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

Ασφάλεια σε Συστήματα Τύπων με Τύπους Τομής και Άρνησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΑΡΩΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάιος 2019

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

Ασφάλεια σε Συστήματα Τύπων με Τύπους Τομής και Άρνησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΑΡΩΝΗΣ

Επιβλέπων : Νικόλαος Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Μαΐου 2019.

.....	.....	.....
Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Παναγιώτης Ροντογιάννης Καθηγητής Ε.Κ.Π.Α.

Αθήνα, Μάιος 2019

.....

Παναγιώτης Αρώνης

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

Περίληψη

Ο προγραμματισμός ηλεκτρονικών υπολογιστών είναι αρκετά πιο δημοφιλής σήμερα, με ολοένα και περισσότερα άτομα να ασχολούνται με αυτόν. Γλώσσες προγραμματισμού υψηλού επιπέδου με πανίσχυρες γενικευμένες έννοιες κάνουν εύκολο για τον οποιονδήποτε να γράψει προγράμματα. Από την άλλη μεριά, το να δείξει κανείς ότι ένα πρόγραμμα έχει την επιθυμητή συμπεριφορά είναι μία αρκετά δυσκολότερη διαδικασία. Ο έλεγχος τύπων είναι η πιο βασική προσέγγιση για να το επιτύχουμε. Σε αυτήν τη διπλωματική, παρουσιάζουμε ένα στατικό σύστημα τύπων, που περιέχει τύπους τομής και άρνησης και μία σχέση υποτύπων, ώστε να αναθέσουμε αρκετά περιγραφικούς τύπους σε προγράμματα μία απλής συναρτησιακής γλώσσας με ταίριασμα προτύπων. Δείχνουμε ότι το σύστημα τύπων μας αναθέτει τύπους μόνο σε προγράμματα με καθορισμένη σημασιολογία, δηλαδή, ότι έχει την ιδιότητα της ασφάλειας τύπων.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Συστήματα τύπων, Ασφάλεια τύπων, Τύποι τομής, Τύποι άρνησης, Υποτύποι.

Abstract

Computer programming is a lot more popular nowadays, with more and more individuals involved with it. High-level programming languages with powerful abstractions make writing programs an easy task for anyone. On the other hand, proving that a program has the intended behavior is a much harder task. Type checking is the most basic approach to achieve this. In this thesis, we present a static type system, that includes intersection and negation types and subtyping, to assign very descriptive types to programs of a simple functional language with pattern matching. We prove that our type system assigns types only to programs with defined semantics, that is, it has the safety property.

Key words

Programming languages, Type systems, Type safety, Intersection types, Negation types, Subtyping.

Ευχαριστίες

Αρχικά, θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της διπλωματικής, Νικόλαο Παπασπύρου, για την πολύτιμη βοήθειά του, αλλά, κυριότερα, επειδή αποτέλεσε πρότυπό μου στη σχολή, ωθώντας με να ασχοληθώ και να αγαπήσω τον προγραμματισμό. Έπειτα, ευχαριστώ τους φίλους μου για τις αμέτρητες συζητήσεις που κάναμε, οι οποίες διεύρυναν τους ορίζοντές μου και διαμόρφωσαν την προσωπικότητά μου. Επίσης, ευχαριστώ τους γονείς μου για την αμέριστη υποστήριξή τους και όλη την προσπάθεια που κατέβαλαν χωρίς δισταγμό. Τέλος, ευχαριστώ τη Νάνσυ που βρίσκεται συνέχεια δίπλα μου, κάνοντας κάθε δύσκολη στιγμή υποφερτή και κάθε χαρούμενη τόσο ομορφότερη.

Παναγιώτης Αρώνης,

Αθήνα, 7η Μαΐου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-6-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Μάιος 2019.

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

Contents

List of Figures

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σε αυτήν τη διπλωματική εργασία, ορίζουμε ένα στατικό σύστημα τύπων, το οποίο συμπεριλαμβάνει τύπους τομής και τύπους άρνησης και ένα σύστημα υποτύπων, έτσι ώστε να αναθέσουμε αρκετά περιγραφικούς τύπους σε προγράμματα μίας απλής συναρτησιακής γλώσσας προγραμματισμού που έχει ταίριασμα προτύπων. Δείχνουμε ότι το σύστημα τύπων μας αναθέτει τύπους μόνο σε προγράμματα με καθορισμένη σημασιολογία, δηλαδή, ότι έχει την ιδιότητα της ασφάλειας τύπων.

1.2 Κίνητρο

Ο προγραμματισμός υπολογιστών συνεχίζει να γίνεται πιο δημοφιλής σήμερα, με ολοένα και περισσότερα άτομα να ασχολούνται με αυτόν. Τα βασικά εργαλεία του προγραμματισμού, οι γλώσσες προγραμματισμού, είναι πλέον αρκετά εύκολες για τον οποιονδήποτε να τις μάθει. Υπολογιστές σε κάθε σημείο χειρίζονται τις περισσότερες από τις καθημερινές μας εργασίες με αρκετά αποδοτικό τρόπο. Παρόλα αυτά, με τα επιπλέον επίπεδα αφαίρεσης, η συλλογιστική περί της συμπεριφοράς των προγραμμάτων αναδύεται ως ολοένα και σημαντικότερο πρόβλημα. Ο περισσότερος χρόνος ενός προγραμματιστή αναλώνεται στην αποσφαλμάτωση και στον έλεγχο λειτουργίας ενός λογισμικού με σκοπό να επιβεβαιώσει ότι πράγματι δουλεύει με τον επιθυμητό τρόπο.

Ο στατικός έλεγχος τύπων είναι η πιο βασική μορφή επαλήθευσης της συμπεριφοράς προγραμμάτων. Έχοντας ένα σύστημα το οποίο προσπαθεί να αποδώσει τύπους σε προγράμματα βάσει της δομής τους, επιτρέπεται ο έγκαιρος εντοπισμός κάποιων προγραμματιστικών σφαλμάτων. Ο τύπος που ανατίθεται σε κάποιο πρόγραμμα παρέχει, επίσης, χρήσιμη πληροφορία σχετικά με τη λειτουργία του, αποτελώντας μία εγγενή επεξήγησή του που εξελίσσεται ταυτόχρονα με τον πηγαίο κώδικά του.

Φυσικά, όσο πιο ακριβής είναι αυτή η πληροφορία για ένα πρόγραμμα, τόσο πιο χρήσιμη είναι. Η ύπαρξη ενός συστήματος τύπων που αναθέτει κατάλληλους αλλά και περιγραφικούς τύπους σε προγράμματα απαιτεί ένα ευρύ σύνολο από κατασκευαστές τύπων, οι οποίοι καταγράφουν συγκεκριμένες σχέσεις μεταξύ άλλων τύπων. Βέβαια, θα πρέπει να εξασφαλίσουμε ότι αρκετά έγκυρα προγράμματα μπορούν ακόμα να αντιστοιχηθούν σε κάποιο τύπο, δηλαδή, οι τύποι των μερών που τα απαρτίζουν είναι δυνατό να ταιριάξουν μεταξύ τους.

Σε αυτήν τη διπλωματική εργασία, παρουσιάζουμε ένα τέτοιο σύστημα τύπων. Δουλεύουμε με μία πλούσια άλγεβρα από κατασκευαστές τύπων, όπως τύποι τομής και τύποι άρνησης. Επιτρέπουμε σε εκφράσεις να έχουν αρκετούς τύπους με τη χρήση μίας συντακτικά ορισμένης σχέσης υποτύπων. Η γλώσσα προγραμματισμού, στην οποία αποδίδουμε τύπους, είναι μία απλή συναρτησιακή γλώσσα που χρησιμοποιεί ταίριασμα προτύπων στον ορισμό συναρτήσεων με πολλαπλές ρήτρες. Αυτό το χαρακτηριστικό της γλώσσας μας επιτρέπει τη χρήση συνολοθεωρητικών κατασκευαστών τύπων, ώστε να αναθέσουμε τύπους στις συναρτήσεις της, οι οποίοι παρέχουν μία πολύ κοντινή περιγραφή σε σχέση με την πραγματική λειτουργία τους.

Κεφάλαιο 2

Ορισμός Γλώσσας

Για να σχεδιάσουμε σύνθετα συστήματα τύπων και να αναλύσουμε τις ιδιότητές τους, πρώτα χρειάζεται να ορίσουμε μία κατάλληλη γλώσσα προγραμματισμού χωρίς τύπους, στην οποία θα εισάγουμε τύπους αργότερα. Αυτή η γλώσσα θα πρέπει να είναι αρκετά απλή και κομψή, έτσι ώστε να αποφύγουμε τη συζήτηση για τις περισσότερες από τις τεχνικές λεπτομέρειες περί των δομών της, αλλά ταυτόχρονα, χρειάζεται κάποια ενδιαφέροντα και χρήσιμα χαρακτηριστικά, που βρίσκονται σε πραγματικές γλώσσες προγραμματισμού, ώστε να αξίζει να ασχοληθούμε με το συλλογισμό της λειτουργίας προγραμμάτων σε αυτήν. Αυτές οι απαιτήσεις μας οδηγούν σε μία συναρτησιακή γλώσσα, παρόμοια με αυτές της οικογένειας της ML, που έχει ως πυρήνα της τον γνωστό αγνό λάμδα λογισμό. Ακολουθεί η παρουσίαση της επιλεγμένης γλώσσας, δίνοντας τους ορισμούς της σύνταξης και της σημασιολογίας της.

2.1 Σύνταξη

Η σύνταξη του λάμδα λογισμού περιέχει μόνο τρία είδη εκφράσεων, συγκεκριμένα, μεταβλητές, λάμδα αφαιρέσεις (ορισμούς συναρτήσεων) και εφαρμογές υποεκφράσεων. Οποιοσδήποτε μαθηματικός υπολογισμός είναι δυνατό να αναχθεί σε αυτές τις βασικές λειτουργίες, κάνοντας το λάμδα λογισμό ταυτόχρονα μία απλή γλώσσα προγραμματισμού και ένα μαθηματικό αντικείμενο για το οποίο σύνθετα θεωρήματα μπορούν να αποδειχθούν.

Επεκτείνουμε αυτή τη στοιχειώδη σύνταξη, για λόγους άνεσης και παρουσίασης, αλλά, επίσης, με σκοπό να παρουσιάσουμε ένα σύστημα τύπων που δουλεύει καλά με ορισμένα χαρακτηριστικά γλωσσών προγραμματισμού. Η γλώσσα μας περιλαμβάνει ένα σύνολο σταθερών και τελεστών, ένα τελεστή σταθερού σημείου και ορισμό συναρτήσεων με πολλαπλές ρήτρες και ταίριασμα προτύπων.

Η πλήρης αφηρημένη σύνταξη της γλώσσας φαίνεται στο σχήμα 2.1. Οι συμβάσεις που χρησιμοποιούνται σε αυτήν τη γραμματική (και στο υπόλοιπο αυτής της εργασίας) είναι παρόμοιες με αυτές των συμβατικών BNF γραμματικών. Η κύρια διαφορά είναι ότι επιτρέπουμε την αναπαράσταση μίας πεπερασμένης ακολουθίας ως $\alpha_1, \ldots, \alpha_n$ για απλότητα.

2.1.1 Μορφές Εκφράσεων

Η γραμματική της αφηρημένης σύνταξης της γλώσσας είναι απλά ένας συμπαγής συμβολισμός για να ορίσουμε επαγωγικά όλες τις έγκυρες δομές που μπορούν να κατασκευαστούν στη γλώσσα μας. Ονομάζουμε αυτές τις δομές όρους ή εκφράσεις (expressions). Χρησιμοποιούμε τη μεταβλητή e (με διάφορους δείκτες) για να αναφερθούμε σε οποιαδήποτε έκφραση, ενώ E είναι το σύνολο που περιέχει όλες τις εκφράσεις.

Ας εξετάσουμε τώρα τις δυνατές συντακτικές μορφές μίας έκφρασης e δίνοτας περισσότερες λεπτομέρειες για αυτές. Για οποιαδήποτε έκφραση e ακριβώς μία από τις παρακάτω προτάσεις είναι αληθής:

- Η έκφραση e έχει τη μορφή μίας μεταβλητής (variable), $e = x$. Χρησιμοποιούμε τις μεταβλητές x και y για να αναφερθούμε σε μεταβλητές της γλώσσας μας. Οι μεταβλητές είναι ονόματα που μπορούν να αναφέρονται σε παραμέτρους συναρτήσεων και χρησιμοποιούνται, στην ουσία, ως

σύμβολα θέσεων αντικατάστασης από άλλες εκφράσεις κατά τη διάρκεια υπολογισμού ενός προγράμματος.

- Η έκφραση e έχει τη μορφή μίας σταθεράς (constant), $e = c\,(e_1, \ldots, e_n)$ για κάποιες εκφράσεις e_i. Οι σταθερές είναι πλειάδες κάποιου μεγέθους και έχουν επίσης ένα όνομα. Χρησιμοποιούμε τη μεταβλητή c για το όνομα σταθερών, ενώ ο αριθμός n είναι το μέγεθός της. Τα στοιχεία μίας σταθεράς, $e_1, \ldots, e_n$, δεν χρειάζεται να είναι τα ίδια σταθερές, μπορεί να είναι οποιεσδήποτε εκφράσεις. Ονομάζουμε αυτήν τη μορφή εκφράσεων σταθερά, καθώς η εξωτερική της δομή (όνομα και μέγεθος) δεν μπορεί να αλλάξει κατά τη διάρκεια υπολογισμών. Επίσης, οι πραγματικές μαθηματικές σταθερές, όπως οι ακέραιοι αριθμοί και οι λογικές σταθερές, είναι σε αυτήν τη μορφή, όπου $n = 0$ και το όνομα της σταθεράς αντιστοιχεί στην τιμή της. Οι σταθερές μπορούν να χρησιμοποιηθούν για να κατασκευάσουμε οποιαδήποτε σύνθετη εμφωλιασμένη δομή ώστε να οργανώσουμε τα δεδομένα μας με κάποιο συγκεκριμένο τρόπο.

- Η έκφραση e έχει τη μορφή ενός τελεστή (operator), $e = op\,(e_1, \ldots, e_n)$ για κάποιες εκφράσεις e_i. Οι τελεστές έχουν ονόματα και πλήθος ορισμάτων. Χρησιμοποιούμε τη μεταβλητή op για ονόματα τελεστών, ενώ ο αριθμός n είναι το πλήθος ορισμάτων του τελεστή. Κάθε τελεστής θα πρέπει να συνοδεύεται από τη σημασιολογική του συνάρτηση, $[\![op]\!]$, η οποία ορίζει πως ο τελεστής εφαρμόζεται στα τελούμενά του $e_1, \ldots, e_n$ και αλλάζει την έκφραση του τελεστή στο υπολογισμένο αποτέλεσμά της. Οπότε, αντίθετα με τις σταθερές, το σύνολο των τελεστών της γλώσσας δε μπορεί να επεκταθεί από τον προγραμματιστή. Συμπεριλαμβάνουμε τελεστές στη γλώσσα μας για να έχουμε μερικές βασικές λειτουργίες πάνω στις απλούστερες σταθερές μας, όπως αριθμητικές και λογικές πράξεις, αλλά θα μπορούσαμε να έχουμε βιβλιοθήκες με διάφορους τελεστές, που υλοποιούν οποιοδήποτε σύνθετο υπολογισμό σε οποιοδήποτε είδος από δομημένες σταθερές.

- Η έκφραση e έχει τη μορφή μίας συνάρτησης (function), $e = \texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$. Οι ορισμοί συναρτήσεων εσωκλείονται από τις λέξεις κλειδιά `fun` και `end` και περιέχουν μία ακολουθία από συναρτησιακές ρήτρες (clauses). Χρησιμοποιούμε τη μεταβλητή d για ρήτρες, ενώ ο αριθμός m είναι το πλήθος της ακολουθίας τους. Κάθε συναρτησιακή ρήτρα d, έχει τη μορφή $p \rightarrow e_r$, όπου ονομάζουμε το p πρότυπο της ρήτρας και το e_r έκφραση επιστροφής της ρήτρας. Υπάρχουν δύο είδη προτύπων, συγκεκριμένα, το πρότυπο μεταβλητής, $p = x$, όπου x είναι μία μεταβλητή, και το σταθερό πρότυπο, $p = c\,(x_1, \ldots, x_n)$, όπου καθένα από τα x_i είναι μία μεταβλητή και c είναι το όνομα μίας σταθεράς μεγεθούς n. Όλη αυτή η σύνταξη των συναρτήσεων περιγράφει τους υπολογισμούς που χρειάζεται να πραγματοποιηθούν σε κάποια αναμενόμενη έκφραση με συγκεκριμένη μορφή (είσοδος συνάρτησης) με σκοπό την επιστροφή μίας άλλης έκφρασης ως αποτέλεσμα (έξοδος συνάρτησης). Τέλος, επιτρέπουμε την ειδική συνάρτηση `fun end`, δηλαδή μία συνάρτηση χωρίς καμμία ρήτρα ($m = 0$), ως έγκυρη έκφραση συνάρτησης, καθώς, όπως θα δούμε αργότερα, έχει έναν ειδικό ρόλο στην αποτίμηση των εκφράσεων.

- Η έκφραση e έχει τη μορφή μίας εφαρμογής (application), $e = e_1\, e_2$ για κάποιες εκφράσεις e_1 και e_2. Οι εφαρμογές συνδέουν τα βασικά δομικά μέρη των προγραμμάτων, τις συναρτήσεις, με άλλες εκφράσεις ως είσοδό τους. Εάν οι εκφράσεις ταιριάζουν, θα πάρουμε τελικά την αναμενόμενη έξοδο της συνάρτησης. Προς αποφυγή πολλών παρενθέσεων, υιοθετούμε τη σύμβαση ότι οι εφαρμογές είναι αριστερά προσεταιριστικές, δηλαδή, $e_1\, e_2\, e_3$ είναι η ίδια έκφραση με $(e_1\, e_2)\, e_3$.

- Η έκφραση e έχει τη μορφή ενός σταθερού σημείου (fix-point), $e = \texttt{fix}\, x\,.\, e_r$ για κάποια έκφραση e_r. Οι ορισμοί σταθερών σημείων ξεκινούν με τη λέξη κλειδί `fix` ακολουθούμενο από το όνομα μίας μεταβλητής και λήγουν με μία άλλη έκφραση, την έκφραση επιστροφής του σταθερού σημείου. Μία έκφραση σταθερού σημείου επιτρέπει στη μεταβλητή της, x, να αναφέρεται πίσω στην ολόκληρη αρχική έκφραση e, κάνοντας δυνατό τον ορισμό αναδρομικών υπολογι-

σμών. Θα δούμε αργότερα πως αυτή η κυκλική συμπεριφορά επιτυγχάνεται εξετάζοντας την αποτίμηση των εκφράσεων.

2.1.2 Τιμές

Ονομάζουμε μία έκφραση που θεωρούμε ως έγκυρο δυνατό τελικό υπολογισμένο αποτέλεσμα ενός προγράμματος τιμή (value). Η αφηρημένη σύνταξη των τιμών στη γλώσσα μας φαίνεται στο σχήμα 2.2. Χρησιμοποιούμε τη μεταβλητή v για να αναφερθούμε σε οποιαδήποτε τιμή, ενώ V είναι το σύνολο που τις περιέχει όλες. Η μεταβλητή d χρησιμοποιείται ξανά για συναρτησιακές ρήτρες, οι οποίες έχουν ακριβώς την ίδια σύνταξη όπως δόθηκε στο σχήμα 2.1.

Από αυτήν τη γραμματική για τις τιμές, έχουμε ότι για οποιαδήποτε τιμή v ακριβώς ένα από τα παρακάτω είναι αληθές:

- Η τιμή v έχει τη μορφή μίας σταθεράς (constant), $v = c\,(v_1, \ldots, v_n)$ για κάποιες τιμές v_i. Δηλαδή, μία έκφραση σταθεράς είναι τιμή μόνο εάν όλα τα στοιχεία της είναι επίσης τιμές.
- Η τιμή v έχει τη μορφή μίας συνάρτησης (function), $v =$ `fun` $d_1 \mid \ldots \mid d_m$`end`. Αντίθετα με τις σταθερές, κάθε έκφραση συνάρτησης είναι επίσης τιμή.

Το σύνολο των τιμών δεν μπορεί να οριστεί αυτόνομα, επειδή οι τιμές είναι συνδεδεμένες με τον τρόπο που διενεργούνται οι υπολογισμοί στη γλώσσα μας. Θα πρέπει να ελέγξουμε ότι οι τιμές, όπως δόθηκαν εδώ, συμφωνούν με τον ορισμό της αποτίμησης των εκφράσεων που θα δοθεί σε επόμενη ενότητα.

2.1.3 Ταίριασμα Προτύπων

Είδαμε προηγουμένως ότι η γλώσσα μας έχει συναρτήσεις με πολλαπλές ρήτρες, που ορίζουν πολλές εκφράσεις επιστροφής, όπου καθεμία αντιστοιχεί σε κάποιο πρότυπο. Με σκοπό να υπολογίσουμε την έξοδο της συνάρτησης για μία δοσμένη έκφραση εισόδου, θα πρέπει να επιλέξουμε μία από αυτές τις εκφράσεις επιστροφής βάσει της δομής της εισόδου και του ποια πρότυπα ταιριάζουν με αυτήν τη δομή. Αυτό το χαρακτηριστικό της γλώσσας δίνει μία ισχυρή δομή προγραμματισμού για εκτέλεση υπολογισμών υπό συνθήκη που ονομάζεται ταίριασμα προτύπων και συναντάται συχνά σε συναρτησιακές γλώσσες προγραμματισμού όπως η Standard ML.

Ο ακόλουθος ορισμός περιγράφει πότε το πρότυπο μίας ρήτρας ταιριάζει με μία δοθείσα έκφραση:

Ορισμός (Ταίριασμα Προτύπου)**.** Ταίριασμα μεταξύ ενός προτύπου p και μίας δοθείσας έκφρασης e.

- Ένα πρότυπο μεταβλητής, $p = x$, ταιριάζει με οποιαδήποτε δοθείσα έκφραση e.
- Ένα πρότυπο σταθεράς, $p = c\,(x_1, \ldots, x_n)$, ταιριάζει με μία δοθείσα έκφραση e, μόνο εάν e έχει τη μορφή μίας σταθεράς με το ίδιο όνομα c και το ίδιο μέγεθος n.

2.1.4 Εμβέλειες και Δέσμευση Μεταβλητών

Ας εξετάσουμε τώρα πως οι μεταβλητές της γλώσσας μας χρησιμοποιούνται ως σύμβολα αντικατάστασης από άλλες εκφράσεις για να ορίσουμε υπολογισμούς με αφηρημένο τρόπο. Κάποιες εκφράσεις, όπως οι ορισμοί συναρτήσεων, εισάγουν ονόματα μεταβλητών για να αναφερθούν στις παραμέτρους που αναμένουν και χρησιμοποιούν αυτά τα ονόματα στο σώμα τους για να επισημάνουν σε ποιο σημείο αυτά χρειάζονται ώστε να υπολογιστεί η επιθυμητή έξοδος. Αυτές οι μεταβλητές ονομάζονται δεσμευμένες μεταβλητές επειδή δεν μπορούν να αντικατασταθούν ελεύθερα από οποιαδήποτε έκφραση, καθώς αναφέρονται πίσω στο ίδιο σημείο, στην εισαγωγή τους (ορισμό τους). Το μέρος μίας έκφρασης που εισάγει (ορίζει) μία νέα δεσμευμένη μεταβλητή ονομάζεται δεσμευτής, ενώ η έκφραση στο εσωτερικό της οποίας μπορεί να χρησιμοποιηθεί η δεσμευμένη μεταβλητή ονομάζεται εμβέλειά του.

Ο παρακάτω ορισμός καταγράφει όλες τις εκφράσεις που εισάγουν δεσμευμένες μεταβλητές στη γλώσσα μας:

Ορισμός (Δεσμευμένες Μεταβλητές)**.** Εκφράσεις που εισάγουν δεσμευμένες μεταβλητές (δεσμευτές) και οι εμβέλειές τους.

- Σε μία ρήτρα συνάρτησης με πρότυπο μεταβλητής, $x \to e_r$, η μεταβλητή x είναι δεσμευμένη και η εμβέλειά της είναι η έκφραση e_r.
- Σε μία ρήτρα συνάρτησης με πρότυπο σταθεράς, $c\,(x_1, \ldots, x_n) \to e_r$, καθεμία από τις μεταβλητές x_i είναι δεσμευμένη και η εμβέλειά της είναι η έκφραση e_r.
- Σε μία έκφραση σταθερού σημείου, $\texttt{fix}\, x\,.\, e_r$, η μεταβλητή x είναι δεσμευμένη και η εμβέλειά της είναι η έκφραση e_r.

Μία εμφάνιση μεταβλητής είναι δεσμευμένη (bound) όταν βρίσκεται στην εμβέλεια ενός δεσμευτή με το ίδιο όνομα. Μία εμφάνιση μεταβλητής είναι ελεύθερη (free) εάν βρίσκεται σε μία θέση όπου δεν δεσμεύεται από οποιονδήποτε δεσμευτή με το ίδιο όνομα. Μία έκφραση της γλώσσας που δεν περιέχει εμφανίσεις ελεύθερων μεταβλητών ονομάζεται κλειστή (closed).

2.1.5 Αντικατάσταση

Μία ελεύθερη εμφάνιση μεταβλητής σε μία έκφραση είναι μία εξωτερική αναφορά που χρειάζεται να επιλυθεί για να αποδοθεί το πλήρες νόημα της έκφρασης. Η διαδικασία που τοποθετεί άλλες εκφράσεις στη θέση μεταβλητών, δίνοντας μία πιο ειδική περίπτωση της αρχικής έκφρασης, ονομάζεται αντικατάσταση εκφράσεων. Χρησιμοποιούμε το συμβολισμό $e'\,[x \mapsto e]$ για την έκφραση που παίρνουμε ύστερα από αντικατάσταση κάθε εμφάνισης της μεταβλητής x στην έκφραση e' από την έκφραση e.

Αν και η διαδικασία της αντικατάστασης μοιάζει απλή, παρουσιάζει κάποιες τεχνικές δυσκολίες όταν εξεταστεί λεπτομερώς. Συγκεκριμένα, θα πρέπει να χειριστούμε τις εξής δύο περιπτώσεις προσεκτικά ώστε να ορίσουμε την αντικατάσταση εκφράσεων σωστά. Πρώτον, θα πρέπει να αντικαταστούμε μόνο ελεύθερες εμφανίσεις μεταβλητών, καθώς οι δεσμευμένες μεταβλητές αναφέρονται στον ορισμό τους μέσα στην αρχική έκφραση. Δεύτερον, θα πρέπει να αποφύγουμε τη δέσμευση ελεύθερων μεταβλητών, που εμφανίζονται στην έκφραση με την οποία αντικαταστούμε από κάποιο δεσμευτή της αρχικής έκφρασης. Και τα δύο αυτά προβλήματα επιλύονται με κατάλληλη μετονομασία των δεσμευμένων μεταβλητών στην αρχική έκφραση πριν την αντικατάσταση. Επιπροσθέτως, μπορούμε πάντα να κάνουμε αυτή τη μετονομασία, καθώς θεωρούμε τις εκφράσεις που διαφέρουν μόνο στα ονόματα δεσμευμένων μεταβλητών ισοδύναμες.

Μία διαδικασία ασφαλούς αντικατάστασης εκφράσεων με την επιθυμητή συμπεριφορά περιγράφεται στον ορισμό 2.1.3. Από εδώ και στο εξής, κάθε αναφορά σε αντικατάσταση εκφράσεων θα νοείται ως ασφαλής, θεωρώντας πως έχει γίνει κατάλληλη μετονομασία δεσμευμένων μεταβλητών. Τέλος, θα χρησιμοποιήσουμε το γενικευμένο συμβολισμό $e\,[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ για την ταυτόχρονη αντικατάσταση κάθε μεταβλητής x_i από την έκφραση e_i στην έκφραση e.

2.2 Σημασιολογία

Έχοντας διατυπώσει λεπτομερώς τη σύνταξη της γλώσσας μας, τώρα χρειαζόμαστε έναν εξίσου ακριβή ορισμό του πως οι εκφράσεις της γλώσσας αποτιμούνται, με άλλα λόγια της σημασιολογίας της γλώσσας μας.

Στον αγνό λάμδα λογισμό, ο μόνος τρόπος με τον οποίο οι εκφράσεις υπολογίζονται είναι η εφαρμογή συναρτήσεων σε ορίσματα. Κάθε βήμα υπολογισμού συνίσταται από την αναδιατύπωση μίας εφαρμογής μίας συνάρτησης σε κάποια άλλη έκφραση, μέσω αντικατάστασης της δεσμευμένης μεταβλητής από αυτή την έκφραση στο σώμα της συνάρτησης.

Έχοντας εμπλουτίσει τη γλώσσα μας με επιπλέον δομές, όπως σταθερές και τελεστές, και χαρακτηριστικά, όπως ταίριασμα προτύπων στον ορισμό συναρτήσεων, υπάρχουν σίγουρα περισσότερες περιπτώσεις να σκεφτούμε στον ορισμό των υπολογισμών, αλλά η διαδικασία της αντικατάστασης εκφράσεων έχει ακόμα τον ίδιο ρόλο της τοποθέτησης εκφράσεων ως ορισμάτων σε συναρτήσεις στην περίπτωση εφαρμογών.

2.2.1 Λειτουργική Σημασιολογία

Υπάρχουν μερικές διαφορετικές βασικές προσεγγίσεις για τη διατύπωση της σημασιολογίας γλωσσών προγραμματισμού. Μία από αυτές τις μεθόδους είναι η λειτουργική σημασιολογία (operational semantics). Στη λειτουργική σημασιολογία προσδιορίζουμε τη συμπεριφορά μίας γλώσσας προγραμματισμού ορίζοντας μία απλή αφηρημένη μηχανή για αυτήν, που χρησιμοποιεί τις εκφράσεις της γλώσσας ως τον κώδικα μηχανής της. Για απλές γλώσσες, μία κατάσταση της μηχανής είναι απλά μία έκφραση, και η συμπεριφορά της μηχανής ορίζεται από μία συνάρτηση μετάβασης, η οποία, για κάθε κατάσταση, είτε δίνει την επόμενη κατάσταση εκτελώντας ένα βήμα απλοποίησης στην τρέχουσα έκφραση, είτε δηλώνει ότι η μηχανή έχει σταματήσει. Η τελική κατάσταση που φτάνει η μηχανή όταν ξεκινά με μία συγκεκριμένη έκφραση ως αρχική της κατάσταση αντιστοιχεί στη σημασία αυτής της έκφρασης.

Η λειτουργική σημασιολογία είναι συνήθως η μέθοδος που προτιμάται για τον ορισμό γλωσσών προγραμματισμού και της μελέτης των ιδιοτήτων τους. Θα χρησιμοποιήσουμε, επίσης, αυτή τη μέθοδο για τον ορισμό της σημασιολογίας, καθώς είναι απλή και συνοπτική αλλά ταυτόχρονα αρκετά εκφραστική.

2.2.2 Σχέση Αποτίμησης

Μπορούμε τώρα να ορίσουμε μία διμελή σχέση στο σύνολο των εκφράσεων, E, η οποία περιέχει ακριβώς όλα τα διατεταγμένα ζεύγη εκφράσεων, τα οποία είναι δυνατές διαδοχικές καταστάσεις της αντίστοιχης αφηρημένης μηχανής, δηλαδή, η μηχανή μπορεί να αλλάξει την κατάστασή της από την πρώτη έκφραση στη δεύτερη, πραγματοποιώντας ένα μόνο απλό βήμα υπολογισμού. Ονομάζουμε αυτή τη σχέση (μονοβηματική) σχέση αποτίμησης (single-step evaluation relation) και γράφουμε $e \longrightarrow e'$ για να σημειώσουμε ότι η έκφραση e αποτιμάται στην έκφραση e' σε ένα βήμα.

Η σχέση αποτίμησης ορίζεται επαγωγικά μέσω ενός συνόλου από κανόνες συμπερασμού και φαίνεται στο σχήμα 2.3. Κάθε κανόνας συμπερασμού έχει ένα πλήθος από προϋποθέσεις και ένα μόνο συμπέρασμα. Κανόνες δίχως οριζόντια γραμμή (γνωστοί και ως αξιώματα) δεν έχουν καμμία προϋπόθεση, οπότε το συμπέρασμά τους είναι πάντα αληθές. Προσοχή πρέπει να δοθεί στην επιλογή των μεταβλητών στους κανόνες συμπερασμού. Υπενθυμίζουμε ότι χρησιμοποιούμε τη μεταβλητή e για οποιαδήποτε έκφραση στο σύνολο E, όπως δόθηκε στο σχήμα 2.1, και τη μεταβλητή v για οποιαδήποτε τιμή στο σύνολο $V \subset E$, όπως δόθηκε στο σχήμα 2.2. Η επιλογή των κατάλληλων μεταβλητών βοηθά στον έλεγχο της σειράς της αποτίμησης. Παρατηρούμε ότι η γλώσσα μας χρησιμοποιεί τη στρατηγική αποτίμησης της κλήσης κατά τιμή (call by value), όπου δεν επιτρέπεται η αποτίμηση στο εσωτερικό μίας συνάρτησης και οι παράμετροι μίας συνάρτησης πάντα αποτιμούνται πριν περάσουν στη συνάρτηση, ακόμα και αν δεν χρησιμοποιούνται καθόλου.

Ας εξετάσουμε τώρα καθέναν από αυτούς τους κανόνες λεπτομερώς για να δούμε ποιες μορφές εκφράσεων χειρίζονται και πως ορίζουν την αποτίμησή τους.

Ο πρώτος κανόνας, (E-CON), χειρίζεται την αποτίμηση των σταθερών. Η εξωτερική δομή της σταθεράς, το όνομα και το μέγεθός της, δεν αλλάζει κατά την αποτίμησή της. Τα στοιχεία της σταθεράς αποτιμώνται από αριστερά προς τα δεξιά. Καθένα πρέπει να αποτιμηθεί σε μία τιμή πριν κινηθούμε στο επόμενο. Όταν όλα τα στοιχεία είναι τιμές, ή εάν το μέγεθος της σταθεράς είναι μηδέν, ολόκληρη η έκφραση σταθεράς είναι πια τιμή και η αποτίμησή της έχει ολοκληρωθεί.

Οι επόμενοι δύο κανόνες, (E-OP) και (E-OPF), χειρίζονται την αποτίμηση των τελεστών. Αρχικά, τα τελούμενα του τελεστή αποτιμώνται χρησιμοποιώντας τον κανόνα (E-OP). Ακριβώς όπως οι σταθερές, τα τελούμενα αποτιμώνται από αριστερά προς τα δεξιά και καθένα πρέπει να αποτιμηθεί σε

μία τιμή πριν κινηθούμε στο επόμενο. Όταν όλα τα τελούμενα είναι τιμές, ή εάν το πλήθος τους είναι μηδέν, ο κανόνας (E-OPF) μπορεί να χρησιμοποιηθεί ώστε να αποτιμήσουμε την έκφραση τελεστή σε μία τιμή, μέσω της αντίστοιχης σημασιολογικής συνάρτησης.

Οι κανόνες (E-APP1), (E-APP2), (E-FUNX), (E-FUNCM) και (E-FUNCN), όλοι χειρίζονται την αποτίμηση των εφαρμογών. Αρχικά, το αριστερό μέρος της εφαρμογής αποτιμάται σε μία τιμή μέσω του κανόνα (E-APP1). Ύστερα, ο κανόνας (E-APP2) χρησιμοποιείται για να αποτιμήσουμε και το δεξιό μέρος σε μία τιμή. Σε αυτό το σημείο, η αποτίμηση μπορεί να συνεχιστεί μόνο εάν έχουμε μία συνάρτηση, με τουλάχιστον μία ρήτρα, που εφαρμόζεται σε μία άλλη τιμή, την οποία θεωρούμε ως όρισμά της. Μέσω του κανόνα (E-FUNX), εάν η πρώτη ρήτρα της συνάρτησης έχει ένα πρότυπο μεταβλητής, μπορούμε να περάσουμε το όρισμα στη συνάρτηση καθώς ταιριάζει με το πρότυπο, αποτιμώντας την εφαρμογή στην έκφραση επιστροφής της ρήτρας, όπου έχουμε αντικαταστήσει τη μεταβλητή του προτύπου από το όρισμα της συνάρτησης. Παρομοίως, ο κανόνας (E-FUNCM) χειρίζεται την περίπτωση που η πρώτη ρήτρα έχει ένα πρότυπο σταθεράς που ταιριάζει με το όρισμα (σταθερά της ίδιας μορφής), αποτιμώντας την εφαρμογή στο αποτέλεσμα της (ταυτόχρονης) αντικατάστασης καθεμίας από τις μεταβλητές του προτύπου από την αντίστοιχη τιμή του στοιχείου του ορίσματος στην έκφραση επιστροφής της ρήτρας. Η μόνη εναπομείνουσα περίπτωση, όπου η πρώτη ρήτρα έχει ένα πρότυπο που δεν ταιριάζει με το όρισμα, χειρίζεται από τον κανόνα (E-FUNCN), προσπαθώντας να ταιριάξει το όρισμα στο πρότυπο της επόμενης ρήτρας (αν υπάρχει), αποτιμώντας την εφαρμογή σε μία νέα, όπου η συνάρτηση δίχως την πρώτη ρήτρα εφαρμόζεται στο ίδιο όρισμα. Για επανεξέταση του πότε ένα πρότυπο ταιριάζει με μία έκφραση δες τον ορισμό 2.1.1.

Τελευταίος αλλά εξίσου σημαντικός, ο κανόνας (E-FIX) χειρίζεται την αποτίμηση των σταθερών σημείων. Η συμπεριφορά ενός σταθερού σημείου μοιάζει με αυτή μίας συνάρτησης που έχει μία μόνο ρήτρα με πρότυπο μεταβλητής. Η διαφορά, βέβαια, είναι ότι, αντίθετα με τις συναρτήσεις, τα σταθερά σημεία δεν είναι τιμές, οπότε δεν χρειάζεται να εφαρμοστούν σε κάποιο όρισμα για να ξεκινήσει η αποτίμησή τους. Η σημασιολογία ενός σταθερού σημείου μπορεί να θεωρηθεί ισοδύναμη με την εφαρμογή της αντίστοιχης συνάρτησης στην ακριβή αρχική έκφραση σταθερού σημείου ως όρισμά της, η οποία περνιέται πάντα έμμεσα στη συνάρτηση προς αποτίμηση. Αυτή η κυκλική συμπεριφορά επιτρέπει τον ορισμό και την αποτίμηση αναδρομικών υπολογισμών. Όλη αυτή η διαδικασία για τον τελεστή σταθερού σημείου επιτυγχάνεται μέσω του κανόνα (E-FIX). Ένα σταθερό σημείο αποτιμάται πάντα (σε ένα βήμα) στο αποτέλεσμα της αντικατάστασης της μεταβλητής του από την ακριβή αρχική έκφραση στην έκφραση επιστροφής του.

2.2.3 Ντετερμινισμός Αποτίμησης και Κανονικές Μορφές

Θα καταγράψουμε τώρα μερικές ιδιότητες της σχέσης αποτίμησης και θα εισάγουμε κάποια κοινή ορολογία για να μπορούμε να αναφερθούμε σε αυτές ευκολότερα.

Παρατηρώντας προσεκτικά τους κανόνες συμπερασμού της σχέσης αποτίμησης, όπως κάναμε προηγουμένως, είναι σαφές ότι καθένας από αυτούς χρησιμοποιείται σε διαφορετικές εκφράσεις (καθορισμένη σειρά αποτίμησης). Αυτό σημαίνει ότι δεν υπάρχει κανένα σημείο κατά τη διάρκεια της αποτίμησης εκφράσεων όπου έχουμε επιλογή χρήσης μεταξύ κανόνων (ένας το πολύ διαθέσιμος κανόνας).

Διατυπώνουμε αυτή την ιδιότητα της σχέσης αποτίμησης με το ακόλουθο θεώρημα:

Θεώρημα (Ντετερμινισμός Μονοβηματικής Αποτίμησης)**.** *Εάν* $e \longrightarrow e'$ *και* $e \longrightarrow e''$*, τότε* $e' = e''$.

Η μονοβηματική σχέση αποτίμησης περιγράφει μεταβάσεις μεταξύ εκφράσεων κατά τη διάρκεια υπολογισμού. Μας ενδιαφέρουν όμως εξίσου τα τελικά αποτελέσματα των υπολογισμών, δηλαδή, οι καταστάσεις από τις οποίες η αφηρημένη μηχανή δε μπορεί να κάνει κανένα βήμα. Αυτές οι εκφράσεις, που έχουν ολοκληρώσει την αποτίμησή τους, λέμε ότι είναι σε κανονική μορφή (normal form). Θα εξετάσουμε τις μορφές τους πιο λεπτομερώς, καθώς αυτές κατηγοριοποιούν τις εκφράσεις βάσει της σημασίας τους.

Ορισμός (Κανονικές Μορφές)**.** Μία έκφραση e είναι σε κανονική μορφή εάν κανένας κανόνας αποτίμησης δε μπορεί να εφαρμοστεί σε αυτήν, δηλαδή, δεν υπάρχει έκφραση e' τέτοια ώστε $e \longrightarrow e'$.

Μερικές φορές, είναι χρήσιμο να συσχετίσουμε μία έκφραση με όλες τις εκφράσεις που μπορούν να παραχθούν από αυτήν κάνοντας μηδέν ή παραπάνω απλά βήματα αποτίμησης.

Ο ορισμός αυτής της πολυβηματικής σχέσης αποτίμησης, ακολουθεί:

Ορισμός (Πολυβηματική Σχέση Αποτίμησης)**.** Η πολυβηματική σχέση αποτίμησης στις εκφράσεις, $e \longrightarrow^* e'$, είναι το ανακλαστικό, μεταβατικό κλείσιμο της μονοβηματικής σχέσης αποτίμησης, $e \longrightarrow e'$.

Δηλαδή, είναι η ελάχιστη σχέση τέτοια ώστε:

- Εάν $e \longrightarrow e'$, τότε $e \longrightarrow^* e'$.
- Για κάθε έκφραση e, $e \longrightarrow^* e$.
- Εάν $e \longrightarrow^* e'$ και $e' \longrightarrow^* e''$, τότε $e \longrightarrow^* e''$.

Ύστερα από αυτούς τους ορισμούς μπορούμε να διατυπώσουμε το ακόλουθο θεώρημα:

Θεώρημα (Μοναδικότητα Κανονικών Μορφών)**.** *Εάν $e \longrightarrow^* e'$ και $e \longrightarrow^* e''$, όπου οι εκφράσεις e' και e'' είναι σε κανονική μορφή, τότε $e' = e''$.*

2.2.4 Κολλημένες Εκφράσεις

Το σύνολο των τιμών της γλώσσας μας παρουσιάστηκε ως ένα υποσύνολο των εκφράσεων, που περιέχει ακριβώς τις μορφές που δεχόμαστε ως έγκυρα τελικά υπολογισμένα αποτελέσματα των προγραμμάτων. Τώρα, έχοντας ορίσει τη σημασιολογία της γλώσσας μας μέσω της σχέσης αποτίμησης, θα πρέπει να είμαστε σε θέση να επιβεβαιώσουμε αυτή την αρχική εικόνα των τιμών. Μία τιμή έχει εξ ορισμού ολοκληρώσει την αποτίμησή της.

Το παρακάτω είναι απλά ένας έλεγχος ότι ο ορισμός των τιμών είναι συνεπής:

Θεώρημα. *Κάθε τιμή v είναι σε κανονική μορφή.*

Δεν είναι δύσκολο να διαπιστώσουμε ότι η αντίθετη κατεύθυνση αυτής της δήλωσης δεν είναι αληθής για τη γλώσσα μας. Μία έκφραση που δε μπορεί να συνεχίσει την αποτίμησή της δεν είναι κατ' ανάγκη μία τιμή. Για παράδειγμα, είδαμε προηγουμένως στην ανάλυση των κανόνων της σχέσης αποτίμησης ότι στην περίπτωση μίας εφαρμογής το αριστερό μέρος της θα πρέπει να αποτιμηθεί σε μία συνάρτηση ώστε να συνεχιστεί η αποτίμησή της. Η αποτίμηση τέτοιων εκφράσεων σταματά ξαφνικά χωρίς να πάρουμε μία τιμή ως αποτέλεσμα. Ονομάζουμε αυτές τις εκφράσεις κολλημένες (stuck).

Ορισμός (Κολλημένες Εκφράσεις)**.** Μία κλειστή έκφραση e είναι κολλημένη εάν είναι σε κανονική μορφή αλλά όχι τιμή.

Η ύπαρξη κολλημένων εκφράσεων δε σηματοδοτεί κάποιο πρόβλημα με τον ορισμό της σχέσης αποτίμησης. Σε κάποιους υπολογισμούς δε μπορεί να αποδοθεί καμμία σημασία, επειδή απλά δε βγάζουν νόημα. Μία κολλημένη έκφραση μπορεί να νοηθεί ως ανάλογη μίας κατάστασης που προκαλεί κάποιο σφάλμα κατά την εκτέλεση ενός προγράμματος (run-time error). Ένα τέτοιο πρόγραμμα εκτελεί μία μη καθορισμένη λειτουργία, που πιθανότατα δεν ήταν στην πρόθεση του προγραμματιστή, καταδεικνύοντας κάποιο λάθος στη λογική του.

Δείνουμε μία σύνοψη των διαφορετικών ειδών των (μικρότερων) κολλημένων εκφράσεων στη γλώσσα μας, περιγράφοντας τη σχετική τους κακή συμπεριφορά, στον ακόλουθο ορισμό:

Ορισμός (Μη Καθορισμένη Σημασιολογία)**.** Εκφράσεις που καταλήγουν κολλημένες και γιατί αυτό συμβαίνει.

- Εάν η αριστερή από δύο τιμές σε μία εφαρμογή δεν είναι συνάρτηση (δηλαδή είναι τιμή σταθεράς), τότε η έκφραση εφαρμογής είναι κολλημένη.

- Εάν οι τιμές τελουμένων $v_1, \ldots, v_n$ ενός τελεστή op δεν ανήκουν στο πεδίο ορισμού της σημασιολογικής του συνάρτησης $[\![op]\!]$, τότε η έκφραση $op\,(v_1, \ldots, v_n)$ είναι κολλημένη.

- Εάν καμμία ρήτρα μίας συνάρτησης δεν έχει ένα πρότυπο που ταιριάζει στη δοθείσα τιμή ορίσματός της v, τότε η προκύπτουσα έκφραση `fun end` v είναι κολλημένη.

2.2.5 Μη-Τερματισμός

Μία τελευταία σημείωση για την αποτίμηση εκφράσεων που χρειάζεται να κάνουμε είναι ότι υπάρχουν εκφράσεις που δεν μπορούν να αποτιμηθούν σε μία κανονική μορφή. Αυτό σημαίνει ότι κάποιες εκφράσεις μπορούν να εκτελούν βήματα αποτίμησης για πάντα. Φυσικά, αυτό δεν θα πρέπει να μας εκπλήσσει, καθώς η γλώσσα μας περιέχει έναν τελεστή σταθερού σημείου. Κάθε έκφραση σταθερού σημείου που χρησιμοποιεί τη μεταβλητή της χωρίς κάποια συνθήκη στην έκφραση επιστροφής της, περιγράφει έναν αναδρομικό υπολογισμό χωρίς μία περίπτωση βάσης, οπότε η αποτίμησή της δε μπορεί να τερματίσει.

Κεφάλαιο 3

Σύστημα Τύπων

Έχοντας ορίσει μία ενδιαφέρουσα γλώσσα, στην οποία μπορούν να περιγραφούν σύνθετοι υπολογισμοί, θέλουμε τώρα να εισάγουμε τύπους για αυτήν. Αυτό που θα προσπαθήσουμε να καταφέρουμε είναι η κατηγοριοποίηση των εκφράσεων της γλώσσας σύμφωνα με τη σημασία τους χωρίς να τις αποτιμήσουμε. Δηλαδή, θέλουμε να πάρουμε κάποια χρήσιμη πληροφορία για το είδος των τιμών που υπολογίζουν απλώς παρατηρώντας τη δομή τους. Ακολουθεί ο ορισμός ενός στατικού συστήματος τύπων με μία πλούσια άλγεβρα τύπων, που περιέχει συνολοθεωρητικούς κατασκευαστές τύπων όπως τύπους τομής και τύπους άρνησης, το οποίο, μέσω μίας συντακτικά ορισμένης σχέσης υποτύπων, συνδέει αυτούς τους τύπους με τις εκφράσεις της γλώσσας μας.

3.1 Τύποι

Έχουμε ήδη ορίσει τη σημασιολογία της γλώσσας μας μέσω μίας σχέσης αποτίμησης 2.3 η οποία περιγράφει ακριβώς πως μία έκφραση τελικώς αποτιμάται (ή όχι) σε μία τιμή. Αυτό που θέλουμε τώρα είναι να αποκτήσουμε μία γενικότερη αντίληψη της σημασίας των εκφράσεων που δεν απαιτεί καμμία αποτίμηση. Αυτή την ανάγκη έρχονται να καλύψουν οι τύποι (types). Μπορούμε να σκεφτούμε τους τύπους ως συγκεκριμένα υποσύνολα του συνόλου των τιμών 2.2 της γλώσσας μας. Σύμφωνα με αυτή την έννοια για τους τύπους, η δήλωση ότι μία έκφραση έχει κάποιο συγκεκριμένο τύπο (λέμε επίσης ότι η έκφραση ανήκει σε αυτόν τον τύπο), σημαίνει ότι, όταν η έκφραση αποτιμηθεί πλήρως (αν αυτό είναι εφικτό), έχει ως αποτέλεσμα μία τιμή που ανήκει στο σύνολο που περιγράφει αυτός ο τύπος.

3.1.1 Μορφές Τύπων

Με σκοπό την ανάθεση τύπων σε εκφράσεις, θα πρέπει πρώτα να ορίσουμε πως μοιάζουν οι τύποι του συστήματός μας. Εισάγουμε κάποιους βασικούς τύπους για να αντιστοιχηθούν σε συγκεκριμένα υποσύνολα από τιμές της γλώσσας αλλά και (περισσότερους από) αρκετούς συνολοθεωρητικούς κατασκευαστές τύπων που επιτρέπουν διάφορους συνδυασμούς από οποιουσδήποτε άλλους τύπους. Αυτή η επιλογή μας οδηγεί σε ένα αρκετά σύνθετο σύστημα τύπων, κάνοντας τη συλλογιστική των ιδιοτήτων του επίπονη (ή ακόμα και αδύνατη), αλλά μας παρέχει την εκφραστική δύναμη για να καταγράψουμε πολύ ακριβείς πληροφορίες για τους τύπους των εκφράσεων.

Το επιλεγμένο σύνολο από τύπους, T, ορίζεται μέσω της εκτεταμένης BNF γραμματικής που φαίνεται στο σχήμα 3.1. Χρησιμοποιούμε τις μεταβλητές τ και σ για οποιουσδήποτε τύπους.

Ας εξετάσουμε τώρα τις δυνατές μορφές τύπων, δίνοντας μία περιγραφή της επιθυμητής σημασίας για καθέναν. Για κάθε τύπο τ ακριβώς ένα εκ των παρακάτω είναι αληθές:

- Ο τύπος τ έχει τη μορφή any, τ = any. Ο τύπος any είναι ο μέγιστος τύπος στο σύστημά μας και μπορούμε να τον σκεφτούμε ως το σύνολο όλων των τιμών, V. Δηλαδή, κάθε έκφραση στην οποία μπορεί να αποδοθεί κάποιος τύπος θα πρέπει να έχει επίσης τύπο any.

- Ο τύπος τ έχει τη μορφή none, τ = none. Ο τύπος none είναι ο ελάχιστος τύπος στο σύστημά μας και μπορούμε να τον σκεφτούμε ως το κενό σύνολο, $\varnothing$. Δηλαδή, καμμία (κλειστή) έκφραση δεν θα πρέπει να έχει τύπο none.

- Ο τύπος τ έχει τη μορφή μίας σταθεράς (constant), $\tau = c\,(\tau_1, \ldots, \tau_n)$. Οι τύποι σταθεράς, όπως οι αντίστοιχες εκφράσεις, έχουν ένα δεδομένο όνομα και μέγεθος. Με εξαίρεση ότι περιέχουν επίσης την πληροφορία ενός ονόματος, συμπεριφέρονται ως (μεγέθους n) γινόμενα των τελούμενων τύπων τους. Φυσικά, περιγράφουν σύνολα από σταθερές τιμές που έχουν την κατάλληλη μορφή και τύπους στοιχείων. Και εδώ, το μέγεθος n μπορεί να είναι μηδέν, οπότε, σε αυτή την περίπτωση, ο τύπος σταθεράς είναι το μονοσύνολο (singleton) της τιμής σταθεράς με το ίδιο όνομα και δίχως στοιχεία.

- Ο τύπος τ έχει τη μορφή ενός βέλους (arrow), $\tau = \tau_1 \to \tau_2$. Οι τύποι βέλους περιγράφουν σύνολα από τιμές συναρτήσεων. Συγκεκριμένα, κατηγοριοποιούν εκείνες τις συναρτήσεις που δέχονται ορίσματα του αριστερού τύπου, τ_1, και επιστρέφουν αποτελέσματα του δεξιού τύπου, τ_2. Ο τύπος βέλους είναι δεξιά προσεταιριστικός, δηλαδή, ο τύπος $\tau_1 \to \tau_2 \to \tau_3$ είναι ο ίδιος με τον $\tau_1 \to (\tau_2 \to \tau_3)$.

- Ο τύπος τ έχει τη μορφή μίας τομής (intersection), $\tau = \tau_1 \wedge \tau_2$. Οι τύποι τομής περιγράφουν τις τιμές που ανήκουν και στους δύο τύπους τους, τ_1 και τ_2. Η προσθήκη των τύπων τομής είναι αυτή που κάνει το σύστημά μας τόσο εκφραστικό. Θα δούμε αργότερα ότι η χρήση τύπων τομής μας επιτρέπει μία αρκετά ακριβή περιγραφή του τύπου των συναρτήσεων με πολλαπλές ρήτρες, όπου οι ξεχωριστοί τύποι κάθε ρήτρας μπορούν να συνδυαστούν σε ένα μόνο τύπο χωρίς απώλεια πληροφορίας για τους αρχικούς τύπους. Ο τύπος τομής είναι προσεταιριστικός και αντιμεταθετικός (όπως μπορεί να δειχθεί και από μετέπειτα ορισμούς), οπότε συνήθως θα χειριζόμαστε εμφωλιασμένες τομές ως μία τομή πάνω σε όλους τους τύπους, συμβολίζοντας $\bigwedge_i (\tau_i)$.

- Ο τύπος τ έχει τη μορφή μίας ένωσης (union), $\tau = \tau_1 \vee \tau_2$. Οι τύποι ένωσης είναι η δυϊκή ιδέα των τύπων τομής και περιγράφουν τις τιμές που ανήκουν σε οποιονδήποτε από τους δύο τύπους τους, τ_1 ή τ_2. Δεν θα χρησιμοποιήσουμε τύπους ένωσης στον ορισμό της ανάθεσης τύπων αργότερα, αλλά είναι αρκετά χρήσιμοι στο παρασκήνιο. Η δοθείσα γραμματική των τύπων δεν φαίνεται να περιέχει κοινούς τύπους όπως `int` και `bool`, αλλά οι τελεστές της γλώσσας σίγουρα περιμένουν ορίσματα αυτών των τύπων. Ορισμοί βασικών τύπων (primitive types), με τη χρήση τύπων ένωσης (και υποτύπων, όπως θα δούμε μετά) σε μονοσύνολα σταθερών τύπων όπως ακέραιες και λογικές σταθερές, βοηθούν το κλείσιμο αυτών των κενών μεταξύ απαιτούμενων τύπων. Ακριβώς όπως με τις τομές, χειριζόμαστε εμφωλιασμένες ενώσεις ως μία πάνω σε πολλούς τύπους.

- Ο τύπος τ έχει τη μορφή μίας άρνησης (negation), $\tau = \neg\tau_p$. Πιθανότατα ο λιγότερο συνήθης κατασκευαστής τύπων στο σύστημά μας, οι τύποι άρνησης περιγράφουν τις τιμές που δεν ανήκουν στον τύπο τους, τ_p. Οι τύποι άρνησης είναι ιδιαίτερα χρήσιμοι σε συνδυασμό με τους τύπους τομής, ώστε να εκλεπτύνουμε τύπους με νέα πληροφορία σχετικά με μορφές τιμών που δεν περιέχουν, συνεπώς προσθέτοντας σε αυτόν μία τομή με την άρνηση του τύπου που περιγράφει αυτές τις μορφές τιμών. Θα δούμε αργότερα, ακριβώς πως χρησιμοποιούμε αυτή την τεχνική στην ανάθεση λεπτομερών τύπων σε συναρτήσεις με πολλαπλές ρήτρες.

- Ο τύπος τ έχει τη μορφή μίας μεταβλητής (variable), $\tau = \alpha$. Οι μεταβλητές τύπων αναφέρονται σε άλλους τύπους. Προς το παρόν, η μόνη έγκυρη χρήση μεταβλητών τύπων είναι στο σώμα αναδρομικών τύπων που τις δεσμεύουν. Χρησιμοποιούμε τη μεταβλητή α για μεταβλητές τύπων.

- Ο τύπος τ έχει τη μορφή ενός αναδρομικού (recursive) τύπου, $\tau = \mu\alpha\,.\,\tau_r$. Ένας αναδρομικός τύπος περιγράφει μία δενδρική δομή που μπορεί πάντα να επεκταθεί κάνοντας ένα ακόμη βήμα αναδίπλωσης (unfold) μέσω του αναδρομικού του ορισμού. Με αυτό τον τρόπο περιγράφεται ένας άπειρος τύπος που περιέχει όλες τις παραγόμενες μορφές τιμών. Οι αναδρομικοί τύποι δεν θα εμφανιστούν στον ορισμό μας για την ανάθεση τύπων, αλλά μπορούν να χρησιμοποιηθούν

για να ορίσουμε τους τύπους ενδιαφέροντων αναδρομικών δομών δεδομένων όπως λίστες, για να δουλεύουμε με αυτούς στη γλώσσα μας.

3.1.2 Περιβάλλοντα Τύπων

Μία ακόμη χρήσιμη έννοια, που μπορούμε να εισάγουμε σε αυτό το σημείο, είναι τα περιβάλλοντα τύπων (typing contexts). Ένα περιβάλλον τύπων καταγράφει τις τρέχουσες υποθέσεις μας σχετικά με τους τύπους των ελεύθερων μεταβλητών σε μία έκφραση τη στιγμή που προσπαθούμε να της αποδώσουμε κάποιο τύπο. Δηλαδή, ένα περιβάλλον τύπων είναι μία (πιθανώς κενή) ακολουθία από αντιστοιχίσεις, ($x : \tau$), μεταξύ (ελεύθερων) μεταβλητών και τύπων που υποθέτουμε για αυτές, που συνοψίζει τη γνώση μας για τις μεταβλητές ενώ ψάχνουμε τον τύπο μίας μεγαλύτερης έκφρασης. Η απλή σύνταξη των περιβαλλόντων τύπων φαίνεται στο σχήμα 3.2. Χρησιμοποιούμε τις μεταβλητές Γ και Δ για περιβάλλοντα τύπων.

Όταν συναντάμε μία έκφραση που εισάγει μία νέα δεσμευμένη μεταβλητή, θα πρέπει να κάνουμε μία υπόθεση για τον τύπο της μεταβλητής, ανανεώνοντας το τρέχον περιβάλλον τύπων. Μία λεπτομέρεια σχετικά με την προσθήκη νέων αντιστοιχίσεων σε ένα περιβάλλον τύπων είναι ότι, φυσικά, δεν έχει νόημα να προσθέσουμε μία εγγραφή για μία μεταβλητή, το όνομα της οποίας ήδη εμφανίζεται σε αυτό. Απαιτούμε, λοιπόν, ότι το όνομα της μεταβλητής είναι ξεχωριστό και ικανοποιούμε αυτή την απαίτηση με τον ίδιο τρόπο όπως στην (ασφαλή) αντικατάσταση εκφράσεων.

3.2 Υποτύποι

Έχοντας ορίσει τη σύνταξη των τύπων στο σύστημά μας, θα μπορούσαμε τώρα να προχωρήσουμε διατυπώνοντας πως αυτοί οι τύποι κατηγοριοποιούν τις εκφράσεις της γλώσσας βάσει των τιμών τους. Αλλά, υπάρχει ένα πρόβλημα με τους τύπους μας που θα πρέπει να διευθετήσουμε πρώτα. Η πλούσια άλγεβρα τύπων μας επιτρέπει να αποδώσουμε πολύ ακριβείς τύπους σε εκφράσεις και, φυσικά, αυτός ακριβώς είναι ο σκοπός μας. Ωστόσο, πρέπει ακόμα να εξασφαλίσουμε ότι αρκετές εκφράσεις μπορούν πράγματι να αποκτήσουν κάποιο τύπο, δηλαδή, ότι (έγκυροι) τύποι ορισμάτων θα πρέπει να είναι ικανοί να ταιριάξουν με τους τύπους των πεδίων ορισμού των συναρτήσεων που εφαρμόζονται σε αυτά. Όπως κανείς μπορεί να φανταστεί, πιο ακριβείς τύποι σημαίνει ότι οι τύποι δεν μπορούν (σχεδόν) ποτέ να ταιριάξουν ακριβώς μεταξύ τους. Αυτό που μας λείπει εδώ είναι η ικανότητα να επιτρέπουμε σε εκφράσεις ενός τύπου να χάνουν ελεύθερα μέρος της πληροφορίας τους και να αποκτούν επίσης άλλους, πιο γενικούς, τύπους. Αυτό σημαίνει ότι χρειαζόμαστε ένα άλλο επίπεδο ορισμών, στο οποίο θα βασιζόμαστε για να αποδίδουμε τύπους σε εκφράσεις, που ονομάζεται σύστημα υποτύπων (subtyping) και μας πληροφορεί για το ποιοι τύποι είναι απλά πιο ακριβείς εκδοχές κάποιων άλλων δεδομένων τύπων.

3.2.1 Σχέση Υποτύπων

Θα ορίσουμε τώρα, όπως αναφέραμε παραπάνω, μία διμελή σχέση στο σύνολο T των τύπων μας 3.1, που ονομάζεται σχέση υποτύπων (subtyping relation). Λέμε ότι ο τύπος τ είναι ένας υποτύπος (subtype) του τύπου τ' (ή τ' είναι ένας υπερτύπος (supertype) του τ) και συμβολίζουμε $\tau <: \tau'$. Η διαίσθηση πίσω από τη δήλωση $\tau <: \tau'$ είναι ότι ο υποτύπος, τ, παρέχει περισσότερη πληροφορία από τον υπερτύπο του, τ', συνεπώς μπορούμε να χρησιμοποιήσουμε με ασφάλεια οποιαδήποτε έκφραση τύπου τ σε κάποιο σημείο όπου αναμένεται μία έκφραση τύπου τ'.

Η σχέση υποτύπων διατυπώνεται μέσω ενός συνόλου από συμπερασματικούς κανόνες και παρουσιάζεται στα σχήματα 3.3 και 3.4.

Θα εξετάσουμε τώρα καθέναν από αυτούς τους κανόνες με τη σειρά, δίνοντας κάποια σύντομη εξήγηση του τι προσθέτουν στη σχέση υποτύπων.

Αρχικά, ο κανόνας (S-REFL) μετατρέπει τη σχέση υποτύπων σε ανακλαστική και ο κανόνας (S-TRANS) αντίστοιχα σε μεταβατική. Και οι δύο αυτές ιδιότητες έπονται άμεσα από τη διαίσθησή μας για τη σχέση υποτύπων.

Έπειτα, ο κανόνας (S-ANY) θέτει κάθε τύπο ως έναν υποτύπο του any, δηλαδή, ο τύπος any είναι ο μέγιστος τύπος στο σύστημά μας. Παρομοίως, ο κανόνας (S-NONE) θέτει κάθε τύπο ως έναν υπερτύπο του none, δηλαδή, ο τύπος none είναι ο ελάχιστος τύπος στο σύστημά μας.

Ο κανόνας (S-CON) χειρίζεται τη σχέση υποτύπων μεταξύ δύο σταθερών τύπων. Ένας σταθερός τύπος μπορεί να είναι υποτύπος ενός άλλου, μόνο εάν τα ονόματα και τα μεγέθη τους ταιριάζουν. Μετά, θα πρέπει επίσης, η σχέση υποτύπου μεταξύ καθενός από τους αντίστοιχους τύπους που περιέχουν να έχει την ίδια κατεύθυνση (covariant) με αυτή στο συμπέρασμα του κανόνα. Έτσι επιτυγχάνεται η επιθυμητή συμπεριφορά υποτύπων για τους τύπους γινομένου.

Ο κανόνας (S-ARROW) χειρίζεται τη σχέση υποτύπων μεταξύ δύο τύπων βέλους. Υπενθυμίζουμε ότι οι τύποι βέλους κατηγοριοποιούν συναρτήσεις οι οποίες αναμένουν ορίσματα του αριστερού τους τύπου και επιστρέφουν αποτελέσματα του δεξιού τους τύπου. Παρατηρούμε ότι η κατεύθυνση της σχέσης υποτύπου είναι ανεστραμμένη (contravariant) για τους τύπους των ορισμάτων ενώ έχει την ίδια κατεύθυνση (covariant) για τους τύπους των αποτελεσμάτων όπως και για τους τύπους των συναρτήσεων. Η ιδέα εδώ είναι πως, μία συνάρτηση μπορεί, επίσης, να θεωρηθεί, ασφαλώς, ότι δέχεται μόνο ένα υποσύνολο των πραγματικών ορισμάτων της και ότι παράγει ένα υπερσύνολο των πραγματικών αποτελεσμάτων της.

Οι επόμενοι δύο κανόνες, (S-INTER1) και (S-INTER2), αποτυπώνουν την αντίληψή μας σχετικά με τους τύπους τομής. Σύμφωνα με αυτούς, οι εκφράσεις που έχουν τον τύπο τομής θα πρέπει να έχουν, επίσης, καθέναν από τους αναφερόμενους τύπους σε αυτόν και αντιστρόφως. Πολλαπλές χρήσεις αυτών των κανόνων μπορούν να συνδυαστούν σε παραγωγές της σχέσης υποτύπων παίρνοντας τους αντίστοιχους πιο γενικούς (αλλά ισοδύναμους) κανόνες για τομές από πολλούς τύπους.

Παρομοίως, οι κανόνες (S-UNION1) και (S-UNION2) αποτυπώνουν την αντίληψή μας για τη δυϊκή έννοια των τύπων ένωσης. Σύμφωνα με αυτούς, οι εκφράσεις που έχουν τον τύπο ένωσης θα πρέπει να έχουν, επίσης, κάποιον από τους αναφερόμενους τύπους σε αυτόν και αντιστρόφως.

Ο κανόνας (S-NEG), ο γενικός μας κανόνας για τους τύπους άρνησης, ορίζει ότι οι τύποι άρνησης αναστρέφουν την κατεύθυνση της σχέσης υποτύπων (contravariant), όπως θα περιμέναμε από την αντίληψή μας για αυτούς.

Δύο επιπλέον κανόνες, οι (S-NEGCA) και (S-NEGCC), εμπεριέχουν τύπους άρνησης αλλά σε συνδυασμό με τύπους σταθεράς και τύπους βέλους. Ο πρώτος αντιμετωπίζει κάθε τύπο σταθεράς ως διακεκριμένο από κάθε τύπο βέλους. Ο δεύτερος αντιμετωπίζει, επίσης, ως διακεκριμένους τους τύπους σταθεράς που είτε δε συμφωνούν στην εξωτερική τους δομή (όνομα και μέγεθος) είτε περιέχουν τουλάχιστον ένα ζευγάρι από διακεκριμένους τύπους μεταξύ των αντίστοιχων τύπων τους.

Τέλος, ο κανόνας (S-REC) ορίζει ότι οι αναδρομικοί τύποι είναι ισοδύναμοι με τις αντίστοιχες εκτεταμένες μορφές τους (equi-recursive).

3.3 Ανάθεση Τύπων

Έχοντας μία σχέση υποτύπων έτοιμη να χρησιμοποιηθεί, ήρθε, επιτέλους, η ώρα να διατυπώσουμε πως, ακριβώς, οι τύποι μας μπορούν να ανατεθούν σε εκφράσεις της γλώσσας μας, ώστε να τις κατηγοριοποιήσουν σύμφωνα με τις μορφές τιμών στις οποίες αποτιμώνται, χωρίς, φυσικά, να τις αποτιμήσουμε. Θα δούμε πως η ανάθεση τύπων χρησιμοποιεί τις πληροφορίες για τους τύπους που καταγράφηκαν μέσω της σχέσης υποτύπων έτσι ώστε να εκλεπτύνει τους δικούς της κανόνες, που θα παρουσιάσουμε σύντομα, επιτυγχάνοντας το ταίριασμα (έγκυρων) τύπων ορισμάτων με τους τύπους των πεδίων ορισμού των συναρτήσεων που εφαρμόζονται σε αυτά. Επιπλέον, πρόκειται να αποκαλύψουμε, επιτέλους, πως οι τύποι τομής και οι τύποι άρνησης μας επιτρέπουν να αποδώσουμε αρκετά ακριβείς τύπους σε συναρτήσεις με πολλαπλές ρήτρες, χρησιμοποιώντας την πληροφορία που παρέχεται μέσω των προτύπων των ρητρών τους.

3.3.1 Σχέση Ανάθεσης Τύπων

Για ακόμη μία φορά, θα χρειαστεί να ορίσουμε επαγωγικά μία νέα σχέση μέσω ενός συνόλου κανόνων συμπερασμού και αξιωμάτων. Η σχέση ανάθεσης τύπων (typing relation) είναι μία τριμελής σχέση που συσχετίζει τρία διαφορετικά είδη συνόλων, συγκεκριμένα, τα περιβάλλοντα τύπων 3.2, τις εκφράσεις της γλώσσας μας 2.1 και τους τύπους του συστήματός μας 3.1. Συμβολίζουμε $\Gamma \vdash e : \tau$ για τις προτάσεις που παράγονται από τους κανόνες της σχέσης ανάθεσης τύπων, υποδηλώνοντας ότι ο τύπος τ μπορεί να ανατεθεί στην έκφραση e υπό τις υποθέσεις, για τους τύπους ελεύθερων μεταβλητών στην έκφραση e, που καταγράφονται από το περιβάλλον τύπων Γ. Η σημασία της δήλωσης $\Gamma \vdash e : \tau$ είναι ότι μπορούμε να συμπεράνουμε ότι η έκφραση e, όταν (και αν) ολοκληρώσει την αποτίμησή της, έχει ως αποτέλεσμα μία τιμή του τύπου τ, δεδομένου ότι οποιεσδήποτε ελεύθερες μεταβλητές της e αντικαθίστανται από κάποιες τιμές του τύπου που θεωρούμε για αυτές σύμφωνα με το περιβάλλον Γ.

Το σύνολο των κανόνων συμπερασμού που ορίζει τη σχέση ανάθεσης τύπων παρουσιάζεται στο σχήμα 3.5.

Θα εξετάσουμε τώρα καθέναν από αυτούς τους κανόνες με τη σειρά, αναλύοντας σε ποιες μορφές εκφράσεων αναθέτουν τύπους και υπό ποιες προϋποθέσεις.

Αρχικά, ο κανόνας (T-VAR) αναθέτει τύπους σε μεταβλητές. Ο μόνος τρόπος με τον οποίο μπορούμε να δώσουμε κάποιο τύπο σε μία μεταβλητή είναι να έχουμε ήδη υποθέσει έναν τύπο για αυτήν νωρίτερα. Δηλαδή, μία μεταβλητή έχει έναν τύπο μόνο εάν υπάρχει μία εγγραφή για αυτήν στο τρέχον περιβάλλον τύπων.

Ο κανόνας (T-CON) αναθέτει τύπους σε σταθερές εκφράσεις. Εάν καθένα από τα στοιχεία της σταθεράς αποτιμάται σε μία τιμή κάποιου τύπου (υπό την υπόθεση ότι οι τιμές που αναπαριστώνται από τις ελεύθερες μεταβλητές του έχουν τους τύπους που δίνονται για αυτές στο Γ), τότε η έκφραση σταθεράς αποτιμάται σε μία τιμή του τύπου σταθεράς με το ίδιο όνομα και μέγεθος που περιέχει τους αντίστοιχους τύπους των στοιχείων (υπό τις ίδιες υποθέσεις για τις μεταβλητές της, Γ).

Παρομοίως, ο κανόνας (T-OP) αναθέτει τύπους σε εκφράσεις τελεστών. Εάν γνωρίζουμε τον τύπο της συνάρτησης του τελεστή, το αποτέλεσμα της εφαρμογής του σε ένα πλήθος τιμών, όπου η καθεμία είναι το αποτέλεσμα του αντίστοιχού του τελουμένου, θα πρέπει να είναι μία τιμή του τύπου του πεδίου τιμών του, αλλά μόνο εάν αυτές οι τιμές ανήκουν στον τύπο του πεδίου ορισμού του (συμπεριλαμβανομένου του πλήθους τους).

Ο κανόνας (T-FUN0) ορίζει απλά τον τύπο που αναθέτουμε στη συνάρτηση που δεν έχει καμμία ρήτρα. Αυτή η συνάρτηση δε δέχεται καμμία τιμή οπότε ο τύπος ορίσματός της θα πρέπει να είναι none. Ο τύπος επιστροφής αυτής της συνάρτησης δε μας ενδιαφέρει, αλλά η επιλογή του τύπου none $\rightarrow$ any για αυτήν τη συνάρτηση είναι βολική, καθώς είναι ο μέγιστος τύπος βέλους, σημειώνοντας ότι κάθε συνάρτηση έχει (τουλάχιστον) τη συμπεριφορά αυτής της συνάρτησης.

Συναρτήσεις που ξεκινούν με μία ρήτρα με πρότυπο μεταβλητής παίρνουν τύπους μέσω του κανόνα (T-FUNX). Δίνουμε σε αυτές τις συναρτήσεις τύπους βέλους σημειώνοντας τον τύπο των τιμών που δέχονται ως ορίσματα και τον τύπο των τιμών που επιστρέφουν ως αποτελέσματα. Σε αυτή την περίπτωση, εάν υποθέτοντας κάποιον τύπο τιμών για τη μεταβλητή της συνάρτησης καταφέρουμε να δείξουμε ότι η έκφραση επιστροφής της μπορεί να αποτιμηθεί σε μία τιμή κάποιου τύπου, τότε, σαφώς, ο πρώτος τύπος είναι ένας πιθανός τύπος ορίσματος ενώ ο δεύτερος ένας πιθανός τύπος αποτελέσματος για τη συνάρτηση, οπότε μπορούμε να αναθέσουμε τον κατάλληλο τύπο βέλους στη συνάρτηση. Επειδή το πρότυπο μεταβλητής της πρώτης ρήτρας θα ταιριάξει με οποιαδήποτε μορφή τιμής ως όρισμα, όλες οι υπόλοιπες ρήτρες της συνάρτησης δεν επηρεάζουν τη λειτουργία της, οπότε μπορούμε να τις αγνοήσουμε.

Αντίθετα, ο κανόνας (T-FUNC) αναθέτει τύπους σε συναρτήσεις που ξεκινούν με μία ρήτρα με πρότυπο σταθεράς. Αυτός ο κανόνας για τους τύπους των συναρτήσεων επιδεικνύει τη βασική μας ιδέα για το πως να δώσουμε ακριβείς τύπους σε συναρτήσεις και χρησιμοποιεί τύπους τομής και τύπους άρνησης για να το επιτύχει, οπότε έχουμε να αναλύσουμε κάποια πράγματα εδώ. Αρχικά, η αριστερή προϋπόθεση του κανόνα είναι μία γενικευμένη περίπτωση του προτύπου μεταβλητής που είδαμε πριν. Εάν υποθέτοντας κάποιο συνδυασμό από τύπους για τις μεταβλητές του προτύπου στα-

θεράς, η έκφραση επιστροφής έχει κάποιο τύπο, τότε έχουμε έναν κατάλληλο τύπο βέλους για την πρώτη ρήτρα. Τώρα, ας προσέξουμε ότι η δεξιά προϋπόθεση του κανόνα περιγράφει τη λειτουργία της συνάρτησης δίχως την πρώτη ρήτρα ως μία τομή από τύπους βέλους. Δηλαδή, αυτή η συνάρτηση μπορεί να συμπεριφερθεί ως οποιοσδήποτε από αυτούς τους τύπους βέλους που αντιστοιχούν στις άλλες ρήτρες, δεχόμενη τιμές του αριστερού τύπου και επιστρέφοντας τιμές του δεξιού τύπου. Η συνάρτηση στο συμπέρασμα του κανόνα θα πρέπει να συνδυάσει αυτές τις λειτουργίες κατάλληλα. Σίγουρα, πρέπει να έχει τη λειτουργία που περιγράφεται από τον τύπο της πρώτης ρήτρας της. Αλλά, επειδή οι τιμές που ταιριάζουν στο πρότυπο αυτής της ρήτρας δεν περνούν στις υπόλοιπες πια, κάθε τύπος ορίσματος των άλλων ρητρών πρέπει να εκλεπτυνθεί. Αυτό επιτυγχάνεται μέσω της τομής καθενός από αυτούς τους τύπους ορισμάτων με την άρνηση του γενικότερου τύπου που περιγράφει κάθε τιμή που ταιριάζει με το πρότυπο της πρώτης ρήτρας. Συνεπώς, ο τύπος της όλης συνάρτησης είναι μία τομή του τύπου βέλους της πρώτης ρήτρας και όλων των άλλων τύπων βέλους όπου οι τύποι ορισμάτων ανανεώνονται όπως περιγράψαμε. Με αυτό τον τρόπο, οι τύποι ορισμάτων όλων των τύπων βέλους στην τομή κρατούνται πάντα διακεκριμένοι μεταξύ τους, οπότε η σειρά των τύπων στην τομή δεν έχει σημασία, όπως είναι επιθυμητό.

Η ανάθεση τύπων σε εκφράσεις εφαρμογών γίνεται μέσω του κανόνα (T-APP). Εάν η αριστερή έκφραση αποτιμάται σε μία τιμή ενός τύπου βέλους, δηλαδή σε μία συνάρτηση, τότε η δεξιά έκφραση πρέπει να αποτιμηθεί σε μία τιμή του τύπου ορίσματος της συνάρτησης, έτσι ώστε η έκφραση εφαρμογής να δώσει ως αποτέλεσμα μία τιμή του τύπου επιστροφής της συνάρτησης.

Ο κανόνας (T-FIX) αναθέτει τύπους σε εκφράσεις σταθερού σημείου. Θυμήσου την αναδρομική συμπεριφορά των σταθερών σημείων. Μία τέτοια έκφραση μπορεί να επιστρέψει μία τιμή κάποιου τύπου, μόνο εάν η έκφραση επιστροφής της επιστρέφει μία τιμή του ίδιου τύπου, υποθέτοντας ότι η μεταβλητή της αντικαθίσταται από κάποια τιμή του ίδιου, επίσης, τύπου. Σημειώνουμε ότι αυτός ο τύπος δεν χρειάζεται να αντιστοιχεί σε συναρτήσεις, αλλά συνήθως λογική χρήση των σταθερών σημείων σημαίνει ότι αυτός ο τύπος είναι τύπος βέλους, δηλαδή, έχουμε ορισμό αναδρομικών συναρτήσεων. Επίσης, το γεγονός ότι μία έκφραση σταθερού σημείου έχει κάποιο τύπο, δε σημαίνει ότι η αποτίμησή της τερματίζει.

Τελευταίος, αλλά ίσως ο σημαντικότερος, ο κανόνας (T-SUB), γνωστός και ως subsumption, είναι ο συνδετικός κρίκος μεταξύ της σχέσης ανάθεσης τύπων και της σχέσης υποτύπων, εκλεπτύνοντας τις προϋποθέσεις όλων των προηγούμενων κανόνων ανάθεσης τύπων, έτσι ώστε οι τύποι να μην χρειάζεται να ταιριάζουν ακριβώς. Αυτός ο κανόνας επιτρέπει σε οποιαδήποτε έκφραση κάποιου τύπου να έχει, επίσης, οποιονδήποτε υπερτύπο αυτού του τύπου.

Κεφάλαιο 4

Μεταθεωρία

Έχοντας μόλις ολοκληρώσει τους ορισμούς του συστήματος τύπων μας, θα θέλαμε να ελέγξουμε ότι είναι συνεπές και ενδεχομένως χρήσιμο. Δηλαδή, ότι αναθέτει τους κατάλληλους τύπους στις εκφράσεις και, συγκεκριμένα, ότι δεν αναθέτει κανέναν τύπο σε εκφράσεις που οδηγούν σε μη καθορισμένη σημασιολογία. Με αυτό τον τρόπο, το σύστημα τύπων μπορεί να χρησιμοποιηθεί για τη στατική αναφορά σφαλμάτων σε προγράμματα προτού τα συναντήσουμε κατά τον χρόνο εκτέλεσης. Παρακάτω, δείχνουμε ότι το σύστημα τύπων μας έχει αυτή την ιδιότητα, που ονομάζεται ασφάλεια τύπων, δίνοντας σχολαστικές αποδείξεις των γνωστών θεωρημάτων της προόδου και της διατήρησης.

4.1 Ασφάλεια Τύπων

Η πιο βασική ιδιότητα ενός συστήματος τύπων, όπως αυτό που παρουσιάσαμε, είναι η ασφάλεια (safety) τύπων (ή ορθότητα (soundness)). Σύμφωνα με αυτή την ιδιότητα, ορθοτυπημένα (well-typed) προγράμματα, δηλαδή, εκφράσεις στις οποίες το σύστημα τύπων μπορεί να αναθέσει κάποιο τύπο, δεν οδηγούν σε σφάλματα. Έχουμε ήδη επιλέξει πως να διατυπώσουμε την κακή συμπεριφορά των προγραμμάτων, ως φτάνοντας σε μία κολλημένη (stuck) κατάσταση, μία έκφραση που δεν είναι μία τελική τιμή αλλά όπου οι κανόνες αποτίμησης δεν ορίζουν κάποιο επόμενο βήμα. Αυτό που θέλουμε να γνωρίζουμε, λοιπόν, είναι ότι οι ορθοτυπημένες εκφράσεις δεν καταλήγουν κολλημένες. Δείχνουμε αυτή την ιδιότητα σε δύο βήματα, γνωστά ως τα θεωρήματα της προόδου (progress) και της διατήρησης (preservation).

- Πρόοδος: Μία ορθοτυπημένη έκφραση δεν είναι κολλημένη (είτε είναι μία τιμή είτε μπορεί να κάνει ένα βήμα σύμφωνα με τους κανόνες αποτίμησης).
- Διατήρηση: Εάν μία ορθοτυπημένη έκφραση κάνει ένα βήμα αποτίμησης, τότε η επακόλουθη έκφραση είναι επίσης ορθοτυπημένη.

Αυτές οι δύο ιδιότητες μαζί συνεπάγονται ότι μία ορθοτυπημένη έκφραση δεν μπορεί ποτέ να φτάσει σε μία κολλημένη κατάσταση κατά την αποτίμησή της.

Παρακάτω ακολουθούν οι αποδείξεις των θεωρημάτων της προόδου και της διατήρησης, μαζί με κάποια λήμματα που χρησιμοποιούνται σε αυτές, για τη γλώσσα και το σύστημα τύπων που ορίσαμε.

4.2 Πρόοδος

Ας ξεκινήσουμε με το θεώρημα της προόδου και τα λήμματα που χρειαζόμαστε για την απόδειξή του.

4.2.1 Αντιστροφή Σχέσης Υποτύπων

Αρχικά, καταγράφουμε κάποιες ιδιότητες σχετικά με το πως σχηματίζονται οι παραγωγές της σχέσης υποτύπων. Το λήμμα της αντιστροφής της σχέσης υποτύπων, δοθείσας μίας συγκεκριμένης σχέσης υποτύπου, περιγράφει τι μπορούμε να πούμε σχετικά με τις μορφές των τύπων που παίρνουν

μέρος σε αυτήν. Η δομή αυτού του λήμματος είναι σαν ένας αλγόριθμος με ένα πλήθος περιπτώσεων που ελέγχονται σειριακά για να δούμε πως μοιάζει η δοθείσα σχέση υποτύπου και να αποφασίσουμε τι μορφές τύπων είναι δυνατές σε αυτή την περίπτωση.

Λήμμα (Αντιστροφή Σχέσης Υποτύπων)**.** *Δυνατές μορφές τύπων βάσει μίας δεδομένης σχέσης υποτύπου.*

Σε κάθε περίπτωση εξετάζουμε μόνο τους συνδυασμούς μορφών τύπων στη σχέση υποτύπου που δεν έχουν ήδη εξεταστεί από κάποια προηγούμενη περίπτωση.

1. - *Εάν* $\tau <: \mu\alpha\,.\,\tau_r$*, τότε* $\tau <: \tau_r\,[\alpha \mapsto \mu\alpha\,.\,\tau_r]$.
   - *Εάν* $\mu\alpha\,.\,\tau_r <: \tau$*, τότε* $\tau_r\,[\alpha \mapsto \mu\alpha\,.\,\tau_r] <: \tau$.
2. - *Εάν* $\tau <: \tau_1 \wedge \tau_2$*, τότε* $\tau <: \tau_1$ *και* $\tau <: \tau_2$.
   - *Εάν* $\tau_1 \vee \tau_2 <: \tau$*, τότε* $\tau_1 <: \tau$ *και* $\tau_2 <: \tau$.
3. - *Εάν* $\tau_1 \wedge \tau_2 <: \tau$*, τότε* $\tau_1 <: \tau$ *ή* $\tau_2 <: \tau$
     ή $\tau = \tau_1' \vee \tau_2'$ *και* $\tau_1 \wedge \tau_2 <: \tau_1'$ *ή* $\tau_1 \wedge \tau_2 <: \tau_2'$.
   - *Εάν* $\tau <: \tau_1 \vee \tau_2$*, τότε* $\tau <: \tau_1$ *ή* $\tau <: \tau_2$
     ή $\tau = \tau_1' \wedge \tau_2'$ *και* $\tau_1' <: \tau_1 \vee \tau_2$ *ή* $\tau_2' <: \tau_1 \vee \tau_2$.
4. - *Εάν* $\tau <: \mathsf{any}$*, τότε* τ *μπορεί να έχει οποιαδήποτε μορφή.*
   - *Εάν* $\mathsf{none} <: \tau$*, τότε* τ *μπορεί να έχει οποιαδήποτε μορφή.*
   - *Εάν* $\mathsf{any} <: \tau$*, τότε* $\tau = \mathsf{any}$.
   - *Εάν* $\tau <: \mathsf{none}$*, τότε* $\tau = \mathsf{none}$.
5. - *Εάν* $\neg\tau_p <: \tau$*, τότε* $\tau = \neg\tau'$ *και* $\tau' <: \tau_p$.
   - *Εάν* $\tau <: \neg\tau_p$*, τότε* $\tau = \neg\tau'$ *και* $\tau_p <: \tau'$*,*
     ή $\tau = c\,(\tau_1, \ldots, \tau_n)$ *και* $\tau_p <: \tau_a \to \tau_a'$*,*
     ή $\tau = \tau_a \to \tau_a'$ *και* $\tau_p <: c\,(\tau_1, \ldots, \tau_n)$*,*
     ή $\tau = c_1\,(\tau_1, \ldots, \tau_n)$ *και* $\tau_p <: c_2\,(\tau_1', \ldots, \tau_m')$
     και $c_1 \neq c_2$ *ή* $n \neq m$ *ή* $\tau_i <: \neg\tau_i'$ *για κάποιο* i.
6. - *Εάν* $\tau <: c\,(\tau_1, \ldots, \tau_n)$*, τότε* $\tau = c\,(\tau_1', \ldots, \tau_n')$ *και* $\tau_i' <: \tau_i$ *για κάθε* i.
   - *Εάν* $\tau <: \tau_1 \to \tau_2$*, τότε* $\tau = \tau_1' \to \tau_2'$ *και* $\tau_1 <: \tau_1'$ *και* $\tau_2' <: \tau_2$.

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.2.1.

□

4.2.2 Αναγνωριστικές Μορφές

Χρησιμοποιώντας το λήμμα της αντιστροφής της σχέσης υποτύπων, μπορούμε τώρα να καταγράψουμε μερικά γεγονότα σχετικά με τις δυνατές δομές των αναγνωριστικών μορφών (canonical forms) κάποιων συγκεκριμένων τύπων. Το ακόλουθο λήμμα περιγράφει ποιες είναι οι δυνατές μορφές των ορθοτυπημένων τιμών βάσει της μορφής του τύπου τους.

Λήμμα (Αναγνωριστικές Μορφές)**.** *Μορφές κλειστών και ορθοτυπημένων τιμών (*$\varnothing \vdash v : \tau$*).*

1. *Εάν* v *είναι μία κλειστή τιμή με τύπο* $c\,(\tau_1, \ldots, \tau_n)$ *(*$\varnothing \vdash v : c\,(\tau_1, \ldots, \tau_n)$*),*
   τότε v *έχει τη μορφή* $c\,(v_1, \ldots, v_n)$.
2. *Εάν* v *είναι μία κλειστή τιμή με τύπο* $\tau_1 \to \tau_2$ *(*$\varnothing \vdash v : \tau_1 \to \tau_2$*),*
   τότε v *έχει τη μορφή* $\mathtt{fun}\,d_1 \mid \ldots \mid d_m \mathtt{end}$.

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.2.2.

□

Ένα πόρισμα του λήμματος των αναγνωριστικών μορφών, δηλώνει ότι ο τύπος none είναι κενός. Αυτή ήταν όντως η επιθυμητή σημασία για τον τύπο none, και αυτό το γεγονός θα φανεί χρήσιμο έτσι ώστε να αποκλείσουμε κάποιες περιπτώσεις σε άλλα θεωρήματα που δεν μπορούν να προκύψουν επειδή κάποια τιμή πρέπει να έχει τύπο none.

Λήμμα. *Δεν υπάρχουν κλειστές τιμές με τύπο* none. *(*$\varnothing \vdash v$: none *δεν μπορεί να συμβεί για καμμία τιμή* v*)*

4.2.3 Πρόοδος

Με τη βοήθεια αυτών των λημμάτων μπορούμε τώρα να αποδείξουμε το θεώρημα της προόδου. Το θεώρημα της προόδου δηλώνει ότι μία ορθοτυπημένη έκφραση δεν είναι (προς το παρόν) κολλημένη, δηλαδή, είτε είναι μία τιμή, είτε μπορεί να κάνει ένα βήμα αποτίμησης. Μία μικρή λεπτομέρεια είναι ότι ενδιαφερόμαστε μονάχα για κλειστές εκφράσεις, δίχως ελεύθερες μεταβλητές. Για ανοιχτές εκφράσεις, το θεώρημα αποτυγχάνει, καθώς οι μεταβλητές δεν αποτιμούνται και δεν είναι τιμές. Παρόλα αυτά, αυτή η αποτυχία δεν αναπαριστά κάποιο πρόβλημα με τη γλώσσα, καθώς τα πλήρη προγράμματα, που είναι οι εκφράσεις τις οποίες νοιαζόμαστε να αποτιμήσουμε, είναι πάντα κλειστά.

Θεώρημα (Πρόοδος)**.** *Εάν* e *είναι κλειστή και ορθοτυπημένη (*$\varnothing \vdash e : \tau$ *για κάποιο* τ*), τότε είτε* e *είναι μία τιμή είτε υπάρχει κάποιο* e' *τέτοιο ώστε* $e \longrightarrow e'$*.*

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.2.4.

□

4.3 Διατήρηση

Τώρα, θα αντιμετωπίσουμε το θεώρημα της διατήρησης, δίνοντας κάποια ακόμα λήμματα που είναι απαραίτητα για την απόδειξή του.

4.3.1 Αντιστροφή Σχέσης Ανάθεσης Τύπων

Όπως κάναμε και με τη σχέση υποτύπων, ξεκινάμε δίνοντας ένα ανάλογο λήμμα, που περιγράφει πως σχηματίζονται οι παραγωγές της σχέσης ανάθεσης τύπων. Το λήμμα της αντιστροφής της σχέσης ανάθεσης τύπων, γνωρίζοντας τη μορφή μίας έκφρασης που έχει κάποιο τύπο, μας πληροφορεί σχετικά με τη μορφή αυτού του τύπου και τους δυνατούς τύπους των υποεκφράσεών της. Αυτό το λήμμα βασίζεται στο αντίστοιχο λήμμα της αντιστροφής της σχέσης υποτύπων, ώστε να απλοποιήσει τις σχέσεις των διαφόρων τύπων που εμπλέκει και περιλαμβάνει μία περίπτωση για κάθε δυνατή μορφή έκφρασης.

Λήμμα (Αντιστροφή Σχέσης Ανάθεσης Τύπων)**.** *Δυνατές μορφές τύπων βάσει της μορφής έκφρασης κάποιου τύπου.*

1. *Εάν* $\Gamma \vdash x : \tau$*, τότε* $\tau_0 <: \tau$*, όπου* $x : \tau_0 \in \Gamma$*.*
2. *Εάν* $\Gamma \vdash c\,(e_1, \ldots, e_n) : \tau$*, τότε* $\tau_0 <: \tau$*, με* $\tau_0 = c\,(\tau_1, \ldots, \tau_n)$*, όπου* $\Gamma \vdash e_i : \tau_i$ *για κάθε* i*.*
3. *Εάν* $\Gamma \vdash op\,(e_1, \ldots, e_n) : \tau$*, τότε* $\tau_0 <: \tau$*, όπου* $\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau_0$ *και* $\Gamma \vdash e_i : \tau_i$ *για κάθε* i*.*
4. *Εάν* $\Gamma \vdash$ `fun end` $: \tau$*, τότε* $\tau_0 <: \tau$*, με* $\tau_0 =$ none $\to$ any*.*
5. *Εάν* $\Gamma \vdash$ `fun` $x \to e_r \mid \ldots \mid d_m$`end` $: \tau$*, τότε* $\tau_0 <: \tau$*, με* $\tau_0 = \tau_1 \to \tau_2$*, όπου* $\Gamma, x : \tau_1 \vdash e_r : \tau_2$*.*

6. *Εάν* $\Gamma \vdash \mathtt{fun}\, c\,(x_1, \ldots, x_n) \to e_r \,|\, d_2 \,|\, \ldots \,|\, d_m \mathtt{end} : \tau$,
   τότε $\tau_0 <: \tau$, *με* $\tau_0 = (c\,(\tau_1, \ldots, \tau_n) \to \tau_r) \wedge \bigwedge_i ((\neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \tau_i') \to \tau_i'')$,
   όπου $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_r : \tau_r$ *και* $\Gamma \vdash \mathtt{fun}\, d_2 \,|\, \ldots \,|\, d_m \mathtt{end} : \bigwedge_i (\tau_i' \to \tau_i'')$.

7. *Εάν* $\Gamma \vdash e_1\, e_2 : \tau$, *τότε* $\tau_0 <: \tau$, *όπου* $\Gamma \vdash e_1 : \tau' \to \tau_0$ *και* $\Gamma \vdash e_2 : \tau'$.

8. *Εάν* $\Gamma \vdash \mathtt{fix}\, x\,.\, e_r : \tau$, *τότε* $\tau_0 <: \tau$, *όπου* $\Gamma, x : \tau_0 \vdash e_r : \tau_0$.

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.3.1.

□

4.3.2 Αντικατάσταση

Σε αυτό το σημείο, δηλώνουμε κάποια δομικά λήμματα για τη σχέση ανάθεσης τύπων. Αυτά τα λήμματα δεν είναι ιδιαίτερα ενδιαφέροντα από μόνα τους, αλλά θα μας επιτρέψουν να εκτελούμε κάποιους χρήσιμους μετασχηματισμούς σε παραγωγές της σχέσης ανάθεσης τύπων σε μετέπειτα αποδείξεις. Το πρώτο λήμμα, επιτρέπει την αναδιάταξη των στοιχείων ενός περιβάλλοντος τύπων, χωρίς αλλαγή των δυνατών παραγωγών της σχέσης ανάθεσης τύπων με χρήση αυτού. Το δεύτερο λήμμα, παρομοίως, επιτρέπει την εξασθένηση των υποθέσεων ενός περιβάλλοντος τύπων, με την προσθήκη μίας εγγραφής για μία καινούργια μεταβλητή σε αυτό.

Λήμμα (Αναδιάταξη). *Εάν* $\Gamma \vdash e : \tau$ *και* Δ *είναι μία μετάθεση του* Γ, *τότε* $\Delta \vdash e : \tau$.

Λήμμα (Εξασθένηση). *Εάν* $\Gamma \vdash e : \tau$ *και δεν υπάρχει εγγραφή για τη* x *στο* Γ, *τότε* $\Gamma, x : \tau' \vdash e : \tau$.

Χρησιμοποιώντας τα παραπάνω τεχνικά λήμματα, μπορούμε να αποδείξουμε μία κρίσιμη ιδιότητα για τη σχέση ανάθεσης τύπων. Δηλαδή ότι, οι τύποι των εκφράσεων διατηρούνται όταν μεταβλητές αντικαθίστανται από άλλες εκφράσεις που έχουν τους κατάλληλους τύπους. Το ακόλουθο λήμμα δηλώνει τη διατήρηση τύπων υπό τη διαδικασία της αντικατάστασης και συχνά ονομάζεται απλά λήμμα αντικατάστασης (substitution lemma).

Λήμμα (Αντικατάσταση). *Εάν* $\Gamma, x : \tau' \vdash e : \tau$ *και* $\Gamma \vdash e' : \tau'$, *τότε* $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.3.4.

□

4.3.3 Διατήρηση

Με τη βοήθεια του λήμματος της αντικατάστασης, μπορούμε τώρα να αποδείξουμε το άλλο μισό της ιδιότητας της ασφάλειας τύπων, δηλαδή, ότι η αποτίμηση εκφράσεων διατηρεί τους τύπους αυτών. Συγκεκριμένα, το θεώρημα της διατήρησης δηλώνει ότι, εάν μία έκφραση κάποιου τύπου κάνει ένα βήμα αποτίμησης, τότε η επακόλουθη έκφραση θα έχει τον ίδιο τύπο.

Θεώρημα (Διατήρηση). *Εάν* $\Gamma \vdash e : \tau$ *και* $e \longrightarrow e'$, *τότε* $\Gamma \vdash e' : \tau$.

Απόδειξη. Δες την απόδειξη στα αγγλικά 4.3.5.

□

Κεφάλαιο 5

Μελλοντική Εργασία και Ανοικτά Ερωτήματα

Παρουσιάσαμε το σύνθετο αλλά εκφραστικό σύστημα τύπων μας για την απλή συναρτησιακή γλώσσα μας με ταίριασμα προτύπων και επίσης δείξαμε ότι οι ορισμοί μας είναι συνεπείς και ότι το σύστημα τύπων αναθέτει τους κατάλληλους τύπους στις εκφράσεις, έχοντας την ιδιότητα της ασφάλειας τύπων. Όμως, για να είναι το σύστημα τύπων χρήσιμο στην πράξη, θα πρέπει να έχουμε, επίσης, έναν απλό τρόπο να αποφασίζουμε ποιοι ακριβώς τύποι αναθέτονται σε μία δοθείσα έκφραση.

Προς το παρόν, δεν έχουμε κανέναν τρόπο για να απαντήσουμε στο ερώτημα του εάν μία συγκεκριμένη έκφραση μπορεί να έχει κάποιο τύπο στο σύστημα τύπων μας. Το βασικότερο εμπόδιο είναι η έλλειψη οποιωνδήποτε επισημειώσεων τύπων στη γλώσσα μας. Αυτό σημαίνει ότι η σχέση ανάθεσης τύπων θα πρέπει να μαντέψει τους τύπους των ορισμάτων των συναρτήσεων και έπειτα να ελέγξει εάν οι εκφράσεις επιστροφής τους μπορούν να αποκτήσουν κάποιο τύπο υποθέτοντας αυτούς τους τύπους για τα ορίσματα. Μηχανισμοί ανακατασκευής τύπων που δουλεύουν χωρίς επισημειώσεις τύπων είναι εφικτοί, όπως ο συμπερασμός τύπων στις γλώσσες προγραμματισμού της οικογένειας της ML, αλλά απαιτεί πιο αυστηρούς περιορισμούς για το τι τύπους έχουν οι συναρτήσεις.

Το πρόβλημα του συμπερασμού τύπων στη γλώσσα μας, όπως έχει, μοιάζει να είναι μη επιλύσιμο. Θα πρέπει να βάλουμε με κάποιον τρόπο περιορισμούς στο τι τύπους τα ορίσματα και τα αποτελέσματα των συναρτήσεων μπορούν να έχουν. Ένας τρόπος είναι να βάλουμε περιορισμούς στον τύπο κάθε ρήτρας συνάρτησης σύμφωνα με τους τύπους των άλλων ρητρών. Ίσως, οι ρήτρες μίας συνάρτησης δεν χρειάζεται να έχουν όλες τον ίδιο ακριβώς τύπο (όπως ο συμπερασμός τύπων στην ML απαιτεί), αλλά θα πρέπει να μοιράζονται κάποιες γενικές μορφές τύπων, έτσι ώστε να μειωθεί ο χώρος αναζήτησης για τους τύπους ορισμάτων. Ένας άλλος τρόπος είναι να περιορίσουμε το χώρο αναζήτησης με άμεσο τρόπο μέσω επισημειώσεων τύπων. Κάθε συνάρτηση θα μπορούσε να συνοδεύεται από μία πεπερασμένη ακολουθία από πιθανούς τύπους ορισμάτων της και οπότε θα μπορούσαμε απλά να δοκιμάζουμε καθέναν από αυτούς τους τύπους με τη σειρά κρατώντας μόνο όσους επιτυγχάνουν να αποδώσουν κάποιο τύπο στις εκφράσεις επιστροφής. Και οι δύο λύσεις οδηγούν σε μία σειρά από αλλαγές στο εκφραστικό σύστημα τύπων μας και η ανάλυσή τους θα ήταν ενδιαφέρουσα προς αναζήτηση ενός αλγορίθμου ανακατασκευής τύπων.

Το απλούστερο πρόβλημα της απόφασης της σχέσης ανάθεσης τύπων δεν μοιάζει εύκολα επιλύσιμο επίσης. Δηλαδή, θα θέλαμε να γνωρίζουμε δεδομένης μίας έκφρασης και ενός τύπου εάν το αντίστοιχο ζευγάρι μπορεί να παραχθεί από τη σχέση ανάθεσης τύπων. Αρχικά, θα πρέπει να οργανώσουμε διαφορετικά τους κανόνες μας έτσι ώστε να ακολουθούν μόνο τη σύνταξη των εκφράσεων, δηλαδή, να βρούμε σε ποια σημεία ακριβώς χρειάζεται η σχέση υποτύπων και να προσθέσουμε κατάλληλες προϋποθέσεις της σχέσης υποτύπων στους νέους κανόνες. Έπειτα, βέβαια θα πρέπει να μπορούμε να αποφασίσουμε αυτές τις προϋποθέσεις υποτύπων, δηλαδή, το αντίστοιχο πρόβλημα για τη σχέση υποτύπων μας θα πρέπει να είναι επίσης επιλύσιμο. Παρόμοια οργάνωση των κανόνων υποτύπων που δεν ακολουθούν τη σύνταξη των τύπων χρειάζεται, έτσι ώστε να μπορούμε εύκολα να βρούμε αν υπάρχει μία παραγωγή υποτύπων για δύο δεδομένουν τύπους. Βήματα προς αυτή την κατεύθυνση, για την τρέχουσα σχέση υποτύπων, έχουν ήδη γίνει με το λήμμα αντιστροφής της σχέσης υποτύπων.

Φυσικά, η τρέχουσα σχέση υποτύπων περιορίζει σημαντικά το σύστημα τύπων μας. Μόνο ένα μικρό μέρος της επιθυμητής συνολοθεωρητικής σημασίας των τύπων επιτυγχάνεται από αυτήν. Δηλαδή, η σχέση υποτύπων μας είναι ορθή, αλλά απέχει πολύ από το να είναι πλήρης. Πειραματισμός

με νέους κανόνες υποτύπων, που προσθέτουν σε αυτή τη σημασία των τύπων θα ήταν χρήσιμος. Η μεγαλύτερη ερώτηση του ποια ακριβώς είναι τα όρια μίας συντακτικά ορισμένης σχέσης υποτύπων είναι επίσης ενδιαφέρουσα για έρευνα. Πιο συγκεκριμένα, χρειαζόμαστε μία σειρά από νέους κανόνες, ώστε να απλοποιούμε σύνθετους τύπους, όπως αυτούς που αποδίδουμε σε συναρτήσεις, κάνοντας κάποια βήματα προς την αντιμετώπιση του μεγαλύτερου προβλήματος της ισοδυναμίας τύπων. Μέρος αυτών των απλοποιήσεων ίσως θα ήταν δυνατό να πραγματοποιηθεί από επιλυτές SMT, που θα δέχονταν κάποιους κατάλληλους μετασχηματισμούς των τύπων μας, οι οποίοι θα πρέπει να διατηρούν την πληροφορία τύπων.

Όλες αυτές οι αλλαγές και οι πειραματισμοί με το σύστημα τύπων έχουν το κόστος της επανεξέτασης των αποδείξεων ώστε να ελέγξουμε ότι οι επιθυμητές ιδιότητες συνεχίζουν να ισχύουν. Όπως διαπιστώσαμε, δουλεύοντας με ένα σύνθετο σύστημα τύπων, και μπορεί να γίνει εύκολα αντιληπτό σε οποιονδήποτε ξεφυλλίζοντας τις σελίδες αυτής της εργασίας, οι αποδείξεις είναι μεγάλες και κάποιες περιπτώσεις αρκετά περίπλοκες. Η υλοποίηση των ορισμών της γλώσσας και του συστήματος τύπων σε ένα περιβάλλον βοηθού αποδείξεων (proof assistant), αν και δύσκολη διαδικασία, θα ήταν καλή επένδυση χρόνου, με σκοπό την αυτοματοποίηση μέρους της διαδικασίας ελέγχου των αποδείξεων και την επαλήθευση των ισχυρισμών μας.

Κείμενο Εργασίας στα Αγγλικά

Chapter 1

Introduction

1.1 Objective

In this thesis, we define a static type system, that includes intersection and negation types and subtyping, to assign very descriptive types to programs of a simple functional language with pattern matching. We show that our type system assigns types only to programs with defined semantics, that is, it has the safety property.

1.2 Motivation

Programming continues to rise in popularity today, with more and more people involved with it. The main tools of programming, programming languages, are quite easy for everyone to learn. Computers everywhere handle most of our daily tasks efficiently. But, with all these abstractions, reasoning about programs emerges as an evergrowing problem. Most of a programmer's time is spent debugging and testing software in order to get some validation that it actually works.

Static type checking is the most basic form of program behavior verification. Having a system that is trying to assign types to programs based on their structure allows early detection of some programming errors. The type given to a program also provides useful information about what the program actually does, being an implicit documentation that evolves with its source code.

Of course, the more precise the information about a program, the more useful it is. Having a type system that assigns both accurate and descriptive types to programs requires a wide collection of type constructors that describe particular relations between types. But, we have to make sure that many valid programs are still able to be assigned some type, that is, the types of their containing parts are able to match one another.

In this thesis, we present such a type system. We introduce a rich algebra of type constructors, like intersection and negation types. We allow expressions to have many types using a syntactically defined subtyping relation. The object language, we assign types to, is a simple functional language featuring pattern matching for its multi-clause functions. This feature of the language enables us to use our set-theoretic type constructors to type functions very closely to their actual behavior.

1.3 Outline

The thesis is organized as follows: First of all, chapter 1 is a short introduction to what we did and why, and includes this outline you are reading right now. In chapter 2, we introduce a simple functional programming language, describing its abstract syntax and evaluation semantics. In chapter 3, we define a type system for this language, that includes a rich type algebra and subtyping. In chapter 4, we deal with the metatheory of this type system, showing that it has the property of type safety, by giving rigorous proofs of the progress and preservation theorems. Finally, in chapter 5, we discuss possible future work and pose some questions for further research.

The style of presentation of definitions and properties used in this work follows the great book *Types and Programming Languages* [Pier02].

Chapter 2

Language Definition

In order to design some complex type systems and analyze their properties, first we need to define an appropriate untyped language, for which we will introduce types later. This language has to be simple and elegant enough, so we do not have to discuss most of the technical details about its structure, but at the same time, it needs some interesting and powerful features, found in real world programming languages, to make the task of reasoning about programs written in it worthwhile. These requirements lead us to a functional language similar to those of the ML family, having in its core the well-known pure lambda-calculus. In this chapter, we present the chosen language, defining its abstract syntax and evaluation semantics.

2.1 Syntax

The syntax of the *lambda-calculus* includes only three kinds of terms, that is, *variables*, *abstractions* (function definitions) and *applications*. All mathematical computations can be reduced to these basic operations, making lambda-calculus both a simple programming language to describe computations and a mathematical object about which complex statements can be proved.

We extend this basic syntax, for convenience and presentation reasons, but also in order to introduce a type system that works well with some specific programming language features. Our language features mathematical *constants* and *operators*, a *fix-point* operator and function clauses with *pattern matching*.

The complete abstract syntax of the language is shown in figure 2.1. The conventions used in this grammar (and throughout this work) are close to those of standard BNF [Aho86]. The main difference is that we allow a finite sequence to be represented as $\alpha_1, \ldots, \alpha_n$ for simplicity.

e	::=	x	(variable)
	|	$c\,(e_1, \ldots, e_n)$	(constant)
	|	$op\,(e_1, \ldots, e_n)$	(operator)
	|	$\texttt{fun}\, d_1 \texttt{\|} \ldots \texttt{\|} d_m \texttt{end}$	(function)
	|	$e_1\, e_2$	(application)
	|	$\texttt{fix}\, x \texttt{.} e$	(fix-point)
d	::=	$p \to e$	(function clause)
p	::=	x	(variable pattern)
	|	$c\,(x_1, \ldots, x_n)$	(constant pattern)

Figure 2.1: Expressions

2.1.1 Expression Forms

The grammar for the abstract syntax of the language is just a compact notation to define inductively all valid constructs that can be formed in our language. We call these constructs *terms* or *expressions*. We use the metavariable e (with various subscripts) to stand for any expression, while E is the set that contains all of them.

Let us now examine the possible syntactic forms of an expression e giving more details about them. For any expression e exactly one of the following is true:

- The expression e has the form of a *variable*, $e = x$. We use the metavariables x and y to stand for language variables. Variables are names that can refer to function parameters and they are used, essentially, as placeholders to be replaced by another expression during the computation of the program.

- The expression e has the form of a *constant*, $e = c\,(e_1, \ldots, e_n)$ for some expressions e_i. Constants are tuples of any size that also have a name. We use the metavariable c to stand for constant names, while the number n is the size of the constant. The elements of a constant, $e_1, \ldots, e_n$, do not need to be constants themselves, they can be any expression. We call this expression form constant because its outer structure (name and size) cannot change during computation. Also, the actual mathematical constants, like integers and boolean values, are in this form, where $n = 0$ and the name of the constant corresponds to its value. Constants can be used to construct any complex nested structure to organize data in a specific way.

- The expression e has the form of an *operator*, $e = op\,(e_1, \ldots, e_n)$ for some expressions e_i. Operators have names and arities. We use the metavariable op to stand for operator names, while the number n is the arity of the operator. Every operator should be accompanied by its semantic function, $[\![op]\!]$, that defines how the operator apply to its operands $e_1, \ldots, e_n$ and transforms the operator expression to its computed result. So unlike constants, the set of built-in language operators cannot be expanded by the programmer. We include operators to have some basic operations defined over our simplest constants, like arithmetic and logical operations for integers and booleans, but we could also have libraries full of operators implementing all sorts of more complex computations over any kind of structured constants.

- The expression e has the form of a *function*, $e = \texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$. Function definitions are enclosed with keywords `fun` and `end` and include a sequence of function *clauses*. We use the metavariable d to stand for function clauses, while the number m is the size of their sequence. Each function clause d, has the form $p \to e_r$, where we call p the pattern of the clause and e_r the return expression of the clause. There are two kind of possible clause patterns, namely, the *variable pattern*, $p = x$, where x is a variable, and the *constant pattern*, $p = c\,(x_1, \ldots, x_n)$, where each x_i is a variable and c is the name of a constant with size n. This whole syntax of functions describes the computations that need to be done to some expected expression with a specific form (function input) in order to return another expression as result (function output). We allow the specific function `fun end`, that is a function with no clauses at all ($m = 0$), as a valid function expression, because, as we will see later, it has a special role in evaluation of expressions.

- The expression e has the form of an *application*, $e = e_1\, e_2$ for some expressions e_1 and e_2. Applications tie together the main building blocks of programs, functions, with other expressions as their inputs. If the expressions match, then we will eventually get the expected function output. To avoid writing too many parentheses, we adopt the convention that application associates to the left, that is $e_1\, e_2\, e_3$ is the same expression as $(e_1\, e_2)\, e_3$.

- The expression e has the form of a *fix-point*, $e = \texttt{fix}\, x\, .\, e_r$ for some expression e_r. Fix-point definitions start with the keyword `fix` followed by a variable name and end with another expression, the return expression of the fix-point. A fix-point expression allows its variable name

x to refer back to the whole initial expression e, making possible the definition of recursive functions. We will see later how this cyclic behavior is achieved by examining the evaluation of expressions.

2.1.2 Values

We call an expression that we consider as a valid possible final computed result of a program *value*. The abstract syntax of values in our language is shown in figure 2.2. We use the metavariable v to stand for values, while V is the set that contains all of them. The metavariable d is used again for function clauses, which have exactly the same syntax as given in figure 2.1.

$$
\begin{array}{lll}
v & ::= & c\,(v_1, \ldots, v_n) \qquad \text{(constant)} \\
 & \mid & \texttt{fun}\, d_1 \texttt{|} \ldots \texttt{|} d_m \texttt{end} \qquad \text{(function)}
\end{array}
$$

Figure 2.2: Values

From this grammar for values, we have that for any value v exactly one of the following is true:

- The value v has the form of a *constant*, $v = c\,(v_1, \ldots, v_n)$ for some values v_i. That is, a constant expression is a value only if its elements are themselves values.
- The value v has the form of a *function*, $v = \texttt{fun}\, d_1 \texttt{|} \ldots \texttt{|} d_m \texttt{end}$. Unlike constants, every function expression is also a value.

The set of values cannot be defined independently, because being a value is tied to the way the computations work. We should check that values as they are given here agree with the definition of evaluation of expressions in section 2.2.2.

2.1.3 Pattern Matching

We saw that our language has functions with multiple clauses, defining many return expressions, each one corresponding to some pattern. In order to compute the output of the function for a given input expression, we have to choose one of these return expressions based on the structure of the input and which clause patterns match with this structure. This language feature gives a powerful conditional programming construct called *pattern matching* that is often found in functional programming languages such as Standard ML [Miln97].

The following definition describes when a clause pattern matches a given expression:

Definition 2.1.1 (Pattern Match)**.** Matching between a pattern p and a given expression e.

- A variable pattern, $p = x$, matches with any given expression e.
- A constant pattern, $p = c\,(x_1, \ldots, x_n)$, matches with a given expression e, only if e has the form of a constant with the same name c and the same size n.

2.1.4 Scopes and Variable Binding

Let us now examine how language variables are used as placeholders for other expressions to abstractly describe computations. Some expressions, like functions, introduce variable names to refer to their expected arguments and then use these names in their body to indicate where they are needed to compute the desired output. These variables are called *bound* or dummy variables because they cannot be freely replaced by any expression, as they all refer back to the same place, their introduction (definition). The part of an expression that introduces (defines) a bound variable is called *binder*, while the expression inside which the bound variable can be used is called its *scope*.

The following definition lists all possible expressions that introduce bound variables in our language:

Definition 2.1.2 (Bound Variables)**.** Expressions that introduce bound variables (binders) and their scopes.

- In a function clause with a variable pattern, $x \to e_r$, the variable x is bound and its scope is the expression e_r.
- In a function clause with a constant pattern, $c\,(x_1, \ldots, x_n) \to e_r$, each one of the variables x_i is bound and its scope is the expression e_r.
- In a fix-point expression, $\texttt{fix}\, x\,.\, e_r$, the variable x is bound and its scope is the expression e_r.

A variable occurrence is said to be *bound* when it occurs in the scope of a binder with the same name. A variable occurrence is *free* if it appears in a position where it is not bound by any binder with the same name. A language term with no free variables is said to be *closed*.

2.1.5 Substitution

A free occurrence of a variable in an expression behaves as an external reference that needs to be resolved in order to specify the complete meaning of the expression. The operation that replaces variable occurrences with other expressions, resulting in a more specific instance of the initial expression, is called expression *substitution*. We use the notation $e'\,[x \mapsto e]$ to denote the expression we get after replacing every occurrence of the variable x in the expression e' by the expression e.

Even though the operation of substitution seems straightforward, it turns out to be quite tricky, when examined in detail. In particular, we have to handle carefully the following two cases in order to define substitution correctly. First, we have to replace only free occurrences of variables, as bound variables refer to some binder inside the expression, being simply placeholders for its use, and their specific names do not matter. Second, we have to avoid capturing free variables, appearing in the expression we replace with, by any binder of the initial expression, turning them into bound variables. Both of these problems can be solved by *renaming* the bound variables of the initial expression. Furthermore, we can always do that, as we consider terms that differ only in the names of bound variables interchangeable in all contexts.

The following definition of substitution has the intended behavior and describes exactly which properties hold after bound variable renaming:

Definition 2.1.3 (Substitution)**.** Capture-avoiding substitution using implicit renaming of bound variables.

The following statements define the safe substitution of the variable x by the expression e inside any expression e', $e'\,[x \mapsto e]$. For any bound variable y in e', we assume that, after the appropriate renaming, $y \neq x$ and y is not free in e.

$$
\begin{aligned}
&x\,[x \mapsto e] && = e \\
&y\,[x \mapsto e] && = y \quad if\, y \neq x \\
&c\,(e_1, \ldots, e_n)\,[x \mapsto e] && = c\,(e_1\,[x \mapsto e], \ldots, e_n\,[x \mapsto e]) \\
&op\,(e_1, \ldots, e_n)\,[x \mapsto e] && = op\,(e_1\,[x \mapsto e], \ldots, e_n\,[x \mapsto e]) \\
&\texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}\,[x \mapsto e] && = \texttt{fun}\, d_1\,[x \mapsto e] \,|\, \ldots \,|\, d_m\,[x \mapsto e]\, \texttt{end} \\
&(y \to e_r)\,[x \mapsto e] && = y \to e_r\,[x \mapsto e] \\
&(c\,(x_1, \ldots, x_n) \to e_r)\,[x \mapsto e] && = c\,(x_1, \ldots, x_n) \to e_r\,[x \mapsto e] \\
&(e_1\, e_2)\,[x \mapsto e] && = e_1\,[x \mapsto e]\ e_2\,[x \mapsto e] \\
&(\texttt{fix}\, y\,.\, e_r)\,[x \mapsto e] && = \texttt{fix}\, y\,.\, e_r\,[x \mapsto e]
\end{aligned}
$$

From this point on, when we mention any expression substitution we will always refer to this capture-avoiding substitution operation as defined here. We will also use the generalized notation $e\,[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ to denote the *concurrent substitution* of each variable x_i by the expression e_i in expression e.

Using the substitution operation is now possible to describe what computations take place in expressions and reason about their meaning.

2.2 Evaluation Semantics

Having formulated in detail the syntax of our language, we next need a similarly precise definition of how language terms are evaluated, or in other words the *semantics* of the language.

In pure lambda-calculus, the only way by which terms compute is the application of functions to arguments. Each step in the computation consists of rewriting an application of an abstraction to some other term, by substitution of the bound variable by that term in the abstraction's body.

Having enriched our language with extra constructs like constants and operators, and features like pattern matching in function definitions, there are certainly more cases to consider in the definition of computations, but the operation of substitution still has the same role of replacing expressions as arguments to functions in applications.

2.2.1 Operational Semantics

There are a couple of different basic approaches to formalizing semantics. One of these methods is *operational semantics* [Plot81]. Operational semantics specifies the behavior of a programming language by defining a simple *abstract machine* for it, that uses the terms of the language as its machine code. For simple languages, a *state* of the machine is just a term, and the machine's behavior is defined by a *transition function* that, for each state, either gives the next state by performing a step of simplification on the term or declares that the machine has halted. The final state that the machine reaches when started with a specific term as its initial state corresponds to the *meaning* of this term.

Operational semantics is often the method of choice for defining programming languages and studying their properties. We will also use this method for defining semantics as it is simple and concise but also expressive.

2.2.2 Evaluation Relation

We can now define a binary relation over the set of expressions, E, that contains exactly all ordered pair of expressions, which are possible consecutive states of the corresponding abstract machine, that means, the machine can change its state from being the first expression to the second one, by making a single step of computation. We call this the (*single-step*) *evaluation relation* and we write $e \longrightarrow e'$ to denote that expression e evaluates to expression e' in one step.

The evaluation relation is defined by induction using a set of *inference rules* and is shown in figure 2.3. Each inference rule has a number of *premises* and a single *conclusion*. Rules without a horizontal line (also known as *axioms*) have no premises, so their conclusion is always true. Special notice should be given to the choice of metavariables in the inference rules. We remind that the metavariable e ranges over the whole set of expressions, E, given in figure 2.1, while the metavariable v ranges over the set of values, $V \subset E$, given in figure 2.2. The choice of the appropriate metavariables helps control the order of evaluation. We can see that our language uses a *call by value* evaluation strategy, which does not allow any evaluation inside a function and always evaluates function arguments before passing them to the function, even if they are not used at all.

Let us now examine each of these rules in more detail to see which expression forms they handle and how their evaluation is defined.

The first rule, (E-CON), handles the evaluation of constants. The outer structure of the constant, its name and size, does not change during its evaluation. The elements of the constant are evaluated

$$\frac{e_i \longrightarrow e_i'}{c\,(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n) \longrightarrow c\,(v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_n)} \quad \text{(E-CON)}$$

$$\frac{e_i \longrightarrow e_i'}{op\,(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n) \longrightarrow op\,(v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_n)} \quad \text{(E-OP)}$$

$$op\,(v_1, \ldots, v_n) \longrightarrow [\![op]\!]\,(v_1, \ldots, v_n) \quad \text{(E-OPF)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\,e_2 \longrightarrow e_1'\,e_2} \quad \text{(E-APP1)}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1\,e_2 \longrightarrow v_1\,e_2'} \quad \text{(E-APP2)}$$

$$\texttt{fun}\,x \to e \,|\, \ldots \,|\, d_m \texttt{end}\,v \longrightarrow e\,[x \mapsto v] \quad \text{(E-FUNX)}$$

$$\texttt{fun}\,c\,(x_1, \ldots, x_n) \to e \,|\, \ldots \,|\, d_m \texttt{end}\,c\,(v_1, \ldots, v_n) \longrightarrow e\,[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \quad \text{(E-FUNCM)}$$

$$\frac{v \neq c\,(v_1, \ldots, v_n)}{\texttt{fun}\,c\,(x_1, \ldots, x_n) \to e \,|\, d_2 \,|\, \ldots \,|\, d_m \texttt{end}\,v \longrightarrow \texttt{fun}\,d_2 \,|\, \ldots \,|\, d_m \texttt{end}\,v} \quad \text{(E-FUNCN)}$$

$$\texttt{fix}\,x\,.\,e \longrightarrow e\,[x \mapsto \texttt{fix}\,x\,.\,e] \quad \text{(E-FIX)}$$

Figure 2.3: Evaluation Relation

from left to right. Each one must evaluate to a value before moving to the next. When all elements are values, or if the constant's size is zero, the whole constant expression is now a value and its evaluation is complete.

The next two rules, (E-OP) and (E-OPF), handle the evaluation of operators. First, the operands of the operator are evaluated using rule (E-OP). Just like constants, the operands are evaluated from left to right and each one must evaluate to a value before moving to the next. When all operands are values, or if the operator's arity is zero, rule (E-OPF) can be used to finally evaluate the operator expression to a value using the corresponding semantic function.

Rules (E-APP1), (E-APP2), (E-FUNX), (E-FUNCM) and (E-FUNCN), all handle the evaluation of applications. First, the left component of the application is evaluated to a value using rule (E-APP1). Next, rule (E-APP2) is used to evaluate the right component to a value too. At this point, the evaluation can continue only if we have a function, having at least one clause, applied to the other value, which we consider to be its argument. By rule (E-FUNX), if the first clause of the function has a variable pattern, we can pass the argument to the function as it matches the pattern, evaluating the application to the clause's return value, where the pattern's variable has been replaced by the argument value using

the operation of substitution. Similarly, the rule (E-FUNCM) handles a first clause with a constant pattern that matches the argument (both constants of the same form), evaluating the application to the result of the (concurrent) substitution of each pattern's variable by the corresponding argument's element value in the clause's return expression. The remaining case, where the first clause's pattern does not match the argument, is handled by the rule (E-FUNCN), trying to match the argument to the next clause's pattern (if any), evaluating the application to a new one, where the function with the first clause missing is applied to the same argument. To review when a pattern matches a given expression see definition 2.1.1.

Last but not least, rule (E-FIX) handles the evaluation of fix-points. The behavior of a fix-point resembles that of a function having a single clause with a variable pattern. The difference of cource is that, unlike functions, fix-points are not values, so they do not need to be applied to some argument in order to start evaluation. The semantics of fix-points can be viewed as equivalent to the application of the corresponding function to the exact initial fix-point expression as its argument, which is always implicitly passed to the function in order to evaluate. This circular behavior is what enables the definition and evaluation of recursive computations. All of that semantics for the built-in fix-point construct is achieved by the seemingly simple rule (E-FIX). The fix-point evaluates (in one step) to the result of the substitution of the variable it introduces by the exact initial fix-point expression in its return expression.

2.2.3 Evaluation Determinacy and Normal Forms

We will now record some important properties about the evaluation relation and introduce some common terminology, so we can refer to them easier.

By inspection of the inference rules (2.3) that define the (single-step) evaluation relation we saw that each rule handles the evaluation of a different kind of expression, because either its conclusion requires a specific unique expression form or it mentions metavariables that range over values, meaning that some other rule has to evaluate a particular subexpression first. In other words, there is not a single point during the evaluation of any expression, where more than one inference rule applies to the expression. The abstract machine never has to make a choice while changing its current state, so for any given expression there is a single unique sequence of states the machine passes through.

We formalize this property of the evaluation relation with the following theorem:

Theorem 2.2.1 (Determinacy of Single-step Evaluation)**.** *If* $e \longrightarrow e'$ *and* $e \longrightarrow e''$*, then* $e' = e''$.

Proof. Straightforward induction on a derivation of $e \longrightarrow e'$. At each step of the induction, we assume the desired result for all smaller derivations, and proceed by case analysis on the last evaluation rule used in the derivation.

The intuition behind the fact that no more than one evaluation rule can be applied to any given expression is described in our analysis of the evaluation relation.

□

The single-step evaluation relation describes how the transition between expressions during computation works. We are just as interested in the final results of computations, in other words, the states from which the abstract machine cannot take a step. These expressions, which have completed their evaluation, is said to be in *normal form*. We will consider their forms in more detail, as these forms classify expressions by their meaning.

Definition 2.2.2 (Normal Forms)**.** An expression e is in normal form if no evaluation rule applies to it, that is, there is no expression e' such that $e \longrightarrow e'$.

Sometimes, it is useful to be able to relate an expression to all the expressions that can be derived from it by zero or more single steps of evaluation.

The definition of this *multi-step* evaluation relation, follows:

Definition 2.2.3 (Multi-step Evaluation Relation)**.** The multi-step evaluation relation over expressions, $e \longrightarrow^* e'$, is the reflexive, transitive closure of the single-step evaluation relation, $e \longrightarrow e'$.

That is, it is the smallest relation such that:

- If $e \longrightarrow e'$, then $e \longrightarrow^* e'$.
- For all expressions e, $e \longrightarrow^* e$.
- If $e \longrightarrow^* e'$ and $e' \longrightarrow^* e''$, then $e \longrightarrow^* e''$.

After these definitions we can now state the following theorem:

Theorem 2.2.4 (Uniqueness of Normal Forms)**.** *If $e \longrightarrow^* e'$ and $e \longrightarrow^* e''$, where expressions e' and e'' are both in normal form, then $e' = e''$.*

Proof. Corollary of the determinacy of single-step evaluation theorem (2.2.1).

□

2.2.4 Stuck Expressions

The set of values V was introduced in section 2.1.2 as a subset of expressions, which contains exactly the syntactic forms that we consider as acceptable final computed results of programs written in our language. Now, having the semantics of the language defined by the evaluation relation, we should be able to verify this initial view of values. Being fully evaluated is part of what it is to be a value.

The following is a sanity check to ensure that our definitions of values are consistent:

Theorem 2.2.5. *Every value v is in normal form.*

Proof. By inspection of the evaluation relation (2.3), we can check that, indeed, there is no evaluation rule that can be applied to a value of any form, as defined in figure 2.2.

□

It is not that hard to see that the converse of statement 2.2.5 is not true for our language. An expression that cannot be evaluated any further is not necessarily a value. For example, in our analysis of the inference rules that define the evaluation relation in section 2.2.2, we saw that, in the case of evaluating an application expression, after reducing both of its subexpressions to values using rules (E-APP1) and (E-APP2), it is required, in order for evaluation to continue, that the left subexpression is a function value. This requirement is obvious, as the other relevant rules, (E-FUNX), (E-FUNCM) and (E-FUNCN), only define evaluation steps for values passing as arguments to functions. The evaluation of such an expression stops abruptly without resulting to a value. Expressions with this behavior are called *stuck*.

Definition 2.2.6 (Stuck)**.** A closed expression is stuck if it is in normal form but not a value.

The existence of stuck expressions does not mean that there is something wrong with our definition of evaluation. Some computations cannot have any meaning assigned to them, they simply do not make sense. A stuck expression can be seen as analogous to a state causing a *run-time error* during execution of some program. Such a program performs an undefined operation, probably not matching the programmer's intention, demonstrating some mistake in its underlying logic.

We give an overview of the different kinds of (minimal) stuck expressions in our language, describing their associated bad behavior, in the following definition:

Definition 2.2.7 (Undefined Semantics)**.** Expressions that eventually get stuck and why this happens.

- If the left of two values in an application is not a function (so it is a constant value), the application expression is stuck.

- If the operand values $v_1, \ldots, v_n$ of an operator op are not in the domain of its semantic function $[\![op]\!]$, the expression $op\,(v_1, \ldots, v_n)$ is stuck.
- If no clause of a function has a pattern matching its argument value v, the resulting expression `fun end` v is stuck.

2.2.5 Non-Termination

One final remark about the evaluation we need to make is that not every expression can be evaluated to a normal form. That means, some expressions can take evaluation steps forever. Of course, this is not suprising, having a built-in fix-point construct in our language. Any fix-point expression using its variable unconditionally in its return expression describes a recursive computation missing a base case, so its evaluation cannot terminate.

Chapter 3

Type System

Having defined an interesting language, in which complex enough computations can be described, we now want to introduce types for it. What we are trying to achieve is to classify language expressions according to their semantics without actually evaluating them. That is, gain some useful information about the kind of values they can compute just by inspecting their structure. In this chapter, we define a static type system with a rich type algebra, involving set-theoretic type constructors like intersection and negation, that, using a syntactically defined subtyping relation, associates these types with our language expressions.

3.1 Types

We already defined the semantics of our language in section 2.2 using an evaluation relation (figure 2.3) that decribes exactly how a given expression eventually evaluates (or not) to a value. What we now want is to get a more general notion of the meaning of expressions that does not require any evaluation to be done. This brings *types* into the discussion. Types can be thought as describing particular subsets of the set of our language's values, V, (see figure 2.2), so an expression having a specific type (we also say the expression belongs to that type) means that, when fully evaluated (if that is possible), the expression results to a value in the set described by that type.

3.1.1 Type Forms

In order to assign types to expressions we have to start by defining how types actually look like. We introduce some basic types matching particular subsets of language values but also (more than) enough set-theoretic *type constructors* that allow combining any types together. This choice leads us to a quite complex type system, making reasoning about it painful (or even impossible), but also granting us the expressive power to describe very precise information about types.

The chosen set of types, T, is defined by the extended BNF grammar (as we did with expressions 2.1) shown in figure 3.1. We use the metavariables τ and σ to range over types.

τ	::=	any	(any)
	|	none	(none)
	|	$c\,(\tau_1, \ldots, \tau_n)$	(constant)
	|	$\tau_1 \to \tau_2$	(arrow)
	|	$\tau_1 \wedge \tau_2$	(intersection)
	|	$\tau_1 \vee \tau_2$	(union)
	|	$\neg\tau$	(negation)
	|	α	(variable)
	|	$\mu\alpha\,.\,\tau$	(recursive)

Figure 3.1: Types

Let us now analyze the possible type forms and describe our *intended semantics* for each type. For any type τ exactly one of the following is true:

- Type τ has the form `any`, τ = `any`. Type `any` is the *maximal* type in our system and can be thought as the set of all values, V. That is, any expression that can be assigned to a type should also have type `any`.
- Type τ has the form `none`, τ = `none`. Type `none` is the *minimal* type in our system and can be thought as the empty set, $\varnothing$. That is, none (closed) expression should have type `none`.
- Type τ has the form of a *constant*, $\tau = c(\tau_1, \ldots, \tau_n)$. Constant types, as their expression counterparts, have a fixed name and size. Apart from also carrying the name information, they behave as (n-size) *products* of their operand types. Of course, they describe sets of constant values of the appropriate form and element types. Again, the size n is allowed to be zero, so, in this case, the constant type is the *singleton* set of the constant value with the same name and no elements.
- Type τ has the form of an *arrow*, $\tau = \tau_1 \to \tau_2$. Arrow types describe sets of function values. In particular, they classify those functions, which expect arguments of the left type, τ_1, and return results of the right type, τ_2. The arrow type constructor is right-associative, that is, the type $\tau_1 \to \tau_2 \to \tau_3$ stands for $\tau_1 \to (\tau_2 \to \tau_3)$.
- Type τ has the form of an *intersection*, $\tau = \tau_1 \wedge \tau_2$. The inhabitants of an intersection type [Copp78] are the values belonging to *both* operand types τ_1 and τ_2. The inclusion of intersection types is what makes our system so expressive. We will see later that the use of intersection types enables a very descriptive view of multi-clause function typing, where individual clause types can be combined into a single type without losing any of the initial type information. The intersection type constructor is both associative and commutative (as shown by later definitions) so we will sometimes treat nested series of intersections as a flat one that operates over all types, written $\bigwedge_i (\tau_i)$.
- Type τ has the form of an *union*, $\tau = \tau_1 \vee \tau_2$. Union types are the dual notion of intersection types and denote the set of values belonging to *either* operand types τ_1 or τ_2. We will not use unions in our definition of typing later, but they are quite useful behind the scene. Our given grammar of types does not seem to include commonly used types like `int` and `bool`, but our built-in operators surely expect arguments of these types. *Primitive type* definitions, using union types (and subtyping, as we will see later) over singleton types like integer and boolean constants, help bridge these gaps between required types. Just like intersections, we can treat nested unions as a flat one over many types.
- Type τ has the form of a *negation*, $\tau = \neg\tau_p$. Most likely the least common type constructor used in our system, negation types describe the values *not* belonging to its operand type τ_p. Negation is very useful in conjunction with intersection types to *refine* a particular type with new information about value forms it cannot contain, thus intersecting it with the negation of the type describing these value forms. We will see later, exactly how we use this technique to precisely type multi-clause functions.
- Type τ has the form of a *type variable*, $\tau = \alpha$. Type variables refer to other types. Currently, the only valid use of type variables is inside the body of a *recursive type* that bounds it. We use the metavariable α to range over type variables.
- Type τ has the form of a *recursive type*, $\tau = \mu\alpha.\tau_r$. A recursive type describes a tree structure that can always expand itself by performing one more step of *unfolding* using its recursive definition. This way an infinite type is defined that contains all generated values. Recursive types will not appear in our definition of typing, but can be used to define interesting recursive data types like lists, so we can work with them in our language.

3.1.2 Typing Contexts

One more useful notion, we should probably introduce at this point, is *typing contexts*. A typing context (also called a *type environment*) collects our current assumptions about the types of *free variables* in an expression while we are trying to match it to a type. That is, a typing context is a (possibly empty) sequence of bindings, $(x : \tau)$, between (free) variables and types we associate with them, that helps us keep track of what we already know about variables while typing a new expression. The simple syntax of typing contexts is summarized by the grammar shown in figure 3.2. We use metavariables Γ and Δ to stand for typing contexts.

$$\begin{array}{lll} \Gamma & ::= \varnothing & \text{(empty)} \\ & \mid \Gamma, x : \tau & \text{(variable-type binding)} \end{array}$$

Figure 3.2: Typing Contexts

When we encounter an expression introducing a new bound variable (binder 2.1.4), we need to make an assumption about this variable's type, updating the current typing context. A technical detail about adding a new binding to a typing context is that, obviously, it makes no sense to add a variable entry whose name already appears in the context. We require that the variable name is distinct and we can always satisfy this requirement the same way we did with (safe) substitution 2.1.5.

3.2 Subtyping

Having defined the syntax of types in our system, we could go ahead and now define how these types classify language expressions by their values. But there is an issue with our types we need to address first. Our rich type algebra enables us to give very precise types to expressions and, of course, that is exactly what we want to do. But then, we should also make sure that enough expressions can actually be typed, that means, (valid) argument types should be able to match the domain types of functions they are being applied to. As one can imagine, more precise types means that types can (almost) never exactly match one another. What we are missing here is the ability to allow expressions of one type to freely lose some information about themselves and also have other, more general, types. That is, we need another layer of definitions, below *typing*, called *subtyping*, telling us whether a particular type is just a more precise version of another given type.

3.2.1 Definition Circularity

So we have to define subtyping. If we follow the interpretation of types, which we have discussed up to this point, as sets of language values, then we can see that subtyping is a matter of whether a type is a subset of another. But, we do not know what values belong to each type. In particular, we do not know what type functions have, and, actually, we have to define how typing expressions works to find out. And then, we just saw that in order to define a useful typing, that can actually match argument types to their function domains, subtyping is essential.

We have to somehow break this cyclic dependency of our definitions. The inclusion of arrow types in our system, that is, having *higher-order* functions in our language is the main reason subtyping leads back to typing every expression first. Languages that do not treat functions as first class values completely avoid the described problem. Functional languages, of course, do not have this option.

A relatively new approach to deal with this circularity of defining subtyping is *semantic subtyping* [Fris08]. Subtyping is defined semantically by an interpretation of types into an untyped denotational model and therefore can be dealt set-theoretically. This approach is mostly independent of the language it is used in, but gives *extensional semantics* to functions, that is, functions are treated as a series of input-output pairs.

Instead, we choose to define subtyping *syntactically*. We can keep both our language semantics and type syntax the same, but will achieve only part of our intended semantics for set-theoretic type constructors.

3.2.2 Subtyping Relation

We will now define, as we discussed above, a binary relation over the set of our types (see 3.1), T, called the *subtyping relation*. We say that type τ is a *subtype* of type τ' (or τ' is a *supertype* of τ) and we write $\tau <: \tau'$. The intuition behind the statement $\tau <: \tau'$ is that the subtype, τ, is more informative than its supertype, τ', so we can safely use any expression of type τ in a context where an expression of type τ' is expected.

The subtyping relation is formalized as a collection of inference rules (as we did for the evaluation relation 2.3) and is shown in figures 3.3 and 3.4.

$$\tau <: \tau \qquad \text{(S-REFL)}$$

$$\frac{\tau <: \tau' \qquad \tau' <: \tau''}{\tau <: \tau''} \qquad \text{(S-TRANS)}$$

$$\tau <: \mathsf{any} \qquad \text{(S-ANY)}$$

$$\mathsf{none} <: \tau \qquad \text{(S-NONE)}$$

$$\frac{\tau_1 <: \tau_1' \ \ldots \ \tau_n <: \tau_n'}{c\,(\tau_1, \ldots, \tau_n) <: c\,(\tau_1', \ldots, \tau_n')} \qquad \text{(S-CON)}$$

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'} \qquad \text{(S-ARROW)}$$

Figure 3.3: Subtyping Relation (part 1 of 2)

We will now go through each one of these rules, giving some brief explanation of what they add to the subtyping relation.

First, rule (S-REFL) makes the subtyping relation *reflexive* and rule (S-TRANS) makes it *transitive*. Both of these properties follow directly from our intuition of subtyping.

Next, rule (S-ANY) makes every type a subtype of any, that is, any is the maximal type in our system. Similarly, rule (S-NONE) makes every type a supertype of none, that is, none is the minimal type in our system.

Rule (S-CON) handles the subtyping between two constant types. A constant type can be subtype of another, only if their names and sizes match. Then, the subtype relation between each of the corresponding operand types should also have the same direction (*covariant*) as in the rule's conclusion. This is the intended subtype behavior for product types.

Rule (S-ARROW) handles the subtyping between two arrow types. Recall that arrow types classify functions that expect arguments of the left type and return results of the right type. Notice that the sense of the subtype relation is reversed (*contravariant*) for argument types, while it runs in the same direction (*covariant*) for the result types as for the function types themselves. The intuition here

$$\tau_1 \wedge \tau_2 <: \tau_1 \qquad \tau_1 \wedge \tau_2 <: \tau_2 \tag{S-INTER1}$$

$$\frac{\tau <: \tau_1 \qquad \tau <: \tau_2}{\tau <: \tau_1 \wedge \tau_2} \tag{S-INTER2}$$

$$\tau_1 <: \tau_1 \vee \tau_2 \qquad \tau_2 <: \tau_1 \vee \tau_2 \tag{S-UNION1}$$

$$\frac{\tau_1 <: \tau \qquad \tau_2 <: \tau}{\tau_1 \vee \tau_2 <: \tau} \tag{S-UNION2}$$

$$\frac{\tau <: \tau'}{\neg\tau' <: \neg\tau} \tag{S-NEG}$$

$$c\,(\tau_1, \ldots, \tau_n) <: \neg(\tau \to \tau') \qquad \tau \to \tau' <: \neg c\,(\tau_1, \ldots, \tau_n) \tag{S-NEGCA}$$

$$\frac{c_1 \neq c_2 \vee n \neq m \vee \exists i.\, \tau_i <: \neg\tau'_i}{c_1\,(\tau_1, \ldots, \tau_n) <: \neg c_2\,(\tau'_1, \ldots, \tau'_m)} \tag{S-NEGCC}$$

$$\tau\,[\alpha \mapsto \mu\alpha\,.\,\tau] <: \mu\alpha\,.\,\tau \qquad \mu\alpha\,.\,\tau <: \tau\,[\alpha \mapsto \mu\alpha\,.\,\tau] \tag{S-REC}$$

Figure 3.4: Subtyping Relation (part 2 of 2)

is that, a function can be viewed as also accepting only a subset of its arguments, while producing a superset of its actual results.

The next two rules, (S-INTER1) and (S-INTER2) capture our intuition of intersection types. That is, expressions of the intersection type should have both operand types and vice versa. Multiple uses of these rules can be combined in subtyping derivations to get the corresponding general (but equivalent) rules for intersection of many types.

Likewise, rules (S-UNION1) and (S-UNION2) capture our intuition of the dual notion of union types. That is, expressions of the union type should have either operand types and vice versa.

Rule (S-NEG), our general rule for negation, defines that negation types reverse the direction of subtyping (*contravariant*), as we would expect from our intuition behind them.

Two more rules, (S-NEGCA) and (S-NEGCC), involve negation types but in conjuction with constant and arrow types. The first one treats every constant type as disjoint of every arrow type. The second treats also as disjoint, constant types that either mismatch in their outer structure (name and size) or have at least one pair of disjoint corresponding operand types.

Finally, rule (S-REC) defines that recursive types are equal to their one-step unfolds, that is, we treat them as *equi-recursive*.

3.3 Typing

Having a subtype relation ready to be used, it is, finally, time to define how, exactly, our types can be assigned to language expressions, classifying them according to the values they evaluate into, without, of course, evaluating them. We will see how typing can take advantage of the subtype relation in order to refine its own rules, that we will define shortly, matching successfully (valid) arguments types to the domain types of the functions they are being applied to. Also, we are going to, finally, reveal how intersection and negation types can be used to precisely describe the types of multi-clause functions, using the information provided by their clause patterns.

3.3.1 Typing Relation

Once more, we will have to define inductively a new relation, using a collection of inference rules (and axioms). The *typing relation* is a ternary (three-place) relation that correlates three different kinds of sets, that is, typing contexts (see 3.2), language expressions (see 2.1) and our system of types (see 3.1). We write $\Gamma \vdash e : \tau$ for statements derived from the typing rules and we say that, the type τ can be assigned to the expression e under the assumptions, for types of free variables in the expression e, recorded by the typing context Γ. The meaning of the statement $\Gamma \vdash e : \tau$ is that, we are able to conclude that expression e, when (if) fully evaluated, results to a value of type τ, given that any free variable in e is replaced with some value of the type assumed for it by the context Γ.

The set of inference rules that define the typing relation is shown in figure 3.5.

We will now go through each one of these typing rules, giving some details about which expression forms they assign types to and under what premises.

First, rule (T-VAR) handles the typing of variables. The only way to assign a type to a variable is to have already assumed one type for it earlier. That is, a variable has a type only if there is an entry for it in the current typing context.

Rule (T-CON) handles the typing of constant expressions. If each one of the constant's elements evaluates to a value of some type (under the assumption that the values represented by its free variables have the types assumed for them in Γ), then the constant expression evaluates to a value of the contant type matching its name and size and having the corresponding element types (under the same assumptions for variables in it, Γ).

Similarly, rule (T-OP) handles the typing of operator expressions. Knowing the operator's signature type, the result of its application to a number of values, each one resulting from its corresponding operand, should be a value of its codomain type, but only if these values belong to its domain type (including their amount matching the operator's arity).

Rule (T-FUN0) just defines the type of the function with no clauses at all. This function expects no argument so its domain type should be none. The return type of the function do not really matter, but choosing the type $\mathsf{none} \rightarrow \mathsf{any}$ for this function might be convenient as it is the maximal arrow type, denoting that any function has (at least) the behavior of the empty function.

Functions that start with a variable pattern clause are typed by rule (T-FUNX). We type functions with arrow types denoting the type of values they accept as arguments and the type of values they return as results. In this case, if by assuming a type of value for the function's variable we can show that its return expression can evaluate to a value of some type, then clearly the first type is a possible argument type while the second a possible return type for the function, so we can assign the appropriate arrow type to the function. Because, the variable pattern of this clause will match any passed argument value, all the other clauses never come into play, so we can completely ignore them.

Instead, rule (T-FUNC) types functions that start with a constant pattern clause. This typing rule for functions demonstrates our key idea of how to precisely type functions and uses intersection and negation types to achieve this, so there are a couple of things to unpack here. First, the left premise of the rule is a general case of the variable pattern case we described above. If by assuming any combination of types for the constant pattern's variables, the return expression can have some type, then we have an appropriate arrow type for the first clause. Now notice that the right premise describes

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash c\,(e_1, \ldots, e_n) : c\,(\tau_1, \ldots, \tau_n)} \quad \text{(T-CON)}$$

$$\frac{\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n}{\Gamma \vdash op\,(e_1, \ldots, e_n) : \tau} \quad \text{(T-OP)}$$

$$\varnothing \vdash \texttt{fun end} : \texttt{none} \to \texttt{any} \quad \text{(T-FUN0)}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \texttt{fun}\, x \to e \,|\, \ldots \,|\, d_m \texttt{end} : \tau \to \tau'} \quad \text{(T-FUNX)}$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \qquad \Gamma \vdash \texttt{fun}\, d_2 \,|\, \ldots \,|\, d_m \texttt{end} : \bigwedge_i (\tau_i' \to \tau_i'')}{\begin{array}{c}\Gamma \vdash \texttt{fun}\, c\,(x_1, \ldots, x_n) \to e \,|\, d_2 \,|\, \ldots \,|\, d_m \texttt{end} : \\ (c\,(\tau_1, \ldots, \tau_n) \to \tau) \wedge \bigwedge_i ((\neg c\,(\texttt{any}_1, \ldots, \texttt{any}_n) \wedge \tau_i') \to \tau_i'')\end{array}} \quad \text{(T-FUNC)}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \tau'} \quad \text{(T-APP)}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix}\, x.e : \tau} \quad \text{(T-FIX)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'} \quad \text{(T-SUB)}$$

Figure 3.5: Typing Relation

the functionality of the function having all the clauses except the first one, as an intersection of arrow types. That is, this function can behave as any of these arrow types corresponding to the other clauses, accepting values of the left type and returning values of the right type. Now, the function in the conclusion should combine this functionality. Surely, it should be able to behave as the first clause. But, because values matched from the first clause do not pass to the others anymore, every argument type of all the other clauses should be refined. This is achieved by intersecting every argument type of the other clauses with the negation of the most general type describing every value that can be matched by the first clause. Thus, the type of the whole function is an intersection of the first clause's arrow type and all the other arrow types with the argument types updated as described. This way, the argument types of all arrow types in the intersection are always kept distinct from one another, so the order of the types in the intersections do not matter, following the intended behavior.

The typing of application expressions is handled by rule (T-APP). If the left expression evaluates to a value of an arrow type, that is, a function, then the right expression has to evaluate to a value matching the function's argument type, in order for the application to result in a value of the function's result type.

Rule (T-FIX) handles the typing of fix-point expressions. Recall the recursive behavior of fix-points. Such an expression can be seen as returning a value of some type, only if its body returns a value of the same type, while we assume its variable is replaced by a value of the same type too. Notice that, this type do not have to correspond to functions, but usually sensible use of fix-points involves this type to be an arrow type, that is, defining recursive functions. Also, the fact that a fix-point has been assigned a type, does not mean that its evaluation has to terminate.

Last, but certainly not least, rule (T-SUB), known as the *subsumption* rule, is the bridge between the typing relation and the subtyping relation, refining the premises of all other typing rules, so types do not have to be an exact match. This rule allows any expression of some type, to be an element of any supertype of that type too.

This analysis completes the presentation of our type system. Notice that now, knowing how expressions (so values too) are classified by types, via the typing relation that relies on the subtyping relation, a new concrete definition of the semantics of our types is possible, as subsets of values that are classified by a given type, thus closing the dependency circle of the definitions.

Chapter 4

Metatheory

Having just completed the definitions of our type system, we would like to check that it is consistent and potentially useful. That is, it assigns the appropriate types to expressions and, in particular, it does not assign any type to expressions that lead to undefined semantics. This way, the type system can be used to statically report errors in programs before encountering them in run time. In this chapter, we show that our type system has this property, called safety, by giving rigorous proofs of the well-known progress and preservation theorems.

4.1 Safety

The most basic property of a type system, like the one we defined, is (type) *safety* (also called *soundness*). According to this property, *well-typed* programs, that is, expressions the type system can assign some type to, do not ”go wrong”. We have already chosen how to formalize bad behavior of programs, as reaching a stuck state 2.2.6, an expression that is not a final value but where the evaluation rules 2.3 do not define what to do next. What we want to know, then, is that well-typed expressions do not get stuck. We show this in two steps, commonly known as the *progress* and *preservation* theorems.

- *Progress:* A well-typed expression is not stuck (either it is a value or it can take a step according to the evaluation rules).
- *Preservation:* If a well-typed expression takes a step of evaluation, then the resulting expression is also well typed.

These properties together tell us that a well-typed expression can never reach a stuck state during evaluation.

The next sections list the proofs of the progress and preservation theorems, along with some lemmas used in them, for our given language and type system definitions.

4.2 Progress

Let us start with the progress theorem and the lemmas needed for its proof.

4.2.1 Subtyping Inversion

First, we record some properties about how subtyping derivations are built. The *inversion* of the subtyping relation lemma, given a particular subtype relation, describes what we can say about the forms of types involved in it. The structure of this lemma is like an algorithm with a number of cases that are being checked in sequence to see how the given subtype relation looks like and decide what type forms are possible in that case.

Lemma 4.2.1 (Subtyping Inversion)**.** *Possible type forms based on given subtype relation.*

On each case we consider only the type form combinations in the subtype relation not already considered by any previous cases.

1.
   - *If* $\tau <: \mu\alpha.\tau_r$, *then* $\tau <: \tau_r\,[\alpha \mapsto \mu\alpha.\tau_r]$.
   - *If* $\mu\alpha.\tau_r <: \tau$, *then* $\tau_r\,[\alpha \mapsto \mu\alpha.\tau_r] <: \tau$.

2.
   - *If* $\tau <: \tau_1 \wedge \tau_2$, *then* $\tau <: \tau_1$ *and* $\tau <: \tau_2$.
   - *If* $\tau_1 \vee \tau_2 <: \tau$, *then* $\tau_1 <: \tau$ *and* $\tau_2 <: \tau$.

3.
   - *If* $\tau_1 \wedge \tau_2 <: \tau$, *then* $\tau_1 <: \tau$ *or* $\tau_2 <: \tau$ *or* $\tau = \tau_1' \vee \tau_2'$ *and* $\tau_1 \wedge \tau_2 <: \tau_1'$ *or* $\tau_1 \wedge \tau_2 <: \tau_2'$.
   - *If* $\tau <: \tau_1 \vee \tau_2$, *then* $\tau <: \tau_1$ *or* $\tau <: \tau_2$ *or* $\tau = \tau_1' \wedge \tau_2'$ *and* $\tau_1' <: \tau_1 \vee \tau_2$ *or* $\tau_2' <: \tau_1 \vee \tau_2$.

4.
   - *If* $\tau <: \mathsf{any}$, *then* τ *can be of any form.*
   - *If* $\mathsf{none} <: \tau$, *then* τ *can be of any form.*
   - *If* $\mathsf{any} <: \tau$, *then* $\tau = \mathsf{any}$.
   - *If* $\tau <: \mathsf{none}$, *then* $\tau = \mathsf{none}$.

5.
   - *If* $\neg\tau_p <: \tau$, *then* $\tau = \neg\tau'$ *and* $\tau' <: \tau_p$.
   - *If* $\tau <: \neg\tau_p$, *then* $\tau = \neg\tau'$ *and* $\tau_p <: \tau'$, *or* $\tau = c\,(\tau_1, \ldots, \tau_n)$ *and* $\tau_p <: \tau_a \to \tau_a'$, *or* $\tau = \tau_a \to \tau_a'$ *and* $\tau_p <: c\,(\tau_1, \ldots, \tau_n)$, *or* $\tau = c_1\,(\tau_1, \ldots, \tau_n)$ *and* $\tau_p <: c_2\,(\tau_1', \ldots, \tau_m')$ *and* $c_1 \neq c_2$ *or* $n \neq m$ *or* $\tau_i <: \neg\tau_i'$ *for some* i.

6.
   - *If* $\tau <: c\,(\tau_1, \ldots, \tau_n)$, *then* $\tau = c\,(\tau_1', \ldots, \tau_n')$ *and* $\tau_i' <: \tau_i$ *for all* i.
   - *If* $\tau <: \tau_1 \to \tau_2$, *then* $\tau = \tau_1' \to \tau_2'$ *and* $\tau_1 <: \tau_1'$ *and* $\tau_2' <: \tau_2$.

Proof. By induction on subtyping derivations.

1.
   - By (S-REC), $\mu\alpha.\tau_r <: \tau_r\,[\alpha \mapsto \mu\alpha.\tau_r]$. By (S-TRANS), $\tau <: \tau_r\,[\alpha \mapsto \mu\alpha.\tau_r]$.
   - By (S-REC), $\tau_r\,[\alpha \mapsto \mu\alpha.\tau_r] <: \mu\alpha.\tau_r$. By (S-TRANS), $\tau_r\,[\alpha \mapsto \mu\alpha.\tau_r] <: \tau$.

2.
   - By (S-INTER1), $\tau_1 \wedge \tau_2 <: \tau_1$ and $\tau_1 \wedge \tau_2 <: \tau_2$. By (S-TRANS), $\tau <: \tau_1$ and $\tau <: \tau_2$.
   - By (S-UNION1), $\tau_1 <: \tau_1 \vee \tau_2$ and $\tau_2 <: \tau_1 \vee \tau_2$. By (S-TRANS), $\tau_1 <: \tau$ and $\tau_2 <: \tau$.

3.
   - By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau_1 \wedge \tau_2 <: \tau$ must be (S-ANY), (S-INTER1), (S-UNION1) or (S-TRANS).
     If the final rule is (S-ANY), then $\tau = \mathsf{any}$ and by (S-ANY), $\tau_1 <: \tau$ and $\tau_2 <: \tau$.
     If the final rule is (S-INTER1), then $\tau = \tau_1$ or $\tau = \tau_2$ and by (S-REFL), $\tau_1 <: \tau$ or $\tau_2 <: \tau$.
     If the final rule is (S-UNION1), then $\tau = \tau_1' \vee \tau_2'$ and $\tau_1' = \tau_1 \wedge \tau_2$ or $\tau_2' = \tau_1 \wedge \tau_2$ and by (S-REFL), $\tau_1 \wedge \tau_2 <: \tau_1'$ or $\tau_1 \wedge \tau_2 <: \tau_2'$.
     If the final rule is (S-TRANS), then $\tau_1 \wedge \tau_2 <: \sigma$ and $\sigma <: \tau$ and we proceed by cases on the form of σ:
     If $\sigma = \mu\alpha.\sigma_r$, then by case 1, $\tau_1 \wedge \tau_2 <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \tau$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.
     If $\sigma = \sigma_1 \wedge \sigma_2$, then by induction hypothesis for $\sigma <: \tau$, $\sigma_1 <: \tau$ or $\sigma_2 <: \tau$ or $\tau = \tau_1' \vee \tau_2'$ and $\sigma <: \tau_1'$ or $\sigma <: \tau_2'$. If $\sigma_1 <: \tau$ or $\sigma_2 <: \tau$, then by case 2, $\tau_1 \wedge \tau_2 <: \sigma_1$ and $\tau_1 \wedge \tau_2 <: \sigma_2$ and we get the desired result by induction on the form of σ_1 or σ_2. Otherwise, if $\tau = \tau_1' \vee \tau_2'$ and $\sigma <: \tau_1'$ or $\sigma <: \tau_2'$, then by (S-TRANS), $\tau_1 \wedge \tau_2 <: \tau_1'$ or $\tau_1 \wedge \tau_2 <: \tau_2'$.
     If σ has any other form, then by induction hypothesis for $\tau_1 \wedge \tau_2 <: \sigma$, $\tau_1 <: \sigma$ or $\tau_2 <: \sigma$ or $\sigma = \sigma_1 \vee \sigma_2$ and $\tau_1 \wedge \tau_2 <: \sigma_1$ or $\tau_1 \wedge \tau_2 <: \sigma_2$. If $\tau_1 <: \sigma$ or $\tau_2 <: \sigma$, then

by (S-TRANS), $\tau_1 <: \tau$ or $\tau_2 <: \tau$. Otherwise, if $\sigma = \sigma_1 \vee \sigma_2$ and $\tau_1 \wedge \tau_2 <: \sigma_1$ or $\tau_1 \wedge \tau_2 <: \sigma_2$, then by case 2, $\sigma_1 <: \tau$ and $\sigma_2 <: \tau$ and we get the desired result by induction on the form of σ_1 or σ_2.

- By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau <: \tau_1 \vee \tau_2$ must be (S-NONE), (S-UNION1), (S-INTER1) or (S-TRANS).

  If the final rule is (S-NONE), then $\tau = \mathsf{none}$ and by (S-NONE), $\tau <: \tau_1$ and $\tau <: \tau_2$.

  If the final rule is (S-UNION1), then $\tau = \tau_1$ or $\tau = \tau_2$ and by (S-REFL), $\tau <: \tau_1$ or $\tau <: \tau_2$.

  If the final rule is (S-INTER1), then $\tau = \tau_1' \wedge \tau_2'$ and $\tau_1' = \tau_1 \vee \tau_2$ or $\tau_2' = \tau_1 \vee \tau_2$ and by (S-REFL), $\tau_1' <: \tau_1 \vee \tau_2$ or $\tau_2' <: \tau_1 \vee \tau_2$.

  If the final rule is (S-TRANS), then $\tau <: \sigma$ and $\sigma <: \tau_1 \vee \tau_2$ and we proceed by cases on the form of σ:

  If $\sigma = \mu\alpha\,.\,\sigma_r$, then by case 1, $\tau <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \tau_1 \vee \tau_2$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.

  If $\sigma = \sigma_1 \vee \sigma_2$, then by induction hypothesis for $\tau <: \sigma$, $\tau <: \sigma_1$ or $\tau <: \sigma_2$ or $\tau = \tau_1' \wedge \tau_2'$ and $\tau_1' <: \sigma$ or $\tau_2' <: \sigma$. If $\tau <: \sigma_1$ or $\tau <: \sigma_2$, then by case 2, $\sigma_1 <: \tau_1 \vee \tau_2$ and $\sigma_2 <: \tau_1 \vee \tau_2$ and we get the desired result by induction on the form of σ_1 or σ_2. Otherwise, if $\tau = \tau_1' \wedge \tau_2'$ and $\tau_1' <: \sigma$ or $\tau_2' <: \sigma$, then by (S-TRANS), $\tau_1' <: \tau_1 \vee \tau_2$ or $\tau_2' <: \tau_1 \vee \tau_2$.

  If σ has any other form, then by induction hypothesis for $\sigma <: \tau_1 \vee \tau_2$, $\sigma <: \tau_1$ or $\sigma <: \tau_2$ or $\sigma = \sigma_1 \wedge \sigma_2$ and $\sigma_1 <: \tau_1 \vee \tau_2$ or $\sigma_2 <: \tau_1 \vee \tau_2$. If $\sigma <: \tau_1$ or $\sigma <: \tau_2$, then by (S-TRANS), $\tau <: \tau_1$ or $\tau <: \tau_2$. Otherwise, if $\sigma = \sigma_1 \wedge \sigma_2$ and $\sigma_1 <: \tau_1 \vee \tau_2$ or $\sigma_2 <: \tau_1 \vee \tau_2$, then by case 2, $\tau <: \sigma_1$ and $\tau <: \sigma_2$ and we get the desired result by induction on the form of σ_1 or σ_2.

4. - Immediate from (S-ANY).
   - Immediate from (S-NONE).
   - By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\mathsf{any} <: \tau$ must be (S-ANY), (S-REFL) or (S-TRANS).

     If the final rule is (S-ANY) or (S-REFL), then $\tau = \mathsf{any}$.

     If the final rule is (S-TRANS), then $\mathsf{any} <: \sigma$ and $\sigma <: \tau$ and we proceed by cases on the form of σ:

     If $\sigma = \mu\alpha\,.\,\sigma_r$, then by case 1, $\mathsf{any} <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \tau$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.

     If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\mathsf{any} <: \sigma_1$ and $\mathsf{any} <: \sigma_2$ and also $\sigma_1 <: \tau$ or $\sigma_2 <: \tau$, so we get the desired result by induction on the form of σ_1 or σ_2.

     If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\mathsf{any} <: \sigma_1$ or $\mathsf{any} <: \sigma_2$ and also $\sigma_1 <: \tau$ and $\sigma_2 <: \tau$, so we get the desired result by induction on the form of σ_1 or σ_2.

     If σ has any other form, then by induction hypothesis for $\mathsf{any} <: \sigma$, $\sigma = \mathsf{any}$. By induction hypothesis for $\sigma <: \tau$, $\tau = \mathsf{any}$.
   - By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau <: \mathsf{none}$ must be (S-NONE), (S-REFL) or (S-TRANS).

     If the final rule is (S-NONE) or (S-REFL), then $\tau = \mathsf{none}$.

     If the final rule is (S-TRANS), then $\tau <: \sigma$ and $\sigma <: \mathsf{none}$ and we proceed by cases on the form of σ:

     If $\sigma = \mu\alpha\,.\,\sigma_r$, then by case 1, $\tau <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \mathsf{none}$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.

If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\tau <: \sigma_1$ and $\tau <: \sigma_2$ and also $\sigma_1 <: \mathsf{none}$ or $\sigma_2 <: \mathsf{none}$, so we get the desired result by induction on the form of σ_1 or σ_2.

If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\tau <: \sigma_1$ or $\tau <: \sigma_2$ and also $\sigma_1 <: \mathsf{none}$ and $\sigma_2 <: \mathsf{none}$, so we get the desired result by induction on the form of σ_1 or σ_2.

If σ has any other form, then by induction hypothesis for $\sigma <: \mathsf{none}$, $\sigma = \mathsf{none}$. By induction hypothesis for $\tau <: \sigma$, $\tau = \mathsf{none}$.

5.
- By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\neg\tau_p <: \tau$ must be (S-NEG), (S-REFL) or (S-TRANS).

  If the final rule is (S-NEG), then $\tau = \neg\tau'$ and $\tau' <: \tau_p$.

  If the final rule is (S-REFL), then $\tau = \neg\tau_p$ and by (S-REFL), $\tau_p <: \tau_p$.

  If the final rule is (S-TRANS), then $\neg\tau_p <: \sigma$ and $\sigma <: \tau$ and we proceed by cases on the form of σ:

  If $\sigma = \mu\alpha\,.\,\sigma_r$, then by case 1, $\neg\tau_p <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \tau$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.

  If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\neg\tau_p <: \sigma_1$ and $\neg\tau_p <: \sigma_2$ and also $\sigma_1 <: \tau$ or $\sigma_2 <: \tau$, so we get the desired result by induction on the form of σ_1 or σ_2.

  If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\neg\tau_p <: \sigma_1$ or $\neg\tau_p <: \sigma_2$ and also $\sigma_1 <: \tau$ and $\sigma_2 <: \tau$, so we get the desired result by induction on the form of σ_1 or σ_2.

  By case 4, $\sigma \neq \mathsf{any}$ and $\sigma \neq \mathsf{none}$.

  If σ has any other form, then by induction hypothesis for $\neg\tau_p <: \sigma$, $\sigma = \neg\sigma'$ and $\sigma' <: \tau_p$. By induction hypothesis for $\sigma <: \tau$, $\tau = \neg\tau'$ and $\tau' <: \sigma'$, and by (S-TRANS), $\tau' <: \tau_p$.
- By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau <: \neg\tau_p$ must be (S-NEGCA), (S-NEGCC), (S-NEG), (S-REFL) or (S-TRANS).

  If the final rule is (S-NEGCA), then $\tau = c\,(\tau_1, \ldots, \tau_n)$ and $\tau_p = \tau_a \to \tau'_a$, so by (S-REFL), $\tau_p <: \tau_a \to \tau'_a$, or $\tau = \tau_a \to \tau'_a$ and $\tau_p = c\,(\tau_1, \ldots, \tau_n)$, so by (S-REFL), $\tau_p <: c\,(\tau_1, \ldots, \tau_n)$.

  If the final rule is (S-NEGCC), then $\tau = c_1\,(\tau_1, \ldots, \tau_n)$ and $\tau_p = c_2\,(\tau'_1, \ldots, \tau'_m)$, so by (S-REFL), $\tau_p <: c_2\,(\tau'_1, \ldots, \tau'_m)$, and also $c1 \neq c2$ or $n \neq m$ or $\tau_i <: \neg\tau'_i$ for some i.

  If the final rule is (S-NEG), then $\tau = \neg\tau'$ and $\tau_p <: \tau'$.

  If the final rule is (S-REFL), then $\tau = \neg\tau_p$ and by (S-REFL), $\tau_p <: \tau_p$.

  If the final rule is (S-TRANS), then $\tau <: \sigma$ and $\sigma <: \neg\tau_p$ and we proceed by cases on the form of σ:

  If $\sigma = \mu\alpha\,.\,\sigma_r$, then by case 1, $\tau <: \sigma_r\,[\alpha \mapsto \sigma]$ and $\sigma_r\,[\alpha \mapsto \sigma] <: \neg\tau_p$ and we get the desired result by induction on the form of $\sigma_r\,[\alpha \mapsto \sigma]$.

  If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\tau <: \sigma_1$ and $\tau <: \sigma_2$ and also $\sigma_1 <: \neg\tau_p$ or $\sigma_2 <: \neg\tau_p$, so we get the desired result by induction on the form of σ_1 or σ_2.

  If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\tau <: \sigma_1$ or $\tau <: \sigma_2$ and also $\sigma_1 <: \neg\tau_p$ and $\sigma_2 <: \neg\tau_p$, so we get the desired result by induction on the form of σ_1 or σ_2.

  By case 4, $\sigma \neq \mathsf{any}$ and $\sigma \neq \mathsf{none}$.

  If σ has any other form, then by induction hypothesis for $\sigma <: \neg\tau_p$, we get the desired property for σ. By induction hypothesis for $\tau <: \sigma$, we get the desired property for τ.

6.
- By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau <: c\,(\tau_1, \ldots, \tau_n)$ must be (S-CON), (S-REFL) or (S-TRANS).

  If the final rule is (S-CON), then $\tau = c\,(\tau'_1, \ldots, \tau'_n)$ and $\tau'_i <: \tau_i$ for all i.

If the final rule is (S-REFL), then $\tau = c(\tau_1, \ldots, \tau_n)$ and by (S-REFL), $\tau_i <: \tau_i$ for all i.

If the final rule is (S-TRANS), then $\tau <: \sigma$ and $\sigma <: c(\tau_1, \ldots, \tau_n)$ and we proceed by cases on the form of σ:

If $\sigma = \mu\alpha.\sigma_r$, then by case 1, $\tau <: \sigma_r[\alpha \mapsto \sigma]$ and $\sigma_r[\alpha \mapsto \sigma] <: c(\tau_1, \ldots, \tau_n)$ and we get the desired result by induction on the form of $\sigma_r[\alpha \mapsto \sigma]$.

If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\tau <: \sigma_1$ and $\tau <: \sigma_2$ and also $\sigma_1 <: c(\tau_1, \ldots, \tau_n)$ or $\sigma_2 <: c(\tau_1, \ldots, \tau_n)$, so we get the desired result by induction on the form of σ_1 or σ_2.

If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\tau <: \sigma_1$ or $\tau <: \sigma_2$ and also $\sigma_1 <: c(\tau_1, \ldots, \tau_n)$ and $\sigma_2 <: c(\tau_1, \ldots, \tau_n)$, so we get the desired result by induction on the form of σ_1 or σ_2.

By cases 4 and 5, $\sigma \neq \mathsf{any}$, $\sigma \neq \mathsf{none}$ and $\sigma \neq \neg\sigma'$.

If σ has any other form, then by induction hypothesis for $\sigma <: c(\tau_1, \ldots, \tau_n)$, $\sigma = c(\sigma_1, \ldots, \sigma_n)$ and $\sigma_i <: \tau_i$ for all i. By induction hypothesis for $\tau <: \sigma$, $\tau = c(\tau'_1, \ldots, \tau'_n)$ and $\tau'_i <: \sigma_i$ for all i, and by (S-TRANS), $\tau'_i <: \tau_i$ for all i.

- By inspection of the subtyping rules 3.3 3.4, excluding type form combinations considered by previous cases, the final rule in the derivation of $\tau <: \tau_1 \to \tau_2$ must be (S-ARROW), (S-REFL) or (S-TRANS).

  If the final rule is (S-ARROW), then $\tau = \tau'_1 \to \tau'_2$ and $\tau_1 <: \tau'_1$ and $\tau'_2 <: \tau_2$.

  If the final rule is (S-REFL), then $\tau = \tau_1 \to \tau_2$ and by (S-REFL), $\tau_1 <: \tau_1$ and $\tau_2 <: \tau_2$.

  If the final rule is (S-TRANS), then $\tau <: \sigma$ and $\sigma <: \tau_1 \to \tau_2$ and we proceed by cases on the form of σ:

  If $\sigma = \mu\alpha.\sigma_r$, then by case 1, $\tau <: \sigma_r[\alpha \mapsto \sigma]$ and $\sigma_r[\alpha \mapsto \sigma] <: \tau_1 \to \tau_2$ and we get the desired result by induction on the form of $\sigma_r[\alpha \mapsto \sigma]$.

  If $\sigma = \sigma_1 \wedge \sigma_2$, then by cases 2 and 3, $\tau <: \sigma_1$ and $\tau <: \sigma_2$ and also $\sigma_1 <: \tau_1 \to \tau_2$ or $\sigma_2 <: \tau_1 \to \tau_2$, so we get the desired result by induction on the form of σ_1 or σ_2.

  If $\sigma = \sigma_1 \vee \sigma_2$, then by cases 3 and 2, $\tau <: \sigma_1$ or $\tau <: \sigma_2$ and also $\sigma_1 <: \tau_1 \to \tau_2$ and $\sigma_2 <: \tau_1 \to \tau_2$, so we get the desired result by induction on the form of σ_1 or σ_2.

  By cases 4 and 5, $\sigma \neq \mathsf{any}$, $\sigma \neq \mathsf{none}$ and $\sigma \neq \neg\sigma'$.

  If σ has any other form, then by induction hypothesis for $\sigma <: \tau_1 \to \tau_2$, $\sigma = \sigma_1 \to \sigma_2$ and $\tau_1 <: \sigma_1$ and $\sigma_2 <: \tau_2$. By induction hypothesis for $\tau <: \sigma$, $\tau = \tau'_1 \to \tau'_2$ and $\sigma_1 <: \tau'_1$ and $\tau'_2 <: \sigma_2$, and by (S-TRANS), $\tau_1 <: \tau'_1$ and $\tau'_2 <: \tau_2$.

□

4.2.2 Canonical Forms

Using the subtyping inversion lemma 4.2.1, we can now record a couple of facts about the possible shapes of the *canonical forms* of some particular types. The following lemma, describes which are the possible forms of values that are well-typed, given the specific form of their type.

Lemma 4.2.2 (Canonical Forms)**.** *Forms of closed and well-typed values ($\varnothing \vdash v : \tau$).*

1. *If v is a closed value of type $c(\tau_1, \ldots, \tau_n)$ ($\varnothing \vdash v : c(\tau_1, \ldots, \tau_n)$), then v has the form $c(v_1, \ldots, v_n)$.*

2. *If v is a closed value of type $\tau_1 \to \tau_2$ ($\varnothing \vdash v : \tau_1 \to \tau_2$), then v has the form $\mathtt{fun}\, d_1 \mid \ldots \mid d_m \mathtt{end}$.*

Proof. By induction on typing derivations.

1. By inspection of the typing rules 3.5, the final rule in the derivation of $\varnothing \vdash v : c\,(\tau_1, \ldots, \tau_n)$ must be (T-CON) or (T-SUB).

   If the final rule is (T-CON), then v has the form $c\,(v_1, \ldots, v_n)$.

   If the final rule is (T-SUB), then $\varnothing \vdash v : \sigma$ with $\sigma <: c\,(\tau_1, \ldots, \tau_n)$.
   By subtyping inversion lemma 4.2.1, the only possible form of σ that also match some other typing rule is $c\,(\sigma_1, \ldots, \sigma_n)$.
   By induction hypothesis for $\varnothing \vdash v : \sigma$, v has the form $c\,(v_1, \ldots, v_n)$.

2. By inspection of the typing rules 3.5, the final rule in the derivation of $\varnothing \vdash v : \tau_1 \to \tau_2$ must be (T-FUN0), (T-FUNX) or (T-SUB).

   If the final rule is (T-FUN0) or (T-FUNX), then v has the form $\texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$.

   If the final rule is (T-SUB), then $\varnothing \vdash v : \sigma$ with $\sigma <: \tau_1 \to \tau_2$.
   By subtyping inversion lemma 4.2.1, the only possible forms of σ that also match some other typing rule are $\sigma_1 \to \sigma_2$ and $\bigwedge_i (\sigma_i \to \sigma_i')$ with $i > 1$.
   If σ has an arrow form, then by induction hypothesis for $\varnothing \vdash v : \sigma$, v has the form $\texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$.
   Otherwise, if σ has an intersection form, then the rule (T-FUNC) is used in the derivation and v also has the form $\texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$.

□

A corollary of the canonical forms lemma 4.2.2, states that type none is empty. That was indeed our intended semantics of type none, and this fact will be important in order to exclude some cases in other theorems that cannot occur because some value has to be assigned the type none.

Lemma 4.2.3 (None Values)**.** *There are no closed values of type* none*. (*$\varnothing \vdash v$: none *cannot occur for any value* v*)*

Proof. Let us assume that there is a closed value v of type none, $\varnothing \vdash v$: none.
By (S-NONE) and (T-SUB), we can get $\varnothing \vdash v : \tau$ for any type τ.
So we have $\varnothing \vdash v : c\,(\tau_1, \ldots, \tau_n)$ and $\varnothing \vdash v : \tau_1 \to \tau_2$.
By canonical forms lemma 4.2.2, v has both the form $c\,(v_1, \ldots, v_n)$ and the form $\texttt{fun}\, d_1 \,|\, \ldots \,|\, d_m \texttt{end}$.
However, it is clear from the syntax 2.2 that a value cannot have both the form of a constant and a function. So our assumption cannot be true, there are no closed values of type none, that means the type none is indeed empty.

□

4.2.3 Progress

With these lemmas we can now prove the *progress* theorem. The progress theorem states that a well-typed expression is not (currently) stuck, that is, either it is a value, or it can take an evaluation step. One small detail is that we are only interested in *closed* expressions, with no free variables. For open expressions, the progress theorem actually fails, as variables cannot evaluate and they are not values. However, this failure does not represent a defect in the language, since complete programs, which are the expressions we actually care about evaluating, are always closed.

Theorem 4.2.4 (Progress)**.** *If* e *is closed and well-typed (*$\varnothing \vdash e : \tau$ *for some* τ*), then either* e *is a value or there is some* e' *such that* $e \longrightarrow e'$*.*

Proof. By induction on a derivation of $\varnothing \vdash e : \tau$. Assuming that the desired property holds for all subderivations, we proceed by case analysis on the final typing rule in the derivation.

Case (T-VAR)**:**
$e = x$
In this case, e has the form of a variable and is not closed, so this case cannot occur.

Case (T-CON)**:**
$e = c\,(e_1, \ldots, e_n)$
$\tau = c\,(\tau_1, \ldots, \tau_n)$
$\varnothing \vdash e_1 : \tau_1 \;\ldots\; \varnothing \vdash e_n : \tau_n$
By induction hypothesis, for each e_i, either e_i is a value or e_i can take an evaluation step.

- If some e_i can take a step, $e_i \longrightarrow e_i'$, while all e_j, $j < i$, are values, $e_j = v_j$, then rule (E-CON) applies to e, so $e \longrightarrow e'$, where $e' = c\,(v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_n)$.
- If all e_i are values, $e_i = v_i$, then $e = c\,(v_1, \ldots, v_n)$ is a value.

Case (T-OP)**:**
$e = op\,(e_1, \ldots, e_n)$
$\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau$
$\varnothing \vdash e_1 : \tau_1 \;\ldots\; \varnothing \vdash e_n : \tau_n$
By induction hypothesis, for each e_i, either e_i is a value or e_i can take an evaluation step.

- If some e_i can take a step, $e_i \longrightarrow e_i'$, while all e_j, $j < i$, are values, $e_j = v_j$, then rule (E-OP) applies to e, so $e \longrightarrow e'$, where $e' = op\,(v_1, \ldots, v_{i-1}, e_i', e_{i+1}, \ldots, e_n)$.
- If all e_i are values, $e_i = v_i$, then rule (E-OPF) applies to e, so $e \longrightarrow e'$, where $e' = [\![op]\!]\,(v_1, \ldots, v_n)$.

Case (T-FUN0)**,** (T-FUNX)**,** (T-FUNC)**:**
$e = \mathsf{fun}\, d_1 \,|\, \ldots \,|\, d_m \mathsf{end}$
In all these cases, e has the form of a function, so e is a value.

Case (T-APP)**:**
$e = e_1\, e_2$
$\varnothing \vdash e_1 : \tau' \to \tau$
$\varnothing \vdash e_2 : \tau'$
By induction hypothesis, for each of e_1 and e_2, either e_i is a value or e_i can take an evaluation step.

- If e_1 can take a step, $e_1 \longrightarrow e_1'$, then rule (E-APP1) applies to e, so $e \longrightarrow e'$, where $e' = e_1'\, e_2$.
- If e_1 is a value, $e_1 = v_1$, and e_2 can take a step, $e_2 \longrightarrow e_2'$, then rule (E-APP2) applies to e, so $e \longrightarrow e'$, where $e' = v_1\, e_2'$.
- If both e_1 and e_2 are values, $e_1 = v_1$ and $e_2 = v_2$, then by canonical forms lemma 4.2.2, v_1 has the form of a function, $v_1 = \mathsf{fun}\, d_1 \,|\, \ldots \,|\, d_m \mathsf{end}$.
  - If v_1 has no clauses ($m = 0$), $v_1 = \mathsf{fun\ end}$, then by (T-FUN0), $\varnothing \vdash v_1 : \mathsf{none} \to \mathsf{any}$, which is the minimal type for v_1, so $\mathsf{none} \to \mathsf{any} <: \tau' \to \tau$.
    By subtyping inversion lemma 4.2.1, $\tau' <: \mathsf{none}$, so by (T-SUB), $\varnothing \vdash v_2 : \mathsf{none}$.
    But, by none values lemma 4.2.3, there are no closed values of type none, so this case ($\varnothing \vdash \mathsf{fun\ end}\, v_2 : \tau$) cannot occur.
  - If the first clause has a variable pattern, $d_1 = x \to e_r$, then rule (E-FUNX) applies to e, so $e \longrightarrow e'$, where $e' = e_r\,[x \mapsto v_2]$.
  - If the first clause has a constant pattern, $d_1 = c\,(x_1, \ldots, x_n) \to e_r$, and v_2 matches this pattern, $v_2 = c\,(v_1', \ldots, v_n')$ for some values v_i', then rule (E-FUNCM) applies to e, so $e \longrightarrow e'$, where $e' = e_r\,[x_1 \mapsto v_1', \ldots, x_n \mapsto v_n']$.
  - If the first clause has a constant pattern, $d_1 = c\,(x_1, \ldots, x_n) \to e_r$, and v_2 does not match this pattern, $v_2 \not= c\,(v_1', \ldots, v_n')$ for any values v_i', then rule (E-FUNCN) applies to e, so $e \longrightarrow e'$, where $e' = \mathsf{fun}\, d_2 \,|\, \ldots \,|\, d_m \mathsf{end}\, v_2$.

Case (T-FIX)**:**

$e = \texttt{fix}\, x\,.\, e_r$

$\varnothing, x : \tau \vdash e_r : \tau$

The rule (E-FIX) always applies to e, so $e \longrightarrow e'$, where $e' = e_r\,[x \mapsto \texttt{fix}\, x\,.\, e_r]$.

Case (T-SUB)**:**

$\varnothing \vdash e : \tau'$

$\tau' <: \tau$

By induction hypothesis, either e is a value or there is some e' such that $e \longrightarrow e'$.

□

4.3 Preservation

Next, it is time to tackle the preservation theorem, giving some more lemmas that are essential for its proof.

4.3.1 Typing Inversion

As we did with the subtyping relation, we start by giving an analogous lemma, describing how typing derivations are built. The *inversion* of the typing relation lemma, given a particular typed expression of some form, informs us about what this expression's type looks like and what types the subexpressions of this expression must have. This lemma depends on the subtyping inversion lemma 4.2.1 to fully break down the relations of the various types involved and has one case for each possible form of expression.

Lemma 4.3.1 (Typing Inversion)**.** *Possible type forms based on given well-typed expression.*

1. *If* $\Gamma \vdash x : \tau$*, then* $\tau_0 <: \tau$*, where* $x : \tau_0 \in \Gamma$.
2. *If* $\Gamma \vdash c\,(e_1, \ldots, e_n) : \tau$*, then* $\tau_0 <: \tau$*, with* $\tau_0 = c\,(\tau_1, \ldots, \tau_n)$*, where* $\Gamma \vdash e_i : \tau_i$ *for all* i.
3. *If* $\Gamma \vdash op\,(e_1, \ldots, e_n) : \tau$*, then* $\tau_0 <: \tau$*, where* $\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau_0$ *and* $\Gamma \vdash e_i : \tau_i$ *for all* i.
4. *If* $\Gamma \vdash \texttt{fun}\,\texttt{end} : \tau$*, then* $\tau_0 <: \tau$*, with* $\tau_0 = \texttt{none} \to \texttt{any}$.
5. *If* $\Gamma \vdash \texttt{fun}\, x \to e_r \,|\, \ldots \,|\, d_m \texttt{end} : \tau$*, then* $\tau_0 <: \tau$*, with* $\tau_0 = \tau_1 \to \tau_2$*, where* $\Gamma, x : \tau_1 \vdash e_r : \tau_2$.
6. *If* $\Gamma \vdash \texttt{fun}\, c\,(x_1, \ldots, x_n) \to e_r \,|\, d_2 \,|\, \ldots \,|\, d_m \texttt{end} : \tau$*,*
   then $\tau_0 <: \tau$*, with* $\tau_0 = (c\,(\tau_1, \ldots, \tau_n) \to \tau_r) \wedge \bigwedge_i ((\neg c\,(\texttt{any}_1, \ldots, \texttt{any}_n) \wedge \tau_i') \to \tau_i'')$*,*
   where $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_r : \tau_r$ *and* $\Gamma \vdash \texttt{fun}\, d_2 \,|\, \ldots \,|\, d_m \texttt{end} : \bigwedge_i (\tau_i' \to \tau_i'')$.
7. *If* $\Gamma \vdash e_1\, e_2 : \tau$*, then* $\tau_0 <: \tau$*, where* $\Gamma \vdash e_1 : \tau' \to \tau_0$ *and* $\Gamma \vdash e_2 : \tau'$.
8. *If* $\Gamma \vdash \texttt{fix}\, x\,.\, e_r : \tau$*, then* $\tau_0 <: \tau$*, where* $\Gamma, x : \tau_0 \vdash e_r : \tau_0$.

Proof. By induction on typing derivations.

The induction proof has the same steps regardless of the form of the given expression e, except of using the corresponding typing rule that matches that specific form, so we present the general case, assuming $\Gamma \vdash e : \tau$ for some e.

By inspection of the typing rules 3.5, we see that each rule matches exclusively one specific expression form, except of rule (T-SUB) that matches any expression form, so the final rule in the derivation of $\Gamma \vdash e : \tau$ must be the one that exclusively matches e or (T-SUB).

If the final rule is the exclusive one, then the desired result is immediate from the rule premises with $\tau_0 = \tau$, so by (S-REFL), $\tau_0 <: \tau$.

If the final rule is (T-SUB), then $\Gamma \vdash e : \sigma$ with $\sigma <: \tau$. By induction hypothesis, there is some τ_0 such that $\tau_0 <: \sigma$ with the desired property. By (S-TRANS), $\tau_0 <: \tau$, which is the desired result.

We conclude that for any well-typed expression e, $\Gamma \vdash e : \tau$, there is some minimal type τ_0 of e, $\Gamma \vdash e : \tau_0$, using the conclusion of the exclusive typing rule for e, such that $\tau_0 <: \tau$ for any type τ of the expression e.

□

4.3.2 Substitution

Here, we state a couple of "structural lemmas" for the typing relation. These are not particularly interesting in themselves, but will permit us to perform some useful manipulations of typing derivations in later proofs. The first one, tells us that we may permute the elements of a typing context, as convenient, without changing the set of typing statements that can be derived under it. The second, allows us to weaken our assumptions recorded in a typing context, by adding a new binding in it (recall that variables names in a context are always kept distinct) about a new unrelated variable.

Lemma 4.3.2 (Permutation)**.** *If $\Gamma \vdash e : \tau$ and Δ is a permutation of Γ, then $\Delta \vdash e : \tau$. Moreover, the latter derivation has the same depth as the former.*

Proof. Straightforward induction on typing derivations.

□

Lemma 4.3.3 (Weakening)**.** *If $\Gamma \vdash e : \tau$ and there is no entry for x in Γ, then $\Gamma, x : \tau' \vdash e : \tau$. Moreover, the latter derivation has the same depth as the former.*

Proof. Straightforward induction on typing derivations.

□

Using these technical lemmas 4.3.2 4.3.3, we can prove a crucial property of the typing relation. That is, well-typedness is preserved when variables are substituted with expressions of the appropriate types. The following lemma states that types are preserved under substitution and is often called just the *substitution lemma*.

Lemma 4.3.4 (Substitution)**.** *If $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e\,[x \mapsto e'] : \tau$.*

Proof. By induction on a derivation of $\Gamma, x : \tau' \vdash e : \tau$. Assuming that the desired property holds for all subderivations, we proceed by case analysis on the final typing rule in the derivation.

Case (T-VAR)**:**
$e = y$
$y : \tau \in (\Gamma, x : \tau')$

- If $y = x$, then $e\,[x \mapsto e'] = e'$ and $\tau = \tau'$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.
- If $y \neq x$, then $e\,[x \mapsto e'] = e$ and $y : \tau \in \Gamma$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-CON)**:**
$e = c\,(e_1, \ldots, e_n)$
$\tau = c\,(\tau_1, \ldots, \tau_n)$
$\Gamma, x : \tau' \vdash e_1 : \tau_1 \ \ldots \ \Gamma, x : \tau' \vdash e_n : \tau_n$
By induction hypothesis, for each e_i, $\Gamma \vdash e_i\,[x \mapsto e'] : \tau_i$.
By (T-CON), $\Gamma \vdash c\,(e_1\,[x \mapsto e'], \ldots, e_n\,[x \mapsto e']) : \tau$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-OP)**:**

$e = op\,(e_1, \ldots, e_n)$
$\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau$
$\Gamma, x : \tau' \vdash e_1 : \tau_1 \ \ldots \ \Gamma, x : \tau' \vdash e_n : \tau_n$
By induction hypothesis, for each e_i, $\Gamma \vdash e_i\,[x \mapsto e'] : \tau_i$.
By (T-OP), $\Gamma \vdash op\,(e_1\,[x \mapsto e'], \ldots, e_n\,[x \mapsto e']) : \tau$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-FUN0)**:**

$e = \mathsf{fun}\,\mathsf{end}$
$\tau = \mathsf{none} \to \mathsf{any}$
It is $e\,[x \mapsto e'] = e$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-FUNX)**:**

$e = \mathsf{fun}\,y \to e_r \,|\, \ldots \,|\, d_m \mathsf{end}$
$\tau = \tau_1 \to \tau_2$
$\Gamma, x : \tau', y : \tau_1 \vdash e_r : \tau_2$
Variable y is bound in e, so we can assume $y \neq x$ and y is not free in e'.
Using permutation lemma 4.3.2, $\Gamma, y : \tau_1, x : \tau' \vdash e_r : \tau_2$.
Using weakening lemma 4.3.3, $\Gamma, y : \tau_1 \vdash e' : \tau'$.
By induction hypothesis, $\Gamma, y : \tau_1 \vdash e_r\,[x \mapsto e'] : \tau_2$.
By (T-FUNX), $\Gamma \vdash \mathsf{fun}\,y \to e_r\,[x \mapsto e'] \,|\, \ldots \,|\, d_m\,[x \mapsto e']\,\mathsf{end} : \tau$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-FUNC)**:**

$e = \mathsf{fun}\,c\,(x_1, \ldots, x_n) \to e_r \,|\, d_2 \,|\, \ldots \,|\, d_m \mathsf{end}$
$\tau = (c\,(\tau_1, \ldots, \tau_n) \to \tau_r) \wedge \bigwedge_i ((\neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \tau_i') \to \tau_i'')$
$\Gamma, x : \tau', x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_r : \tau_r$
$\Gamma, x : \tau' \vdash \mathsf{fun}\,d_2 \,|\, \ldots \,|\, d_m \mathsf{end} : \bigwedge_i (\tau_i' \to \tau_i'')$
Variables $x_1, \ldots, x_n$ are bound in e, so for each x_i, we can assume $x_i \neq x$ and x_i is not free in e'.
Using permutation lemma 4.3.2, $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n, x : \tau' \vdash e_r : \tau_r$.
Using weakening lemma 4.3.3, $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e' : \tau'$.
By induction hypothesis for e_r, $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_r\,[x \mapsto e'] : \tau_r$.
By induction hypothesis for $\mathsf{fun}\,d_2 \,|\, \ldots \,|\, d_m \mathsf{end}$, $\Gamma \vdash \mathsf{fun}\,d_2 \,|\, \ldots \,|\, d_m \mathsf{end}\,[x \mapsto e'] : \bigwedge_i (\tau_i' \to \tau_i'')$,
so $\Gamma \vdash \mathsf{fun}\,d_2\,[x \mapsto e'] \,|\, \ldots \,|\, d_m\,[x \mapsto e']\,\mathsf{end} : \bigwedge_i (\tau_i' \to \tau_i'')$.
By (T-FUNC), $\Gamma \vdash \mathsf{fun}\,c\,(x_1, \ldots, x_n) \to e_r\,[x \mapsto e'] \,|\, d_2\,[x \mapsto e'] \,|\, \ldots \,|\, d_m\,[x \mapsto e']\,\mathsf{end} : \tau$,
so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-APP)**:**

$e = e_1\,e_2$
$\Gamma, x : \tau' \vdash e_1 : \tau'' \to \tau$
$\Gamma, x : \tau' \vdash e_2 : \tau''$
By induction hypothesis for e_1, $\Gamma \vdash e_1\,[x \mapsto e'] : \tau'' \to \tau$.
By induction hypothesis for e_2, $\Gamma \vdash e_2\,[x \mapsto e'] : \tau''$.
By (T-APP), $\Gamma \vdash e_1\,[x \mapsto e']\ e_2\,[x \mapsto e'] : \tau$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-FIX)**:**

$e = \mathtt{fix}\,y\,.\,e_r$
$\Gamma, x : \tau', y : \tau \vdash e_r : \tau$
Variable y is bound in e, so we can assume $y \neq x$ and y is not free in e'.
Using permutation lemma 4.3.2, $\Gamma, y : \tau, x : \tau' \vdash e_r : \tau$.
Using weakening lemma 4.3.3, $\Gamma, y : \tau \vdash e' : \tau'$.
By induction hypothesis, $\Gamma, y : \tau \vdash e_r\,[x \mapsto e'] : \tau$.
By (T-FIX), $\Gamma \vdash \mathtt{fix}\,y\,.\,e_r\,[x \mapsto e'] : \tau$, so $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

Case (T-SUB)**:**
$\Gamma, x : \tau' \vdash e : \tau''$
$\tau'' <: \tau$
By induction hypothesis, $\Gamma \vdash e\,[x \mapsto e'] : \tau''$.
By (T-SUB), $\Gamma \vdash e\,[x \mapsto e'] : \tau$.

□

4.3.3 Preservation

Using the substitution lemma 4.3.4, we can now prove the other half of the type safety property, that evaluation preserves well-typedness. In particular, the *preservation* theorem, states that, if a well-typed expression can take an evaluation step, then the resulting expression will also be well-typed and, actually, has the same type.

Theorem 4.3.5 (Preservation)**.** *If* $\Gamma \vdash e : \tau$ *and* $e \longrightarrow e'$*, then* $\Gamma \vdash e' : \tau$.

Proof. By induction on a derivation of $\Gamma \vdash e : \tau$. Assuming that the desired property holds for all subderivations, we proceed by case analysis on the final typing rule in the derivation. Then for each case, we consider the possible evaluation rules by which $e \longrightarrow e'$ can be derived.

Case (T-VAR)**:**
$e = x$
In this case, e has the form of a variable and there are no possible evaluation rules for variables, so this case cannot occur.

Case (T-CON)**:**
$e = c\,(e_1, \ldots, e_n)$
$\tau = c\,(\tau_1, \ldots, \tau_n)$
$\Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n$
The only possible evaluation rule, that can derive $e \longrightarrow e'$ for some e', is (E-CON).

Subcase (E-CON)**:**
$e' = c\,(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n)$
all e_j, $j < i$, are values, $e_j = v_j$
$e_i \longrightarrow e'_i$
By induction hypothesis, $\Gamma \vdash e'_i : \tau_i$.
By (T-CON), $\Gamma \vdash e' : \tau$.

Case (T-OP)**:**
$e = op\,(e_1, \ldots, e_n)$
$\varnothing \vdash op : (\tau_1, \ldots, \tau_n) \to \tau$
$\Gamma \vdash e_1 : \tau_1 \ \ldots \ \Gamma \vdash e_n : \tau_n$
The only possible evaluation rules, that can derive $e \longrightarrow e'$ for some e', are (E-OP) and (E-OPF).

Subcase (E-OP)**:**
$e' = op\,(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n)$
all e_j, $j < i$, are values, $e_j = v_j$
$e_i \longrightarrow e'_i$
By induction hypothesis, $\Gamma \vdash e'_i : \tau_i$.
By (T-OP), $\Gamma \vdash e' : \tau$.

Subcase (E-OPF)**:**
$e' = [\![op]\!]\,(v_1, \ldots, v_n)$

all e_i are values, $e_i = v_i$
For every operator op of type $(\tau_1, \ldots, \tau_n) \to \tau$, if its semantic function $[\![op]\!]$ is applied over some values $v_1, \ldots, v_n$ in its domain $(\tau_1, \ldots, \tau_n)$, then it returns a value $v = [\![op]\!]\,(v_1, \ldots, v_n)$ in its codomain τ. In our case, $[\![op]\!]\,(v_1, \ldots, v_n)$ is a resulting value of a valid application of operator's op semantic function, so $\Gamma \vdash e' : \tau$.

Case (T-FUN0)**,** (T-FUNX)**,** (T-FUNC)**:**
$e = \mathsf{fun}\, d_1 \,|\, \ldots \,|\, d_m \mathsf{end}$
In all these cases, e has the form of a function and there are no possible evaluation rules for functions, so this case cannot occur.

Case (T-APP)**:**
$e = e_1\, e_2$
$\Gamma \vdash e_1 : \tau' \to \tau$
$\Gamma \vdash e_2 : \tau'$
The only possible evaluation rules, that can derive $e \longrightarrow e'$ for some e', are (E-APP1), (E-APP2), (E-FUNX), (E-FUNCM), and (E-FUNCN).

Subcase (E-APP1)**:**
$e' = e'_1\, e_2$
$e_1 \longrightarrow e'_1$
By induction hypothesis, $\Gamma \vdash e'_1 : \tau' \to \tau$.
By (T-APP), $\Gamma \vdash e' : \tau$.

Subcase (E-APP2)**:**
$e' = v_1\, e'_2$
e_1 is a value, $e_1 = v_1$
$e_2 \longrightarrow e'_2$
By induction hypothesis, $\Gamma \vdash e'_2 : \tau'$.
By (T-APP), $\Gamma \vdash e' : \tau$.

Subcase (E-FUNX)**:**
$e' = e_r\,[x \mapsto v_2]$
$e_1 = v_1 = \mathsf{fun}\, x \to e_r \,|\, \ldots \,|\, d_m \mathsf{end}$
e_2 is a value, $e_2 = v_2$
By typing inversion lemma 4.3.1 for e_1, $\sigma' \to \sigma <: \tau' \to \tau$, where $\Gamma, x : \sigma' \vdash e_r : \sigma$.
By subtyping inversion lemma 4.2.1, $\tau' <: \sigma'$ and $\sigma <: \tau$.
By (T-SUB), $\Gamma \vdash v_2 : \sigma'$.
By substitution lemma 4.3.4, $\Gamma \vdash e' : \sigma$.
By (T-SUB), $\Gamma \vdash e' : \tau$.

Subcase (E-FUNCM)**:**
$e' = e_r\,[x_1 \mapsto v'_1, \ldots, x_n \mapsto v'_n]$
$e_1 = v_1 = \mathsf{fun}\, c\,(x_1, \ldots, x_n) \to e_r \,|\, d_2 \,|\, \ldots \,|\, d_m \mathsf{end}$
$e_2 = v_2 = c\,(v'_1, \ldots, v'_n)$ for some values v'_i
By typing inversion lemma 4.3.1 for e_2, $c\,(\tau_1, \ldots, \tau_n) <: \tau'$, where $\Gamma \vdash v'_i : \tau_i$ for all i.
By (S-ARROW) and (T-SUB), $\Gamma \vdash e_1 : c\,(\tau_1, \ldots, \tau_n) \to \tau$.
By typing inversion lemma 4.3.1 for e_1,
$(c\,(\sigma_1, \ldots, \sigma_n) \to \sigma) \wedge \bigwedge_i ((\neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma'_i) \to \sigma''_i) <: c\,(\tau_1, \ldots, \tau_n) \to \tau$,
where $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e_r : \sigma$ and $\Gamma \vdash \mathsf{fun}\, d_2 \,|\, \ldots \,|\, d_m \mathsf{end} : \bigwedge_i (\sigma'_i \to \sigma''_i)$.
By subtyping inversion lemma 4.2.1, either $c\,(\sigma_1, \ldots, \sigma_n) \to \sigma <: c\,(\tau_1, \ldots, \tau_n) \to \tau$
or for some i, $(\neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma'_i) \to \sigma''_i <: c\,(\tau_1, \ldots, \tau_n) \to \tau$.
Let us assume that for some i, $(\neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma'_i) \to \sigma''_i <: c\,(\tau_1, \ldots, \tau_n) \to \tau$.
By subtyping inversion lemma 4.2.1, $c\,(\tau_1, \ldots, \tau_n) <: \neg c\,(\mathsf{any}_1, \ldots, \mathsf{any}_n)$ and $\sigma''_i <: \tau$.

The first subtype relation can only be true if $\tau_i <: \mathsf{none}$ for some i, so by (T-SUB), $\Gamma \vdash v_i' : \mathsf{none}$ for some i. But, by none values lemma 4.2.3, there are no values of type none, so this case cannot occur.

So we have that $c(\sigma_1, \ldots, \sigma_n) \to \sigma <: c(\tau_1, \ldots, \tau_n) \to \tau$.
By subtyping inversion lemma 4.2.1, $c(\tau_1, \ldots, \tau_n) <: c(\sigma_1, \ldots, \sigma_n)$ and $\sigma <: \tau$.
From the first subtype relation, we have $\tau_i <: \sigma_i$ for all i,
so by (T-SUB), $\Gamma \vdash v_i' : \sigma_i$ for all i.
By substitution lemma 4.3.4, $\Gamma \vdash e' : \sigma$.
By (T-SUB), $\Gamma \vdash e' : \tau$.

Subcase (E-FUNCN)**:**

$e' = \mathsf{fun}\, d_2 \mid \ldots \mid d_m \mathsf{end}\, v_2$
$e_1 = v_1 = \mathsf{fun}\, c(x_1, \ldots, x_n) \to e_r \mid d_2 \mid \ldots \mid d_m \mathsf{end}$
e_2 is a value, $e_2 = v_2$
$v_2 \neq c(v_1', \ldots, v_n')$ for any values v_i'
By typing inversion lemma 4.3.1 for e_1,
$(c(\sigma_1, \ldots, \sigma_n) \to \sigma) \wedge \bigwedge_i ((\neg c(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma_i') \to \sigma_i'') <: \tau' \to \tau$,
where $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e_r : \sigma$ and $\Gamma \vdash \mathsf{fun}\, d_2 \mid \ldots \mid d_m \mathsf{end} : \bigwedge_i (\sigma_i' \to \sigma_i'')$.
By subtyping inversion lemma 4.2.1, either $c(\sigma_1, \ldots, \sigma_n) \to \sigma <: \tau' \to \tau$
or for some i, $(\neg c(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma_i') \to \sigma_i'' <: \tau' \to \tau$.

Let us assume that $c(\sigma_1, \ldots, \sigma_n) \to \sigma <: \tau' \to \tau$.
By subtyping inversion lemma 4.2.1, $\tau' <: c(\sigma_1, \ldots, \sigma_n)$ and $\sigma <: \tau$.
By (T-SUB), $\Gamma \vdash v_2 : c(\sigma_1, \ldots, \sigma_n)$.
By canonical forms lemma 4.2.2, v_2 has the form $c(v_1', \ldots, v_n')$ for some values v_i'.
But we know that v_2 cannot have this form for any values v_i', so this case cannot occur.

So we have that for some i, $(\neg c(\mathsf{any}_1, \ldots, \mathsf{any}_n) \wedge \sigma_i') \to \sigma_i'' <: \tau' \to \tau$.
By subtyping inversion lemma 4.2.1, $\tau' <: \sigma_i'$ and $\sigma_i'' <: \tau$.
By (S-INTER1) and (T-SUB), for all i, $\Gamma \vdash \mathsf{fun}\, d_2 \mid \ldots \mid d_m \mathsf{end} : \sigma_i' \to \sigma_i''$.
By (T-SUB), $\Gamma \vdash v_2 : \sigma_i'$.
By (T-APP), $\Gamma \vdash e' : \sigma_i''$.
By (T-SUB), $\Gamma \vdash e' : \tau$.

Case (T-FIX)**:**

$e = \mathsf{fix}\, x.e_r$
$\Gamma, x : \tau \vdash e_r : \tau$
The only possible evaluation rule, that can derive $e \longrightarrow e'$ for some e', is (E-FIX).

Subcase (E-FIX)**:**

$e' = e_r[x \mapsto \mathsf{fix}\, x.e_r]$
The expression that takes the place of x in e' is e, $e' = e_r[x \mapsto e]$,
so by substitution lemma 4.3.4, $\Gamma \vdash e' : \tau$.

Case (T-SUB)**:**

$\Gamma \vdash e : \tau'$
$\tau' <: \tau$
By induction hypothesis, $\Gamma \vdash e' : \tau'$.
By (T-SUB), $\Gamma \vdash e' : \tau$.

□

Chapter 5

Future Work and Open Questions

We presented our complex but expressive type system for our simple functional language featuring pattern matching and we also showed that our definitions are consistent and that our type system appropriately assigns types to expressions, having the type safety property. But, in order for a type system to be useful in practice, we should also have a simple way of deciding what types we assign to each expression.

Currently, we have no way of figuring out if a particular expression can be assigned any type in our type system. The most important obstacle is the lack of any type annotations in the language. That is, the typing relation has to guess the types of function arguments and then check if the function body can be typed assuming these types. Type reconstruction mechanisms that work with no type annotations are possible, like type inference in ML-family programming languages, but require more strict constraints on what type functions can have.

The problem of type inference in our language, as is, seems to be undecidable. We have to restrict in some way what type arguments and results of functions can have. One way is to put constraints on each function clause based on the types of the other clauses. Maybe, function clauses do not all need to have the same exact type (as ML type inference requires), but they should share some general common type forms, known beforehand to reduce the type search space. Another way is to explicitly reduce the search space with type annotations. Each function could list a finite sequence of possible argument types and then we can just check the function's result type assuming each argument type in turn, keeping only types that lead to a well-typed return expression. Both solutions lead to a series of changes in our expressive type system and their analysis would be interesting in search for a type reconstruction algorithm.

The simpler problem of the decision of the typing relation does not seem easily solvable either. That is, we would like given an expression and a specific type to answer if that pair can be derived from our typing relation. First, we should reorganize the typing rules to be only syntax oriented, that is figure out the exact points subsumption needs to be used and change our rules accordingly with subtype premises and abandon the subsumption rule. Then, of course we should be able to decide these subtype premises, that is, our subtyping relation should also be decidable. Similar reorganization of the rules of the subtyping relation is needed, incorporating syntax ambiguous rules, like reflexive and transitive properties, to all other rules and removing them from the relation. A step to this direction, for the current subtyping relation, is already done with its inversion lemma.

Of course, the current subtyping relation limits our type system. Only a small part of our intended set-theoretic semantics for type constructors is achieved by it. That is, our subtyping relation is sound, but far from complete. Experimenting with new subtype rules, that add on our semantics of type constructors is needed. The big question of what are the limits of the syntactic approach of defining subtyping is also interesting to look at. In particular, we need a series of rewriting rules for complex type expressions, to be able to simplify the types we assign to functions, making the first steps to deal with the great problem of type equivalence. Part of these type simplifications might be possible using SMT solvers, taking some appropriate transformation of our type expressions that also retain our type information.

All these changes and experimentation with the type system is costly, because we have to make sure our proofs and desired properties still hold. As we learned from dealing with a complex type

system, and can be quickly obvious to anyone browsing through the pages of this work, proofs tend to be quite large and specific cases can be very complex. The implementation of the language and type system definitions in a proof assistant environment, even though far from trivial, would be a great time investment, in order to automate a lot of the proof checking process and have machine error-free verification.

Bibliography

[Aho86] Alfred V Aho, Ravi Sethi and Jeffrey D Ullman, "Compilers, principles, techniques", *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[Copp78] Mario Coppo and Mariangiola Dezani-Ciancaglini, "A new type assignment for λ-terms", *Archiv für mathematische Logik und Grundlagenforschung*, vol. 19, no. 1, pp. 139–156, 1978.

[Fris08] Alain Frisch, Giuseppe Castagna and Véronique Benzaken, "Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types", *Journal of the ACM (JACM)*, vol. 55, no. 4, p. 19, 2008.

[Miln97] Robin Milner, Mads Tofte, Robert Harper and David MacQueen, *The definition of standard ML: revised*, MIT press, 1997.

[Pier02] Benjamin C Pierce and C Benjamin, *Types and programming languages*, MIT press, 2002.

[Plot81] Gordon Plotkin, "Structural operational semantics", *Aarhus University, Denmark*, 1981.