



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

Σχεδίαση και Υλοποίηση ενός Στατικού Συστήματος Τύπων για Επικοινωνούσες Διεργασίες

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΧΙΛΛΕΑΣ ΜΠΕΝΕΤΟΠΟΥΛΟΣ

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

Αθήνα, Ιούλιος 2018



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

Σχεδίαση και Υλοποίηση ενός Στατικού Συστήματος Τύπων για Επικοινωνούσες Διεργασίες

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΧΙΛΛΕΑΣ ΜΠΕΝΕΤΟΠΟΥΛΟΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 17η Ιουλίου 2018.

.....
Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

.....
Γεώργιος Στάμου
Αν. Καθηγητής Ε.Μ.Π.

.....
Αριστείδης Παγουρτζής
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

.....
Αχιλλέας Μπενετόπουλος

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

Copyright © Αχιλλέας Μπενετόπουλος, 2018.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Κινούμαστε όλο και περισσότερο προς κατανεμημένα προγραμματιστικά μοντέλα. Τα περισσότερα σύγχρονα συστήματα λογισμικού είναι κατανεμημένα και ταυτόχρονα, που σημαίνει ότι αποτελούνται από διάφορα κομμάτια, το καθένα από τα οποία εκτελεί ένα σύνολο από "εσωτερικές" σε αυτά λειτουργίες συγχρόνως με τα υπόλοιπα, τη στιγμή όμως που όλα τους εξαρτώνται από ένα υποσύνολο των υπολοίπων για να τις εκτελέσουν, λόγω διαφόρων εξαρτήσεων. Αυτό σημαίνει ότι η επικοινωνία μεταξύ των κομματιών αυτών ανάγεται σε ένα κρίσιμο κομμάτι του συστήματος, και η ορθότητα της επικοινωνίας αυτής αποτελεί μια απαραίτητη προϋπόθεση για την ορθότητα του όλου συστήματος.

Παρουσιάζουμε ένα στατικό σύστημα τύπων συνεδρίας για Erlang, το οποίο ελπίζουμε ότι θα εξυπηρετήσει διττό σκοπό:

- πρώτον, θα δείξει ότι ουσιώδης στατικός έλεγχος τύπων συνεδρίας είναι εφαρμόσιμος για μια δυναμική γλώσσα προγραμματισμού, και
- δεύτερον, θα αποτελέσει θεμέλιο για ένα εργαλείο αρκετά ισχυρό και χρήσιμο για να γίνει τελικά κομμάτι της εργαλειοθήκης ενός προγραμματιστή κατά τη φάση ανάπτυξης ενός έργου

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Τύποι Συνεδριών, Συστήματα Τύπων, Συστήματα Επιδράσεων, Ταυτοχρονισμός.

Abstract

We are increasingly moving towards distributed programming models. Most software systems are distributed and concurrent, meaning that they are composed of various components, each of them performing a certain set of self-contained operations simultaneously with all the others, but all of them relying on a subset of the others to progress with their work, due to data dependencies and the like. This means that communication between the components is nothing less than a critical part of the system, and its correctness therefore a prerequisite to the correctness of the system as a whole.

We present a static session type checking tool for Erlang, which we hope will serve a twofold purpose:

- firstly, show that meaningful static session type checking is viable for a dynamic programming language, and
- secondly, become a foundation for a tool that would be useful and powerful enough to eventually become part of a programmer's toolchain during a project's development lifecycle

Key words

Programming languages, Session Types, Type systems, Effect Systems, Concurrency.

Ευχαριστίες

Not a placeholder

Αχιλλέας Μπενετόπουλος,

Αθήνα, 17η Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-42-14, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

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

FTP: <ftp://ftp.softlab.ntua.gr/pub/techrep/>

Περιεχόμενα

Περίληψη	5
Abstract	7
Ευχαριστίες	9
Περιεχόμενα	11
Κατάλογος πινάκων	13
Κατάλογος σχημάτων	15
1. Εισαγωγή	17
1.1 Στόχοι της παρούσας εργασίας	17
1.2 Σύνοψη της εργασίας	18
2. Το σύστημα Erlang/OTP	19
2.1 Η γλώσσα προγραμματισμού Erlang	19
2.2 Ο μεταγλωττιστής της Erlang	19
2.3 Το "σύστημα τύπων" της Erlang	20
2.4 Το εργαλείο Dialyzer	20
2.5 Ταυτοχρονισμός σε Erlang	21
3. Συστήματα τύπων και επιδράσεων	23
3.1 Κλασσικά συστήματα τύπων με συμπερασμό	23
3.2 Συστήματα τύπων με ετικέτες	24
3.3 Συστήματα τύπων και επιδράσεων	24
4. Τύποι Συνεδρίας	25
5. Ένα στατικό σύστημα τύπων συνεδρίας σε Erlang	27
5.1 Το σύστημα τύπων	27
5.2 Υλοποίηση	29
5.2.1 Τμήματα	29
5.2.2 Συμπερασμός Τύπων Συνεδριών	30
5.2.3 Έλεγχος σε σχέση με το specification	31
5.2.4 Έλεγχος peers	32
5.3 Παράδειγμα υλοποίησης: Arithmetic Server	37
6. Συμπεράσματα και μελλοντική έρευνα	45
Βιβλιογραφία	47

Παράρτημα	49
A. Ευρετήριο συμβολισμών	49
B. Ευρετήριο γλωσσών	51
C. Ευρετήριο αριθμών	53

Κατάλογος πινάκων

Κατάλογος σχημάτων

5.1	server/0	35
5.2	simple_client/1	36
5.3	recursive_client/2	37
5.4	server/0	40
5.5	sum_client/2	41
5.6	neg_client/2	42
5.7	sqrt_client/2	43

Κεφάλαιο 1

Εισαγωγή

Τα κατανεμημένα προγραμματιστικά μοντέλα αποκτούν όλο και μεγαλύτερη δημοτικότητα στο σύγχρονο κόσμο της βιομηχανίας παραγωγής λογισμικού. Πολλά σύγχρονα συστήματα που πρέπει να ανταποκρίνονται σε υψηλούς φόρτους εργασίας με ισχυρές απαιτήσεις ως προς την αξιοπιστία και τη διαθεσιμότητά τους είναι κατανεμημένα και ταυτόχρονα, που σημαίνει ότι αποτελούνται από διάφορα κομμάτια, το καθένα από τα οποία εκτελεί ένα σύνολο από "εσωτερικές" σε αυτό λειτουργίες συγχρόνως με τα υπόλοιπα, τη στιγμή όμως που όλα τους εξαρτώνται από ένα υποσύνολο των υπολοίπων για να τους παρέχουν πληροφορίες προκειμένου να φέρουν τη δουλειά τους εις πέρας με επιτυχία. Αυτό σημαίνει ότι η επικοινωνία μεταξύ των κομματιών αυτών ανάγεται σε ένα κρίσιμο κομμάτι του συστήματος, και ότι η ορθότητα της επικοινωνίας αυτής αποτελεί μια απαραίτητη προϋπόθεση για την ορθότητα του όλου συστήματος.

Προκειμένου να καθορίσουμε τα μοτίβα επικοινωνίας που θεωρούνται αποδεκτά στα πλαίσια των εφαρμογών μας, ορίζουμε *πρωτόκολλα* επικοινωνίας, δηλαδή προσπαθούμε να προσδιορίσουμε τους τύπους των μηνυμάτων που ανταλλάσσονται, τότε κατά τη διάρκεια της επικοινωνίας εμφανίζονται αυτά τα μηνύματα, και ανάμεσα σε ποια κομμάτια του συστήματος αυτά ανταλλάσσονται. Συνεπώς, μπορούμε να εκφράσουμε την ορθότητα της επικοινωνίας μεταξύ των επιμέρους συστημάτων ως τη *συμμόρφωση* σε αυτό το πρωτόκολλο από τους συμμετέχοντες στην επικοινωνία. Οι τύποι συνεδριών μας επιτρέπουν να εκφράσουμε αυτά τα πρωτόκολλα επικοινωνίας, καθώς και τις υλοποιήσεις τους από τα άκρα αυτών, με μια μορφή που πηγάζει από τη θεωρία τύπων, και κατ'επέκταση να επαληθεύσουμε την ορθότητα της επικοινωνίας εφαρμόζοντας μεθόδους από τον χώρο της θεωρίας τύπων.

Σε αυτήν την εργασία διερευνούμε την αποτελεσματικότητα ενός στατικού συστήματος τύπων δυαδικών συνεδριών για την Erlang, μια γλώσσα προγραμματισμού για την οποία ο ταυτοχρονισμός αποτελεί κύριο χαρακτηριστικό. Επιχειρούμε να δουλέψουμε με ένα υποσύνολο της γλώσσας το οποίο, αν και περιορισμένο, θεωρούμε ότι είναι ουσιώδες, αποκλείοντας μόνο εξωτικούς εγγενείς τύπους δεδομένων της γλώσσας, καθώς και συναρτήσεις υψηλότερης τάξης.

1.1 Στόχοι της παρούσας εργασίας

Σκοπός της παρούσας εργασίας είναι η μελέτη των συστημάτων τύπων συνεδριών, και η διερεύνηση της αποτελεσματικότητας ενός στατικού συστήματος τύπων δυαδικών συνεδριών για την Erlang, μια συναρτησιακή γλώσσα προγραμματισμού με δυναμικό σύστημα τύπων, η οποία παρέχει ισχυρούς μηχανισμούς ταυτοχρονισμού.

Στη βιβλιογραφία υπάρχει ένας αριθμός από υλοποιήσεις μηχανισμών χρόνου εκτέλεσης για την παρακολούθηση της επικοινωνίας μεταξύ διεργασιών σε γλώσσες με δυναμικό σύστημα τύπων όπως η Erlang [reference to Python's SPY, reference to Fowler's implementation]. Στη βάση τους, όλες χτίζουν πάνω στη γλώσσα Scribble, η οποία είναι μια DSL για την περιγραφή πρωτοκόλλων επικοινωνίας. Στη δική μας δουλειά, επιχειρούμε να κατασκευάσουμε ένα σύστημα στατικής ανάλυσης, το οποίο μπορεί να:

- Συμπεράνει τους τύπους συνεδριών των συναρτήσεων που αποτελούν ένα πρόγραμμα.
- Εξακριβώσει τη συμμόρφωσή τους με το πρωτόκολλο που πρέπει να υλοποιούν.

- Επαληθεύσει ότι δύο διεργασίες αποτελούν πράγματι τα δύο "άκρα" ενός δυαδικού πρωτοκόλλου επικοινωνίας.

1.2 Σύνοψη της εργασίας

Στο κεφάλαιο 2 παρουσιάζουμε τη γλώσσα Erlang, και όλες τις πληροφορίες που είναι απαραίτητες για την κατανόηση του τρόπου λειτουργίας του συστήματός μας. Εξηγούμε τι είναι η Erlang, και για ποιο λόγο ένα σύστημα τύπων συνεδρίας έχει ιδιαίτερη αξία στα πλαίσια αυτής της γλώσσας. Καθώς ο συμπερασμός τύπων συνεδρίας βασίζεται ιδιαίτερα στο σύστημα τύπων της γλώσσας, θα πραγματοποιήσουμε μια παρουσίαση αυτού, καθώς και του Dialyzer, του εργαλείου (μέρος του OTP) το οποίο χρησιμοποιούμε για τον συμπερασμό των τύπων δεδομένων των προγραμμάτων που αναλύουμε.

Στο κεφάλαιο 3 πραγματοποιούμε μια παρουσίαση των συστημάτων τύπων και επιδράσεων (type and effect systems), τα οποία αποτελούν μια επέκταση των κλασσικών συστημάτων τύπων με έναν τρόπο περιγραφής των παρενεργειών που ενδέχεται να έχει ένα κομμάτι κώδικα.

Στο κεφάλαιο 4 παρουσιάζουμε τη θεωρία και την υπάρχουσα δουλειά πίσω από τα συστήματα τύπων συνεδρίας.

Στο κεφάλαιο 5 αναλύουμε το σύστημα που υλοποιήσαμε, και παρουσιάζουμε μερικά παραδείγματα χρήσης του συστήματός μας. Τα παραδείγματα αυτά αποτελούνται τόσο από σχετικά απλουστευμένα προγράμματα, τα οποία είναι τυποποιημένα στη σχετική βιβλιογραφία, αλλά και παραδείγματα προσαρμοσμένα στο να αναδείξουν τα προτερήματα της υλοποίησής μας.

Τέλος, στο κεφάλαιο 7 προσφέρουμε μερικά συμπεράσματα που προέκυψαν από την υλοποίηση του συστήματος αυτού, αλλά και τα πειράματά μας. Επιπλέον, εκθέτουμε μερικές ιδέες για μελλοντική δουλειά πάνω στο συγκεκριμένο αντικείμενο.

Κεφάλαιο 2

Το σύστημα Erlang/OTP

2.1 Η γλώσσα προγραμματισμού Erlang

Η Erlang είναι μια συναρτησιακή γλώσσα προγραμματισμού, η οποία σχεδιάστηκε τη δεκαετία του 1980 στο Εργαστήριο Επιστήμης Υπολογιστών της Ericsson. Αρχικά προοριζόταν για προγραμματισμό συστημάτων και εφαρμογών στον τομέα των τηλεπικοινωνιών, και κατα συνέπεια έπρεπε να ανταποκρίνεται στις απαιτήσεις αυτής της βιομηχανίας για την ανάπτυξη κατανεμημένων συστημάτων υψηλού επιπέδου παραλληλισμού, τα οποία επιδεικνύουν υψηλή ανοχή σε σφάλματα. Έκτοτε η χρήση της έχει διαδοθεί και σε βιομηχανίες που μοιράζονται τις ίδιες λειτουργικές απαιτήσεις, όπως η τραπεζική, οι επικοινωνίες πραγματικού χρόνου (WhatsApp), κα.

Το OTP είναι μια συλλογή εργαλείων, βιβλιοθηκών, middleware και αρχών σχεδιασμού για προγράμματα Erlang. Είναι αναπόσπαστο κομμάτι της διανομής ελεύθερου κώδικα της Erlang. Μερικά από τα περιεχόμενά του είναι:

- Ένα σύστημα χρόνου εκτέλεσης (εικονική μηχανή), ο BEAM
- Ένας μεταγλωττιστής σε κώδικα μηχανής
- Ένα εργαλείο στατικής ανάλυσης, ο Dialyzer, στο οποίο θα αναφερθούμε παρακάτω

2.2 Ο μεταγλωττιστής της Erlang

Μια εφαρμογή σε Erlang δομείται σε modules, τα οποία μεταγλωττίζονται ανεξάρτητα και φορτώνονται στο σύστημα χρόνου εκτέλεσης. Ο μεταγλωττιστής της Erlang μεταγλωττίζει τον πηγαίο κώδικα σε κώδικα εικονικής μηχανής (bytecode), ο οποίος εκτελείται από την εικονική μηχανή BEAM.

Ο μεταγλωττιστής δε μεταφράζει απευθείας τον πηγαίο κώδικα των προγραμμάτων σε κώδικα μηχανής, αλλά πρώτα σε μια ενδιάμεση αναπαράσταση, που ονομάζεται Core Erlang. Αυτή είναι μια συμπαγής, υψηλού επιπέδου αναπαράσταση του προγράμματος, με μια αρκετά απλουστευμένη γραμματική, και ξεκάθαρους σημασιολογικούς κανόνες.

Το κύριο κίνητρο πίσω από το σχεδιασμό της Core Erlang ήταν το να αναπτυχθεί μια αναπαράσταση προγραμμάτων Erlang που να καθιστά εύκολη την ανάπτυξη εργαλείων που πρέπει να διατρέξουν και να επεξεργαστούν προγράμματα Erlang, όπως αποσφαλματωτές, εργαλεία στατικής ανάλυσης κτλ. Κάτι τέτοιο δεν ήταν εφικτό απευθείας σε πηγαίο κώδικα Erlang, λόγω της συντακτικής πολυπλοκότητάς της, και οι προσπάθειες απλούστευσης της γλώσσας οδηγήθηκαν σε αδιέξοδο.

Το OTP παρέχει τη δυνατότητα σε οποιοδήποτε πρόγραμμα Erlang να διαβάσει την Core Erlang αναπαράσταση ενός οποιουδήποτε Erlang module. Σε αυτό το χαρακτηριστικό βασίζεται το σύστημά μας για την ανάλυση των προγραμμάτων που του δίνονται ως είσοδος. Φορτώνει τα modules στο σύστημα χρόνου εκτέλεσης, και αποσπά το Core Erlang Αφηρημένο Δένδρο Σύνταξης αυτού σε μορφή ενός συνόλου από εγγραφές (records, εγγενής τύπος δεδομένων της Erlang), όπως αυτά ορίζονται στο `otp/lib/compiler/src/core_parse.hrl`.

2.3 Το "σύστημα τύπων" της Erlang

Η Erlang είναι μια γλώσσα με δυναμικό σύστημα τύπων. Αυτό σημαίνει ότι ο μεταγλωττιστής της γλώσσας δεν πραγματοποιεί έλεγχο τύπων κατά τη μεταγλώττιση ενός προγράμματος. Το σύστημα χρόνου εκτέλεσης είναι υπεύθυνο για να εγγυηθεί την ασφάλεια τύπων, με το να ελέγχει την πληροφορία τύπων που είναι συσχετισμένη με κάθε κομμάτι δεδομένων ενός προγράμματος κατά τη διάρκεια της εκτέλεσης.

Ένα κομμάτι δεδομένων σε Erlang ονομάζεται όρος (*term*). Ο τύπος ενός όρου μπορεί να είναι είτε κάποιος από τους ενσωματωμένους τύπους της Erlang, είτε ένας τύπος που έχει προσδιορίσει ο χρήστης. Οι ενσωματωμένοι τύποι της Erlang είναι:

- Pid
- Port
- Reference
- Atom
- Bitstring
- Float
- Fun
- Integer
- List
- Map
- Tuple
- Union

Οποιοσδήποτε όρος της Erlang μπορεί να αποτελέσει το περιεχόμενο των μηνυμάτων που ανταλλάσσουν διεργασίες μεταξύ τους.

2.4 Το εργαλείο Dialyzer

Οι κόμβοι του Core Erlang Δένδρου Αφηρημένης Σύνταξης δε διαθέτουν πληροφορίες τύπων δεδομένων. Σε ορισμένες "απλές" περιπτώσεις, όπως όταν έχουμε κυριολεκτικούς όρους στον κώδικα που αναλύουμε (για παράδειγμα $x = 42$), μπορούμε να βρούμε τον τύπο των όρων χρησιμοποιώντας συναρτήσεις βιβλιοθήκης της γλώσσας. Προκειμένου να πραγματοποιήσουμε οποιαδήποτε ουσιώδη ανάλυση σχετικά με τους τύπους συνεδριών σε ένα πρόγραμμα, χρειάζεται να γνωρίζουμε τους τύπους των δεδομένων που περιέχονται στα μηνύματα που ανταλλάσσονται μεταξύ διεργασιών. Συνεπώς, χρειαζόμαστε ένα εργαλείο το οποίο, δοθέντος ενός προγράμματος σε Erlang, να πραγματοποιεί συμπερασμό τύπων δεδομένων.

Ο Dialyzer (DIscrepancy AnaLYZer) είναι ένα σύστημα που πραγματοποιεί μιας μορφής στατικό type checking σε προγράμματα Erlang. Σε πρώτη φάση επεξεργάζεται ένα πρόγραμμα Erlang, προκειμένου να εξάγει τα λεγόμενα success typings. Τα success typings είναι οι τύποι που πρέπει να έχουν οι συναρτήσεις (δηλαδή ο συνδυασμός των τύπων των ορισμάτων μιας συνάρτησης, και του τύπου του αποτελέσματος που αυτή η συνάρτηση επιστρέφει) προκειμένου να εξασφαλιστεί ότι η συνάρτηση δε θα παρουσιάσει κάποιο σφάλμα χρόνου εκτέλεσης που οφείλεται σε κάποια ασυνέπεια στους τύπους. Για παράδειγμα, στο κομμάτι κώδικα του σχήματος 2.1 το να καλέσουμε τη συνάρτηση `foo/1` ως `foo("Hello")` θα οδηγούσε σε σφάλμα χρόνου εκτέλεσης, καθώς ο τελεστής `+` δε μπορεί να

εφαρμοστεί σε ένα όρισμα τύπου *List* σε Erlang. Αυτό οδηγεί το Dialyzer στο να συμπεραίνει τύπους με έναν συντηρητικό τρόπο: οι τύποι που κάνει assign σε συναρτήσεις, και οι οποίοι απορρέουν από τους τύπους που αποδίδει στα επιμέρους τμήματα δεδομένων στο σώμα των συναρτήσεων, αποτελούν ένα υπερσύνολο των πραγματικών τύπων των δεδομένων μιας συνάρτησης. Στο ίδιο κομμάτι κώδικα, ο τύπος που θα συμπεραίνει ο Dialyzer για τη συνάρτηση *foo/1* είναι *Number* \rightarrow *Number*.

Listing 2.1: Simple receive

```
1  foo (X) ->
2      X + 42.
```

Listing 2.1: Simple receive

Προκειμένου ο Dialyzer να φτάσει στα success typings, πραγματοποιεί έναν συμπερασμό τύπων για όλα τα επιμέρους terms που εμφανίζονται μέσα στα σώματα των συναρτήσεων. Η έκδοση του Dialyzer που δημοσιεύεται ως μέρος του OTP δεν αποθηκεύει αυτήν την πληροφορία για όλη τη διάρκεια της ανάλυσης, αλλά μόνο για όσο χρειάζεται για την εκάστοτε συνάρτηση. Για τους σκοπούς της δικής μας ανάλυσης, χρειάζεται να γνωρίζουμε τους τύπους δεδομένων των Erlang terms σε κάθε γραμμή πηγαιού κώδικα ενός Erlang module. Προκειμένου να εξασφαλίσουμε αυτήν την πληροφορία, προβήκαμε στην τροποποίηση του Dialyzer με σκοπό την αποστολή της πληροφορίας τύπων δεδομένων στο σύστημά μας κατά τη διάρκεια της εκτέλεσης του dataflow analysis loop του.

2.5 Ταυτοχρονισμός σε Erlang

Ένα ισχυρό χαρακτηριστικό της Erlang είναι η υποστήριξη της για ταυτοχρονισμό. Παρέχει στον προγραμματιστή ένα σύνολο από primitives για τη δημιουργία διεργασιών, και τη διαχείριση της επικοινωνίας μεταξύ αυτών. Αυτές δεν είναι διεργασίες σε επίπεδο λειτουργικού συστήματος, ούτε όμως και νήματα, αλλά αποτελούν οντότητες τις οποίες διαχειρίζεται η εικονική μηχανή εκτέλεσης της Erlang, ο BEAM.

Οι διεργασίες σε Erlang δε μοιράζονται αποθηκευτικό χώρο. Ο μοναδικός μηχανισμός που παρέχεται στον προγραμματιστή για την υλοποίηση της επικοινωνίας και της ανταλλαγής πληροφοριών μεταξύ αυτών των διεργασιών είναι η ασύγχρονη ανταλλαγή μηνυμάτων. Κάθε διεργασία διαθέτει ένα "κουτί εισερχομένων", μια ουρά από μηνύματα που έχει λάβει από άλλες διεργασίες. Για να καταναλώσει ένα μήνυμα από αυτήν την ουρά, ο προγραμματιστής χρησιμοποιεί την έκφραση *receive*, με την οποία παίρνει ένα μήνυμα από την κεφαλή της ουράς, και το συγκρίνει με μια σειρά από μοτίβα (patterns) τα οποία αντιστοιχούν στα μηνύματα που περιμένει ότι το πρόγραμμά του θα λάβει στη συγκεκριμένη φάση της εκτέλεσης. Όταν μια σύγκριση πετύχει, η διεργασία συνεχίζει την εκτέλεσή της.

Listing 2.2: Simple receive

```
1  foo () ->
2      receive
3          {num, Sender, N} -> io:format("Got a number ~p from ~p\n", [N, Sender]);
4          {atom, Sender, A} -> io:format("Got an atom ~p from ~p\n", [A, Sender]);
5          42 -> io:format("Got 42\n")
6      end.
```

Listing 2.2: Simple receive

Στο listing 2.2 φαίνεται ένα παράδειγμα μιας συνάρτησης που περιμένει να διαβάσει από το κουτί εισερχομένων της:

- Μια πλειάδα τιμών (tuple) που αποτελείται από ένα atom, ένα pid κι έναν αριθμό
- Μια πλειάδα τιμών που αποτελείται από ένα atom, ένα pid κι ένα atom

- Έναν απλό αριθμό

Αν μια διεργασία που επικοινωνεί με αυτήν της αποστέλλει ένα μήνυμα της μορφής $\{num, 42\}$, τότε η $foo/1$ δε θα το "καταναλώσει" ποτέ. Αντίστοιχα, αν καμία διεργασία δεν αποστέλλει στη διεργασία που εκτελεί τον κώδικα του listing 2.2 κάποιο μήνυμα που να αντιστοιχεί σε ένα από τα μοτίβα της εντολής *receive*, η εκτέλεση θα "κρεμάσει" επ' αόριστον. Σε αυτήν την περίπτωση θα είχαμε ένα προβληματικό σενάριο, που προκύπτει από τη μη-συμμόρφωση ενός εκ των δύο μερών της επικοινωνίας στο αναμενόμενο πρωτόκολλο.

Κεφάλαιο 3

Συστήματα τύπων και επιδράσεων

Πολλές γλώσσες προγραμματισμού υλοποιούν ένα στατικό σύστημα τύπων, προκειμένου να εγγυηθούν ότι τελεστές εφαρμόζονται μόνο σε τελεσταίους της κατάλληλης μορφής, δηλαδή του κατάλληλου είδους. Για παράδειγμα, ένα κλασσικό σύστημα τύπων θα απέρριπτε το πρόγραμμα 3.1 (γραμμένο σε μια υποθετική γλώσσα), καθώς η πράξη της αφαίρεσης ($-$), δεν έχει νόημα όταν εφαρμόζεται σε μια συμβολοσειρά και έναν αριθμό (υποθέτουμε ότι η γλώσσα δεν παρέχει τη δυνατότητα υπερφόρτωσης τελεστών).

Listing 3.1: Ασύμβατοι Τελεσταίοι

```
1 begin
2   print('Hello , world!' - 42)
3 end.
```

Listing 3.1: Ασύμβατοι Τελεσταίοι

3.1 Κλασσικά συστήματα τύπων με συμπερασμό

Τα πρώτα στατικά συστήματα τύπων βασίζονταν σε type annotations τα οποία παρείχε ο χρήστης. Αυτά ήταν ετικέτες που συνόδευαν τις δηλώσεις μεταβλητών και συναρτήσεων, και περιέγραφαν τον τύπο που αυτές θα περιείχαν (στην περίπτωση μεταβλητών), ή τους τύπους των ορισμάτων και της τελικής τιμής τους (στην περίπτωση συναρτήσεων), και υπηρετούσαν ως οδηγίες προς τον μεταγλωττιστή. Κατά το χρόνο μεταγλώττισης, αυτός ήταν υπεύθυνος να συγκρίνει τον δηλωθέντα τύπο μιας μεταβλητής ή συνάρτησης με τους προσδοκώμενους τύπους στο σημείο που αυτά τα constructs χρησιμοποιούνται σε ένα πρόγραμμα, προκειμένου να διαπιστώσει αν υπάρχουν ασυμφωνίες τύπων.

Προκειμένου να εξαλειφθεί η ανάγκη για τη χειρονακτική δήλωση των τύπων τμημάτων δεδομένων σε ένα πρόγραμμα, αναπτύχθηκαν συστήματα τύπων με αυτόματο συμπερασμό τύπων, όπως αυτό των Hindley-Milner (ή αλλιώς Damas-Milner), με την γλώσσα ML να είναι η πρώτη που υλοποίησε ένα τέτοιο σύστημα.

Η προσέγγιση των συστημάτων τύπων είναι να συσχετίζουν τύπους με προγράμματα. Οι τύποι περιγράφουν το είδος των δεδομένων τα οποία μεταχειρίζεται το πρόγραμμα. Ο συσχετισμός αυτός γίνεται μέσω ενός συνόλου κανόνων συμπερασμού, με βάση τη σύνταξη του προγράμματος, κάνοντας χρήση ενός περιβάλλοντος τύπων, το οποίο αντιστοιχεί τις ελεύθερες μεταβλητές ενός προγράμματος με τους τύπους τους. Η αντιστοίχιση αυτή εκφράζεται με μια σχέση του τύπου $\Gamma \mid e : \tau$, όπου:

- Γ είναι το περιβάλλον τύπων
- e είναι το πρόγραμμα υπό εξέταση
- τ είναι ο τύπος του προγράμματος e , δεδομένων των "περιορισμών" που τίθενται σχετικά με τις ελεύθερες μεταβλητές που εμφανίζονται στο σώμα του e από το περιβάλλον Γ

Οι προκλήσεις του σχεδιασμού ενός συστήματος συμπερασμού τύπων είναι:

- Το σύστημα τύπων να είναι σημασιολογικά ορθό
- Το σύστημα τύπων να είναι αποφάνσιμο, το οποίο σημαίνει ότι πρέπει να είναι δυνατός ο σχεδιασμός ενός αλγορίθμου ο οποίος να τερματίζει και να ελέγχει τις ιδιότητες του συστήματος

3.2 Συστήματα τύπων με ετικέτες

Μια προσέγγιση για την επέκταση της εκφραστικότητας ενός συστήματος τύπων είναι η προσθήκη ετικετών στους τύπους με αποτέλεσμα την παραμετροποίησή τους. Στόχος αυτής της επέκτασης είναι το να καταστεί εφικτό ένα σύνολο στατικών αναλύσεων, όπως:

- Ανάλυση χρόνου δεσίματος, κατά την οποία προσπαθούμε να πραγματοποιήσουμε μια διάκριση ανάμεσα σε αυτά δεδομένα των οποίων η τιμή είναι γνωστή κατά το χρόνο μεταγλώττισης (στατικά), και σε αυτά τα οποία αποκτούν τιμή κατά την εκτέλεση του προγράμματος (δυναμικά). Αυτού του τύπου η ανάλυση αποτελεί τη βάση της μερικής αποτίμησης που πραγματοποιούν μεταγλωττιστές
- Ανάλυση χρήσεων, κατά την οποία θέλουμε να μάθουμε πόσες φορές χρησιμοποιείται η τιμή μιας έκφρασης στη διάρκεια της εκτέλεσης ενός προγράμματος. Αν αποφανθούμε ότι δε γίνεται ποτέ χρήση μιας τιμής, ο μεταγλωττιστής δε χρειάζεται να παράξει κώδικα για αυτήν. Επίσης, αν μια τιμή χρησιμοποιείται μόνο μια φορά, η μεταβλητή που την περιέχει μπορεί να συλλεγεί από τον συλλέκτη σκουπιδιών του περιβάλλοντος εκτέλεσης αμέσως μετά τη χρήση της.
- Ανάλυση ροής ελέγχου, κατά την οποία προσπαθούμε, συνήθως σε γλώσσες με συναρτήσης υψηλότερης τάξης, να συμπεράνουμε ποιες συναρτήσεις είναι πιθανοί "αποδέκτες" μιας κλήσης σε ένα πρόγραμμα, δηλαδή ποιες συναρτήσεις είναι πιθανό να αποκτήσουν τον έλεγχο ενός προγράμματος (το οποίο δεν είναι πάντα εμφανές σε γλώσσες τέτοιου τύπου από τον πηγαίο κώδικά τους).

3.3 Συστήματα τύπων και επιδράσεων

Ο στόχος της προσθήκης πληροφορίας επιδράσεων στα κλασσικά συστήματα τύπων είναι η επέκτασή τους ώστε να εκφράζουν επιπλέον ιδιότητες που προκύπτουν από τους σημασιολογικούς κανόνες ενός προγράμματος.

Τα συστήματα τύπων και επιδράσεων αποτελούν ουσιαστικά μια παραλλαγή των συστημάτων τύπων με ετικέτες στα οποία αναφερθήκαμε παραπάνω, αλλά με μια διαφορά: οι ετικέτες σε αυτά τα συστήματα τύπων περιέχουν πληροφορία σχετικά με την συμπεριφορά του προγράμματος κατά τον χρόνο αποτίμησης και εκτέλεσής του.

Σε ένα σύστημα τύπων και επιδράσεων, η αντιστοίχιση ενός προγράμματος με έναν τύπο και μια επίδραση γράφεται ως $\Gamma \vdash e : \tau; \phi$, που σημαίνει ότι στο περιβάλλον Γ , το πρόγραμμα e εμφανίζει μια επίδραση ϕ , και εν τέλει επιστρέφει μια τιμή τύπου τ .

Κεφάλαιο 4

Τύποι Συνεδρίας

Στα δίκτυα, μια συνεδρία είναι μια λογική οντότητα ανταλλαγής πληροφοριών μεταξύ δύο ή περισσότερων πρακτόρων. Ο κύριος σκοπός μιας συνεδρίας είναι το να εγκαθιδρύσει το θέμα της επικοινωνίας, αλλά και το περιεχόμενο και την κατεύθυνση των μηνυμάτων που ανταλλάσσουν οι πράκτορες της επικοινωνίας.

Στα [] έχουμε τους πρώτους φορμαλισμούς της επικοινωνίας παράλληλων διεργασιών σε όρους ενός κλασσικού συστήματος τύπων.

Στο [] έχουμε μια πρώτη απόπειρα φορμαλισμού της επικοινωνίας παράλληλων διεργασιών, με την εισαγωγή των θεμελιωδών λειτουργιών επικοινωνίας:

- $k!x$, δηλαδή αποστολή μιας τιμής x μέσω του καναλιού k .
- $k?x$, δηλαδή λήψη μιας τιμής x μέσω του καναλιού k .

Το κανάλι επικοινωνίας k δημιουργείται ρητά από τους δύο πράκτορες με βάση μια κοινώς αποδεκτή θύρα στην αρχή της επικοινωνίας, με χρήση δύο άλλων θεμελιωδών λειτουργιών

- $request(a, k)$, που αποτελεί ένα αίτημα εγκαθίδρυσης ενός καναλιού επικοινωνίας k μέσω της θύρας a .
- $accept(a, k)$, που αποτελεί αποδοχή ενός αίτημα εγκαθίδρυσης ενός καναλιού επικοινωνίας k μέσω της θύρας a .

Έχουμε επίσης την εισαγωγή της έννοιας ενός typed καναλιού επικοινωνίας, δηλαδή την απόδοση ενός τύπου στο κανάλι επικοινωνίας με βάση τους τύπους των δεδομένων που ανταλλάσσονται μέσω αυτού, και την κατεύθυνση με την οποία αυτά "ρέουν" μέσω του καναλιού.

Αυτό πρακτικά σημαίνει ότι ένα κανάλι επικοινωνίας μπορεί να έχει τον τύπο

$$a : \uparrow int \downarrow int$$

που σημαίνει ότι μια διεργασία που χρησιμοποιεί τη θύρα a για να επικοινωνεί θα:

- στείλει μια τιμή τύπου ακεραίου (int), το οποίο συμβολίζεται με \uparrow , και ακολούθως θα
- λάβει μια τιμή τύπου ακεραίου, το οποίο συμβολίζεται με \downarrow

Επακόλουθο του typing ενός καναλιού επικοινωνίας, είναι ο ορισμός των σφαλμάτων επικοινωνίας, που διαχωρίζονται σε άμεσα σφάλματα, δηλαδή σφάλματα όμοια με αυτά κλασσικών συστημάτων τύπων όπου έχουμε εφαρμογή τελεστών σε τελεσταίους λάθος τύπων, και προβλήματα ασυμβατότητας των στοιχειωδών πράξεων επικοινωνίας που πραγματοποιούν ανα πάσα στιγμή οι δύο πράκτορες. Αυτό το είδος προβλήματος εμφανίζεται όταν οι πράξεις αυτές δεν είναι συμπληρωματικές. Αξιοσημείωτο εδώ είναι ότι οι Takeuchi et al. δεν ασχολούνται με τη συμπληρωματικότητα ως προς τον τύπο των δεδομένων που ανταλλάσσονται (δηλαδή μια διεργασία να στέλνει έναν ακέραιο και η διεργασία με την οποία επικοινωνεί να περιμένει έναν ακέραιο), αλλά μόνο ως προς το είδος της πράξης (μια διεργασία να στέλνει, ενώ η άλλη περιμένει να λάβει μια τιμή από το κανάλι).

Ο πιο πρόσφατος φορμαλισμός των τύπων συνεδριών κάνει διαχωρισμό ανάμεσα σε μια καθολική και μια τοπική περιγραφή της επικοινωνίας. Στο [] αναλύονται δύο άλγεβρες βασισμένες σε μια επέκταση της π-άλγεβρας και των τύπων συνεδριών:

- Η πρώτη είναι η άλγεβρα που αποτελεί το φορμαλισμό της καθολικής περιγραφής. Σε αυτήν μπορεί να περιγραφεί η αλληλεπίδραση των πρακτόρων της επικοινωνίας σε ένα αφηρημένο επίπεδο, μόνο σαν μια αλληλουχία από ενέργειες μεταξύ συμμετεχόντων
- Η δεύτερη είναι ο φορμαλισμός της τοπικής περιγραφής. Σε αυτήν περιγράφεται η επικοινωνία από τη σκοπιά ενός συμμετέχοντα στην επικοινωνία (ενός πράκτορα)

Οι δύο άλγεβρες συνδέονται μεταξύ τους με μια σχέση προβολής, η οποία με βάση την καθολική περιγραφή της επικοινωνίας (τη χορευογραφία), παράγει την περιγραφή που αντιστοιχεί στα διάφορα endpoints αυτής.

Κεφάλαιο 5

Ένα στατικό σύστημα τύπων συνεδρίας σε Erlang

Οι τύποι συνεδρίας είναι ένας από τους φορμαλισμούς που έχουν προταθεί για την δόμηση και την ανάλυση της αλληλεπίδρασης μεταξύ διεργασιών.

5.1 Το σύστημα τύπων

Τα annotations που αποτελούν την επέκταση του υπάρχοντος συστήματος τύπων της Erlang με τις επιδράσεις έχουν ως εξής:

$sestype$	$::=$	$monosestype$ $ $ $sestype$
$monosestype$	$::=$	$(datatype_1, \dots, datatype_n) \rightarrow$ $datatype; \phi$
ϕ	$::=$	\emptyset $ $ $sesaction. \phi$ $ $ $\mu \sigma . \phi$ $ $ $(\mu \sigma . \phi). \phi$
$sesaction$	$::=$	$datatype; !^P datatype$ $ $ $datatype_1; ? datatype_2$ $ $ $\& < datatype_1; \phi_1, \dots, datatype_n; \phi_n >$ $ $ $\oplus < datatype_1; \phi_1, \dots, datatype_n; \phi_n >$ $ $ σ
$datatype$	$::=$	$pid[process]$ $ $ $Type$ $ $ $sestype$
$process$	$::=$	$self$ $ $ P

$$\begin{array}{ll} PE & ::= \emptyset \\ & | PE, P \end{array} \quad \begin{array}{ll} SE & ::= \emptyset \\ & | SE, \sigma \end{array}$$

Ο τύπος $Type$ στον ορισμό του $datatype$ είναι ο τύπος ενός Erlang term, όπως αυτό περιγράφεται στο Erlang TypeSpec.

Ως ϕ ορίζουμε την επίδραση μιας έκφρασης e σε Erlang.

Τα περιβάλλοντα τύπων μας είναι τα εξής:

- DE το περιβάλλον των ζευγών (x, t) , όπου x ένα Erlang Term, και t ο τύπος του
- PE το περιβάλλον των ζευγών (n, p) , όπου n το όνομα ενός process, όπως αυτό συναντάται στο πρόγραμμα, και p η διεύθυνση χρόνου εκτέλεσης της διεργασίας.

- SE το περιβάλλον των ζευγών (f, st) όπου f το όνομα μιας συνάρτησης, και st ο τύπος συνεδρίας της.

Ορίζουμε ως $\Gamma = \{DE, PE, SE\}$ την τριάδα που μας δίνει τα type schemes.

Οι κανόνες συμπερασμού τύπων ακολουθούν:

$$\frac{\Gamma \vdash e_1 : pid[process]; \phi_1 \quad \Gamma \vdash e_2 : datatype; \phi_2}{\Gamma \vdash e_1 ! e_2 : datatype; \phi_2, \phi_1, !^{process} datatype} (send)$$

$$\frac{\Gamma \vdash e : datatype_e; nothing \quad DE'; PE'; SE \vdash S : datatype; \phi_S}{\Gamma \vdash receive \rightarrow Send : datatype; ?datatype_e, \phi_S} (recv)$$

$$\frac{\Gamma \vdash e_i : datatype_{e_i}; nothing \quad DE'; PE'; SE \vdash S_i : datatype_i; \phi_{S_i}}{\Gamma \vdash receive e_i \rightarrow S_i^{i \in \{1..N\}} end : datatype_j^{j \in \{1..N\}} \& \langle ?datatype_1 . \phi_1, \dots, ?datatype_N . \phi_N \rangle} (offer)$$

$$\frac{\Gamma \vdash S_i : datatype_i; \phi_i \quad Pref(\phi_i^{i \in \{1..N\}}) = !^P datatype}{\Gamma \vdash case e of p_i \rightarrow S_i^{i \in \{1..N\}} end : datatype_j^{j \in \{1..N\}}; \oplus \langle \phi_1, \dots, \phi_N \rangle} (choice_case)$$

$$\frac{\Gamma \vdash Expr : datatype_e; sesaction \quad \Gamma \vdash ExprSequence : datatype; \phi}{\Gamma \vdash Expr, ExprSequence : datatype; sesaction, \phi} (sequencing)$$

Ο κανόνας *send* περιγράφει την επίδραση μιας αποστολής μηνύματος. Καθώς πρώτα αποτιμώνται οι τιμές των e_2 και e_1 , οι επιδράσεις τους θα παρατηρηθούν πρώτες. Ο τύπος του αριστερού τελεσταίου της έκφρασης αποστολής πρέπει πάντα να είναι η διεύθυνση μιας διεργασίας. Ο τύπος του μηνύματος μπορεί να είναι οποιοσδήποτε έγκυρος τύπος δεδομένων ενός Erlang term. Στο τέλος, παρατηρείται και η επίδραση της αποστολής του μηνύματος e_2 .

Ο κανόνας *recv* περιγράφει την επίδραση μιας λήψης μηνύματος, όταν μια διεργασία "ακούει" μόνο για μηνύματα ενός συγκεκριμένου τύπου $datatype_e$. Σε Erlang, το pattern του μηνύματος πρέπει να είναι ένα απλό term, με αποτέλεσμα ποτέ να μην είναι effectful, εξού και η επίδρασή του (*nothing*). Κατά την αποτίμηση του σώματος του receive clause, χρησιμοποιούμε το περιβάλλον $\Gamma' = \{DE', PE', SE\}$, όπου τα DE' και PE' προκύπτουν από τα DE και PE προσθέτοντας τυχόν μεταβλητές που περιέχονται στο receive pattern.

Ο κανόνας *offer* περιγράφει την επίδραση μιας λήψης μηνύματος, όταν μια διεργασία "ακούει" για μηνύματα διαφορετικών μορφών. Και εδώ τα patterns είναι effectless.

Ο κανόνας *choice_case* περιγράφει την επίδραση μιας έκφρασης που προκαλεί διακλάδωση στην εκτέλεση, σε περίπτωση που τα σώματα των clauses είναι effectful, και οι επιδράσεις τους ξεκινούν όλες με ένα *send* action.

Ο κανόνας *sequencing* περιγράφει την επίδραση ενός συνόλου από εκφράσεις, οι οποίες αποτιμώνται με μια γραμμική σειρά. Αυτός είναι ο κανόνας με βάση των οποίου συμπεραίνουμε τους τύπους και τις επιδράσεις συναρτήσεων.

Στο σύστημα τύπων και επιδράσεων μας, οι μόνες εκφράσεις που έχουν κάποια επίδραση είναι αυτή της αποστολής μηνύματος (!), και αυτή της λήψης μηνύματος (*receive...end*). Όλες οι υπόλοιπες Erlang εκφράσεις είναι effectless.

5.2 Υλοποίηση

5.2.1 Τμήματα

Καθόλη τη διάρκεια αυτής της ενότητας, θα χρησιμοποιούμε το απλό πρόγραμμα που φαίνεται στο σχήμα 5.1 ως βάση για να εξηγήσουμε τη λειτουργία του συστήματός μας.

Listing 5.1: Απλό παράδειγμα

```
1 run() ->
2   ProcServer = spawn(fun() -> server() end),
3   Foo = spawn(fun() -> foo() end),
4   Biz = spawn(fun() -> biz() end),
5   spawn(fun() -> baz(Biz) end),
6   spawn(fun() -> bar(Foo) end),
7   spawn(fun() -> simple_client(ProcServer) end),
8   spawn(fun() -> recursive_client(ProcServer, rand:uniform(100)) end).
9
10 server() ->
11   receive
12     {num_1, Proc, X} ->
13       receive
14         {num_2, Proc, Y} ->
15           Proc ! {num_sum, self(), X + Y + 42},
16           server()
17       end;
18     {foo, Proc, X} -> server()
19   end.
20
21 simple_client(Server) ->
22   Server ! {num_1, self(), 17},
23   Server ! {num_2, self(), rand:uniform(100)},
24   receive
25     {num_sum, Server, X} -> X
26   end.
27
28 recursive_client(Server, 0) -> ok;
29 recursive_client(Server, N) ->
30   Server ! {foo, self(), N},
31   recursive_client(Server, N - 1).
32
33 foo() ->
34   receive
35     {foo, P, 42} ->
36       receive
37         {foo, P, 17} ->
38           P ! {foo, self(), 42 + 17},
39           foo()
40       end
41   end.
42
43 bar(Foo) ->
44   Foo ! {foo, self(), 42},
45   Foo ! {foo, self(), 17},
46   receive
47     {foo, Foo, X} ->
48       Foo ! {foo, self(), 42},
49       Foo ! {foo, self(), 17},
```

```

50         receive
51             {foo, Foo, X} -> X + 42
52         end
53     end.
54
55     baz(Biz) ->
56         Biz ! {foo, self(), 42},
57         Biz ! {bar, self(), 17},
58         baz(Biz).
59
60     biz() ->
61         receive
62             {foo, Pid, X} ->
63                 receive
64                     {bar, Pid, _Y} ->
65                         receive
66                             {foo, Pid, X} -> biz()
67                         end
68                 end
69         end.

```

Listing 5.1: Απλό παράδειγμα

Το συστήμά μας αποτελείται από τα εξής επιμέρους κομμάτια:

- Το κομμάτι του συμπερασμού των τύπων συνεδριών του προγράμματος που δεχόμαστε ως είσοδο
- Το κομμάτι του ελέγχου των τύπων που συμπέρανε το σύστημά μας με τους τύπους που παρέχονται από τον χρήστη ως ο "προδιαγεγραμμένος" τύπος συνεδρίας ενός τμήματος του προγράμματος
- Το κομμάτι της σύγκρισης των τύπων δύο συναρτήσεων προκειμένου να διαπιστωθεί αν αποτελούν τα "άκρα" ενός πρωτοκόλλου επικοινωνίας

Όλα τα προγράμματα που παρέχονται προς ανάλυση "περνάνε" από το πρώτο κομμάτι. Το δεύτερο και το τρίτο ενεργοποιούνται με βάση την είσοδο του χρήστη.

5.2.2 Συμπερασμός Τύπων Συνεδριών

Όπως έχουμε αναφέρει και προηγουμένως, η Erlang είναι μια γλώσσα με δυναμικούς τύπους, πράγμα το οποίο σημαίνει ότι δεν υπάρχουν πληροφορίες τύπων για τα δεδομένα ενός προγράμματος κατά το χρόνο μεταγλώττισης.

Στο κεφάλαιο 4 είδαμε ότι στους τύπους συνεδρίας, ένα θεμελιώδες κομμάτι της ανάλυσης είναι η γνώση των τύπων δεδομένων που περιέχονται στα μηνύματα που ανταλλάσσονται μεταξύ των επικοινωνουσών διεργασιών.

Προκειμένου να αποκτήσουμε αυτήν την πληροφορία για τους σκοπούς της δικής μας ανάλυσης, αξιοποιούμε τον Dialyzer. Με αυτόν τον τρόπο, ο συμπερασμός τύπων συνεδρίας του συστήματός μας πραγματοποιείται σε δύο φάσεις:

- Στην πρώτη φάση, δίνουμε το προς ανάλυση πρόγραμμα ως είσοδο στον Dialyzer. Μετά το τέλος της εκτέλεσής του, έχουμε συμπεράνει τους τύπους των δεδομένων του προγράμματος.
- Στη δεύτερη φάση, επανεπεξεργαζόμαστε όλο το πρόγραμμα, και συμπεραίνουμε τους τύπους συνεδριών των συναρτήσεων του με βάση τους κανόνες της προηγούμενης ενότητας, και αξιοποιώντας την πληροφορία που εξήγαγε ο Dialyzer, την οποία χρησιμοποιήσαμε για να "γεμίσουμε" το περιβάλλον Γ με τις αντιστοιχίες μεταβλητών με τύπους.

Για μια ανάλυση του τρόπου με τον οποίο δουλεύει ο Dialyzer, ο αναγνώστης παραπέμπεται στα [].

Χρησιμοποιούμε μια έκδοση του Dialyzer βασισμένη σε αυτήν που περιέχεται στην έκδοση 19.0 του Erlang/OTP, ελαφρώς τροποποιημένη για τους σκοπούς μας, ώστε να διατηρεί την πληροφορία που σχετίζεται με τους τύπους δεδομένων των επιμέρους Erlang terms καθόλη τη διάρκεια της ανάλυσης, και ακολούθως να τη μεταδίδει στο σύστημά μας.

Μόλις ολοκληρωθεί η ανάλυση του προγράμματος από τον Dialyzer, τρέχουμε τη δική μας σημασιολογική ανάλυση πάνω στο Core Erlang AST του κάθε module, αποδίδοντας σε κάθε κόμβο ένα effect, με βάση τους κανόνες συμπερασμού που περιγράφηκαν παραπάνω.

Οι κλήσεις συναρτήσεων βιβλιοθήκης της Erlang, πέραν του `!(send)`, είναι effectless.

Κατά την ανάλυση των συναρτήσεων ενός module, είναι πολύ πιθανό να συναντήσουμε κλήσεις σε άλλες, ορισμένες από τον χρήστη, συναρτήσεις εντός του module (ή και μέσα σε άλλα modules του προγράμματος), για τις οποίες όμως δεν έχουμε ακόμη διαθέσιμη την πλήρη πληροφορία σχετικά με τον τύπο συνεδρίας τους, είτε επειδή θα αναλυθούν σε μετέπειτα χρονική στιγμή, είτε επειδή και εκείνες περιείχαν κλήση σε κάποια άλλη συνάρτηση.

Κατά τη φάση του συμπερασμού τύπων, στη θέση της επίδρασης της καλούμενης συνάρτησης βάζουμε μια τιμή placeholder, την `{unavailable, {Module, Function, Arity}}`, όπου *Module* το module στο οποίο ορίζεται η καλούμενη συνάρτηση, *Function* το όνομα της καλούμενης συνάρτησης, και *Arity* η πολλαπλότητα αυτής.

Στο τέλος της ανάλυσης των modules του προγράμματος, πραγματοποιούμε ένα iteration πάνω σε όλες τις ορισμένες συναρτήσεις, και κάνουμε resolve όλες τις επιδράσεις που δεν ήταν διαθέσιμες κατά τη διάρκεια αυτής. Εδώ, resolution σημαίνει η αντικατάσταση των placeholders `{unavailable, {Module, Function, Arity}}` με το πλήρες session type της καλούμενης συνάρτησης στο session type της καλούσας.

Αυτός είναι και ο λόγος που δε μπορούμε να διαχειριστούμε αμοιβαίως αναδρομικές συναρτήσεις. Η πλήρης ανάπτυξη του session type της μιας στο session type της άλλης θα οδηγούσε σε ένα undecidable πρόβλημα.

Για το παράδειγμα του σχήματος 5.1, το σύστημά μας συμπεραίνει τους εξής τύπους συνεδριών:

Listing 5.2: Inferred session types

```
1 {server,0} : "u (4). Of<any; ?(atom x {atom, pid, (atom x atom)} x number). ?(atom x {atom, pid, (atom x atom)} x number).
2           pid(1) !(atom x pid x number).rec(4).,any; ?(atom x {atom, pid, (atom x atom)} x number).rec(4).>."
3
4 {simple_client,1} : "pid(1) !(atom x pid x integer). pid(1) !(atom x pid x integer). ?(atom x pid x any)."
```

Listing 5.2: Inferred session types

5.2.3 Έλεγχος σε σχέση με το specification

Η Erlang παρέχει τη δυνατότητα στον προγραμματιστή να προσθέσει annotations στα modules των προγραμμάτων του, τα οποία είναι διαθέσιμα κατά τη φάση της μεταγλώττισης.

Εισάγουμε ένα νέο τέτοιο annotation, `-sestype()`. Παρέχουμε στον προγραμματιστή ένα σύνολο από macros, με βάση τα οποία μπορεί να περιγράψει τον αναμενόμενο τύπο συνεδρίας για μια συνάρτηση, όπως φαίνεται στο σχήμα 5.3. Τα annotations αυτά αντιστοιχούν στις συναρτήσεις `server/0`, `simple_client/1` και `recursive_client/2` του σχήματος 5.1.

Listing 5.3: -sestype() annotation

```
1 -sestype({{server, 0},
2           ?f_rec_tag(1, [?sa_offer_tag([?dt_any_tag, [?sa_recv_tag(?dt_any_tag,
```

```

3                                     ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
4                                     ?sa_recv_tag(?dt_any_tag,
5                                     ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
6                                     ?sa_send_tag(1,
7                                     ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
8                                     ?sa_rec_tag(1)[]),
9                                     {?dt_any_tag, [?sa_recv_tag(?dt_any_tag,
10                                     ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
11                                     ?sa_rec_tag(1)[]])}).
12
13 -sestype({{simple_client, 1},
14          [?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
15          ?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
16          ?sa_recv_tag(?dt_integer_tag,
17          ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag]))]}).
18
19 -sestype({{recursive_client, 2},
20          ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])), ?sa_rec_tag(1)[]])}).

```

Listing 5.3: -sestype() annotation

Για κάθε τέτοιο annotation που έχει χρησιμοποιήσει ο προγραμματιστής, το εργαλείο μας συγκρίνει τον τύπο συνεδρίας που συμπέρανε, με αυτόν που περιγράφεται στο annotation.

Η σύγκριση μεταξύ αυτών των τύπων συνεδρίας και των συμπερασμένων γίνεται καταναλώνοντας ένα-ένα τα session actions και από τους δύο τύπους, και ελέγχοντας:

- Ότι τα actions που περιγράφουν είναι ίδια
- Ότι το datatype του user annotation για το συγκεκριμένο action είναι υποτύπος του inferred session type τύπου.

Το δεύτερο κριτήριο είναι αυτό που κάνει το σύστημά μας πλήρες: ποτέ δε θα απορρίψει ένα σωστό πρόγραμμα.

Αναγνωρίζοντας τους περιορισμούς που αντιμετωπίζουμε στο συμπερασμό των τύπων δεδομένων, το να απαιτούμε οι συμπερασμένοι τύποι δεδομένων να είναι πάντοτε ακριβώς ίδιοι με αυτούς που προσδιορίζει ο χρήστης στα `—sestype()` annotations θα οδηγούσε σε απόρριψη αρκετών σωστών προγραμμάτων, λόγω mismatch.

Για να ξεπεράσουμε αυτό το πρόβλημα, επιτρέπουμε στους τύπους που παρέχει ο χρήστης να είναι τουλάχιστον τόσο συγκεκριμένοι (με την έννοια του subtyping) όσο οι τύποι που συμπεραίνει ο Dialyzer. Αν ο παρεχόμενος τύπος δεν είναι υποτύπος του συμπερασμένου, σημειώνουμε ένα error στην υλοποίηση του πρωτοκόλλου επικοινωνίας.

Για παράδειγμα, στο πρώτο clause της `simple_send/3`, αν ο χρήστης είχε δώσει

Listing 5.4: -sestype() annotation

```

1 -sestype({{server, 0},
2          ?f_rec_tag(1, [?sa_offer_tag([?dt_any_tag, [?sa_recv_tag(?dt_any_tag,
3          ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
4          ...

```

Listing 5.4: -sestype() annotation

ο έλεγχος θα είχε αποτύχει, καθώς το `any` αποτελεί τον υπερτύπο όλων των τύπων δεδομένων της Erlang.

Το type lattice δίνεται στο σχήμα 5.X.

5.2.4 Έλεγχος peers

Πέραν του `—sestype()`, εισάγουμε και άλλο ένα annotation: `—peers()`. Χρησιμοποιώντας αυτό, ο προγραμματιστής μπορεί να υποδείξει στο σύστημά μας πως πρέπει να ελέγξει ότι δύο συναρτήσεις αποτελούν τα δύο άκρα ενός πρωτοκόλλου επικοινωνίας. Αυτό σημαίνει ότι πρέπει να ελεγχθεί αν οι τύποι τους είναι συμπληρωματικοί.

Δύο συναρτήσεις είναι communication peers αν, καταναλώνοντας τους τύπους τους ένα-ένα action, κάθε ζεύγος από αυτά είναι συμπληρωματικό, και εάν κάποιος τύπος εξαντληθεί πρόωρα, ο τύπος της άλλης συνάρτησης μένει σε μια κατάσταση η οποία μπορεί να θεωρηθεί τερματική.

Ο Αλγόριθμος Μοντελοποιούμε τους τύπους συνεδριών της κάθε συνάρτησης ως ένα (πιθανώς κυκλικό) κατευθυνόμενο γράφημα $G(V, E)$. Κάθε κορυφή στο V αναπαριστά ένα action. Μια ακμή e από την κορυφή u στην κορυφή v σημαίνει ότι το action που περιγράφεται από τη v ακολουθεί αυτό που περιγράφεται από τη u στον τύπο συνεδρίας της συνάρτησης.

Algorithm: Session Type Matching

```

1  input: graph  $G_1$ , graph  $G_2$ 
2  output: list of successful routes
3  begin
4     $S = \{ \{ \text{root}(G_1), \text{root}(G_2) \} \}$ 
5     $\text{res} = []$ 
6    while  $S$  not empty:
7       $\text{Pair} = S.\text{pop}()$ 
8      if  $\text{complementary}(\text{Pair})$ :
9        if  $\text{next}(\text{Pair}[G_1])$  is empty:
10         if  $\text{next}(\text{Pair}[G_2])$  is empty:
11            $\text{res}.\text{push}(\text{current route})$ 
12         else:
13           if  $\text{valid}(\text{recursive\_criteria}(\text{Pair}[G_2]))$ :
14              $\text{res}.\text{push}(\text{current route})$ 
15       else:
16         if  $\text{next}(\text{Pair}[G_2])$  is empty:
17           if  $\text{valid}(\text{recursive\_criteria}(\text{Pair}[G_1]))$ :
18              $\text{res}.\text{push}(\text{current route})$ 
19         else:
20           if  $\text{valid}(\text{recursive\_criteria}(\text{Pair}[G_1]))$  and
21              $\text{valid}(\text{recursive\_criteria}(\text{Pair}[G_2]))$ :
22              $\text{res}.\text{push}(\text{current route})$ 
23         else:
24           foreach  $v$  in  $\text{recursive\_criteria}(\text{Pair}[G_1])$ :
25             foreach  $u$  in  $\text{recursive\_criteria}(\text{Pair}[G_2])$ :
26                $S.\text{push}(\{u, v\})$ 
27
28   return  $\text{res}$ 
29 end

```

Algorithm: Session Type Matching

Το σύνολο αναζήτησης S είναι μια ουρά από ζευγάρια κορυφών, των οποίων τα actions προσπαθούμε να ελέγξουμε ως προς τη συμπληρωματικότητα. Κρατάμε το μονοπάτι το οποίο μας οδήγησε στο κάθε ζευγάρι κορυφών, τόσο ως προς το από ποιες κορυφές περάσαμε, όσο και ως προς τον τύπο συνεδρίας με τον οποίο "φτάσαμε" στα παρόντα actions.

Ο αλγόριθμός μας ξεκινά με το να προσθέτει τα ζευγάρια των κορυφών-ριζών του κάθε γραφήματος στο σύνολο αναζήτησης S , όπως φαίνεται στη γραμμή 4 του παραπάνω αλγορίθμου. Κάθε γράφημα μπορεί να έχει περισσότερους του ενός αρχικούς κόμβους, σε περιπτώσεις όπου το πρώτο action είναι ένα *choice* ή ένα *offer*.

Συνεχίζει με το να βγάξει από την ουρά το πρώτο ζευγάρι κορυφών, και να συγκρίνει τα actions που αναπαριστούν. Αν τα actions τους δεν είναι συμπληρωματικά, τότε θεωρούμε ότι το παρόν μονοπάτι αποτελεί ένα μη επιτυχές match, και προχωράμε στο επόμενο ζεύγος κορυφών.

Αν τα actions τους είναι συμπληρωματικά, ελέγχουμε το σύνολο των έξω-γειτόνων της κάθε κορυφής. Εάν και τα δύο είναι άδεια, τότε καταναλώσαμε πλήρως και τους δύο τύπους, και μπορούμε να κρατήσουμε το μονοπάτι που ακολουθήσαμε στο σύνολο των έγκυρων ακολουθιών.

Εάν οποιοδήποτε από τα δύο σύνολα δεν είναι άδειο, ελέγχουμε το είδος των έξω-γειτόνων με τον αλγόριθμο:

Algorithm: Out-neighbour criteria

```

1  input: graph  $G$ , vertex  $v$ 
2  output: list of out-neighbours of  $v$  which are not
3           starts of recursive sequences
4  begin
5      res = []
6      min_depth = inf
7      foreach out-neighbour  $u$  of  $v$ :
8          if  $u$  not (start of recursive sequence or
9                  unvisited start of recursive sequence):
10             res.push( $u$ )
11          else:
12             min_depth = min(min_depth, depth( $u$ ))
13
14      if res == [] && min_depth == 1:
15          return {true, []}
16      else:
17          return {false, res}
18  end

```

Algorithm: Out-neighbour criteria

Αυτός ο αλγόριθμος ελέγχει όλους τους έξω-γείτονες μιας κορυφής v , και μας λέει αν όλοι τους αποτελούν την πρώτη κορυφή μιας αναδρομικής ακολουθίας, και αν ναι, το ελάχιστο "βάθος" της αναδρομής. Εδώ ορίζουμε ως το επίπεδο εμφώλευσης της αναδρομικής ακολουθίας, καθώς σε ένα session type μπορούμε να έχουμε αναδρομικές υποακολουθίες.

Για παράδειγμα, στον τύπο συνεδρίας

$$\begin{aligned}
 &\mu s. pid(1) ! atom\ x\ integer. \\
 &\quad pid(1) ! atom\ x\ self\ x\ integer. \\
 &\quad (\mu \phi. ? atom\ x\ integer. \phi). \\
 &\quad \sigma
 \end{aligned}$$

η κορυφή που αντιστοιχεί στην πρώτη *send* λειτουργία έχει βάθος 1, καθώς δεν είναι εμφωλευμένη. Η *receive* λειτουργία εντός της αναδρομικής υποακολουθίας έχει βάθος 2.

Οι συνθήκες ενός επιτυχούς ταιριάσματος είναι:

- Δεν υπάρχουν άλλοι έξω-γείτονες, δηλαδή εξαντλήσαμε τα δύο γραφήματα.
- Όλοι οι έξω-γείτονες είναι αρχικές κορυφές αναδρομικών ακολουθιών βάθους 1.

Εάν υπάρχουν έξω-γείτονες οι οποίοι είτε δεν είναι αρχικές κορυφές αναδρομικών ακολουθιών, είτε είναι αρχικές κορυφές αναδρομικών ακολουθιών με βάθος μεγαλύτερο του 1, ή αποτελούν έναν συνδυασμό από αρχικές και μη κορυφές αναδρομικών ακολουθιών βάθους 1, τις οποίες δεν έχουμε επισκεφθεί, τότε προσθέτουμε όλα τα ζευγάρια έξω-γειτόνων στο τέλος του συνόλου αναζήτησης, προκειμένου να τα ελεγχουμε στο μέλλον.

Ακολουθούν μερικά peer annotations για το πρόγραμμα 5.1.

Listing 5.7: -sestype() annotation

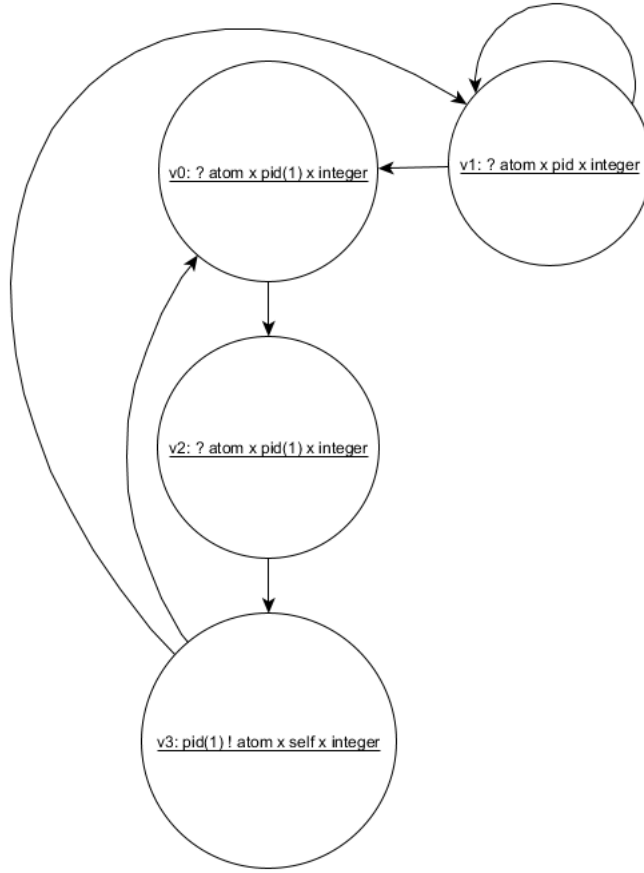
```

1  -peers([{{server, 0}, []}, {{simple_client, 1}, []}]).
2
3  -peers([{{server, 0}, []}, {{recursive_client, 2}, []}]).
4
5  -peers([{{foo, 0}, []}, {{bar, 1}, []}]).
6
7  -peers([{{biz, 0}, []}, {{baz, 1}, []}]).

```

Listing 5.7: -sestype() annotation

Σχήμα 5.1: server/0



Ας αναλύσουμε τα δύο πρώτα ζευγάρια annotations. Τα κατευθυνόμενα γραφήματα των τύπων συνεδρίας των συναρτήσεων *server/0*, *simple_client/1* και *recursive_client/2* φαίνονται στα σχήματα 5.X έως 5.Y.

Στην ανάλυση που ακολουθεί, θα αναφερόμαστε στην κορυφή v του γραφήματος που αντιστοιχεί στη συνάρτηση f/a ως $f/a[v]$.

Αρχικά προσπαθούμε να συσχετίσουμε τους τύπους συνεδρίας των *server/0* και *simple_client/1*. Σύμφωνα με τον αλγόριθμο που περιγράφηκε πριν, το αρχικό σύνολο αναζήτησης έχει ως εξής:

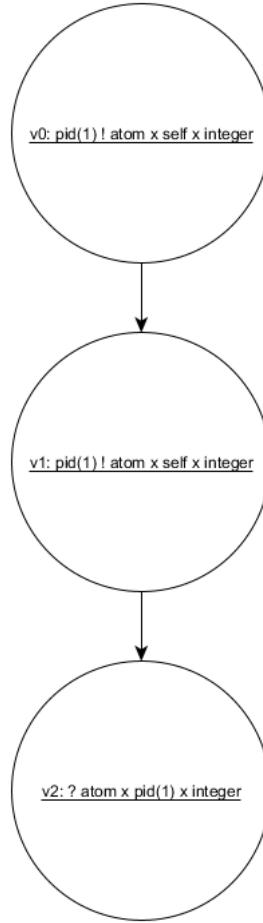
$$S = \{\{server/0[v_0], simple_client/1[v_0]\}, \{server/0[v_1], simple_client/1[v_0]\}\}$$

Θεωρούμε το πρώτο εκ των δύο ζευγών. Ο έλεγχος της γραμμής 8 είναι επιτυχής, καθώς τα session actions που αναπαριστούν οι δύο κορυφές είναι συμπληρωματικά. Κανένα από τα δύο σύνολα έξω-γειτόνων δεν είναι άδεια: το σύνολο *out* για την κορυφή *server/0*[v_0] είναι $\{server/0[v_2]\}$, ενώ για την *simple_client/1*[v_0] είναι το $\{simple_client/1[v_1]\}$. Καθώς όλες οι κορυφές που ανήκουν σε αυτά τα σύνολα αποτυγχάνουν στον έλεγχο των γραμμών 20 και 21, προσθέτουμε το καρτεσιανό τους γινόμενο στο τέλος του συνόλου αναζήτησης, με αποτέλεσμα να έχουμε ως νέο σύνολο αναζήτησης το

$$S = \{\{server/0[v_1], simple_client/1[v_0]\}, \{server/0[v_2], simple_client/1[v_1]\}\}$$

Εκτελώντας τον αλγόριθμο για για το δεύτερο ζεύγος κορυφών του αρχικού συνόλου αναζήτησης, και πάλι έχουμε επιτυχία του ελέγχου της γραμμής 8 καθώς τα session actions είναι συμπληρωματικά. Και πάλι κανένα από τα σύνολα των έξω-γειτόνων των δύο κορυφών δεν είναι άδαιο, οπότε καταλήγουμε να προσθέτουμε το καρτεσιανό γινόμενο των συνόλων αυτών στο σύνολο αναζήτησης, το οποίο

Σχήμα 5.2: *simple_client/1*



σημαίνει ότι σε αυτήν τη φάση αυτό αποτελείται από τα εξής ζεύγη κορυφών:

$$\begin{aligned}
 S = \{ & \{server/0[v_2], simple_client/1[v_1]\}, \\
 & \{server/0[v_1], simple_client/1[v_1]\}, \\
 & \{server/0[v_0], simple_client/1[v_1]\} \}
 \end{aligned} \tag{5.1}$$

Εκτελώντας το βρόχο των γραμμών 6-26 του αλγορίθμου μέχρι να εξαντλήσουμε το σύνολο αναζήτησης, βρίσκουμε ότι πράγματι οι συναρτήσεις *server/0* και *simple_client/1* αποτελούν communication peers, και τα συμπληρωματικά session sequences μεταξύ τους είναι τα:

$$\begin{aligned}
 & \{server/0[v_0], simple_client/1[v_0]\} \rightarrow \\
 \rightarrow & \{server/0[v_2], simple_client/1[v_1]\} \rightarrow \\
 \rightarrow & \{server/0[v_3], simple_client/1[v_2]\}
 \end{aligned} \tag{5.2}$$

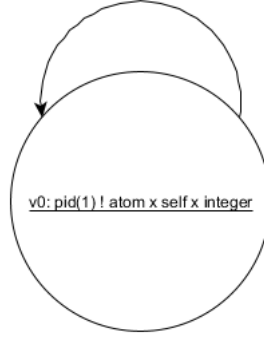
Κανένα sequence που ξεκινάει από το ζευγάρι $\{server/0[v_1], simple_client/1[v_0]\}$ δεν οδηγεί σε μια επιτυχή συνεδρία.

Προχωράμε στην επικοινωνία μεταξύ των *server/0* και *recursive_client/2*. Το αρχικό σύνολο αναζήτησης είναι το

$$S = \{ \{server/0[v_0], recursive_client/2[v_0]\}, \{server/0[v_1], recursive_client/2[v_0]\} \}$$

Και τα δύο ζευγάρια αποτελούνται από συμπληρωματικά μεταξύ τους actions, οπότε αφού έχουν

Σχήμα 5.3: recursive_client/2



καταναλωθεί από το σύνολο αναζήτησης, αυτό θα έχει διαμορφωθεί ως εξής:

$$S = \{\{server/0[v_2], recursive_client/2[v_0]\}, \\ \{server/0[v_1], recursive_client/2[v_0]\}, \\ \{server/0[v_0], recursive_client/2[v_0]\}\} \quad (5.3)$$

Εξαντλώντας και πάλι το σύνολο αναζήτησης, βλέπουμε ότι το ζευγάρι session actions

$$\{server/0[v_1], recursive_client/2[v_0]\}$$

καθιστά τις *server/0* και *recursive_client/2* communication peers.

Πρωτόκολλα Πολλαπλών Συμμετεχόντων Ο παραπάνω αλγόριθμος θεωρεί ότι υπάρχουν μόνο δύο συμμετέχοντες σε κάποια συνεδρία. Σε περίπτωση που θέλαμε να επεκτείνουμε και να γενικεύσουμε τον αλγόριθμο αυτό ώστε να δουλεύει για πολλαπλούς συμμετέχοντες, θα ερχόμασταν αντιμέτωποι πολύ γρήγορα με το πρόβλημα της συνδυαστικής έκρηξης (combinatoric explosion).

Εαν υποθέσουμε ότι προσπαθούμε να τσεκάρουμε την επικοινωνία μεταξύ n διεργασιών, τότε σε κάθε βήμα θα έπρεπε να θεωρούμε έως και $\binom{n}{2}$ ζευγάρια. Κάτι τέτοιο, αν και αργό, ίσως να είναι εφικτό μόνο για μικρές τιμές του n , και για σχετικά σύντομους και απλούς τύπους συνεδριών.

5.3 Παράδειγμα υλοποίησης: Arithmetic Server

Στην παρούσα ενότητα, παρουσιάζουμε ένα κλασσικό παράδειγμα στην ανάλυση συστημάτων τύπων συνεδριών. Το παράδειγμα αυτό αποτελεί μια υλοποίηση του αναδρομικού arithmetic server που πρωτοπεριγράφηκε στο [], με μια διαφορά: έχουμε εξαλείψει την έννοια του "καναλιού" μέσω του οποίου ένας client αποστέλει στον διακομιστή το αίτημά του, και το οποίο δημιουργείται κατά το αίτημα.

Το πρόγραμμα που υλοποιεί τον arithmetic server, καθώς και clients που αιτούνται πράξεις από αυτόν, έχει ως εξής:

Listing 5.8: Arithmetic Server Implementation

```

1 run() ->
2   Server = spawn(fun server/0),
3   process_flag(trap_exit, true),
4   spawn_link(fun() -> sum_client(Server, rand:uniform(10)) end),
5   spawn_link(fun() -> neg_client(Server, rand:uniform(10)) end),
6   spawn_link(fun() -> sqrt_client(Server, rand:uniform(10)) end),
7   await_termination(Server, 3).
8
9 await_termination(Server, 0) -> Server ! {quit, self()}, ok;
10 await_termination(Server, N) ->
11   receive
12     {'EXIT', _, _} -> await_termination(Server, N - 1)

```

```

13     end.
14
15 server() →
16     receive
17         {sum, Pid} →
18             receive
19                 {sum_o1, Pid, Opand1} →
20                     receive
21                         {sum_o2, Pid, Opand2} →
22                             Pid ! {res, self(), Opand1 + Opand2},
23                             server()
24                     end
25             end;
26     {neg, Pid} →
27         receive
28             {neg_o, Pid, Opand} →
29                 Pid ! {res, self(), - Opand},
30                 server()
31         end;
32     {sqrt, Pid} →
33         receive
34             {sqrt_o, Pid, Opand} →
35                 Pid ! {res, self(), math: sqrt(Opand)},
36                 server()
37         end;
38     {quit, _} →
39         exit(normal)
40 end.
41
42 -spec sum_client(pid(), non_neg_integer()) → ok.
43 sum_client(_Server, 0) → ok;
44 sum_client(Server, N) →
45     Op1 = rand:uniform(1000),
46     Op2 = rand:uniform(1000),
47     Server ! {sum, self()},
48     Server ! {sum_o1, self(), Op1},
49     Server ! {sum_o2, self(), Op2},
50     receive
51         {res, Server, _Res} →
52             io:format("sum_client: Sent ~p and ~p, received ~p\n", [Op1, Op2, _Res]),
53             sum_client(Server, N - 1)
54     end.
55
56 -spec neg_client(pid(), non_neg_integer()) → ok.
57 neg_client(_Server, 0) → ok;
58 neg_client(Server, N) →
59     Op = rand:uniform(1000),
60     Server ! {neg, self()},
61     Server ! {neg_o, self(), Op},
62     receive
63         {res, Server, Res} →
64             io:format("neg_client: Sent ~p, received ~p\n", [Op, Res]),
65             neg_client(Server, N - 1)
66     end.
67
68 sqrt_client(_Server, 0) → ok;
69 sqrt_client(Server, N) →
70     Op = rand:uniform(1000),
71     Server ! {sqrt, self()},
72     Server ! {sqrt_o, self(), Op},
73     receive
74         {res, Server, _Res} →
75             io:format("sqrt_client: Sent ~p, received ~p\n", [Op, _Res]),
76             sqrt_client(Server, N - 1)
77     end.

```

Listing 5.8: Arithmetic Server Implementation

Και ακολουθούν τα annotations προς το σύστημά μας, που περιγράφουν το πρωτόκολλο που πρέπει να ακολουθείται από την υλοποίησή μας.

Listing 5.9: Arithmetic Server Implementation

```

1 -sestype({{server, 0},
2     ?f_rec_tag(1, [?sa_offer_tag([?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([])),
3         ?sa_recv_tag(?dt_integer_tag, ?dt_tuple_tag([])),
4         ?sa_recv_tag(?dt_integer_tag, ?dt_tuple_tag([])),
5         ?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_rec_tag(1)]),
6     ?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([])),
7     ?sa_recv_tag(?dt_integer_tag, ?dt_tuple_tag([])),
8     ?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_rec_tag(1)]),
9     ?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([])),
10    ?sa_recv_tag(?dt_integer_tag, ?dt_tuple_tag([])),
11    ?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_rec_tag(1)]),
12    ?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([]))]}))}.
13
14 -sestype({{sum_client, 2},
15     ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_send_tag(1, ?dt_tuple_tag([])),
16     ?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([])),
17     ?sa_rec_tag(1)]))}.
18
19 -sestype({{neg_client, 2},
20     ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([]), ?sa_send_tag(1, ?dt_tuple_tag([])),
21     ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([]), ?sa_rec_tag(1)]))}.

```

```

22 -sestype({{sqrt_client, 2},
23         ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([])), ?sa_send_tag(1, ?dt_tuple_tag([])),
24         ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([])), ?sa_rec_tag(1)]))}.
25

```

Listing 5.9: Arithmetic Server Implementation

Τα contracts αυτά μας λένε το εξής:

- Η συνάρτηση *server/0* αρχικά περιμένει να λάβει μια τούπλα, η οποία αποτελείται από ένα *atom* (το οποίο, όπως φαίνεται από τον κώδικα, περιγράφει την αριθμητική πράξη που ο *client* θέλει να εκτελεστεί εκ μέρους του από τον *server*), και ένα *pid*, το οποίο αντιστοιχεί στο *process* του *client*. Το επόμενο βήμα, ασχέτως με τα περιεχόμενα της τούπλας που θα λάβει, θα είναι να περιμένει και πάλι να λάβει μια τούπλα, η οποία αποτελείται από ένα *atom*, το *pid* που έλαβε και στο προηγούμενο μήνυμα, και έναν αριθμό, στις δύο περιπτώσεις ακεραίου τύπου. Το επόμενο βήμα εξαρτάται από το *path* που έχουμε διανύσει ως τώρα:
 - Υπάρχει περίπτωση ο *client* να αιτείται πράξη αντιστροφής προσήμου, οπότε εδώ ο *server* θα αποστέλλει μια απάντηση στο *pid* του *client*, η οποία αποτελείται από μια τούπλα που περιέχει ένα *atom*, το *pid* του *server*, και το αποτέλεσμα της πράξης
 - Αν από την άλλη ο *client* αιτείται εκτέλεση πράξης εύρεσης τετραγωνικής ρίζας, τότε ο *server* και πάλι θα αποστέλλει μια τούπλα στον *client*, ίδιου τύπου με αυτήν της προηγούμενης περίπτωσης.
 - Σε περίπτωση που το αίτημα είναι για άθροιση αριθμών, τότε ο *server* θα περιμένει ένα ακόμη μήνυμα από τον *client*, με τον δεύτερο τελεσταίο της πράξης. Μόλις λάβει και αυτόν, θα επιστρέψει μια τούπλα αντίστοιχη με τις προηγούμενες δύο.

Και στις τρεις περιπτώσεις, η εκτέλεση του *server* συνεχίζεται με την αναδρομική κλήση της *server/0*.

- Η συνάρτηση *sum_client/2* υλοποιεί έναν *client* ο οποίος αιτείται μόνο πράξεις άθροισης από τον *server*. Στέλνει τρία μηνύματα στη σειρά, εκ των οποίων το πρώτο περιέχει ένα *atom*, *sum* και το *pid* του, και τα επόμενα δύο περιέχουν ένα *atom*, το *pid* του και από έναν ακεραίο, και περιμένει να λάβει πίσω μια τούπλα που περιέχει ένα *atom*, στην προκειμένη το *res*, το *pid* του *server*, και το αποτέλεσμα της πράξης που αιτήθηκε. Ακολούθως, καλείται αναδρομικά η *sum_client/2*.
- Η συνάρτηση *neg_client/2* υλοποιεί έναν *client* που αιτείται συνεχώς πράξεις αντιστροφής προσήμου από τον *server*. Αποστέλλει στην αρχή ένα μήνυμα που περιέχει το *atom* *neg*, ακολουθούμενο από το *pid* του *negation client*, και στη συνέχεια αποστέλλει μια τούπλα που περιέχει ένα *atom*, το *pid* του, και τον αριθμό που θα τελέσει χρέη τελεσταίου. Ακολούθως, περιμένει απάντηση με ένα *atom* (το *res*), το *pid* του *server*, και το αποτέλεσμα της πράξης, πρωτού καλέσει αναδρομικά τον εαυτό της.
- Η συνάρτηση *sqrt_client/2* συμπεριφέρεται με ακριβώς τον ίδιο τρόπο όπως η *neg_client/2*, με τη διαφορά ότι αποστέλλει ένα *atom* *sqrt* αντί για *neg* στο πρώτο της μήνυμα.

Στα σχήματα 6.1 έως 6.4 φαίνονται οι τύποι συνεδρίας των παραπάνω συναρτήσεων, σε μορφή γραφήματος.

Αναλύοντας την υλοποίηση αυτή με το σύστημά μας, παίρνουμε αρχικά τους συμπερασμένους τύπους διεργασιών των συναρτήσεων που αποτελούν το πρόγραμμα:

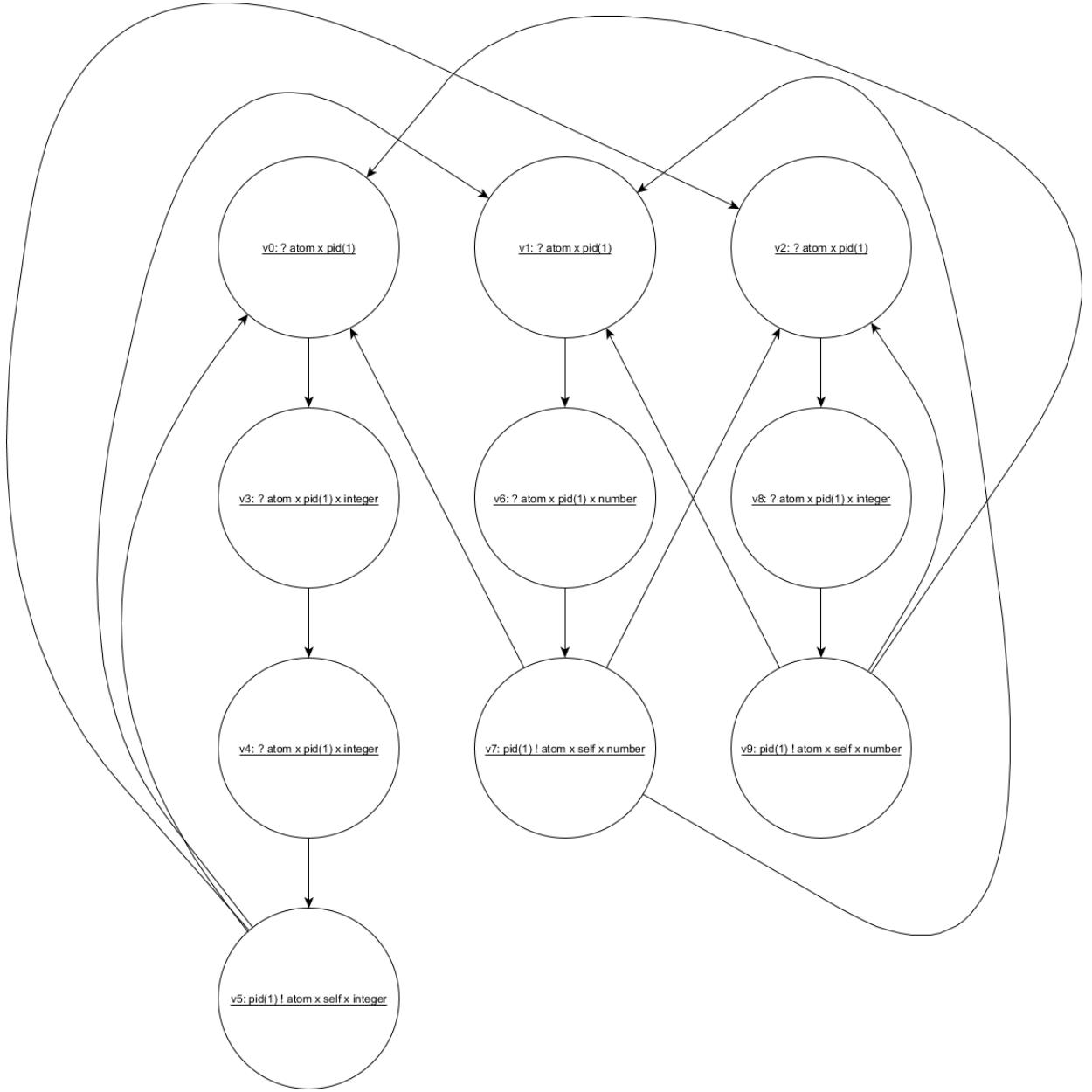
Listing 5.10: Inferred Session Types of Arithmetic Server

```

1 {server,0} : "u (4). Of<any; ?(atom x {atom, pid, (atom x atom)}).
2         ?(atom x {atom, pid, (atom x atom)} x number).
3         ?(atom x {atom, pid, (atom x atom)} x number).
4         pid(1) !(atom x pid x number).rec(4)., any;

```

Σχήμα 5.4: server/0



```

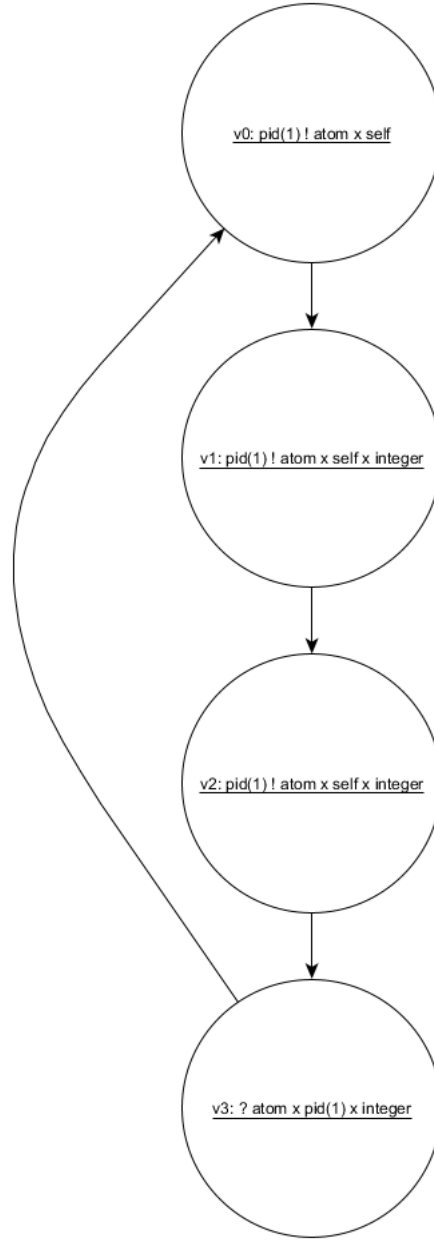
5      ?(atom x {atom, pid, (atom x atom)}).
6      ?(atom x {atom, pid, (atom x atom)} x number).
7      pid(1) !(atom x pid x number).rec(4).,any;
8      ?(atom x {atom, pid, (atom x atom)}).
9      ?(atom x {atom, pid, (atom x atom)} x number).
10     pid(1) !(atom x pid x float).rec(4).,any;
11     ?(atom x any).>."
12
13 {sum_client,2} : "u (3). pid(1) !(atom x pid).
14                 pid(1) !(atom x pid x integer).
15                 pid(1) !(atom x pid x integer).
16                 ?(atom x pid x any).rec(3)."

```

Listing 5.10: Inferred Session Types of Arithmetic Server

Καθώς δεν εμφανίστηκε κάποιο σφάλμα κατά τη σύγκριση των τύπων που το σύστημά μας συμπεράνε

Σχήμα 5.5: `sum_client/2`



για τις υλοποιηθείσες συναρτήσεις με τους τύπους που παρείχε ο χρήστης μέσω των `-sestype()` annotations, διαπιστώνουμε ότι οι συναρτήσεις πράγματι υλοποιούν το επιθυμητό πρωτόκολλο.

Ακολουθώς, προσπαθούμε να δούμε αν όντως μπορεί να διαδραματιστεί επιτυχής επικοινωνία ανάμεσα στα endpoints μας. Αρχικά, έχουμε τον server με τον client άθροισης, `sum_client/2`.

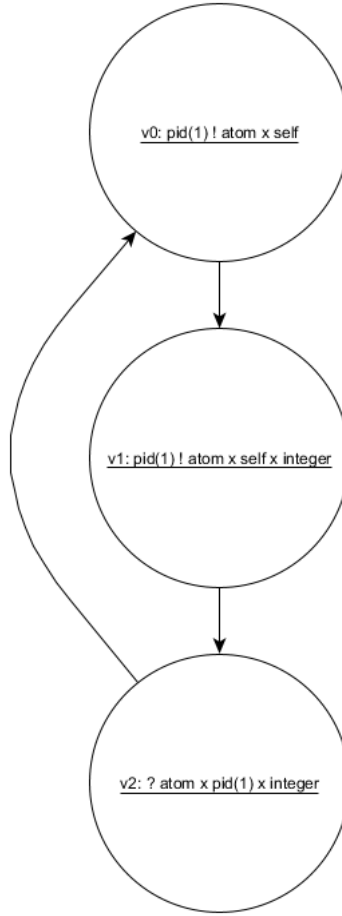
Listing 5.11: `server/0` and `sum_client/2`

```

1 {server,0} and {sum_client,2} are peers
2 The matching sequences are:
3 [{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
4  {send,{other_proc,1},{tuple,[atom,pid]}},
5  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
6  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
7  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
8  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
9  {send,{other_proc,1},{tuple,[atom,pid,number]}},
10 {recv,any,{tuple,[atom,pid,any]}]}]
```

Listing 5.11: `server/0` and `sum_client/2`

Σχήμα 5.6: *neg_client/2*



Όπως θα έπρεπε, το σύστημά μας βρίσκει ένα παράδειγμα επιτυχούς επικοινωνίας μεταξύ των δύο συναρτήσεων, παρουσιάζοντάς μας τα ζευγάρια των δικών πράξεων επικοινωνίας.

Τα πράγματα για τους *neg_client/2* και *sqr_client/2* είναι ελαφρώς διαφορετικά:

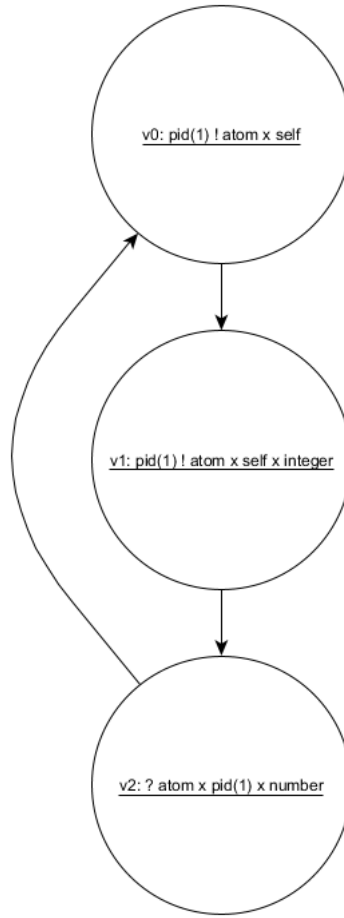
Listing 5.12: *neg_client/2* and *sqr_client/2*

```

1
2 {server,0} and {neg_client,2} are peers
3 The matching sequences are:
4 [{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
5  {send,{other_proc,1},{tuple,[atom,pid]}},
6  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
7  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
8  {send,{other_proc,1},{tuple,[atom,pid,float]}},
9  {recv,any,{tuple,[atom,pid,any]}},
10 [{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
11  {send,{other_proc,1},{tuple,[atom,pid]}},
12  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
13  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
14  {send,{other_proc,1},{tuple,[atom,pid,number]}},
15  {recv,any,{tuple,[atom,pid,any]}},
16
17 {server,0} and {sqr_client,2} are peers
18 The matching sequences are:
19 [{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
20  {send,{other_proc,1},{tuple,[atom,pid]}},
21  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
22  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
23  {send,{other_proc,1},{tuple,[atom,pid,float]}},
24  {recv,any,{tuple,[atom,pid,any]}},
25 [{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
26  {send,{other_proc,1},{tuple,[atom,pid]}},
27  {recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}],number}}},
28  {send,{other_proc,1},{tuple,[atom,pid,integer]}},
29  {send,{other_proc,1},{tuple,[atom,pid,number]}},

```

Σχήμα 5.7: sqrt_client/2



30 {recv , any , { tuple , [atom , pid , any] } }]

Listing 5.12: neg_client/2 and sqrt_client/2

Παρόλο που (σωστά) το σύστημά μας αποφαίνεται ότι τα ζευγάρια των δύο clients με τον server είναι πράγματι εταίροι επικοινωνίας, εσφαλμένα υπολογίζει δύο εναλλακτικά επιτυχή "σενάρια" επικοινωνίας για κάθε έναν από τους δύο clients. Αυτό οφείλεται στο γεγονός ότι κατά την ανάλυση λαμβάνουμε υπόψιν μας μόνο τους τύπους των στοιχείων δεδομένων, και όχι τις τιμές τους. Καθώς τα μονοπάτια επικοινωνίας για τις περιπτώσεις της αντιστροφής προσήμου και του υπολογισμού τετραγωνικής ρίζας είναι ίδια τόσο ως προς τα βήματα, όσο και στους τύπους δεδομένων των μηνυμάτων που ανταλλάσσονται ανάμεσα στους clients και τον server, το σύστημά μας τα θεωρεί έγκυρα, ενώ στην πραγματικότητα, μόνο το ένα από τα δύο είναι έγκυρο για κάθε client.

Μια πιθανή λύση γι' αυτό το πρόβλημα θα μπορούσε να είναι το να θεωρούσαμε το κάθε literal atom στον κώδικα των προγραμμάτων που αναλύουμε σαν ένα singleton type, κατα παρόμοιο τρόπο όπως αντιμετωπίζουμε τα pids.

Κεφάλαιο 6

Συμπεράσματα και μελλοντική έρευνα

Σε αυτήν την εργασία παρουσιάσαμε ένα στατικό σύστημα τύπων συνεδρίας για τη γλώσσα Erlang. Παρουσιάσαμε την αρχιτεκτονική του, και τις αρχές πίσω από την ανάλυση που πραγματοποιεί.

Κατά την ανάπτυξη και χρήση του εν λόγω συστήματος, διαπιστώσαμε ότι υπάρχει ένα σύνολο προσθηκών και αλλαγών που θα έπρεπε να γίνουν σε αυτό, προκειμένου να μπορεί να χρησιμοποιηθεί αποτελεσματικά για την ανάλυση υπάρχουσων βάσεων κώδικα, αλλά και ως ένα βοηθητικό εργαλείο αποσφαλμάτωσης κατά τη φάση της ανάπτυξης νέων εφαρμογών.

Συγκεκριμένα:

- Το παρόν σύστημα τύπων μας δεν είναι σε θέση να μοντελοποιήσει αμιγώς αναδρομικές συναρτήσεις.
- Τα session type annotations που μπορεί να εισάγει κανείς στα modules θα έπρεπε να παρέχονται με έναν τρόπο πιο φιλικό προς τον χρήστη, όπως, για παράδειγμα, σαν μια συμβολοσειρά που μοιάζει περισσότερο με την περιγραφή των session types με τον τρόπο που αυτή συναντάται στη βιβλιογραφία.
- Ο μηχανισμός επαλήθευσης των peer session types επιδέχεται σοβαρών βελτιστοποιήσεων, κυρίως σε περιπτώσεις όπου αντιμετωπίζουμε αναδρομικούς τύπους.
- Ο μηχανισμός επαλήθευσης των peer session types μπορεί να επεκταθεί ώστε να επαληθεύει τη συμμόρφωση σε πρωτόκολλα επικοινωνίας σε περίπτωση multiparty sessions, δηλαδή σε περιπτώσεις όπου οι συμμετέχουσες διεργασίες ξεπερνούν τις δύο σε αριθμό. Ο υφιστάμενος μηχανισμός στην παρούσα του μορφή δε μπορεί να υποστηρίξει αναζητήσεις σε ένα τέτοιο search space. Η εισαγωγή τεχνικών που έχουν χρησιμοποιηθεί με επιτυχία στην περίπτωση του stateless model checking [], καθώς επίσης και η εφαρμογή μεθόδων επέκτασης κατευθυνόμενων κυκλικών γραφημάτων μπορεί να αποδειχθούν ιδιαίτερα αποτελεσματικές για την επίτευξη αυτού του στόχου.
- Ο μηχανισμός συμπερασμού τύπων δεδομένων απέχει από το να είναι ο βέλτιστος δυνατός, όμως δεν είναι ξεκάθαρο το πώς θα μπορούσε να βελτιωθεί, δεδομένων των περιορισμών των γλωσσών προγραμματισμού με δυναμικά συστήματα τύπων.
- Η επέκταση του συστήματος ώστε να είναι σε θέση να διαχειριστεί και τύπους δεδομένων ορισμένους από τον χρήστη.

Βιβλιογραφία

Παράρτημα Α

Ευρετήριο συμβολισμών

$A \rightarrow B$: συνάρτηση από το πεδίο A στο πεδίο B .

Παράρτημα Β

Ευρετήριο γλωσσών

Haskell : η γλώσσα της ζωής μου.

Παράρτημα C

Ευρετήριο αριθμών

42 : life, the universe and everything.