αντιστοιχίζονται τα τυχόν token με "ιδιότητες" της γλώσσας (μέχρι εδώ αντιλαμβάνεται όλα τα συντακτικά λάθη), και έπειτα παραγωγή intermediate κώδικα. Σε αυτό το σημείο ο κώδικας αυτός θα υποστεί ανάλυση (βλ. παρακάτω) και θα βελτιωθεί.

2.2 Τεχνικές Βελτιστοποίσης

Οι περισσότερες τεχνικές βελτιστοποίησης βασίζονται σε ανάλυση του *"control flow"* (ροή) του προγράμματος κατά το runtime (Control flow analysis) Μέχρι και πριν από το σημείο της ανάλυσης αυτής ο μεταφραστής έχει μόνο τυπικές πληροφορίες για το τι κάνει το πρόγραμμα που μεταφράζει. Εδώ βρίσκει περισσότερες "σημαντικές" πληροφορίες για την φύση του προγράμματος με το να αναλύει την ροή του (δηλαδή τα forks λόγω if, τα loops κ.α.). Φυσικά ποτέ δεν έχει πλήρη επίγνωση.

Η ανάλυση αυτή γίνεται με το να κατασκευάζεται το *γράφημα ροής* (control flow graph), το οποίο καταγράφει όλη την ροή δηλαδή όλα τα πιθανά "μονοπάτια" που είναι δυνατόν να ακολουθήσει το πρόγραμμα. Το βασικό στοιχείο είναι η *συνάρτηση*. Δημιουργείται ένα γράφημα δηλαδή για την κάθε συνάρτηση με ένα σημείο εισόδου (entry point), στην αρχή φυσικά της συνάρτησης και ένα ή περισσότερα σημεία εξόδου

ανάλογα με τις ανάγκες. Για την παραγωγή του κάθε γραφήματος ο κώδικας χωρίζεται σε κομμάτια (*basic blocks*), στα οποία η ροή του προγράμματος ξεκινάει μόνο από ένα σημείο στην αρχή και τελειώνει σε ένα και μοναδικό σημείο στο τέλος. Τα τυχόν παρακλάδια (branches ή forks) του προγράμματος οργανώνονται βάσει τον blocks αυτών. Αυτό σημαίνει ότι δεν μπορούν να υπάρξουν branches στη μέση των blocks, οπότε όλες οι δηλώσεις και οι εντολές (*statements*) θα πρέπει να τρέξουν διαδοχικά. Τα branches όπως είπαμε βρίσκονται έξω από τα blocks και οδηγούν στα επόμενα σύμφωνα με την ροή του προγράμματός μας. Από το σημείο του γραφήματος, μόνο το πρώτο statement είναι "ορατό" και αν η ροή πάει στο block, θα πρέπει αυτό να τρέξει ολόκληρο.

Ένα block μπορεί να ξεκινάει είτε με το entry point ενός branch είτε να είναι το target ενός branch, και να τελειώνει είτε με μια οδηγία άλματος (jump statement) είτε με οδηγία συνθήκης (conditional statement· δηλαδή if) είτε τέλος με οδηγία "επιστροφής" (return statement), η οποία τερματίζει ολόκληρο το graph της εκάστοτε συνάρτησης.

Οι μέθοδοι βελτιστοποίσης χωρίζονται σε τοπικές (local) και καθολικές (global). Οι πρώτες δουλεύουν αποκλειστικά αλλάζοντας statements κ.α. μέσα στα blocks και οι δεύτερες δουλεύουν στις σχέσεις μεταξύ των blocks. Φυσικά πολύ πιο απλές στον σχεδιασμό και στην υλοποίηση είναι οι πρώτες. Παρακάτω δίνονται κάποια παραδείγματα. (Πολλές από τις τοπικές που θα συζητήσουμε πρώτα έχουν αντίστοιχες καθολίκες που βασίζονται στην ίδια αρχή.) Επίσης όπως θα δούμε υπάρχουν οι λεγόμενες βελτιστοποιήσεις μηχανής.

2.2.1 Τοπικές Μέθοδοι Βελτιστοποίησης

- Αναδίπλωση Σταθερών

  Με τον όρο "Αναδίπλωση Σταθερών" (*Constant folding*) αναφερόμαστε στον εντοπισμό των τελεστέων κάποιων δηλώσεων/διαδικασιών/οδηγιών που μπορούν να αποτελέσουν ή ήδη αποτελούν σταθερές. Στην πιο απλή μορφή της θα αντικαταστήσει αριθμιτικές πράξεις, π.χ.: Ένα statement όπως: $a = 4 * 2 + 3$ θα μετατραπεί σε $a = 11$, αποφεύγοντας έτσι τις πράξεις αύτες κατά το runtime. Η απόδειξη ότι αυτή η αντικατάσταση μπορεί να γίνει διατηρώντας την ορθότητα είναι προφανής, αφού η πράξη αυτή απλώς πρέπει να εκτελεστεί μια φορά. Άπαξ και την εκτελέσει ο μεταφραστής, η ορθότητα διατηρείται.

- Αναδίπλωση Επαναλήψεων

  Σε πολλές περιπτώσεις – ειδικά αν ο μεταγλωττιστής αναγνωρίσει ότι πρόκειται για επανάληψη με συγκεκριμένο σκοπό π.χ. αρχικοποίηση πίνακα – η επανάληψη μπορεί να εξαλειφθεί. Στην θέση της θα εισαχθεί ίσως κάποιο ειδικό construction του compiler ή της γλώσσας για αρχικοποίηση σε $O(1)$ (ή έστω κάτι πιο γρήγορο από το αρχικό).

- Αναδιάδοση Σταθερών

Σε περίπτωση που ένα statement που έχει μορφή σταθεράς (είτε όπως προηγουμένως μια σειρά από πράξεις είτε απλώς ένας αριθμός είτε τέλος μια συμβολοσειρά) ανατεθεί σε μια μεταβλητή, τότε ο μεταφραστής μπορεί να την αντικαταστήσει με την ίδια την σταθερά. Αυτό μπορεί να συμβεί μόνο στην περίπτωση που ενδιάμεσα στις χρήσεις της μεταβλητής δεν υπάρχει άλλη ανάθεση (δηλαδή αλλαγή των περιεχομένων της μεταβλητής). Μπορεί αρχικά να φαίνεται ότι η βελτίωση θα είναι μικρή αφού σώνουμε έναν μικρό αριθμό προσπελάσεων μνήμης αλλά σε κάποιες αρχιτεκτονικές (π.χ. RISC) υπάρχουν τεράστια αποτελέσματα και η μέθοδος αυτή είναι εξαιρετικά αποτελεσματική καθώς μειώνεται και ο αριθμός των καταχωρητών (registers) που χρησιμοποιούνται στην τελική έκδοση του προγράμματος σε κώδικα μηχανής. Αυτό συμβαίνει γιατί στην περίπτωση της MISC το σύστημα διευθυνσιοδότησης δουλεύει με πρόσθεση ένος register και ενός σταθερού offset.

- Αναδιάδοση

  Ομοίως με παραπάνω μπορεί να γίνει κάτι παρόμοιο στην περίπτωση απλώς μεταβλητών.

- Αλγεβρική Απλοποίηση

  Ο μεταφραστής μπορεί να χρησιμοποιήσει συγκεκριμένους αλγεβρικούς συνδιασμούς (τους οποίους γνωρίζει *a-priori*) και ξέρει ότι λειτουργούν και τρέχουν πιο γρήγορα από άλλους. Επίσης είναι δυνατόν να χρησιμοποιηθούν αλγεβρικές ιδιότητες για να απλοποιηθούν οι παραστάσεις και να χρειάζονται λιγότερες πράξεις, λ.χ:

  $$(x + y)^2 = x^2 + 2xy + y^2$$

  Προφανώς το αριστερό μέλος θέλει πολύ λιγότερες πράξεις από το δεξί. Τέλος πολλές πράξεις μπορούν να αφαιρεθούν αν κριθούν μη-απαραίτητες. Λαμπρό παράδειγμα η πρόσθεση με το μηδέν κλπ.

- Ενίσχυση Δύναμης Τελεστή

  Κατά την εφαρμογή της μεθόδου "Ενίσχυσης Δύναμης Τελεστή" ο μεταφραστής αποπειράται να αντικαταστήσει συγκεκριμένους τελεστές με άλλους λιγότερο "ακριβούς". Φυσικά αυτό έχει να κάνει με την αρχιτεκτονική. Λ.χ σε κάποια μπορεί η έκφραση $x * 2$ να είναι πιο ακριβή από την $2 * x$.

- Διαγραφή "Νεκρού" Κώδικα

  Στην περίπτωση που ο μεταφραστής αντιληφθεί ότι κάποια σημεία του γραφήματος ροής δεν μπορούν να προσπελαστούν σε καμιά περίπτωση και με καμία είσοδο, τότε το απροσπέλαστο αυτό block καλείται *νεκρός κώδικας* και μπορεί να διαγραφεί με ασφάλεια.

- Εξάλειψη Εκφράσεων Κοινών Αποτελεσμάτων

  Όταν ο μεταφραστής εντοπίσει δύο εκφράσεις που παράγουν το ίδιο αποτέλεσμα τότε μπορεί με ασφάλεια να αφαιρέσει την μια και να εισάγει ένα reference προς την άλλη.

2.2.2 Καθολικές Μέθοδοι Βελτιστοποίησης

Μέθοδοι που αλλάζουν statements και συνδέσεις (links) μεταξύ των blocks.

- Μετακίνηση Κώδικα (Code Motion)

  Σε κάποιες περιπτώσεις ο μεταφραστής εντοπίζει κομμάτια κώδικα μέσα σε διάφορα και/ή ξεχωριστά block, που είτε επαναλαμβάνονται είτε παράγουν το ίδιο αποτέλεσμα. Τότε αυτά μπορούν να μετακινηθούν σε ένα σημείο και να συμβούν μια φορά. Σημαντικότερο παράδειγμα, κώδικας μέσα σε loop που απαιτείται να τρέξει μόνο μια φορά και όχι σε κάθε επανάληψη (iteration). [Λέγεται επίσης και code hoisting.]

2.2.3 Μέθοδοι Βελτιστοποίησης Μηχανής

- Δέσμευση Καταχωρητών

  Ίσως η πιο σημαντική μέθοδος βελτιστοποίησης είναι η σωστή δέσμευση των καταχωρητών (registers) του επεξεργαστή. Οι καταχωρητές είναι η πιο γρήγορη μορφή μνήμης αφού βρίσκονται πολύ κοντά στον επεξεργαστή. Για αυτό όμως είναι και λίγοι και σπανίζουν. Οι πιο αποτελεσματικές μέθοδοι πρέπει να προσδιορίσουν ποιές μεταβλητές και πότε θα βρίσκονται στους καταχωρητές για να ελαχιστοποιήσουν τα memory accesses, τα conflicts και τα races. Δηλαδή να ελαχιστοποιήσουν την κίνηση δεδομένων από και προς τον επεξεργαστή. Ένας πολύ αποτελεσματικός αλγόριθμος είναι ο λεγόμενος "χρωματισμός" (register coloring). Εκτός από τους καταχωρητές ο εκάστοτε αλγόριθμος θα πρέπει να οργανώσει φυσικά και όλη την ιεραρχία μνήμης, λαμβάνοντας υπόψιν του και τα κρυφά επίπεδα μνήμης της κάθε αρχιτεκτονικής (cache).

- Χρονολόγηση Εντολών (Instruction Scheduling)

  Άλλη μία σημαντική μέθοδος. Ο μεταφραστής καλείται να ανακαλύψει την ιδανικότερη σειρά με την οποία θα οργανωθούν οι εντολές στον χρόνο, έχοντας υπόψιν του τις ιδιορρυθμίες και τα ειδικά χαρακτηριστικά του κάθε επεξεργαστή και αρχιτεκτονικής. Οι λεπτομέρειες περιλαμβάνουν: ικανότητες pipelining, πλήθος διαθέσιμων εντολών (RISC/MISC), την χρήση big ή small endian κ.α.

- Βελτιστοποιήσεις "Κλειδαρότρυπας" (Peephole)

  Εδώ περιλαμβάνονται μέθοδοι που σχετίζονται με την εκάστοτε μηχανή. Σε κάθε επανάληψη εξετάζονται μερικές από τις επόμενες εντολές (εξού και το όνομα textitκλειδαρότρυπα) και βελτιώνονται αντίστοιχα. Παράδειγμα: η αποφυγή φόρτωσης δεδομένων όταν η προηγούμενη εντολή φορτώνει τα δεδομένα αυτόματα.

Στο επόμενο κεφάλαιο αναλύεται αρχικά η διακασία μεταγλώττισης του pypy (για λόγους κατανόησης), και έπειτα πιο λεπτομερώς η μέθοδος που θα χρησιμοποιήσουμε στην υλοποίηση του pypy module.

Κεφάλαιο 3

Διαδικασία Μεταγλώττισης στο RPython

Το κεφάλαιο αυτό θα αναλύσει ότι χρειάζεται να ξέρει ο αναγνώστης για να κατανοήσει το κεφάλαιο της υλοποίησης. Θα περιγράψουμε εκτενέστερα την διαδικασία που ακολουθείται στο σύστημα RPython για την μεταγλώττιση (π.χ. του PyPy), έτσι ώστε να ξέρει ο αναγνώστης σε ποιο "χρονικό" σημείο θα λάβει χώρα το εγχείρημά μας. Στο επόμενο κεφάλαιο θα παραθέσουμε με λεπτομέρεια την θεωρία της μεθόδου που θα υλοποιήσουμε.

3.1 Επισκόπηση

Ο σκοπός του συστήματος και του *toolchain* του RPython είναι να μεταφράσει προγράμματα (κυρίως μεταφραστές) γραμμένα σε RPython με αποτελεσματικότητα ανεξάρτητα με το ποια πλατφόρμα είναι το target του. Το default target είναι το *C backend* (παραγωγός κώδικα C) και σε αυτό θα αναφερόμαστε όταν δίνουμε παραδείγματα.

Το σύστημα στην πραγματικότητα δεν βλέπει ποτέ κανονικό κώδικα Python ή δέντρα, αλλά ξεκινά με *αντικείμενα κώδικα* (code objects) βασισμένα στις συναρτήσεις που έχουν δοθεί ως είσοδος και τα μετατρέπει με *abstract interpretation*[3][5] σε γραφήματα ροής. Η βασική δηλαδή δομή δεδομένων (unit), πάνω στην οποία δουλεύει το σύστημα επομένως, είναι τα γραφήματα ροής (σε SSA[4] form) βασισμένα σε function code objects. Ένα γράφημα για κάθε συνάρτηση. Αυτά τα γραφήματα είναι απλώς ένας ακόμα τρόπος αναπαράστασης του προγράμματος, αλλά πιο κατάλληλος για "αφαιρετικές" εργασίες μεταγλώττισης και βελτιστοποίησης (π.χ. εξαγωγή τύπων[1]).

Η διαδικασία είναι οργανωμένη σε στάδια (βλ. Σχήμα 3.1).

1. **Import & Flow Analysis**: Εδώ ο αρχικός κώδικας φορτώνεται στην μνήμη και υφίσταται τις βασικές διαδικασίες συντακικής ανάλυσης. Μετατρέπεται αρχικά σε tokens, έπειτα σε μια ενδιάμεση μορφή (Python bytecode) – όπως σε όλους τους μεταγλωττιστές – και τέλος στα διαγράμματα ροής του PyPy. Από αυτό το στάδιο και έπειτα φυσικά βρίσκεται μόνο μέσα στην τοπική μνήμη, σε μια αρκετά "στατική" μορφή για τα δεδομένα του RPython.

[1] βλ. type inference

2. **Annotator subsystem**: Εδώ ενεργοποιούνται οι περισσότερο αφαιρετικές μέθοδοι ερμηνείας του κώδικα. Μια καθολική ανάλυση, που θα ξεκινήσει από το σημείο εισόδου του κάθε διαγράματος (entry point), θα "εξάγει" γενικές πληροφορίες για τον κώδικα και θα συμπεράνει τους τύπους που μπορεί η κάθε μεταβλητή να φιλοξενήσει κατά το runtime βάσει φυσικά των συμφραζομένων. Μετά από αυτό το βήμα, έχουμε annotated flow graphs.

3. **RTyper**: Αυτό το σύστημα χρησιμοποιεί τους *υποσημειώμένους* είδικούς προσωρινούς τύπους που έχει προσθέσει ο annotator για να μετατρέψει τα statements και τις εντολές στα γραφήματα σε εντολές χαμηλότερου επιπέδου. Είναι ουσιαστικά μια γέφυρα μεταξύ των γραφημάτων και των παραγωγών κώδικα.

4. **Optimizations**: Η βελτιστοποίηση στο PyPy γίνεται εδώ. Το project περιλαμβάνει – και εφαρμόζει – μια πληθώρα βελτιστοποιήσεων, όπως αυτά που αναφέραμε στο κεφάλαιο 2. Επίσης το δικό μας module θα λάβει χώρα εδώ.

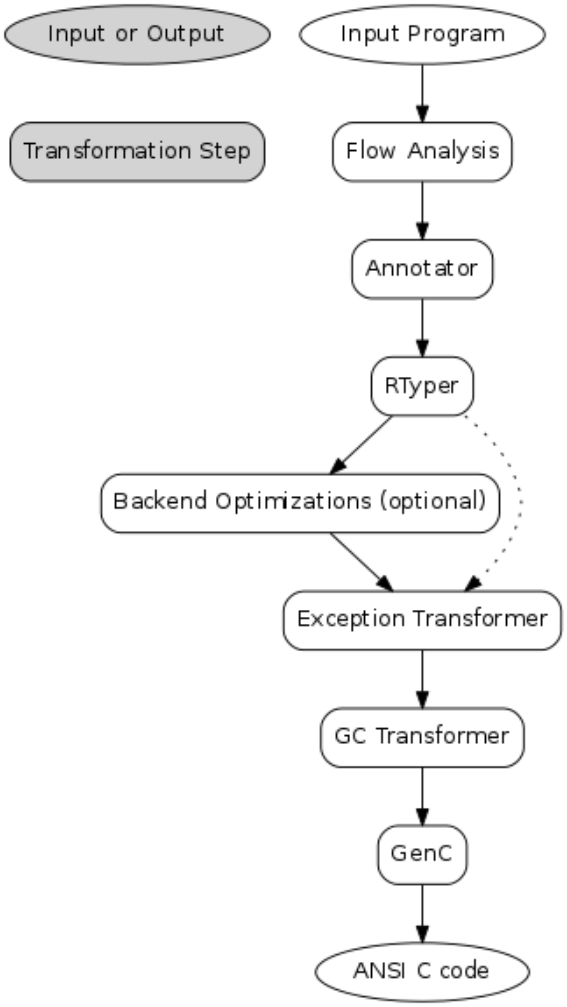
5. **Προετοιμασία Γραφημάτων**: Εδώ γίνονται οι τελικές ενέργειες πριν την παραγωγή κώδικα. Τα γραφήματα εδώ λ.χ. υφίσταντο ανάλυση για τον υπολογισμό ονομάτων μεταβλητών (για λόγους όπως debugging) κ.α. Σημαντικότερες όμως είναι οι εξής ενέργειες:

   - **Exception transformer**: Ο "μετατροπέας εξαιρέσεων" ενεργοποιείται σε αυτό το σημείο και θα εισάγει τον απαραίτητο κώδικα για τη διαχείριση των εξαιρέσεων που τυχαίνει να υπάρχουν.

   - **Garbage Collection Transformer**: Εδώ παίρνει σειρά το ειδικό εργαλείο διαχείρισης "σκουπιδιών μνήμης". Αυτό εκτελεί όποιες ενέργειες χρειάζονται για την σωστή διαχείριση μνήμης. Αφού η Python είναι πλήρως δυναμική γλώσσα και δεν απαιτεί από τον προγραμματιστή να διαχειρίζεται την μνήμη που χρειάζεται μόνος του, είναι απαραίτητη η ύπαρξη ενός υποσυστήματος διαχείρισης σκουπιδιών μνήμης.

6. **Code Generation**: Εδώ τα ήδη μαρκαρισμένα, σημειωμένα, βελτιστοποιημένα γραφήματα θα μετατραπούν σε κώδικα μηχανής. Εδώ υπάρχουν πολλοί "γεννήτορες" κώδικα ανάλογα με τις ανάγκες. Ο βασικότερος και ο πιο σημαντικός είναι ο παραγωγός κώδικα C. Αρχικά, τα γραφήματα θα μετατραπούν σε άλλα αντίστοιχα, ανάλογα με τον γεννήτορα και έπειτα αυτά θα μετατραπούν σε κώδικα ανάλογα με την επιλογή (π.χ σε κώδικα C)

7. Τέλος ο κώδικας αυτός θα οδηγηθεί στον μεταφραστή της εκάστοτε επιλογής (π.χ. `gcc`) για να παραχθεί το εκτελέσιμο αρχείο.



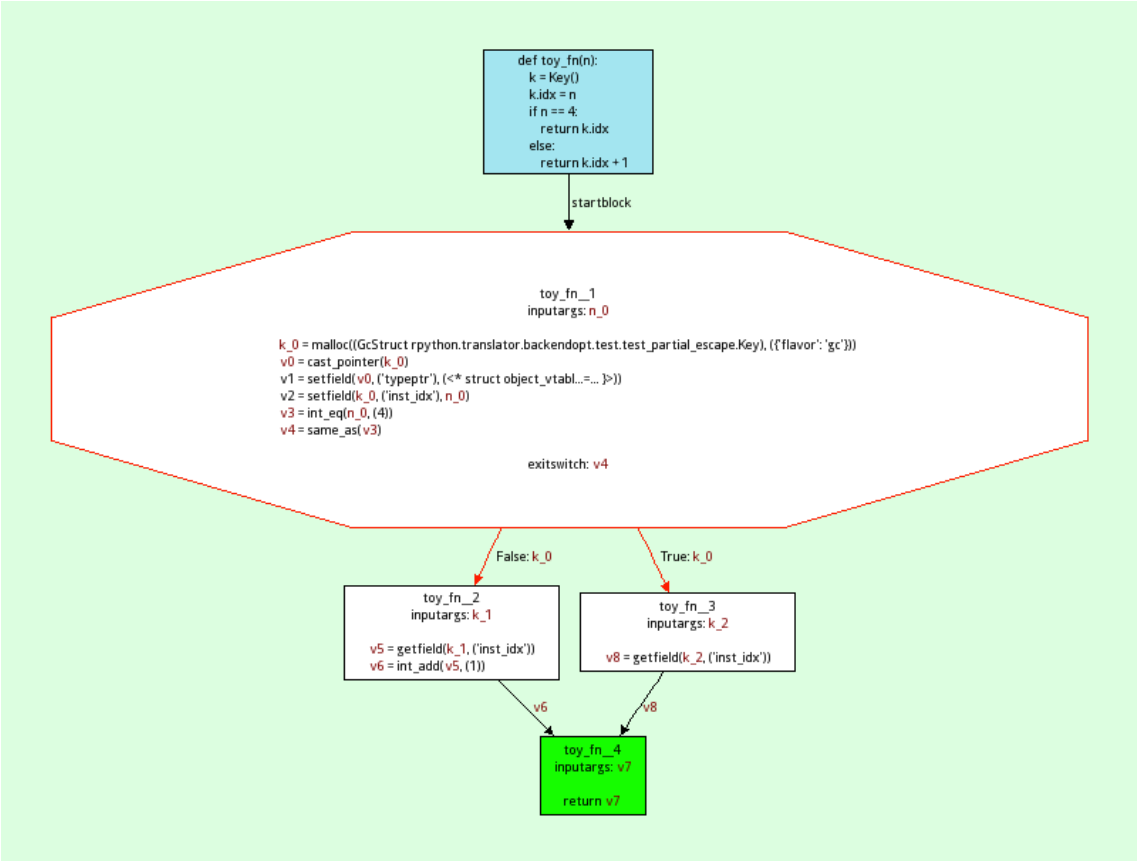
Σχήμα 3.1: Diagram of RPython translation process. [1]

3.2 Γραφήματα Ροής

3.2.1 Τρόπος δημιουργίας

Ο σκοπός του κατασκευαστή γραφημάτων (υλοποιημένος στο module `rpython.flowspace`) είναι να μετατρέψει τα αντικείμενα συναρτήσεων σε γραφήματα. Κανονικά οι μεταγλωττιστές παίρνουν μια συνάρτηση και την μετατρέπουν σε ενδιάμεσο bytecode κώδικα σε κάποια εικονική μηχανή (VM) και έπειτα την τρέχουν. Στο RPython όμως χρησιμοποιείται ένα *abstract interpretation* το οποίο λειτουργώντας αφαιρετικά μετατρέπει τον ενδιάμεσο κώδικα στα βασικα `Block`, τα οποία περιέχουν όλες τις εντολές οι οποίες επιδρούν πάνω σε αντικείμενα Python. Το αποτέλεσμα της κάθε εντολής αποθηκεύεται προσωρινά σε μια μεταβλητή η οποία μπορεί να χρησιμοποιηθεί στις επόμενες. Για ένα παράδειγμα βλ. Σχήμα 3.2 το οποίο είναι ένα γράφημα μιας απλής συνάρτησης με ένα if branch. Φαίνεται η γενική εικόνα των γραφημάτων, το branch καθώς ο τρόπος που επαναχρησιμοποιούνται οι προσωρινές μεταβλητές αποτελεσμάτων ανάλογα με τις ανάγκες.

Ο κατασκευαστής "κλείνει" ένα `Block` και πάει στο επόμενο σε 2 περιπτώσεις:



Σχήμα 3.2: Simple function flow diagram. Screenshot from the diagram editor.

1. Περίπτωση κλήσης `is_true()`. Όταν συμβαίνει αυτό, ο *abstract interpreter* δεν ξέρει φυσικά αν η συνθήκη θα είναι *True* ή *False* - αφού η γλώσσα μας είναι δυναμική – οπότε θα πρέπει να τις ακολουθήσει και τις 2, δημιουργώντας αντίστοιχα άλλες ροές (δηλαδή `Blocks`) για την κάθεμια.

2. Περίπτωση εμφάνισης *joinpoint*. Αυτή η περίπτωση λαμβάνει χώρα όταν η επόμενη εντολή, που πρόκειται να καταγράψουμε στο τρέχον `Block`, έχει ήδη καταγραφεί ή καταγράφεται τώρα από άλλο `Block`. Αυτό σημαίνει ότι ο μεταγλωττιστής έχει κλείσει κάποιον βρόγχο και πρόκειται να μεταγλωττίσει κώδικα τον οποίο έχει ήδη "δει". Τότε ο μεταγλωττιστής σταματά, και δημιουργείται ένα `Link` απο το τρέχον `Block` στο προηγούμενο, έτσι ώστε να κλείσει ο βρόγχος και στο διάγραμμα και να υπάρχει πιστή αναπαράσταση του κώδικα.

3.2.2 Το μοντέλο

Η βασική συνάρτηση που δημιουργεί τις δομές δεδομένων που απαρτίζουν το μοντέλο είναι η `build_flow()`. Η μονάδα (unit) ροής είναι το αντικείμενο `FunctionGraph` το οποίο μαζί με τα υπόλοιπα (που συνήθως περιέχονται σε αυτό) ορίζονται στο αρχείο `rpython/flowspace/model.py`. Τα περιγράφουμε παρακάτω:

- `FunctionGraph` Το διάγραμμα ροής μια συνάρτησης. Είναι η βασική δομή δεδομένων του μεταγλωττιστή. Περιέχει αναφορές και μια λίστα απο `Blocks`, τα οποία συνδέονται με `Links`. Βασικά κομμάτια του είναι:

  1. **startblock:** Το πρώτο block του γραφήματος. Είναι το σημείο εισόδου για την ροή του προγράμματος όταν αυτή φτάσει σε αυτή την συνάρτηση. Οι παράμετροι εισόδου θα δοθούν σε αυτό το block. Αυτό το block θα δείχνει όπως και όλα τα άλλα σε επόμενα ανάλογα με την ροή.

  2. **returnblock:** Το μοναδικό block στο κάθε γράφημα που μπορεί να εκτελέσει μια "επιστροφή" της συνάρτησης στην ροή του ανώτερου επίπεδου (function return). Είναι πάντα άδειο και δεν περιέχει ούτε την εντολής επιστροφής – αυτή υπονοείται. Τα links δείχνουν εδώ όταν και μόνο όταν θέλουν να τερματίσουν την συνάρτηση. Η μοναδική παράμετρος είναι η τιμή επιστροφής.

  3. **exceptblock:** Αυτό και το παραπάνω είναι τα μόνα blocks που μπορούν να τερματίσουν την ροή. Είναι το μοναδικό block που μπορεί να "πυροδοτήσει" μια εξαίρεση. Δέχεται 2 τιμές· η πρώτη είναι η κλάση της εξαίρεσης και η δεύτερη η τιμής της. Η μόνη περίπτωση να υπάρχει link προς αυτό το block είναι η ρητή πυροδότηση εξαίρεσης από το πρόγραμμα.

- `Block` Το πιο σημαντικό κομμάτι ενός γραφήματος. Ένα γράφημα ουσιαστικά είναι μια λίστα από blocks. Ένα block περιέχει μια λίστα από `SpaceOperations`, δηλαδή μια λίστα από εντολές. Λεπτομερειακά αποτελείται από:

  1. **inputargs:** Είναι μια λίστα με όλες τις μεταβλητές (με καινούργια ονόματα) που μπορούν να εισέλθουν στο block από οποιοδήποτε προηγούμενο. Η αντιστοίχηση των παλιών ονομάτων που υπάρχουν στα links με τα καινούργια γίνεται ένα προς ένα (π.χ. η πρώτη στα link με την πρώτη στη λίστα).

  2. **operations:** Η λίστα με τις εντολές που θα τρέξουν όταν η ροή καταλήξει σε αυτό το block. Θα τρέξουν όλες σειριακά χωρίς εξαιρέσεις. Οι εντολές υποδεικνύονται από `SpaceOperations` (βλ. πιο κάτω).

  3. **exits:** Λίστα με πιθανά *άλματα* (jumps). Φυσικά περιέχει links. Σε "συνεργασία" με το επόμενο μέλος καθορίζει που θα προχωρήσει η ροή του προγράμματος. Φυσικά περιέχει ένα ή περισσότερα links.

  4. **exitswitch:** Τα περιεχόμενα αυτού του μέλους ποικίλλουν:

     (α΄) Δεν υπάρχει *jump* και περιέχει `None`. Το *exits* περιέχει ένα link.

     (β΄) Άλμα υπό συνθήκη. Σε αυτή την περίπτωση το *exitswitch* είναι μια από τις μεταβλητές του block. Σε συνεργασία με ένα αντίστοιχο *exitcase* στα `Links`, διευθετεί τις περιπτώσεις των branches. Η ροή θα ακολουθήσει το `Link`, του οποίου το *exitcase*, ταιριάζει με το *exitswitch* του `Block`. Αν δεν υπάρχει ταίριασμα τότε έχουμε runtime error.

     (γ΄) Εξαίρεση. Το *exitswitch* περιέχει `Constant(last_exception)`. Το πρώτο `Link` του *exits* περιέχει `None` στο *exitcase* του για την περίπτωση που δεν υπάρχει εξαίρεση. Τα υπόλοιπα links δείχνουν στα διάφορα

classes των εξαιρέσεων και ακολουθούνται αντίστοιχα. Φυσικά με αυτόν τον τρόπο "προστατεύτεται" μόνο η τελευταία εντολή.

(δ΄) Επιστροφή. Περίπτωση *Returnblock*. Το *exitswitch* και το *operations* είναι άδεια και το *exits* είναι ρυθμισμένο σε `None`.

- `Link` Αποτελεί την σύνδεση μεταξύ των Blocks. Περιέχει:
  1. **prevblock:** Το προηγούμενο `Block` από το οποίο δείχνει αυτό το `Link`.
  2. **target:** Το `Block` στο οποίο δείχνει. Μπορεί να είναι μόνο ένα. Αν το `Block` πρέπει να δείξει σε περισσότερα, τότε πρέπει να υπάρχουν πολλά `Links`.
  3. **args:** Λίστα με `Variables` και `Constants`. Βλ. παραπάνω.
  4. **exitcase:** Βλ. παραπάνω.
  5. **last_exception:** Εδώ θα τοποθετηθεί (στο runtime) η κλάση εξαίρεσης σε τέτοια περίπτωση, αν το `Link` δείχνει σε `Block` εξαιρέσεων.
  6. **last_exc_value:** Ομοίως εδώ θα τοποθετηθεί η τιμή.
- `SpaceOperation` Υποδεικνύει μια "εντολή". Είτε καταγεγραμμένη είτε δημιουργημένη από τον ίδιο τον μεταφραστή. Σε αυτό το σημείο οι εντολές είναι σχετικά περιορισμένες (λ.χ. δεν μπορούν να πυροδοτήσουν μια εξαίρεση). Αυτό σημαίνει ότι ο μεταφραστής μπορεί να υποθέσει ότι είναι ασφαλείς και να εκτελέσει ενεργειες (όπως ανάγνωση μνήμης) που σε άλλες περιπτώσεις θα απαιτούσαν π.χ. locking. Υπάρχουν 2 πιθανές εξαιρέσεις σε αυτό το σενάριο.
  - η περίπτωση κλήσης άλλης συνάρτησης (βλ. `simple_call()`)
  - η τελευταία εντολή σε ένα `Block` χειρισμού εξαιρέσεων μπορεί να μην είναι ασφαλής.

  Περιέχει:
  1. **opname:** Το όνομα της εντολής. Η λίστα με τις εντόλες βρίσκεται στο αρχείο `rpython.flowspace.operation`.
  2. **args:** Τα ορίσματα της εντολής. Μπορεί να είναι `Constant` ή `Variable` αλλά να περιέχεται στο Block.
  3. **result:** καινούργια μεταβλητή στην οποία θα αποθηκευτεί το αποτέλεσμα.
- `Variable` Αποτελεί ένα placeholder και θα αποκτήσει τιμή κατά το runtime. Μπορεί προφανώς να είναι όρισμα σε κάποια εντολή. Περιέχονται κάποια μέλη με σκοπούς debugging.
  1. **name:** Εγγυημένα μοναδικό όνομα. Δεν ταιριάζει φυσικά με το όνομα που έχει η μεταβλητή στον κώδικα του χρήστη.
- `Constant` Αποτελεί την αναπαράσταση μιας σταθεράς. Όπως και παραπάνω μπορεί να είναι όρισμα σε κάποια εντολή, ή να βρίσκεται στην λίστα κάποιου Link για την αρχικοποίηση των μεταβλητών του επόμενου Block.
  1. **value:**

2. **key:** hashed[2] αντικείμενο που αντιπροσωπεύει την τιμή του *value*, για λόγους ασφάλειας κ.α.

3.3 Πέρασμα Υποσημειώσεων – Annotation Pass

Αυτό το υποσύστημα "υποσημειώνει" τα γραφήματα με ειδικές πληροφορίες (υπογραφές) έτσι ώστε τα επόμενα υποσυστήματα να γνωρίζουν τους εκάστοτε τύπους των μεταβλητών και των αντικειμένων. Μπορούμε να πούμε ότι είναι ένα είδος *type inference*. Μία από αυτές τις υπογραφές για τις μεταβλητές, που θα υποσημειωθούν, θα περιγράφει όλους τις πιθανούς τύπους που θα είναι δυνατόν να περιέχει η μεταβλητή αυτή κατά την διάρκεια του runtime. Η ανάλυση/πέρασμα γίνεται ανά συνάρτηση-γράφημα. Το αποτέλεσμα είναι ένα μεγάλο dictionary που "δείχνει" από μεταβλητές σε τέτοιες υπογραφές.

Η υπογραφή είναι ένα "στιγμιότυπο" (*instance*) μιας υποκλάσης ενός `SomeObject` αντικειμένου. Κάθε υποκλάση αντιπροσωπεύει και μια άλλη οικογένεια αντικειμένων. Η κλάση βάσης, όπως είπαμε, είναι η `SomeObject`, η οποία αντιπροσωπεύει ένα αντικείμενο Python. Υποκλάσεις της είναι – μεταξύ άλλων – οι: `SomeInteger()` (με την επιλογή `nonneg=True` ανάλογα άν χρειάζονται αρνητικοί αριθμοί), `SomeString()`, `SomeChar()` και `SomeTuple()`, οι οποίοι αντιπροσωπεύουν τα προφανή αντικείμενα. Να σημειώσουμε εδώ ότι όλα τα παραπάνω αντικείμενα υπογραφών είναι αμετάβλητα, δηλαδή δεν αλλάζουν *in-place*. Αυτό σημαίνει ότι εάν ο Annotator αποφασίσει να αλλάξει την υπογραφή, θα πρέπει να δημιουργήσει ένα καινούργιο αντικείμενο.

Υπάρχουν αντικείμενα βέβαια τα οποία είναι μεταβλητά, καθώς και αυτά που αντιπροσωπεύουν χρήζουν προσοχής και ειδικής μεταχείρησης. Σε αυτά, οι πληροφορίες μεταβάλλονται καθ όλη την διάρκεια του annotation pass. Περιλαμβάνουν το `SomeList()` και το `SomeDict()`.

Οι πιο εξειδικευμένες περιπτώσεις έχουν να κάνουν με κλάσεις ορισμένες από τον χρήστη (*user-defined classes*). Το σύστημα τις διαχειρίζεται με το `SomeInstance`. Για κάθε τέτοια κλάση διατηρούμε ένα *ClassDef*, που ουσιαστικά περιέχει τα ορίσματα και τα στοιχεία (attibutes) της κλάσης αυτής σε επίπεδο ορίσματος (class level attibutes) και στιγμιότυπου (instance level attibutes). Τα τελευταία "ανακαλύπτονται" προοδευτικά καθώς ο μεταγλωττιστής προχωρεί κατά μήκος του κώδικα. Για παράδειγμα, το παρακάτω θα μεταβάλει το *ClassDef* του στιγμιότυπου με το attribute.

```
inst.attr = value
```

Η διαφορές μεταξύ των επιπέδων class και των instance είναι λίγες και λεπτές. Γενικά τα attibutes των class-level χρησιμοποιούνται ως αρχικοποιητές για τα αντίστοιχα των instance-level[3]. Για κάθε attribute σημειώνονται οι θέσεις που λαμβάνουν χώρα αναγνώσεις και έπειτα, κατά την ανάλυση, αν υπάρχει γενικοποίηση του attribute για κάποιο λόγο, ο μεταγλωττιστής επιστρέφει σε αυτά τα σημεία και τα μεταβάλει ανάλογα, κάνοντάς τα *valid* ξανά.

[2] βλ. hash function

[3] Το ίδιο ισχύει και για (μεταβλητές που περιέχουν) συναρτήσεις.

3.4 RTyper

Ο RTyper είναι μια γέφυρα μεταξύ του Annotator και των παραγωγών κώδικα. Είναι απαραίτητος γιατί ο Annotator υποσημειώνει τον κώδικα με *υπογραφές* τύπων είτε δημιουργημένους από τον χρήση[4] είτε πολύ κοντά στο υψηλό επίπεδο της RPython. Οπότε – για να μπορούμε να παράγουμε κώδικα – απαιτείται η υποσημείωση των γραφημάτων για το επίπεδο της γλώσσας στόχου, δηλαδή χαμηλού επιπέδου με δείκτες και arrays. Ο RTyper αναλαμβάνει να αποφανθεί για τον τύπο και να αντικαταστήσει τις εντολές υψηλού επιπέδου με άλλες αντίστοιχες χαμηλότερου επιπέδου στα γραφήματα. Προφανώς η αντιστοιχία δεν είναι 1 προς 1, και μερικές φορές απαιτούνται πολύ περισσότερες εντολές στο χαμηλό επίπεδο για να μιμηθούμε τις αντίστοιχες του υψηλού. Επιπλέον οι διαθέσιμες εντολές και οι τύποι είναι σαφώς περιορισμένοι. Τελος, πρέπει να σημειωθεί ότι αυτό το βήμα θα μπορούσε να μην γίνει, αλλά οι σχεδιαστές έχουν επιλέξει να είναι ένα αυτόνομο module του συστήματος έτσι ώστε να κάνει την δουλεία του παραγωγού κώδικα πιο εύκολη και πιο αποδοτική. Μετά το πέρασμα του RTyper η παραγωγή κώδικα είναι σχετικά τετριμμένη.

3.5 Προετοιμασία

3.5.1 Διαχείριση Μνήμης

Αφού η Python είναι δυναμική ενώ οι περισσότερες από τις γλώσσες παραγωγών (compiled target languages) δεν είναι· θα πρέπει το σύστημα να κάνει κάποιες επιλογές για την διαχείριση της μνήμης. Αυτές οι επιλογές είναι εξαιρετικά σημαντικές για την τελική ταχύτητα των προγραμμάτων καθώς γενικά ο κώδικας Python τείνει να δεσμεύει μνήμη με πολύ γρήγορους ρυθμούς. Το σύστημα είναι ευέλικτο και υπάρχουν πολλές επιλογές. Δεν θα μπούμε σε επιπλέον πληροφορίες γιατί είναι έξω από τα όρια σκοπού της εργασίας αυτής, αλλά πληροφοριακά υπάρχουν:

- reference counting (απαρχαιωμένο – δεν χρησιμοποιείται πλέον στο σύστημα)
- ένας συντηρητικός BDW (Bˆohm-Demers-Weiser) συλλέκτης[2]
- πλέον χρησιμοποιούνται άλλοι custom συλλέκτες (βλ. [16]).

3.5.2 Διαχείριση Εξαιρέσεων

Ο κώδικας RPython υποστηρίζει πλήρως συντακτικό εξαιρέσεων ακριβώς όπως και η κανονική Python. Όπως και πριν βέβαια οι γλώσσες παραγωγών δεν ”γνωρίζουν“ την έννοια των εξαιρέσεων οπότε ο απαραίτητος κώδικας διαχείρισης θα πρέπει να προστεθεί. Το σύστημα δουλεύει όπως το κανονικό σύστημα της Python στον μεταγλωττιστή CPython: οι εξαιρέσεις υποδεικνύονται με ειδικές τιμές επιστροφής (return values) και η τρέχουσα εξαίρεση αποθηκεύεται σε μια καθολική (global) δομή δεδομένων, ορατή από όλα τα πεδία δράσης (scope) του προγράμματος.

[4] user-defined classes

Κεφάλαιο 4

Θεωρία Μερικής Ανάλυσης Διαφυγής

4.1 Εισαγωγικά

Αρχικά να σημειώσουμε, ότι η Θεωρία που παραθέτουμε εδώ είναι βασισμένη κυρίως στο paper των Würthinger et al[17] αλλά και σε επιπλέον έρευνα και εμπειρία που αποκτήσαμε κατά την υλοποίηση του module. Η μέθοδος, λοιπόν, που θα προσπαθίσουμε να υλοποιήσουμε στο module μας λέγεται *Μερική Ανάλυση Διαφυγής* (*Partial Escape Analysis*) και είναι μια γενίκευση της κανονικής "Απλής" Ανάλυσης Διαφυγής. Σκοπός μας είναι προφανώς η βελτιστοποίηση των προγραμμάτων και για να το πετύχουμε αυτό στοχεύουμε στο να μειώσουμε τον αριθμό των μεταβλητών που χρησιμοποιεί ο χρήστης, και τον αριθμό των προσβάσεων (*accesses*) που κάνει στην μνήμη για αυτές.

Η πιο προφανής βελτιστοποίηση με βάση αυτή την ανάλυση είναι η πλήρης εξάλειψη των δεικτών που δεν διαφεύγουν ή η αντικατάστασή τους με βαθμωτούς μέσα στο δυναμικό τους πεδίο. Επίσης δυνατή είναι η αντικατάσταση καταχωρήσεων μνήμης (malloc) στον σωρό (heap) με απλές κατωχηρώσεις στη στοίβα (stack) πράγμα που κάνει το πρόγραμμα πολύ πιο γρήγορο, και στην περίπτωση γλώσσας με σύστημα "συλλογής απορριμμάτων" αυτό οδηγεί στο τρέξιμο του συλλέκτη λιγότερες φορές. Τέλος μπορούμε να έχουμε κάποια οφέλη στα συστήματα συγχρονισμού. Αν ο δείκτης βρεθεί να μπορεί να προσπελαστεί μόνο από ένα νήμα, τότε μπορούμε να αφαιρέσουμε τις δομές συγχρονισμού. Εμείς θα ασχοληθούμε μόνο με αντικατάσταση βαθμωτών.

Να αναφέρουμε εδώ ότι μέχρι την στιγμή που γράφονται αυτές οι γραμμές, η στρατηγική που ακολουθεί το paper για τα loops δεν έχει υλοποιηθεί. Για την ακρίβεια δεν έχει υλοποιηθεί καμία στρατηγική για τα loops, και απλά αγνοούμε την "ιδιότητα" που έχουν να επιστρέφουν την ροή της εκτέλεσης προς τα πίσω. Εχουμε λάβει υπόψιν μας την μη γραμμική ροή σε περίπτωση ύπαρξης *jump back* στο γράφημα – aka: *loop* αλλά δεν έχουμε τρόπο ανάλυσης των εσωτερικών μεταβλητών και αντικειμένων. Όταν εντοπιστεί *loop*, τότε *επανατοποθετούνται* όλα τα εικονικά αντικείμενα. Θα δούμε τι σημαίνουν οι παραπάνω παράξενοι ίσως όροι παρακάτω. Κανονική υλοποίηση του *loop handling* συμπεριλαμβάνεται στις μελλοντικές εργασίες. Η υλοποίηση δεν έγινε γιατί κρίναμε "βαρύ" και "ακριβό"[1] το *approach* – την στρατηγική που παραθέτεται: Αυτό που πρότεινε το paper ήταν ένα είδος *testing* του *escapability* των εσωτερικών

[1] αν όχι λανθασμένο

αντικειμένων που χρησιμοποιούνται στο loop, και η "ανάκληση" όλων των ενεργειών (*reverse*) σε περίπτωση που στο τέλος κριθεί μη αποδοτική ή λανθασμένης λογικής.

Μια ανάλυση της πολυπλοκότητάς της δίνεται εδώ[6] και επιπλέον παραθέτουμε εμείς κάποιες λεπτομέρειες σε επόμενη παράγραφο.

4.2 Η Απλή Ανάλυση

Για να πετύχουμε τα παραπάνω αρκεί να "αναλύσουμε" τις μεταβλητές μια-μια και να καθορίσουμε όλα τα μέρη όπου μια μεταβλητή απαιτείται να υπάρχει καθώς επίσης και το αν η διάρκεια ζωής της μπορεί να αποδειχθεί να περιορίζεται μόνο στην τρέχουσα διαδικασία και/ή νήμα[2], δηλαδή το *scope*. Με τον όρο μεταβλητή εδώ εννοούμε ένας δείκτης σε μια θέση μνήμης. Χρησιμοποιούμε τους όρους "μεταβλητή" και "δείκτης" εναλλακτικά.

Με άλλα λόγια, η *Ανάλυση Διαφυγής* είναι μια μέθοδος για τον καθορισμό του *δυναμικού πεδίου* των δεικτών ενός προγράμματος. Την περιοχή δηλαδή, στην οποία οι δείκτες αυτοί είναι ενεργοί και έγκυροι ή αλλιώς την περιοχή που μπορεί το πρόγραμμα να έχει "πρόσβαση" σε αυτούς. Πιο συγκεκριμένα η *Ανάλυση Διαφυγής* ελέγχει εάν ένα (δεσμευμένο από το σύστημα) αντικείμενο *διαφεύγει* από αυτό το scope.

Ένας δείκτης που δημιουργείται από κάποια συνάρτηση – δηλαδή ένα *reference* σε ένα αντικείμενο (της Python) – μπορεί να *διαφύγει* σε κάποια άλλη. Τότε το δυναμικό του πεδίο μεγαλώνει. Ένας δείκτης λέμε ότι έχει *διαφύγει* όταν αυτός είναι διαθέσιμος (ή αλλιώς ορατός) και από άλλα scopes στο πρόγραμμα, οπότε **απαιτείται** να υπάρχει αυτός καθ'αυτός. Ο σημαντικότερος τρόπος για διαφυγή είναι η επιστροφή του αντικειμένου ως return value μιας συνάρτησης. Θα δούμε αναλυτικά και άλλους τρόπους στο επόμενο κεφάλαιο απλά να σημειώσουμε ότι υπάρχουν και άλλοι πιο περίπλοκοι τρόποι διαφυγής όπως στην περίπτωση functional γλωσσών και tail-call optimization, όμως δε θα ασχοληθούμε με αυτές σε αυτή την εργασία.

Ως ιστορική σημείωση να πούμε ότι παλαιότερα οι μέθοδοι αναλύσης διαφυγής χρησιμοποιούσαν αλγόριθμους που ονομάζονται "Equi-Escape Sets"[10] για να αποφανθούν εάν τα αντικείμενα διαφεύγουν του τρέχοντος scope. Δημιουργούσαν σύνολα (sets) από αντικείμενα που ανήκουν στην ίδια κατηγορία διαφυγής, έπειτα με το να αναλύουν τις μεθόδους και τις συναρτήσεις, μπορούσαν να μαρκάρουν τα διάφορα αντικείμενα και σύνολα (π.χ. αν το αντικείμενο από το ένα σύνολο αναθέτονταν σε ένα άλλο σύνολο) και να τα συγχωνεύσουν.[17]

Τα αποτελέσματα που θα επιστρέψουμε θα χρησιμοποιηθούν για να τροποποιηθεί ο κώδικας με βέλτιστο τρόπο. Παρακάτω θα παραθέσουμε ένα παράδειγμα για τον τρόπο με τον οποίο ο βελτιστοποιητής μας θα μετατρέψει ένα κομμάτι κώδικα. Να υπογραμμίσουμε εδώ ότι το παράδειγμα είναι από το paper, τροποποιημένο όμως με τέτοιο τρόπο έτσι ώστε να είναι σχεδιασμένο φυσικά για Python και να αναδεικνύει τις δυνατότητες και λεπτομέρειες της γλώσσας.

```
1 class Key(object):
2     def __init__(self, idx, ref):
```

[2] *thread*

```
3             self . idx  = idx
4             self . ref  = ref
5         def  equals ( self ,  other ):
6             return  ( self . idx  == other . idx  && self . ref  == other . ref )
7
8   def  createValue  (...) :
9         ...
10
11  cacheKey = None
12  cacheValue  = None
13
14  def  getValue ( idx ,  ref ):
15        key  = Key(idx,  ref )
16        if  key. equals ( cachekey ):
17            return  cacheValue
18        else :
19            return  createValue  (...)
```

Το παραπάνω κομμάτι (μετά από ανάλυση διαφυγής και inlining – που εφαρμόζεται από το PyPy) θα γίνει κάπως έτσι:

```
1   ...
2   def  getValue ( idx ,  ref ):
3         idx1  = idx
4         ref1  = ref
5         tmp = cacheKey
6         if  (idx1  == tmp.idx  && ref1 == tmp.ref ):
7             return  cacheValue
8         else :
9             return  createValue  (...) ;
10  ...
```

Αυτό που θα αλλάξει εδώ είναι η συνάρτηση `getValue()` κατά το compilation της. Όταν ο μεταγλωττιστής φτάσει σε αυτή, θα καταλήξει στο συμπέρασμα ότι κανένα reference στο δεσμευμένο αντικείμενο `Key` δεν διαφεύγει από το τρέχων compilation scope της συνάρτησης. Αυτό σημαίνει ότι κανένα reference δεν θα υπάρχει αφού τερματίσει και επιστρέψει η συνάρτηση και ότι κανένα άλλο αντικείμενο ή κατασκευή ή συνάρτηση δεν θα "θέλει" να αναφερθεί σε αυτό. Οπότε, μπορούμε να τροποποιήσουμε τον κώδικα με τους παρακάτω τρόπους:

- Η δέσμευση μνήμης του αντικειμένου στον σωρό (*garbage collected heap*) μπορεί να αντικατασταθεί με μια απλή δέσμευση στην στοίβα (*stack*) της συνάρτησης ή σε κάποια άλλη περιοχή ή ζώνη[3] η οποία δεν υπόκειται συλλογή απορριμάτων. Αυτό σημαίνει ότι μια κανονική δέσμευση (με την εντολή *malloc*), που θα έχει μεγάλη διάρκεια ζωής και θα είναι ακριβή και στην αρχικοποίηση και στον καθαρισμό της, μπορεί να μεταβληθεί σε μια απλή προσωρινή μεταβλητή στην συνάρτηση που μετά το πέρας του runtime της, θα ελευθερωθεί. Αυτό μας εξοικονομεί και χρόνο (σε δύο περιπτώσεις – *allocation* και *freeing*) και μνήμη.

[3]Ζώνες ονομάζονται περιοχές του σωρού (heap) που έχουν ήδη αρχικοποιηθεί και έχουν περιορισμένη διάρκεια ζωής με σκοπό αυτό ακριβώς το είδος χρήσης.

- *Αντικατάσταση Βαθμωτών* Εκτός από το παραπάνω μπορούμε να προχωρήσουμε σε αντικατάσταση με βαθμωτούς. Αυτό θα εξαλείψει τελείως την δέσμευση, με το να αντικαταστήσει τα fields του αντικειμένου με τοπικές μεταβλητές.

Βλέπουμε ότι η δέσμευση του αντικειμένου `Key` αντικαταστήθηκε με τις τοπικές μεταβλητές `idx1` και `ref1`. Αν και αυτό είναι εξιδανικευμένο παράδειγμα, μπορούμε να περιμένουμε πολλές τέτοιες βελτιώσεις ακόμα και σε *real-world* κομμάτια κώδικα και βιβλιοθήκες, όπως θα δούμε στο τελευταίο κεφάλαιο.

Επίσης σημαντική αλλαγή είναι το inlining που πραγματοποιήθηκε στον κώδικα, και αυτό ήταν ζωτικό στο να μπορέσει η ανάλυσή μας να "εξάγει" το σωστό αποτέλεσμα για τις μεταβλητές. Σε αντίθετη περίπτωση η "μονή" χρήση του αντικειμένου (του `Key`) θα είχε ουσιαστικά κρυφτεί πίσω από μια κλήση στην μέθοδο του αντικειμένου.

Σημειώνουμε ότι το παραπάνω είναι κανονικός κώδικας Python – ελλιπής βέβαια και δεν μπορεί να τρέξει – για παιδαγωγικούς σκοπούς. Για παράδειγμα δεν παραθέτουμε τον ορισμό της συνάρτηση `createValue()` που προφανώς είναι η "ακριβή" συνάρτηση που δημιουργεί το αντικείμενο που θέλουμε να αποφύγουμε (βλ. γραμμές 8-9 και 19). Όσων αφορά το δεύτερο βελτιστοποιημένο κομμάτι, δεν παραθέτουμε καθόλου την κλάση `Key()` και δεν δίνουμε το τι περιέχουν οι καθολικές μεταβλητές (βλ. γραμμή 5). Ο αναγνώστης αρκεί να ξέρει ότι προσπαθούμε να αποφύγουμε την δέσμευση και χρήση μνήμης. Να υπογραμμίσουμε επίσης ότι, στην πραγματικότητα ο βελτιστοποιητής μας δεν αλλάζει – ούτε παράγει – κώδικα Python, αλλά middleware κώδικα, και δουλεύει πάνω σε διαγράμματα ροής όπως θα δούμε στα επόμενα κεφάλαια.

Τέλος να πούμε ότι η απλή ανάλυση διαφυγής είναι ήδη υλοποιημένη στο PyPy στο αρχείο `escape.py`, και χρησιμοποιείται κανονικά και αποδοτικά στο compiling του ίδιου του μεταγλωττιστή καθώς και άλλων προγραμμάτων. Στοχεύουμε σε τέτοιου intergration του module μας στο σύστημα.

4.3 Η Μερική Ανάλυση

Η "Μερική Ανάλυση Διαφυγής" δουλεύει ομοίως με παραπάνω, αλλά είναι πιο ισχυρή με την έννοια ότι λαμβάνει υπόψιν της και τα διάφορα παρακλάδια (branches) της ροής εκτέλεσης κατά την ανάλυση. Με άλλα λόγια η κανονική Ανάλυση Διαφυγής χειρίζεται μια περίπτωση `if` "στατικά" – ουσιαστικά την αγνοεί, ενώ η αντίστοιχη μερική ακολουθεί και τις 2 πιθανές ροές. Το γεγονός αυτό όμως καθιστά την σχεδίαση της υλοποίησης πιο δύσκολη.

Σε πολλές περιπτώσεις το γεγονός ότι η απλή ανάλυση αγνοεί τα *if splits* και απλά παίρνει καθολικές αποφάσεις, οδηγεί στην ανικανότητα να ανιχνευτεί η δυνατότητα αφαίρεσης κάποιων αντικειμένων. Για αυτό τον λόγο η μερική ανάλυση, η οποία μπαίνει στο κάθε branch και το αναλύει ξεχωριστά, θεωρείται ισχυρότερη. Διατρέχει τον κώδικα με τον ίδιο τρόπο όπως η απλή ανάλυση αλλά διατηρεί "καταστάσεις" (states) για κάθε branch που έχει γίνει στην ροή και όταν τα branches συγκλίνουν ξανά, μπορεί να αποφασίσει αν το αντικείμενο διαφεύγει ανάλογα με το τι συνέβη στο κάθε ένα

από αυτά. Οι αποφάσεις αυτές λαμβάνουν χώρα και για κάθε state ξεχωριστά αλλά το ένα επηρεάζει το άλλο αφού φυσικά ο μεταγλωττιστής δεν μπορεί να ξέρει ποιο από τα branches θα ακολουθηθεί πραγματικά.

Όλα τα επόμενα κεφάλαια απο τώρα και στο εξής αναφέρονται στην μερίκη ανάλυση διαφυγής. Στο επόμενο υποκεφάλαιο, αφού πρώτα συζητήσουμε για την πολυπλοκότητα της μεθόδου, Θα συνεχίσουμε με κάποιες λεπτομέρειες για τον τρόπο που δουλεύει η μερική ανάλυση προκειμένου να εγκλιματιστεί ο αναγνώστης και έπειτα θα δώσουμε ένα εκτενές παράδειγμα με όλες τις λεπτομέριες που θα μπορούν τότε να κατανοηθούν καλύτερα.

4.4 Πολυπλοκότητες

Στο paper που βασιζόμαστε[17] δεν γίνεται καμία αναφορά στην πολυπλοκότητα της μεθόδου είτε της απλής είτε της μερικής ανάλυσης διαφυγής. Εμείς θα αποπειραθούμε να δώσουμε μια πρόβλεψη αν και θα παραλείψουμε τυχών αποδείξεις καθώς είναι εκτός του scope της εργασίας. Στην καλύτερη περίπτωση (*best case scenario* – BCS) η πολυπλοκότητα προφανώς περιορίζεται στον αριθμό των operations δήλαδή $O(operations)$, αφού δεν γίνονται καθόλου αλλαγές και οι εντολές απλά "αγγίζονται" όλες μια φορά. Παράδειγμα ένα πάρα πολύ απλό πρόγραμμα χωρίς *if* που επιστρέφει μια σταθερά και όχι ένα αντικείμενο. Από την άλλη στην χειρότερη περίπτωση (*worst case scenario* – WCS) η πολυπλοκότητά μας ανεβαίνει στο επίπεδο:

$$O(operations \times blocks)$$

αφού θα πρέπει όλες οι εντολές να αφαιρεθούν και να ξανατοποθετηθούν στα επόμενα `Blocks`. Ως παράδειγμα να αναφέρουμε την περίπτωση όπου έχουμε μια εντολή `malloc` και αμέσως μετά ένα *split* με χιλιάδες branched- off *Block*s. Σε αυτή την περίπτωση η `malloc` θα πρέπει να επανατοποθετηθεί (ή να τεσταριστεί αν πρέπει να επανατοποθετηθεί) τόσες φορές όσες και τα *Block*s. Βέβαια η πολυπλοκότητα μένει σαφώς στην ίδια τάξη μεγέθους οπότε η μέση περίπτωση (*average case scenario* – ACS) είναι ίδια.

Τα παραπάνω ισχύουν για την στρατηγική που ακολουθούμε εμείς. Η στρατηγική του paper είναι ίδια με την διαφορά του χαρακτηριστικού της υλοποίησής τους για τα loops. Εμείς, όπως είπαμε παραπάνω, ουσιαστικά αγνοούμε τα loops. Στο paper οι συγγραφείς προτείνουν μια μέθοδο "testing" και "reversing" σε περίπτωση λάθους. Αυτό όμως είναι εξαιρετικά μη αποδοτικό καθώς σίγουρα θα υπάρχει η περίπτωση επανατοποθέτησης και μετά ανάγκη αναίρεσης οπότε στην χειρότερη περίπτωση (WCS) η πολυπλοκότητα ανεβαίνει τάξη μεγέθους και γίνεται τετραγωνική (quadratic factor).

4.5 Τρόπος λειτουργείας – Λεπτομέρειες

4.5.1 Γενικά

Η βελτιστοποίηση αυτή θα κρίνει ποιες μεταβλητές είναι απαραίτητο να υπάρχουν αυτές καθ' αυτές (ίσως γιατί βασίζεται κάποιος εξωτερικός παράγωντας σε αυτές) και θα αποπειραθεί να αφαιρέσει τις υπόλοιπες. Η αφαίρεση αυτή θα πρέπει φυσικά να είναι έξυπνη καθώς η λειτουργία και η ορθότητα του προγράμματος θα πρέπει προφανώς να διατηρηθούν.

Οι 2 κυριότεροι λόγοι για να απαιτείται η ύπαρξη της μεταβλητής είναι:

1. η χρήση της ως τιμή επιστροφής από την συνάρτηση (*return*)
2. η αλλαγή του scope της (*globalization*)

4.5.2 Τρόπος ανάλυσης & δομές δεδομένων

Γενικά η μέθοδος, που περιγράφεται στο paper που ακολουθούμε [17] και θα υλοποιήσουμε εμείς, σκανάρει το πρόγραμμα και "παρακολουθεί" τις μεταβλητές και τις κατασκευές του προγραμματιστή. Ξεκινά σειριακά ακολουθόντας την ροή της εκτέλεσης στο κάθε γράφημα – αφού είναι graph- based – από ένα ειδικό Block που είναι το πρώτο (*startblock*). Η σειρά που θα ακολουθήσει στα `Blocks` είναι φυσικά η σειρά της ροής πηγαίνοντας από πατέρα σε παιδί/παιδιά, ανοίγοντας καταλλήλως ανάλογα με τα *splits* και τα *merges*. Το *merging* γίνεται στα ειδικά `Blocks` που λέγονται *mergeblocks* όπως θα δούμε παρακάτω. Η ανάλυση αυτή σταματά όταν φτάσουμε σε κάτι που αποκαλείται *control sink*. Αυτό είναι συνήθως κάποιο `Return Block` αλλά μπορεί να έιναι και `Throw Block` – το τελευταίο συμπεριφέρεται με τον ίδιο τρόπο με ένα `Return Block` και εσωτερικά είναι ομοίως υλοποιημένο.

Καθώς γίνεται αυτό το παραπάνω πέρασμα μια φορά από κάθε `Block`, διατηρούμε κάποιες πληροφορίες για τις μεταβλητές ανάλογα με την εντολή (operation) που θα συναντήσει ο μεταγλωττιστής στο εκάστοτε `Block`. Εδώ να πούμε ότι οι εντολές είναι αυστηρά σειριακές μέσα σε κάθε `Block` από την φύση των γραφημάτων. Αυτός είναι και ο τρόπος που τις συναντά λοιπόν ο μεταγλωττιστής, χωρίς εμείς να χρειαστεί να κάνουμε κάτι συγκεκριμένο όπως στην περίπτωση της σειράς επεξεργασίας των `Blocks`. Όλα γίνονται αυτόματα από το σύστημα των γραφημάτων του PyPy.

Η βασική δομή που χρησιμοποιούμε είναι τα "εικονικά αντικείμενα" (*VirtualObjects*). Η χρήση τους έγκειται στη αναπαράσταση των πιθανών μεταβλητών που μπορεί να είναι "υποψήφιες" για αφαίρεση (αντικατάσταση με βαθμωτούς).

Ο μεταγλωττιστής, για κάθε καινούργια μεταβλητή που συναντά, δημιουργεί και διατηρεί ένα τέτοιο εικονικό αντικείμενο, και διαγράφει την ακριβή (σε χρόνο και σε μνήμη) δέσμευση με την εντολή `malloc` που υπήρχε προηγουμένως. Αυτό γίνεται στην αρχή, κατευθείαν με το που θα εντοπιστεί η καινούργια μεταβλητή, για 2 λόγους:

- Εαν δεν έχει εντοπιστεί κάποιος λόγος για να υπάρχει η μεταβλητή αυτή καθ' αυτή στον σωρό (*heap*) τότε την αφαιρούμε – οπότε ουσιαστικά την τοποθετούμε στην στοίβα της τρέχουσας συνάρτησης.
- Δεν υπάρχει λόγος επιστροφής του focus της επεξεργασίας του μεταγλωττιστή πίσω στο `Block` που πρωτοεμφανίζεται η μεταβλητή μετά την απόφαση για την αναγκαιότητα ύπαρξής της. Σε αυτή την περίπτωση η επεξεργασία θα έπρεπε να επισκευτεί όλα τα `Blocks` πολλές φορές και η πολυπλοκότητα χρόνου αλλά και η πολυπλοκότητα του codebase θα ανέβαινε επικίνδυνα. Οπότε αντί να διατηρούμε *metadata* και να κάνουμε την αλλαγή μια φορά, παρακολουθούμε τις ενέργειες της εντολής και κάνουμε αντίστοιχα *mirror* στο εικονικό αντικείμενο (βλ. παρακάτω).

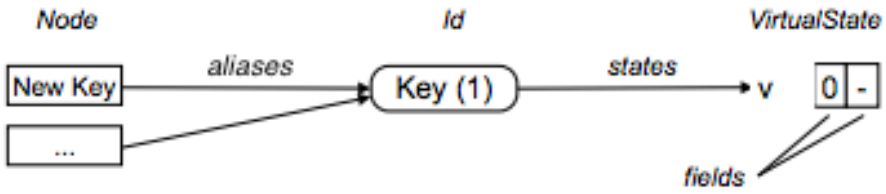
Έπειτα, όπως είπαμε παραπάνω, σε κάθε "χρήση"[4] της μεταβλητής και σε κάθε άλλη εντολή που μπορεί να την επηρεάσει, το αντίστοιχο εικονικό αυτό αντικείμενο μεταβάλλεται. Όταν ο μεταγλωττιστής αντιληφθεί ότι βρίσκεται σε κάποια από τις περιπτώσεις που η μεταβλητή πρέπει να υπάρχει (βλ. παραπάνω) τότε διαγράφει το εικονικό αντικείμενο και αντιστρέφει οποιαδήποτε ενέργεια είχε κάνει.

Τα εικονικά αυτά αντικείμενα περιέχουν όλες τις πληροφορίες που χρειάζεται ο μεταγλωττιστής για να μιμηθεί τις αντιστοιχες μεταβλητές. Πρώτα απ'όλα περιέχουν όλα τα object attributes τους, καθώς τα πάντα στην Python είναι αντικείμενα. Έπειτα μπορούν να περιέχουν πληροφορίες locking και syncing για τις περιπτώσεις κατανεμημένου κώδικα. Τέλος περιέχουν πληροφορίες για την θέση που βρίσκονταν καθώς και για τον τρόπο που πρέπει να λάβει χώρα η δέσμευση μνήμης τους (βλ. τρόπους casting ή λ.χ. αν θα γίνει ή όχι χρήση του garbage collector).

Ο αναλυτής μας όμως δεν διατηρεί μόνο τέτοια αντικείμενα αλλά επιπλέον και δομές δεδομένων προκειμένου να τα αποθηκεύει. Αυτό είναι ένα *dictionary* που δείχνει από ids μεταβλητών σε εικονικά αντικείμενα. Διατηρεί επίσης και ένα σύστημα για να δημιουργεί aliases για τα αντικείμενα αυτά ανάλογα με τις μετονομασίες και τα castings που γίνονται στον κώδικα. Λόγω της υψηλής δυναμικότητας της γλώσσας τα casting βρίθουν και δυσκολεύουν το έργο της σχεδίασης. Αυτή η δομή στο paper περιγράφεται ως ένα άλλο *dictionary* που δείχνει στο id του πρώτου *dictionary*. Εμείς θεωρήσαμε αυτό υπερβολικό και αφού μπορούμε να εκμεταλλευτούμε τις δυνατότητες της Python, μπορούμε να έχουμε δύο - ή προφανώς περισσότερα - keys (aka ids) που δείχνουν στο ίδιο εικονικό αντικείμενο οπότε ουσιαστικά έχουμε σύστημα aliases. Το σύστημα, για την διαχείριση της πληροφορίας που ακολουθούμε, μπορεί να περιγραφεί από το σχήμα 4.1.

Αυτά τα αντικείμενα *dictionaries* λέγονται *state* γιατί διατηρούν την κατάσταση των μεταβλητών για το εκάστοτε `Blocks` που βρίσκεται τώρα ο μεταγλωττιστής. Κάθε φορά που η επεξεργασία μετακινείται στο επόμενο `Block` τότε δημιουργείται ένα αντίγραφο του *state*. Επιπλέον λεπτομέρειες για ποιό λόγο και πως συμβαίνει αυτό στο Κεφάλαιο 4. Λοιπόν η πληροφορία για την κατάσταση των τρέχουσων μεταβλητών βρίσκεται σε αυτή την δομή με μορφή αντικειμένων και όπως θα δούμε όταν υπάρχει ένα τέτοιο αντικείμενο για μια μεταβλητή (συνδεόμενο φυσικά με αυτή με aliases και ids στο *state*) τότε το αντικείμενο είναι πάντα εικονικό. Αν δεν είναι εικονικό το

[4] βλ. ανάγνωση(read) ή εγγραφή(write) σε αυτή



Σχήμα 4.1: Schema of Basic Data Structures. [17]

`VirtualObject` διαγράφεται από το *state*, σε αντίθεση με την μέθοδο που ακολουθείται στο paper. Εκεί προτείνεται η ύπαρξη δύο είδών εικονικών αντικειμένων – ένα για την περίπτωση εικονικότητας (`VirtualState`) και ένα για την περίπτωση αναγκαιότητάς του (`EscapedState`). Οι διαφορές ήταν μικρές και κρίναμε ότι δεν ήταν αναγκαίο να υπάρχει κατάσταση και αντικείμενο πληροφοριών για ένα αντικείμενο που υπάρχει στην μνήμη έτσι και αλλίως αυτό καθ' αυτό.

Όπως βλέπουμε στο 4.1, το πρώτο λευκό κουτάκι είναι το "node". Δηλαδή μια αφαιρετική έννοια που περιλαμβάνει κυρίως ονόματα και references σε μεταβλητές που υπήρχαν στην αρχική έκδοση του κώδικα. Αυτά "δείχνουν" στο id, το οποίο είναι ουσιαστικά το όνομα που έχουμε δώσει στην μεταβλητή[5] αυτή εσωτερικά – για δική μας χρήση μέσα στο σύστημά μας. Αυτό δείχνει φυσικά στο εικονικό αντικείμενο που είναι μοναδικό για κάθε μεταβλητή ανεξάρτητα από το πόσα ονόματα (aliases) έχει. Στην πραγματικότητα χρησιμοποιούμε ένα *defaultdict* της Python για να δείχνουμε κατευθείαν απο τα nodes στα αντίστοιχα εικονικά αντικείμενα. Αυτό που φαίνεται στην εικόνα είναι το θεωρητικό μοντέλο μας.

Να πούμε επίσης ότι στο paper εξηγείται πλήρως – και υπάρχει στο μοντέλο του – ο τρόπος για το *handling* όλων αυτών των διαδικασιών με δυνατότητες κλειδώματος (locking) για παράλληλη επεξεργασία (βλ. *multithreading*). Εμείς δεν το συμπεριλαμβάνουμε αυτό, καθώς:

1. Δεν είναι αναγκαίο γιατί ασχολούμαστε με μεταγλωττιστές, ο εσωτερικός κώδικας των οποίων δεν είναι συνήθως *multi-threaded*.
2. Ακόμαι και αν χρειαστεί η δυνατότητα, η Python δεν το υποστηρίζει καλά λόγω του καθολικού κλειδώματος που έχει (Global Interpreter Lock).

4.5.3 Πότε αλλάζουν τα states

Οι περιπτώσεις που επειρεάζονται τα state αντικείμενα – σύμφωνα πάντα με την ιδανική κατάσταση του paper[**standler2014partial**] και την υλοποίησή τους σε Java (η οποία είναι ένας βελτιστοποιητής στον μεταγλωττιστή της που λέγεται *Graal*) – ή αλλιώς αυτά που χρήζουν προσοχής είναι:

- Κόμβοι (δηλαδή `Blocks`) που περιέχουν εντολές δέσμευσης μνήμης στον σωρό (malloc). Θα δημιουργήσουν εικονικά αντικείμενα.

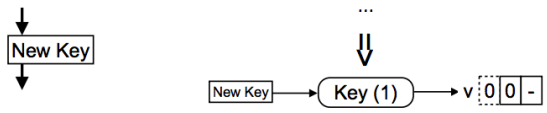
[5] ή μεταβλητές

- Εντολές που κάποιο από τα argument τους περιέχουν keys σε υπάρχοντα εικονικά αντικείμενα ή aliases σε αυτα. Οι εντολές τότε πρέπει να επεξεργαστούν και είτε να μεταβληθούν είτε να αφαιρεθούν από τον κώδικα ολοκληρωτικά.
- `Blocks` που είναι ειδικά όπως *mergeblock Returnblock* και *LoopBegin*.

4.5.4 Πώς αλλάζουν τα states

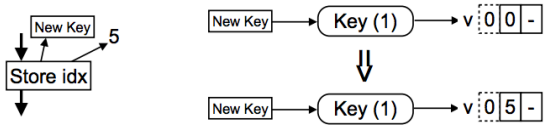
Θεωρητικά, σύμφωνα πάντα με το paper, αυτές οι αλλαγές στα states είναι οι παρακάτω (τα γράμματα στις παρακάτω περιγραφές αντιστοιχούν στις εικόνες 4.2 έως 4.5):

α΄ Για κάθε δέσμευση, δημιουργούνται καινούργια εικονικά αντικείμενα (`VirtualObject` – VOs) και τα αντίστοιχα entries στα mapping dictionaries.



Σχήμα 4.2: (ά) Actions upon allocation of a VO. [17]

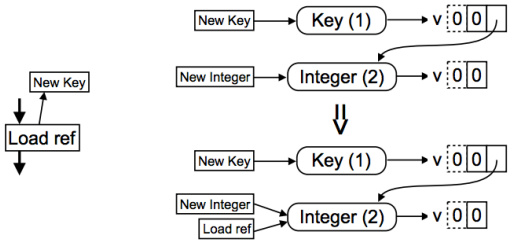
β΄ Για κάθε "ανάθεση" (εντολή `setfield`) αντικειμένου που δεν είναι σε κάποιο dictionary (states ή aliases – στην δική μας υλοποίηση: μόνο states) σε πεδίο εικονικού αντικειμένου μεταβάλλεται η αντίστοιχη θέση μνήμης στο attribute του εικονικού αντικειμένου ώστε αυτό να "ξέρει" περί της ανάθεσης. Το γεγονός ότι το πρώτο αντικείμενο δεν ανήκε σε κάποιο dictionary μας δείχνει ότι δεν είναι και αυτό εικονικό αντικείμενο, καθώς αν ήταν θα υπήρχε το alias του στο dictionary. Αρα η παραπάνω περίπτωση είναι η περίπτωση ανάθεσης πραγματικού – υπάρχοντος – αντικειμένου σε εικονικό.



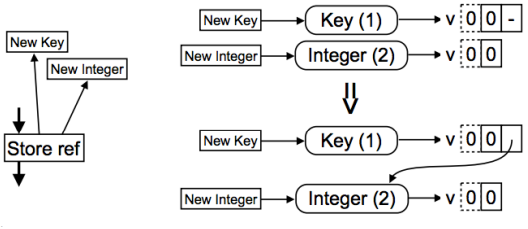
Σχήμα 4.3: (β') Actions upon simple external object storing inside a VO. [17]

γ΄ Από την άλλη υπάρχει και η περίπτωση ανάθεσης εικονικού αντικειμένου σε επίσης εικονικό αντικείμενου. Εδώ, θα αποθηκευτεί-ανατεθεί ένα *reference* του id του αντικειμένου που θα αποθηκευτεί, μέσα στο αντικείμενο που αποθηκεύεται.

δ΄ Το παράνω φυσικά αναφέραται στην περίπτωση που θέλουμε να αποθηκεύσουμε (store) το αντικείμενο. Στην αντίθετη, δηλαδή όταν θέλουμε να το ανακτήσουμε μέσα από ένα άλλο που είναι επίσης εικονικό (με την εντολή `getfield`), τότε η ανάγνωση αυτή θα οδηγήσει σε ανάθεση ενός καινούργιου alias έτσι ώστε ο καινούργιος κώδικας, με τον οποίο θα αντικατασταθεί ο παλιός, να ξέρει που υπάρχουν τώρα τα δεδομένα που χρειάζεται για την στιγμή που θα χρειαστεί να γίνει το πραγματικό Load στο runtime.



Σχήμα 4.4: (γ') Actions upon assignment/loading of a VO inside another VO. [17]



Σχήμα 4.5: (δ') Actions upon storing a VO inside another VO. [17]

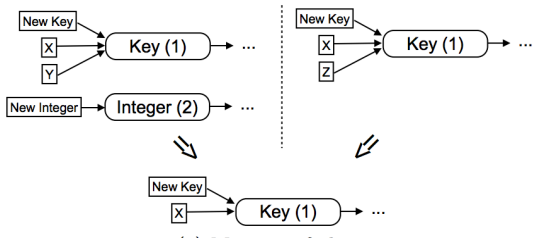
Όλα αυτές οι εντολές θα αφαιρεθούν από τον κώδικα αφού, μόλις είπαμε οι ενέργειες που λαμβάνει ο μεταγλωττιστής όταν τις συναντά αρκούν για να τις κάνουν λειτουργικές και για τα side effects τους. Πολλές άλλες εντολές μπορούν επίσης να αφαιρεθούν βάσει στις πληροφορίες που έχουμε στο states. Tests ισότητας μεταξύ αντικειμένων επιστρέφουν πάντα *false* αν ένα και μόνο ένα από τα αντικείμενα που λάμβάνει το test είναι εικονικό. Εάν είναι και τα δύο εικονικά, τότε θα παραχθεί *true* αν φυσικά αναφέρονται στο ίδιο id, και *false* σε άλλη περίπτωση. (Αυτό το τσεκάρισμα, αργότερα στην υλοποίηση, θα γίνεται κυρίως σε merge nodes όταν έχουμε αποφανθεί ότι το αντικείμενο θα πρέπει να επανατοποθετηθεί στον κώδικα και θα θέλουμε να δοκιμάσουμε αν τα αντικείμενα είναι ένα ή δυο.) Tests τύπων μπορούν να εκτελεστούν και σε αυτό το σημείο (compile time). Αν κάποια εντολή που δεν είναι ειδική για αυτές τις ενέργειες και δεν αναφέρεται εδώ απαιτεί ένα reference σε αντικείμενο το οποίο είναι εικονικό τότε φυσικά αυτό θα πρέπει να επανατοποθετηθεί.

4.5.5 Ειδικές κατηγορίες: Merge

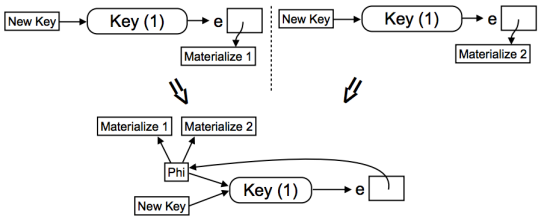
Όταν πολλά branches της ροής θα συναντηθούν σε ένα *mergenode*, πρέπει να παράγουμε ένα state από όσα και τα branches. Αρχικά ο "επεξεργαστής" της συγχώνευσης θα δημιουργήσει μια *τομή* από όλα τα δεδομένα των αντικειμένων state από όλα τα branches. Αυτό σημαίνει ότι τα εικονικά αντικείμενα που θα επιβιώσουν την συγχώνευση είναι αυτά που είναι κοινά σε όλα τα branches της ροής και έχουν τουλάχιστον έναν κοινό alias.

Οι περιπτώσεις που μπορούν να συμβούν ανάλογα με την κατάσταση των αντικειμένων είναι οι εξής (σχεδιαγράμματα στις εικόνες 4.6 και 4.7):

- Αν το αντικείμενο έχει διαφύγει σε όλες τις προηγούμενες καταστάσεις (τις οποίες θέλουμε να συγχωνεύσουμε) τότε φυσικά το αντικείμενο θα διαφεύγει και στο καινούργιο state.
- Αν κάποια από τα αντικείμενα είναι εικονικά και άλλα διαφεύγουν, τότε πρέπει να επανατοποθετηθούν και όσα αντικείμενα είναι εικονικά στα αντίστοιχα states τους.
- Η πιο ειδική περίπτωση είναι όλα τα αντικείμενα να είναι εικονικά. Τότε στο καινούργιο αντικείμενο όλες οι τιμές από όλα τα states πρέπει να συγχωνευτούν. Για κάθε field, ο μεταγλωττιστής θα κοιτάξει και θα συγκρίνει τις εκδόσεις από όλα τα states και:



Σχήμα 4.6: Merging of aliases. [17]



Σχήμα 4.7: Merging of escaped objects. [17]

  - αν όλα τα πεδία είναι πανομοιότυπα και συμφωνούν, τότε το καινούργιο αντικείμενο στο συγχωνευμένο state μπορεί επίσης να είναι εικονικό. Ο μεταγλωττιστής θα δημιουργήσει εδώ ένα καινούργιο `VirtualObject` και θα αντιγράψει τα fields. Τα references θα δείχνουν στα ίδια ids κτλ.
  - αν κάποιο από τα πεδία διαφέρει το αντικείμενο πρέπει να επανατοποθετηθεί, όπως επίσης και τα αντικείμενα για τα οποία περιέχει αναφορές.

Φυσικά αυτή η διαδικασία μπορεί να οδηγήσει στην επανατοποθέτηση αντικειμένων που σημαίνει ότι προηγούμενες υποθέσεις που έχουν γίνει για τον κώδικα δεν είναι πλέον αληθείς άρα θα πρέπει να γίνει επεξεργασία ξανά. Οπότε επαναλαμβάνουμε το παραπάνω μέχρι να μην υπάρχουν αλλαγές.

4.5.6 Ειδικές κατηγορίες: Loops

Οι βρόγχοι (loops) είναι μια πολύ ειδική περίπτωση. Όλοι οι κόμβοι (`Blocks`) που συμμετέχουν σε έναν βρόγχο πρεπεί και θα εξεταστούν με σειριακό τρόπο ακολουθώντας την σειρά ροής, όμως αυτό ενέχει πρόβλημα με τις “ακμές επιστροφής” (back

edges) του βρόγχου. Φυσικά η κανονική ροή εδώ μπορεί να επισκεφτεί κόμβους παραπάνω από μια φορά – αυτό είναι και το νόημα των loops – αλλά εμείς δεν μπορούμε να το έχουμε αυτό κατα το compile time.

Η θεωρία που ακολουθείται στο paper, έχει φυσικά το παραπάνω χαρακτηριστικό αλλά κρίναμε ότι είναι τουλάχιστον μη αποδοτική και δεν την υλοποιήσαμε στο δικό μας σύστημα. Συνοπτικά να πούμε ότι ξεκινάει "μαντεύοντας" ένα state αντικείμενο (το καλούν speculative) βάσει του κόμβου πατέρα του αρχικού κόμβου του βρόγχου και από εκεί συνεχίζουν ακολουθώντας την ροή, σταματώντας μόνο στα back edges και στα exits του loop. Το τελικό προϊόν είναι σωστό και η επεξεργασία συνεχίζεται μόνο αν το αρχικο speculative state ήταν τελικά σωστό. Αν διαφέρουν, η διαδικασία θα ξαναγίνει και το καινούργιο state θα χρησιμοποιηθεί ως speculative state αυτή την φορά.

4.5.7 Πυροδότηση & σύνδεση με άλλους βελτιστοποιητές

Στην θέση της μεταβλητής ή αντικειμένου που αφαιρέθηκε φυσικά θα πρέπει να υπάρξει κάτι το οποίο "διατηρεί" την λογική του προγράμματος ως έχει. Αυτό μπορεί να συμβεί με διάφορους τρόπους. Συνήθως επιτυγχάνεται με αντικατάσταση βαθμωτών στην θέση της μεταβλητής. Δηλαδή το σημείο στο οποίο "ζητούνταν" κάποιου είδους πληροφορία τοποθετείται ένας βαθμωτός, δηλαδή μια απλή "φθηνή" μεταβλητή, που απλά υπάρχει στην στοίβα (*function stack*) της συνάρτησης και όχι στον σωρό (*heap*). Σε αυτόν τον βαθμωτό ο μεταγλωττιστής τοποθετεί την απαραίτητη πληροφορία που χρειάζεται το πρόγραμμα εκείνη την στιγμή και κανονικά θα την εξήγαγε από το αντικείμενο που αφαιρέσαμε. Φυσικά την πληροφορία αυτή ο μεταγλωττιστής την εξάγει επίσης από το αντικείμενο αλλά κατά την ανάλυσή μας (στο *compile-time*) και όχι στο *runtime*. Ο μεταγλωττιστής αποθηκεύει προσωρινά τις πληροφορίες που χρειάζεται για τυχόν υπολογισμούς με τα αντικείμενα (για χρήση στο επόμενο σημείο που θα απαιτηθεί βαθμωτός) στα εικονικά αντικείμενα που αναφέρουμε παραπάνω. Αν το αντικείμενο πρέπει να επανατοποθετηθεί όλοι αυτοί οι βαθμωτοί θα αφαιρεθούν και οι ενέργειες θα αναιρεθούν. Τέλος να πούμε ότι φυσικά πολλές από αυτές τις πράξεις θα μεταφερθούν (*carried on*) μέσω των εικονικών αντικειμένων και των "προσωρινών" πράξεων και τα αποτελέσματά τους απλά θα τοποθετηθούν εκεί που χρειάζονται "πραγματικά" – δηλαδή έξω από το scope της συνάρτησης που αναλύουμε. Μπορεί κανείς να πει ότι αυτό είναι ένα είδος *constant folding*.

Όλοι οι σύγχρονοι μεταγλωττιστές – συμπεριλαμβανομένου του PyPy – δουλεύουν με κάποιου είδους εικονική μηχανή που αναλαμβάνει να τρέξει τον κώδικα και αυτές οι μηχανές, εκτός απο πολλά πλεονεκτήματα ασφαλείας και ευκολίας προγραμματισμού, έχουν φυσικά πολλούς τρόπους βελτιστοποίησης του κώδικα, όπως έχουμε πει. Σε αυτά περιλαμβάνεται και *alias analysis* που εντοπίζει τα επιπλέον ονόματα των μεταβλητών και σίγουρα μειώνει τον φόρτο εργασίας στην μνήμη. Επίσης συμπεριλαμβάνεται πάντα ένας inliner (βλ. παραπάνω παράδειγμα). Η βελτιστοποίηση με την μέθοδο μερικής διαφυγής λοιπόν είναι πολύ πιο αποδοτική από αυτά μόνη της αλλά σίγουρα ακόμα περισσότερο όταν συνδυάζεται με επιπλέον βελτιστοποιήσεις και τροποποιητές του κώδικα όπως το inlining.

Το *inling* είναι η μεταφορά κομματιών κώδικα σε άλλα σημεία έτσι ώστε να αποφευχθεί κάποια κλήση συνάρτησης. Ο μεταγλωττιστής φυσικά προβαίνει σε αυτό όταν αποφανθεί ότι η κλήση αυτή καθ' αυτή – δηλαδή η δημιουργεία stack και ότι άλλο περιλαμβάνει αυτό – είναι ακριβή και δεν αξίζει. Οπότε αντί για κλήση σε συνάρτηση θα μεταφέρει τον κώδικα της συνάρτησης στο σημείο που βρίσκεται η ροή την εκάστοτε στιγμή. Φυσικά πολλές φορές ο μεταγλωττιστής δεν θα το κάνει – υπάρχει λόγος που πολλές φορές ο προγραμματιστής αποφασίζει να χρησιμοποιήσει συναρτήσεις[6] όταν π.χ. η κλήση θα συμβεί πολλές φορές. Εαν προβούμε σε inlining σε αυτή την περίπτωση το μέγεθος του κώδικα θα αυξηθεί δραματικά. Είναι προφανές ότι αυτού του είδους η ενέργεια βοηθά το δικό μας έργο σε τεράστιο βαθμό. Οι μεταβλητές που αποφασίζουμε ότι μπορούν να αφαιρεθούν είναι πολύ περισσότερες μετά το πέρασμα του inliner, καθώς οποιαδήποτε μεταβλητή μπορεί να μεταφερθεί στην συνάρτηση έχει μεταφερθεί. Ο inliner υλοποιείται στο αρχείο `inliner.py`.

Για επιπλέον πρακτικές λεπτομέρειες για το πως υλοποιήσαμε την παραπάνω θεωρία και φυσικά ότι προβλήματα πραγματικού περιβάλλοντος συναντήσαμε κατά το εγχείρημα αυτό βλ. επόμενο κεφάλαιο.

4.6 Παράδειγμα

Εδώ παραθέσουμε ένα εκτενές παράδειγμα, αυτή τη φορά φυσικά για τις διαφορές και λεπτομέρειες της μερικής ανάλυσης με περισσότερη *real-world* λεπτομέρεια από το αντίστοιχο της απλής ανάλυσης, ακολουθώντας φυσικά το παράδειγμα του paper[17] εξιδανικευμένο για την Python. Αυτή την φορά ο αρχικός κώδικας είναι μετά το inlining για να αναδείξουμε καλύτερα τα αποτελέσματα της ανάλυσης διαφυγής.

```
1  ...
2  def getValue(idx, ref):
3      key = Key()
4      key.idx = idx
5      key.ref = ref
6      tmp1 = cacheKey # getting from a global
7      tmp2 = (key.idx == tmp1.idx && key.ref == tmp1.ref)
8      if tmp2:
9          return cacheValue
10     else:
11         cacheKey = key # assigning to a global
12         cacheValue = createValue (...)
13         return cacheValue
14 ...
```

Ξανά δεν δίνουμε τις καθολικές (global) μεταβλητές (βλ. γραμμή 5) και πολλές λεπτομέρειες που θα επιβάρυναν τον κώδικα χωρίς να προσφέρουν επιπλέον πληροφορία. Αναδεικνύουμε παρακάτω πως θα ήταν σε κώδικα Python η βελτιστοποιημένη έκδοση του παραπάνω κώδικα. Επαναλαμβάνουμε ότι αυτή η λειτουργία γίνεται σε γραφήματα ροής.

[6] εκτός από την ευκολότερη διαχείρηση του κώδικα

```
1  ...
2  def getValue(idx, ref):
3      tmp = cacheKey
4      if (key.idx == tmp.idx && key.ref == tmp.ref):
5          return cacheValue
6      else:
7          key = Key()
8          key.idx = idx
9          key.ref = ref
10         cacheKey = key
11         cacheValue = createValue (...)
12         return cacheValue
13 ...
```

Βλέπουμε τον τρόπο που ο βελτιστοποιητής άλλαξε τον κώδικάς μας.

- Στην γραμμή 3 η δέσμευση (η δημιουργία ουσιαστικά του αντικειμένου) έχει αφαιρεθεί. Δημιουργείται εικονικό αντικείμενο.[7]
- Οι αναθέσεις στις γραμμές 4 και 5 έχουν επίσης αφαιρεθεί. Γίνεται "ανάκλαση" των αποτελεσμάτων τους και των side-effect τους στα εικονικά αντικείμενα.
- Οι απόπειρες πρόσβασης σε περιοχές δεσμευμένης μνήμης στην γραμμή 7 έχουν αντικατασταθεί με απλές προσβάσεις σε τοπικές περιοχές.
- Στην περίπτωση του `if` στην γραμμή 8, δημιουργείται ένα επιπλέον αντίγραφο της δομής δεδομένων που διαχειρίζεται τα εικονικά αντικείμενα (*state* – βλ. παρακάτω).
- Στην γραμμή 9 το αντικείμενο είναι ακόμα εικονικό. Λόγω της εντολής `return` η διαδικασία ανάλυσης αυτού του branch (του if) τερματίζει εδώ.
- Η ανάλυση για το δεύτερο branch όμως συνεχίζεται στο *else case* εδώ. Στην στιγμή της γραμμή 11 το αντικείμενο είναι ακόμα εικονικό όμως λόγω της ανάθεσης σε καθολική μεταβλητή θα "διαφύγει". Για να μπορεί να γίνει αυτό όμως, το αντικείμενο θα πρέπει να υπάρχει αυτό καθ' αυτό, οπότε θα πρέπει να τοποθετηθεί στον κώδικα η δέσμευση της μνήμης, η αρχικοποίησή της, η ανάθεση των field σύμφωνα με το εικονικό αντικείμενο (και ότι άλλο υπεισέρχεται στην δημιουργία ενός αντικειμένου). Ονομάζουμε αυτή την διαδικασία *materialization* (βλ. επόμενο κεφάλαιο).
- Οι γραμμές 12 και 13 δεν επηρεάζουν πλέον την κατάσταση του κώδικα – αφού έχουμε ήδη αποφανθεί ότι το αντικείμενο πρέπει να υπάρχει.

Γενικά, όπως βλέπουμε, η δέσμευση και όλες οι ακριβές ενέργειες έχουν τοποθετηθεί ακριβώς εκεί που απαιτούνται και όχι οπουδήποτε αλλού – δηλαδή στο *else case* του *if branch*. Αυτό προφανώς δεν οδηγεί σε μικρότερο αριθμό εντολών `malloc` (ούτε άλλων ακριβών εντολών) αλλά μειώνει τον δυναμικό αριθμό των δεσμεύσεων μνήμης αφού αυτοί θα συμβούν πιο σπάνια δεδομένου ότι βρίσκονται σε ένα από τα *cases* και

[7] Για περισσότερες λεπτομέρειες για τα εικονικά αντικείμενα βλ. προηγούμενο κεφάλαιο

όχι στην κυρίως ροή του κώδικα. Με άλλα λόγια μειώνουν το μέσο κόστος της συνάρτησης, βασιζόμενοι στην μικρότερη συχνότητα με την οποία τρέχουν τα *branches*. Όπως θα δούμε στο κεφάλαιο των μετρήσεων (Κεφ. 5.) αυτό συμβαίνει συχνά.

Κεφάλαιο 5

Υλοποίηση

5.1 Γενικά

Σε αυτό το κεφάλαιο θα αναλύσουμε τις λεπτομέρειες περί της υλοποίησης και τις δυσκολίες που προέκυψαν σε αυτή. Η βάση μας είναι ένα προηγούμενο paper των Standler, Würthinger et al[17] που μελετά, περιγράφει και υλοποιεί την μέθοδο για την γλώσσα Java. Ακολουθήσαμε σε γενικές γραμμές – και ειδικότερα στην αρχή – τις περιγραφές που δίνουν για την υλοποίηση και δομήσαμε την υλοποίησή μας όμοιως. Βέβαια όσο προχωρούσαμε τόσο αποκλίναμε από αυτή την περίπτωση, καθώς εμείς είχαμε να κάνουμε με μια δυναμική γλώσσα σε αντίθεση με την στατική Java. Επιπλέον πολλές λεπτομέρειες έλειπαν από την εργασία.

Ο κυρίως κώδικας (δηλαδή το αρχείο `partial_escape.py`) δίνεται στο παράρτημα Α'. Οι γραμμές κώδικα που εμφανίζονται στο παρακάτω κείμενο αναφέρονται σε αυτόν.

Όπως έχουμε ήδη αναφέρει ο κώδικας του project του PyPy χωρίζεται σε subsystems, και modules. Το module που υλοποιούμε υπόκειται στο σύστημα RPython, στο subsystem του translator, στην ομάδα των backend βελτιστοποιήσεων. Βρίσκεται επομένως στο ακόλουθο μονοπάτι: `pypy/rpython/translator/backendopt/partial_escape.py`. Ο κώδικας βρίσκεται σε ένα fork[8] του επίσημου *repository* του PyPy[11] στη σελίδα bitbucket[1]. Η υλοποίηση έγινε με την βοήθεια του εργαλείου διαχείρησης εκδόσεων κώδικα (SCM) mercurial[12] το οποίο χρησιμοποιείται και εσωτερικά από το project. Φυσικά γίνονται προσπάθειες προκειμένου το module μας να συμπεριληφθεί στο επίσημο repo του PyPy.

Test-driven development

Στην υλοποίηση χρησιμοποιήθηκε η μέθοδος ανάπτυξης λογισμικού βασισμένη σε tests (*test-driven development*[18]). Είναι μια σχετικά καινούργια μέθοδος ανάπτυξης που χρησιμοποιεί test κώδικα για να ελέγξει την ορθότητα του κανονικού κώδικα των προγραμμάτων. Η μέθοδος αναπτύχθηκε φυσικά λόγω της μεγάλης πολυπλοκότητας που έχουν τα διάφορα προγράμματα και συστήματα την σήμερον ημέρα. Ο προγραμματιστής αφού κατανοήσει περίπου πως πρέπει να δομήσει το πρόγραμμα

[1] `http://bitbucket.com`

του ξεκινά με την συγγραφή του test κώδικα. Αρχικά για τις απλές (και ίσως τετριμένες) περιπτώσεις και σταδιακά αυξάνει την πολυπλοκότητα των tests και φυσικά του κώδικα αντιστοίχως. Αφού γράψει ένα ή περισσότερα tests, συγγράφει το πρόγραμμά του προσπαθώντας να κάνει τα tests να "τρέξουν" επιτυχώς. Η διαδικασία συνεχίζεται φυσικά μέχρι να ολοκληρωθεί η διαδικασία ανάπτυξής ή να φτάσει ο προγραμματιστής στην επιθυμητή πολυπλοκότητα. Με αυτόν τον τρόπο – αν προφανώς όλα τα tests είναι επιτυχή – ο προγραμματιστής είναι σίγουρος ότι οι καινούργιες αλλαγες που επέφερε στο project δεν επηρέασαν την ορθότητα του προγράμματος σε κάποιο προηγούμενο στάδιο.

Το PyPy χρησιμοποιεί αυτή την μέθοδο, οπότε και η υλοποίησή μας την ακολουθεί. Ο test κώδικας του module που υλοποιήθηκε βρίσκεται στο ακόλουθο μονοπάτι: `pypy/rpython/translator/backendopt/test/test_partial_escape.py`. Δεν συμπεριλαμβάνεται στην εργασία αυτή για λόγους λακωνικότητας.

5.2 Δομή του κώδικα

5.2.1 Βασική Συνάρτηση

Ο κώδικας είναι σε ένα αρχείο. Χωρίζεται φυσικά σε συναρτήσεις. Η κυρίως συνάρτηση είναι η `partial_escape()`. Αυτή αποτελεί το κυρίως module που θα τρέξει για κάθε γράφημα με σκοπό την βελτιστοποίηση. Το σύστημα θα καλεί την συνάρτηση αυτή δίνοντας της ένα γράφημα την φορά σε ένα loop. Τα διαθέσιμα γραφήματα θα είναι όλες οι συναρτήσεις του μεταγλωττιστή. Το γράφημα μεταβάλλεται *in-place* και το return value είναι ένας μετρητής των `getfield` εντολών που αφαιρέθηκαν. Πρακτικά η μείωση των εντολών `getfield` στο γράφημα είναι ο κύριος στόχος της συνάρτησης καθώς το module μας θα τρέξει μετά το ήδη υπάρχον module για ανάλυση διαφυγής (non- partial escape analysis) στο PyPy. Παρόλα αυτά δεν είναι μόνο αυτή η αλλαγή που επιφέρεται από το module. Σίγουρα αφαιρούνται πολλές εντολές `malloc` – οι οποίες είναι εξαιρετικά ακριβές. Επιπλέον πολλές ακριβές εντολές μετακινούνται σε κάποιο μονοπάτι της ροής εκτέλεσης το οποίο εκτελείται σπάνια ή λιγότερες φορές, οπότε εν δυνάμει είναι σαν να αφαιρούνται.

5.2.2 worklist

Με τον όρο *Worklist* καλούμε την δομή δεδομένων που χρησιμοποιούμε για να μας βοηθά στην επιλογή του επόμενου `Block` προς επεξεργασία. Επιλέξαμε ένα deque[7] (από τα εσωτερικά structures της Python από το module `collections`). Γενικά, στον κώδικα, χρησιμοποιούμε πολύ συχνά constructions της Python προς διευκόλυνσή μας. Ένα deque ή dequeue (*double ended queue*) είναι μια ουρά (queue) που έχει δυνατότητα εισαγωγής και εξαγωγής και από τις δύο άκρες. Η επιλογή του επόμενου `Block` εξαρτάται από πολλούς παράγοντες – η μεγαλύτερη πολυπλοκότητα απαντάται στην περίπτωση των loops, καθώς κάθε `Block` πρέπει να επεξεργαστεί προφανώς μόνο μια φορά και στην σωστή σειρά. Μια τέτοια δομή δεδομένων ήταν η ιδανική για τον

σκοπό μας. Η επεξεργασία του *worklist* γίνεται στο τέλος κάθε iteration επεξεργασίας κάποιου `Block` (στις γραμμές $371 - 381$ του κώδικα.)

5.2.3 Αφαίρεση μεταβλητών και αντικατάσταση με βαθμωτούς

Όταν ο μεταγλωττιστής εντοπίσει μια καινούργια μεταβλητή τότε την αφαιρεί από τον κώδικα, δημιουργεί ένα εικονικό αντικείμενο (βλ. παρακάτω) και την "παρακολουθεί". Δηλαδή ανιχνεύει τις ενέργειες που γίνονται με αυτή, καταγράφει τις εντολές που τρέχουν και τις αντικαθιστά αναλόγως, έτσι ώστε η λογική, η εκτέλεση, και η ορθότητα του προγράμματος να παραμένει ίδια παρόλο που αφαιρέσαμε μια ολόκληρη μεταβλητή. Για παράδειγμα το παρακάτω κομμάτι κώδικα:

```
1 k = Key()
2 k.id = 1
```

θα μεταφραστεί σε:

```
1 k1 = malloc (...)
2  setfield (k1, id, 1)
```

Εκτός από την αφαίρεση της εντολής `malloc` εδώ, θα αφαιρεθεί επίσης και η εντολή `setfield`, καθώς έχει να κάνει με το αντικείμενο που αφαιρείται. Εκτός από την αφαίρεση, η κάθε εντολή έχει φυσικά και συνέπειες για τα εικονικά αντικείμενα. Για παράδειγμα η `setfield` προκαλεί το *set* σε ένα από τα fields του εικονικού αντικειμένου σύμφωνα με τα ορίσματά της. Στο παραπάνω παράδειγμα θα δημιουργηθεί ένα field "id" στο εικονικό αντικείμενο. Επίσης (εκτός από την `malloc` και την `setfield`) υπάρχουν και άλλες εντολές που είτε μεταβάλλονται είτε μεταβάλλουν τα εικονικά αντικείμενα. Η σημαντικότερη από αυτές είναι η εντολή `getfield`. Αυτή η εντολή είναι το σημείο που θα γίνει η πραγματική αντικατάσταση βαθμωτού, καθώς εκεί είναι απαραίτητη η τιμή. Το `getfield` κάνει το access σε κάποιο field μιας μεταβλητής. Εάν εμείς αφαιρέσουμε την μεταβλητή θα πρέπει να αντικαταστήσουμε την τιμή εδώ με κάποιον αντίστοιχο βαθμωτό ή με κάποιον υπολογισμό, σύμφωνα πάντα με τα συμφραζόμενα. Η "πιστοποίηση" για το αν οι εντολές επηρεάζουν τα εικονικά αντικείμενα γίνεται πολύ απλά με έναν έλεγχο για το αν το αντικείμενο υπάρχει μέσα στο *state*. Αν δεν υπάρχει τότε η εντολή έχει να κάνει με κάποιο αντικείμενο που δεν μπορούμε να πειράξουμε είτε με κάποιο που έχει ήδη επανατοποθετηθεί (βλ. επόμενη παράγραφο).

5.2.4 materialization

Με τον όρο *materialization* καλούμε την "επανατοποθέτηση" του αντικειμένου στον κώδικα. Αυτό περιλαμβάνει την επανατοποθέτηση των εντολών για την δέσμευση της μνήμης και οποιασδήποτε αρχικοποίησης ή τυχών casting ή aliasing πρέπει να εφαρμοστεί στο αντικείμενο, καθώς και την "ακύρωση" οποιονδήποτε άλλων ενεργειών

έχουμε ήδη εκτελέσει. Οι κυρίως λόγοι που μπορεί να προκαλέσουν κάποιο αντικείμενο να κάνει materialization είναι η επιστροφή του ως return value ή το globaliazation του, δηλαδή η αλλαγή scope του (η ανάθεσή του, με άλλα λόγια, σε κάποια global μεταβλητή). Οι ενέργειες που πρέπει να γίνουν φυσικά διαφέρουν ανάλογα με το αντικείμενο και ανάλογα με το πόσες εντολές (π.χ. `setfield`) έχουν εκτελεστεί σε αυτό. Οι εντολές για την αρχικοποίησή του (`malloc` και `cast`) θα τοποθετηθούν είτε στο προηγούμενο Block είτε στο τρέχον. Στο τρέχον θα τοποθετηθούν αν ο μεταγλωττιστής ανακαλύψει την ανάγκη για materialization πάνω στην κανονική επεξεργασία των εντολών ενώ στο προηγούμενο αν την ανακαλύψει οπουδήποτε αλλού (π.χ. κατά την επεξεργασία των link arguments).

Η πιο περίπλοκη περίπτωση είναι αυτή που το αντικείμενο περιέχει ένα reference σε άλλο εικονικό αντικείμενο. Αν ένα εικονικό αντικείμενο πρέπει να υλοποιηθεί ξανά (materialization) τότε πρέπει να συμβεί το ίδιο και για όλα τα εικονικά που περιέχει. Αυτό το διαχειριζόμαστε με recursion στην συνάρτηση του materialization, οπότε ακολουθούνται οι ίδιες ενέργειες.

Υπάρχει επίσης μια bypass δυαδική μεταβλητή η οποία μπορεί να εξαναγκάσει το materialization (`must_be_materialized`).

5.2.5 aliasing

Η Python είναι δυναμική οπότε τα εσωτερικά castings των μεταβλητών και τα διαφορετικά ονόματά τους (*aliases*) βρίθουν. Το σύστημά μας έπρεπε να διαχειρίζεται τα castings και να ακολουθεί το σύστημα με τα aliasings που περιγράφεται από το paper έτσι ώστε να καλύψει τις περιπτώσεις όλων των ονομάτων για τις ίδιες μεταβλητές – Η Python πολλές φορές χρησιμοποιεί διαφορετικά ονόματα για την ίδια μεταβλητή προκειμένου να χρησιμοποιήσει λιγότερη μνήμη. Το paper κηρύττει την χρήση ενός δεύτερου dictionary που θα “δείχνει” στο *id* που με την σειρά του θα “δείχνει” στο εικονικό αντικείμενο. Εμείς – αφού έχουμε στην διάθεσή μας τις κατασκευές και ευκολίες που προσφέρει η Python – χρησιμοποιούμε επιπλέον keys (στο ίδιο dictionary) που απλά “δείχνουν” στο ίδιο εικονικό αντικείμενο. Με αυτόν τον τρόπο διατηρούμε κάθε όνομα που υπάρχει για την μεταβλητή και μπορούμε να “ανιχνεύουμε” για αυτά τον κώδικα κατά την μεταγλώττιση. Εκτός από αυτό, διατηρούμε και ένα *set* από τα ίδια κλειδιά/ονόματα μέσα σε κάθε εικονικό αντικείμενο έτσι ώστε να ξέρουμε εύκολα τα κλειδιά όταν πρέπει να επανατοποθετήσουμε το αντικείμενο στον κώδικα (βλ. materialization), αφού προφανώς πρέπει να “υπακούει” και πάλι σε όλα τα ονόματα που είχε. Το *set* αυτό αρχικοποιείται, φυσικά, στην `__init__()` συνάρτηση του `VirtualObject`.

5.2.6 Άλλες συναρτήσεις και κλάσεις

VirtualObject

Η κυρίως κλάση του module. Αντιπροσωπεύει ένα εικονικό αντικείμενο και περιέχει ότι χρειάζεται. Είναι σχετικά απλή καθώς οι περισσότερες διαδικασίες διαχείρησης λαμβάνουν χώρα στις συναρτήσεις. Περιέχει μόνο μια μέθοδο λ.χ. η οποία είναι η `indencical_malloc_args()` και αυτό που κάνει είναι απλά να ελέγχει αν το εκάστοτε στιγμιότυπο στο οποίο τρέχει (*self*) και ένα όρισμά της, έχουν ίδια ορίσματα στην εντολή δέσμευσης μνήμης. Η εντολή δέσμευσης μνήμης για το κάθε αντικείμενο αποθηκεύεται μέσα στο αντίστοιχο εικονικό αντικείμενο πριν το πρώτο αφαιρεθεί από τον κώδικα. Έκτος από αυτό, κάθε τέτοιο εικονικό αντικείμενο αποθηκεύει ένα dictionary με όλα τα πεδία (*fields*) – που μπορεί να περιείχε το αρχικό αντικείμενο, τον τύπο του cast για το malloc του αντικειμένου και ένα σετ με όλα τα aliases (βλ. [που?]) για το αντικείμενο.

get_current_state()

Απλή, βοηθητική συνάρτηση η οποία δέχεται ως όρισμα μια λίστα η οποία περιέχει dual tuples - ζευγάρια δηλαδή, από `Links` και *states*. Αποτελεί μια wrapper συνάρτηση που κάνει μικρότερης σημασίας τετριμμένες εργασίες. Στις περισσότερες φορές το μέγεθος της λίστας είναι 1 (σειριακή περίπτωση) και επιστρέφεται αμέσως το *state*. Σε άλλες καλείται η συνάρτηση `merge()` Δέχεται επίσης την bypass μεταβλητή `must_be_materialized` και την περνάει στις επόμενες συναρτήσεις όπου χρειάζεται.

can_remove()

Απλή, βοηθητική συνάρτηση η οποία χρησιμοποιείται για να κρίνει εάν το δοθέν αντικείμενο μπορεί να αφαιρεθεί με ασφάλεια. Οι έλεγχοι που κάνει είναι δευτερεύοντες, καθώς το αν το αντικείμενο μπορεί να αφαιρεθεί ή όχι είναι το όλο νόημα της βελτιστοιποίησης αυτής. Περιλαμβάνουν τελικούς ελέγχους για το αν το αντικείμενο λ.χ. έχει destructors[2] ή αν είναι υπόκειται σε garbage collection.

remove_virtual_inputargs()

Επίσης απλή, βοηθητική συνάρτηση που εκτελεί ένα είδους “syncing” μεταξύ δύο λιστών. Κάνει ένα iteration στα αντικείμενα του *state* και διαγράφει την αντίστοιχη μεταβλητή στα `inputargs` των `Blocks` και των `Links` για οποιαδήποτε από αυτά υπάρχει. Αυτές είναι πλέον άχρηστες καθώς χρησιμοποιούνταν πριν για να αναφερθούν σε αυτό, ενώ τώρα το αντικείμενο δεν υπάρχει αφού είναι εικονικό.

[2]Εξαιρέσαμε τα αντικείμενα με destructors από την ανάλυση γιατί αυξάνονταν η πολυπλοκότητα χωρίς πολύ κέρδος στον χρόνο και στην μνήμη.

materialize_object()

Μια από τις τρεις δευτέροντες βασικές συναρτήσεις του module. Αναλαμβάνει το materialization του αντικειμένου για το οποίο θα καλεστεί. Δέχεται ως όρισμα το "κλειδί" του αντικειμένου αυτού στο *state*, το ίδιο το *state* και την λίστα με τις εντολές (operations). Στην λίστα αυτή θα προστεθούν οι εντολές που απαιτούνται. Αυτή μεταβάλλεται *in- place* και χρησιμοποιείται από την καλούσα συνάρτηση για να αντικαταστήσει τις εντολές του `Block`.

Λειτουργεί επίσης ως "κριτής" για την συνάρτηση που την κάλεσε, καθώς επιστρέφει `True` ή `False` ανάλογα με το αν η λειτουργία της ήταν επιτυχής.

merge()

Αυτή η συνάρτηση αναλαμβάνει να συγχωνεύσει δύο ή περισσότερα *states* σε ένα ανάλογα με τα εικονικά αντικείμενα και φυσικά το context από τα `Blocks` και τα `Links`. Δέχεται (και αυτή) την λίστα με τα δυαδικά tuples από *states* και `Links`. Καλείται μόνο από την συνάρτηση `get_current_state()` εάν χρειάζεται, δηλαδή σε περιπτώσεις if και loop. Η μεταβλητή `must_be_materialized` κυρίως επηρεάζει την λειτουργία αυτής της συνάρτησης.

Για περισσότερες λεπτομέρειες βλ. παράγραφος Merge.

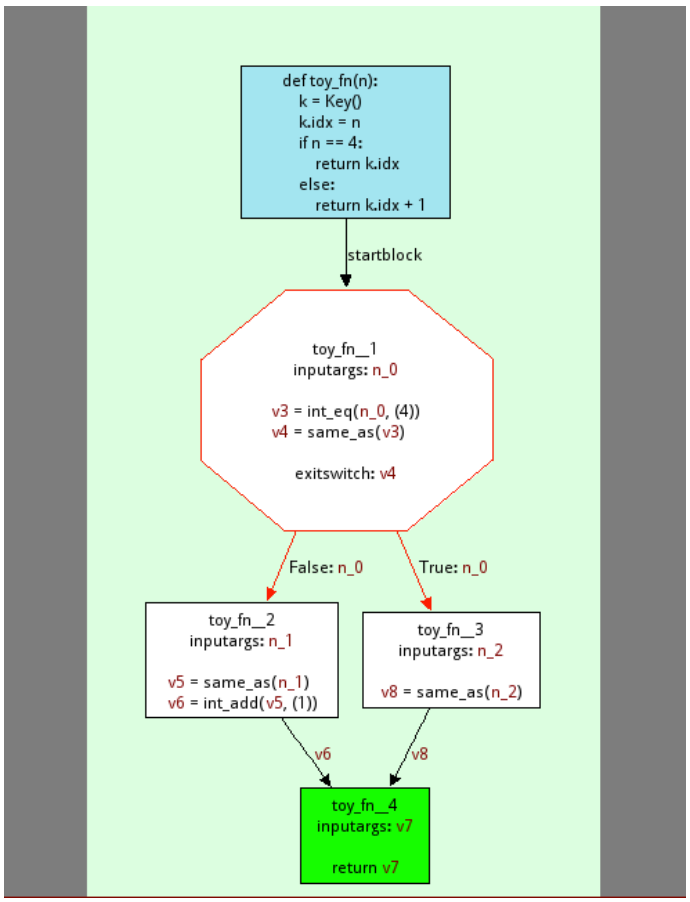
copy_state()

Η τελευταία από τις τρεις δευτέροντες βασικές συναρτήσεις του module. Η λειτουργία της είναι η αντιγραφή του *state* για το επόμενο `Block`. Φυσικά θα εκτελέσει και οποιεσδήποτε άλλες ενέργειες απαιτούνται. Δέχεται προφανώς μια αντικείμενο *state*, και επιπλέον ένα `Link`[3], και την ειδική μεταβλητή `must_be_materialized`. Επιστρέφει ένα καινούργιο αντικείμενο *state* φτιαγμένο για το καινούργιο `Block`. Μπορεί με την πρώτη ματιά η υλοποίησή αυτής της συνάρτησης να φαίνεται τετριμμένη, όμως έκρυβε δυσκολίες. Κατ' αρχάς η αντιγραφή ενός τέτοιο dictionary είναι αυτή καθ' αυτή δύσκολη καθώς τα εικονικά αντικείμενα θα έχουν διαφορετικά ονόματα στο επόμενο `Block` – δηλαδή άλλες μεταβλητές που θα δείχνουν σε αυτά, οπότε πρέπει να αντιστοιχηθούν σωστά. Αυτό συμβαίνει γιατί το σύστημα των γραφημάτων στο PyPy είναι προγραμματισμένο να χρησιμοποιεί καινούργιες μεταβλητές σε κάθε καινούργιο `Block` για λόγους ανεξαρτησίας των `Blocks`. Η αντιστοιχία γίνεται ένα προς ένα μεταξύ των λιστών ενός `Block` και ενός `Link`. Λ.χ. το πρώτο μέλος της λίστας από τα arguments του `Block` θα μπει στην μεταβλητή που υπάρχει επίσης στο πρώτο μέλος της λίστας του `Link`. Φυσικά αυτές οι λίστες θα πρέπει επίσης να είναι "συγχρονισμένες" με το ποια αντικείμενα είναι εικονικά και αυτές οι αλλαγές λαμβάνουν χώρα εδώ. Όταν ένα αντικείμενο υπάρχει στο *state* τότε οποιαδήποτε μεταβλητή αναφέρεται σε αυτό στα arguments θα πρέπει να σβηστεί. Ο κύριως σχεδιασμός της έγινε κατά την φάση υλοποίησης Split (βλ. παρακάτω).

[3]κυρίως απαιτούνται τα arguments και το target `Block`

5.3 Γραφήματα

Η όλη διαδικασία ουσιαστικά αλλάζει το γράφημα που δέχεται ως είσοδο. Αν η συνάρτησή μας δεχτεί ως όρισμα το γράφημα 3.2 τότε θα επιστρέψει το βελτιστοποιημένο ως προς τα `mallocs` γράφημα που φαίνεται παρακάτω στο 5.1.



Σχήμα 5.1: Simple function flow diagram after the optimization took place.

Φυσικά τα γραφήματα ακολουθούν τις τυπικές νόρμες που ακολουθούν όλα τα μοντέλα γραφημάτων όπως *SSA* και το απλό διάγραμμα ροής. Πολλές *προσωρινές-βοηθητικές* μεταβλητές που πρέπει να υπάρχουν στον κώδικα αφαιρούνται γιατί ουσιαστικά μεταβάλλονται σε "κατασκευές" στα γραφήματα κατά το contruction – όπως π.χ. merge `Blocks`. Στα γραφήματα, οι εξαρτήσεις της ροής εκτέλεσης αναπαριστωνται ως έντονα βελάκια που δείχνουν προς τα κάτω όπως επίσης φάινεται στο 5.1.

5.3.1 Πως μεταβάλλουμε τα γραφήματα

Τα γραφήματα εσωτερικά ορίζονται στο `pypy/rpython/flowspace/model.py` και σχεδιάζονται από το εξωτερικό module pygame. Μπορούμε μέσω μιας μεταβλητής graph να βρούμε σειριακά όλα τα `Block` και `Links` και μετά στο μέλος

`operations` του `Block` υπάρχει η λίστα με τις εντολές. Μεταβάλλοντάς την μπορούμε να μεταβάλουμε το `Block` οπότε και το γράφημα. Φυσικά τα παραπάνω είναι υπεραπλουστευμένα, αλλά δίνουν μια καλή εικόνα. Για σωστή διαχείριση απαιτούνται επιπλέον πληροφορίες αλλά το σύστημα του framework του PyPy παρέχει ότι μπορεί να χρειαστεί ο προγραμματιστής. Το `Block` λ.χ. περιέχει επίσης λίστες με τα *exits* του, λίστα με τα ορίσματα που δέχεται[4], βοηθητικές συναρτήσεις (π.χ. `is_final_block()`), τη μεταβλητή exitswitch που χρησιμοποιείται για τα branches (ifs), και τέλος πληροφορίες για την διαχείριση του από το σύστημα του μεταγλωττιστή και από το σύστημα που τα "ζωγραφίζει" στην οθόνη. Τα `Links` περιέχουν τα αντίστοιχα arguments και την μεταβλητή `target` που "δείχνει" στο επόμενο `Block`. Επιπλέον περιέχουν τις ειδικές μεταβλητές `last_exception` & `last_exc_value` που χρησιμοποιεί το μοντέλο για την διαχείριση των εξαιρέσεων και φυσικά όπως και τα `Blocks`, εσωτερικές πληροφορίες. Από τα περιεχόμενα των graph μεταβλητών αυτά που πρέπει να τονίσουμε είναι τα `exceptblock` `returnblock` `startblock`. Τέλος περιέχει τους iterators που χρησιμοποιούμε για να αποκτήσουμε τα `Blocks`.

Να αναφέρουμε επίσης ότι το framework περιλαμβάνει πολλές χρήσιμες συναρτήσεις και κατασκευάσματα που βοηθούν σε μεγάλο βαθμό το έργο του προγραμματιστή. Ακόμα και σε τόσο χαμηλό επίπεδο προγραμματισμού οι βοηθητικές αυτές συναρτήσεις είναι πάντα καλοδεχούμενες και βοηθούν στην ταχύτατη ανάπτυξη. Μερικές από αυτές παραθέτονται παρακάτω. Τις περισσότερες φορές το όνομα τους αρκεί για την κατανόηση. Σε αντίθετη περίπτωση τις εξηγούμε συνοπτικά.

- `mkentrymap()` Δέχεται ένα γράφημα και επιστρέφει ένα λεξικό (*dictionary*). Τα κλειδιά (keys) είναι τα `Blocks` και τα values είναι λίστες με όλα τα `Links` που δείχνουν στο εκάστοτε `Block` του κλειδιού
- `find_backedges()` Όμοιως δέχεται ένα γράφημα και επιστρέφει μια λίστα Python με όλες τις ακμές επιστροφής (*backedges*) του γραφήματος. Ένα back edge σε ένα γράφημα είναι μια ακμή η οποία δείχνει σε έναν προηγούμενο κόμβο ο οποίος έχει ήδη καταγραφεί από τις κανονικές ακμές του δέντρου/γραφήματος.
- `find_loop_blocks()`

Μετά το πέρας της δικιά μας βελτιστοποίησης τρέχουμε επίσης κάποιες συναρτήσεις "καθαρισμού" των `Blocks` έτσι ώστε να αφαιρέσουν τυχόν απομεινάρια εντολών ή εντολές που πλέον δεν απαιτούνται. Για παράδειγμα πολλές φορές μένουν στον κώδικα εντολές `same_as` (μετονομάσιας μεταβλητών) για μεταβλητές που αφαιρέσαμε κατά την βελτιστοποίηση. Αυτές είναι:

- `remove_same_as()` Αφαιρεί τα `same_as`
- `transform_dead_op_vars()` Αφαιρεί μεταβλητές που ορίζονται αλλά δεν χρησιμοποιούνται.

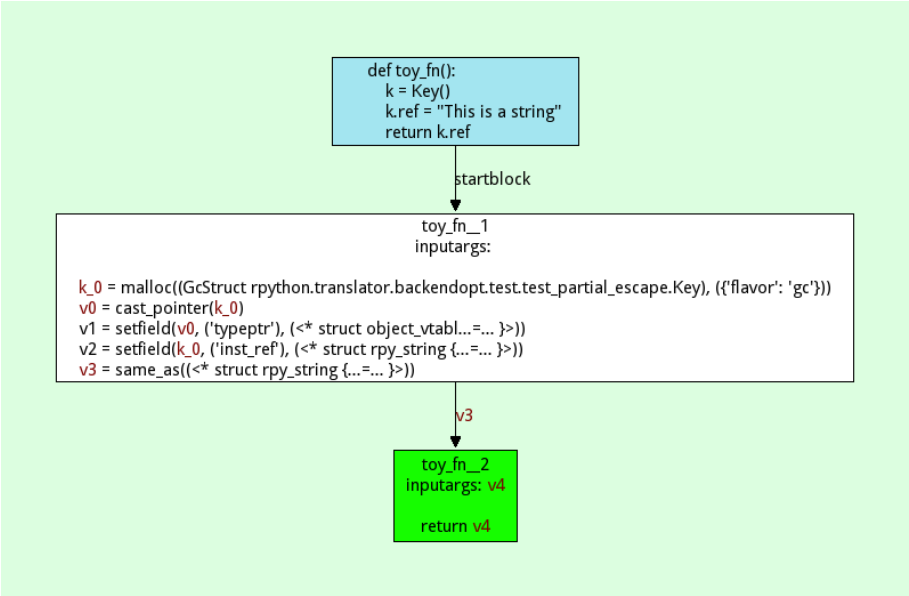
[4]Το κάθε `Block` δέχεται – όπως και οι συναρτήσεις – ορίσματα και "επιστρέφει" μεταβλητές στα επόμενα μέσω των `Links`

5.4 Φάσεις σχεδιασμού

Εδώ θα αναλύσουμε τον τρόπο σκεψης που ακολουθήσαμε για τον σχεδιασμό του module αυτού. Ξεκινήσαμε προφανώς από την απλούστερη περίπτωση και ακολουθώντας φυσικά τον προγραμματισμό με βάση τα tests. Ανεβαίνουμε σε πολυπλοκότητα άπαξ και τα tests είναι επιτυχή, προσθέτοντας κάθε φορά και ένα κομμάτι.

5.4.1 Σειριακά

Αρχικά ξεκινάμε με την περίπτωση που ολόκληρο το σώμα του κώδικα βρίσκεται σε ένα μόνο `Block` και δεν υπάρχει καμία πολυπλοκότητα στην ροή του γραφήματος όπως loops ή if branches. Εδώ στοχεύουμε στο να κατασκευάσουμε την αρχική μορφή και βασική λειτουργία του module μας. Όσο έχει να κάνει δηλαδή με την λειτουργία στην "παρακολούθηση" των μεταβλητών μέσα στα `Blocks`. Αφού τα `Blocks` είναι το κυρίως κομμάτι των γραφημάτων, η ανάλυση μέσα σε αυτά είναι από τα πιο σημαντικά κομμάτια του εγχειρήματός μας. Παρακάτω (στην εικόνα 5.2) δίνουμε πως εμφανίζεται ένα τέτοιο απλό γράφημα. Τα tests περιλάμβαναν περιπτώσεις που κάποιο αντικείμενο έπρεπε να "διαφύγει" – οπότε ουσιαστικά το γράφημα παρέμενε απαράλλακτο, και περιπτώσεις που δεν έπρεπε οπότε αφαιρούνταν.



Σχήμα 5.2: Very simple diagram. Screenshot from the diagram editor.

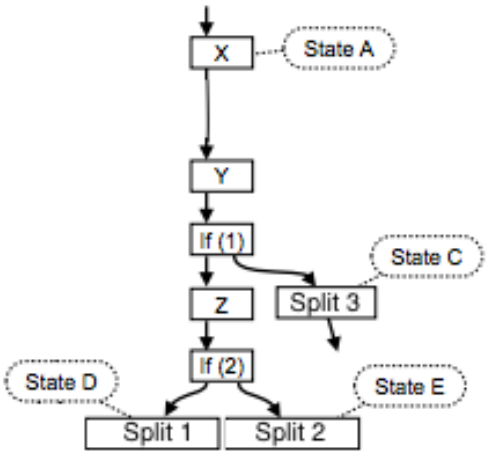
Σε αυτού του είδους τα γραφήματα δημιουργείται μόνο ένα *state*. Το *state* είναι η δομή δεδομένων την οποία χρησιμοποιούμε για να αποθηκεύουμε την κατάσταση των μεταβλητών (όπως λέει και το όνομα) σε κάθε `Block`. Περιέχει δηλαδή τα εικονικά αντικείμενα που αναπαριστούν μεταβλητές. Εάν μια μεταβλητή αναπαριστάται στο *state* αυτό σημαίνει ότι μέχρι τώρα δεν διαφεύγει και έχει αφαιρεθεί από τον κώδικα. Ένα το σύστημα αντιληφθεί ότι διαφεύγει τότε την επανατοποθετεί στον κώδικα (όπως αυτό χρειάζεται βλ. malloc και casts) και την αφαιρεί από το *state*.

Αφού βελτιστοποιήσουμε τις εντολές μέσα στο μοναδικό `Block`, συνεχίζουμε με τις επιπλέον περιπτώσεις. Να σημειώσουμε ότι πιο πριν ξεκινήσαμε χωρίς να πειράξουμε το `Block` καθόλου έτσι ώστε να δούμε αν το σύστημα αλλάζει λόγω side-effects. Φυσικά δεν υπήρχαν.

5.4.2 Split

Εδώ, στις συναρτήσεις (στις οποίες χρησιμοποιούνταν ως tests) είχαμε προσθέσει ένα *if*. Αυτό σημαίνει ότι η ροή θα μπορεί να ακολουθήσει δύο μονοπάτια οπότε θα υπάρχουν παραπάνω από τρία `Blocks`. Εμείς εδώ θα πρέπει να προβλέψουμε το τι συμβαίνει σε όλα τα παρακλάδια της ροής εκτέλεσης. Δημιουργούμε για αυτόν τον λόγο ένα *state* για κάθε `Block`. Αν έχουμε ένα *split*, τότε το *state* του αρχικού `Block` θα αντιγραφεί στα άλλα `Blocks`, και προφανώς το καθένα από αυτά θα ακολουθήσει άλλη πορεία ζωής όταν γίνει η ανάλυση των επόμενων `Block`. (Βλ. σχήμα 5.3 δίπλα.)

Σχήμα 5.3: State split. Screenshot from the diagram editor.



Φυσικά αν υπάρχει *split* από κάποιο `Block` με *if*, τότε θα υπάρχει και *merge* σε κάποιο `Block` καθώς το *returnblock* κάθε γραφήματος είναι μοναδικό. Βέβαια όταν αυτό λαμβάνει χώρα στο *returnblock* δεν μας ενδιαφέρει καθώς το return γίνεται αυτόματα και δεν πρέπει να αλλάξουμε κάτι, οπότε την διαχείριση των *merges* την έχουμε στο επόμενο επίπεδο δυσκολίας.

5.4.3 Merge

Σε αυτό το επίπεδο πολυπλοκότητας, δημιουργήσαμε tests και χειριστήκαμε τις περιπτώσεις όπου τα *merges* απαντώνται σε άλλα `Blocks` πλην του *returnblock*. Έπρεπε να διαχειριστούμε την ένωση δύο *state* μεταβλητών σε ένα. Εδώ ήταν το δυσκολότερο κομμάτι της υλοποίησης καθώς οι λεπτομέρειες βρίθουν. Έπρεπε να ελέγξουμε και να συγκρίνουμε τα δύο *states*. Σε περίπτωση που ένα αντικείμενο υπήρχε και στα δύο, τότε φυσικά έπρεπε να αντιγραφεί στο καινούργιο συγχωνευμένο και να μεταφερθούν όλα τα περιεχόμενά του. Αυτή είναι η περίπτωση που το αντικείμενο δεν έχει επηρεαστεί από το if branch (είτε δεν λάμβανε μέρος σε αυτό) και μπορεί ακόμα να είναι εικονικό. Επίσης για να συμβεί το παραπάνω πρέπει τα ορίσματα της εντολής δέσμευσης του αντικειμένου να είναι ίδια δηλαδή το αντικείμενο να είναι πανομοιότυπο. Σε άλλες περιπτώσεις, δηλαδή όταν το αντικείμενο είναι σε ένα από τα δύο *states* dictionaries (είναι εικονικό σε ένα μόνο branch), τότε το κάνουμε materialization και στα 2 branches, καθώς ο μεταγλωττιστής δεν μπορεί να ξέρει φυσικά σε ποιο θα καταλήξει η ροή εκτέλεσης.

Σαφώς το *merge* δεν λαμβάνει χώρα μόνο σε αυτές τις περιπτώσεις που ξεκίνησαν με if αλλά και μετά από loops.

Να σημειώσουμε επίσης ότι η έκδοση της συνάρτησης που υλοποιεί το *merge* που παραθέτεται σε αυτή την εργασία είναι πολύπλοκη καθώς συμπεριλαμβάνει όλες τις παραπάνω περιπτώσεις καθώς και τις περιπτώσεις των loops και του multi-merge (βλ. επόμενη παράγραφος). Για μια πιο απλή κατανοητή έκδοση ανατρέχουμε στο log του mercurial.

5.4.4 Multi-merge

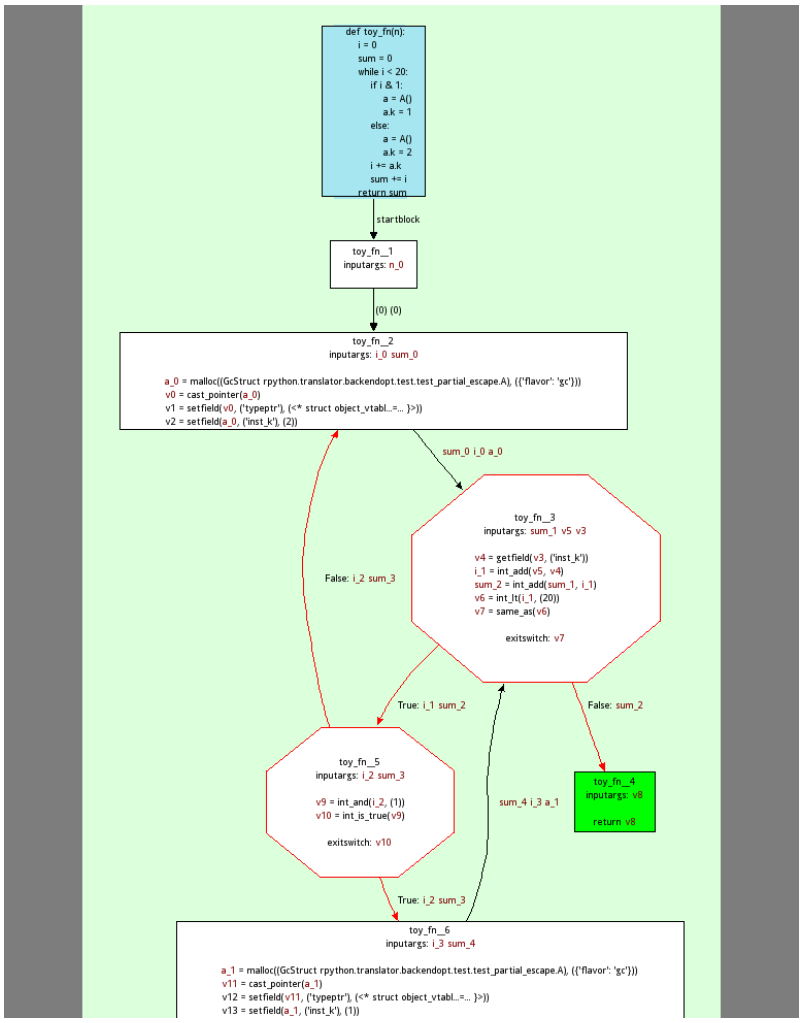
Τα *splits* είναι πάντα δυαδικά. Δηλαδή χωρίζονται σε δύο μόνο. Αν υπάρχει τριπλό if case τότε το ένα από τα `Blocks` ξαναχωρίζεται. Τα *merges* όμως μπορούν να λάβουν χώρα όχι μόνο για δύο αλλά και για τρία και παραπάνω *Block*s σε ένα. Τότε η λογική για την συγχώνευση των αντικειμένων *state* θα πρέπει να τα λαμβάνει υπόψιν της. Χρησιμοποιούμε ένα ζευγάρι για τον αρχικό έλεγχο – το οποίο είναι το αρχικό, μοναδικό ζευγάρι στην περίπτωση απλής συγχώνευσης – και έπειτα εάν αυτός είναι επιτυχής συγκρίνουμε με όσα αντικείμενα απομένουν. Η σύγκριση των ορισμάτων για τον έλεγχο της ομοιότητας λαμβάνει χώρα στην – ορισμένη μέσα στο αντικείμενο `VirtualObject` – μέθοδο `identical_malloc_args()`.

5.4.5 Loops

Η λογική των βρόγχων (loops) ήταν η δυσκολότερη. Μέχρι τώρα δεν έχουμε λάβει υπόψιν μας τα loops στην λογική του προγράμματος. Όταν ο μεταγλωττιστής εντοπίσει μια μεταβλητή που παίρνει μέρος σε ένα loop τότε την εξαναγκάζει να διαφύγει. Ένα μεγάλο κομμάτι της σχεδίασης παρόλα αυτά (ακόμα και χωρίς την κεντρική λογική) έχει υλοποιηθεί, και αυτό είναι το κομμάτι της διαχείρισης της σειράς με την οποία *βλέπουμε* και *επεξεργαζόμαστε* τα `Blocks` (βλ. **worklist**). Στα μελλοντικά μας σχέδια περιλαμβάνεται η πλήρης υλοποίηση της λογικής των loops. Παράδειγμα του πως φαίνεται και υλοποιείται ένα γράφημα με loops δίνεται στο σχήμα 5.4.

5.4.6 Function Calling etc

Τέλος υπάρχουν επιπλέον επίπεδα πολυπλοκότητας που το σύστημα μπορεί να τα διαχειριστεί μόνο του ή πολύ απλά η διαχείρισή τους να είναι τετριμμένη. Λ.χ. η κλήση άλλων συναρτήσεων με την εντολή `direct_call()` μπορεί απλά να μείνει αυτούσια και να μην επηρεάζει την λογική μας. Φυσικά λαμβάνουμε υπόψιν τα ορίσματα που παίρνει και επιστρέφει, καθώς αυτά μπορούν προφανώς να την επηρεάσουν λ.χ. με την χρήση ενός αντικειμένου ως όρισμα.



Σχήμα 5.4: Loop diagram. Screenshot from the diagram editor.

5.5 Γενικά για προβλήματα

Αρχικά να σημειώσουμε ότι δεν αντιμετωπίσαμε πολλά από τα προβλήματα που αναφέρονταν στο paper[17] που βασιζόμαστε, με βασικότερο όλων την έλλειψη μηχανισμών κλειδώματος (locking). Αυτό το αποφύγαμε λόγω του καθολικού κλειδώματος που υπάρχει στην Python και στην πλειοψηφία των υλοποιήσεών της (Global Interpreter Lock – GIL). Το πρόβλημα με αυτή την περίπτωση έγκειτο στο να συνειδητοποιήσουμε ότι δεν απαιτούνται επιπλέον ενέργειες για την κάλυψη του locking.[20] Τα περισσότερα προβλήματα που αντιμετωπίσαμε είχαν να κάνουν κυριώς με περιπτώσεις τις οποίες δεν είχαμε σκεφτεί και δεν είχαμε προβλέψει.

5.5.1 Edge cases

Όπως είπαμε, σε ένα τέτοιο εγχείρημα, ακόμα και με την καλύτερη ανάλυση και τον καλύτερο σχεδιασμό, θα προκύψουν θέματα που ο προγραμματιστής δεν είχε σκεφτεί. Αυτά είναι τα λεγόμενα *edge cases* του σχεδιασμού της λογικής του προγράμματος. Στην προσπάθειά μας για την υλοποίηση του module βρεθήκαμε μπροστά σε πολλές τέτοιες περιπτώσεις. Το μεγαλύτερο παράδειγμα είναι η παρακάτω περίπτωση.

Περίπτωση Αναγκαιότητας Extra Block

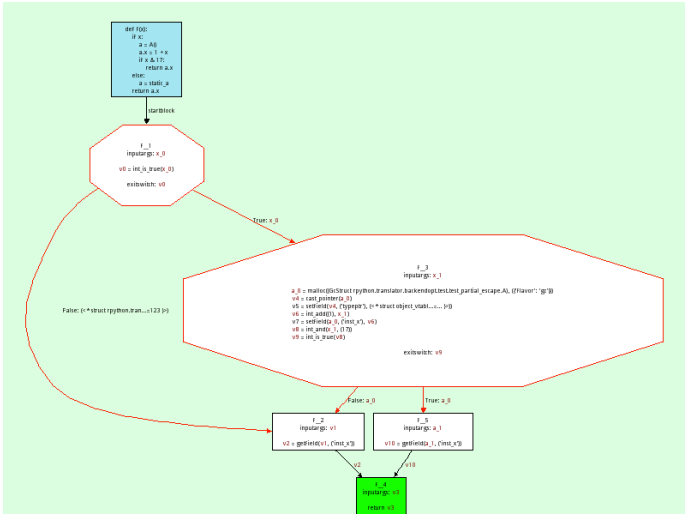
Η περίπτωση αυτή ήταν ίσως η δυσκολότερη στην ανακάλυψή της καθώς δεν οδηγούσε σε *complilation error* ούτε σε *segmentation fault* αλλά μόνο σε μη αποδοτικό κώδικα σε κάποιες πολύ συγκερκιμένες καταστάσεις. Θα εξηγήσουμε το παράδειγμα αυτό και με εικόνες που δείχνουν το πρόβλημα καθώς είναι δύσκολο στην κατανόησή του. Το πρόβλημα αυτό λάμβανε χώρα διότι σε κάποιες περιπτώσεις με branch στην ροή του προγράμματος στις οποίες κανονικά τα αντικείμενα θα έπρεπε να παραμείνουν εικονικά στην μια, αυτά επανατοποθετούνταν στον κώδικα και για τα δύο branches της ροής. Αυτό όπως ανακαλύψαμε έχει να κάνει με την ανικανότητα του μεταγλωττιστή να βρει ένα σωστό σημείο για να επανατοποθετήσει τις εντολές δέσμευσης και αρχικοποίησης του αντικειμένου.

Στο 5.5α΄ φαίνεται ένα διάγραμμα που έχει ακριβώς αυτό το πρόβλημα. Το σημείο που έβρισκε ήταν το `Block` που ήταν ο μόνος κοινός "πρόγονος" του branch, οπότε οι εντολές τοποθετούνταν εκεί, άρα ουσιαστικά το αντικείμενο δεν ήταν εικονικό πλέον σε κανένα από τα branches. Ανάμεσα στο *if* `Block` και στο αριστερό παιδί του απαιτείται ένα επιπλέον `Block` για αυτές τις εντολές διότι το αντικείμενο εκεί θα πρέπει να *υπάρχει* καθώς εκεί καταλήγει και ένα `Link` με μια σταθερά (βλ. το αριστερότερο κόκκινο βελάκι στο 5.5α΄). Από την άλλη όμως, στο δεξιό παιδί το αντικείμενο δεν χρειάζεται να υπάρχει και μπορεί να παραμείνει εικονικό.

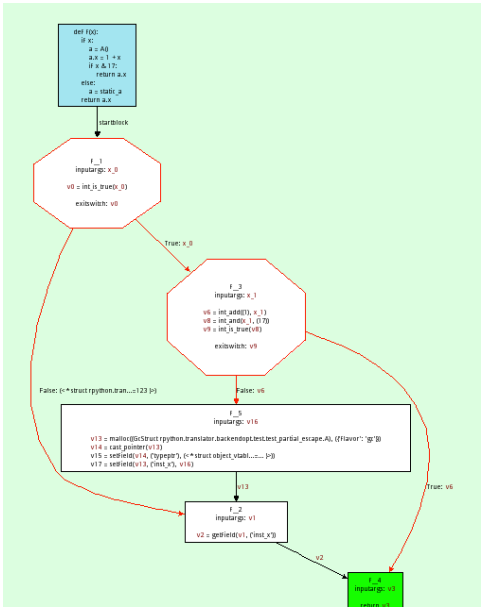
Στο 5.5β΄ φαίνεται το διάγραμμα μετά την λύση που εφαρμόσαμε. Το επιπλέον `Block` έχει τώρα τον κώδικα δέσμευσης, οπότε αυτό οδηγεί στο *materialization* του αντικειμένου *μόνο* στο branch που θέλουμε.

Όπως είπαμε η λύση αυτού του θέματος ήταν πολύπλοκη. Η ανεύρεση των περιπτώσεων που απαιτούνται τα επιπλέον `Blocks` πριν την έναρξη του editing του κάθε `Block` ήταν εξαιρετικά πολύπλοκη. Από την άλλη εάν αποφασίζαμε να αποφανθούμε την ανάγκη των επιπλέον `Blockss` κατά το editing, τότε υπήρχε τεράστιο πρόβλημα στην εισαγωγή του `Block` καθώς αυτό οδηγούσε στην ανάγκη επιπλέον ενεργειών για την επεξεργασία των (ήδη-επεξεργασμένων) `Links` και *arguments*. Οπότε αποφασίσαμε στην τοποθέτηση ενός άδειου `Block` σε κάθε `Link` του διαγράμματος! Αυτό φαίνεται στο διάγραμμα 5.6. Μπορεί με την πρώτη ματιά αυτό να ακούγεται υπερβολικό και καθόλου αποδοτικό, όμως δεν είναι. Όταν η ειδική συνάρτηση[5] – η οποία ανατρέχει το πρόγραμμα πρώτη πριν την έναρξη της ανάλυσης – ανακαλύψει ένα `Link` τότε τοποθετεί ένα άδειο `Block` στο κέντρο και μεταβάλει αναλόγως τα *in & out* `Links` του, έτσι ώστε να "τρέχει" φυσικά στο ενδιάμεσο. Έπειτα η ανάλυση ξεκινά – τα περισσότερα από αυτά τα καινούργια `Blocks` θα παραμείνουν άδεια, και

[5] `(insert_links())`



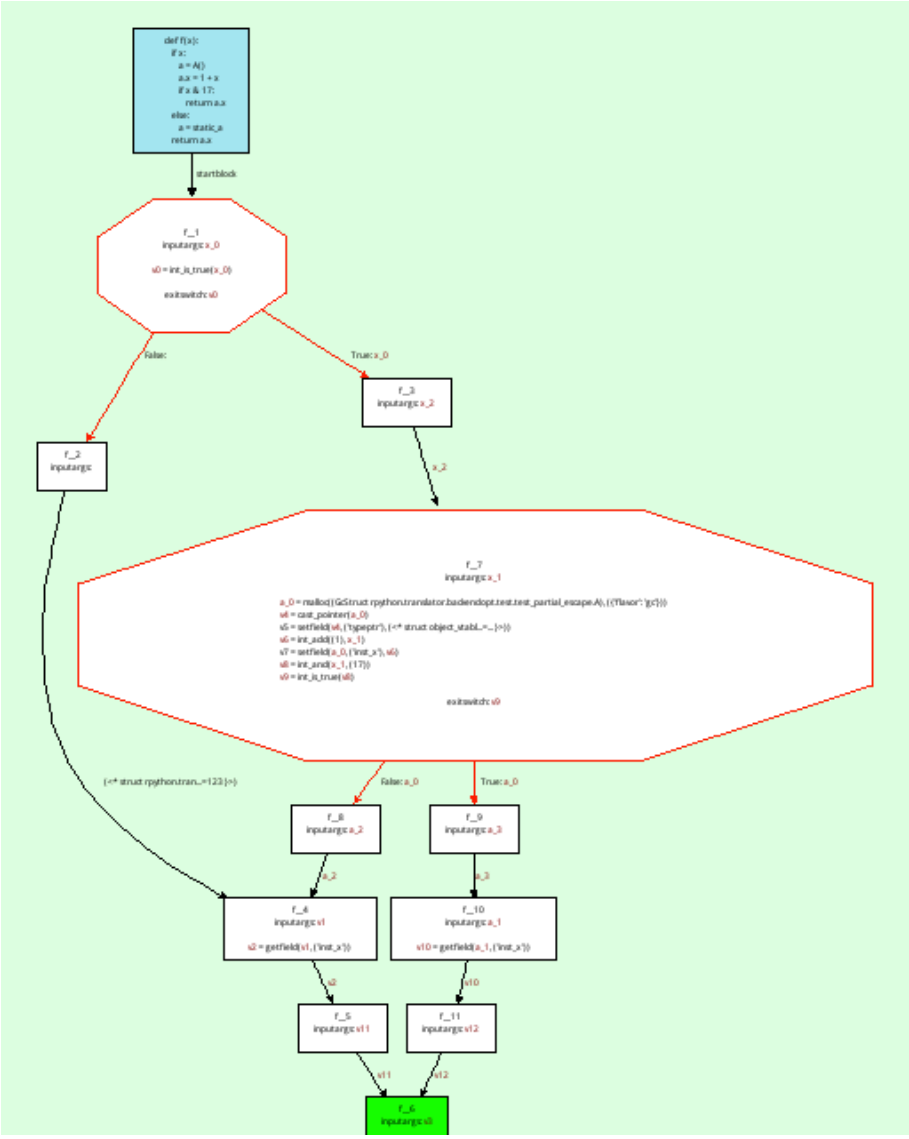
(α΄) Extra Block Problem



(β΄) Extra Block Problem Solution

Σχήμα 5.5: Screenshot from the diagram editor.

στο τέλος θα τρέξει η συνάρτηση *καθαρισμού* `eliminate_empty_blocks()`, η οποία θα κάνει αυτό ακριβώς που αναφέρει το όνομά της – θα αφαιρέσει τα άδεια `Blocks`. Επίσης τρέχουμε την συνάρτηση `join_blocks()` η οποία συγχωνεύει τυχών συνεχόμενα `Blocks` που δεν έχουν λόγο να είναι χωρισμένα (π.χ. *if case*).



Σχήμα 5.6: Extra Block Problem Solution (before cleaning up). Screenshot from the diagram editor.

Κεφάλαιο 6

Αποτελέσματα

Ακολουρούν μετρήσεις που πήραμε από το καινούργιο module todo... (σκέτο, CPython [ίσως Cython], PyPy με partial escape κλπ.)

Η κυρίως καταμέτρηση ήταν αυτή για τις εντολές `getfield`. Στην έκδοση του κώδικα που παραθέτουμε, αυτή γίνεται σε κάθε iteration μέσα στην κεντρική συνάρτηση και ο αριθμός επιστρέφεται. Φυσικά στην τελική έκδοση αυτό το κομμάτι κώδικα θα αφαιρεθεί.

Υπάρχει περίπτωση αύξησης του αριθμού των malloc καθώς ... Για αυτό τον λόγω έχουμε την επόμενη μετρική. φδσφσδ. Θα μετρήσει το μέσο κόστος της κάθε συνάρτησης.

Κεφάλαιο 7

Συμπεράσματα - Μελλοντική Εργασία

Το παρόν εγχείρημα προέκυψε παραγωγικό. Πραγματοποιήσαμε και προσφέραμε στο ευρύ κοινό μια υλοποίηση της μεθόδου ανάλυσης μερικής διαφυγής και βελτιστοποίησης μέσω αντικατάστασης βαθμωτών και συμβάλαμε φυσικά με τον δική μας μικρή βοήθεια στην βελτίωση της ταχύτητας του μεταγλωττιστή του PyPy.

Βλέπουμε από το προηγούμενο κεφάλαιο τα νούμερα. (todo)

Ο μεγάλος όγκος των `mallocs` αφαιρούνται κατά την πρώτη *build- in* βελτιστοποίηση του PyPy (*non-partial escape analysis*[1]). Η δικιά μας μέθοδος πυροδοτείται αργότερα και παρόλα αυτά επιτυγχάνει να αποφύγει περισσότερα `getfields`.

Καταλήξαμε ότι δεν μπορεί να γίνει ανάλυση και υλοποίηση της μεθόδου αυτής για δυναμικές γλώσσες χωρίς να λάβουμε υπόψιν μας το aliasing και τις λεπτομέρειες που επιφέρει. Φυσικά αυτό το πρόβλημα δεν υπάρχει στις στατικές γλώσσες καθώς ο προγραμματιστής φροντίζει για τους τύπους, ενώ στην περίπτωση μας είναι αρμοδιότητα του μεταγλωττιστής μας, οπότε το aliasing των τύπων υπεισέρχεται σχεδόν σε όλη την έκταση των προγραμμάτων.

Μελλοντικές εργασίες περιλαμβάνουν πλήρης υλοποίηση της λογικής των loops καθώς τώρα η υλοποίησή μας τα αγνοεί κάνοντάς τα όλα να διαφύγουν.

[1] escape.py

Παράρτημα Α΄

Κώδικας - Code listing

```
from collections import deque, defaultdict, OrderedDict
import pdb

from rpython.rtyper.lltypesystem import lltype
from rpython.translator.backendopt.removenoops import remove_same_as
from rpython.translator.simplify import (transform_dead_op_vars,
    eliminate_empty_blocks, join_blocks)
from rpython.translator import unsimplify
from rpython.flowspace.model import (SpaceOperation, Constant, Variable, Block,
                                     mkentrymap, checkgraph)
from rpython.translator.backendopt.support import (find_backedges,
                                                   find_loop_blocks)

from rpython.translator.backendopt.support import log


class VirtualObject(object):
    def __init__(self, concretetype, malloc_args):
        # vars maps (variable, concretetype) to the field value
        self.vars = OrderedDict()
        self.malloc_args = malloc_args
        self.concretetype = concretetype
        # a set of variables that all alias the current virtual
        self.aliases = set()

    def identical_malloc_args(self, other):
        if len(self.malloc_args) == 0:
            return other.malloc_args == []
        if len(self.malloc_args) == 2 and len(other.malloc_args) == 2:
            return (self.malloc_args[0] == other.malloc_args[0] and
                    self.malloc_args[1].value == other.malloc_args[1].value)
        return self.malloc_args == other.malloc_args


def materialize_object(obj_key, state, ops):
    """ Accepts a VirtualState object and creates the required operations, for
    its materialization/initialization. XXX: Edits ops in-place
    """

    if obj_key not in state:
        return False

    # We're gonna delete the object from the state dict first (since it has
    # escaped) for correct recursion reasons in case of cyclic dependency.
```

```
45      # this needs to be done with all the aliases of the object!
46      vo = state[obj_key] # Thus, we'll make a copy first.
47      assert obj_key in vo.aliases
48      for key in vo.aliases:
49          del state[key]
50
51      # Starting assembling the operations. Creation and required castings:
52      newvar = Variable()
53      newvar.concretetype = vo.concretetype
54      ops.append(SpaceOperation('malloc', vo.malloc_args, newvar))
55
56      # recreate the aliases
57      for var in vo.aliases:
58          if var.concretetype != vo.concretetype:
59              ops.append(SpaceOperation('cast_pointer', [newvar], var))
60          else:
61              ops.append(SpaceOperation('same_as', [newvar], var))
62
63      #  Initialization
64      for (key, concretetype), value in vo.vars.items():
65          if concretetype != vo.concretetype:
66              # we need a cast_pointer
67              v = Variable()
68              v.concretetype = concretetype
69              op = SpaceOperation('cast_pointer', [newvar], v)
70              ops.append(op)
71              target = v
72          else:
73              target = newvar
74          # What if the assigned is a virtual object? Recursion:
75           materialize_object(value, state, ops)
76          m = Variable()
77          m.concretetype = lltype.Void
78          ops.append(SpaceOperation('setfield', [target,
79                                                 Constant(key, lltype.Void),
80                                                 value], m))
81      return True
82
83
84
85  def copy_state(state, exit, must_be_materialized=False):
86      """ Required function to copy/rename state accordingly for splitting
87      """
88      new_state = {}
89
90      # vars in prevblock -> vars in target
91      copied_vars = {}
92      # map vobj in state -> vobj in new_state
93      copied_vobjs = {}
94
95      sent = exit.args
96      received = exit.target.inputargs
97
98      i = 0
99      while i < len(sent):
100         old_var = sent[i]
101         new_var = received[i]
```

```
102             copied_vars[old_var] = new_var
103             if old_var in state:
104                 if must_be_materialized:
105                     materialize_object(old_var, state, exit.prevblock.operations)
106                 else:
107                     # delete, because they are virtual
108                     del sent[i]
109                     del received[i]
110 
111                     # creating a copy of the object and putting *that* in new_state
112                     old_vobj = state[old_var]
113                     if old_vobj in copied_vobjs:
114                         new_vobj = copied_vobjs[old_vobj]
115                     else:
116                         new_vobj = VirtualObject(
117                             old_vobj.concretetype, old_vobj.malloc_args)
118                         copied_vobjs[old_vobj] = new_vobj
119                         # now need a loop to fill new_vobj.vars
120                         for key, var in old_vobj.vars.iteritems():
121                             if isinstance(var, Variable):
122                                 if var in copied_vars:
123                                     new_vobj.vars[key] = copied_vars[var]
124                                 else:
125                                     var_copy = var.copy()
126                                     copied_vars[var] = var_copy
127                                     new_vobj.vars[key] = var_copy
128                                     # we need them in the sent/received lists
129                                     sent.append(var)
130                                     received.append(var_copy)
131                             else:
132                                 new_vobj.vars[key] = var
133                     new_state[new_var] = new_vobj
134                     new_vobj.aliases.add(new_var)
135 
136             else:
137                 i += 1
138 
139         return new_state
140 
141 
142 def merge(link_state_tuples, orig_must_be_materialized=False):
143         """
144         Function that actually merges states.
145         Objects also get materialized here.
146         """
147         new_state = {}
148 
149         link_state_tuples = link_state_tuples[:] # copy so we can mutate it
150 
151         # We'll have one of the links in a proper variable to "guide" the algorithm
152         # around it. We'll iterate thru the array to mirror the changes to the rest
153         sample_link, sample_state = link_state_tuples.pop()
154         inputargs = sample_link.target.inputargs
155 
156         # the big problem here is aliasing. there can be several variables that
157         # store the same vobj. this mapping must be the same across all links
158         # to keep track, use the following dict, mapping
```

```
159     # vobj -> (vobj1, ..., vobjn)
160     # for each vobj from all the incoming links
161     passed_vobjs = {}
162
163     # map vobj in sample_state -> vobj in new_state
164     merged_vobjs = {}
165
166     inputargsindex = 0
167     while inputargsindex < len(sample_link.args):
168         sample_obj = sample_link.args[inputargsindex]
169         sample_vobj = sample_state.get(sample_obj, None)
170         targ = inputargs[inputargsindex]
171         must_be_materialized = orig_must_be_materialized
172         if sample_vobj is None:
173             must_be_materialized = True
174
175         # set the flag accordingly instead of a huge if clause
176         if not must_be_materialized:
177             vobj_list = [sample_vobj]
178             for lnk, state in link_state_tuples:
179                 obj = lnk.args[inputargsindex]
180                 vobj = state.get(obj)
181                 if vobj is None:
182                     must_be_materialized = True
183                     break
184                 if not sample_vobj.identical_malloc_args(vobj):
185                     must_be_materialized = True
186                     vobj_list.append(vobj)
187             else:
188                 # all the same! check aliasing
189                 for vobj in vobj_list:
190                     if vobj in passed_vobjs:
191                         if passed_vobjs[vobj] != vobj_list:
192                             # different aliasing! too bad
193                             must_be_materialized = True
194                     else:
195                         passed_vobjs[vobj] = vobj_list
196
197         if must_be_materialized:
198             # We can't merge! materialize all objects!
199             changed = materialize_object(sample_obj,
200                                          sample_state,
201                                          sample_link.prevblock.operations)
202             for lnk, state in link_state_tuples:
203                 changed = materialize_object(
204                         lnk.args[inputargsindex], state,
205                         lnk.prevblock.operations) or changed
206             if changed:
207                 # we forced something! that can have all kinds of weird effects
208                 # if the virtual has already been passed to the target block
209                 # earlier. therefore, we restart.
210                 inputargsindex = 0
211                 new_state.clear()
212                 passed_vobjs.clear()
213                 merged_vobjs.clear()
214                 continue
215         else:
```

```
216                 # We can merge: objects are virtual and classes are the same
217                 new_vobj = merged_vobjs.get(sample_vobj)
218                 if new_vobj is None:
219                     new_vobj = VirtualObject(sample_vobj.concretetype,
220                                              sample_vobj.malloc_args)
221                     merged_vobjs[sample_vobj] = new_vobj
222                     for key, v in sample_vobj.vars.iteritems():
223                         m = Variable()
224                         m.concretetype = v.concretetype
225                         inputargs.insert((inputargsindex + 1), m)
226                         sample_link.args.insert((inputargsindex + 1), v)
227                         for lnk, state in link_state_tuples:
228                             vo = state[lnk.args[inputargsindex]]
229                             try:
230                                 newarg = vo.vars[key]
231                             except KeyError:
232                                 # uninitialized field!
233                                 newarg = Constant(
234                                         lltype.nullptr(v.concretetype.TO),
235                                         v.concretetype)
236                             lnk.args.insert((inputargsindex + 1), newarg)
237                         new_vobj.vars[key] = m
238                 new_state[targ] = new_vobj
239                 new_vobj.aliases.add(targ)
240             inputargsindex += 1
241         # safety check: size of state can only shrink
242         vobjset = set(new_state.values())
243         for _, state in link_state_tuples:
244             assert len(vobjset) <= len(set(state.values()))
245         return new_state
246
247
248 def remove_virtual_inputargs(state, link_state_tuples):
249     """Remove all inputargs and the corresponding positions that are in state
250     """
251     inputargs = link_state_tuples[0][0].target.inputargs
252
253     i = 0
254     while i < len(inputargs):
255         if inputargs[i] in state:
256             del inputargs[i]
257             for lnk, _ in link_state_tuples:
258                 del lnk.args[i]
259         else:
260             i += 1
261     return
262
263
264 def get_current_state(link_state_tuples, must_be_materialized=False):
265     """
266     Accepts an array of link-state tuples.
267     Returns the appropriate state dict for every block-editing iteration
268     This is a wrapper around other functions
269     """
270     if len(link_state_tuples) == 1:
271         # Normal block. We make a copy of the state
272         exit, state = link_state_tuples[0]
```

```
273             if exit is None:
274                 # startblock, no need to copy
275                 return state
276             return copy_state(state, exit, must_be_materialized)
277         new = merge(link_state_tuples, must_be_materialized)
278         remove_virtual_inputargs(new, link_state_tuples)
279         return new
280
281
282 def can_remove(op):
283         S = op.args[0].value
284         if op.args[1].value != {'flavor': 'gc'}:
285             return False
286         try:
287             # cannot remove if there is a destructor
288             destr_ptr = lltype.getRuntimeTypeInfo(S)._obj.destructor_funcptr
289             if destr_ptr:
290                 return False
291         except (ValueError, AttributeError):
292             pass
293         return True
294
295
296 def insert_links(graph):
297         # insert a new empty block along every link, as a place to put the forcings
298         for link in list(graph.iterlinks()):
299             unsimplify.insert_empty_block(link)
300
301
302 def partial_escape(translator, graph):
303         """
304         Main function.
305         Blocks, which we'll work on, are in a dequeue, called "worklist", and are
306         indexing link-state tuples in "statemap".
307         """
308         insert_links(graph)
309         worklist = deque([graph.startblock])
310         statemap = defaultdict(list)
311         statemap[graph.startblock] = [(None, {})]
312         finished = set()
313         entrymap = mkentrymap(graph)
314         backedges = find_backedges(graph)
315
316         number_getfield_removed = 0
317
318         while worklist:
319             block = worklist.popleft()
320             must_be_materialized = block.is_final_block()
321             for link in entrymap[block]:
322                 if link in backedges:
323                     must_be_materialized = True
324             state = get_current_state(statemap[block],
325                                       must_be_materialized=must_be_materialized)
326             if block.is_final_block():
327                 continue
328
329             new_operations = []
```

```
330         # Going through the operations
331         for op in block.operations:
332             if op.opname == 'malloc':
333                 # Create new entry for every allocation that is not returned
334                 if can_remove(op):
335                     vobj = VirtualObject(op.result.concretetype, op.args)
336                     state[op.result] = vobj
337                     vobj.aliases.add(op.result)
338                 else:
339                     new_operations.append(op)
340             elif op.opname == 'cast_pointer':
341                 if op.args[0] in state:
342                     # Creating something like an 'alias' for the casting
343                     state[op.result] = vobj = state[op.args[0]]
344                     vobj.aliases.add(op.result)
345                 else:
346                     new_operations.append(op)
347             elif op.opname == 'setfield':
348                 if op.args[0] in state:
349                     state[op.args[0]].vars[op.args[1].value,
350                                            op.args[0].concretetype] = op.args[2]
351                 else:
352                     materialize_object(op.args[2], state, new_operations)
353                     new_operations.append(op)
354             elif op.opname == 'getfield':
355                 key = op.args[1].value, op.args[0].concretetype
356                 if op.args[0] in state and key in state[op.args[0]].vars:
357                     targ = state[op.args[0]].vars[key]
358                     number_getfield_removed += 1
359                     if targ in state:
360                         state[op.result] = vobj = state[targ]
361                         state[targ].aliases.add(vobj)
362                     else:
363                         new_operations.append(SpaceOperation('same_as',
364                                                              [targ],
365                                                              op.result))
366                 else:
367                     materialize_object(op.args[0], state, new_operations)
368                     new_operations.append(op)
369             else:
370                 for arg in op.args:
371                     materialize_object(arg, state, new_operations)
372                 new_operations.append(op)
373         # for all backedges, materialize all arguments (loops aren't supported
374         # properly yet)
375         for exit in block.exits:
376             if exit in backedges or exit.target.is_final_block():
377                 for arg in exit.args:
378                     materialize_object(arg, state, new_operations)
379         block.operations = new_operations
380
381         # We're done with the internals of the block. Editing the lists:
382         finished.add(block)
383         for exit in block.exits:
384             # Only adding to the worklist if all its ancestors are processed
385             for lnk in entrymap[exit.target]:
386                 if lnk.prevblock not in finished and lnk not in backedges:
```

```
387                     break
388                 else:
389                     if exit.target not in finished and exit.target not in worklist: # XXX
390                         worklist.append(exit.target)
391                 # setting statemaps:
392                 statemap[exit.target].append((exit, state))
393         if number_getfield_removed:
394             if translator.config.translation.verbose:
395                 log.cse("partial escape analysis removed %s getfields in graph %s" % (
    number_getfield_removed, graph))
396             else:
397                 log.dot()
398 
399         # Done. Cleaning up.
400         remove_same_as(graph)
401         transform_dead_op_vars(graph)
402         eliminate_empty_blocks(graph)
403         join_blocks(graph)
404         checkgraph(graph)
405 
406         return number_getfield_removed
```

partial_escape.py

Bibliography

[1] Samuele Pedroni (AB Strakt) Armin Rigo (HHU) Michael Hudson (HHU). "D05.1: Compiling Dynamic Language Implementations". In: (2005).

[2] Boehm-Demers-Weiser. URL: http://hboehm.info/gc/ (visited on 08/28/2016).

[3] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pro Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[4] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: ACM Transactions on Programming Languages and Systems (TOPLAS) 13.4 (1991), pp. 451–490.

[5] Saumya Debray. "Abstract interpretation and low-level code optimization". In: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based progr ACM. 1995, pp. 111–121.

[6] Alain Deutsch. "On the Complexity of Escape Analysis". In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langu POPL '97. Paris, France: ACM, 1997, pp. 358–371. ISBN: 0-89791-853-3. DOI: 10.1145/263699.263750. URL: http://doi.acm.org/10.1145/263699.263750.

[7] Double-ended queue. URL: https://en.wikipedia.org/wiki/Double-ended_queue (visited on 09/19/2016).

[8] Fork of official PyPy Repository. URL: https://bitbucket.org/papanikge/pypy (visited on 09/11/2016).

[9] Guido van Rossum. URL: https://www.python.org/~guido/ (visited on 08/27/2016).

[10] Thomas Kotzmann and Hanspeter Mössenböck. "Escape analysis in the context of dynamic compilation and deoptimization". In: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments. ACM. 2005, pp. 111–120.

[11] Official PyPy Repository (BitBucket). URL: https://bitbucket.org/pypy/pypy (visited on 09/11/2016).

[12] Official PyPy Repository (BitBucket). URL: https://www.mercurial-scm.org/ (visited on 09/11/2016).

[13] Premature Optimization. URL: http://c2.com/cgi/wiki?PrematureOptimization (visited on 08/27/2016).

[14] pypy framework project website. URL: http://pypy.org/ (visited on 08/26/2016).

[15] Python Org. URL: https://www.python.org/ (visited on 08/26/2016).
[16] RPython Garbage Collectors. URL: http://rpython.readthedocs.io/en/latest/garbage_collection.html (visited on 08/28/2016).
[17] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. "Partial escape analysis and scalar replacement for Java". In: (2014), p. 165.
[18] Test-driven development. URL: https://en.wikipedia.org/wiki/Test-driven_development (visited on 09/11/2016).
[19] topaz Ruby compiler. URL: https://github.com/topazproject (visited on 08/26/2016).
[20] Understanding the Python GIL. URL: http://www.dabeaz.com/python/UnderstandingGIL.pdf (visited on 09/18/2016).