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 Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

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 Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21η Οκτωβρίου 2019.

.....	.....	.....
Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.	Γεώργιος Στάμου Αν. Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2019

Αχιλλέας Μπενετόπουλος

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

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, λόγο 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

Αρχικά θα ήθελα να ευχαριστήσω τους δύο συνεπιβλέπωντες καθηγητές της παρούσας εργασίας, Νίκο Παπασπύρου και Κωστή Σαγώνα, τόσο για τη βοήθεια και υποστήριξή τους κατά την εκπόνησή της, όσο και γιατί, με τα μαθήματα που δίδασκαν αλλά και με το ενδιαφέρον που επιδείκνυαν, τροφοδότησαν και διαμόρφωσαν το δικό μου ενδιαφέρον στον τομέα.

Ακολούθως, θα ήθελα να ευχαριστήσω τον Σταύρο Αρώνη, για τη βοήθεια και τις συμβουλές του στο κομμάτι της τροποποίησης του Dialyzer προκειμένου να εξυπηρετήσει τους σκοπούς της παρούσας εργασίας.

Επιπλέον, θα ήθελα να ευχαριστήσω τους συμφοιτητές και φίλους μου, καθώς η συναναστροφή μαζί τους μέσα στα χρόνια με διαμόρφωσε και με έκανε τον άνθρωπο που είμαι σήμερα.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου, για την αστείρευτη υποστήριξη και εμπιστοσύνη τους. Ξέρω ότι η εκτίμησή μου δεν είναι πάντα εμφανής, αλλά είναι πάντα εκεί.

Αχιλλέας Μπενετόπουλος,

Αθήνα, 21η Οκτωβρίου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-8-19, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2019.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος σχημάτων

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 ανταλλάσονται, πότε κατα τη διάρκεια της επικοινωνίας εμφανίζονται αυτά τα μηνύματα, και ανάμεσα σε ποια κομμάτια του συστήματος αυτά ανταλλάσονται. Συνεπώς, μπορούμε να εκφράσουμε την ορθότητα της επικοινωνίας μεταξύ των επιμέρους συστημάτων ως τη *συμμόρφωση* σε αυτά τα πρωτόκολλα από τους συμμετέχοντες στην επικοινωνία. Οι τύποι συνεδριών μας επιτρέπουν να εκφράσουμε αυτά τα πρωτόκολλα επικοινωνίας, καθως 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 [Neyk13, Fowl16]. Στη βάση τους, όλες χτίζουν πάνω στη γλώσσα Scribble, η οποία είναι μια domain-specific language (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 της υλοποίησης της γλώσσας) 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 κώδικά.

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 6 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

Η γλώσσα Erlang και το σύστημα OTP

2.1 Η γλώσσα προγραμματισμού Erlang

Η Erlang είναι μια συναρτησιακή γλώσσα προγραμματισμού, η οποία σχεδιάσθηκε 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 διαδωθεί και σε βιομηχανίες που μοιράζονται τις ίδιες λειτουργικές απαιτήσεις, όπως η τραπεζική, οι επικοινωνίες πραγματικού χρόνου (WhatsApp), κα.

Το Open Telecom Platform (OTP) είναι μια συλλογή εργαλείων, βιβλιοθηκών, middleware και αρχών σχεδιασμού για προγράμμτα 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 Οι τύποι δεδομένων της Erlang

Η Erlang είναι μια γλώσσα ασφαλής ως προς τους τύπους (type safe), αλλά με δυναμικό συστήμα 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 κατα 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: τύπος διακριτικών διεργασιών
- Port: τύπος διακριτικών μιας θύρας Erlang
- Reference: τύπος όρων που είναι μοναδικοί στο σύστημα χρόνου εκτέλεσης
- Atom: τύπος κυριολεκτικών όρων, σταθερών
- Bitstring: τύπος μια περιοχή μνήμης, τα περιεχόμενα της οποίας δεν έχουν κάποιον συγκεκριμένο τύπο
- Float: τύπος αριθμών κινητής υποδιαστολής
- Fun: τύπος συναρτήσεων
- Integer: τύπος ακεραίων αριθμών
- List: τύπος μιας αθροιστικής δομής δεδομένων, με μεταβλητό πλήθος στοιχείων
- Map: τύπος μιας αθροιστικής δομής δεδομένων, με μεταβλητό πλήθος αντιστοιχίσεων κλειδιού-τιμής
- Tuple: τύπος μιας αθροιστικής δομής δεδομένων, με σταθερό πλήθος στοιχείων

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

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 εργαλείο 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 $foo(\text{"}Hello\text{"})$ 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 συμπέραινε ο Dialyzer για τη συνάρτηση $foo/1$ είναι $Number \rightarrow Number$.

```
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 πηγαίου 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 κατα τη διάρκεια της εκτέλεσης του dataflow analysis loop του.

2.5 Ταυτοχρονισμός σε Erlang

Ένα ισχυρό χαρακτηριστικό της Erlang είναι η εγγενής 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.

Ο μηχανισμός που παρέχεται στον προγραμματιστή για την υλοποίηση της επικοινωνίας και της ανταλλαγής πληροφοριών μεταξύ των διεργασιών του προγράμματός του 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.

```
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 τριάδα τιμών (tuple μεγέθους 3) που αποτελείται από το atom num, ένα pid κι έναν αριθμό.
- Μια τριάδα τιμών που αποτελείται από το atom $atom$, ένα pid κι ένα ακόμη στοιχείο.
- Τον ακέραιο 42.

Αν μια διεργασία που επικοινωνεί με αυτήν της αποστείλει ένα μήνυμα της μορφής $\{num, 42\}$, τότε η $foo/1$ δε θα το "καταναλώσει" ποτέ. Αυτό δεν αποτελεί πρόβλημα από μόνο του, αλλά η αποστολή πολλών μηνυμάτων σε μια διεργασία, τα οποία αυτή δεν καταναλώνει, μπορεί να αποτελέσει πρόβλημα δυνητικά, καθώς ενδέχεται να οδηγήσει σε εξάντληση της μνήμης του συστήματος χρόνου εκτέλεσης της Erlang, και κατά συνέπεια στον τερματισμό της εκτέλεσης ενός κόμβου του συστήματος. Αντίστοιχα, αν καμία διεργασία δεν αποστείλει 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 τουλάχιστον ενός εκ των δύο μερών της επικοινωνίας στο αναμενόμενο πρωτόκολλο.

Κεφάλαιο 3

Συστήματα τύπων και επιδράσεων

Πολλές γλώσσες προγραμματισμού ενσωματώνουν ένα στατικό σύστημα τύπων, προκειμένου να εγγυηθούν ότι τελεστές εφαρμόζονται μόνο σε τελεσταίους της κατάλληλης μορφής, δηλαδή του κατάλληλου είδους [Pier02, Pier04]. 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, με την οποία ένας προγραμματιστής θα μπορούσε να δώσει... νόημα σε αυτήν την πράξη).

```
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 επισημειώσεις τύπων (type annotations) τις οποίες παρείχε ο χρήστης. Αυτές 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 προσδοκόμενους τύπους στο σημείο που αυτές 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 Dammas-Milner) [Dama82], με την γλώσσα ML να είναι η πρώτη για την οποία υλοποιήθηκε 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 αντιστοίχηση αυτή εκφράζεται με μια σχέση του τύπου $\Gamma \vdash e : \tau$, 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 [Solb95]. Στόχος αυτής της επέκτασης είναι το να καταστεί εφικτό ένα σύνολο στατικών αναλύσεων, όπως:

- Ανάλυση χρόνου δεσίματος (bind-time analysis), κατα την οποία προσπαθούμε να πραγματοποιήσουμε μια διάκριση ανάμεσα σε αυτά τα δεδομένα των οποίων η τιμή είναι γνωστή κατα το χρόνο μεταγλώττισης (στατικά), και σε αυτά τα οποία αποκτούν τιμή κατα την εκτέλεση του προγράμματος (δυναμικά). Αυτού του τύπου η ανάλυση αποτελεί τη βάση της μερικής αποτίμησης που πραγματοποιούν μεταγλωττιστές
- Ανάλυση χρήσεων (usage analysis), 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.
- Συλλογή σκουπιδιών (garbage collection): 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 (control flow analysis), κατά την οποία προσπαθούμε, συνήθως σε γλώσσες με συναρτήσεις υψηλότερης τάξης, να συμπεράνουμε ποιες συναρτήσεις είναι πιθανοί ”αποδέκτες” μιας κλήσης σε ένα πρόγραμμα, δηλαδή ποιες συναρτήσεις είναι πιθανό να αποκτήσουν τον ελέγχο 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 [Niel99, Heng94].

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.

Ο αναγνώστης μπορεί να σκέφτεται τις επιδράσεις ενός προγράμματος σαν τις παρενέργειες που μπορεί να προκαλέσει στο περιβάλλον του σαν αποτέλεσμα της εκτέλεσής του. Για παράδειγμα, έχουν αναπτυχθεί συστήματα τύπων κι επιδράσεων που αντιμετωπίζουν τις προσβάσεις (εγγραφή και ανάγνωση) στη μνήμη από προγράμματα ως επιδράσεις των προγραμμάτων, τις οποίες και κωδικοποιούν [Pier04].

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, αλλα και το περιεχόμενο και την κατεύθυνση των μηνυμάτων που ανταλλάσουν οι πράκτορες της επικοινωνίας.

Οι τύποι συνεδρίας είναι περιγραφές πρωτοκόλλων επικοινωνίας, που διαγράφουν τις επιτρεπόμενες ακολουθίες μηνυμάτων, μαζί με τους τύπους των μηνυμάτων που ανταλλάσσονται ανάμεσα στους συμμετέχοντες. Μας παρέχουν εγγυήσεις σχετικά με χαρακτηριστικά της επικοινωνίας, όπως την ποιότητα και την ασφάλειά τους. Ένα καλώς ορισμένο ως προς τους τύπους πρόγραμμα αποκλείεται να στείλει ή να λάβει μήνυμα του λάθος τύπου.

Οι τύποι συνεδρίας μοντελοποιούν την επικοινωνία μεταξύ δύο ή περισσότερων οντοτήτων. Ορίζονται ως μια σειρά από λειτουργίες εισόδου/εξόδου, οι οποίες περιγράφουν τους τύπους των μηνυμάτων που ανταλλάσσονται. Στη συγκεκριμένη εργασία, απασχολούμαστε μόνο με συνεδριές μεταξύ δύο συμμετεχόντων.

Στο [Take94] 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 αιτήματος 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. Άξιοσημείωτο 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).

Ακολούθως στο [Hond98] έχουμε μια επέκταση της δουλειάς του [Take94], στην οποία πλέον επιτρέπεται η περιγραφή αναδρομικών τύπων συνεδρίας, πράγμα που επεκτείνει την πρακτική εφαρμογή της θεωρίας των τύπων συνεδριών.

Μέχρι πρόσφατα, η έρευνα επικεντρωνόταν σε τύπους συνεδριών δύο συμμετεχόντων. Αυτές ήταν σε θέση να περιγράφουν επικοινωνία με πολλαπλούς συμμετέχοντες, όμως τα επιμέρους πρωτόκολλα έπρεπε να αφορούν δύο συγκεκριμένους συμμετέχοντες και να είναι ανεξάρτητα μεταξύ τους. Συνέπεια αυτού είναι ότι τα συστήματα αυτά δε μπορούν να "περιορίσουν" ή να διασφαλίσουν τη σειρά με την οποία ανταλλάσονται μηνύματα σε δύο διαφορετικά πρωτόκολλα.

Στο [Hond16], οι Honda et al αναπτύσσουν ένα σύστημα για να κάνουν reason για συνεδρίες πολλαπλών συμμετεχόντων.

Σε αυτήν τη δημοσίευση 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 της επικοινωνίας. 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 χορεογραφία), παράγει την περιγραφή που αντιστοιχεί στα διάφορα endpoints αυτής.

4.1 Τύποι Συνεδρίας και Γλώσσες Προγραμματισμού

Σε πρακτικό επίπεδο, έχει υπάρξει μια πληθώρα εφαρμογών της θεωρίας των τύπων συνεδριών, και με διαφορετικές φιλοσοφίες.

Αρχικά, υπάρχει ο διαχωρισμός ανάμεσα σε στατικά, δυναμικά, και υβριδικά συστήματα τύπων συνεδριών.

- Ένα στατικό σύστημα τύπων συνεδριών πραγματοποιεί όλους τους ελέγχους του κατα το χρόνο μεταγλώττισης ενός προγράμματος. Οποιαδήποτε παράβαση του πρωτοκόλλου θα αναφερθεί πριν την εκτέλεση του προγράμματος.

- Ένα δυναμικό σύστημα τύπων συνεδριών παρακολουθεί την επικοινωνία ανάμεσα στα επιμέρους τμήματα ενός προγράμματος. Οι τύποι συνεδρίας μετασχηματίζονται σε μηχανές πεπερασμένων καταστάσεων, και τα μηνύματα που ανταλλάσονται επαληθεύονται με βάση αυτές.

- Ένα υβριδικό σύστημα τύπων συνεδριών επαληθεύει ορισμένες ιδιότητες της επικοινωνίας (όπως τη σειρά ανταλλαγής μηνυμάτων) στατικά, κατά τη μεταγλώττιση, και άλλες (όπως τη γραμμικότητα της επικοινωνίας, ή τους τύπους των ανταλλασσόμενων μηνυμάτων) κατά το χρόνο εκτέλεσης του προγράμματος.

Επιπλέον, παρατηρούμε και διαφορές ως προς τον τρόπο με τον οποίο τα συστήματα τύπων συνεδριών παρέχονται στους προγραμματιστές.

- Υπάρχουν γλώσσες, όπως η MOOL [] και η SePi [], που παρέχουν τους τύπους συνεδριών ως built-in primitives.

- Σε άλλες υλοποιήσεις, όπως στην Multiparty Session C [] ή στη Session Java [] έχουμε την υλοποίηση του συστήματος συνεδριών μέσω μιας βιβλιοθήκης χρόνου εκτέλεσης, με την οποία μπορεί ο προγραμματιστής να συνδέσει το πρόγραμμά του.
- Τέλος, υπάρχουν εξωτερικά εργαλεία τα οποία αναλύουν τον πηγαίο κώδικα ενός προγράμματος προκειμένουν να πραγματοποιήσουν την ανάλυση σε επίπεδο τύπων συνεδρίας. Ένα παράδειγμα ενός τέτοιου εργαλείου περιγράφεται στο επόμενο κεφάλαιο της παρούσας εργασίας.

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. Παρουσιάζουμε ένα σύστημα τύπων συνεδρίας εκφρασμένο σαν ένα σύστημα τύπων κι επιδράσεων, εμπνευσμένο από σχετική δουλειά στον τομέα [DOrc16].

5.1 Το σύστημα τύπων

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

$$
\begin{array}{lcl}
sestype & ::= & monosestype \\
monosestype & ::= & (datatype_1, \ldots, datatype_n) \rightarrow \\
 & & datatype;\ \phi \\
\phi & ::= & \varnothing \\
 & | & sesaction.\ \phi \\
 & | & \mu\,\sigma\,.\,\phi \\
 & | & (\mu\,\sigma\,.\,\phi).\phi \\
sesaction & ::= & datatype; !^P datatype \\
 & | & datatype_1; ?\ datatype_2 \\
 & | & \& < datatype_1; \phi_1, \ldots, datatype_n; \phi_n > \\
 & | & \oplus < datatype_1; \phi_1, \ldots, datatype_n; \phi_n > \\
 & | & \sigma \\
datatype & ::= & pid[process] \\
 & | & Type \\
 & | & sestype \\
process & ::= & self \\
 & | & P
\end{array}
$$

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. Το περιβάλλον αυτό ορίζεται ως

$$
\begin{array}{lcl}
PE & ::= & \varnothing \\
 & | & PE, P
\end{array}
$$

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

$$\begin{aligned} SE \quad ::= \quad & \varnothing \\ | \quad & SE, \sigma \end{aligned}$$

Ορίζουμε ως $\Gamma = \{DE, PE, SE\}$ την τριάδα που μας δίνει τα type schemes.
Οι κανόνες συμπερασμού τύπων ακολουθούν:

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

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

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

$$\frac{\Gamma \vdash S_i : datatype_i; \phi_i}{\Gamma \vdash case \, e \, of \, p_i \rightarrow S_i{}^{i \in \{1..N\}} \; end : datatype_j^{j \in \{1..N\}}; \oplus < \phi_1, \ldots, \phi_N >} \begin{array}{l} Pref(\phi_i^{i \in \{1..N\}}) \; = \; !^P datatype \\ \qquad\qquad\qquad\qquad\quad | \; nothing \end{array} (\textit{choice\_case})$$

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

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 σαν ένα απλό term (με τη διαφορά ότι επιτρέπεται να περιέχει μεταβλητές που δεν έχουν συσχετισθεί ακόμη με κάποια τιμή), 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 τυχών 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 οποίο συμπεραίνουμε τους τύπους και τις επιδράσεις συναρτήσεων.

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

5.2 Υλοποίηση

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.

```
run() ->
    ProcServer = spawn(fun() -> server() end),
    Foo = spawn(fun() -> foo() end),
    Biz = spawn(fun() -> biz() end),
    spawn(fun() -> baz(Biz) end),
    spawn(fun() -> bar(Foo) end),
    spawn(fun() -> simple_client(ProcServer) end),
    spawn(fun() -> recursive_client(ProcServer, rand:uniform(100)) end).

server() ->
    receive
        {num_1, Proc, X} ->
            receive
                {num_2, Proc, Y} ->
                    Proc ! {num_sum, self(), X + Y + 42},
                    server()
            end;
        {foo, Proc, X} -> server()
    end.

simple_client(Server) ->
    Server ! {num_1, self(), 17},
    Server ! {num_2, self(), rand:uniform(100)},
    receive
        {num_sum, Server, X} -> X
    end.

recursive_client(Server, 0) -> ok;
recursive_client(Server, N) ->
    Server ! {foo, self(), N},
    recursive_client(Server, N - 1).

foo() ->
    receive
        {foo, P, 42} ->
            receive
                {foo, P, 17} ->
                    P ! {foo, self(), 42 + 17},
                    foo()
            end
    end.

bar(Foo) ->
    Foo ! {foo, self(), 42},
    Foo ! {foo, self(), 17},
    receive
        {foo, Foo, X} ->
            Foo ! {foo, self(), 42},
            Foo ! {foo, self(), 17},
            receive
                {foo, Foo, X} -> X + 42
            end
    end.

```

```
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

5.2.1 Τμήματα

Το συστημά μας αποτελείται από τα εξής επιμέρους τμήματα:

- Το τμήμα του συμπερασμού των τύπων συνεδριών του προγράμματος που δεχόμαστε ως είσοδο.
- Το τμήμα 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 τμήμα 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 τμήμα. 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 συμπερασμό τύπων του 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 αντιστοιχήσεις μεταβλητών με τύπους.

Για μια ανάλυση του τρόπου με τον οποίο δουλεύει ο Dialyzer, ο αναγνώστης παραπέμπεται στο [TLin06].

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 της.

Στο τέλος της ανάλυσης των modules του προγράμματος, πραγματοποιούμε ένα iteration πάνω σε όλες τις ορισμένες συναρτήσεις, και επεκτείνουμε 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.

Λόγω του τρόπου με τον οποίο διαχειριζόμαστε τις επιδράσεις συναρτήσεων που δεν έχουμε αναλύσει ακόμη, το σύστημα συμπερασμού τύπων μας δε μπορεί να διαχειριστεί 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:

```
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)."
5
6  {recursive_client,2} : "u (1). pid(1) !(atom x pid x integer).rec(1)."
7
8  {foo,0} : "u (5). ?(atom x {atom, pid, (atom x atom)} x integer). ?(atom x {atom, pid, (atom x atom)} x
       ↪ integer).
9             pid(1) !(atom x pid x integer).rec(5)."
10
11 {bar,1} : "pid(1) !(atom x pid x integer). pid(1) !(atom x pid x integer). ?(atom x pid x number).
12            pid(1) !(atom x pid x integer). pid(1) !(atom x pid x integer). ?(atom x any x number)."
13
14 {baz,1} : "u (2). pid(1) !(atom x pid x integer). pid(1) !(atom x pid x integer).rec(2)."
15
16 {biz,0} : "u (3). ?(atom x any x any). ?(atom x any x any). ?(atom x any x any).rec(3)."
```

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 κατα 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.

sestype Annotations			
Action Tag	First Argument	Second Argument	Explanation
?f_rec_tag(T, E)	T: fix reference στην αναδρομική συν/ση	E: session type της συν/σης	Περιγράφει τον τύπο συνεδρίας μιας αναδρομικής συνάρτησης
?sa_recv_tag(DT1, DT2)	$DT1$: τύπος δεδομένων του σώματος του receive clause	$DT2$: τύπος δεδομένων που θα κάνει match με το receive clause	Session Type ενός receive clause
?sa_send_tag(P, DT)	P: pid reference του παραλήπτη	DT: τύπος δεδομένων του μηνύματος που αποστέλεται	Session Type ενός send clause
?sa_offer_tag(L)	L: λίστα με session types των receive clauses	-	Session Type ενός multi-branch receive statement
?sa_choice_tag(L)	L: λίστα με session types των send clauses	-	Session Type ενός branching execution όπου το πρώτο session action κάθε branch είναι ένα send clause
?sa_rec_tag(T)	T: reference σε μια αναδρομική συν/ση	-	Session Type αναδρομικής κλήσης συν/σης

```
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 συμπερασμένου τύπου δεδομένων.

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 συμπερασμενοι 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 πρωτοκολλου επικοινωνίας.

Για παράδειγμα, στο πρώτο clause της $simple\_send/3$, αν ο χρήστης είχε δώσει

```
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: Type mismatch

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

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 $e = (u, v)$ σημαίνει ότι το action που περιγράφεται από τη v ακολουθεί αυτό που περιγράφεται από τη u στον τύπο συνεδρίας της συνάρτησης.

```
1   input: graph G1, graph G2
2   output: list of successful routes
3   begin
4     S = { {root(G1), root(G2)}}
5     res = []
6     while S not empty:
7       Pair = S.pop()
8       if complementary(Pair):
9         if next(Pair[G1]) is empty:
10          if next(Pair[G2]) is empty:
11            res.push(current route)
12          else:
13            if valid(recursive_criteria(Pair[G2])):
14              res.push(current route)
15        else:
16          if next(Pair[G2]) is empty:
17            if valid(recursive_criteria(Pair[G1])):
```

```
18                res.push(current route)
19            else:
20              if valid(recursive_criteria(Pair[G1])) and
21                 valid(recursive_criteria(Pair[G2])):
22                res.push(current route)
23              else:
24                foreach v in recursive_criteria(Pair[G1]):
25                  foreach u in recursive_criteria(Pair[G2]):
26                    S.push({u, v})
27
28      return res
29    end
```

Algorithm: 1. Session Type Matching

Το σύνολο αναζήτησης S είναι μια ουρά από ζευγάρια κορυφών, των οποίων τις πράξεις 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 στις παρούσες πράξεις.

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 βγάζει 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:

```
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: 2. 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 ώς το επίπεδο εμφώλευσης της αναδρομικής ακολουθίας, καθώς σε ένα session type μπορούμε να έχουμε αναδρομικές υποακολουθίες.

Για παράδειγμα, στον τύπο συνεδρίας

$$\mu\sigma.pid(1)\ !\ atom\ x\ integer.$$
$$pid(1)\ !\ atom\ x\ self\ x\ integer.$$
$$(\mu\phi.?\ atom\ x\ integer.\phi).$$
$$\sigma$$

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 ελεγξουμε στο μέλλον.

Ακολουθούν μερικά peer annotations για το πρόγραμμα 5.1.

```
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

Το όρισμα του $-peers()$ annotation είναι μια λίστα, στοιχεία της οποίας είναι tuples μεγέθους 2, τα οποία αναπαριστούν τις συναρτήσεις των οποίων τους τύπους συνεδρίας θέλουμε να συγκρίνουμε. Συγκεκριμένα:

- Το πρώτο στοιχείο του tuple είναι επίσης ένα tuple μεγέθους 2, που προσδιορίζει την συνάρτηση που μας ενδιαφέρει, σε μορφή $\{function\_name, arity\}$.
- Το δεύτερο στοιχείο είναι μια λίστα, που επιτρέπει στον προγραμματιστή να παραμετροποιήσει τη σύγκριση. Ο προγραμματιστής μπορεί να δώσει συνδυασμό των επιλογών
  - *total*, με την οποία ο προγραμματιστής μας λέει ότι για να είναι επιτυχές το ταίριασμα, πρέπει ο τύπος συνεδρίας της συνάρτησης να καταναλωθεί πλήρως.
  - *user_type*, με την οποία ο προγραμματιστής μας λέει να μη χρησιμοποιήσουμε τον τύπο συνεδρίας που συμπεράναμε, αλλά τον τύπο που μας παρείχε ο ίδιος μέσω του *-sestype()* annotation.

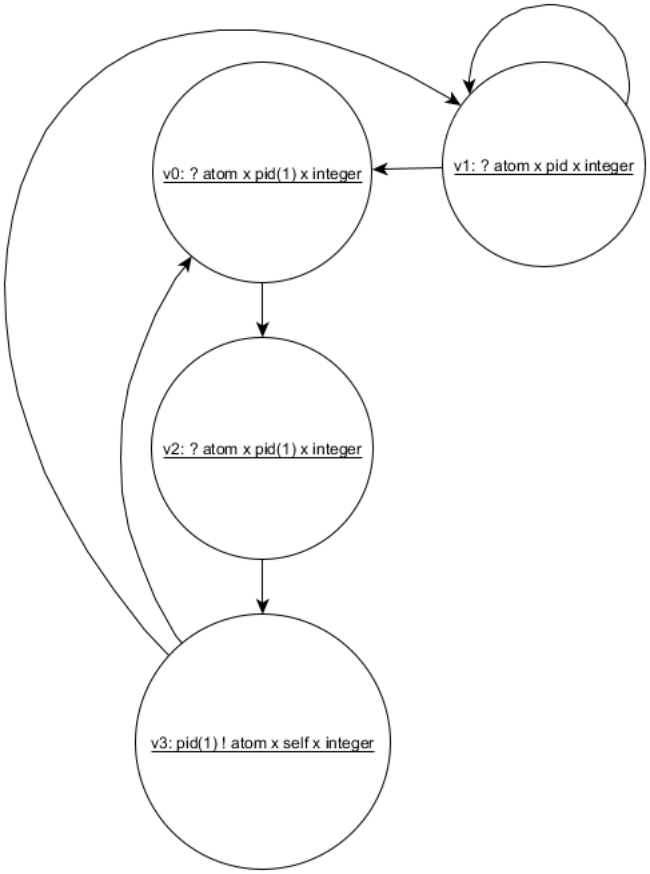
Ας αναλύσουμε τα δύο πρώτα ζευγάρια annotations του listing 5.7. Τα κατευθυνόμενα γραφήματα των τύπων συνεδρίας των συναρτήσεων $server/0$, $simple\_client/1$ και $recursive\_client/2$ φαίνονται στα σχήματα 5.1 έως 5.3.

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:

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

Σχήμα 5.1: server/0



Θεωρούμε το πρώτο εκ των δύο ζευγών. Ο έλεγχος της γραμμής 8 του αλγορίθμου 1 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 τελος του συνόλου αναζήτησης, με αποτέλεσμα να έχουμε ως νέο σύνολο αναζήτησης το

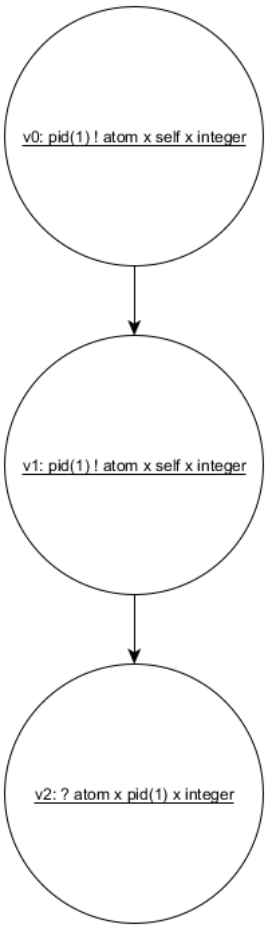
$$S = \{\{server/0[v_1], simple\_client/1[v_0]\}, \{server/0[v_2], simple\_client/1[v_1]\}\}$$

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 άδειο, οπότε καταλήγουμε να προσθέτουμε το καρτεσιανό γινόμενο των συνόλων αυτών στο σύνολο αναζήτησης, το οποίο 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 1 μέχρι να εξαντλήσουμε το σύνολο αναζήτησης, βρίσκουμε ότι πράγματι οι συναρτήσεις $server/0$ και $simple\_client/1$ αποτελούν communication

Σχήμα 5.2: simple_client/1



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

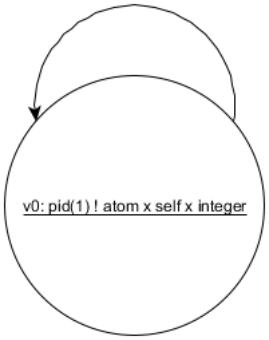
$$\begin{aligned} S = \{&\{server/0[v_2], recursive\_client/2[v_0]\}, \\ &\{server/0[v_1], recursive\_client/2[v_0]\}, \\ &\{server/0[v_0], recursive\_client/2[v_0]\}\} \end{aligned} \tag{5.3}$$

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.

Σχήμα 5.3: recursive_client/2



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 υλοποίσης: 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, φαίνεται στο listing 5.8.

```
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},
```

```
                    server()
            end;
        {sqrt, Pid} ->
            receive
                {sqrt_o, Pid, Opand} ->
                    Pid ! {res, self(), math:sqrt(Opand)},
                    server()
            end;
        {quit, _} ->
            exit(normal)
    end.

-spec sum_client(pid(), non_neg_integer()) -> ok.
sum_client(_Server, 0) -> ok;
sum_client(Server, N) ->
    Op1 = rand:uniform(1000),
    Op2 = rand:uniform(1000),
    Server ! {sum, self()},
    Server ! {sum_o1, self(), Op1},
    Server ! {sum_o2, self(), Op2},
    receive
        {res, Server, _Res} ->
            io:format("sum_client: Sent ~p and ~p, received ~p\n", [Op1, Op2, _Res]),
            sum_client(Server, N - 1)
    end.

-spec neg_client(pid(), non_neg_integer()) -> ok.
neg_client(_Server, 0) -> ok;
neg_client(Server, N) ->
    Op = rand:uniform(1000),
    Server ! {neg, self()},
    Server ! {neg_o, self(), Op},
    receive
        {res, Server, Res} ->
            io:format("neg_client: Sent ~p, received ~p\n", [Op, Res]),
            neg_client(Server, N - 1)
    end.

sqrt_client(_Server, 0) -> ok;
sqrt_client(Server, N) ->
    Op = rand:uniform(1000),
    Server ! {sqrt, self()},
    Server ! {sqrt_o, self(), Op},
    receive
        {res, Server, _Res} ->
            io:format("sqrt_client: Sent ~p, received ~p\n", [Op, _Res]),
            sqrt_client(Server, N - 1)
    end.
```

Listing 5.8: Arithmetic Server Implementation

Στο listing 5.9 φαίνονται τα session type annotations που περιγράφουν το πρωτόκολλο που πρέπει να ακολουθείται από την υλοποίηση.

```
-sestype({{server, 0},
        ?f_rec_tag(1, [?sa_offer_tag([{?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
                                                      ?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                      ?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                      ?sa_send_tag(1,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                                   ?sa_rec_tag(1)]},
                                    {?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
                                                      ?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                      ?sa_send_tag(1,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                                   ?sa_rec_tag(1)]},
                                    {?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
                                                      ?sa_recv_tag(?dt_atom_tag,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
                                                      ?sa_send_tag(1,
                                                                   ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_number_tag])),
                                                                   ?sa_rec_tag(1)]},
                                    {?dt_atom_tag, [?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([]))]}])])}).

```

```
27 -sestype({{sum_client, 2},
28          ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
29                         ?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
30                         ?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
31                         ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
32                         ?sa_rec_tag(1)])}).
33
34 -sestype({{neg_client, 2},
35          ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
36          '              ?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
37                         ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
38                         ?sa_rec_tag(1)])}).
39
40 -sestype({{sqrt_client, 2},
41          ?f_rec_tag(1, [?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag])),
42                         ?sa_send_tag(1, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_integer_tag])),
43                         ?sa_recv_tag(?dt_atom_tag, ?dt_tuple_tag([?dt_atom_tag, ?dt_pid_tag, ?dt_number_tag])),
44                         ?sa_rec_tag(1)])}).
```

Listing 5.9: Arithmetic Server Implementation

Τα contracts αυτά μας λένε το εξής:

- Η συνάρτηση $server/0$ αρχικά περιμένει να λάβει μια πλειάδα, 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 πλειάδας που θα λάβει, θα είναι να περιμένει και πάλι να λάβει μια πλειάδα, 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 αιτείτα πράξη αντιστροφής προσήμου, οπότε εδώ ο server θα αποστείλει μια απάντηση στο pid του client, η οποία αποτελείται από μια τουπλα 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 αποστείλει μια πλειάδα 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 απο τον client, με τον δεύτερο τελεσταίο της πράξης. Μόλις λάβει και αυτόν, θα επιστρέψει μια πλειάδα αντίστοιχη με τις προηγούμενες δύο.

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

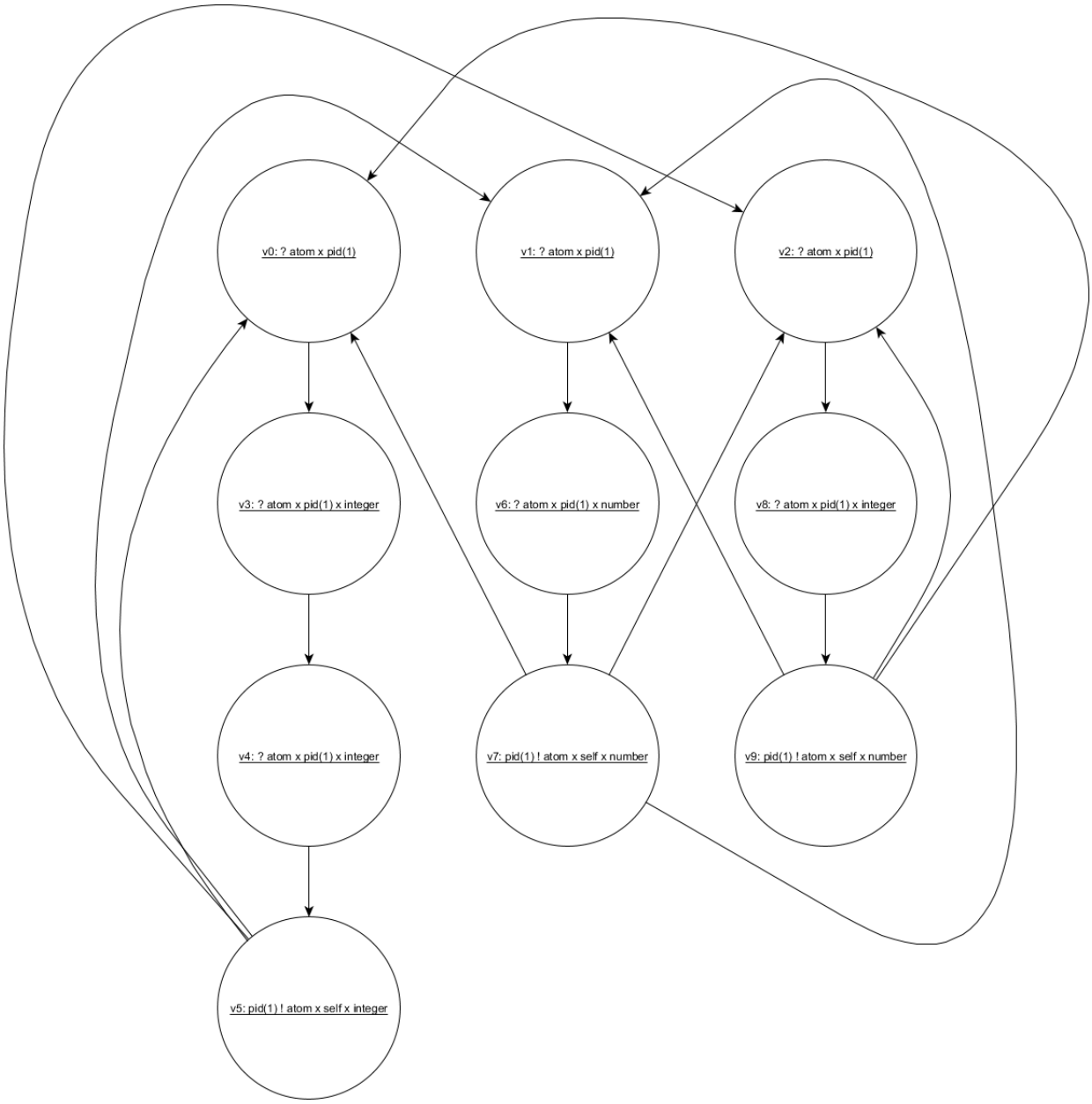
- Η συνάρτηση $sum\_client/2$ υλοποιεί έναν client ο οποίος αιτείται μόνο πράξεις άθροισης από τον server. Στέλνει τρία μηνύματα στη σειρά, εκ των οποίων το πρώτο περιέχει ένα $atomsum$ και το pid του, και τα επόμενα δύο περιέχουν ένα $atom$, το pid του και από έναν ακέραιο, και περιμένει να λάβει πίσω μια πλειάδα 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. Αποστέλει στην αρχή ένα μήνυμα που περιέχει το $atomneg$, ακολουθούμενο από το pid του negation client, και στη συνέχεια αποστέλει μια πλειάδα 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 αποστέλει ένα $atomsqrt$ αντί για neg στο πρώτο της μήνυμα.

Στα σχήματα 5.4 έως 5.7 φαίνονται οι τύποι συνεδρίας των παραπάνω συναρτήσεων, σε μορφή γραφήματος.

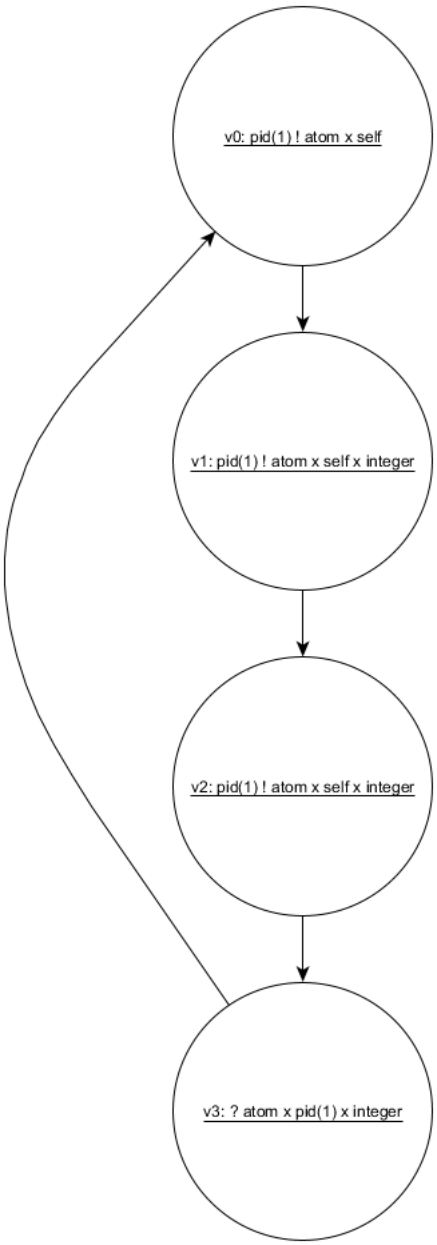
Σχήμα 5.4: server/0



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

```
{server,0} : ”u (4). Of<any; ?(atom x {atom, pid, (atom x atom)}).
                             ?(atom x {atom, pid, (atom x atom)} x number).
                             ?(atom x {atom, pid, (atom x atom)} x number).
                             pid(1) !(atom x pid x number).rec(4).,
                        any; ?(atom x {atom, pid, (atom x atom)}).
                             ?(atom x {atom, pid, (atom x atom)} x number).
                             pid(1) !(atom x pid x number).rec(4).,
                        any; ?(atom x {atom, pid, (atom x atom)}).
                             ?(atom x {atom, pid, (atom x atom)} x number).
                             pid(1) !(atom x pid x float).rec(4).,
                        any; ?(atom x any).>.”
```

Σχήμα 5.5: sum_client/2

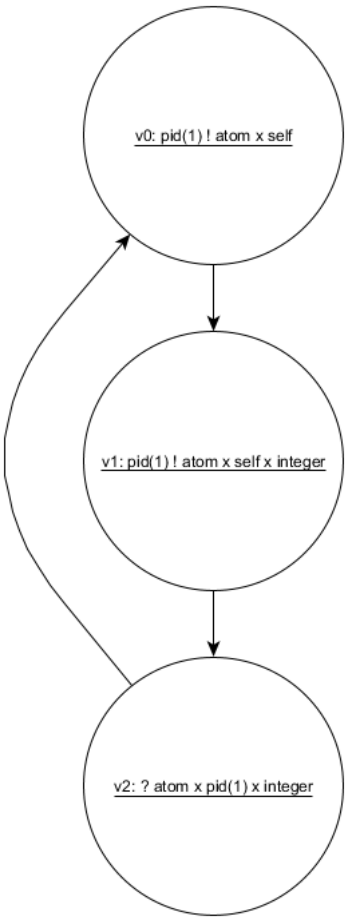



```
{sum_client,2} : "u (3). pid(1) !(atom x pid).
                         pid(1) !(atom x pid x integer).
                         pid(1) !(atom x pid x integer).
                         ?(atom x pid x any).rec(3)."

{neg_client,2} : "u (2). pid(1) !(atom x pid).
                         pid(1) !(atom x pid x integer).
                         ?(atom x pid x any).rec(2)."

{sqrt_client,2} : "u (1). pid(1) !(atom x pid).
                          pid(1) !(atom x pid x integer).
                          ?(atom x pid x any).rec(1)."
```

Listing 5.10: Inferred Session Types of Arithmetic Server

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

Σχήμα 5.6: neg_client/2



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

Ακολούθως, προσπαθούμε να δούμε αν όντως μπορεί να διαδραματιστεί επιτυχής επικοινωνία ανάμεσα στα endpoints μας. Αρχικά, έχουμε τον server με τον client άθροισης, $sum\_client/2$.

```
{server,0} and {sum_client,2} are peers
The matching sequences are:
[{{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
  {send,{other_proc,1},{tuple,[atom,pid]}}},
 {{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]},number]}},
  {send,{other_proc,1},{tuple,[atom,pid,integer]}}},
 {{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]},number]}},
  {send,{other_proc,1},{tuple,[atom,pid,integer]}}},
 {{send,{other_proc,1},{tuple,[atom,pid,number]}},
  {recv,any,{tuple,[atom,pid,any]}}}]
```

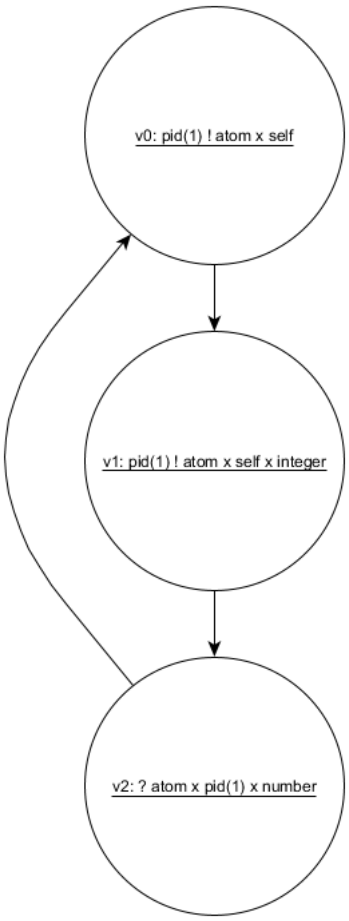
Listing 5.11: server/0 and sum_client/2

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

Τα πράγματα για τους $neg\_client/2$ και $sqrt\_client/2$ είναι ελαφρώς διαφορετικά:

```

{server,0} and {neg_client,2} are peers
The matching sequences are:
[{{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]}]}},
  {send,{other_proc,1},{tuple,[atom,pid]}}},
 {{recv,any,{tuple,[atom,{union,[atom,pid,{tuple,[atom,atom]}]},number]}},
  {send,{other_proc,1},{tuple,[atom,pid,integer]}}},
 {{send,{other_proc,1},{tuple,[atom,pid,float]}},
```

Σχήμα 5.7: sqrt_client/2




```
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 {sqrt_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]}},
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 τε-

τραγωνικής 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

[AB] Ericsson AB, "Erlang Typespec".

[Dama82] Luis Damas and Robin Milner, "Principal Type-schemes for Functional Programs", in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pp. 207–212, New York, NY, USA, 1982, ACM.

[DOrc16] N.Yoshida D. Orchard, "Effects as Sessions, Sessions as Effects", in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pp. 568–581, ACM New York, 2016.

[Fowl16] Simon Fowler, "An Erlang Implementation of Multiparty Session Actors", in *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pp. 36–50, 2016.

[Heng94] Fritz Henglein and Christian Mossin, "Polymorphic Binding-Time Analysis", in *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*, ESOP '94, pp. 287–301, Berlin, Heidelberg, 1994, Springer-Verlag.

[Hond98] Kohei Honda, Vasco Thudichum Vasconcelos and Makoto Kubo, "Language Primitives and Type Discipline for Structured Communication-Based Programming", in *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pp. 122–138, London, UK, UK, 1998, Springer-Verlag.

[Hond16] Kohei Honda, Nobuko Yoshida and Marco Carbone, "Multiparty Asynchronous Session Types", *J. ACM*, vol. 63, no. 1, pp. 9:1–9:67, March 2016.

[Neyk13] Rumyana Neykova, Nobuko Yoshida and Raymond Hu, "SPY: Local Verification of Global Protocols", pp. 358–363, 09 2013.

[Niel99] Flemming Nielson and Hanne Riis Nielson, "Type and Effect Systems", in *Correct System Design*, 1999.

[Pier02] Benjamin C. Pierce, *Types and Programming Languages*, The MIT Press, 1st edition, 2002.

[Pier04] Benjamin C. Pierce, *Advanced Topics in Types and Programming Languages*, The MIT Press, 2004.

[Solb95] Kirsten Solberg, "Annotated Type Systems for Program Analysis", *DAIMI Report Series*, vol. 24, no. 498, Nov. 1995.

[Take94] Kaku Takeuchi, Kohei Honda and Makoto Kubo, "An Interaction-based Language and its Typing System", in *In PARLE'94, volume 817 of LNCS*, pp. 398–413, Springer-Verlag, 1994.

[TLin06] K. Sagonas T. Lindahl, "Practical Type Inference Based on Success Typings", 2006.