ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΠΟΔΕΙΞΗ ΟΡΘΟΤΗΤΑΣ ΣΥΛΛΕΚΤΗ ΣΚΟΥΠΙΔΙΩΝ ΑΝΤΙΓΡΑΦΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Ηρόδοτος Δ. Χαριτάτος

Επιβλέπων : Νικόλαος Σ. Παπασπύρου

Επίκουρος Καθηγητής Ε. Μ. Π.

Αθήνα, Δεκέμβριος 2011

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΠΟΔΕΙΞΗ ΟΡΘΟΤΗΤΑΣ ΣΥΛΛΕΚΤΗ ΣΚΟΥΠΙΔΙΩΝ ΑΝΤΙΓΡΑΦΗΣ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αλέξανδρος Ηρόδοτος Δ. Χαριτάτος

Επιβλέπων : Νικόλαος Σ. Παπασπύρου

Επίκουρος Καθηγητής Ε. Μ. Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Δεκεμβρίου 2011.

Αθήνα, Δεκέμβριος 2011

.....	.....	.....
Νικόλαος Παπασπύρου	Νεκτάριος Κοζύρης	Παναγιώτης Τσανάκας
Επίκουρος Καθηγητής Ε. Μ. Π	Αναπληρωτής Καθηγητής Ε. Μ. Π	Καθηγητής Ε. Μ. Π

.....

Αλέξανδρος Ηρόδοτος Δ. Χαριτάτος,

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

Περίληψη

Η διαχείριση της μνήμης που έχει εκχωρηθεί σε ένα πρόγραμμα αποτελεί πολύ σημαντικό κομμάτι του προγράμματος. Με την έννοια της διαχείρισης της μνήμης εννοούμε όλες τις απαραίτητες ενέργειες που πρέπει να γίνουν για να διασφαλιστεί ότι όλα τα αντικείμενα της μνήμης των οποίων ο χρόνος ζωής έχει παρέλθει θα πρέπει να αφαιρεθούν και η μνήμη την οποία καταλάμβαναν να αποδοθεί ελεύθερη προς χρήση. Η διαχείριση αυτή μπορεί να γίνει είτε από τον ίδιον τον προγραμματιστή, πράγμα που συμβαίνει σε γλώσσες όπως η C και η Pascal, είτε από μια διαδικασία η οποία ονομάζεται συλλέκτης σκουπιδιών όπως στις συναρτησιακές γλώσσες προγραμματισμού αλλά και σε ευρέως χρησιμοποιούμενες γλώσσες όπως η Java αλλά και η C#.

Η διαχείριση της μνήμης από τον ίδιον τον προγραμματιστή έχει κάποια σημαντικά μειονεκτήματα όπως ο χρόνος που θα πρέπει να αφιερώσει κατά τη γραφή του πηγαίου κώδικα για να υλοποιήσει την διαχείριση αυτή, ο κώδικας θα γίνει δυσκολότερος στην συντήρησή του αλλά και πιθανώς θα υπάρξουν και λάθη στον κώδικα τα οποία θα οδηγήσουν σε σημαντικές δυσλειτουργίες του προγράμματος. Η διαδικασία της συλλογής σκουπιδιών πραγματοποιείται κατά την ώρα εκτέλεσης του προγράμματος. Την διαδικασία αυτή αναλαμβάνει το ίδιο το πρόγραμμα και ο προγραμματιστής δεν αναμιγνύεται.

Σκοπός της εργασίας αυτής είναι η απόδειξη της ορθότητας της λειτουργίας ενός τέτοιου συλλέκτη σκουπιδιών. Γίνεται εύκολα αντιληπτό πως μια δυσλειτουργία ενός συλλέκτη σκουπιδιών μιας γλώσσας αυτόματα θα περνούσε σε όλα τα προγράμματα που θα παράγονταν από την γλώσσα αυτήν. Ο συλλέκτης σκουπιδιών που επιλέχτηκε για απόδειξη είναι ο semi-space garbage collector.

Για την απόδειξη δημιουργήθηκε ένας συλλέκτης σκουπιδιών semi-space σε γλώσσα ANSI C και χρησιμοποιήθηκαν τα εργαλεία Caduceus και Frama-c.

Λέξεις Κλειδιά

Ορθότητα προγράμματος, συλλέκτης σκουπιδιών, διακοπή-αντιγραφή, Caduceus, Frama-c.

Abstract

The management of memory allocated to a program is a very important part of it. With the term “memory management” one describes all the necessary actions to be taken, which ensure that all memory objects whose lifetime has expired will be removed and the memory that they occupied will be free to use. This management can be performed either manually by the developer, as it happens in languages like C and Pascal, or automatically by a process called the garbage collector, as it happens in functional programming languages but also in widely used languages such as Java and C#.

Manual memory management has some major disadvantages, such as the time spent in writing the source code to implement it. The management of this code is a hard job and it is highly probable that it will contain errors, which will lead to significant failures of the program. The process of garbage collection takes place during program execution. It is executed by the program itself and the programmer does not need to interfere with it.

The purpose of this thesis is to prove the correctness of such a garbage collector. It is easy to understand that a malfunction of a garbage collector in the runtime system of a language would automatically pass to all the programs executed by this language implementation. The garbage collector that we chose to prove correct is the semi-space garbage collector.

For this purpose, a semi-space garbage collector was created in ANSI C and the tools Caduceus and Frama-c were used.

Keywords

Program verification, garbage collector, semi-space, Caduceus, Frama-c.

Ευχαριστίες

Πάντα πίστευα ότι «οι ευχαριστίες» είναι το πιο άχαρο πράγμα που μπορούσε να γράψει κανείς. Τώρα, ξέρω ότι είναι κάτι που πραγματικά βγαίνει από την καρδία αυτού που το γράφει. Και ο, τι βγαίνει από την καρδία δε μπορεί να είναι άχαρο.

Θέλω λοιπόν, να ευχαριστήσω τον καθηγητή κ. Παπασπύρου για την επίβλεψη της εργασίας αλλά και για την για την όλη προσπάθεια που καταβάλει καθημερινά για την εκπαίδευσή μας. Επίσης ευχαριστώ όλου τους ανθρώπους που μου διέθεσαν χώρο και χρόνο για να πραγματοποιήσω το έργο μου, όπως τη σύζυγό μου Ελευθερία, τον αδελφό μου Δημήτρη, τον συνεργάτη μου Κώστα και πολλούς ακόμα, τόσους που θα έπρεπε να καταχραστώ πολύ χρόνο από τον αναγνώστη. Τέλος, τους γονείς μου Δημήτρη και Άρτεμη για την υπομονή και την αγάπη τους.

Αλέξανδρος Ηρόδοτος Χαριτάτος,

Αθήνα, 19η Δεκεμβρίου 2011.

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Δεκέμβριος 2011.

URL: http://www.softlab.ntua.gr/techrep

FTP: ftp://www.softlab.ntua.gr/pub/techrep

Πίνακας Περιεχομένων

Ευρετήριο Πινάκων και Εικόνων

Εισαγωγή

Σκεπτικό της Εργασίας

Η σωστή λειτουργία του προγράμματος είναι κάτι που επιθυμεί ο κάθε προγραμματιστής για το προϊόν του. Δυστυχώς όμως δεν είναι και τόσο εύκολο να επιτευχθεί. Τα περισσότερα προγράμματα ηλεκτρονικών υπολογιστών, που αυτή τη στιγμή ανέρχονται σε εκατομμύρια, έχουν λάθη και δυσλειτουργίες, που προέρχονται είτε από μικρές αβλεψίες κάποιου ανθρώπου είτε από λανθασμένο αρχικό σχεδιασμό.

Προγράμματα ηλεκτρονικών υπολογιστών, πέρα από προσωπικούς οικιακούς υπολογιστές και από μεγάλα δίκτυα όπως αυτά των τραπεζών, χρησιμοποιούνται πια σε σχεδόν σε κάθε συσκευή που κυκλοφορεί, όπως σε βιομηχανικές μονάδες, αυτοκίνητα, οπλικά συστήματα, τηλέφωνα, τηλεοράσεις ακόμα και σε ψυγεία και φούρνους. Είναι εύκολα αντιληπτό ότι μια δυσλειτουργία ενός προγράμματος μπορεί να είναι από μη επιζήμια μέχρι και κρίσιμη και σημαντική για την οικονομία ή ακόμα και την ζωή ανθρώπων.

Επιγραμματικά θα αναφερθούν κάποια τέτοια σφάλματα που συνέβησαν στο παρελθόν και είχαν αποτέλεσμα την απώλεια της ζωής ανθρώπων αλλά και τεράστιων χρηματικών ποσών.

Για παράδειγμα η Therac-25 ήταν μια συσκευή που χρησιμοποιούσε ακτινοβολία για την καταπολέμηση του καρκίνου. Η συσκευή αυτή είχε δύο επίπεδα ακτινοβολίας, χαμηλό και υψηλό. Δυσλειτουργία του προγράμματος σε μερικές περιπτώσεις ενεργοποιούσε την υψηλή αντί της χαμηλής ακτινοβολίας. Το γεγονός αυτό οδήγησε σε τουλάχιστον 6 θανάτους την περίοδο 1985 - 1987.

Επίσης στις 4 Ιουνίου του 1996 η Ευρωπαϊκή Υπηρεσία Διαστήματος εκτόξευσε για πρώτη φορά τον πύραυλο Ariane 5. Η πτήση του πυραύλου διήρκεσε 37 δευτερόλεπτα επειδή υπήρχε σφάλμα στο πρόγραμμα ελέγχου του πυραύλου. Συγκεκριμένα μια μετατροπή από αριθμό κινητής υποδιαστολής 64 bit σε ακέραιο 16 bit ήταν αυτό που προκάλεσε την καταστροφή.

Η ανάγκη της απόδειξης της ορθότητας των προγραμμάτων είναι παραπάνω από προφανής. Ο ορισμός σαφών κριτηρίων που πρέπει να τηρεί ένα πρόγραμμα και η απόδειξη της διασφάλισης τους από αυτό σε επίπεδο πηγαίου κώδικά, μπορεί να οδηγήσει στην εν μέρει αποφυγή τέτοιων σφαλμάτων και δυσλειτουργιών. Εξάλλου "το ίδιο το κείμενο του προγράμματος περιέχει όλες τις συνέπειες λειτουργίας του" (Hoare). Βεβαίως η επιλογή των κριτηρίων πρέπει να είναι τέτοια ,ώστε να μπορεί να διασφαλίζει την επιδιωκόμενη ορθότητα.

Οι ίδιες οι γλώσσες προγραμματισμού αποτελούν προγράμματα και αυτό τις καθιστά ύποπτες ότι θα περιέχουν σφάλματα και δυσλειτουργίες. Επειδή οι γλώσσες προγραμματισμού είναι το εργαλείο με το οποίο παράγεται κάθε πρόγραμμα ηλεκτρονικού υπολογιστή, ένα σφάλμα που μπορεί να περιέχει μια γλώσσα είναι δυνατόν να το "περάσει" σε κάθε προϊόν της δηλαδή σε κάθε πρόγραμμα που παράγεται από αυτήν.

Το σκεπτικό της εργασίας αυτής είναι πως, αν αποδειχτεί η ορθότητα μιας γλώσσας προγραμματισμού, τότε αυτομάτως θα αποδειχτεί και η ορθότητα των προϊόντων της στα κομμάτια που αφορούν στις λειτουργίες τους οι οποίες απορρέουν από την γλώσσα αυτή.

Σκοπός της Εργασίας

Με το σκεπτικό, το οποίο αναπτύχθηκε πιο πάνω, στην παρούσα εργασία θα αποδειχτεί η ορθότητα της λειτουργίας ενός συλλέκτη σκουπιδιών. Η σωστή λειτουργία ενός συλλέκτη σκουπιδιών αφορά σε όλα τα προγράμματα που αποτελούν προϊόντα της γλώσσας που τον χρησιμοποιεί.

Ο συλλέκτης σκουπιδιών είναι μια αυτόματη διεργασία η οποία αναλαμβάνει την απελευθέρωση της μνήμης από όλα εκείνα τα αντικείμενα των οποίων ο κύκλος ζωής έχει τερματιστεί. Αυτά τα αντικείμενα, όσο μένουν στην μνήμη χωρίς να

εκκαθαριστούν, ενώ είναι παντελώς άχρηστα πια, δεσμεύουν μνήμη, εμποδίζοντας νέα αντικείμενα να αποθηκευθούν.

Η διεργασία του συλλέκτη σκουπιδιών εκτελείται από το πρόγραμμα κατά την διάρκεια λειτουργίας του. Όταν το πρόγραμμα προσπαθήσει να δεσμεύσει μνήμη και δε βρει ελεύθερη, τότε η διεργασία του συλλέκτη ξεκινά.

Για την παρούσα εργασία ο συλλέκτης σκουπιδιών που επιλέχθηκε προς απόδειξη είναι ο συλλέκτης αντιγραφής (semi space garbage collector). Ο συλλέκτης αυτός χωρίζει την εκχωρημένη μνήμη σε δύο ίσα μέρη, το ενεργό και το ανενεργό. Όταν το πρόγραμμα αποτύχει στην δέσμευση μνήμης, δηλαδή ο ενεργός χώρος γεμίσει, τότε ο συλλέκτης σκουπιδιών κάνει τον ανενεργό χώρο ενεργό και τανάπαλιν και αντιγράφει όλα τα ενεργά αντικείμενα από την πρώην ενεργό μνήμη στην νυν ενεργό. Μ' αυτόν τον τρόπο όσα αντικείμενα δεν είναι ενεργά δεν μεταφέρονται στην ενεργό μνήμη και ουσιαστικά δεν αποτελούν πια σκουπίδια.

Για να γίνει η απόδειξη της ορθότητας του συλλέκτη αυτού χρησιμοποιήθηκαν τα εργαλεία Frama-C, Jessie, caduceus, why, alt-ergo, simplify, z3, CVC. Η Frama-C είναι ένα επεκτάσιμο εργαλείο στατικής ανάλυσης πηγαίου κώδικα που είναι γραμμένος σε C. Μια από αυτές τις επεκτάσεις είναι η Jessie. Η επέκταση Jessie, χρησιμοποιώντας εσωτερικά τις γλώσσες και τα εργαλεία της πλατφόρμας WHY, επιτρέπει την επαλήθευση προγραμμάτων που είναι γραμμένα σε γλώσσα C, τα οποία περιέχουν σχόλια ACSL(ANSI/ISO C Specification Language). Οι προς επαλήθευση συνθήκες που δημιουργούνται υποβάλλονται σε αυτόματες μηχανές απόδειξης θεωρημάτων όπως τα alt-ergo, simplify, z3, CVC.Ο πηγαίος κώδικας μαζί με τα σχόλια της ACSL μεταφράζονται σε κώδικα WHY, ο οποίος με την σειρά του μεταφράζεται σε κώδικα κατάλληλο για κάθε αυτόματη μηχανή απόδειξης θεωρημάτων.

Για την απλοποίηση του επιδιωκόμενου στόχου γράφτηκε κώδικας συλλέκτη σκουπιδιών αντιγραφής σε γλώσσα C. Ο κώδικας αυτός είναι μια απλοποιημένη μορφή του αντίστοιχου συλλέκτη που υπάρχει στο mmtk του jikes.

Συλλέκτης Σκουπιδιών Αντιγραφής

Συλλέκτες σκουπιδιών γενικά

Η συλλογή σκουπιδιών είναι μια τεχνική που ξεκίνησε να αναπτύσσεται στα τέλη της δεκαετίας του 1950. Ο John McCarthy προσπαθώντας να επιλύσει προβλήματα στη γλώσσα LISP, ήταν από τους πρώτους που ασχολήθηκαν με αυτό το θέμα. Έκτοτε αναπτύχθηκαν αρκετοί αλγόριθμοι για την υλοποίηση συλλέκτη σκουπιδιών όπως αλγόριθμοι καταμέτρησης αναφορών, εκκαθάρισης με σήμανση (Mark – Sweep), συλλογής με αντιγραφή, κ.α.. Συλλέκτες σκουπιδιών είναι ιδιαίτερα δημοφιλείς στις συναρτησιακές γλώσσες προγραμματισμού όπως Haskell, Erlang, ML, αλλά χρησιμοποιούνται και σε άλλες γλώσσες υψηλού επιπέδου όπως Java και C#.

Ο συλλέκτης σκουπιδιών οφείλει να απελευθερώνει την μνήμη από όλα τα σκουπίδια, όλα δηλαδή εκείνα τα αντικείμενα τα οποία δε χρησιμοποιούνται από το πρόγραμμα. Για να το κάνει αυτό θα πρέπει πρώτα να αποφασίσει ποια αντικείμενα

είναι ενεργά και ποια είναι σκουπίδια. Ενεργά αντικείμενα είναι όλα αυτά τα αντικείμενα για τα οποία υπάρχει έστω και ένας δείκτης προς αυτά. Όταν ένα αντικείμενο δεν έχει κάποιον δείκτη προς αυτό, τότε αυτό το αντικείμενο δεν είναι πια προσβάσιμο από το πρόγραμμα και επομένως είναι σκουπίδι. Στην περίπτωση που υπάρχουν δύο ή παραπάνω αντικείμενα που δημιουργούν κυκλική αναφορά, δηλαδή ο δείκτης του ενός δείχνει στο επόμενο και του τελευταίου στο πρώτο, αλλά δεν υπάρχει κάποιος δείκτης από αντικείμενο έξω από τον κύκλο να δείχνει σε ένα από αυτά τα αντικείμενα, τότε όλα αυτά τα αντικείμενα είναι σκουπίδια.

Ένας πιο πλήρης ορισμός του πότε ένα αντικείμενο είναι ενεργό, είναι όταν το αντικείμενο αυτό μπορεί να προσπελαστεί ακολουθώντας μια ‘‘αλυσίδα’’ έγκυρων δεικτών στο σωρό, η οποία ξεκινάει από τις λεγόμενες ρίζες (roots) του προγράμματος. Οι ρίζες αυτές πρέπει να βρίσκονται έξω από τον σωρό. Ειδικότερα, ως ρίζες ορίζονται οι τιμές εκείνες στις οποίες έχει άμεση πρόσβαση ένα πρόγραμμα. Οι τιμές αυτές βρίσκονται στους καταχωρητές (registers), στη στοίβα του προγράμματος (π.χ. προσωρινές μεταβλητές) και στις ολικές (global) μεταβλητές. Προφανώς τα υπόλοιπα αντικείμενα, τα οποία δεν μπορούν να προσπελαστούν από καμία από τις παραπάνω αλυσίδες δεικτών, είναι σκουπίδια[1].

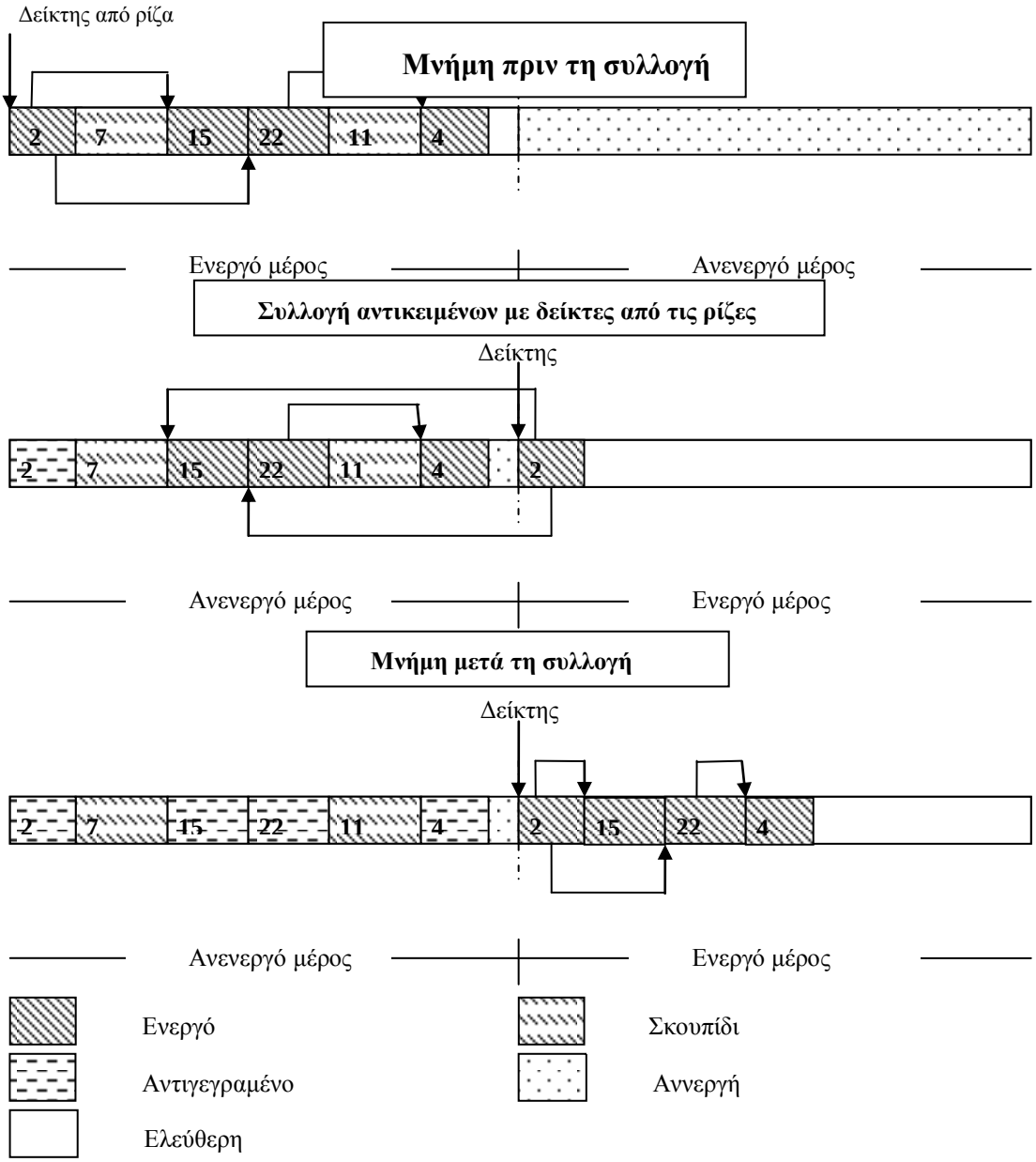
Συλλέκτης σκουπιδιών αντιγραφής

Το 1963 ο Marvin Minsky παρουσίασε την μέθοδο της αντιγραφής. Η μέθοδος αυτή καλύπτει την αδυναμία, που διαπίστωσε το ίδιο έτος ο Harold McBeth, την οποία έχουν οι αλγόριθμοι καταμέτρησης αναφορών στην ανάκτηση κυκλικών δομών. Η μέθοδος της αντιγραφής εξελίχθηκε από τον C.J. Chimney, ο οποίος το 1970 παρουσίασε τον μέχρι και σήμερα πιο διαδεδομένο αλγόριθμο για την μέθοδο της αντιγραφής.

Σύμφωνα με τον αλγόριθμο αντιγραφής η μνήμη χωρίζεται σε δύο ίσα μέρη. Κάθε στιγμή μόνο το ένα μέρος της μνήμης είναι ενεργό. Το ενεργό κομμάτι της μνήμης είναι αυτό που ικανοποιεί τις ανάγκες του προγράμματος σε μνήμη. Όταν εξαντληθεί ο χώρος στη μνήμη τότε καλείται η διεργασία του συλλέκτη. Αρχικά η ανενεργή μνήμη γίνεται ενεργή και τανάπαλιν. Εξετάζονται οι ρίζες για ενεργούς δείκτες προς την μνήμη (που τώρα είναι ανενεργή). Εάν υπάρχουν τα αντικείμενα στα οποία δείχνουν οι δείκτες αυτοί, αντιγράφονται στην ενεργή μνήμη. Στη συνέχεια τα αντιγεγραμμένα αντικείμενα εξετάζονται με την σειρά τους για δείκτες και ούτω καθ’ εξής. Με αυτόν τον τρόπο όλα τα ενεργά αντικείμενα βρίσκονται στην ενεργή μνήμη, ενώ όλα τα

[1]Συλλέκτης Σκουπιδιών σε Συστήματα Πιστοποιημένου Κώδικα, Διπλωματική εργασία Ευθυμίου Γ. Βερβαινιώτη, ΕΜΠ 2011

σκουπίδια αφού δεν έχουν προσπελαστεί δεν έχουν αντιγραφεί και επομένως η μνήμη που καταλάμβαναν απελευθερώθηκε.



Εικόνα 1 Διαδικασία συλλογής σκουπιδιών αντιγραφής

Ένα σημείο που πρέπει να προσεχθεί και θα φανεί χρήσιμο στην απόδειξη που επιχειρείται, είναι ότι όλα τα αντικείμενα της μνήμης είναι πάντα σε σειρά χωρίς να υπάρχουν κενά ανάμεσά τους

Η Λογική Hoare

Γενικά

Η λογική Hoare είναι ένα τυπικό σύστημα με μια σειρά από λογικούς κανόνες που επιτρέπουν την απόφαση για την ορθότητα ενός προγράμματος. Προτάθηκε το 1969 από τον Βρετανό επιστήμονα C.A.R. Hoare, ενώ σχετίζεται με παλαιότερη εργασία του Robert Floyd, η οποία περιέγραφε ένα ανάλογο σύστημα για διαγράμματα ροής. Η λογική Hoare αποτελεί το κανονικό παράδειγμα αξιωματικής σημασιολογίας.

Η βασική έννοια της λογικής Hoare είναι η τριάδα Hoare. Η τριάδα Hoare περιγράφει τον τρόπο με τον οποίον η εκτέλεση ενός κομματιού κώδικα επηρεάζει την κατάσταση του προγράμματος. Οι τριάδες Hoare είναι της μορφής {P } S {Q}, όπου P η προσυνθήκη Q η μετασυνθήκη και S ένα πρόγραμμα ή τμήμα προγράμματος. Η προσυνθήκη και η μετασυνθήκη αποτελούν κατηγορήματα στην κατάσταση του προγράμματος, και μπορούν να εμφανίζονται σε αυτές οι μεταβλητές του. Μια τριάδα Hoare αποτελεί μια προδιαγραφή για τη λειτουργία του προγράμματος. Εφόσον η τριάδα είναι αληθής, δηλαδή η προδιαγραφή ικανοποιείται αν η αρχική κατάσταση

ικανοποιεί την προσυνθήκη, τότε μετά την εκτέλεση του προγράμματος η κατάσταση εκτέλεσης θα ικανοποιεί τη μετασυνθήκη.[2]

Κανόνες

Στην λογική Hoare χρησιμοποιείται μια απλή προστακτική γλώσσα. Στη συνέχεια θα παρατεθούν διάφοροι κανόνες και αξιώματα συμπερασμού.

Αξίωμα κενής εντολής

{P} skip {P}

Η εντολή skip δεν μεταβάλει την κατάσταση του προγράμματος, άρα η μετασυνθήκη θα είναι ίδια με την προσυνθήκη.

Κανόνας ανάθεσης

{P[E/V]} V := E {P}.

Ο συμβολισμός {P [E/V]} δηλώνει αντικατάσταση των ελεύθερων εμφανίσεων της μεταβλητής V με την έκφραση E στη συνθήκη P . Για παράδειγμα είναι έγκυρη η τριάδα {x = 18} x = 24 + x {x = 42}, αφού αντικαθιστώντας στη μετασυνθήκη το χ με χ+24 προκύπτει η προσυνθήκη.

Ο κανόνας της σύνθεσης

$$\frac{\{P\}\ S_1\ \{R\},\ \{R\}\ S_2\{Q\}}{\{P\}\ S_1; S_2\ \{Q\}}$$

Σύμφωνα με τον κανόνα αυτόν για εντολές ή προγράμματα, τα οποία εκτελούνται διαδοχικά, η μετασυνθήκη της πρώτης αποτελεί την προσυνθήκη της δεύτερης. Σε αυτή την περίπτωση η τριάδα είναι αληθής με την προσυνθήκη του πρώτου προγράμματος και την μετασυνθήκη του δεύτερου.

Ο κανόνας του if

$$\frac{\{P \wedge C\}\ S_1\{Q\}, \{P \wedge \neg C\}\ S_2\ \{Q\}}{\{P\}\ \text{if C then } S_1 \text{ then } S_2\ \{Q\}}$$

Σύμφωνα με τον κανόνα αυτό η μετασυνθήκη θα πρέπει να ισχύει ανεξαρτήτως πιο σκέλος του if εκτελεστεί.

Ο κανόνας του while

---

[2] Μηχανική Επαλήθευση Προστακτικών Προγραμμάτων, Διπλωματική Εργασία Βασίλειος Παπαβασιλείου, ΕΜΠ 2009

$$\frac{\{P \wedge C\}\ S\ \{P\}}{\{P\}\ \text{while } C \text{ do } S\ \{P \wedge \neg C\}}$$

Σύμφωνα με αυτόν τον κανόνα το κομμάτι του κώδικα S δε θα εκτελεστεί αν δεν ισχύει η προσυνθήκη C. Η συνθήκη P λέγεται αναλλοίωτη και ισχύει πριν, κατά την διάρκεια και μετά τον τερματισμό του βρόχου.

Ο κανόνας της ενδυνάμωσης της προσυνθήκης

$$\frac{P \Rightarrow R, \{R\}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

Με τον κανόνα αυτόν επιτρέπεται η αντικατάσταση μιας προσυνθήκης με κάποια πιο ισχυρή.

Ο κανόνας της αποδυνάμωσης της μετασυνθήκης.

$$\frac{\{P\}\ C\ \{R\},\ R \Rightarrow Q}{\{P\}\ C\ \{Q\}}$$

Αντίστοιχα με τον προηγούμενο κανόνα ο κανόνας αυτός επιτρέπει την αντικατάσταση κάποιας μετασυνθήκης με κάποια πιο ασθενή

Ο κανόνας της σύζευξης

$$\frac{\{P_1\}\ C\ \{Q_1\}\quad \{P_2\}\ C\ \{Q_2\}}{(\qquad)}$$

Σύμφωνα με αυτόν τον κανόνα αν είναι αληθείς δύο τριάδες με κοινό πρόγραμμα C τότε θα ισχύει και η σύζευξη των προ και μετασυνθηκών τους.

Παράδειγμα λογικής Hoare

Παρακάτω θα δειχθεί η εφαρμογή της λογικής πάνω σε ένα βρόχο while. Θα φανεί πως τοποθετούνται οι προσυνθήκες, οι αναλλοίωτες και οι μετασυνθήκες.

Έστω το πρόγραμμα

```
j := 0;
s := 0;
while (j < N) do
  s := s + a[j];
  j := j + 1;
```

```
end
```

Το πρόγραμμα αυτό έχει σαν αποτέλεσμα το άθροισμα όλων των αριθμών από 0 έως N που περιέχονται στον πίνακα a. Προφανώς η προσυνθήκη είναι N>0 και η μετασυνθήκη s = (Σi | 0≤i<N • a[i]).

Σε έναν βρόχο το σημαντικότερο πρόβλημα είναι να βρεθεί η κατάλληλη αναλλοίωτη. Ένας βασικός κανόνας για τον ορισμό της κατάλληλης αναλλοίωτης είναι ότι αυτή θα πρέπει να έχει την ίδια φόρμα με την μετασυνθήκη. Ένας δεύτερος αφορά στην ίδια τη συνθήκη τερματισμού, που η εφαρμογή του στο παράδειγμά μας είναι 0 <= j <= N. Με αυτό το σκεπτικό τελικά η αναλλοίωτη του βρόχου θα είναι { 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }

```
{N≥0}
j := 0;
{N≥0 && j = 0}
s := 0;
{N≥0 && j = 0 && s = 0}
(loop invariant){ 0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
while (j < N) do
(invariant) {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) && j < N}
s := s + a[j];
j := j + 1;
(invariant) {0 ≤ j ≤ N && s = (Σi | 0≤i<j • a[i]) }
end
{ s = (Σi | 0≤i<N • a[i]) }
```

Στο παραπάνω πρόγραμμα έγινε μια παραδοχή για λόγους απλότητας. Η παραδοχή αυτή είναι ότι ο πίνακας a θεωρείται έγκυρος και ότι έχει μέγεθος τουλάχιστον ίσο με N - 1. Η παραδοχή αυτή, αν και σ' αυτό το παράδειγμα έγινε για λόγους απλότητας, είναι μια σημαντική παράλειψη, που θα πρέπει να λαμβάνεται υπόψη όποιου προσπαθήσει να κάνει μια ολοκληρωμένη απόδειξη.

Οι προσυνθήκες, οι αναλλοίωτες και οι μετασυνθήκες τοποθετήθηκαν ανάμεσα στον πηγαίο κώδικα. Αυτόν τον τρόπο της τοποθέτησης τον χρησιμοποιούν και οι γλώσσες προδιαγραφών (ACSL, JML).

Η Γλώσσα ACSL

Τα εργαλεία Frama-C, Jessie και WHY

Η Frama-C είναι μια επεκτάσιμη σουΐτα εργαλείων με σκοπό την στατική ανάλυση πηγαίου κώδικα ο οποίος είναι γραμμένος σε γλώσσα C. Η δυνατότητα επεκτάσεων επιτρέπει στους χρήστες να χρησιμοποιήσουν για την ανάλυσή τους έτοιμα αποτελέσματα που έχουν βγει από άλλους χρήστες.

Με την έννοια στατική ανάλυση νοείται η ανάλυση και η συλλογή πληροφοριών για τον τρόπο που θα εκτελεστεί ο πηγαίος κώδικας, πριν αυτός το κάνει. Στόχος της στατικής ανάλυσης, όπως περιγράφεται από τους δημιουργούς του εργαλείου, είναι η εύρεση των σφαλμάτων ενός προγράμματος πριν αυτά γίνουν. Σε αντίθεση με εργαλεία που χρησιμοποιούν ευριστικές μεθόδους για τον εντοπισμό σφαλμάτων, η Frama-C προτρέπει τον αναλυτή να ορίσει τις προδιαγραφές λειτουργίας και να αποδείξει ότι αυτές τηρούνται.

Η επέκταση Jessie της Frama-C είναι μια υλοποίηση της γλώσσας ACSL. Η γλώσσα αυτή είναι εμπνευσμένη από την JML, η οποία χρησιμοποιείται για την

απόδειξη ορθότητας προγραμμάτων, που είναι γραμμένα σε JAVA. Η ACSL είναι μια γλώσσα προδιαγραφών για προγράμματα γραμμένα σε γλώσσα C. Στηρίζεται στη λογική Hoare των προσυνθηκών, αναλλοίωτων και μετασυνθηκών. Οι προδιαγραφές γράφονται σαν σχόλια της γλώσσας, επιτρέποντας το αρχικό πρόγραμμα να περάσει από οποιονδήποτε μεταγλωττιστή.

Το εργαλείο Jessie μετατρέπει τον κώδικα και τα σχόλια που αποτελούν τις προδιαγραφές σε γλώσσα κατάλληλη για το εργαλείο WHY. Το WHY είναι ένα εργαλείο επαλήθευσης λογισμικού.

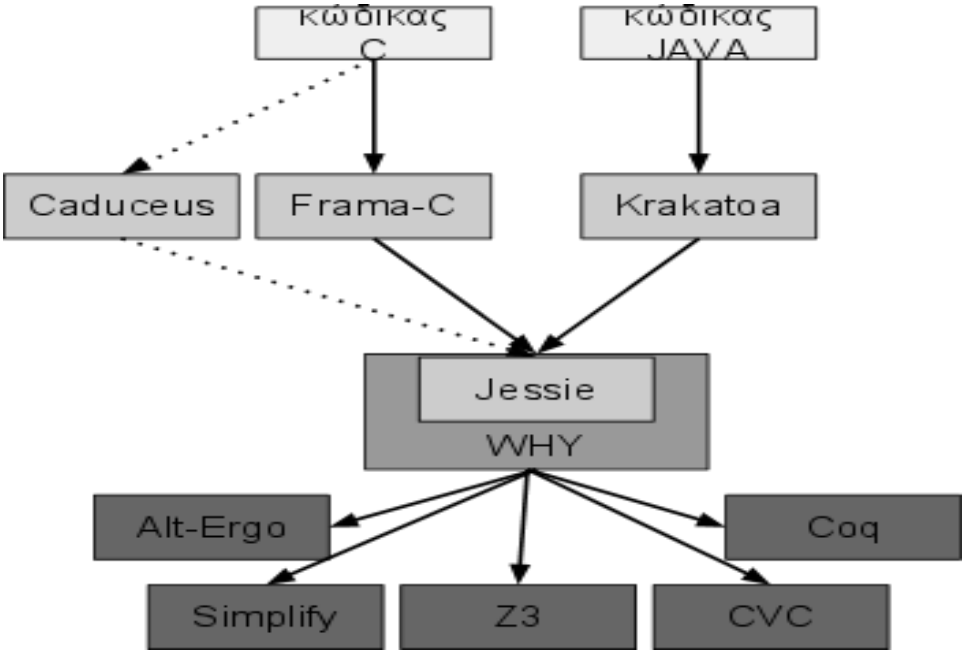
Έχει αναπτυχθεί από τον Jean-Christophe Filliâtre και άλλους ερευνητές στο γαλλικό Laboratoire de Recherche en Informatique. Το Why επιτρέπει επαλήθευση προγραμμάτων σε διαφορετικές γλώσσες, ενώ ο έλεγχος των συνθηκών επαλήθευσης πραγματοποιείται από εξωτερικά εργαλεία απόδειξης θεωρημάτων.

Το WHY δεν προορίζεται για μια και μόνο γλώσσα προγραμματισμού αλλά, έχοντας την δική του γλώσσα την WL η οποία αποτελεί μια παραλλαγή της ML, επιτρέπει προγράμματα μέσω κατάλληλων εργαλείων να μεταφραστούν στη δική του. Τέτοια εργαλεία είναι το caduceus για την C, τού οποίου η ανάπτυξη σταμάτησε για χάρη της Frama-C και το krakatoa για την JAVA.

Το WHY δέχεται σαν είσοδο κώδικα WL και παράγει κώδικα κατάλληλο για την χρήση αυτόματων μηχανών απόδειξης θεωρημάτων(provers), όπως τα Alt-Ergo, Simplify, Z3, CVC αλλά και μη αυτόματων, όπως το coq.

Τέλος οι provers θα αποφανθούν για την ορθότητα των αρχικών προδιαγραφών που ετέθησαν στον πηγαίο κώδικα. Οι provers αυτοί λειτουργούν με λογική πρώτης τάξης. Στο παρακάτω σχήμα φαίνεται όλη η πορεία που ακολουθείται από των πηγαίο κώδικα ως την τελική απόφαση των provers.

Η εργασία αυτή δεν ασχολείται με ο,τι βρίσκεται κάτω από την γλώσσα ACSL. Τα επίπεδα που βρίσκονται πιο κάτω, θεωρείται ότι είναι ένα μαύρο κουτί και ότι λειτουργούν σωστά.



Εικόνα 2 Εργαλεία για την απόδειξη ορθότητας προγραμμάτων

Η Γλώσσα ACSL

Η γλώσσα ACSL είναι γλώσσα προδιαγραφών για πηγαίο κώδικα γραμμένο σε ANSI-C. Οι προδιαγραφές που ορίζονται με την γλώσσα αυτή γράφονται εμβόλιμα στον πηγαίο κώδικα σαν σχόλια, με αποτέλεσμα ο κώδικας να μπορεί να περάσει από οποιονδήποτε μεταγλωττιστή. Τα σχόλια που αφορούν τις προδιαγραφές πρέπει να ξεκινάνε με /*@ ή με //@. Όσα σχόλια δε ξεκινούν έτσι θεωρούνται απλά σχόλια και δεν αφορούν προδιαγραφές. Το τέλος κάθε προδιαγραφής ορίζεται από το ελληνικό ερωτηματικό (;) .

Στη γλώσσα επιτρέπεται να μπουν σχολιασμοί με προδιαγραφές που αφορούν συναρτήσεις πριν από αυτές, γενικές αναλλοίωτες που αφορούν όλο το πρόγραμμα, αναλλοίωτες τύπων που αφορούν δομές και ενώσεις, λογικές προδιαγραφές όπως λογικές συναρτήσεις, αξιώματα και λήμματα. Επιπλέον, επιτρέπονται σχολιασμοί με assertions σε διάφορα σημεία του κώδικα, αναλλοίωτες βρόχων πριν από κάθε βρόχο, κώδικας φάντασμα, δηλαδή επιπλέον C κώδικας ο οποίος είναι ορατός μόνο για τις προδιαγραφές.

1.1.1. Λέξεις κλειδιά

requires: Με αυτή ορίζονται όλες οι προσυνθήκες για μια συνάρτηση.
ensures: Με αυτή ορίζονται όλες οι μετασυνθήκες για μια συνάρτηση.
\result: Είναι το αποτέλεσμα μιας συνάρτησης.
\valid(p): Ο δείκτης p είναι έγκυρος
\valid_range(a, 0, n-1): Ο πίνακας a[n] είναι έγκυρος
\old(a): η τιμή της μεταβλητής α πριν την εκτέλεση του κώδικα

\forall: ∀ για κάθε

\exists: ∃ υπάρχει

\at(a, LABEL): η τιμή της μεταβλητής a στη θέση του LABEL

assigns: Ποια μνήμη επηρεάζεται από το πρόγραμμα

predicate: χρησιμοποιείται για την δήλωση ενός λογικού κατηγορήματος που μπορεί να είναι αληθές η όχι

inductive: χρησιμοποιείται για την δήλωση ενός λογικού κατηγορήματος που προκύπτει επαγωγικά

case: χρησιμοποιείται για την δήλωση περίπτωσης ενός λογικού κατηγορήματος που προκύπτει επαγωγικά

axiomatic: χρησιμοποιείται για την δήλωση ενός λογικού συναρτήσεων που προκύπτουν με χρήση αξιωμάτων

axiom: Προηγείται της δήλωσης ενός αξιώματος.

reads[a]: "διαβάζει" τα πραγματικά δεδομένα σε μια λογική συνάρτηση

loop invariant: χρησιμοποιείται πριν από κάθε βρόχο για να δηλωθεί η αναλλοίωτή του

loop variant: χρησιμοποιείται πριν από κάθε βρόχο για να δηλωθεί η μεταβλητή του

loop assigns: χρησιμοποιείται πριν από κάθε βρόχο για να δηλωθεί ποια μνήμη μεταβάλλει ο βρόχος

ghost: Κώδικας φάντασμα της C που χρησιμοποιείται μόνο για την απόδειξη

1.1.2. Παραδείγματα

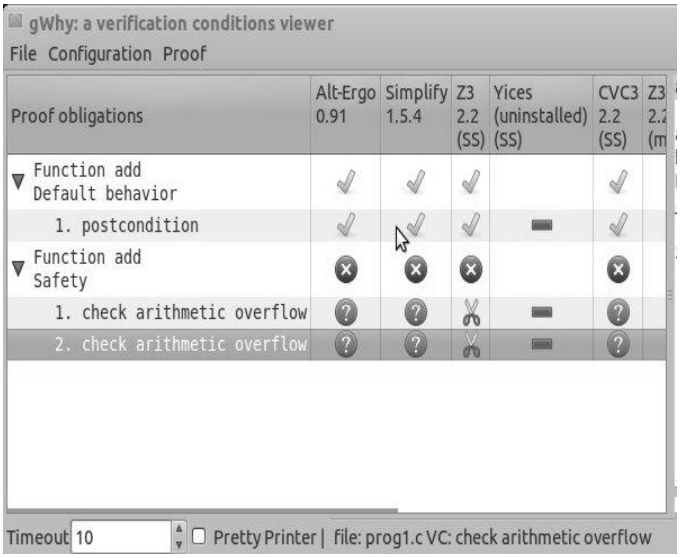
Ένα απλό παράδειγμα (4.1), για να αρχίσει να γίνεται αντιληπτή η χρήση των παραπάνω, είναι η συνάρτηση

```
int add(int a, int b){
  return a + b;
}
```

Για αυτήν την συνάρτηση μπορούμε να γράψουμε τη μετασυνθήκη //@ ensures \result == a + b;

```
//@ ensures \result == a + b;
int add(int a, int b){
  return a + b;
}
```

Ο έλεγχος της ορθότητας του παραδείγματος δίνει το πιο κάτω αποτέλεσμα



Εικόνα 3 Αποτέλεσμα της απόδειξης ορθότητας παρδείγματος 4.1

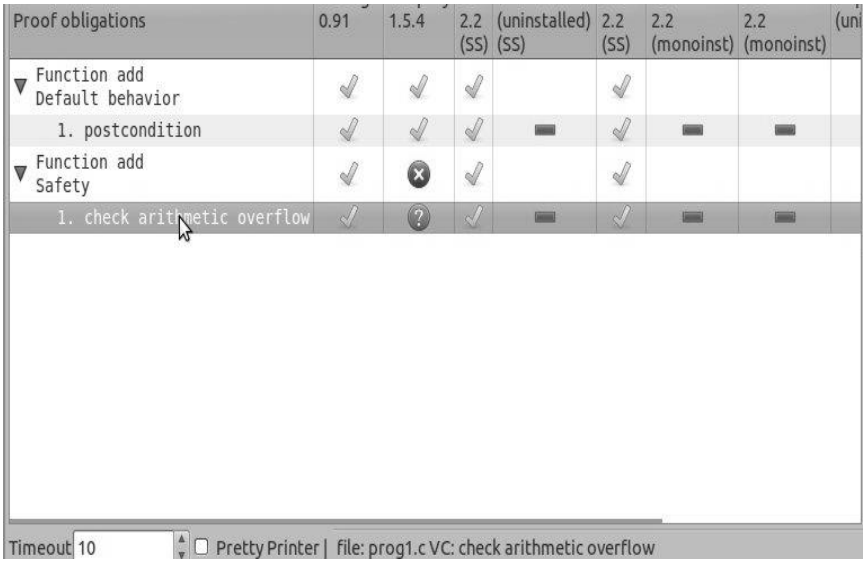
Σύμφωνα με το αποτέλεσμα ισχύει η μετασυνθήκη.

Το πρόβλημα στην ορθότητα υπάρχει στη ίδια την πρόσθεση, αφού δεν υπάρχει έλεγχος για υπερχείλιση. Αυτό μας οδηγεί στο συμπέρασμα ότι η προσυνθήκη θα πρέπει να είναι της μορφής //@ requires -MAX_INT <= a + b < = MAX_INT;. Όπου MAX_INT = 2147483647 (signed int 32 bit).

Το πρόγραμμα θα γίνει

```
/*@ requires -MAX_INT <= a + b < = MAX_INT;
  @ ensures \result == a + b;
  @*/
int add(int a, int b){
  return a + b;
}
```

Ο έλεγχος της ορθότητας του παραδείγματος αυτή τη φορά δίνει το πιο κάτω αποτέλεσμα



Εικόνα 4 Σωστό Αποτέλεσμα της απόδειξης ορθότητας παρδείγματος 4.1

Από την εικόνα των αποτελεσμάτων εξάγεται το συμπέρασμα ότι η συνάρτηση είναι ορθή. Οι provers από τους οποίους αποδείχθηκε αυτό είναι οι alt-ergo, z3 και CVC ενώ ο simplify έβγαλε σαν σφάλμα το αποτέλεσμα.

Για να είναι ορθό ένα αποτέλεσμα αρκεί να έχει αποδειχθεί κάθε συνθήκη από έναν τουλάχιστον από όλους τους provers, χωρίς να θεωρούμε υποχρεωτικό ένας prover να αποδεικνύει όλες τις συνθήκες.

Θεωρώντας ότι με το παραπάνω παράδειγμα έγινε κατανοητή η χρήση της γλώσσας θα παρουσιαστεί ένα λίγο πιο περίπλοκο πρόβλημα (4.2)[3]

```
typedef int value_type;
typedef int size_type;
typedef int bool;

/*@ axiomatic CountAxiomatic
  @{
  @ logic integer Count{L}(value_type* a, value_type v,
  @   integer i, integer j) reads a[i..(j-1)];
  @ axiom Count0:
  @   \forall value_type *a, v, integer i;
  @      Count(a, v, i, i) == 0;
  @ axiom Count1:
  @   \forall value_type *a, v, integer i, j, k;
```

[3] Virgile Prevosto ACSL mini tutorial

```
  @      0 <= i <= j <= k ==> Count(a, v, i ,k) ==
  @      Count(a, v, i, j) + Count(a, v, j, k);
  @ axiom Count2:
  @   \forall value_type *a, v, integer i;
  @      (a[i] != v ==> Count(a, v, i, i+1) == 0) &&
  @      (a[i] == v ==> Count(a, v, i, i+1) == 1);
  @ }
  @ lemma CountLemma: \forall value_type *a, v, integer i;
  @   0 <= i ==> Count(a, v, 0, i+1) ==
  @   Count(a, v, 0, i) + Count(a, v, i, i+1);
  @ */

/*@
  @ requires \valid_range(a, 0, n-1);
  @ requires 0 <= n;[4]
  @ assigns \nothing;
  @ ensures \result == Count(a, val, 0, n);
  @*/
size_type count(const value_type* a, size_type n, value_type val)
{
  size_type cnt = 0;
  /*@
    @ loop invariant 0 <= i <= n;
    @ loop invariant 0 <= cnt <= i;
    @ loop invariant cnt == Count(a, val, 0, i);
    @ loop variant n-i;
    @ */
for (size_type i = 0; i < n; i++)
 if (a[i] == val)
cnt++;
return cnt;
}
```

Στο παράδειγμα αυτό, η συνάρτηση count μετράει πόσες εμφανίσεις έχει η τιμή val μέσα στις πρώτες "n" θέσεις ενός πίνακα a. Για την απόδειξη αυτής της συνάρτησης κατασκευάστηκε μια λογική συνάρτηση, η Count. Η συνάρτηση δέχεται τις πιο κάτω τέσσερεις παραμέτρους: Τον προς εξέταση πίνακα, την τιμή της οποίας μετράει τις εμφανίσεις, τη θέση από όπου ξεκινάει η μέτρηση και τη θέση που σταματά η μέτρηση. Για το ορισμό της λογικής αυτής συνάρτησης δόθηκαν τα αξιώματα:

α. \forall value_type *a, v, integer i; Count(a, v, i, i) == 0;

[4] Στο ACSL mini tutorial η προσυνθήκη αυτή έλλειπε με αποτέλεσμα η απόδειξη να μην είναι εφικτή.

Η τιμή της λογικής συνάρτησης Count θα είναι 0 όταν η θέση του πίνακα, από όπου ξεκινάει η μέτρηση, ταυτίζεται με την θέση του πίνακα, στην οποία σταματά.

β. \forall value_type *a, v, integer i, j, k; 0 <= i <= j <= k ==> Count(a, v, i ,k) == Count(a, v, i, j) + Count(a, v, j, k);

Οι εμφανίσεις (το αποτέλεσμα της συνάρτησης) για της θέσεις έναρξης i και τερματισμού k θα είναι ίσες με το άθροισμα των εμφανίσεων από i έως j συν αυτές από j έως k, εφόσον το j είναι μικρότερο του k.

γ. \forall value_type *a, v, integer i;(a[i] != v ==> Count(a, v, i, i+1) == 0) && (a[i] == v ==> Count(a, v, i, i+1) == 1);

Η τιμή της συνάρτησης όταν εξετάζεται μία μόνο θέση του πίνακα a θα είναι 1, αν στη θέση αυτή εμφανίζεται η τιμή, της οποίας ερευνάται ο αριθμός εμφανίσεων, και 0 αν δεν εμφανίζεται η τιμή στη συγκεκριμένη θέση.

Επίσης ορίζεται ένα λήμμα, που θα βοηθήσει στην απόδειξη, το οποίο είναι

lemma CountLemma: \forall value_type *a, v, integer i; 0 <= i ==> Count(a, v, 0, i+1) ==

Count(a, v, 0, i) + Count(a, v, i, i+1);

Βάσει του λήμματος αυτού οι εμφανίσεις της τιμής v στον πίνακα a, από την αρχή του πίνακα έως την θέση i+1, θα είναι ίσες με το άθροισμα των εμφανίσεων της τιμής αυτής στο δεδομένο πίνακα από την αρχή του έως την θέση i συν 1 αν εμφανίζεται στη θέση i η ερευνούμενη τιμή, ή 0 αν δεν εμφανίζεται.

Με την βοήθεια της λογικής αυτής συνάρτησης μπορούν τώρα να ορισθούν οι προσυνθήκες, οι μετασυνθήκες αλλά και οι αναλλοίωτες του βρόχου

Η πρώτη προσυνθήκη της συνάρτησης είναι ότι απαιτείται ένας πίνακας με “n” έγκυρες θέσεις. Η δεύτερη απαιτεί ο αριθμός “n” να είναι μεγαλύτερος ή ισος με το μηδέν.

Η μετασυνθήκη είναι το αποτέλεσμα της συνάρτησης να ταυτίζεται με αυτό της λογικής συνάρτησης.

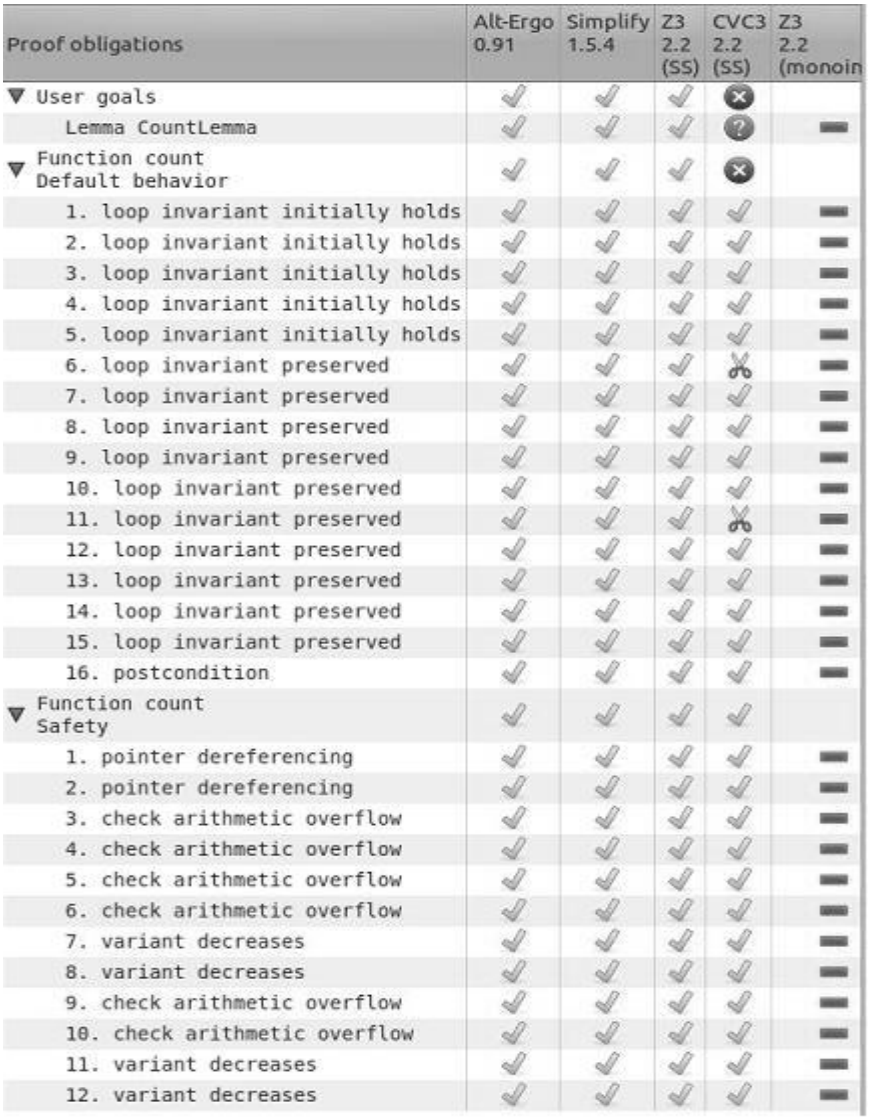
Οι αναλλοίωτες του βρόχου είναι:

α. το 0 <= i <= n, δηλαδή ότι η μεταβλητή του βρόχου θα από 0 έως και “n”, που είναι και η συνθήκη τερματισμού

β. 0 <= cnt <= i, η εμφανίσεις θα είναι λιγότερες ή το πολύ ίσες από τις θέσεις που εξετάσθηκαν

γ. cnt == Count(a, val, 0, i), για τις θέσεις που εξετάστηκαν θα έχουμε τόσες εμφανίσεις, όσες η λογική συνάρτηση δίνει σαν αποτέλεσμα για τις ίδιες θέσεις ανά πάσα στιγμή

Ο έλεγχος της ορθότητας του παραδείγματος αυτού δίνει το πιο κάτω αποτέλεσμα



Proof obligations	Alt-Ergo 0.91	Simplify 1.5.4	Z3 2.2 (SS)	CVC3 2.2 (SS)	Z3 2.2 (monoin
▼ User goals	✓	✓	✓	✗	
Lemma CountLemma	✓	✓	✓	?	—
▼ Function count Default behavior	✓	✓	✓	✗	
1. loop invariant initially holds	✓	✓	✓	✓	—
2. loop invariant initially holds	✓	✓	✓	✓	—
3. loop invariant initially holds	✓	✓	✓	✓	—
4. loop invariant initially holds	✓	✓	✓	✓	—
5. loop invariant initially holds	✓	✓	✓	✓	—
6. loop invariant preserved	✓	✓	✓	✂	—
7. loop invariant preserved	✓	✓	✓	✓	—
8. loop invariant preserved	✓	✓	✓	✓	—
9. loop invariant preserved	✓	✓	✓	✓	—
10. loop invariant preserved	✓	✓	✓	✓	—
11. loop invariant preserved	✓	✓	✓	✂	—
12. loop invariant preserved	✓	✓	✓	✓	—
13. loop invariant preserved	✓	✓	✓	✓	—
14. loop invariant preserved	✓	✓	✓	✓	—
15. loop invariant preserved	✓	✓	✓	✓	—
16. postcondition	✓	✓	✓	✓	—
▼ Function count Safety	✓	✓	✓	✓	
1. pointer dereferencing	✓	✓	✓	✓	—
2. pointer dereferencing	✓	✓	✓	✓	—
3. check arithmetic overflow	✓	✓	✓	✓	—
4. check arithmetic overflow	✓	✓	✓	✓	—
5. check arithmetic overflow	✓	✓	✓	✓	—
6. check arithmetic overflow	✓	✓	✓	✓	—
7. variant decreases	✓	✓	✓	✓	—
8. variant decreases	✓	✓	✓	✓	—
9. check arithmetic overflow	✓	✓	✓	✓	—
10. check arithmetic overflow	✓	✓	✓	✓	—
11. variant decreases	✓	✓	✓	✓	—
12. variant decreases	✓	✓	✓	✓	—

Εικόνα 5 Σωστό αποτέλεσμα της απόδειξης ορθότητας παρδείγματος 4.2

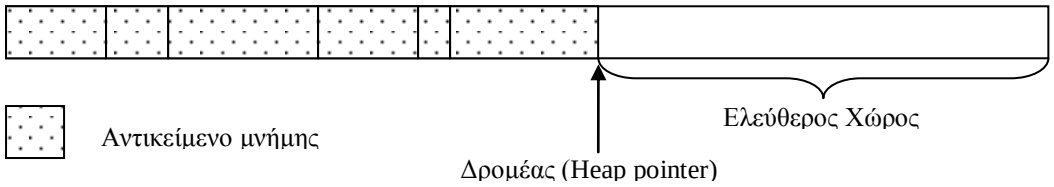
Βασικές Αρχές για την Απόδειξη

Για να γίνει κατανοητή η διαδικασία της απόδειξης πρέπει αρχικά να τεθούν οι επιδιωκόμενοι στόχοι, οι βασικές αρχές που ακολουθήθηκαν, οι παραδοχές που έγιναν και γενικά όλες οι λεπτομέρειες που αφορούν σ' αυτή.

Η δομή της μνήμης

Η μνήμη, που είναι ένας σωρός, είναι ένας πίνακας που δέχεται τιμές ακεραίων (signed short 16 bit). Το μέγεθος του πίνακα είναι 32768 θέσεις. Η τιμή αυτή προκύπτει από την χρησιμοποίηση των 15 bit από τα 16 των ακεραίων. Τα δύο κομμάτια της μνήμης θα έχουν μέγεθος 16384 θέσεις το καθένα.

Τα αντικείμενα τοποθετούνται στη μνήμη σε σειρά. Αυτό σημαίνει πως πάντα ελεύθερος χώρος είναι αυτός που μένει από το τέλος του τελευταία αποθηκευμένου αντικειμένου έως το τέλος του ενεργού κομματιού της μνήμης. Οριζουμε ένα δρομέα (heap pointer), ο οποίος θα δείχνει κάθε φορά στη πρώτη θέση του ελεύθερου χώρου της ενεργής μνήμης.



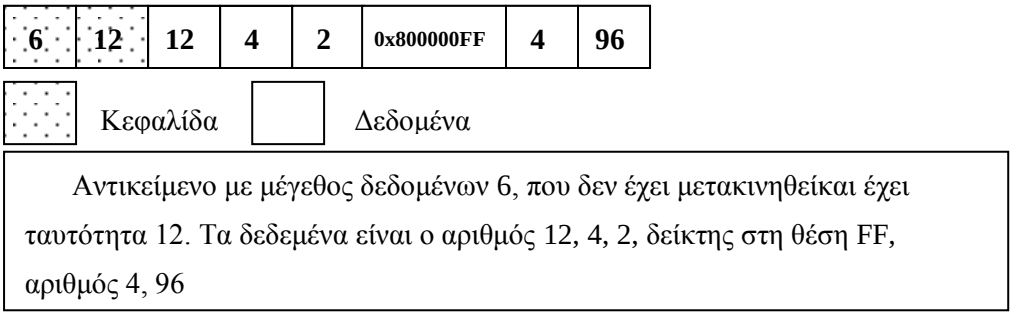
Εικόνα 6 Δομή της ενεργής μνήμης

Κάθε αντικείμενο, που βρίσκεται αποθηκευμένο στην μνήμη, θα πρέπει να έχει μια συγκεκριμένη μορφή. Για την υλοποίηση της απόδειξης που πραγματοποιείται, το κάθε αντικείμενο της μνήμης αποτελείται από δύο μέρη, την κεφαλίδα και τα δεδομένα.

Η κεφαλίδα έχει το σταθερό μέγεθος των δύο θέσεων. Οι δύο θέσεις της κεφαλίδας περιέχουν πληροφορίες που αφορούν στο αντικείμενο. Στην πρώτη θέση αποθηκεύεται το μήκος του κομματιού των δεδομένων. Στη δεύτερη θέση βρίσκεται αποθηκευμένος ο μοναδικός αναγνωριστικός αριθμός, ταυτότητα, του αντικειμένου.

Στα αντικείμενα που έχουν μετακινηθεί, κατά την διάρκεια της συλλογής, στο ενεργό κομμάτι της μνήμης, η κεφαλίδα παίρνει τιμές που αφορούν την μετακίνηση. Στην πρώτη θέση αποθηκεύεται η τιμή FORWARDED = 0x0bad0bad που υποδεικνύει ότι το αντικείμενο αυτό έχει μετακινηθεί κατά την διάρκεια της συλλογής. Στη δεύτερη θέση αποθηκεύεται η θέση του αντιγράφου στην ενεργή μνήμη.

Δεν υπάρχει περιορισμός για το μήκος του μέρους των δεδομένων, εκτός από το μέγεθος της μνήμης. Τα δεδομένα μπορεί να είναι δύο ειδών, είτε δείκτες προς άλλα αντικείμενα, είτε ακέραιοι αριθμοί. Η διαφοροποίησή τους γίνεται χρησιμοποιώντας το πρώτο (msb) bit..



Εικόνα 7 Παράδειγμα αντικειμένου μνήμης

Το συνολικό μέγεθος ενός αντικειμένου είναι η τιμή της πρώτης θέσης της κεφαλίδας συν δύο, που είναι το μέγεθος της κεφαλίδας

Επιδιωκόμενοι στόχοι

Η απόδειξη της ορθότητας του συλλέκτη σκουπιδιών σημαίνει την απόδειξη κάποιων επιμέρους στόχων. Η απόδειξη αυτών των στόχων θα σημαίνει και την συνολική απόδειξη.

Ο πρώτος στόχος είναι να αποδειχτεί ότι ο ελεύθερος χώρος μετά την συλλογή σκουπιδιών θα είναι περισσότερος ή το πολύ ίσος με τον ελεύθερο χώρο που υπήρχε πριν την συλλογή. Δηλαδή ο δρομέας μετά την συλλογή έχει τιμή ίση ή μικρότερη από αυτήν που είχε πριν.

Κάθε αντικείμενο θα πρέπει να περιέχει τα ίδια δεδομένα, δηλαδή θα πρέπει να έχει δείκτες στα ίδια αντικείμενα με αυτά που είχε πριν την συλλογή. Επίσης οι θέσεις των δεδομένων θα πρέπει να είναι η ίδια. Αυτό εξασφαλίζει ότι η δομή των δεδομένων της μνήμης δεν έχει αλλάξει.

Όλα τα αντικείμενα που πριν την συλλογή ήταν συνδεδεμένα με τις ρίζες, μετά την συλλογή θα πρέπει να βρίσκονται στην ενεργή μνήμη. Όλα τα αντικείμενα που πριν τη συλλογή δεν ήταν συνδεδεμένα με τις ρίζες, στο τέλος της συλλογής δε θα πρέπει να βρίσκονται στην ενεργή περιοχή. Μετά την συλλογή σκουπιδιών θα πρέπει το κάθε αντικείμενο της μνήμης να είναι συνδεδεμένο με τις ρίζες. Αυτό σημαίνει πως ο συλλέκτης έχει μεταφέρει μόνο ενεργά αντικείμενα και όχι σκουπίδια.

Τέλος, δεν θα πρέπει να υπάρχουν δείκτες από την ενεργή μνήμη που να δείχνουν στην ανενεργή, δηλαδή όλα τα ενεργά αντικείμενα έχουν μεταφερθεί στην ενεργή περιοχή της μνήμης. Αν κάποιος δείκτης έδειχνε στην μη ενεργή περιοχή της μνήμης τότε αυτό θα οδηγούσε σε απώλεια δεδομένων.

Η επαλήθευση όλων των παραπάνω επιμέρους στόχων είναι ικανή συνθήκη για την συνολική επαλήθευση της ορθότητας της λειτουργίας του συλλέκτη.

Υλοποίηση του Συλλέκτη Σκουπιδιών

Οι λόγοι της υλοποίησης

Η απόδειξη του κώδικα ενός συλλέκτη που χρησιμοποιείται από κάποια γλώσσα θα ήταν η ιδανική περίπτωση. Δυστυχώς όμως, καθώς ο κώδικας αυτός δεν είναι καθαρός, δηλαδή δεν περιέχει μόνο στοιχεία που αφορούν αποκλειστικά τον συλλέκτη, αλλά περιέχει και άλλα στοιχεία που χρησιμοποιεί η γλώσσα, έκανε το εγχείρημα ιδιαιτέρως δύσκολο.

Για παράδειγμα, παρότι το Jikes είναι γραμμένο σε JAVA και θα μπορούσε να γίνει η απόδειξη σε JML με το εργαλείο krakatoa του WHY, αυτό δεν ήταν δυνατό, καθώς ο κώδικας του συλλέκτη περιείχε πολλά κομμάτια που είχαν σχέση με άλλες λειτουργίες αυτής της εικονικής μηχανής.

Για λόγους απλότητας έπρεπε να γραφτεί κώδικας όσο το δυνατόν πιο απλός αλλά και παρόμοιος με αυτόν που χρησιμοποιείται πραγματικά. Σαν κώδικας πρότυπο, δηλαδή μοντέλο βάσει του οποίου θα γραφόταν ο νέος κώδικας, επιλέχθηκε αυτός που

βρίσκεται στο MMTk(Memory Management Toolkit) του Jikes. Το Jikes υλοποιεί όλους τους αλγόριθμους που αφορούν στη συλλογή σκουπιδιών.

Ο κώδικας του συλλέκτη

Ο κώδικας τελικά υλοποιήθηκε σε γλώσσα ANCI C. Τα ονόματα όλων των συναρτήσεων είναι ίδια με αυτά που χρησιμοποιούνται από το Jikes, έτσι αν κάποιος θελήσει να αντιπαραθέσει τους δύο κώδικες θα το κάνει με σχετική ευκολία.

Ορισμοί

```
typedef unsigned int word;

// Macros for boxing numbers and pointers in words

#define WORD_OF_NUM(n)     ((n) & 0x7ffffffU)
#define WORD_OF_OFS(p)     ((p) | 0x8000000U)
#define OF_WORD(w)         ((w) & 0x7ffffffU)
#define IS_OFFSET(w)       (((w) & 0x8000000U) == 0x8000000U)


// The root
extern word root;


// The heap space

#define SIZE_OF_SPACE 16384

extern word heap [2 * SIZE_OF_SPACE];
extern word hpmin, hpmax;
extern word hp;
extern word id;

#define OBJ_HEADER_SIZE 2
#define MAX_ID    0xffffffffU
#define FORWARDED 0x0bad0badU
```

Η μνήμη είναι η μεταβλητή heap. heap είναι ένας πίνακας 32768 θέσεων όπου η κάθε θέση έχει μέγεθος 32 bit. Το συνολικό μέγεθος της μνήμης είναι 128k. Για την αναφορά σε μια θέση της μνήμης απαιτείται ένας ακέραιος μήκους 15 bit.

Το ενεργό κομμάτι της μνήμης ξεκινάει από το hpmin και έχει σαν μέγιστη τιμή το hpmax. Το μέγεθος του κάθε κομματιού της μνήμης είναι 16384 θέσεις. Εύκολα κατανοητό είναι ότι το hpmax = hpmin + 16384.

Τα αντικείμενα στη μνήμη γράφονται διαδοχικά το ένα ακριβώς μετά το επόμενό του. Με αυτό τον τρόπο το ελεύθερο κομμάτι της ενεργής μνήμης θα είναι στο τέλος της. Η μεταβλητή hp (heap pointer) είναι ένας δρομέας που δείχνει πάντα στην πρώτη θέση της ελεύθερης μνήμης. Από τα παραπάνω συμπεραίνουμε ότι ο δρομέας παίρνει τιμές από hpmin έως hpmax.

Κάθε αντικείμενο της μνήμης έχει τα εξής χαρακτηριστικά:

α. Μια κεφαλίδα που αποτελείται από δύο λέξεις (OBJ_HEADER_SIZE = 2). Η πρώτη λέξη περιέχει το μήκος του αντικειμένου, δηλαδή τον συνολικό αριθμό λέξεων που χρησιμοποιεί για την αποθήκευση των δεδομένων που περιέχει. Η δεύτερη λέξη περιέχει έναν αριθμό ταυτότητας που είναι μοναδικός για κάθε αντικείμενο. Δεν μπορεί δύο ενεργά αντικείμενα να έχουν τον ίδιο αριθμό ταυτότητας.

β. Τα δεδομένα, τα οποία είναι αποθηκευμένα στο αντικείμενο της μνήμης, μπορεί να είναι δύο ειδών, είτε αριθμητικά δεδομένα, είτε δείκτες προς άλλα αντικείμενα. Για τον διαχωρισμό, των δύο αυτών ειδών των δεδομένων, χρησιμοποιούμε ένα bit της λέξης. Έτσι από τα 32 bit της λέξης όταν το ποιο σημαντικό είναι ίσο με 1 σημαίνει ότι το περιεχόμενο της λέξης είναι δείκτης προς άλλο αντικείμενο, ενώ όταν είναι ίσο με 0 σημαίνει πως το περιεχόμενο είναι αριθμητικά δεδομένα.

Από τα παραπάνω γίνεται αντιληπτό πως το μήκος των αριθμητικών δεδομένων και των δεικτών είναι 31 bit. Για τους δείκτες και για την μνήμη, που χρησιμοποιούμαι στην απόδειξή μας, αυτό το μήκος είναι αρκετό αφού το μέγεθος της μνήμης είναι 128k και δεν χρειαζόμαστε πάνω από 15bit για της αναφορές προς τις θέσεις αυτής.

Οι δείκτες προς αντικείμενα της μνήμης δείχνουν την πρώτη θέση των δεδομένων των αντικειμένων αυτών. Για παράδειγμα αν ένας δείκτης έχει τιμή 4242 (1092 στο δεκαεξαδικό σύστημα). Η τιμή που θα έχει το δεδομένο είναι 80001092 αφού το πρώτο bit της λέξης θα είναι 1. Η θέση 4242 στη μνήμη θα είναι η αρχή των δεδομένων ενός αντικειμένου, του οποίου ο αριθμός ταυτότητας θα είναι στη θέση 4241 και το μεγεθός του θα είναι αποθηκευμένο στη θέση 4240.

Για την εύκολη διαχείριση χρησιμοποιούνται οι macro

WORD_OF_NUM(n): η οποία επιστρέφει τον αριθμό n με το πρώτο bit ορισμένο σε 0. Η μορφή αυτή χρησιμοποιείται για την αποθήκευση δεδομένων.

WORD_OF_OFS(p): η οποία επιστρέφει τον αριθμό p με το πρώτο bit ορισμένο σε 1. Η μορφή αυτή χρησιμοποιείται για την αποθήκευση δεικτών σε αντικείμενα.

OF_WORD(w): η οποία επιστρέφει έναν αριθμό από το αριθμητικό δεδομένο w έχοντας καθαρίσει το πρώτο bit από την σήμανση.

OF_OFFSET(w): η οποία επιστρέφει αληθείς αν η τιμή w είναι δείκτης.

Για να διευκολυνθεί η απόδειξη, ορίζεται μια και μοναδική ρίζα. Η ρίζα αυτή μπορεί να περιέχει είτε δεδομένο είτε δείκτη προς κάποιο αντικείμενο της μνήμης.

Δέσμευση νέας μνήμης

```
word allocate (word size)
{
  word start = hp;
  word i;
  hp += OBJ_HEADER_SIZE + size;
  heap[start] = size;
  heap[start+1] = id++;
  for (i=0; i<size; i++)
    heap[start + OBJ_HEADER_SIZE + i] = WORD_OF_NUM(0);
  return start + OBJ_HEADER_SIZE;
}
```

Η συνάρτηση allocate δεσμεύει τόσες θέσεις δεδομένων, όσες απαιτούνται από την παράμετρο size, συν δύο που είναι το μέγεθος της κεφαλίδας για το νέο αντικείμενο της μνήμης. Επιστρέφει την πρώτη θέση της μνήμης στις οποίες θα αποθηκευτούν τα δεδομένα του αντικειμένου.

Όπως έχει ήδη αναφερθεί, ο δρομέας της μνήμης δείχνει την πρώτη θέση του ελεύθερου χώρου. Σε αυτή τη θέση αποθηκεύεται το μέγεθος του αντικειμένου. Στην επόμενη αποθηκεύεται ο αριθμός ταυτότητάς του και στη συνέχεια τα δεδομένα του. Από τα παραπάνω συνεπάγεται ότι η συνάρτηση επιστρέφει την θέση του δρομέα συν το μέγεθος της κεφαλίδας, που είναι ίσο με δύο.

Τέλος η θέση του δρομέα μετακινείται κατά το μέγεθος του αντικειμένου, κεφαλίδας και δεδομένων.

Η συνάρτηση αυτή είναι μια απλοποιημένη μορφή της κανονικής συνάρτησης. Κανονικά θα έπρεπε για κάθε νέα αίτηση για χώρο στη μνήμη να γίνεται έλεγχος αν αυτός ο χώρος είναι διαθέσιμος. Αν ο χώρος δεν είναι διαθέσιμος, τότε θα έπρεπε να ξεκινάει η διαδικασία του συλλέκτη. Η συνάρτηση αυτή δεν χρησιμοποιείται από τον συλλέκτη και απλά παρατίθεται για να φανεί με έναν πιο γενικό τρόπο η υλοποίηση.

Αντιγραφή αντικειμένου

```
word forward (word p)
{
   if (heap[p-OBJ_HEADER_SIZE] == FORWARDED)
       return heap[p-OBJ_HEADER_SIZE+1];
   else {
       word size = heap[p-OBJ_HEADER_SIZE];
       word i;

       for (i=0; i < OBJ_HEADER_SIZE + size; i++)
         heap[hp + i] = heap[p - OBJ_HEADER_SIZE + i];

       heap[p-OBJ_HEADER_SIZE] = FORWARDED;
       heap[p-OBJ_HEADER_SIZE+1] = hp + OBJ_HEADER_SIZE;
       hp += OBJ_HEADER_SIZE + size;

       return heap[p-OBJ_HEADER_SIZE+1];
   }
}
```

Η αντιγραφή των αντικειμένων είναι η βασική λειτουργία του συλλέκτη αντιγραφής. Η συνάρτηση forward, η οποία υλοποιεί την αντιγραφή, δέχεται σαν παράμετρο τη θέση του προς αντιγραφή αντικειμένου (πηγαίου αντικείμενου).

Η συνάρτηση καταρχάς ελέγχει αν το αντικείμενο έχει ήδη αντιγραφεί σε προηγούμενη φάση της λειτουργίας του συλλέκτη. Αν αυτό έχει συμβεί τότε η συνάρτηση επιστρέφει την διεύθυνση στην ενεργή μνήμη στην οποία βρίσκεται το αντικείμενο.

Αν το αντικείμενο δεν έχει ήδη αντιγραφεί η συνάρτηση ξεκινάει την αντιγραφή του στην πρώτη ελεύθερη θέση της ενεργής μνήμης. Με το τέλος της αντιγραφής υπάρχουν δύο ίδια αντικείμενα ένα στην ενεργή και ένα στην μη ενεργή μνήμη.

Τέλος στο πηγαίο αντικείμενο, δηλαδή σε αυτό που βρίσκεται στη μη ενεργή μνήμη αποθηκεύονται μια σημαία και ένας δείκτης στο αντίγραφο αντικείμενο. Η σημαία θα σημαίνει ότι το αντικείμενο έχει ήδη αντιγραφεί. Η σημαία αποθηκεύεται στη πρώτη θέση της κεφαλίδας αντί του μεγέθους του αντικειμένου που είχε πριν. Ο δείκτης στο αντίγραφο αντικείμενο αποθηκεύεται στη δεύτερη θέση της κεφαλίδας αντί του αριθμού ταυτότητας.

Η συνάρτηση επιστρέφει τη θέση του αντιγράφου και αυξάνει την τιμή του δρομέα της μνήμης κατά τιμή ίση με το συνολικό μέγεθος του αντικειμένου.

Αξίζει να επισημάνουμε δύο σημεία.

α. Το αρχικό αντικείμενο, αν έχει δείκτες, τότε αυτοί δείχνουν στο μη ενεργό κομμάτι της μνήμης. Αυτό είναι λογικό, αφού πριν να ξεκινήσει η συλλογή σκουπιδιών το αντικείμενο αυτό ήταν στην ενεργή περιοχή της μνήμης και έδειχνε σε αντικείμενα επίσης της ενεργής περιοχής. Με την αλλαγή της ενεργής περιοχής, η οποία γίνεται με την εκκίνηση του συλλέκτη όλα τα αντικείμενα βρίσκονται και δείχνουν προς αντικείμενα της μη ενεργής περιοχής της μνήμης

Το αντικείμενο-αντίγραφο που δημιουργήθηκε στο ενεργό κομμάτι της μνήμης αν έχει δείκτες, αυτοί δείχνουν προς αντικείμενα στο μη ενεργό κομμάτι της μνήμης. Αυτό είναι λογικό αφού είναι το ακριβές αντίγραφού του πηγαίου αντικειμένου.

β. Η κάθε αντιγραφή που γίνεται αυξάνει την τιμή του δρομέα κατά το μέγεθος του αντικειμένου. Άρα το αντικείμενο αντιγράφεται σε θέσεις με τιμή ανάμεσα στην παλιά τιμή του δρομέα και την νέα. Άρα οι θέσεις αυτές αν περιέχουν δείκτες, αυτοί θα δείχνουν σε αντικείμενα του ανενεργού κομματιού της μνήμης.

Η συνάρτηση scan

```
word scanobj (word q)
{
  word size = heap[q];
  word i;
  for (i=0; i<size; i++)
    if (IS_OFFSET(heap[q + OBJ_HEADER_SIZE + i])) {
      word p = OF_WORD(heap[q + OBJ_HEADER_SIZE + i]);
      word t = forward(p);
      heap[q + OBJ_HEADER_SIZE + i] = WORD_OF_OFS(t);
    }
  return OBJ_HEADER_SIZE + size;
}
```

Η συνάρτηση scan εξετάζει το αντικείμενο, που βρίσκεται στη θέση q, για το αν έχει δείκτες προς άλλα αντικείμενα. Αν υπάρχουν τέτοιοι δείκτες, τότε τα αντικείμενα, στα οποία δείχνουν οι δείκτες αυτοί, ελέγχονται από τη συνάρτηση forward, για το αν πρέπει να αντιγραφούν, αν αυτό δεν έχει ήδη γίνει, και αντιγράφονται.

Στις θέσεις που έδειχναν οι αρχικοί δείκτες αποθηκεύονται οι θέσεις των αντιγράφων των αρχικών αντικειμένων. Με αυτόν τον τρόπο με την επιστροφή της συνάρτησης το αντικείμενο που εξετάστηκε θα έχει δείκτες που θα δείχνουν μόνο στο ενεργό κομμάτι της μνήμης.

Αν έχουν γίνει αντιγραφές, αφού καλείται η forward, τότε ο δρομέας θα έχει μετακινηθεί και όλες οι θέσεις μνήμης, ανάμεσα στη παλιά και τη νέα τιμή του δρομέα, αν περιέχουν δείκτες, αυτοί θα δείχνουν σε αντικείμενα του ανενεργού κομματιού της μνήμης.

Η συνάρτηση gc

```
void gc ()
{
  word scan;
  hpmin = hpmax % (2*SIZE_OF_SPACE);
  hpmax = hpmin + SIZE_OF_SPACE;
  hp = scan = hpmin;

  if (IS_OFFSET(root)) {
    word p = OF_WORD(root);
    root = WORD_OF_OFS(forward(p));
  }

  while (scan < hp)
    scan += scanobj(scan);
}
```

Η συνάρτηση gc είναι αυτή που καλείται για να ξεκινήσει η διαδικασία της συλλογής. Ξεκινώντας αλλάζει το ενεργό κομμάτι της μνήμης, ορίζοντας σαν αρχή της ενεργής μνήμης το τέλος της τρέχουσας ενεργού μνήμης ή το 0 . Αντίστοιχα παίρνουν τιμές και οι μεταβλητές hp και hpmax.

Ορίζεται μια νέα τοπική μεταβλητή, η scan. Αυτή αρχικά παίρνει τιμή ίση με τον δρομέα της μνήμης. Αρχικά εξετάζεται η ρίζα. Αν η τιμή της ρίζας είναι δείκτης προς αντικείμενο τότε το αντικείμενο αυτό αντιγράφεται στο νέο ενεργό κομμάτι της μνήμης. Η θέση του δρομέα μετακινείται κατά το μέγεθος του αντικειμένου που αντιγράφηκε.

Τώρα ανάμεσα στη θέση που δείχνει η μεταβλητή scan και ο δρομέας, αν υπάρχουν δείκτες θα δείχνουν σε αντικείμενα στο μη ενεργό κομμάτι της μνήμης.

Η δομή επανάληψης while εξετάζει με τη σειρά που τα βρίσκει τα αντικείμενα μέσα στο ενεργό κομμάτι της μνήμης. Τα αντικείμενα εξετάζονται για το αν έχουν δείκτες προς άλλα αντικείμενα όπως περιγράφτηκε στη συνάρτηση scanobj.

Σε κάθε επανάληψη η τιμή της μεταβλητής scan αυξάνει κατά το μέγεθος του αντικειμένου που εξετάστηκε, δείχνοντας το επόμενο, στη σειρά, αντικείμενο. Με τον τρόπο αυτό διασφαλίζεται ότι αν υπάρχουν δείκτες ανάμεσα στις θέσεις από hpmin και scan τότε αυτοί θα δείχνουν σε αντικείμενα που είναι αποθηκευμένα στο ενεργό κομμάτι της μνήμης.

Αντίθετα, αφού έγιναν αντιγραφές τότε ο δρομέας μετακινήθηκε και αν υπάρχουν δείκτες στις θέσεις ανάμεσα στις θέσεις από scan έως hp, τότε αυτοί θα δείχνουν στο μη ενεργό κομμάτι της μνήμης.

Η διαδικασία ολοκληρώνεται αφού εξεταστεί και το τελευταίο αντικείμενο που έχει αντιγραφεί και δε χρειαστεί να γίνουν νέες αντιγραφές. Σε αυτή την περίπτωση η τιμή της scan θα γίνει ίση με αυτή του δρομέα hp.

Με το τέλος της διαδικασίας όλοι οι δείκτες στην ενεργή περιοχή θα δείχνουν σε ζωντανά αντικείμενα της ενεργής περιοχής.

Απόδειξη στη Frama-C

Διαδικασία της απόδειξης

Για να γίνει η απόδειξη του συλλέκτη, όπως αυτή έχει περιγραφεί στους επιδιωκόμενους στόχους, θα πρέπει να γίνει η απόδειξη της κάθε συνάρτησης ξεχωριστά με τέτοιες προδιαγραφές, που να οδηγούν στο τελικό αποτέλεσμα. Για αυτό θα γίνεται μια αναλυτική παρουσίαση των λόγων για τους οποίους επιλέγονται οι εκάστοτε προδιαγραφές.

Στην αρχή θα παρουσιαστούν τα κατηγορήματα και οι λογικές συναρτήσεις που χρησιμοποιούνται και στη συνέχεια η απόδειξη της κάθε συνάρτησης χωριστά.

Κατηγορήματα

Μεταβλητές «φαντάσματα»

Η Frama-C δίνει την δυνατότητα να προσθέσεις, στον προς απόδειξη κώδικα, κώδικα φάντασμα. Ο κώδικας αυτός χρησιμοποιείται μόνο για την απόδειξη και δεν επηρεάζει τη λειτουργία του προγράμματος.

Έτσι για την διευκόλυνση της απόδειξης προστέθηκαν τέτοια κομμάτια στο παράδειγμα που παρουσιάζεται. Επίσης χρησιμοποιήθηκαν και μεταβλητές φαντάσματα, οι οποίες δηλώθηκαν όπως παρουσιάζεται παρακάτω:

```
//@ ghost word oldmin;
//@ ghost word oldhp;
//@ ghost word oldmax;
//@ ghost int  oldleft;.
```

Η μεταβλητή φάντασμα oldmin δείχνει την αρχή του μη ενεργού κομματιού της μνήμης. Αντίστοιχα η oldmax δείχνει το τέλος του μη ενεργού κομματιού της μνήμης. Οι μεταβλητές αυτές «αποθηκεύουν» τις αντίστοιχες μεταβλητές (hpmin, hpmax) ακριβώς πριν αρχίσει η διαδικασία της συλλογής, όταν δηλαδή αυτές έδειχναν την αρχή και το τέλος της τότε ενεργού περιοχής της μνήμης .

Η μεταβλητή oldhp δείχνει την τιμή που είχε ο δρομέας της μνήμης πριν αρχίσει η διαδικασία της συλλογής.

Τέλος η μεταβλητή oldleft θα χρησιμοποιηθεί για να αποθηκεύσει το άθροισμα του μεγέθους των αντικειμένων που δεν έχουν συλλεχθεί.

Κατηγορήματα που αφορούν αντικείμενα

```
/*@ inductive size_of_obj{L} (integer p, integer size) {
  @   case sz_reg{L}:
  @     \forall integer p, size;
  @          \at(heap[p], L) != FORWARDED
  @      ==> \at(heap[p], L) == size
  @      ==> size_of_obj{L}(p, size);
  @   case sz_fwd{L}:
  @     \forall integer p, q, size;
  @          \at(heap[p], L) == FORWARDED
```

```
  @      ==> \at(heap[p+1], L) == q
  @      ==> \at(heap[q-OBJ_HEADER_SIZE], L) == size
  @      ==> size_of_obj{L}(p, size);
  @ }
  @*/
```

Το κατηγόρημα size_of_obj δηλώνει πως το μέγεθος ενός αντικειμένου που βρίσκεται στη θέση p είναι size. Αν το αντικείμενο δεν έχει αντιγραφεί τότε το size θα είναι ίσο με την τιμή που βρίσκεται στη πρώτη θέση της κεφαλίδας του υπό εξέταση αντικειμένου. Αν το αντικείμενο έχει αντιγραφεί τότε η τιμή size θα είναι ίση με την τιμή που βρίσκεται στη πρώτη θέση της κεφαλίδας του αντικειμένου αντιγράφου.

Υπενθυμίζεται ότι αν ένα αντικείμενο έχει αντιγραφεί, η θέση του αντιγράφου είναι αποθηκευμένη στη δεύτερη θέση της κεφαλίδας του αντικειμένου**.**

```
/*@ inductive id_of_obj{L} (integer p, integer id) {
  @   case id_reg{L}:
  @     \forall integer p, id;
  @          \at(heap[p], L) != FORWARDED
  @      ==> \at(heap[p+1], L) == id
  @      ==> id_of_obj{L}(p, id);
  @   case id_fwd{L}:
  @     \forall integer p, q, id;
  @          \at(heap[p], L) == FORWARDED
  @      ==> \at(heap[p+1], L) == q
  @      ==> \at(heap[q-OBJ_HEADER_SIZE+1], L) == id
  @      ==> id_of_obj{L}(p, id);
  @ }
  @*/
```

Το κατηγόρημα id_of_obj δηλώνει πως ο αριθμός ταυτότητας ενός αντικειμένου που βρίσκεται στη θέση p είναι id. Αν το αντικείμενο δεν έχει αντιγραφεί τότε το id θα είναι ίσο με την τιμή που βρίσκεται στη δεύτερη θέση της κεφαλίδας του υπό εξέταση αντικειμένου. Αν το αντικείμενο έχει αντιγραφεί τότε η τιμή id θα είναι ίση με την τιμή που βρίσκεται στη δεύτερη θέση της κεφαλίδας του αντικειμένου αντιγράφου.

Υπενθυμίζεται ότι αν ένα αντικείμενο έχει αντιγραφεί, η θέση του αντιγράφου είναι αποθηκευμένη στη δεύτερη θέση της κεφαλίδας του αντικειμένου**.**

```
/*@ inductive content_of_obj{L} (integer p, integer i, integer w) {
  @   case cnt_reg{L}:
  @     \forall integer p, i, size, w;
  @          \at(heap[p], L) != FORWARDED
  @      ==> \at(heap[p], L) == size
  @      ==> 0 <= i < size
```

```
@      ==> \at(heap[p+OBJ_HEADER_SIZE+i], L) == w
@      ==> content_of_obj{L}(p, i, w);
@   case cnt_fwd{L}:
@     \forall integer p, i, size, w, q;
@          \at(heap[p], L) == FORWARDED
@      ==> \at(heap[p+1], L) == q
@      ==> \at(heap[q-OBJ_HEADER_SIZE], L) == size
@      ==> 0 <= i < size
@      ==> \at(heap[q+i], L) == w
@      ==> content_of_obj{L}(p, i, w);
@ }
@*/
```

Το κατηγόρημα content_of_obj δηλώνει πως ο αριθμός w ενός αντικειμένου που βρίσκεται στη θέση p είναι το περιεχόμενο του δεδομένου του αντικειμένου που βρίσκεται στη θέση i. Το κατηγόρημα εξετάζει το αντικείμενο ή το αντίγραφο του αντικειμένου αν αυτό έχει αντιγραφεί.

```
/*@ inductive good_obj{L} (integer min, integer max) {
  @   case go_none{L}:
  @     \forall integer min, max;
  @          good_obj{L}(min, min);
  @   case so_some{L}:
  @     \forall integer min, max, size;
  @          size_of_obj{L}(min, size)
  @      ==> good_obj{L}(min+(OBJ_HEADER_SIZE+size), max)
  @      ==> good_obj{L}(min, max);
  @ }
  @*/
```

Το κατηγόρημα good_obj δηλώνει πως αν ξεκινώντας από το αντικείμενο που βρίσκεται στη θέση min προσθέτοντας τα μεγέθη των αντικειμένων μέχρι τη θέση max τότε το άθροισμα θα είναι ίσο με max. Αυτό εξασφαλίζει ότι δεν υπάρχουν κενά ανάμεσα στα αντικείμενα, αλλά και ότι η θέση max είναι έγκυρη θέση για αντικείμενο.

```
*@ inductive scan_obj{L} (integer p, integer min, integer max) {
  @   case so_this{L}:
  @     \forall integer min, max, size;
  @          size_of_obj{L}(min, size)
  @      ==> good_obj{L}(min+(OBJ_HEADER_SIZE+size), max)
  @      ==> scan_obj{L}(min, min, max);
  @   case so_not_this{L}:
  @     \forall integer p, min, max, size;
  @          size_of_obj{L}(min, size)
  @      ==> scan_obj{L}(p, min+(OBJ_HEADER_SIZE+size), max)
  @      ==> scan_obj{L}(p, min, max);
```

```
  @ }
  @*/
```

Το κατηγόρημα scan_obj δηλώνει πως το αντικείμενο που βρίσκεται στη θέση p είναι έγκυρο αντικείμενο της περιοχής min-max.

```
/*@ predicate new_obj{L} (integer p) =
  @   scan_obj{L}(p-OBJ_HEADER_SIZE, hpmin, hp);
  @*/
```

Το κατηγόρημα new _obj δηλώνει πως το αντικείμενο που βρίσκεται στη θέση p βρίσκεται στο ενεργό κομμάτι της μνήμης.

```
/*@ predicate old_obj{L} (integer p) =
  @   scan_obj{L}(p-OBJ_HEADER_SIZE, oldmin, oldhp);
  @*/
```

Το κατηγόρημα old _obj δηλώνει πως το αντικείμενο που βρίσκεται στη θέση p βρίσκεται στο μη ενεργό κομμάτι της μνήμης.

```
  //@ predicate some_obj{L} (integer p) = old_obj{L}(p) ||
new_obj{L}(p);
```

Το κατηγόρημα old _obj δηλώνει πως το αντικείμενο που βρίσκεται στη θέση p βρίσκεται στο ενεργό ή στο μη ενεργό κομμάτι της μνήμης.

Ορισμός τύπου

```
//@ type value = Num (integer) | Ptr (integer);
```

```
/*@ inductive word_value{L} (integer w, value v) {
  @   case wv_num{L}:
  @     \forall integer w;
  @       !IS_OFFSET(w) ==> word_value{L}(w, Num(OF_WORD(w)));
  @   case wv_ptr{L}:
  @     \forall integer w, p, i;
  @           IS_OFFSET(w)
  @       ==> p == OF_WORD(w)
  @       ==> id_of_obj(p-OBJ_HEADER_SIZE, i)
  @       ==> word_value{L}(w, Ptr(i));
  @ }
  @*/
```

Το κατηγόρημα word_value δηλώνει πως η τιμή ενός δεδομένου είναι ίση με το περιεχόμενό του, αν είναι αριθμός, ή, αν είναι δείκτης, ίση με τον αριθμό ταυτότητας του αντικειμένου που δείχνει.

Κατηγορήματα που αφορούν συνδέσεις αντικειμένων

Ορισμός τύπου

```
//@ type path = Nil | Cons (integer, path);
```

Ορίζουμε έναν τύπο path. Ο path θα είναι μια λίστα από αριθμούς. Η λίστα αυτή δηλώνει ένα μονοπάτι που μπορούμε να ακολουθήσουμε, με αφετηρία το ν-οστό δεδομένο ενός αντικειμένου, όπου ν ο πρώτος αριθμός της λίστας.

```
/*@ inductive lookup{L} (integer w, path l, value v) {
  @   case lk_none{L}:
  @     \forall integer min, w, value v;
  @          word_value{L}(w, v)
  @      ==> lookup{L}(w, Nil, v);
  @   case lk_some{L}:
  @     \forall integer w, p, i, ww, path l, value v;
  @          IS_OFFSET(w)
  @      ==> p == OF_WORD(w)
  @      ==> some_obj{L}(p)
  @      ==> content_of_obj(p, i, ww)
  @      ==> lookup{L}(ww, l, v)
  @      ==> lookup{L}(w, Cons(i, l), v);
  @ }
  @*/
```

Το κατηγόρημα lookup δηλώνει πως αν ξεκινώντας από το δεδομένο w ακολουθήσουμε ένα μονοπάτι l θα πάρουμε μια τιμή v.

```
/*@ predicate same_reachable{L1, L2} =
  @   \forall path l, value v;
  @         \at(lookup(root, l, v), L1)
  @    <==> \at(lookup(root, l, v), L2);
  @*/
```

Το κατηγόρημα same_reachable δηλώνει πως αν ξεκινώντας από την ρίζα, όλα τα ίδια μονοπάτια θα δίνουν τις ίδιες τιμές ανάμεσα σε ένα στιγμιότυπο της μνήμης L1 και L2.

```
/*@ predicate is_reachable{L} (integer r, integer i) =
```

```
  @   \exists path l;
  @     \at(lookup(r, l, Ptr(i)), L);
  @*/
```

Το κατηγόρημα is_reachable δηλώνει πως υπάρχει μονοπάτι που να ενώνει το αντικείμενο που βρίσκεται στη θέση r με το αντικείμενο που έχει τιμή i

```
/*@ predicate all_reachable{L}(integer r,integer min,integer max) =
  @   \forall integer p, i;
  @         scan_obj{L}(p, min, max)
  @     ==> id_of_obj{L}(p, i)
  @     ==> is_reachable{L}(r, i);
  @*/
```

Το κατηγόρημα all _reachable δηλώνει πως για όλα τα αντικείμενα, που βρίσκονται ανάμεσα στις θέσεις min και max, υπάρχει μονοπάτι που να τα ενώνει με το r.

Κατηγορήματα που αφορούν αναλλοίωτες

```
/*@ inductive size_of_old{L}(integer min,integer max,integer total)
  @{
  @   case so_none{L}:
  @     \forall integer min;
  @       size_of_old{L}(min, min, 0);
  @   case so_reg{L}:
  @     \forall integer min, max, size, total;
  @          \at(heap[min], L) != FORWARDED
  @      ==> \at(heap[min], L) == size
  @      ==> size_of_old{L}(min+(OBJ_HEADER_SIZE+size), max, total)
  @      ==> size_of_old{L}(min, max, size+total);
  @   case so_fwd{L}:
  @     \forall integer min, max, q, size, total;
  @          \at(heap[min], L) == FORWARDED
  @      ==> \at(heap[min+1], L) == q
  @      ==> \at(heap[q-OBJ_HEADER_SIZE], L) == size
  @      ==> size_of_old{L}(min+(OBJ_HEADER_SIZE+size), max, total)
  @      ==> size_of_old{L}(min, max, total);
  @ }
  @*/
```

Το κατηγόρημα size_of_old δηλώνει πως το συνολικό μέγεθος μνήμης που καταλαμβάνουν τα αντικείμενα, τα οποία δεν έχουν συλλεχθεί ακόμα και βρίσκονται ανάμεσα στις θέσεις min και max, είναι total.

```
/*@ predicate sane_heap (integer min, integer hp, integer max) =
  @     min <= hp <= max
  @  && (min == 0 || min == SIZE_OF_SPACE)
  @  && max == min + SIZE_OF_SPACE;
  @*/
```

Το κατηγόρημα sane_heap δηλώνει πως ο δρομέας της μνήμης θα πρέπει να έχει τιμές ανάμεσα των min και max. Επίσης, ότι η αρχή του κομματιού της μνήμης θα πρέπει να είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του ενεργού κομματιού της μνήμης πρέπει να είναι το μισό μέγεθος της συνολικής μνήμης.

```
/*@ predicate sanity =
  @     sane_heap(hpmin, hp, hpmax)
  @  && \valid_range(heap, 0, 2 * SIZE_OF_SPACE - 1)
  @  && 0 < id <= MAX_ID;
  @*
```

Το κατηγόρημα sane_heap δηλώνει πως η μνήμη είναι έγκυρη, πως ισχύουν όλα όσα περιγράφθηκαν από την sane_heap και ότι ο τρέχων αριθμός ταυτότητας είναι έγκυρος.

```
/*@ predicate point_to_old{L} (integer q) =
  @   \forall integer i, w, p;
  @         content_of_obj{L}(q-OBJ_HEADER_SIZE, i, w)
  @     ==> IS_OFFSET(w)
  @     ==> p == OF_WORD(w)
  @     ==> old_obj{L}(p);
  @*/
```

Το κατηγόρημα point_to_old δηλώνει πως το αντικείμενο που βρίσκεται στην θέση q, αν περιέχει δείκτες τότε όσοι οι δείκτες αυτοί δείχνουν σε αντικείμενα που είναι στο μη ενεργό κομμάτι της μνήμης.

```
/*@ predicate point_to_new{L} (integer q) =
  @   \forall integer i, w, p;
  @         content_of_obj{L}(q-OBJ_HEADER_SIZE, i, w)
  @     ==> IS_OFFSET(w)
  @     ==> p == OF_WORD(w)
  @     ==> new_obj{L}(p);
  @*/
```

Το κατηγόρημα point_to_new δηλώνει πως το αντικείμενο που βρίσκεται στην θέση q, αν περιέχει δείκτες τότε όσοι οι δείκτες αυτοί δείχνουν σε αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης.

```
/*@ predicate all_point_to_old{L} (integer min, integer max) =
  @   \forall integer p;
  @         scan_obj{L}(p, min, max)
  @     ==> point_to_old{L}(p+OBJ_HEADER_SIZE);
  @*/
```

Το κατηγόρημα all_point_to_old δηλώνει πως όλα τα αντικείμενα, τα οποία βρίσκονται ανάμεσα στις θέσεις min και max, αν έχουν δείκτες τότε όλοι οι δείκτες αυτοί δείχνουν σε αντικείμενα, τα οποία βρίσκονται στο μη ενεργό κομμάτι της μνήμης.

```
/*@ predicate all_point_to_new{L} (integer min, integer max) =
  @   \forall integer p;
  @         scan_obj{L}(p, min, max)
  @     ==> point_to_new{L}(p+OBJ_HEADER_SIZE);
  @*/
```

Το κατηγόρημα all_point_to_new δηλώνει πως όλα τα αντικείμενα, τα οποία βρίσκονται ανάμεσα στις θέσεις min και max, αν έχουν δείκτες τότε όλοι οι δείκτες αυτοί δείχνουν σε αντικείμενα, τα οποία βρίσκονται στο ενεργό κομμάτι της μνήμης.

```
/*@ predicate all_not_forwarded{L} (integer min, integer max) =
  @   \forall integer p;
  @         scan_obj{L}(p, min, max)
  @     ==> \at(heap[p], L) != FORWARDED;
  @*/
```

Το κατηγόρημα all_not_forwarded δηλώνει πως όλα τα αντικείμενα, τα οποία βρίσκονται ανάμεσα στις θέσεις min και max, είναι έγκυρα αντικείμενα τα οποία δεν έχουν αντιγραφθεί από τον συλλέκτη.

```
/*@ predicate separated =
  @   oldmax == hpmin || hpmax == oldmin;
  @*/
```

Το κατηγόρημα separated δηλώνει πως τα δύο κομμάτια της μνήμης, ενεργό και μη ενεργό, δεν αλληλεπικαλύπτονται.

Λήμματα

Για την υποβοήθηση των prover δηλώθηκαν και χρησιμοποιήθηκαν τα παρακάτω λήμματα.

```
/*@ lemma bw_and_uint32:
  @   \forall integer x, y;
  @        0 <= x <= 4294967295 && 0 <= y <= 4294967295
  @    ==> 0 <= (x & y) <= 4294967295;
  @*/
```

Το λήμμα bw_and_uint32 αναφέρει ότι το αποτέλεσμα της λογικής πράξης and μεταξύ δύο ακεραίων, οι οποίοι δεν υπερβαίνουν την τιμή του MAX_INT (0xFFFFFFFF), δε θα υπερβαίνει και αυτό την τιμή MAX_INT.

```
/*@ lemma bw_or_uint32:
  @   \forall integer x, y;
  @        0 <= x <= 4294967295 && 0 <= y <= 4294967295
  @    ==> 0 <= (x | y) <= 4294967295;
  @*/
```

Το λήμμα bw_ or _uint32 αναφέρει ότι το αποτέλεσμα της λογικής πράξης or μεταξύ δύο ακεραίων, οι οποίοι δεν υπερβαίνουν την τιμή του MAX_INT (0xFFFFFFFF), δε θα υπερβαίνει και αυτό την τιμή MAX_INT.

```
/*@ lemma offset_is_offset:
  @   \forall integer p;
  @        0 <= p < 2 * SIZE_OF_SPACE
  @    ==> IS_OFFSET(WORD_OF_OFS(p));
  @*/
```

Το λήμμα offset_is_offset αναφέρει την ότι η μετατροπή ενός αριθμού σε δείκτη, με την βοήθεια των macro WORD_OF_OFS και IS_OFFSET, θα δώσει δείκτη.

```
/*@ lemma word_ofs_id:
  @   \forall integer p;
  @        0 <= p < 2 * SIZE_OF_SPACE
  @    ==> OF_WORD(WORD_OF_OFS(p)) == p;
  @*/
```

Το λήμμα word_ofs_id αναφέρει την ότι η μετατροπή ενός αριθμού σε δείκτη και μετά πάλι σε αριθμό, με την βοήθεια των macro WORD_OF_OFS και OF_WORD αντίστοιχα, θα δώσει τον αρχικό αριθμό

```
/*@ lemma ofs_word_id:
  @   \forall integer w;
  @        0 <= w <= 4294967295
  @    ==> IS_OFFSET(w)
  @    ==> WORD_OF_OFS(OF_WORD(w));
  @*/
```

Το λήμμα ofs_word_id αναφέρει την ότι η μετατροπή ενός δείκτη σε αριθμό και μετά πάλι σε δείκτη, με την βοήθεια των macro WORD_OF_OFS και OF_WORD αντίστοιχα, θα δώσει τον αρχικό δείκτη

```
/*@ lemma size_is_nonneg:
  @   \forall integer p, size;
  @        size_of_obj(p, size)
  @    ==> size >= 0;
  @*/
```

Το λήμμα size_is_nonneg αναφέρει ότι το μέγεθος ενός αντικειμένου δεν μπορεί να είναι αρνητικό

```
/*@ lemma good_sanity:
  @   \forall integer min, max;
  @     good_obj(min, max) ==> min <= max;
  @*/
```

Το λήμμα good_sanity αναφέρει ότι αν ανάμεσα στα min και max υπάρχουν αντικείμενα, τα οποία δεν περιέχουν κενά ανάμεσά τους, τότε το max είναι μεγαλύτερο του min.

```
/*@ lemma scan_sanity:
  @   \forall integer min, max, p;
  @         scan_obj(p, min, max)
  @     ==> min <= p && p + OBJ_HEADER_SIZE <= max;
  @*/
```

Το λήμμα scan_sanity αναφέρει ότι αν το αντικείμενο, που βρίσκεται στο σημείο p, είναι ανάμεσα στα min και max, τότε το max είναι μεγαλύτερο του p και αυτό είναι μεγαλύτερο του min.

```
/*@ lemma scan_implies_good:
  @   \forall integer min, max, p;
  @     scan_obj(p, min, max) ==> good_obj(min, max);
  @*/
```

Το λήμμα scan_implies_good αναφέρει ότι αν το αντικείμενο, που βρίσκεται στο σημείο p, είναι ανάμεσα στα min και max τότε, ανάμεσα στα min και max, τα αντικείμενα θα είναι τοποθετημένα στη μνήμη διαδοχικά και χωρίς κενά ανάμεσά τους.

```
/*@ lemma scan_of_empty:
  @   \forall integer min, p;
  @     !scan_obj(p, min, min);
  @*/
```

Το λήμμα scan_of_empty αναφέρει πως δεν μπορεί να υπάρξει αντικείμενο σε μια μνήμη με μηδενικό μέγεθος.

```
/*@ lemma scan_of_larger:
  @   \forall integer min, max, p, q;
  @         scan_obj(p, min, max)
  @     ==> scan_obj(q, p, max)
  @     ==> scan_obj(q, min, max);
  @*/
```

Το λήμμα scan_of_larger αναφέρει πως αν ένα αντικείμενο, που βρίσκεται στο σημείο p, είναι ανάμεσα στα min και max και ένα αντικείμενο, που βρίσκεται στη θέση q, είναι ανάμεσα στα p και max, τότε το δεύτερο αντικείμενο θα είναι ανάμεσα στα min και max.

```
/*@ lemma scan_of_smaller:
  @   \forall integer min, max, p, q;
  @         scan_obj(p, min, max)
  @     ==> scan_obj(q, min, max)
  @     ==> p <= q
  @     ==> scan_obj(q, p, max);
  @*/
```

Το λήμμα scan_of_smaller αναφέρει πως αν δύο αντικείμενα, που βρίσκονται στα σημεία p και q, είναι ανάμεσα στα min και max και το p <= q τότε το δεύτερο αντικείμενο θα είναι ανάμεσα στα p και max.

```
/*@ lemma good_of_next:
  @   \forall integer min, max, size;
  @         good_obj(min, max)
  @     ==> min < max
  @     ==> size_of_obj(min, size)
  @     ==> good_obj(min+(OBJ_HEADER_SIZE+size), max);
  @*/
```

Το λήμμα good_of_next αναφέρει ότι αν μεταξύ της θέσης min και της θέσης max υπάρχουν αντικείμενα, τα οποία είναι διαδοχικά τοποθετημένα χωρίς κενά ανάμεσα τους, τότε και ανάμεσα στη θέση του επόμενου αντικειμένου, από αυτού που βρίσκεται στη θέση min, και του max υπάρχουν αντικείμενα, τα οποία είναι διαδοχικά τοποθετημένα χωρίς κενά ανάμεσά τους

Διαδικασία απόδειξης

Τρεις είναι οι συναρτήσεις που υλοποιούν τον συλλέκτη σκουπιδιών, η forward, η scanobj και η gc. Επιλέχτηκε, για την διαδικασία της απόδειξης, να γίνει πρώτα η απόδειξη της συνάρτησης gc, στη συνέχεια της scanobj και τέλος της forward. Δηλαδή, ξεκινώντας πρώτα από το γενικότερο και σταδιακά εξειδικεύοντας (top-down).

Απόδειξη της συνάρτησης gc

Η συνάρτηση gc, όπως έχει ήδη αναφερθεί, είναι αυτή που υλοποιεί τον συλλέκτη. Επομένως, οι υποχρεώσεις του συλλέκτη θα είναι ίδιες με αυτές της συνάρτησης gc.

Όπως έχει ήδη περιγραφεί οι απαιτήσεις από τον συλλέκτη είναι:

- Ο ελεύθερος χώρος μετά την συλλογή σκουπιδιών θα είναι περισσότερος ή το πολύ ίσος με τον ελεύθερο χώρο που υπήρχε πριν την συλλογή.
- Κάθε αντικείμενο θα πρέπει να περιέχει τα ίδια δεδομένα με αυτά που είχε πριν την συλλογή. Επίσης οι θέσεις των δεδομένων θα πρέπει να είναι η ίδια.
- Όλα τα αντικείμενα που πριν την συλλογή ήταν συνδεδεμένα με τις ρίζες, μετά την συλλογή θα πρέπει να βρίσκονται στην ενεργή μνήμη.
- Όλα τα αντικείμενα που πριν τη συλλογή δεν ήταν συνδεδεμένα με τις ρίζες, στο τέλος της συλλογής δε θα πρέπει να βρίσκονται στην ενεργή περιοχή.
- Μετά την συλλογή σκουπιδιών θα πρέπει το κάθε αντικείμενο της μνήμης να είναι συνδεδεμένο με τις ρίζες.
- Τέλος, δεν θα πρέπει να υπάρχουν δείκτες από την ενεργή μνήμη που να δείχνουν στην ανενεργή.

Οι παραπάνω απαιτήσεις θα πρέπει να περάσουν και σαν απαιτήσεις για την συνάρτηση που υλοποιεί τον συλλέκτη. Έτσι ο συλλέκτης μετά το τέλος της εργασίας του θα πρέπει να έχει εξασφαλίσει τις απαιτήσεις αυτές. Ουσιαστικά αυτές θα είναι οι μετασυνθήκες για την συνάρτηση gc. Αυτές ορίζονται όπως φαίνεται παρακάτω:

Μετασυνθήκες

```
@ ensures  sanity;
```

Διασφαλίζει ότι η μνήμη μετά τη συλλογή είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, ότι η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ ensures  good_obj(hpmin, hp);
```

Διασφαλίζει ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι τοποθετημένα διαδοχικά χωρίς κενά ανάμεσά τους.

```
@ ensures  all_not_forwarded(hpmin, hp);
```

Διασφαλίζει ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι νέα αντικείμενα.

```
@ ensures  all_point_to_new(hpmin, hp);
```

Διασφαλίζει ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα αν έχουν δείκτες τότε αυτοί δείχνουν σε αντικείμενα που βρίσκονται στο ενεργό κομμάτι της μνήμης.

```
@ ensures  same_reachable{Pre, Here};
```

Διασφαλίζει ότι όποιο αντικείμενο ήταν προσβάσιμο πριν θα είναι και τώρα, δηλαδή δεν δημιούργησε ο ίδιος ο συλλέκτης σκουπίδια. Επίσης διασφαλίζει ότι όλα τα αντικείμενα έχουν την ίδια δομή με αυτήν που είχαν πριν τη συλλογή.

```
@ ensures  all_reachable(root, hpmin, hp);
```

Διασφαλίζει ότι όλα τα αντικείμενα είναι προσβάσιμα από την ρίζα

```
@ ensures  IS_OFFSET(root) ==> new_obj(OF_WORD(root));
```

Διασφαλίζει ότι αν η ρίζα δείχνει σε αντικείμενο τότε αυτό το αντικείμενο βρίσκεται στο ενεργό κομμάτι της μνήμης

```
@ ensures  hp - hpmin + oldleft == oldhp - oldmin;
```

Διασφαλίζει ότι ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος μετά την συλλογή, από αυτόν που ήταν πριν.

Προσυνθήκες

Για να αποδειχτεί ότι θα ισχύουν όλα τα παραπάνω μετά τη συλλογή, υπάρχει η απαίτηση να ισχύουν τα παρακάτω πριν ξεκινήσει η διαδικασία της συλλογής. Αυτά είναι οι προσυνθήκες για την συνάρτηση gc και είναι ορισμένες ως εξής:

```
@ requires sanity;
```

Απαιτείται ότι η μνήμη πριν τη συλλογή είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, ότι η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ requires good_obj(hpmin, hp);
```

Απαιτείται ότι, πριν τη συλλογή, στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι τοποθετημένα διαδοχικά χωρίς κενά ανάμεσά τους.

```
@ requires all_not_forwarded(hpmin, hp);
```

Απαιτείται ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι νέα αντικείμενα.

```
@ requires all_point_to_new(hpmin, hp);
```

Απαιτείται ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα αν έχουν δείκτες τότε αυτοί δείχνουν σε αντικείμενα που βρίσκονται στο ενεργό κομμάτι της μνήμης.

```
@ requires size_of_old(hpmin, hp, hp-hpmin);
```

Απαιτείται ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα έχουν συνολικό μέγεθος hp - hpmin.

```
@ requires IS_OFFSET(root) ==> new_obj(OF_WORD(root));
```

Απαιτείται ότι αν η ρίζα δείχνει σε αντικείμενο τότε αυτό το αντικείμενο βρίσκεται στο ενεργό κομμάτι της μνήμης

Κώδικας της συνάρτησης

```
void gc ()
{
  word scan;

  //@ ghost oldmin = hpmin;
  //@ ghost oldhp = hp;
  //@ ghost oldmax = hpmax;
  //@ ghost oldleft = oldhp - oldmin;

  //@ assert hpmax == SIZE_OF_SPACE
    @      || hpmax == 2*SIZE_OF_SPACE;
  hpmin = hpmax % (2*SIZE_OF_SPACE);
  //@ assert hpmin == 0 || hpmin == SIZE_OF_SPACE;
  hpmax = hpmin + SIZE_OF_SPACE;
  hp = scan = hpmin;

  if (IS_OFFSET(root)) {
    word p = OF_WORD(root);
    root = WORD_OF_OFS(forward(p));
  }

  //@ ghost before:
  /*@ loop invariant sanity;
    @ loop invariant separated;
    @ loop invariant hpmin <= scan <= hp <= hpmax;
    @ loop invariant hp - hpmin + oldleft == oldhp - oldmin;
    @ loop invariant size_of_old(oldmin, oldhp, oldleft);
    @ loop invariant all_not_forwarded(hpmin, hp);
    @ loop invariant all_point_to_new(hpmin, scan);
    @ loop invariant all_point_to_old(scan, hp);
    @ loop invariant same_reachable{before, Here};
    @ loop invariant all_reachable(root, hpmin, hp);
    @ loop variant   lexicographic(oldleft, hp-scan);
    @*/
  while (scan < hp)
    scan += scanobj(scan);
}
```

Πριν την έναρξη της διαδικασίας συλλογής οι τιμές των hpmin, hp και hpmax αποθηκεύονται αντίστοιχα στις φανταστικές μεταβλητές oldmin, oldhp, oldmax αντίστοιχα. Τέλος υπολογίζεται η φανταστική μεταβλητή oldleft που θα είναι ίση με τον καταλυμένο χώρο της μνήμης.

Για την βοήθεια της απόδειξης από τους αυτόματους prover γίνονται οι ισχυρισμοί //@ **assert** hpmax == SIZE_OF_SPACE || hpmax == 2*SIZE_OF_SPACE; και //@ **assert** hpmin == 0 || hpmin == SIZE_OF_SPACE; που αφορούν στο μέγεθος του ενεργού κομματιού της μνήμης.

Για τον βρόχο while δίνονται οι παρακάτω αναλλοίωτες

```
@ loop invariant sanity;
```

Μετά την κάθε επανάληψη, η μνήμη θα είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, ότι η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ loop invariant separated;
```

Το ενεργό και το ανενεργό κομμάτι της μνήμης δε θα αλληλεπικαλύπτονται.

```
@ loop invariant hpmin <= scan <= hp <= hpmax;
```

Η τιμή της scan θα είναι ίση ή μεγαλύτερη από την αρχή της μνήμης, μικρότερη από τον δρομέα και αυτός με τη σειρά του μικρότερος από το μέγιστο της μνήμης

```
@ loop invariant hp - hpmin + oldleft == oldhp - oldmin;
```

Τα νέα αντικείμενα στην ενεργή μνήμη, αυτά που δεν έχουν αντιγραφεί ακόμα και βρίσκονται στη παλιά μνήμη, και τα σκουπίδια που βρίσκονται στη παλιά μνήμη έχουν συνολικό μέγεθος όσο η μνήμη που χρησιμοποιούταν πριν τη συλλογή.

```
@ loop invariant size_of_old(oldmin, oldhp, oldleft);
```

Τα αντικείμενα που δεν έχουν αντιγραφεί ακόμα και βρίσκονται στη παλιά μνήμη, και τα σκουπίδια που βρίσκονται στη παλιά μνήμη έχουν συνολικό μέγεθος *oldleft*

```
@ loop invariant all_not_forwarded(hpmin, hp);
```

Στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι νέα αντικείμενα.

```
@ loop invariant all_point_to_new(hpmin, scan);
```

Όλα τα αντικείμενα που έχουν εξεταστεί με την scanobj εάν έχουν δείκτες, τότε αυτοί δείχνουν σε νέα αντικείμενα. Τα αντικείμενα αυτό βρίσκονται ανάμεσα στις θέσεις *hpmin και scan*

```
@ loop invariant all_point_to_old(scan, hp);
```

Όλα τα αντικείμενα που δεν έχουν εξεταστεί με την scanobj εάν έχουν δείκτες, τότε αυτοί δείχνουν σε αντικείμενα που βρίσκονται στο ανενεργό κομμάτι της μνήμης. Τα αντικείμενα αυτό βρίσκονται ανάμεσα στις θέσεις *scan και hp*

```
@ loop invariant same_reachable{before, Here};
```

Όποιο αντικείμενο ήταν προσβάσιμο πριν θα είναι και τώρα, δηλαδή δεν δημιούργησε ο ίδιος ο συλλέκτης σκουπίδια. Επίσης διασφαλίζει ότι όλα τα αντικείμενα έχουν την ίδια δομή με αυτήν που είχαν πριν τη συλλογή.

```
@ loop invariant all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα είναι προσβάσιμα από την ρίζα

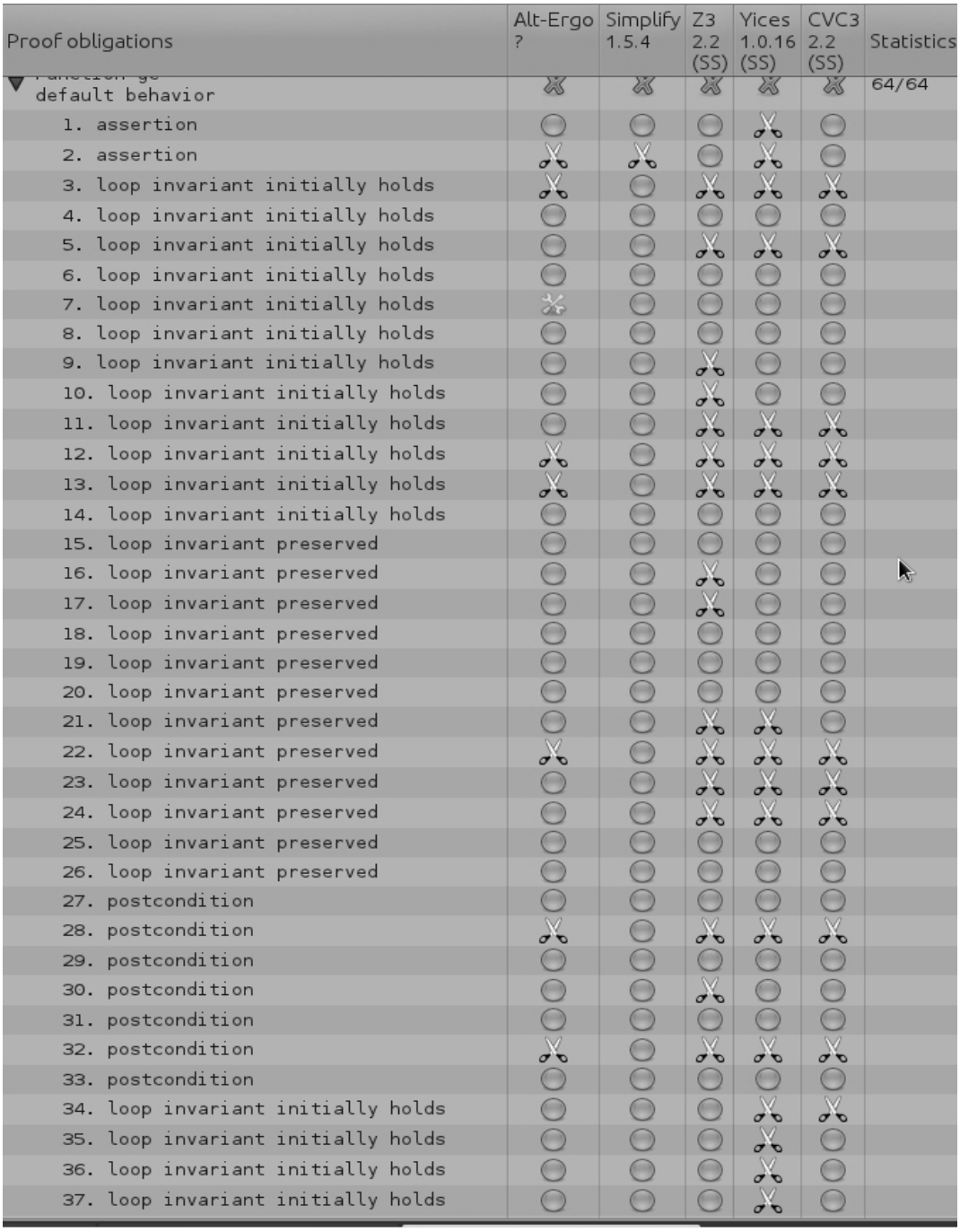
Η συνθήκη τερματισμού του βρόχου είναι όταν έχουν εξεταστεί όλα τα αντικείμενα που αντιγράφθηκαν

```
@ loop variant   lexicographic(oldleft, hp-scan);
```

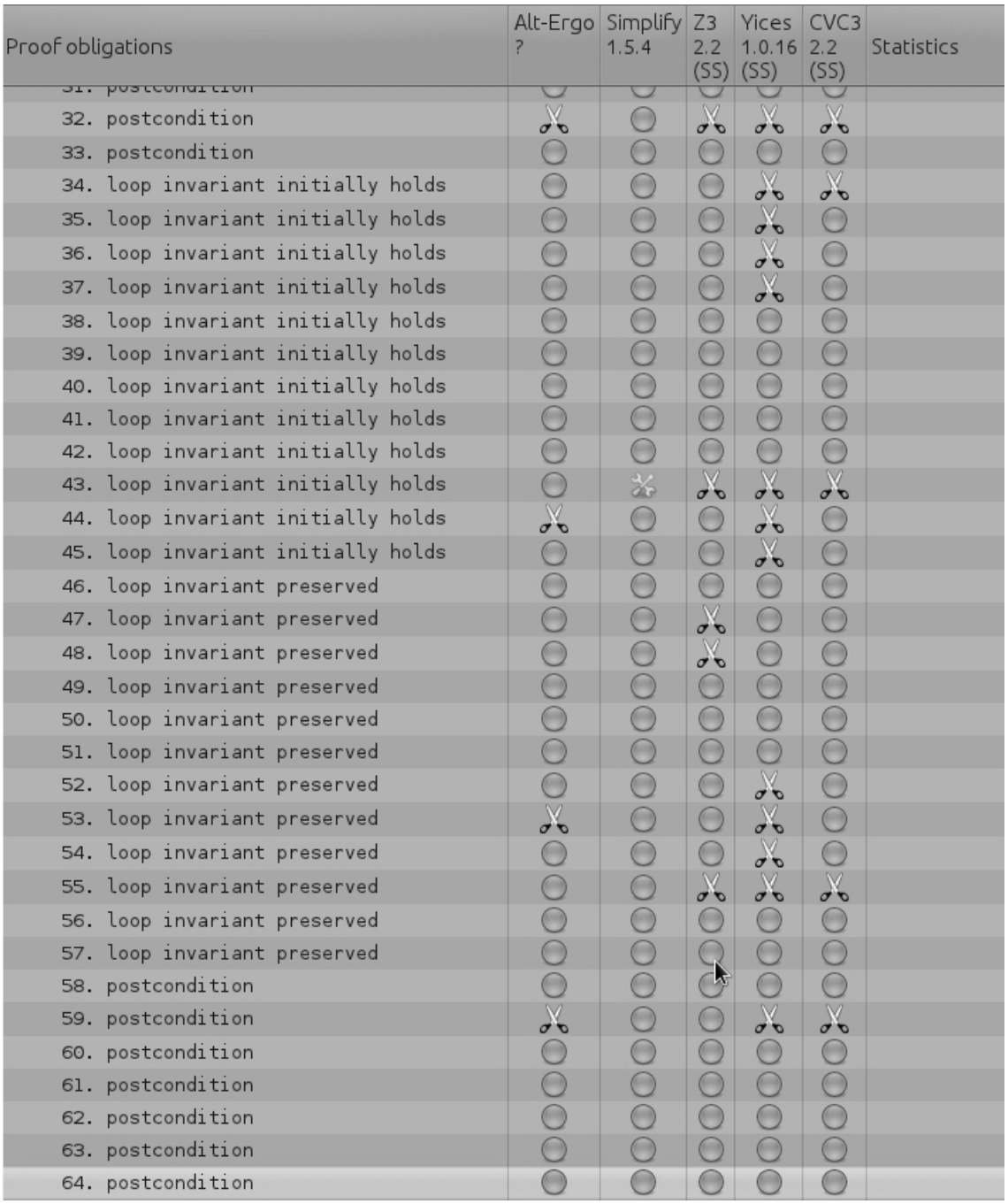
Όπου

```
/*@ logic integer lexicographic (integer x, integer y) =
  @   x * (SIZE_OF_SPACE+1) + y;
  @*/
```

Η εκτέλεση της Frama-c έδωσε την παρακάτω εικόνα



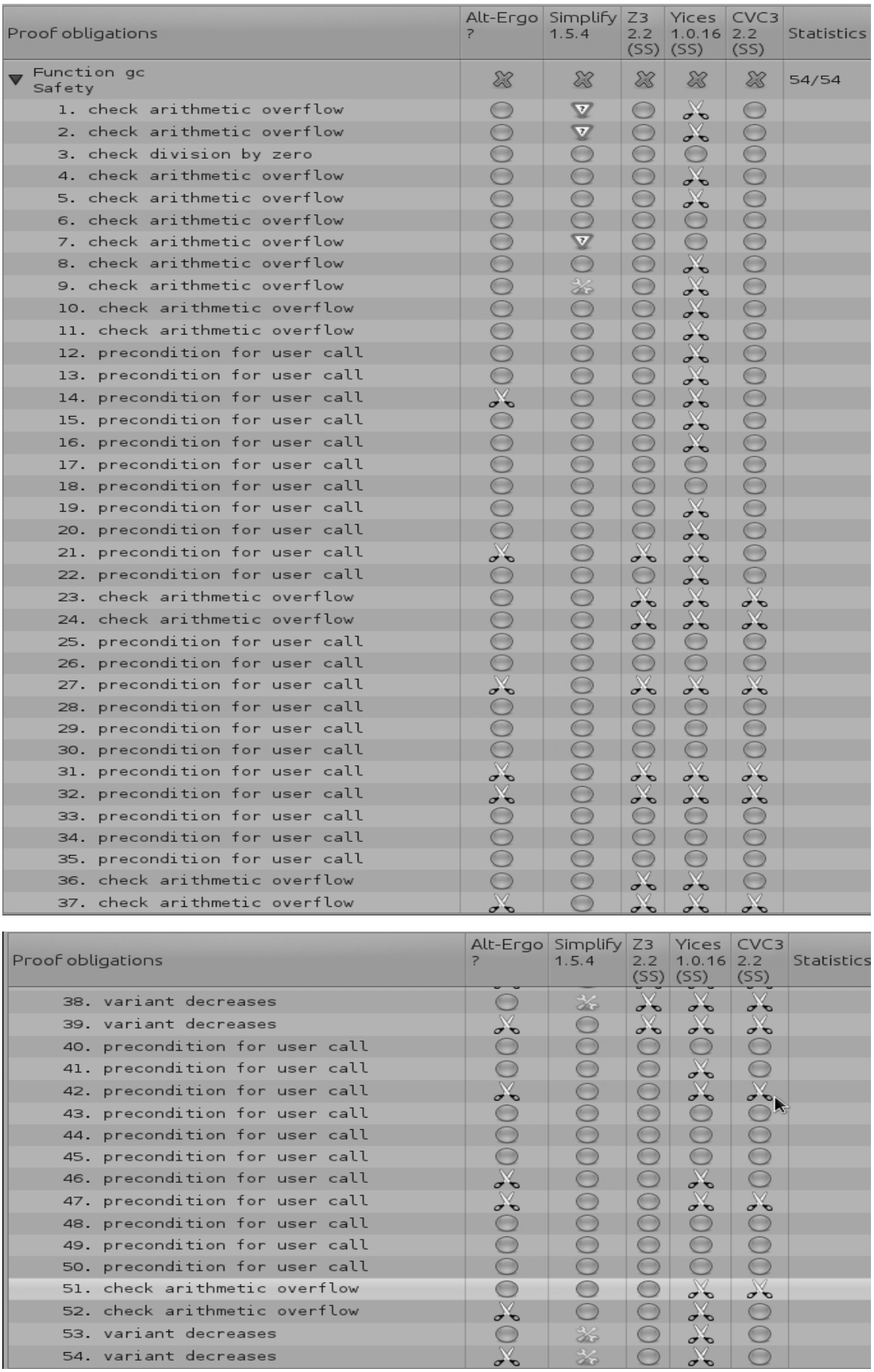
Εικόνα 8- Απόδειξη της συνάρτησης gc



Εικόνα 9- Απόδειξη της συνάρτησης gc (συνέχεια)

Παρήχθησαν 64 obligation τα οποία αποδείχθηκαν όλα. Οι prover τα απέδειξαν ως εξής: Ergo 54 από τα 64, Simplify 62 από τα 64, Z3 44 από τα 64, yices 39 από τα 64, CVC3 50 από τα 64

Επίσης για την ασφάλεια των δεδομένων (όπως υπερχειλίσεις κ.α.) παρήχθησαν 54 obligation τα οποία και αυτά αποδείχθηκαν όλα. Οι prover τα απέδειξαν ως εξής: Ergo 42 από τα 54, Simplify 47 από τα 54, Z3 44 από τα 54, yices 20 από τα 54, CVC3 43 από τα 54 όπως φαίνονται στην παρακάτω εικόνα.



Εικόνα 10Απόδειξη της συνάρτησης gc (ασφάλεια)

Απόδειξη της συνάρτησης scanobj

Η συνάρτηση scanobj, όπως έχει περιγραφθεί πιο πάνω, εξετάζει τα δεδομένα ενός αντικειμένου. Αν στα δεδομένα αυτά βρεθούν δείκτες σε αντικείμενα, τότε αντιγράφει τα αντικείμενα αυτά στην ενεργή μνήμη, αν αυτό δε έχει ήδη γίνει σε προηγούμενη φάση της συλλογής, και ενημερώνει τους δείκτες του ώστε να δείχνουν στην διεύθυνση στην οποία βρίσκονται τα νέα αντικείμενα.

Η συνάρτηση αυτή καλείτε να εξετάσει αντικείμενα τα οποία έχουν αντιγραφεί από τον συλλέκτη. Τα αντικείμενα αυτά βρίσκονται στο ενεργό κομμάτι της μνήμης. Κάθε αντικείμενο εξετάζεται από την συνάρτηση μια μόνο φορά.

Η συνάρτηση οφείλει, με την λειτουργία της, να εξασφαλίζει ότι:

- Μετά την εκτέλεσή της η μνήμη θα εξακολουθήσει να είναι έγκυρη, ο δρομέας της μνήμης θα έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, η αρχή του ενεργού κομματιού της μνήμης θα είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του θα είναι το μισό μέγεθος της συνολικής μνήμης
- Όποιο αντικείμενο ήταν προσβάσιμο πριν θα είναι και τώρα. όλα τα αντικείμενα έχουν την ίδια δομή με αυτήν που είχαν πριν την εκτέλεσή της.
- Ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος μετά την κλήση της, από αυτόν που ήταν πριν.
- Αν το αντικείμενο περιέχει δείκτες τότε όλοι οι δείκτες αυτοί μετά την κλήση της συνάρτησης θα δείχνουν σε αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης.

Μετασυνθήκες

```
@ ensures  sanity;
```

Διασφαλίζει ότι η μνήμη μετά τη κλήση είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, ότι η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ ensures  hp >= \old(hp);
```

Ο δρομέας είτε έμεινε ακίνητος, αν δεν έγιναν αντιγραφές, είτε προχώρησε.

```
@ ensures  same_reachable{Pre, Here};
```

Διασφαλίζει ότι όποιο αντικείμενο ήταν προσβάσιμο πριν θα είναι και τώρα. Επίσης διασφαλίζει ότι όλα τα αντικείμενα έχουν την ίδια δομή με αυτήν που είχαν πριν τη συλλογή.

```
@ ensures  all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα είναι προσβάσιμα από τη ρίζα

```
@ ensures  size_of_old(oldmin, oldhp, oldleft);
```

Το μέγεθος των αντικειμένων που δεν έχουν αντιγραφθεί και είναι στην ανενεργή μνήμη είναι oldleft

```
@ ensures  hp - hpmin + oldleft == oldhp - oldmin;
```

Διασφαλίζει ότι ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος μετά την κλήση, από αυτόν που ήταν πριν.

```
@ ensures  good_obj(hpmin, hp);
```

Τα αντικείμενα, που βρίσκονται στο ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ ensures  all_point_to_new(hpmin, q + \result);
```

Όλα τα αντικείμενα, τα οποία βρίσκονται από την αρχή της ενεργής μνήμης μέχρι και την θέση q, αν έχουν δείκτες, μετά την κλήση της συνάρτησης, αυτοί θα δείχνουν σε αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης

```
@ ensures  all_point_to_old(q + \result, hp);
```

Όλα τα αντικείμενα, τα οποία βρίσκονται από την θέση q μέχρι και τον δρομέα, αν έχουν δείκτες, μετά την κλήση της συνάρτησης, αυτοί θα δείχνουν σε αντικείμενα που είναι στο μη ενεργό κομμάτι της μνήμης. Δηλαδή η συνάρτηση δεν πείραξε αυτούς τους δείκτες

```
@ ensures  all_not_forwarded(hpmin, hp);
```

Διασφαλίζει ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι νέα αντικείμενα.

```
@ ensures  \result == OBJ_HEADER_SIZE + \old(heap[q]);
```

Διασφαλίζει ότι ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος μετά την συλλογή, από αυτόν που ήταν πριν.

Προσυνθήκες

Για να αποδειχτεί ότι θα ισχύουν όλα τα παραπάνω μετά τη κλήση της συνάρτησης scanobj, υπάρχει η απαίτηση να ισχύουν τα παρακάτω πριν την κλήση της συνάρτησης αυτής. Αυτά είναι οι προσυνθήκες για την συνάρτηση scanobj και είναι ορισμένες ως εξής:

```
/*@ requires sanity;
```

η μνήμη πριν την κλήση είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ requires sane_heap(oldmin, oldhp, oldmax);
@ requires good_obj(oldmin, oldhp);
```

Τα αντικείμενα, που βρίσκονται στο μη ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ requires separated;
```

Η ενεργή και η μη ενεργή μνήμη δεν αλληλεπικαλύπτονται

```
@ requires all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα στην ενεργή μνήμη είναι προσβάσιμα

```
@ requires size_of_old(oldmin, oldhp, oldleft);
```

Το συνολικό μέγεθος των αντικειμένων στο μη ενεργό κομμάτι της μνήμης, που δεν έχουν αντιγραφθεί, είναι oldleft.

```
@ requires hp - hpmin + oldleft == oldhp - oldmin;
```

Τα νέα αντικείμενα στην ενεργή μνήμη, αυτά που δεν έχουν αντιγραφεί ακόμα και βρίσκονται στη παλιά μνήμη, και τα σκουπίδια που βρίσκονται στη παλιά μνήμη έχουν συνολικό μέγεθος όσο η μνήμη που χρησιμοποιούταν πριν τη συλλογή.

```
@ requires new_obj(q+OBJ_HEADER_SIZE);
```

Το αντικείμενο που εξετάζεται να είναι στην ενεργή περιοχή της μνήμης

```
@ requires q < hp;
@ requires good_obj(hpmin, hp);
```

Τα αντικείμενα, που βρίσκονται στο ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ requires all_point_to_old(q, hp);
```

Αν το αντικείμενο, το οποίο εξετάζεται, έχει δείκτες τότε αυτοί βρίσκονται στο μη ενεργό κομμάτι της μνήμης.

```
@ requires all_point_to_new(hpmin, q);
```

Όλα τα αντικείμενα που έχουν ήδη εξεταστεί αν έχουν δείκτες, αυτοί δείχνουν στο ενεργό κομμάτι της μνήμης.

```
@ requires all_not_forwarded(hpmin, hp);
```

Όλα τα αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης είναι νέα αντικείμενα.

Κώδικας της συνάρτησης

```
word scanobj (word q)
{
  word size = heap[q];
  word i;

  //@ assert size != FORWARDED;
  //@ assert size_of_obj(q, size);
  //@ assert hpmin <= q;
  //@ assert q + OBJ_HEADER_SIZE + size <= hp;
  //@ assert point_to_old(q+OBJ_HEADER_SIZE);

  //@ ghost before:
  /*@ loop invariant 0 <= i <= size;
    @ loop invariant sanity;
    @ loop invariant sane_heap(oldmin, oldhp, oldmax);
    @ loop invariant separated;
    @ loop invariant hp >= \at(hp, before);
    @ loop invariant same_reachable{before, Here};
    @ loop invariant all_reachable(root, hpmin, hp);
    @ loop invariant size_of_old(oldmin, oldhp, oldleft);
    @ loop invariant hp - hpmin + oldleft == oldhp - oldmin;
    @ loop invariant good_obj(hpmin, hp);
    @ loop invariant all_point_to_new(hpmin, q);
    @ loop invariant all_point_to_old(q+OBJ_HEADER_SIZE+size,
    @                                 \at(hp, before));
    @ loop invariant all_point_to_old(\at(hp, before), hp);
    @ loop invariant (\forall integer k, w, p;
    @                       0 <= k < i
    @                   ==> content_of_obj(q, k, w)
    @                   ==> IS_OFFSET(w)
    @                   ==> p == OF_WORD(w)
    @                   ==> new_obj(p));
    @ loop invariant (\forall integer k, w, p;
    @                       i <= k < size
    @                   ==> content_of_obj(q, k, w)
    @                   ==> IS_OFFSET(w)
    @                   ==> p == OF_WORD(w)
    @                   ==> old_obj(p));
    @ loop invariant size_of_obj(q, size);
    @ loop invariant heap[q] == size;
    @
    @
    @ loop invariant \forall integer kk;
    @               q+OBJ_HEADER_SIZE+size <= kk < \at(hp, before)
    @           ==> \at(heap[kk], before) == \at(heap[kk], Here);
    @ loop assigns   hp, oldleft;
    @ loop assigns   heap[oldmin..oldhp-1];
    @ loop assigns   heap[q+OBJ_HEADER_SIZE..q+OBJ_HEADER_SIZE+i-
1];
    @ loop assigns   heap[\at(hp, before)..hpmax-1];
    @ loop variant   size - i;
```

```
    @*/
  for (i=0; i<size; i++)
    if (IS_OFFSET(heap[q + OBJ_HEADER_SIZE + i])) {
      word p = OF_WORD(heap[q + OBJ_HEADER_SIZE + i]);
      /*@ assert content_of_obj(q, i,
        @                       heap[q + OBJ_HEADER_SIZE + i]);
        @*/
      //@ assert old_obj(p);
      //@ assert oldmin <= p-OBJ_HEADER_SIZE;
      //@ assert p <= oldhp;
      word t = forward(p);
     /*@ assert (\forall integer k, w, p;
       @            i <= k < size
       @                   ==> heap[q + OBJ_HEADER_SIZE + k] == w
       @                   ==> IS_OFFSET(w)
       @                   ==> p == OF_WORD(w)
       @                   ==> old_obj(p));
       @*/

      //@ assert q != p-OBJ_HEADER_SIZE;
      //@ assert q != p-OBJ_HEADER_SIZE+1;
      //@ assert heap[q] == size;
      heap[q + OBJ_HEADER_SIZE + i] = WORD_OF_OFS(t);
    }
  //@ assert point_to_new(q+OBJ_HEADER_SIZE);
  return OBJ_HEADER_SIZE + size;
}
```

Για τον βρόχο for δίνονται οι παρακάτω αναλλοίωτες

```
    @ loop invariant 0 <= i <= size;
```

Το i μπορεί να πάρει τιμές από 0 έως size

```
    @ loop invariant sanity;
```

Η μνήμη διατηρεί την εγκυρότητά της

```
    @ loop invariant sane_heap(oldmin, oldhp, oldmax);
    @ loop invariant separated;
```

Οι δύο περιοχές της μνήμης δεν αλληλεπικαλύπτονται

```
    @ loop invariant hp >= \at(hp, before);
```

Ο δρομέας έχει τιμή ίση ή μεγαλύτερη από αυτή που είχε αρχικά

```
    @ loop invariant same_reachable{before, Here};
```

Η δομή της μνήμης δεν έχει αλλάξει

```
    @ loop invariant all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα είναι προσβάσιμα από τη ρίζα

```
    @ loop invariant size_of_old(oldmin, oldhp, oldleft);
```

Το συνολικό μέγεθος των αντικειμένων στο μη ενεργό κομμάτι της μνήμης, που δεν έχουν αντιγραφθεί, είναι oldleft.

```
@ loop invariant hp - hpmin + oldleft == oldhp - oldmin;
```

ότι ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος σε κάθε επανάληψη, από αυτόν που ήταν πριν.

```
@ loop invariant good_obj(hpmin, hp);
```

Τα αντικείμενα, που βρίσκονται στο ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ loop invariant all_point_to_new(hpmin, q);
```

Όλα τα αντικείμενα, τα οποία βρίσκονται από την αρχή της ενεργής μνήμης μέχρι και την θέση q, αν έχουν δείκτες, μετά την κλήση της συνάρτησης, αυτοί θα δείχνουν σε αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης

```
@ loop invariant all_point_to_old(q+OBJ_HEADER_SIZE+size,
@                                 \at(hp, before));
```

Όλα τα αντικείμενα, τα οποία βρίσκονται από τη θέση του επόμενου αντικειμένου από αυτό της θέσης q μέχρι και τον δρομέα, αν έχουν δείκτες, αυτοί θα δείχνουν σε αντικείμενα που είναι στο μη ενεργό κομμάτι της μνήμης.

```
@ loop invariant all_point_to_old(\at(hp, before), hp);
```

Στο κομμάτι της μνήμης από hp (πριν την αρχή του βρόχου) μέχρι το τρέχων hp τα αντικείμενα αν έχουν δείκτες, αυτοί θα δείχνουν σε αντικείμενα που είναι στο μη ενεργό κομμάτι της μνήμης.

```
@ loop invariant (\forall integer k, w, p;
@                       0 <= k < i
@                   ==> content_of_obj(q, k, w)
@                   ==> IS_OFFSET(w)
@                   ==> p == OF_WORD(w)
@                   ==> new_obj(p));
```

Όλοι οι δείκτες τους οποίους έχει περάσει ο βράχος δείχνουν σε νέα αντικείμενα

```
@ loop invariant (\forall integer k, w, p;
@                       i <= k < size
@                   ==> content_of_obj(q, k, w)
@                   ==> IS_OFFSET(w)
@                   ==> p == OF_WORD(w)
@                   ==> old_obj(p));
```

Όλοι οι δείκτες τους οποίους δεν έχει περάσει ο βράχος δείχνουν σε αντικείμενα στο ανενεργό κομμάτι της μνήμης

Η συνθήκη τερματισμού του βρόχου είναι

```
@ loop variant   size - i;
```

Για την υποβοήθηση των prover δόθηκαν οι παρακάτω ισχυρισμοί

Αρχικά

```
//@ assert size != FORWARDED;
```

Το αντικείμενο δε έχει αντιγραφεί σε άλλο

```
//@ assert size_of_obj(q, size);
```

Το μέγεθος του αντικειμένου είναι ίσο με την τιμή της μεταβλητής size

```
//@ assert hpmin <= q;
```

Τα αντικείμενο είναι μετά την αρχή του ενεργού κομματιού της μνήμης

```
//@ assert q + OBJ_HEADER_SIZE + size <= hp;
```

Το μέγεθος του αντικειμένου δεν υπερβαίνει τη θέση του δρομέα

```
//@ assert point_to_old(q+OBJ_HEADER_SIZE);
```

Αν έχει δείκτες τότε αυτοί δείχνουν σε αντικείμενα στο ανενεργό κομμάτι της μνήμης

Μέσα στο βρόχο πριν την κλήση της forward

```
/*@ assert content_of_obj(q, i,
  @                       heap[q + OBJ_HEADER_SIZE + i]);
  @*/
```

Το τρέχον δεδομένο ανήκει στα δεδομένα του αντικειμένου

```
//@ assert old_obj(p);
```

Το αντικείμενο στο οποίο δείχνει ο δείκτης είναι στην ανενεργή μνήμη

```
//@ assert oldmin <= p-OBJ_HEADER_SIZE;
//@ assert p <= oldhp;
```

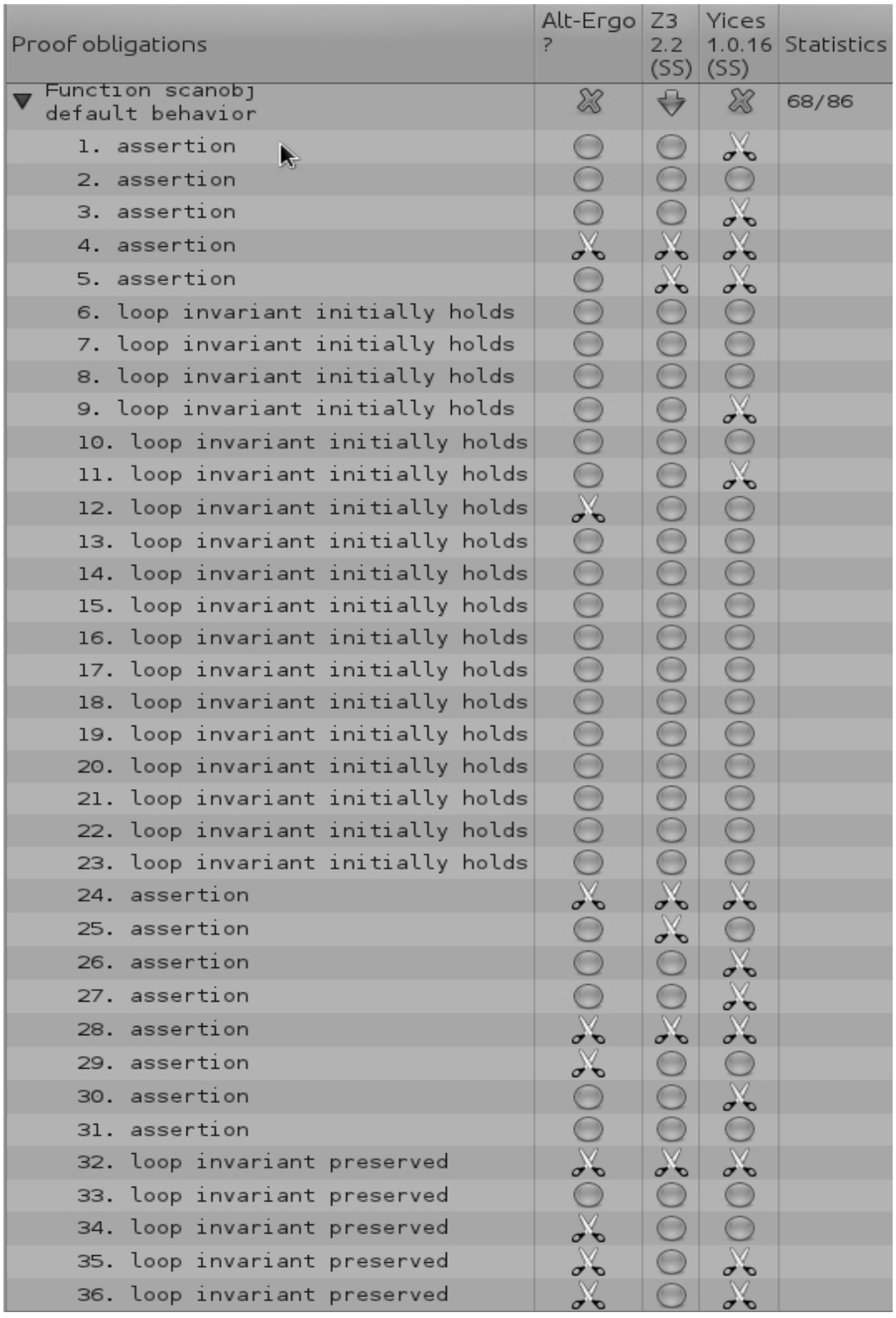
Το αντικείμενο στο οποίο δείχνει ο δείκτης χώραγε μέσα στο ανενεργό κομμάτι της μνήμης

Μέσα στο βρόχο μετά την κλήση της forward

```
/*@ assert (\forall integer k, w, p;
  @             i < k < size
  @                    ==> heap[q + OBJ_HEADER_SIZE + k] == w
  @                    ==> IS_OFFSET(w)
  @                    ==> p == OF_WORD(w)
  @                    ==> old_obj(p));
  @*/
```

Όλοι οι δείκτες οι οποίοι δεν έχουν εξεταστεί δείχνουν στο ανενεργό κομμάτι της μνήμης

Η εκτέλεση της Frama-c έδωσε την παρακάτω εικόνα

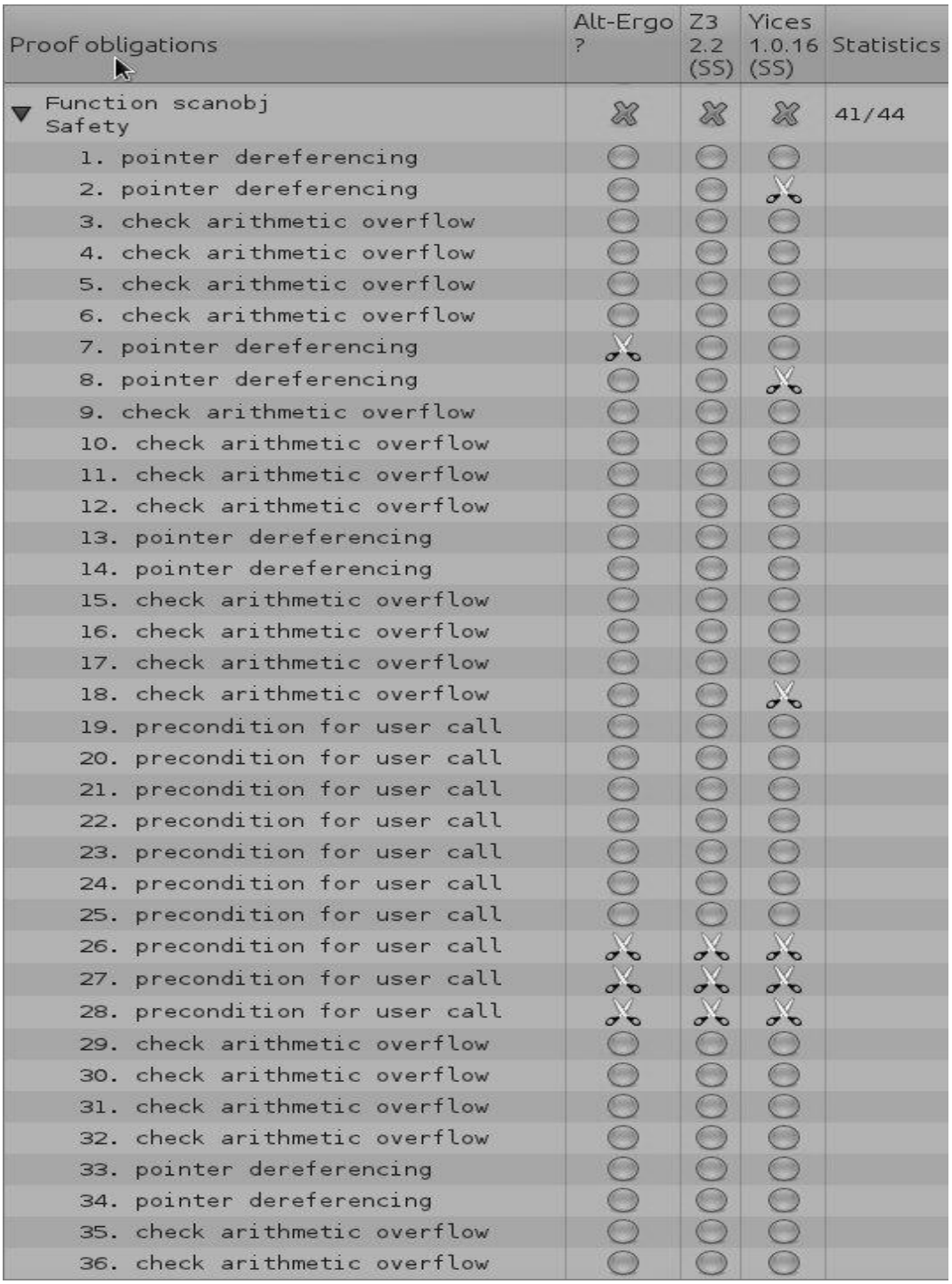


Εικόνα 11- Αποτελέσματα απόδειξης συνάρτησης scanobj

Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
37. loop invariant preserved	✂	✂	✂	
38. loop invariant preserved	✂	✂	✂	
39. loop invariant preserved	✂	✂	✂	
40. loop invariant preserved	✂	✂	✂	
41. loop invariant preserved	✂	✂	✂	
42. loop invariant preserved	✂	✂	✂	
43. loop invariant preserved	✂	✂	✂	
44. loop invariant preserved	○	○	○	
45. loop invariant preserved	✂	✂	✂	
46. loop invariant preserved	✂	✂	✂	
47. loop invariant preserved	○	○	○	
48. loop invariant preserved	○	○	○	
49. loop invariant preserved	○	○	○	
50. loop invariant preserved	○	○	○	
51. loop invariant preserved	✂	✂	✂	
52. loop invariant preserved	✂	○	✂	
53. loop invariant preserved	○	○	○	
54. loop invariant preserved	○	○	○	
55. loop invariant preserved	○	○	○	
56. loop invariant preserved	○	○	○	
57. loop invariant preserved	○	○	○	
58. loop invariant preserved	○	○	○	
59. loop invariant preserved	○	○	○	
60. loop invariant preserved	✂	✂	✂	
61. loop invariant preserved	○	○	○	
62. loop invariant preserved	○	○	○	
63. loop invariant preserved	○	○	○	
64. loop invariant preserved	○	○	○	
65. loop invariant preserved	○	○	○	
66. loop invariant preserved	○	○	○	
67. loop invariant preserved	○	○	○	
68. loop invariant preserved	○	○	○	
69. loop invariant preserved	○	○	○	
70. loop invariant preserved	○	○	○	
71. loop invariant preserved	○	○	○	
72. loop invariant preserved	✂	○	✂	
73. loop invariant preserved	○	○	○	

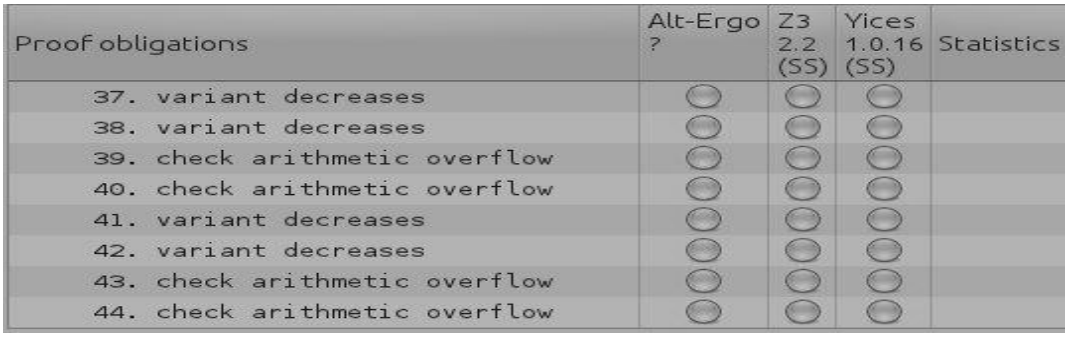
Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
74. assertion	○	○	✂	
75. postcondition	○	○	○	
76. postcondition	○	○	○	
77. postcondition	○	○	○	
78. postcondition	○	○	○	
79. postcondition	○	○	○	
80. postcondition	○	○	○	
81. postcondition	○	○	○	
82. postcondition	✂	✂	✂	
83. postcondition	✂	✂	✂	
84. postcondition	✂	✂	✂	
85. postcondition	○	○	○	
86. postcondition	○	○	○	

Εικόνα 12- Αποτελέσματα απόδειξης συνάρτησης scanobj (συνέχεια)



Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
Function scanobj Safety	✗	✗	✗	41/44
1. pointer dereferencing	✓	✓	✓	
2. pointer dereferencing	✓	✓	✂	
3. check arithmetic overflow	✓	✓	✓	
4. check arithmetic overflow	✓	✓	✓	
5. check arithmetic overflow	✓	✓	✓	
6. check arithmetic overflow	✓	✓	✓	
7. pointer dereferencing	✂	✓	✓	
8. pointer dereferencing	✓	✓	✂	
9. check arithmetic overflow	✓	✓	✓	
10. check arithmetic overflow	✓	✓	✓	
11. check arithmetic overflow	✓	✓	✓	
12. check arithmetic overflow	✓	✓	✓	
13. pointer dereferencing	✓	✓	✓	
14. pointer dereferencing	✓	✓	✓	
15. check arithmetic overflow	✓	✓	✓	
16. check arithmetic overflow	✓	✓	✓	
17. check arithmetic overflow	✓	✓	✓	
18. check arithmetic overflow	✓	✓	✂	
19. precondition for user call	✓	✓	✓	
20. precondition for user call	✓	✓	✓	
21. precondition for user call	✓	✓	✓	
22. precondition for user call	✓	✓	✓	
23. precondition for user call	✓	✓	✓	
24. precondition for user call	✓	✓	✓	
25. precondition for user call	✓	✓	✓	
26. precondition for user call	✂	✂	✂	
27. precondition for user call	✂	✂	✂	
28. precondition for user call	✂	✂	✂	
29. check arithmetic overflow	✓	✓	✓	
30. check arithmetic overflow	✓	✓	✓	
31. check arithmetic overflow	✓	✓	✓	
32. check arithmetic overflow	✓	✓	✓	
33. pointer dereferencing	✓	✓	✓	
34. pointer dereferencing	✓	✓	✓	
35. check arithmetic overflow	✓	✓	✓	
36. check arithmetic overflow	✓	✓	✓	

Εικόνα 13- Αποτελέσματα απόδειξης συνάρτησης scanobj (ασφάλεια)



Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
37. variant decreases	✓	✓	✓	
38. variant decreases	✓	✓	✓	
39. check arithmetic overflow	✓	✓	✓	
40. check arithmetic overflow	✓	✓	✓	
41. variant decreases	✓	✓	✓	
42. variant decreases	✓	✓	✓	
43. check arithmetic overflow	✓	✓	✓	
44. check arithmetic overflow	✓	✓	✓	

Εικόνα 14- Αποτελέσματα απόδειξης συνάρτησης scanobj (ασφάλειας συνέχεια)

Παρήχθησαν 84 obligation από τα οποία αποδείχθηκαν τα 68 ενώ για τν ασφάλεια παρήχθησαν 44 από τα οποία αποδείχθηκαν τα 41

Μετά από πολλές δοκιμές που έγιναν σε αυτόν τον κώδικα αλλά και σε άλλους που δοκιμάστηκαν βγήκε το εξής συμπέρασμα:

Το Frama-C δεν μπορεί να παράγει αποτελεσματικό τρόπο τα obligation που αφορούν στη μνήμη, που επηρεάζεται μέσα στο βρόχο. Παρότι σε πολλές δοκιμές δόθηκε με μεγάλη λεπτομέρεια τι επηρεάζεται στη μνήμη και τι όχι, οι prover δεν κατάφεραν να αποδείξουν οτιδήποτε είχε να κάνει με την δομή της μνήμης

Απόδειξη της συνάρτησης forward

Η συνάρτηση forward αντιγράφει ένα αντικείμενο από την μη ενεργή μνήμη στην ενεργή. Το αρχικό αντικείμενο σημαίνεται ότι έχει αντιγραφθεί και δείχνει τη θέση του αντιγράφου.

Αν το αντικείμενο έχει ήδη αντιγραφεί η συνάρτηση επιστρέφει τη θέση του νέου αντικειμένου. Αν το αντικείμενο δεν έχει αντιγραφθεί τότε αυτό αντιγράφεται αυτούσιο και η συνάρτηση επιστρέφει τη θέση του νέου αντικειμένου.

Αφού το νέο αντικείμενο είναι ακριβές αντίγραφο του αρχικού, όλοι οι δείκτες του θα δείχνουν στο μη ενεργό κομμάτι της μνήμης

Μετασυνθήκες

```
@ ensures  sanity;
```

Διασφαλίζει ότι η μνήμη μετά τη κλήση είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, ότι η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ ensures  hp >= \old(hp);
```

Ο δρομέας είτε έμεινε ακίνητος, αν δεν έγιναν αντιγραφές, είτε προχώρησε.

```
@ ensures  old_obj(p);
@ ensures  good_obj(hpmin, hp);
```

Τα αντικείμενα, που βρίσκονται στο ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ ensures  new_obj(\result);
```

Το αποτέλεσμα της συνάρτησης είναι δείκτης που δείχνει στην ενεργή περιοχή της μνήμης

```
@ ensures  heap[p-OBJ_HEADER_SIZE] == FORWARDED;
```

Το αντικείμενο έχει σημανθεί ότι έχει αντιγραφεί

```
@ ensures  heap[p-OBJ_HEADER_SIZE+1] == \result;
```

Το παλιό αντικείμενο δείχνει στο νέο

```
@ ensures  all_point_to_old(\old(hp), hp);
```

Όλα τα αντικείμενα, τα οποία βρίσκονται από την θέση του δρομέα, που είχε πριν την κλήση της συνάρτησης, μέχρι και τον δρομέα τώρα, αν έχουν δείκτες, μετά την κλήση της συνάρτησης, αυτοί θα δείχνουν σε αντικείμενα που είναι στο μη ενεργό κομμάτι της μνήμης. Δηλαδή η συνάρτηση δεν πείραξε αυτούς τους δείκτες

```
@ ensures  same_reachable{Pre, Here};
```

Διασφαλίζει ότι όποιο αντικείμενο ήταν προσβάσιμο πριν θα είναι και τώρα. Επίσης διασφαλίζει ότι όλα τα αντικείμενα έχουν την ίδια δομή με αυτήν που είχαν πριν τη συλλογή.

```
@ ensures  all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα είναι προσβάσιμα από τη ρίζα

```
@ ensures  size_of_old(oldmin, oldhp, oldleft);
```

Το μέγεθος των αντικειμένων που δεν έχουν αντιγραφθεί και είναι στην ανενεργή μνήμη είναι oldleft

```
@ ensures  hp - hpmin + oldleft == oldhp - oldmin;
```

Διασφαλίζει ότι ο ελεύθερος χώρος της μνήμης είναι ίσος ή μεγαλύτερος μετά την κλήση, από αυτόν που ήταν πριν.

```
@ ensures  all_not_forwarded(hpmin, hp);
```

Διασφαλίζει ότι στο κομμάτι της μνήμης από hpmin έως hp τα αντικείμενα είναι νέα αντικείμενα.

```
@ ensures \at(heap[p-OBJ_HEADER_SIZE], Pre) != FORWARDED
@         ==> point_to_old(\result);
```

Προσυνθήκες

Για να αποδειχτεί ότι θα ισχύουν όλα τα παραπάνω μετά τη κλήση της συνάρτησης forward, υπάρχει η απαίτηση να ισχύουν τα παρακάτω πριν την κλήση της συνάρτησης αυτής. Αυτά είναι οι προσυνθήκες για την συνάρτηση forwardκαι είναι ορισμένες ως εξής:

```
/*@ requires sanity;
```

η μνήμη πριν την κλήση είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ requires sane_heap(oldmin, oldhp, oldmax);
@ requires separated;
```

Η ενεργή και η μη ενεργή μνήμη δεν αλληλεπικαλύπτονται

```
@ requires good_obj(hpmin, hp);
```

Τα αντικείμενα, που βρίσκονται στο μη ενεργό κομμάτι της μνήμης, είναι διαδοχικά αποθηκευμένα και δεν έχουν κενά μεταξύ τους

```
@ requires all_reachable(root, hpmin, hp);
```

Όλα τα αντικείμενα στην ενεργή μνήμη είναι προσβάσιμα

```
@ requires size_of_old(oldmin, oldhp, oldleft);
```

Το συνολικό μέγεθος των αντικειμένων στο μη ενεργό κομμάτι της μνήμης, που δεν έχουν αντιγραφθεί, είναι oldleft.

```
@ requires hp - hpmin + oldleft == oldhp - oldmin;
```

Τα νέα αντικείμενα στην ενεργή μνήμη, αυτά που δεν έχουν αντιγραφεί ακόμα και βρίσκονται στη παλιά μνήμη, και τα σκουπίδια που βρίσκονται στη παλιά μνήμη έχουν συνολικό μέγεθος όσο η μνήμη που χρησιμοποιούταν πριν τη συλλογή.

```
@ requires old_obj(p);
```

Το αντικείμενο που θα αντιγραφεί βρίσκεται στο ανενεργό κομμάτι της μνήμης

```
@ requires all_not_forwarded(hpmin, hp);
```

Όλα τα αντικείμενα που είναι στο ενεργό κομμάτι της μνήμης είναι νέα αντικείμενα.

```
@ requires     heap[p-OBJ_HEADER_SIZE] != FORWARDED
@          ==> point_to_old(p);
```

Αν το αντικείμενο δε έχει ήδη αντιγραφθεί τότε αν έχει δείκτες, αυτοί δείχνουν στο ανενεργό κομμάτι της μνήμης

Κώδικας της συνάρτησης

```
word forward (word p)
{
  if (heap[p-OBJ_HEADER_SIZE] == FORWARDED)
    return heap[p-OBJ_HEADER_SIZE+1];
  else {
    word size = heap[p-OBJ_HEADER_SIZE];
    word i;

    //@ ghost before:
    //@ ghost word oid = heap[p-OBJ_HEADER_SIZE+1];
    //@ assert size != FORWARDED;

    //@ assert oldleft >= OBJ_HEADER_SIZE + size;
    //@ assert hp + OBJ_HEADER_SIZE + size <= hpmax;

    /*@ loop invariant 0 <= i <= OBJ_HEADER_SIZE + size;
      @ loop invariant sanity;
      @ loop invariant
      @   (\forall integer j;
      @         \at(oldmin, before) <= j < \at(oldhp, before)
      @     ==> \at(heap[j], before) == \at(heap[j], Here));
      @ loop invariant
      @   (\forall integer j;
      @         \at(hpmin, before) <= j < \at(hp, before)
      @     ==> \at(heap[j], before) == \at(heap[j], Here));
      @ loop invariant i > 0 ==> heap[hp] == size;
      @ loop invariant i > 1 ==> heap[hp+1] == oid;
      @ loop invariant
      @   (\forall integer j;
      @         0 <= j < i
      @     ==> heap[p+j] == heap[hp+OBJ_HEADER_SIZE+j]);
      @ loop variant OBJ_HEADER_SIZE + size - i;
      @*/
    for (i=0; i < OBJ_HEADER_SIZE + size; i++)
      heap[hp + i] = heap[p - OBJ_HEADER_SIZE + i];

    //@ ghost word q = hp + OBJ_HEADER_SIZE;
    //@ ghost oldleft -= OBJ_HEADER_SIZE + size;
    heap[p-OBJ_HEADER_SIZE] = FORWARDED;
    heap[p-OBJ_HEADER_SIZE+1] = hp + OBJ_HEADER_SIZE;
    hp += OBJ_HEADER_SIZE + size;

    //@ assert separated;

    return heap[p-OBJ_HEADER_SIZE+1];
  }
}
```

Για τον βρόχο for δίνονται οι παρακάτω αναλλοίωτες

```
/*@ loop invariant 0 <= i <= OBJ_HEADER_SIZE + size;
  @ loop invariant sanity;
```

η μνήμη είναι έγκυρη, δηλαδή ο δρομέας της μνήμης έχει τιμές ανάμεσα των hpmin και hpmax. Επίσης, η αρχή του ενεργού κομματιού της μνήμης είναι ή το 0 ή το μέσο της μνήμης και ότι το μέγεθος του να είναι το μισό μέγεθος της συνολικής μνήμης

```
@ loop invariant
@   (\forall integer j;
@         \at(oldmin, before) <= j < \at(oldhp, before)
@     ==> \at(heap[j], before) == \at(heap[j], Here));
@ loop invariant
@   (\forall integer j;
@         \at(hpmin, before) <= j < \at(hp, before)
@     ==> \at(heap[j], before) == \at(heap[j], Here));
```

Κομμάτια της μνήμης που δεν επηρεάζονται από το βρόχο

```
@ loop invariant i > 0 ==> heap[hp] == size;
```

Έχει αντιγραφεί το μέγεθος του αντικειμένου

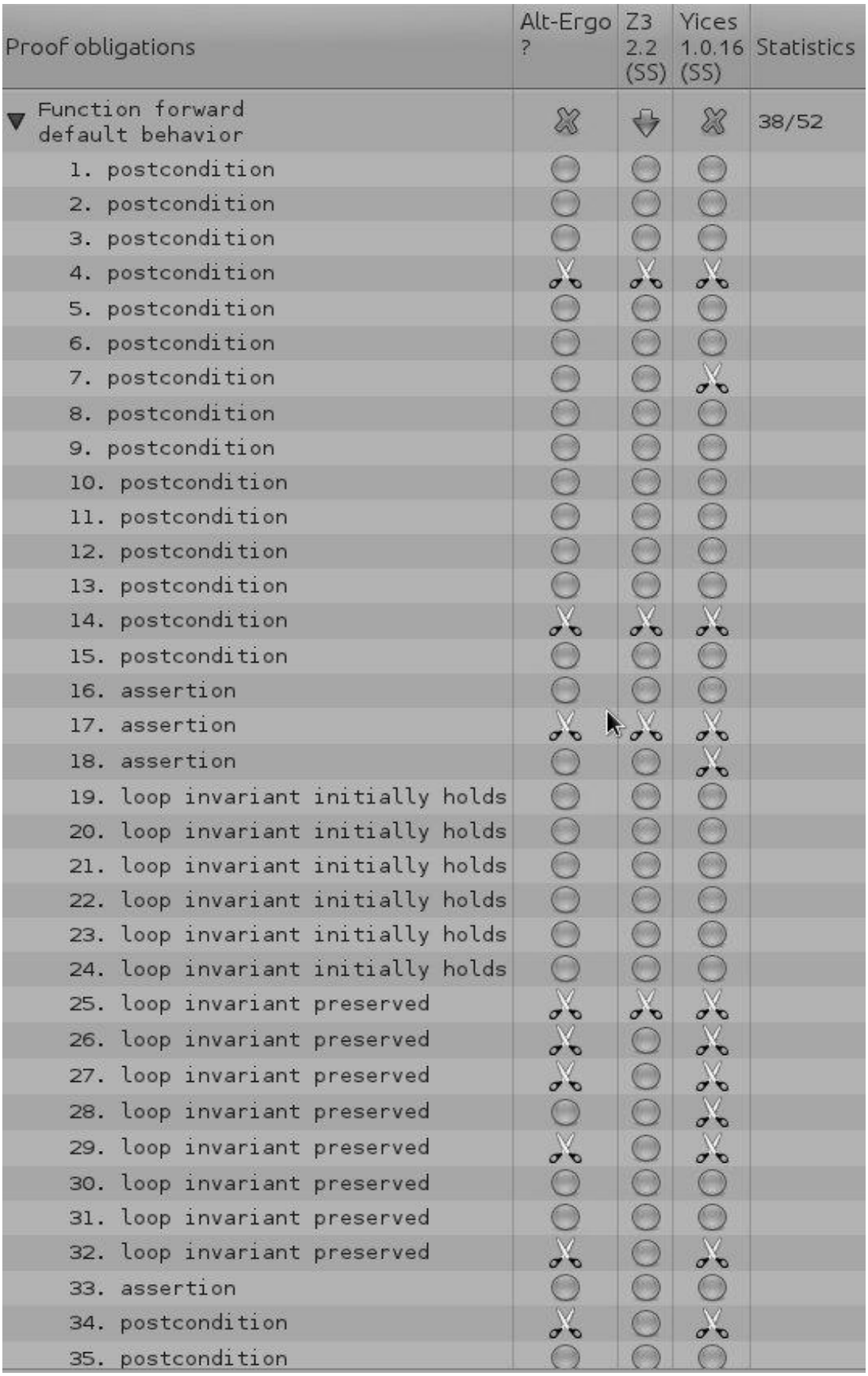
```
@ loop invariant i > 1 ==> heap[hp+1] == oid;
```

Έχει αντιγραφεί η ταυτότητα του αντικειμένου

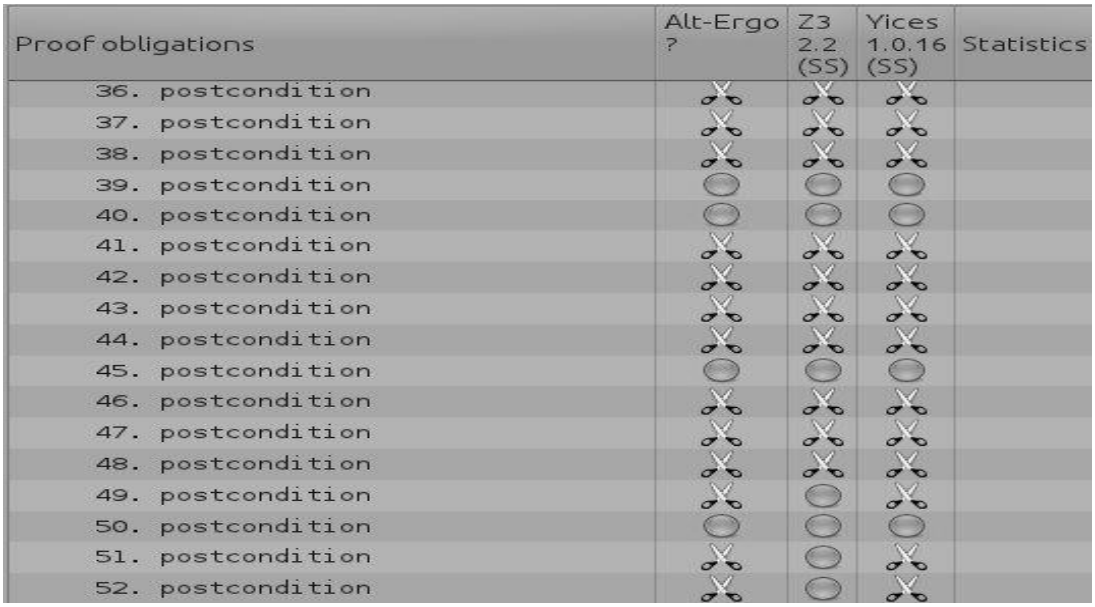
```
@ loop invariant
@   (\forall integer j;
@         0 <= j < i
@     ==> heap[p+j] == heap[hp+OBJ_HEADER_SIZE+j]);
@ loop variant OBJ_HEADER_SIZE + size - i;
@*/
```

Έχουν αντιγραφεί τα δεδομένα του αντικειμένου

Η εκτέλεση της Frama-c έδωσε την παρακάτω εικόνα

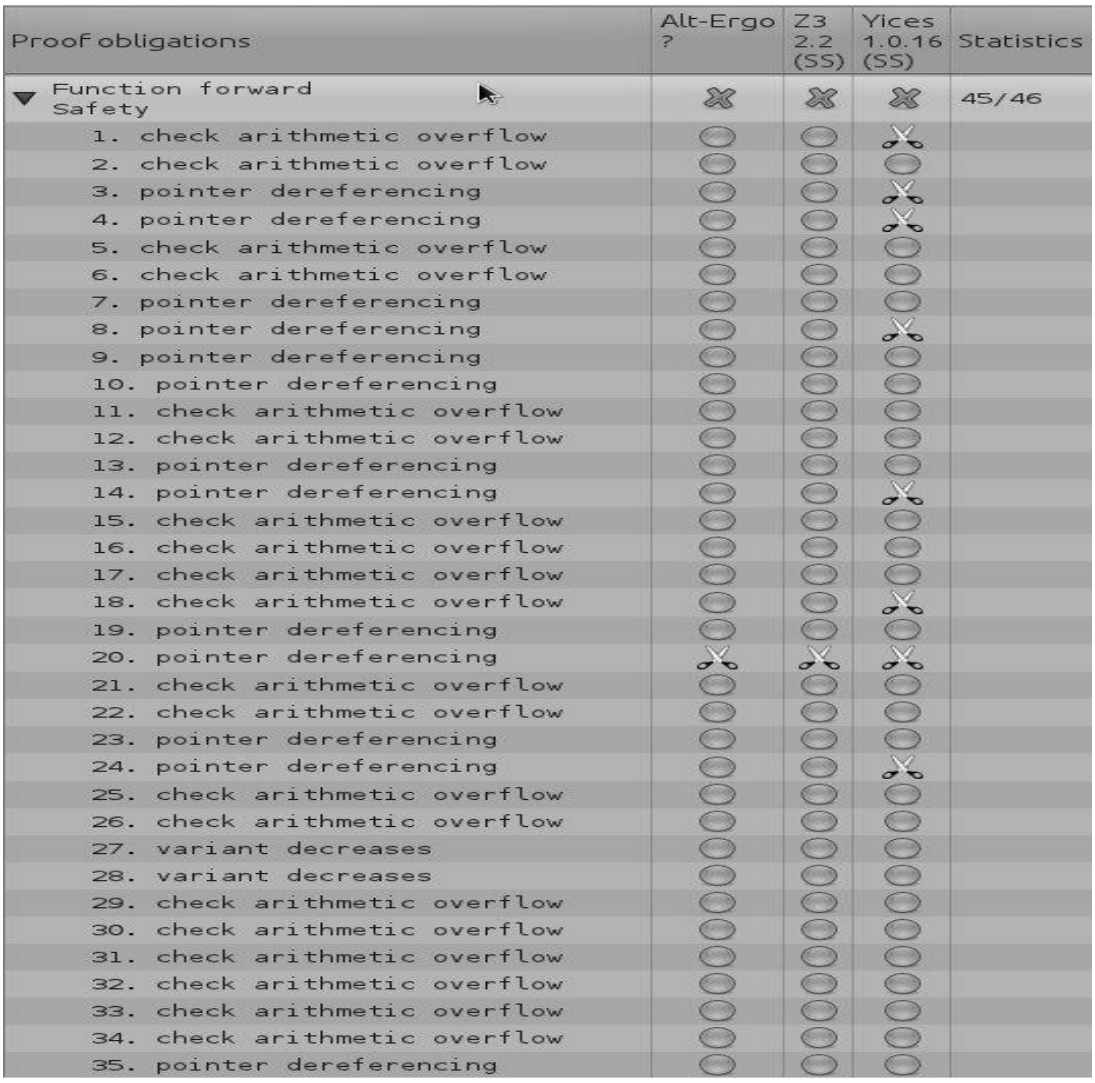


Εικόνα 15- Απόδειξη συνάρτησης forward



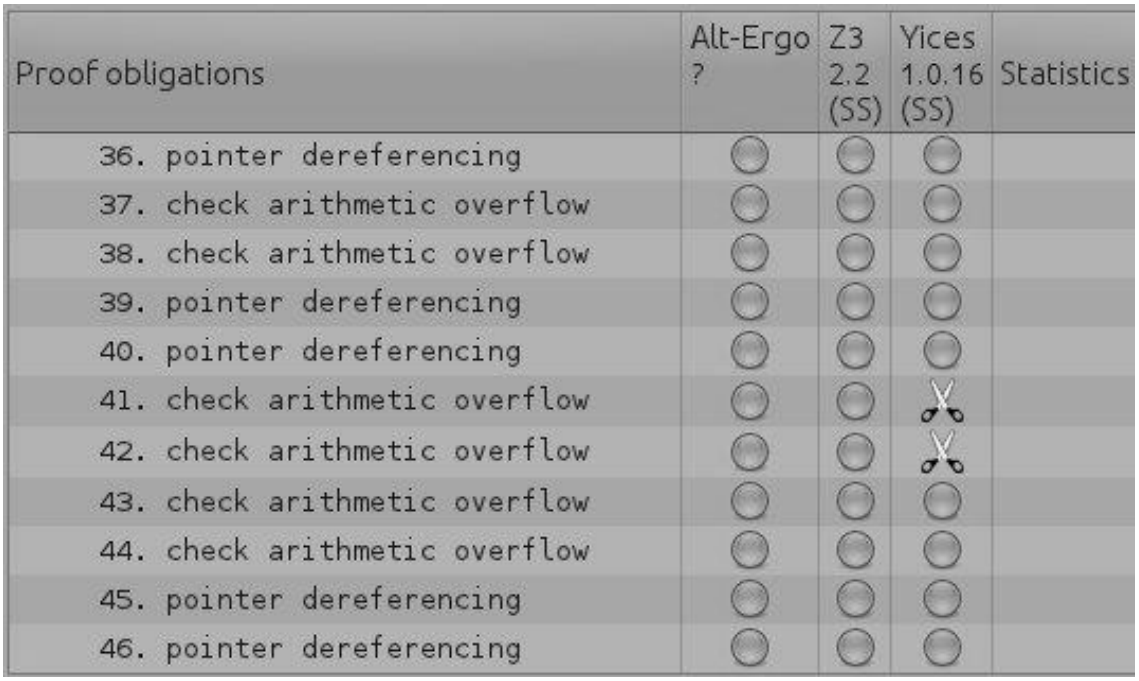
Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
36. postcondition	✂	✂	✂	
37. postcondition	✂	✂	✂	
38. postcondition	✂	✂	✂	
39. postcondition	○	○	○	
40. postcondition	○	○	○	
41. postcondition	✂	✂	✂	
42. postcondition	✂	✂	✂	
43. postcondition	✂	✂	✂	
44. postcondition	✂	✂	✂	
45. postcondition	○	○	○	
46. postcondition	✂	✂	✂	
47. postcondition	✂	✂	✂	
48. postcondition	✂	✂	✂	
49. postcondition	✂	○	✂	
50. postcondition	○	○	○	
51. postcondition	✂	○	✂	
52. postcondition	✂	○	✂	

Εικόνα 16- Απόδειξη συνάρτησης forward (συνέχεια)



Proof obligations	Alt-Ergo ?	Z3 2.2 (SS)	Yices 1.0.16 (SS)	Statistics
▼ Function forward Safety	✘	✘	✘	45/46
1. check arithmetic overflow	○	○	✂	
2. check arithmetic overflow	○	○	○	
3. pointer dereferencing	○	○	✂	
4. pointer dereferencing	○	○	✂	
5. check arithmetic overflow	○	○	○	
6. check arithmetic overflow	○	○	○	
7. pointer dereferencing	○	○	○	
8. pointer dereferencing	○	○	✂	
9. pointer dereferencing	○	○	○	
10. pointer dereferencing	○	○	○	
11. check arithmetic overflow	○	○	○	
12. check arithmetic overflow	○	○	○	
13. pointer dereferencing	○	○	○	
14. pointer dereferencing	○	○	✂	
15. check arithmetic overflow	○	○	○	
16. check arithmetic overflow	○	○	○	
17. check arithmetic overflow	○	○	○	
18. check arithmetic overflow	○	○	✂	
19. pointer dereferencing	○	○	○	
20. pointer dereferencing	✂	✂	✂	
21. check arithmetic overflow	○	○	○	
22. check arithmetic overflow	○	○	○	
23. pointer dereferencing	○	○	○	
24. pointer dereferencing	○	○	✂	
25. check arithmetic overflow	○	○	○	
26. check arithmetic overflow	○	○	○	
27. variant decreases	○	○	○	
28. variant decreases	○	○	○	
29. check arithmetic overflow	○	○	○	
30. check arithmetic overflow	○	○	○	
31. check arithmetic overflow	○	○	○	
32. check arithmetic overflow	○	○	○	
33. check arithmetic overflow	○	○	○	
34. check arithmetic overflow	○	○	○	
35. pointer dereferencing	○	○	○	

Εικόνα 17- Απόδειξη συνάρτησης forward (ασφάλεια)



Εικόνα 18- Απόδειξη συνάρτησης forward (ασφάλειας συνέχεια)

Παρήχθησαν 52 obligation από τα οποία αποδείχθηκαν τα 38 ενώ για τν ασφάλεια παρήχθησαν 46 από τα οποία αποδείχθηκαν τα 45

Συμπεράσματα

Περιορισμοί από τη γλώσσα ACSL

Στην εργασία αυτή παρουσιάστηκε η διαδικασία απόδειξης ενός συλλέκτη σκουπιδιών αντιγραφής. Στη διαδικασία, έγινε χρήση της πλατφόρμας Frama-C και της επέκτασης Jessie που είναι η υλοποίηση της γλώσσας ACSL. Η υλοποίηση αυτή δεν είναι πλήρης και έχει πάρα πολλές δυσλειτουργίες. Πολλές φορές τα αποτελέσματα που δίνονται από το εργαλείο αυτό είναι τόσο αντιφατικά που ουσιαστικά είναι αναξιόπιστα.

Η έκδοση του Jessie, που χρησιμοποιήθηκε, ήταν αυτή της έκδοσης 2 του WHY. Την ώρα που γράφονται αυτές οι γραμμές, η τρέχουσα έκδοση του WHY είναι η 3, η οποία λύνει πολλές από αυτές τις δυσλειτουργίες. Η χρήση του εργαλείου αυτού είναι ουσιαστικά σε επίπεδο που δεν διευκολύνει αρκετά το ούτως ή άλλως δύσκολο έργο της απόδειξης ενός προγράμματος.

Πέρα από τα αντιφατικά αποτελέσματα που εξάγονται κατά τη διάρκεια της απόδειξης, παρατηρήθηκε να βγαίνουν λάθος αποτελέσματα που ήταν προφανώς σωστά. Τέτοια αποτελέσματα είτε θα πρέπει να αποδειχθούν με μη αυτόματα εργαλεία, όπως είναι το Coq, είτε θα πρέπει να ξαναδοθούν οι προδιαγραφές με τέτοιον τρόπο, ώστε να μπορέσουν οι αυτόματες μηχανές απόδειξης θεωρημάτων να τα επαληθεύσουν.

Συμπερασματικά, η απόδειξη προγραμμάτων με τη Frama-C είναι μια διαδικασία δύσκολη, που απαιτεί πολύ μεγάλη προσοχή και που το τελικό αποτέλεσμα δεν πρέπει να θεωρείται σωστό, λόγω όλων των δυσλειτουργιών των εργαλείων.

Για την απόδειξη ενός προγράμματος η διαδικασία που θα πρέπει να ακολουθείται είναι η εξής:

- Ορίζονται στόχοι της απόδειξης.
- Δημιουργείται ένα μοντέλο που επιλύει το ίδιο πρόβλημα σε επίπεδο λογικής.
- Γίνεται η αντιπαράθεση του μοντέλου με τον αλγόριθμο του προγράμματος.
- Αποδεικνύεται ο τρόπος που η κάθε συνάρτηση, προάγει τη λύση του λογικού μοντέλου.

Μελλοντικές κατευθύνσεις

Η διαδικασία απόδειξης που παρουσιάστηκε στην εργασία αυτή μπορεί να είναι μια βάση για τον τρόπο απόδειξης μεγαλύτερων και πιο περίπλοκων αλγορίθμων συλλεκτών αλλά και άλλων παρόμοιων προγραμμάτων.

Η απόδειξη προγραμμάτων είναι ένας τομέας της πληροφορικής που θα αναπτυχθεί τα επόμενα χρόνια. Η ανάγκη επιβεβαίωσης της ορθής λειτουργίας των κρισίμων , κατ ελάχιστο, κομματιών κώδικα πάντα υπήρχε. Όσο εργαλεία σαν τη Frama-C και το Krakatoa αναπτύσσονται και τελειοποιούνται, τόσο πιο διαδεδομένη θα γίνεται η επιβεβαίωση ορθότητας κώδικα.

Βιβλιογραφία

[1] . Steven R. Rakitin, *Software Verification and Validation for Practitioners and Managers,* Artech House

[2] . C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*

[3] . Jonathan Aldrich, *Hoare Logic*, lecture presentation.

[4] . Νικόλαος Παπασπύρου και Κωστής Σαγώνας, *Γλώσσες Προγραματισμού II*, σημειώσεις μαθήματος ΕΜΠ

[5] . Βασίλειος Παπαβασιλείου, *Μηχανική Επαλήθευση Προστακτικών Προγραμμάτων*, διπλωματική εργασία ΕΜΠ 2009

[6] . Ευθύμιος Βερβαινιώτης, *Συλλογή Σκουπιδιών σε Συστήματα Πιστοποιημένου Κώδικα*, διπλωματική εργασία ΕΜΠ 2011

[7] . Ιωάννης Ζούλιας, *Τεχνικές Συλλογής Σκουπιδιών στη Διαχείριση Μνήμης*, διπλωματική εργασία ΕΜΠ 2011

[8] . Steve Blackburn, Robin Garner and Daniel Frampton, *MMTk: The Memory Management Toolkit,*

[9] . Wolfgang Schreiner, *The Java Modeling Language (Part 1)*, Johannes Kepler University lecture presentation

[10] . Wolfgang Schreiner, *The Java Modeling Language (Part 2),* Johannes Kepler University lecture presentation

[11] . Claude Marché, *The Krakatoa Verification Tool for JAVA programs,* INRIA

[12] . Virgile Prevosto, *ACSL Mini-Tutorial*, CEA LIST

[13] . Virgile Prevosto, *Specification and Proof of C Programs using ACSL and Frama-C,* CEA LIST

[14] . Jochen Burghardt, Jens Gerlach, Liangliang Gu, Kerstin Hartig, Hans Pohl, Juan Soto and Kim Vollinger, *ACSL by Example*

[15] . Claude Marché, Yannick Moy, Jessie *Plugin Tutorial,* INRIA

[16] . Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy and Virgile Prevosto, *ACSL: ANSI/ISO C Specification Language,* CEA LIST and INRIA

[17] . Chunxiao Lin, Yiyun Chen, and Bei Hua, *Verification of an Incremental Garbage Collector in Hoare-Style Logic*, University of Science and Technology of China

[18] . José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto and Simão Melo de Sousa, *An Introduction to Program Verification*, Springer