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

Μεταγλώττιση αμοιβαία αναδρομικών τύπων σε μία συναρτησιακή γλώσσα έξυπνων συμβολαίων στο blockchain

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΚΟΥΜΑΣ ΒΑΣΙΛΕΙΟΣ

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

Αθήνα, Μάρτιος 2019

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

Μεταγλώττιση αμοιβαία αναδρομικών τύπων σε μία συναρτησιακή γλώσσα έξυπνων συμβολαίων στο blockchain

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΓΚΟΥΜΑΣ ΒΑΣΙΛΕΙΟΣ

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

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Μαρτίου 2019.

Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Ι. Γκούμας Αν. Καθηγητής Ε.Μ.Π

Αθήνα, Μάρτιος 2019

Γκούμας Βασίλειος

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

Περίληψη

Στην εργασία αυτή θα παρουσιάσουμε την FIR, μια συναρτησιακή ενδιάμεση αναπαράσταση, στενά συνδεδεμένη με τον λ-λογισμό, που μπορεί να χρησιμοποιηθεί κατά την μεταγλώττιση των προγραμμάτων από μία γλώσσα υψηλού επιπέδου σε μία γλώσσα στόχο. Η FIR υποστηρίζει χαρακτηριστικά υψηλότερης τάξης, όπως οι συναρτήσεις τύπων και ο πολυμορφισμός και έχει μπορεί να κωδικοποιήσει για αναδρομικούς τύπους δεδομένων. Οι τεχνικές που θα χρησιμοποιήσουμε, παρόλο που μπορούν να εντοπιστούν στην βιβλιογραφία, δεν έχουν συνδυαστεί ξανά κατά αυτόν τον τρόπο.

Η FIR δεν αποτελεί μια καθαρά ακαδημαική άσκηση, καθώς χρησιμοποιείται στην ανάπτυξη του Plutus, μιας αρχιτεκτονικής για smart contracts ως ενδιάμεσο βήμα κατά την μεταγλώττιση του κώδικα Haskell που γράφει ο τελικός χρήστης, σε μια γλώσσα χαμηλού επιπέδου, που στη συνέχεια εκτελείται στο blockchain.

Αρχικά θα δώσουμε το κίνητρο για όσα θα κάνουμε, την ανάπτυξη μιας ασφαλούς γλώσσας για χρήση στο blockchain. Αφού παρουσιάσουμε το συντακτικό και την σύνθεση τύπων της FIR, θα επικεντρωθούμε στην μεταγλώττιση ορισμένων χαρακτηριστικών της στην System F^{μ}_{ω}, μια θεωρητική επέκταση του απλού λ-λογισμού.

Λέξεις κλειδιά

Γλώσσες προγραμματισμού, Haskell, λ-λογισμός, μεταγλωττιστές, συστήματα τύπων, blockchain, έξυπνα συμβόλαια, System F_{ω}, αμοιβαία αναδρομικοί τύποι

Abstract

In this diploma thesis we present FIR, a functional intermediate representation, heavily influenced by the System F_ω, that can be used during the compilation step from a high-level source language to a target l anguage. FIR has support for higher-order features like type-level functions, polymorphism, and can encode mutually recursive datatypes. The techniques that we use, although known in the literature, have not been combined in that way before.

FIR is not a purely academic exploration, but is used in the development of Plutus, a smart contract platform, as an intermediate representation in the compilation of the Haskell code written by the end-user, to a lower-level language that goes into the blockchain.

We will start by providing the motivation for our work, which is a safe blockchain language. After presenting the syntax and type synthesis in FIR, we will focus on the compilation of certain features of the language to System F^{μ}_{ω}, a theoretical extension of lambda calculus.

Key words

Programming languages, Haskell, λ-calculus, compilers, type systems, blockchain, smart contracts, System F_ω, mutually recursive datatypes

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή της εργασίας, κ. Νίκο Παπασπύρου για τις υποδείξεις του καθώς και τον Manuel Chakravarty, υπεύθυνο της ομάδας των μεταγλωττιστών στην IOHK, για την πρότασή του να δουλέψω στο συγκεκριμένο θέμα. Θα ήθελα επίσης να ευχαριστήσω τους Michael Peyton Jones και Roman Kireev για τις συζητήσεις μας και την πολύτιμη καθοδήγήση τους. Ιδιαίτερη μνεία θέλω να δώσω και στον κ. Δημήτρη Φωτάκη, κ. Νεκτάριο Κοζύρη και κ. Μιχαήλ Λουλάκη για την έμπνευσή που μου έδωσαν κατά την διάρκεια των σπουδών μου. Τέλος, δεν γίνεται να μην αναφερθώ σε όλους τους φίλους που ήταν δίπλα μου κατά την διάρκεια των σπουδών μου και με έκαναν πιο πλούσιο άνθρωπο, καθώς και στην οικογένεια μου για την στήριξή τους σε αυτό το ταξίδι.

Γκούμας Βασίλειος,

Αθήνα, 26η Μαρτίου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-19, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Μάρτιος 2019.

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

Περιεχόμενα

Κατάλογος σχημάτων

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Η εργασία αυτή αποσκοπεί στον ορισμό μιας συναρτησιακής ενδιάμεσης γλώσσας προγραμματισμού για χρήση από μεταγλωττιστές, για χρήση ως ενδιάμεσο βήμα κατά την μεταγλώττιση από μία γλώσσα προέλευσης σε μία γλώσσα στόχο. Η γλώσσα αυτή αποτελεί προέκταση του απλού λ-λογισμού, εμπλουτισμένο με πολυμορφισμό, αναδρομικούς τύπους μεγαλύτερης τάξης και συναρτήσεις στο επίπεδο των τύπων και των kinds. Επιπλέον, η γλώσσα υποστηρίζει αναδρομικά `let` στο επίπεδο των όρων και στο επίπεδο των τύπων, χωρίς να βασίζεται στην οκνηρή αποτίμηση του λ-λογισμού στη βάση της, γεγονός που δεν έχει εξερευνηθεί στην βιβλιογραφία.

Στην συνέχεια δείχνουμε πώς μπορούν να μεταγλωττιστούν χαρακτηριστικά της παραπάνω ενδιάμεσης γλώσσας στο σύστημα System F^{μ}_{ω} (System F_{ω} εμπλουτισμένο με αναδρομικούς τύπους). Συγκεκριμένα στην παρούσα εργασία επικεντρωνόμαστε στην μεταγλώττιση των αναδρομικών και αμοιβαία αναδρομικών τύπων δεδομένων. Τέλος θα δούμε την υλοποίηση αυτής της μεταγλώττισης όπως είναι υλοποιημένη σε πραγματικό μεταγλωττιστή σε Haskell.

Η προέκταση αυτής της εργασίας παρουσιάζει και την μεταγλώττιση των (αμοιβαία) αναδρομικών όρων, τεχνική που δεν έχει εξερευνηθεί στην βιβλιογραφία.

1.2 Κίνητρο

Οι περισσότεροι μεταγλωττιστές κάνουν χρήση ενδιάμεσων αναπαραστάσεων κατά την μεταγλώττιση του πηγαίου κώδικα από την προερχόμενη γλώσσα στην γλώσσα στόχο. Αποτελεί συνήθη πρακτική, ειδικά για τους μεταγλωττιστές συναρτησιακών γλωσσών, η ενδιάμεση αυτή γλώσσα να είναι επέκταση κάποιας εκδοχής του λ-λογισμού, λόγω της απλότητας και εκφραστικότητας της αναπαράστασης.

Ταυτόχρονα όμως, η χρήση τέτοιων γλωσσών έχει το μειονέκτημα ότι καθώς προορίζεται για χρήση από το επόμενο στάδιο της μεταγλώττισης, συνήθως για βελτιστοποίηση και επιπλέον περάσματα, δεν είναι εύκολο να κατανοηθεί και να γραφεί από άνθρωπο. Επίσης το βήμα της μεταγλώττισης γίνεται συνήθως αρκετά μεγάλο, γεγονός που καθιστά συμφέρουσα την ανάλυση της μεταγλώττισης σε μικρότερα βήματα.

Η χρήση μιας ενδιάμεσης γλώσσας με ισχυρούς τύπους έχει προταθεί παλαιότερα ([Jone97]). Οι εν λόγω συγγραφείς βασίζουν την γλώσσα τους σε λ-λογισμό με εξαρτημένους τύπους. Για την ανάπτυξη των τεχνικών που θα παρουσιαστούν δεν είναι αναγκαία η επιστράτευση τόσο δυνατού λογισμού. Κατά αυτόν τον τρόπο λοιπόν κάνουμε τις λιγότερες δυνατές προσθήκες που επιτρέπουν την έκφραση των επιθυμητών χαρακτηριστικών στην γλώσσα.

Η ενδιάμεση γλώσσα που ορίζουμε επιτρέπει την χρήση `let`-bindings για όρους και για τύπους δεδομένων, προσφέροντας δομές προγραμματισμού πιο κοντά στον χρήστη.

Η γλώσσα που παρουσιάζουμε χρησιμοποιείται σαν ενδιάμεση αναπαράσταση στην αρχιτεκτονική Plutus, μια πλατφόρμα έξυπνων συμβολαίων βασισμένο στο Cardano blockchain. Σύντομη περιγραφή της αρχιτεκτονικής γίνεται στο κεφάλαιο 2.3.1.

1.3 Δομή της εργασίας

Αρχικά γίνεται μια περιγραφή του τομέα εφαρμογής των κύριων ιδεών της εργασίας, δηλαδή του blockchain και των έξυπνων συμβολαίων (2.

Στο κεφάλαιο 2 παρουσιάζεται η απαραίτητη θεωρία γλωσσών και αναδρομικών τύπων για την παρουσίαση των τεχνικών των επόμενων κεφαλαίων.

Στη συνέχεια έχουμε τον ορισμό και τις ιδιότητες της γλώσσας System F^{μ}_{ω}, άμεσα προερχόμενη από τον λ-λογισμό και της FIR, που επεκτείνει την System F^{μ}_{ω} με `let`-bindings (κεφάλαιο 4)

Προχωρώντας στο κεφάλαιο 5 γίνεται η μεταγλώττιση των αμοιβαία αναδρομικών τύπων δεδομένων από την FIR στην System F^{μ}_{ω}. Ο χειρισμός των αμοιβαία αναδρομικών τύπων απαιτεί την χρήση τεχνικών από την βιβλιογραφία γύρω από τον *γενικευμένο προγραμματισμό* (generic programming). Συγκεκριμένα, θα συζητηθούν οι *δεικτοδοτημένοι τελεστές σταθερού σημείου* (indexed fixpoints) και ετικέτες στο επίπεδο των τύπων (type-level tags) ώστε να συνδυαστούν διαφορετικοί αμοιβαία αναδρομικοί τύποι σε έναν κοινό τύπο. Η τεχνική αυτή αναφέρεται στο [Yaku09], παρόλα αυτά η προσέγγιση που ακολουθείται εδώ είναι διαφορετική, καθώς βάζει στην εξίσωση και την "ορθότερη" κωδικοποίηση Scott, και έχουμε σαν στόχο την System F^{μ}_{ω}, σε αντίθεση με μία γλώσσα με πλήρης εξαρτημένους τύπους (dependent types), όπως οι παραπάνω συγγραφείς.

Καταλήγοντας, στο κεφάλαιο 6 αναφέρονται συμπεράσματα, μειονεκτήματα και μελλοντικές κατευθύνσεις αυτής της δουλειάς, μαζί με παραδείγματα κάποιων βελτιστοποιήσεων που ενεργοποιεί η χρήση `let` όρων στην γλώσσα.

Κεφάλαιο 2

Blockchain & Smart Contracts

Η τεχνολογία του blockchain παρουσιάστηκε για πρώτη φορά από τον Satoshi Nakamoto το 2008 [Naka08], και αποτελεί την πρώτη υλοποίηση ψηφιακού νομίσματος, που λύνει αποκεντρωμένα το πρόβλημα του διπλοξοδέματος (double spending). Αποτελεί ένα μέσο επίλυσης χρηματικών συναλλαγών χωρίς την επέμβαση κάποιας κεντρικής αρχής, που βασίζεται σε κρυπτογραφικές μεθόδους για την επίτευξη της συναίνεσης των συμμετεχόντων.

Σύντομα μετά την εν λόγω δημοσίευση που περιγράφει τις αρχές λειτουργίας του bitcoin και εισάγει την έννοια του blockchain, δημοσιεύθηκε ως λογισμικό ανοιχτού κώδικα η υλοποίηση του συστήματος σε C++ το 2009, πάλι από τον Satoshi Nakamoto και την ομάδα του, ο οποίος λίγο αργότερα εξαφανίστηκε χωρίς να γνωρίζουμε την πραγματική του ταυτότητα. μέχρι και σήμερα.

Η δημιουργία ενός ψηφιακού νομίσματος ελεύθερο από τον έλεγχο μιας κεντρικής αρχής απασχολούσε αρκετά χρόνια την κοινότητα των cypherpunks την δεκαετία του 90', όπου μπήκαν τα θεμέλια για την εισαγωγή της τεχνολογίας του blockchain από τον Nakamoto, όπως η χρήση των Merkle trees και των ψηφιακών υπογραφών.

Απο τότε, η τεχνολογία του blockchain έχει προσελκύσει μεγάλο ενδιαφέρον λόγω των εφαρμογών της και θεωρείται από τις πιο πρωτοποριακές τεχνολογικές εφευρέσεις του 21ου αιώνα.

2.1 Smart Contracts

Η έννοια των έξυπων συμβολαίων (smart contracts), προηγείται αυτής του Bitcoin και μπορεί να εντοπιστεί στο 1997, από τον Nick Szabo [Szab97], ο οποίος οραματίστηκε αρκετές από τις σημερινές εφαρμογές των έξυπνων συμβολαίων, όπως η εκκαθάριση πληρωμών, η διαχείριση και μεταφορά ιδιοκτησίας, η εμπιστοσύνη σε ανώνυμα και ψευδώνυμα δίκτυα, και η αποκεντρωμένη εκδοχή κλασσικών χρηματοοικονομικών συμβολαίων όπως τα παράγωγα. Προτείνει ακόμα μία γλώσσα για χρήση στον προγραμματισμό έξυπνων συμβολαίων, βαθιά επηρεασμένη από τον συναρτησιακό προγραμματισμό. [Szab02a]
Ο όρος έξυπνο συμβόλαιο απέκτησε νέα σημασία με την εισαγωγή της τεχνολογίας του blockchain. Η πρώτη πλατφόρμα που υποστήριξε έξυπνα συμβόλαια ήταν το Ethereum.

2.1.1 Ethereum

Η πλατφόρμα του Ethereum έχει αρκετά κοινά με το Bitcoin, όπως ότι η λειτουργία του βασίζεται σε ένα *αποκεντρωμένο δίκτυο κόμβων*, η ύπαρξη ενός *αλγορίθμου συναίνεσης* (consensus) μεταξύ των κόμβων, που τους επιτρέπει να διατηρήσουν μια κατανεμημένη βάση δεδομένων. Στην περίπτωση του Bitcoin η δομή αυτή συμφωνεί στο "ποιος έχει τι".

Το Ethereum επεκτείνει την ιδέα αυτή, χτίζοντας πάνω στην τεχνολογία των κρυπτονομισμάτων της εποχής. Κάνοντας χρήση του στρώματος συναίνεσης που μοιράζει σωστά και δίκαια τους πόρους του δικτύου, παρέχει έξυπνα συμβόλαια με αρκετά μεγαλύτερη εκφραστικότητα. Σε αντίθεση με το Bitcoin, του οποίου η γλώσσα επαλήθευσης των συναλλαγών περιορίζεται στο να επαληθεύει αν ισχύουν οι συνθήκες που επιτρέπουν το ξόδεμα των πόρων, η γλώσσα του Ethereum είναι Turing-complete. Αυτό πρακτικά σημαίνει ότι τα έξυπνα συμβόλαια μπορούν πλέον να εκφράσουν οποιον-

δήποτε υπολογισμό μπορεί να γίνει με μία mainstream γλώσσα προγραμματισμού. Το γεγονός αυτό έχει δώσει στο Ethereum την ονομασία “κατανεμημένος παγκόσμιος υπολογιστής”.

2.1.2 Εφαρμογές

Οι δυνατότητες που παρέχει η τεχνολογία του blockchain σε συνδυασμό με έξυπνα συμβόλαια γενικού σκοπού βρίσκουν πληθώρα εφαρμογών σε διάφορες βιομηχανίες και κλάδους. Το κύριο χαρακτηριστικό τους που κάνει δελεαστική την χρήση τους σε παραδοσιακές βιομηχανίες είναι ότι αποτελούν συμβόλαια που μπορούν να εκτελεστούν και να επιβληθούν αυτόματα χωρίς την επέμβαση μεσάζοντα, απλοποιώντας έτσι πολλές πτυχές των σημερινών βιομηχανιών όπως στον *εφοδιασμό* (supply chain) σε *συναλλαγές και πληρωμές*, *μεταφορά ακινήτων*, *ασφάλεια - υπηρεσίες υγείας* και *ηλεκτρονικές ψηφοφορίες*

Οι παραπάνω βιομηχανικοί κλάδοι αποτελούν παράδειγμα περιπτώσεων χρήσης που μπορούν να ωφεληθούν μειώνοντας τα κόστη λειτουργίας μέσω της χρήσης έξυπνων συμβολαίων. Υπάρχουν όμως και εφαρμογές που έχουν γίνει δυνατές μόνο μέσω της νέας τεχνολογίας αυτής, όπως οι *αγορές προβλέψεων* (prediction markets) ([augu18]) και οι *αποκεντρωμένοι αυτόνομοι οργανισμοί* (DAO).

Οι χρήσεις αυτές των έξυπνων συμβολαίων αφορούν παραδοσιακούς τομείς που ωφελούνται σε μεγάλο βαθμό από την πιο αποδοτικότερη εκτέλεση των συμβολαίων - συμφωνιών, αλλά και δίνουν δυνατότητες για δημιουργία εφαρμογών με γνώμονα την ιδιωτικότητα, την ασφάλεια, την ακρίβεια, την ταχύτητα και την διαφάνεια.

Είναι αναγκαίο επομένως η γραφή, ο έλεγχος και η επαλήθευση των προγραμμάτων αυτών, ώστε να μπορέσουν να ενσωματωθούν ομαλώς στην σε καθημεριν εφαρμογές .

2.2 Γλώσσες συγγραφής έξυπνων συμβολαίων

Το θέμα της επιλογής κατάλληλης γλώσσας για τον προγραμματισμό έξυπνων συμβολαίων έχει απασχολήσει αρκετά τους ερευνητές στον τομέα ακόμα και πριν την έλευση του blockchain. Στο κεφάλαιο 2.3 θα δούμε και άλλα παραδείγματα εφαρμογής του συναρτησιακού προγραμματισμού σε DSLs που προορίζονται για περιγραφή οικονομικών συμβολαίων.

Η υλοποίηση των έξυπων συμβολαίων στο Bitcoin γίνεται δυνατή μέσω της γλώσσας *Bitcoin Script*, η εκτελέση της οποίας γίνεται με τον χειρισμό μίας δομής στοίβας. Κάθε συναλλαγή που εκτελεί ο τελικός χρήστης, μεταφράζεται σε μια σειρά εντολών στην γλώσσα Bitcoin Script.

Η γλώσσα προγραμματισμού του Ethereum τρέχει πάνω από την εικονική μηχανή EVM [Wood14]. Η γλώσσα αυτή, όπως και αυτή που θα εξετάσουμε στα κεφάλαια 4. 5 είναι αρκετά χαμηλού επιπέδου, και δεν είναι σχεδιασμένη για να γράφεται ή να διαβάζεται από τον προγραμματιστή, αλλά να αποτελεί την “assembly-type” γλώσσα που θα γράφεται στο blockchain, διαθέσιμη για εξέταση αν αυτό χρειαστεί.

Η ύπαρξη μίας εικονικής μηχανής για την εκτέλεση έξυπνων συμβολαίων δίνει την δυνατότητα για επαλήθευση των προγραμμάτων που εκτελούνται σε αυτές. Αυτό μπορεί να γίνει με το verification αυτής της εικονικής μηχανής, που μπορεί να γίνει με τεχνικές *τυπικών μεθόδων* (formal methods). Εργαλεία απόδειξης ορθότητας προγραμμάτων που τρέχουν στο EVM, καθώς και η διατύπωση ολοκληρωμένων semantics για την εικονική μηχανή υπάρχουν στην βιβλιογραφία ([Park18], [Hild18]).

Κατά τον προγραμματισμό έξυπνων συμβολαίων όμως, όλες οι πλατφόρμες υποστηρίζουν γλώσσες πιο υψηλού επιπέδου, για χρήση από τον προγραμματιστή. Για παράδειγμα, στο Ethereum η πιο δημοφιλής γλώσσα προγραμματισμού συμβολαίων είναι η γλώσσα Solidity [soli19], επηρεασμένη κυρίως από την γλώσσα Javascript, και μεταγλωττίζεται σε κώδικα της εικονικής μηχανής EVM.

Η χρήση γλωσσών υψηλότερου επιπέδου για προγραμματισμό συμβολαίων, εκτός από την ευκολία που παρέχει, ανοίγει ένα μέτωπο ευπαθειών προς εκμετάλλευση από κακόβουλους παίκτες.

2.3 Συναρτησιακές γλώσσες έξυπνων συμβολαίων

Η ύπαρξη αυτών των ευπαθειών κάνει την επιλογή της γλώσσας προγραμματισμού συμβολαίων πολύ σημαντική. Οι συναρτησιακές γλώσσες έχουν συζητηθεί και ερευνηθεί αρκετά στην κοινότητα των γλωσσών προγραμματισμού και φημίζονται για την ασφάλειά και την χρησιμότητά τους για την κατασκευή μαθηματικά ορθών προγραμμάτων. Ένα πρόγραμμα γραμμένο σε μία συναρτησιακή γλώσσα επιδίδεται αρκετά πιο εύκολα σε τυπική ανάλυση και απόδειξη ιδιοτήτων σχετικά με την λειτουργία του, και μπορεί να εκφραστεί πιο εύκολα με μαθηματικό συμβολισμό από αντίστοιχα προγράμματα σε προστακτικές γλώσσες. Η ύπαρξη του συστήματος τύπων μπορεί να εντοπίσει και να εξαλείψει πολλές κατηγορίες λαθών, ήδη κατά την μεταγλώττιση. Η πλατφόρμα έξυπνων συμβολα Τα χαρακτηριστικά αυτά κάνουν τις συναρτησιακές γλώσσες δελεαστική επιλογή για τον προγραμματισμό έξυπνων συμβολαίων.

Όπως αναφέρθηκε προηγουμένως, η γλώσσα που προτείνει ο Szabo για την διατύπωση machine-readable συμβολαίων είναι άμεσα επηρεασμένη από το έργο των S.P.Jones et.al [Jone00] και [Jone03], όπου γίνεται χρήση μίας συναρτησιακής DSL (domain specific language) για την περιγραφή κλασσικών χρηματοοικονομικών προιόντων. Συγκεκριμένα, η γλώσσα που χρησιμοποιούν είναι ευέλικτη και συνθέσιμη, χρησιμοποιώντας απλά συστατικά στοιχεία τα οποία συνδυάζει για να κατασκευάσει πιο σύνθετα συμβόλαια.

Αρκετά συνηθισμένη επίσης είναι η χρήση τεχνικών τυπικής επαλήθευσης (formal verification) για την εξάλειψη λαθών κατά τον προγραμματισμό έξυπνων συμβολαίων. Όπως αναφέρθηκε παραπάνω, μπορούν να χρησιμοποιηθούν για την απόδειξη ορθότητας προγραμμάτων που στοχεύουν την εικονική μηχανή, ή για προγράμματα γλωσσών υψηλότερου επιπέδου. Οι τεχνικές τυπικής επαλήθευσης είναι παραδοσιακά ακριβές και δύσκολο να εφαρμοστούν σε μεγάλη κλίμακα σε πραγματικά έργα λογισμικού, για αυτό βρίσκουν εφαρμογή κυρίως σε έργα λογισμικού όπου τα λάθη στοιχίζουν ακριβά, όπως στον τομέα της αεροναυπηγικής και της κατασκευής μικροεπεξεργαστών.

Κοντά στην Plutus Core, το θεωρητικό μοντέλο της οποίας θα εξετάσουμε στην συνέχεια, είναι η γλώσσα Simplicity [OCon17], μια γλώσσα *συνδέσμων* (combinator-based), συνοδευόμενη με μία αφηρημένη μηχανή που ορίζει την λειτουργική σημασιολογία της. Όπως και η Plutus Core, η Simplicity έχει ελεγχθεί υπολογιστικά για την ορθότητά της με την βοήθεια προγραμμάτων αποδείξεων. Αντίθετα από την Plutus Core όμως, δεν είναι Turing-complete.

Επιπλέον, ενώ είναι αρκετά εύκολη η χρήση προχωρημένων τεχνικών συναρτησιακού προγραμματισμού για την συγγραφή προγραμμάτων σε Plutus Core καθώς βασίζεται απευθείας στον λάμδα λογισμό, και συγκεκριμένα στην System F_ω, δεν ισχύει το ίδιο για την Simplicity.

Μία ακόμα συναρτησιακή πλατφόρμα συμβολαίων, η Tezos, εισάγει την γλώσσα Michelson, ως έναν χαμηλού επιπέδου συνδυασμό της Forth με Lisp, υποστηρίζοντας ισχυρούς τύπους. Η σημασιολογία της Micheson έχει αποδειχθεί σωστή, και όπως και η πλατφόρμα Plutus, υποστηρίζει προγραμματισμό συμβολαίων από τον χρήστη σε γλώσσα υψηλότερου επιπέδου.

Οι παραπάνω γλώσσες καλύπτουν αποκλειστικά το on-chain κομμάτι των συμβολαίων, δηλαδή το χαμηλού επιπέδου bytecode που εν τέλει θα καταλήξει να κατοικεί στο blockchain. Στην αρχιτεκτονική Plutus, όπως θα συζητηθεί στο κεφάλαιο 2.3.1, είναι σχεδιασμένη ώστε να αντιμετωπίζει ομοιόμορφα τόσο τον on-chain, όσο και το κομμάτι του off-chain κώδικα, όπως το wallet και το ui με το οποίο έρχεται σε επαφή ο προγραμματιστής. Για να επιτύχει αυτόν τον σκοπό δεν επινοεί μια εντελώς καινούργια γλώσσα, αλλά βασίζεται αρκετά σε "γνωστές" και παλιές τεχνολογίες όπως το System F_ωκαι η Haskell, και μπορεί να αξιοποιήσει το μεγάλο σώμα γνώσεων γύρω από αυτές.

2.3.1 Plutus και Plutus Core

Η πλατφορμα έξυπνων συμβολαίων Plutus βασίζεται πάνω στο Cardano blockchain. Πρόκεται για ένα Proof-of-Stake blockchain πρωτόκολλο για επίτευξη κατανεμημένης συμφωνίας (distributed consensus) [Kiay17].

Η αρχιτεκτονική Plutus έχει δύο κύρια συστατικά. Το πρώτο είναι ένα GHC plugin που επιτρέπει στον προγραμματιστή να γράψει συμβόλαια σε Haskell. Στη συνέχεια, ο υψηλού επιπέδου κώδικας

Haskell μεταγλωττίζεται στη γλώσσα Plutus Core, μια χαμηλότερου επιπέδου γλώσσα, που προορίζεται να κατοικήσει στο blockchain.

γλώσσα που μπορεί να χρησιμοποιηθεί για τον προγραμματισμό έξυπνων συμβολαίων. Ο προγραμματιστής μπορεί να γράφει μαζί τον κώδικα που εκτελείται τοπικά (off-chain κώδικας) μαζί με τον κώδικα που ανεβαίνει στο blockchain (on-chain κώδικας) σε Haskell, με τον on-chain κώδικα να μεταγλωττίζεται στην ενδιάμεση αναπαράσταση FIR, η οποία στη συνέχεια μεταγλωττίζεται σε Plutus Core.

Ο κώδικας Plutus Core, όπως και ο GHC Core, είναι επεκτάσεις του System F_ω, του πολυμορφικού λ-λογισμού. Ο GHC Core είναι πιο πλούσια επέκταση και υποστηρίζει αμοιβαία αναδρομικά bindings, αλγεβρικούς τύπους δεδομένων, εκφράσεις case, μετατροπές τύπων (coercions) μεταξύ άλλων.

Αντίθετα, ο κώδικας Plutus Core παραμένει απλούστερος ως προς τα δομικά χαρακτηριστικά που υποστηρίζει, προσπαθώντας να μείνει κοντά στην μαθηματική μορφή του υποκείμενου λογισμού. Όλα αυτά τα επιπλέον στοιχεία και χαρακτηριστικά του GHC Core που προσθέτουν εκφραστικότητα μπορούν να μεταφραστούν στον πιο "μαθηματικό" λογισμό αν η γλώσσα παρέχει έναν απλό τρόπο έκφρασης της αναδρομής. Στα επόμενα κεφάλαια θα δειχθεί πως με χρήση των στοιχειωδών εργαλείων που παρέχει η Plutus Core μπορούμε να εκφράσουμε χαρακτηριστικά υψηλού επιπέδου.

Κύριο χαρακτηριστικό της πλατφόρμας Plutus είναι η παραγωγή του validator script που θα καθορίσει τι σημαίνει ορθή εκτέλεση του συμβολαίου και τις συνθήκες που πρέπει να επικρατούν για να ξοδέψει κάποιος τους πόρους του συμβολαίου. Η ορθότητα του validator script είναι κεντρική σε κάθε blockchain, καθώς μόλις ανέβει στο blockchain δεν μπορεί να τροποποιηθεί.

Μία γλώσσα για τέτοια χρήση οφείλει να είναι μικρή, συναρτησιακή, ώστε ο προσδιορισμός της σημασιολογίας και της ανάλυσης της να απλοποιείται. Το σύστημα του λ-λογισμού System F_ω αποτελεί καλή αφετηρία για τους παραπάνω σκοπούς, με μικρές αλλά ουσιαστικές τροποποιήσεις. Η γλώσσα δεν περιέχει απευθείας τύπους δεδομένων και εκφράσεις case. Από κατασκευή, η System F_ω περιέχει παραμετροποιημένους τύπους , όπως η λιστα, (List A), όπου ο τύπος List είναι μεγαλύτερης τάξης, συγκεκριμένα $* \to *$. Αρκετές ενδιάμεσες γλώσσες υποστηρίζουν απευθείας τύπους δεδομένων, με τίμημα πιο σύνθετη σημασιολογία. Οι τύποι δεδομένων υποστηρίζονται επομένως μέσω της κωδικοποίησης Scott, που θα συζητηθεί στο κεφάλαιο 3.

Η Plutus Core δεν προορίζεται για μεταγλώττιση σε κώδικα μηχανής, και λόγω του πεδίου χρήσης της, το μεγαλύτερο ποσοστό του χρόνου κατά την εκτέλεσή της αφιερώνεται σε κρυπτογραφικές λειτουργίες. Το γεγονός αυτό καθιστά το επιπλέον κόστος της κωδικοποίησης, σε αντίθεση με την απευθείας υποστήριξη τύπων δεδομένων, έναν αποδεκτό συμβιβασμό.

2.3.2 Marlowe

Η παραπάνω προσέγγιση στην σχεδίαση μίας γλώσσας *ειδικού-σκοπού* (Domain Specific Language) για προγραμματισμό συμβολαίων είναι η πρώτη ιστορικά συνάντηση του συναρτησιακού προγραμματισμού και του κόσμου των ηλεκτρονικών συμβολαίων. Οι ιδέες αυτές έχουν επηρεάσει αρκετά τον σχεδιασμό της γλώσσας Marlowe [Thom18], που έχει σχεδιαστεί με γνώμονα την χρήση της από κάποιον που είναι ειδικός στην συγγραφή συμβολαίων, αλλά όχι στον προγραμματισμό. Η χρήση μίας DSL για αυτόν τον σκοπό προσφέρει αρκετά πλεονεκτήματα στους συγγραφείς των συμβολαίων:

Η Marlowe είναι μία γλώσσα ειδικού σκοπού ενσωματωμένη στην γλώσσα Haskell. Προορίζεται για χρήση με την πλατφόρμα έξυπνων συμβολαίων Plutus που παρουσιάζεται αναλυτικότερα στο επόμενο κεφάλαιο.

Ο κώδικας Marlowe γίνεται συμβατός με την αρχιτεκτονική Plutus μέσω ενός ερμηνευτή, καθώς η συγκεκριμένη προσέγγιση προσφέρει διάφορα σχεδιαστικά πλεονεκτήματα όπως η δυνατότητα επαναχρησιμοποίησης της ίδιας υλοποίησης τόσο στον on-chain, όσο και στον off-chain κώδικα, και η στενή σχέση με την σημασιολογία της γλώσσας Marlowe.

Μπορούμε να είμαστε σίγουροι ότι κάποιοι τύποι "κακών" προγραμμάτων δεν μπορούν να εκφραστούν στην γλώσσα. Με αυτόν τον τρόπο εξαλείφονται παθολογικές συμπεριφορές που είναι δυνατό να εμφανιστούν σε γλώσσες γενικού σκοπού, όπως π.χ C++, Javascript.

Για τα προγράμματα που κατασκευάζονται στην γλώσσα είναι πιο εύκολο να επαληθευτεί ότι ικανοποιούν ορισμένες ιδιότητες. Για παράδειγμα, μπορούμε να είμαστε σίγουροι ότι το συμβόλαιο θα εκτελέσει κάθε πληρωμή που είναι προγραμματισμένο να εκτελέσει, ή ότι δεν θα βρεθεί ποτέ σε συγκεκριμένη ροή εκτέλεσης. Κατά αυτόν τον τρόπο αυξάνουμε το επίπεδο εμπιστοσύνης που έχουμε στο συμβόλαιο, και γινόμαστε πιο σίγουροι ότι εκτελεί όντως τις λειτουργίες που είναι προγραμματισμένο να εκτελέσει και τίποτα περισσότερο ή λιγότερο.

Καθώς η Marlowe είναι γλώσσα ειδικού σκοπού, γίνεται δυνατή η σχεδίαση εργαλείων για την προσομοίωση των προγραμμάτων. Συνυπολογίζοντας ότι πρόκειται για συναρτησιακή γλώσσα, γίνεται εύκολο να γραφτούν προσομοιωτές και ερμηνευτές της γλώσσας με στόχο την εξαντλητική εξέταση και προσομοίωση των έξυπνων συμβολαίων που γράφονται. Ο ερμηνευτής αυτός μετατρέπει τον κώδικα Marlowe σε Plutus Core, έτοιμο για εκτέλεση μόλις του δοθεί η κατάλληλη είσοδος.

Γίνεται επομένως δυνατόν μέσω της Marlowe να γραφτούν σύνθετα συμβόλαια, όπως συλλογικής χρηματοδότησης (crowdfunding), εγγύησης (escrow) ακόμα και χρηματοοικονομικών παραγώγων.

Κεφάλαιο 3

Αναδρομικοί Τύποι

Σε αυτό το κεφάλαιο θα παρουσιάσουμε την προαπαιτούμενη θεωρία αναδρομικών τύπων που χρειάζεται για να εκφράσουμε την System F^{μ}_{ω}. Ακολουθώντας το [Pier02], στις επόμενες παραγράφους θα δούμε τις διαφορετικές επιλογές που παρουσιάζονται στον κατασκευαστή μιας γλώσσας με ισχυρό σύστημα τύπων με αναδρομή.

Στην συνέχεια θα συζητηθεί η επιλογή του τελεστή σταθερού σημείου που θα χρησιμοποιηθεί για την υποστήριξη της αναδρομής στην γλώσσα και η κωδικοποίηση των τύπων δεδομένων που ακολουθούμε. Στο τέλος του κεφαλαίου βρίσκεται η σχετική βιβλιογραφία γύρω από τα θέματα που θα αναπτυχθούν.

3.1 Τελεστής σταθερού σημείου

Η προσθήκη αναδρομικών τύπων σε μία συναρτησιακή γλώσσα γίνεται με την χρήση του *τελεστή σταθερού σημείου* (fixpoint operator). Ο τελεστής σταθερού σημείου είναι μια συνάρτηση μεγαλύτερης τάξης (higher-order function), που βρίσκει υπολογιστικά το σταθερό σημείο συναρτήσεων, αν αυτό υπάρχει και είναι καλός ορισμένο. Η παράσταση $\text{fix}\, f$ αντιστοιχεί στο x για το οποίο ισχύει $x = fx$. Η εξίσωση που ικανοποιεί ο τελεστής σταθερού σημείου είναι η εξής:

$$\text{fix}\, f = f(\text{fix}\, f)$$

Ο τελεστής σταθερού σημείου fix βρίσκει εφαρμογή σε διάφορους κλάδους των μαθηματικών, κυρίως γύρω από τον τομέα της θεωρητικής επιστήμης υπολογιστών, του λ-λογισμού με ή χωρίς τύπους και στον συναρτησιακό προγραμματισμό. Ο τελεστής fix παίρνει ως όρισμα μια μη αναδρομική συνάρτηση και "δένει τον κόμπο" γύρω από τον εαυτό της ώστε να την μετατρέψει σε αναδρομική. Η χρήση του fix γίνεται εμφανής με το παρακάτω παράδειγμα:

$$\begin{aligned}
\texttt{fix}\ f &= f\ (\texttt{fix}\ f) \\
\textbf{List}\ a &= \texttt{fix}(\lambda r.\Lambda a.1 + a * r) \\
&= (\lambda r.\Lambda a.1 + a * r)(\texttt{fix}\ \lambda r.\Lambda a.1 + a * r) \\
&= \Lambda\alpha.1 + a * (\texttt{fix}\lambda r.\Lambda a.1 + a * r)) \\
&= \Lambda a.1 + a * \textbf{List}\ a \\
&= \cdots \\
&= \Lambda\alpha.1 + a * \cdots * a
\end{aligned}$$

Ως όρισμα στον τελεστή σταθερού σημείου δίνεται η συνάρτηση $(\lambda r.\Lambda 1 + a * r)$, και η μεταβλητή r αντιστοιχεί στην αναδρομική εμφάνιση ενός τύπου. Η εφαρμογή του fix στην συνάρτηση αυτή γεννάει τον τελικό αναδρομικό τύπο. Τα fixpoints με τα οποία ασχολούμαστε στην γλώσσα μας ανήκουν στην κατηγορία των *least fixpoint*, όπως και οι περισσότερες χρήσεις τελεστών σταθερών σημείων στην βιβλιογραφία.

3.2 Isorecursive και equirecursive τύποι

Αρχικά καλούμαστε να επιλέξουμε ποια εκδοχή των αναδρομικών τύπων θα υποστηρίξουμε. Οι *equirecursive* τύποι ταυτίζουν τον τύπο (fix f) με τον f(fix f) ενώ οι *isorecursive* τύποι τους θεωρούν ισομορφικούς. Ως μάρτυρες (witness) του ισομορφισμού χρησιμοποιούνται οι όροι `wrap` και `unwrap` που συνοδεύουν την μετατροπή από την μία μορφή στην άλλη. Η ακριβής σύνταξη και χρήση των όρων αυτών συζητάται στο επόμενο κεφάλαιο όπου παρουσιάζεται η σύνταξη και οι κανόνες της System F^{μ}_{ω} και FIR.

Το τίμημα είναι ότι ενώ οι equirecursive τύποι δεν προσθέτουν επιπλέον όρους στο επίπεδο της γλώσσας, έχουν πιο περίπλοκη θεωρητική ανάλυση. Συγκεκριμένα, ο έλεγχος τύπων στην System F^{μ}_{ω} με equirecursive types έχει δειχθεί να είναι undecidable στην γενική περίπτωση [Drey07], [Cai16]. Η χρήση isorecursive τύπων στην γλώσσα μας κάνει την συγγραφή προγραμμάτων πιο δύσκολη, αλλά δέχεται πιο εύκολη ανάλυση και οι κανόνες τύπων μπορούν να εκφραστούν αρκετά απλά. Η γλώσσα που θα ορίσουμε έχει κύρια χρήση ως ενδιάμεση γλώσσα επομένως δεν είναι κύριος στόχος η εύκολη σύνθεση από τον προγραμματιστή, γεγονός που δικαιολογεί την επιλογή των isorecursive τύπων.

Ως παράδειγμα, βλέπουμε παρακάτω την έκφραση του "κλασσικότερου" αναδρομικού τύπου, της λίστας, στην γλώσσα της θεωρία τύπων, ακολουθώντας τον συμβολισμό του [Wadl90]. Βλέπουμε επίσης πως λειτουργούν οι όροι `wrap` και `unwrap` που παρουσιάζονται αναλυτικότερα στο κεφάλαιο 4

$$\mathsf{List} = \mu\alpha.\tau = \mu\alpha.1 + \alpha * \mathsf{List}$$
$$\texttt{unwrap}(\mu\alpha.\tau) = \tau\{\mu\alpha.\tau/\alpha\} = 1 + \alpha * (\mu\alpha.\tau)$$
$$\texttt{wrap}(1 + \alpha * (\mu\alpha.\tau)) = \mu\alpha.\tau$$

3.3 Επιλογή του κατάλληλου τελεστή σταθερού σημείου

Πρέπει ακόμα να επιλεχθεί ο κατάλληλος τελεστής σταθερού σημείου που θα χρησιμοποιήσουμε. Μία κλασσική επιλογή είναι ο τελεστής fix που επιτρέπει να πάρουμε τα σταθερά σημεία συναρτήσεων στο επίπεδο των τύπων, για όλα τα kinds K (fix : $(K \Rightarrow K) \Rightarrow$). Στην γλώσσα μας αντιθέτως θα γίνει χρήση του τελεστή `ifix` ("indexed fix") που επιτρέπει σταθερά σημεία μόνο σε kinds $K \Rightarrow *$.

Το πλεονέκτημα του `ifix` είναι ότι διευκολύνει την διατύπωση κανόνων σύνθεσης τύπων, όπως φαίνεται και από το παρακάτω παράδειγμα:

$$\begin{array}{lll}
\texttt{wrap}_0\ f_0 & t : \text{fix}\ f_0 & \text{where } t : f_0\ (\text{fix}\ f_0) \\
\texttt{wrap}_1\ f_1\ a1 & t : \text{fix}\ f_1\ a1 & \text{where } t : f_1\ (\text{fix}\ f_1)\ a1 \\
\texttt{wrap}_2\ f_2\ a1\ a2 & t : \text{fix}\ f_2\ a1\ a2 & \text{where } t : f_2\ (\text{fix}\ f_2)\ a1\ a2 \\
\ldots & &
\end{array}$$

Με τη χρήση του κλασσικού τελεστή fix πρέπει να είμαστε σε θέση να υποστηρίξουμε όλα τα arities των συναρτήσεων τύπων που μπορούμε να εκφράσουμε, καθώς δεν υποστηρίζεται πολυμορφισμός στο επίπεδο των kinds. Οι όροι `wrap` και `unwrap` τελικά πρέπει να ανήκουν στο επίπεδο των όρων, δηλαδή με kind $*$.

Είναι δυνατόν να δοθούν κανόνες τύπων συμβατοί με τον τελεστή fix, όπως έχει δειχθεί στο [Drey05], κάνοντας χρήση της τεχνικής των *elimination contexts*. Η προσέγγιση αυτή λύνει το πρόβλημα που αναφέραμε, αλλά περιπλέκει αρκετά τους όρους καθώς χρειάζεται να συνοδεύονται από επιπλέον πληροφορία και κάνει ακόμα πιο απαιτητική την σύνθεση τύπων.

Επομένως η χρήση του διαφορετικού τελεστή σταθερού σημείου που αναφέρθηκε απλοποιεί τους κανόνες σύνθεσης τύπων, λειτουργεί ομοιόμορφα για όλες τις περιπτώσεις, δεχόμενος μόνο ένα όρισμα με kind k, και επιστρέφει έναν όρο.

Ο τελεστής `ifix` έχει την ίδια εκφραστική δύναμη με τον fix, παρά την επιφανειακά περιορισμένη μορφή του. Στην επόμενη παράγραφο θα δούμε την απόδειξη της ισοδυναμίας εκφραστικότητας.

3.3.1 Επάρκεια του `ifix`

Ο τελεστής `ifix` είναι αρκετά εκφραστικός για να μας δώσει σταθερά σημεία συναρτήσεων σε αυθαίρετα kinds k. Η διαίσθηση πίσω από την απόδειξη είναι ότι μπορούμε να μετασχηματίσουμε κάθε kind K στο $(K \Rightarrow *) \Rightarrow *$, που έχει την κατάλληλη μορφή για το `ifix`.

Ορισμός 1: *Έστω J, kinds. Τότε ονομάζουμε το J* συμπύκνωση *του K εάν υπάρχουν συναρτήσεις $\phi : J \Rightarrow K$ και $\psi : K \Rightarrow J$ τέτοιες ώστε $\psi \circ \phi = id$.*

Πρόταση 1: *Έστω J συρρίκνωση του K και fix_K ένας fixpoint τελεστής στο K. Τότε υπάρχει fixpoint τελεστής fix_J στο J.*

Απόδειξη 1: *Αρκεί να πάρουμε $\mathrm{fix}_J(f) = \psi(\mathrm{fix}_K(\phi \circ f \circ \psi))$.*

Πρόταση 2: *Πρόταση 2 Έστω K ένα kind της System F^{μ}_{ω}. Τότε υπάρχει μοναδική (πιθανώς κενή) ακολουθία από kinds $(K_0, \ldots, K_n)$ τέτοια ώστε $K = \overline{K} \Rightarrow *$.*

Απόδειξη 2: *Από επαγωγή στην κατασκευή των kinds.*

Πρόταση 3: *Κάθε kind K στην System F^{μ}_{ω}, είναι συρρίκνση $(K \Rightarrow *) \Rightarrow *$.*

Απόδειξη 3: *Έστω $K = \overline{K} \Rightarrow *$ (από **Πρόταση 2**), και αρκεί να πάρουμε:*

$$\phi : K \Rightarrow (K \Rightarrow *) \Rightarrow *$$
$$\phi = \lambda(x :: K).\lambda(f :: K \Rightarrow *).fx$$
$$\psi : ((K \Rightarrow *) \Rightarrow *) \Rightarrow K$$
$$\psi = \lambda(w :: (K \Rightarrow *) \Rightarrow *).\lambda(\overline{a :: K}).w(\lambda o :: K.o\ \overline{a})$$

Λήμμα 1: *Αν υπάρχει fixpoint τελεστής στο kind $(K \Rightarrow *) \Rightarrow *$, τότε υπάρχει και για κάθε K*

Τέλος, η απόδειξη ολοκληρώνεται αν αντικαταστήσουμε το K στον ορισμό του `ifix` με $K \Rightarrow *$ για να πάρουμε σταθερό σημείο στο kind $(K \Rightarrow *) \Rightarrow *$, που από το τελευταίο λήμμα αρκεί να μας δώσει fixpoint σε κάθε K.

Η παραπάνω απόδειξη στηρίζεται στην πρόταση 2, επομένως δεν αληθεύει για αυθαίρετα kinds. Η System F^{μ}_{ω} που θα εξετάσουμε υποστηρίζει μόνο kinds που κατασκευάζονται από τα $\Rightarrow$ και $*$, και είναι αληθής στην περίπτωσή μας.

Το γεγονός ότι οι συρρικνώσεις διατηρούν την ιδιότητα του σταθερού σημείου είναι γνωστό στον τομέα της αλγεβρικής τοπολογίας [Stee52]. Αυτός είναι και ο λόγος που στις διατυπώσεις των παραπάνω ορισμών και θεωρημάτων δεν γίνεται ιδιαίτερη αναφορά στην φύση των kinds, καθώς ισχύουν σε αρκετά γενικευμένο περιβάλλον στην θεωρία κατηγοριών, που εμπεριέχει προφανώς την δομή των kinds που εξετάζουμε.

Η έννοια της συρρίκνωσης στην μελέτη των τύπων δεδομένων είναι γνωστό εργαλείο στην θεωρητική επιστήμη υπολογιστών, για παράδειγμα [Stir13], αλλά δεν έχουμε βρει εκδοχή της πρότασης 1 στην βιβλιογραφία.

Η παραπάνω απόδειξη παρουσιάστηκε στο πλαίσιο της εργασίας μας από τον Chad Nester.

3.4 Scott κωδικοποίηση των τύπων δεδομένων

Η κωδικοποίηση Scott ταυτίζει τον τύπο ενός datatype ως τον τύπο της συνάρτησης που κάνει pattern match σε αυτόν. Για παράδειγμα για τον τύπο των Booleans έχουμε:

$$\forall R.R \to R \to R$$

Μια συνάρτηση που κάνει ταίριασμα (pattern matching) σε μία συνάρτηση τύπου Bool, πρέπει για κάθε τύπο αποτελέσματος R, να μπορεί να δώσει ένα R στην περίπτωση που η τιμή είναι `True` και ένα R στην περίπτωση του `False`. Στην γενική περίπτωση όπου οι κατασκευαστές του τύπου δεδομένων δέχονται παραμέτρους, πρέπει να ο τύπος να μπορεί να επιστρέψει R, με είσοδο τα ορίσματα του κατασκευαστή.

Ο τύπος των φυσικών, Nat, γίνεται:

$$\forall R.R \to (\mathsf{Nat} \to R) \to R$$

Παρατηρούμε την εμφάνιση του Nat στον ορισμό, που χρησιμοποιείται από τον αναδρομικό κατασκευαστή στην περίπτωση του `Suc`. Για να αποκτήσει νόημα ο τελικός τύπος θα πρέπει να παντρευτεί με την αναδρομή στο επίπεδο των τύπων που υποστηρίζει η γλώσσα μας, με τη χρήση του fixpoint τελεστή.

Η κωδικοποίηση Church του τύπου Bool , και κάθε μη αναδρομικού τύπου ταυτίζεται με την Scott, αλλά είναι διαφέρουν στην περίπτωση αναδρομικών τύπων. Η Church κωδικοποίηση των Nat είναι:

$$\forall R.R \to (R \to R) \to R$$

Εδώ η αναδρομική εμφάνιση του Nat έχει απαλειφθεί και αντικατασταθεί με R. Σε αντίθεση με την κωδικοποίηση Scott που αντιστοιχεί στο ταίριασμα προτύπου πάνω στον τύπο, η κωδικοποίηση Church δίνει πρόσβαση στον πλήρη αναδρομικό τύπο.

Συνοπτικά οι διαφορές των δύο κωδικοποιήσεων είναι οι εξής:

- Για να επεξεργαστούμε μια τιμή κωδικοποιημένη κατά Church, πρέπει να "σκανάρουμε" (fold), ολόκληρη την δομή, που οδηγεί σε θέματα απόδοσης. Για μια τιμή κωδικοποιημένη κατά Scott, αρκεί να κοιτάξουμε το τελευταίο επίπεδο. Το θέμα αυτό ονομάζεται successor problem (από την δομή των φυσικών αριθών στην Church κωδικοποίηση) και αναλύεται στο [Koop14].

- Στην κωδικοποίηση Church η αναδρομική εμφάνιση είναι ήδη "τυλιγμένη" στην αναδρομή, επομένως δεν χρειαζόμαστε επιπλέον εργαλεία από την θεωρία αναδρομικών τύπων για να την χρησιμοποιήσουμε. Αντιθέτως στην κωδικοποίηση Scott, χρειαζόμαστε κατάλληλη αναδρομή στο επίπεδο των τύπων ώστε να ερμηνεύσουμε κατάλληλα τον Scott τύπο.

3.5 Σχετική βιβλιογραφία

Όπως αναφέρθηκε στην προηγούμενη παράγραφο, διαφορετικές προσεγγίσεις για την κωδικοποίηση τύπων δεδομένων έχουν συζητηθεί στο [Koop14], μαζί με μία τυπική περιγραφή της κωδικοποίησης Scott. Στην εργασία αυτή η κωδικοποίηση παρουσιάζεται πιο αναλυτικά, μαζί με πλήρες χειρισμό των αναδρομικών τύπων.

Στο [Yaku09] γίνεται λόγος για την χρήση *σταθερών σημείων με δείκτη* (indexed fixpoints), στην ανάπτυξη τεχνικών generic programming. Η παραπάνω δουλειά επεκτείνεται στο [Loh11] ώστε να υποστηρίζουν παραμετροποιημένους τύπους.

Μια άλλη υλοποίηση της System F^{μ}_{ω}με ισοαναδρομικούς τύπους παρουσιάζεται στο [Brow17]. Περιλαμβάνει και έναν τελεστή που πραγματοποιεί το pattern matching στους τύπους, εξυπηρετώντας αντίστοιχο ρόλο με την match function στον ορισμό ενός τύπου δεδομένων στην FIR, όπως θα δούμε στη συνέχεια. Χρησιμοποιούν επίσης παρόμοιο τελεστή σταθερού σημείου, μόνο για την περίπτωση που ο δείκτης έχει kind $*$, ενώ ο τελεστής `ifix` που παρουσιάσαμε παραπάνω λειτουργεί για κάθε kind k.

Κεφάλαιο 4

Η γλώσσα FIR

Στο κεφάλαιο αυτό θα ορίσουμε την γραμματική και τους κανόνες σύνθεσης και ισοδυναμίας τύπων της γλώσσας την γλώσσα FIR ως μια προέκταση του συστήματος System F^{μ}_{ω}. Ο λογισμός System F^{μ}_{ω} αποτελείται από το System F_{ω}, τον συνδυασμό των αξόνων του λάμδα κύβου (lambda cube, [Bare91]) που αντιστοιχούν στον πολυμορφισμό και στις συναρτήσεις τύπων. Τα τελευταία αποτελούν αρκετά χρήσιμα στοιχεία σε μια γλώσσα που υποστηρίζει χαρακτηριστικά υψηλότερης τάξης. Συγκεκριμένα, η ενδιάμεση αναπαράσταση που χρησιμοποιεί ο compiler GHC της γλώσσας Haskell, εν ονόματι GHC Core, είναι συνδυασμός της System F_{ω}μαζί με πρόσθετους κανόνες για την αποδοτική κωδικοποίηση αλγεβρικών τύπων δεδομένων.

Στην System F^{μ}_{ω} οι αλγεβρικοί τύποι δεδομένων που συναντάμε στις δημοφιλείς γλώσσες προγραμματισμού μπορούν να κωδικοποιηθούν εμμέσως, χωρίς να χρειάζεται να προσθέσουμε επιπλέον χαρακτηριστικά. Ο τρόπος κωδικοποίησης των τύπων δεδομένων όμως είναι αρκετά δυσνόητος, γεγονός που δυσχεραίνει τον χρήστη της γλώσσας. Αυτό δεν αποτελεί πρόβλημα για την System F^{μ}_{ω}, που προορίζεται για ένα σύστημα ”χαμηλού” επιπέδου, το οποίο δεν θα διαβάζεται ή γράφεται από τον χρήστη, αλλά θα παράγεται αυτόματα κατά την μεταγλώττιση από τον κώδικα Haskell.

Όπως αναφέρθηκε και στην παράγραφο 2.3.1, ο κώδικας Haskell που γράφει ο προγραμματιστής των συμβολαίων μεταγλωττίζεται από ένα GHC plugin σε FIR και στη συνέχεια σε Plutus Core. Η γλώσσα Plutus Core είναι άμεση επέκταση της System F^{μ}_{ω}. Η χρήση μιας απλής γλώσσας σαν typed bytecode κάνει πιο εύκολη την χρήση τυπικών μεθόδων ανάλυσης και επαλήθευσης.

Καθώς το βήμα της μεταγλώττισης από την Haskell στην System F^{μ}_{ω} είναι μεγάλο, είναι εύκολο να υπάρξει λάθος κατά την μεταγλώττιση. Επίσης η System F^{μ}_{ω} κάνει κάποιες βελτιστοποιήσεις, όπως την εξάλειψη των αδρανών `let` αρκετά πιο δύσκολη, σε σύγκριση με μία γλώσσα που υποστηρίζει let-bindings. Στο κεφάλαιο 6 γίνεται αναφορά σε δύο τέτοιες βελτιστοποιήσεις.

Στο κεφάλαιο 4.2 θα παρουσιαστούν οι κανόνες σύνταξης και τύπων της System F^{μ}_{ω}. Στη συνέχεια, στο κεφάλαιο 4.3 προσθέτουμε αναδρομικά let-bindings στην System F^{μ}_{ω}που υποστηρίζουν την δήλωση αμοιβαία αναδρομικών όρων και τύπων.

4.1 System F_{ω}

Η γλώσσα System F_{ω} αποτελεί επέκταση του λ-λογισμού με απλούς τύπους. Αν θέλουμε να ορίσουμε την θέση της στον λαμβδα κύβο, αποτελεί την προέκταση του λ-λογισμού με απλούς τύπους με δύο χαρακτηριστικά, *τελεστές τύπων* (*type operators*) και πολυμορφισμό.

Η System F_{ω} ως έχει είναι ισχυρά κανονικοποιήσιμη (*strongly normalizing*), δηλαδή κάθε ακολουθία αποτιμήσεων $t_1 \rightarrow \ldots \rightarrow$ τερματίζει. Από το γεγονός αυτό προκύπτει ότι ο έλεγχος και η ανακατασκευή τύπων για την System F_{ω} είναι decidable, αναμενόμενο, καθώς δεν έχουμε προσθέσει αναδρομή στην γλώσσα. Η αποτίμηση στο επίπεδο των τύπων πραγματοποιείται κατά το typechecking της γλώσσας

4.2 Ορισμός της System F^{μ}_{ω} και της FIR

Η FIR αποτελεί μια προέκταση της System F^{μ}_{ω}, που με τη σειρά της είναι επέκταση της System F_{ω}. Στα σχήματα που ακολουθούν βλέπουμε τους κανόνες σύνταξης (σχήμα 4.1), σύνθεσης και ισοδυναμίας τύπων και kinds(σχήματα 4.5, 4.4 , 4.6 αντίστοιχα), καθώς και ορθής κατασκευής των bindings και των κατασκευαστών που υποστηρίζει η FIR (σχήμα 4.3) . Οι μη-υπογραμμισμένες περιπτώσεις ανήκουν στην System F_{ω}, και οι προσθήκες στις System F^{μ}_{ω} και FIR αντίστοιχα.

Σε όλα τα παρακάτω, χρησιμοποιούμε τον συμβολισμό $\bar{t}$ για να δηλώσουμε την ακολουθία $t_1, \cdots, t_n$, καθώς και τις βοηθητικές συναρτήσεις στο σχήμα 4.2. Οι συναρτήσεις αυτές θα φανούν χρήσιμες στην έκφραση των κανόνων τύπων και ακόμα περισσότερο στην μεταγλώττιση των binding τύπων δεδομένων από την FIR σε καθαρή System F^{μ}_{ω} στο επόμενο κεφάλαιο.

Στις παρακάτω παραγράφους θα σχολιαστούν λεπτά σημεία που αφορούν τον ορισμό της System F^{μ}_{ω} και FIR που αξίζουν ιδιαίτερη αναφορά.

4.2.1 Γραμματική της System F^{μ}_{ω} και FIR

Στο σχήμα 4.1 φαίνεται η χρήση των όρων `wrap` και `unwrap` που αποτελούν τους μάρτυρες του ισομορφισμού των αναδρομικών τύπων που υποστηρίζονται. Παρατηρούμε πως το `wrap` της γλώσσας μας είναι πλήρως saturated, μια επιλογή που απλοποιεί την χρήση του καθώς και τους κανόνες τύπων που δίνονται.

Το πρώτο όρισμα του `wrap` αντιστοιχεί στην δομή που ονομάζουμε *pattern functor*, και περιγράφει την δομή του αναδρομικού τύπου, που ουσιαστικά πρόκειται για συνάρτηση στο επίπεδο των τύπων. Το δεύτερο όρισμα του `wrap` αποθηκεύει τις παραμέτρους του αναδρομικού τύπου δεδομένων που περιγράφει ο pattern functor. Τέλος το τρίτο όρισμα περιέχει τον όρο που θέλουμε να τυλίξουμε.

Στην περίπτωση του `unwrap` δεν χρειαζόμαστε ως επιπλέον όρισμα τον αναδρομικό τύπο, ή τις παραμέτρους του, καθώς η πληροφορία για το πως πρέπει να ξετυλιχθεί ο όρος περιέχεται σε αυτόν, και συγκεκριμένα στον τύπο του. Από την μορφή των κανόνων τύπων συμπεραίνουμε πως κάθε όρος που μπορεί να χρησιμοποιηθεί σε έναν `unwrap` έχει ήδη "τυλιχτεί" προηγουμένως με χρήση του `wrap`.

Για αυτό λοιπόν και στις τιμές της System F^{μ}_{ω} προστίθενται και οι όροι `wrap`, που μπορούν να υπάρξουν ως τιμές αν παραμετροποιηθούν από τιμές-όρους. Οι όροι `unwrap` αντιθέτως δεν μπορούν να είναι values καθώς πάντα μπορούμε να εφαρμόσουμε `wrap` σε έναν όρο `unwrap`.

Ο τύπος του όρου είναι αυτός της συνάρτησης στο επίπεδο των τύπων, ακολουθώντας το παράδειγμα στην παράγραφο 3.2, όπου εκφράσαμε τον αναδρομικό τύπο List σε θεωρητικό πλαίσιο χρησιμοποιώντας συναρτήσεις τύπων. Ο συμβολισμός προσεγγίζει αρκετά αυτόν της θεωρίας, με μόνη διαφορά την χρήση του λ, αντί για μ που έχουμε στην θεωρία αναδρομικών τύπων.

4.2.2 Κανόνες και ισοδυναμία τύπων

Η System F^{μ}_{ω} περιέχει το σύστημα του λ-λογισμού με απλούς τύπους στο επίπεδο των τύπων, καθώς υποστηρίζει δημιουργία συνάρτησης (abstraction) και εφαρμογή (application), μεταξύ τύπων με το σωστό kind, βλέπε σχήμα 4.6. Επομένως δημιουργείται υπολογισμός στο επίπεδο των τύπων και χρειάζεται να γίνει αποτίμηση των τύπων στην τελική τους μορφή. Στο σχήμα 4.4 παρουσιάζονται οι κανόνες για το πότε δύο τύποι είναι ισοδύναμοι, προκύπτουν δηλαδή από την διαδικασία της αποτίμησης.

Η διαδικασία αυτή είναι ασφαλής καθώς η γλώσσα System F^{μ}_{ω} είναι ισχυρά κανονικοποιήσιμη, όπως αναφέρθηκε και στην παράγραφο 4.1. Ο υπολογισμός στο επίπεδο των τύπων τερματίζει πάντα και μας δίνει έγκυρο τύπο.

Σε μία γλώσσα με αυτό το χαρακτηριστικό που προορίζεται για χρήση στο blockchain, είναι σημαντικό οι τύποι να κανονικοποιούνται πριν αποθηκευτούν ως on-chain κώδικας, για μείωση χώρου αλλά και του κόστος εκτέλεσης (gas) κατά την εκτέλεση των προγραμμάτων.

όροι	t, u	::=	x	variable
			$\lambda x : T.t$	lambda abstraction
			$t\ t$	function application
			$\Lambda X :: K.t$	type abstraction
			$t\ \{T\}$	type application
			`wrap` $T\ U\ t$	wrap
			`unwrap` t	unwrap
			`let [rec]` $\overline{b}$ `in` t	let
bindings	b	::=	$x : T = t$	term binding
			$X :: K = T$	type binding
			`data` $X\ (\overline{Y :: K}) = \overline{c}$ `with` x	datatype binding
κατασκευαστές	c	::=	$x\ (\overline{T})$	
τιμές	v	::=	$\lambda x : T.t$	lambda abstraction
			$\Lambda X :: K.t$	type abstraction
			`wrap` $T\ U\ v$	wrap
τύποι	T, U	::=	X	type variable
			$T \to U$	arrow type
			$\forall X :: K.T$	universal type
			$\lambda X :: K.T$	function type
			$T\ U$	function application
			`ifix` $T\ U$	fixpoint type
πλαίσια	Γ	::=	$\varnothing$	empty
			$\Gamma, x : T$	term variable binding
			$\Gamma, X :: K$	type variable binding
kinds	K	::=	$*$	type kind
			$K \Rightarrow K$	arrow kind

Σχήμα 4.1: Σύνταξη και γραμματική της FIR

4.2.3 Τύποι και όροι παραμετροποιήσιμοι από τύπους

Ως επέκταση της γλώσσας System F_ω, υποστηρίζονται όροι παραμετροποιήσιμοι από τύπους. (*type abstraction* Έτσι εκφράζεται ο πολυμορφισμός στην γλώσσα, οι πολυμορφικοί όροι παραμετροποιούνται από μία μεταβλητή τύπου, που δίνεται σαν όρισμα όταν καλείται η συνάρτηση. Με παρόμοιο τρόπο χειρίζεται ο πολυμορφισμός και στον GHC, τον compiler της Haskell. Οι συναρτήσεις στο επίπεδο των όρων που παραμετροποιούνται από τύπους δημιουργούνται με την χρήση του Λ. Στο κεφάλαιο 4 θα γίνει έντονη χρήση των type abstractions κατά την μεταγλώττιση των τύπων δεδομένων από την FIR σε System F^μ_ω.

Επομένως όταν βλέπουμε κατασκευές της γλώσσας που περιέχουν Λ ξέρουμε ότι ανήκουν στο επίπεδο των **όρων**. Ο τύπος του όρου $\Lambda X :: K.t$ είναι ο universally quantified ($\forall X :: K.T$), όπως φαίνεται και από τον κανόνα T-TAbs του σχήματος 4.5

Στην γραμματική της System F^μ_ω βλέπουμε και την ύπαρξη τύπων που παραμετροποιούνται από τύπους ($\lambda X :: K.T$) Τα type-level lambdas αυτά"κατοικούν" στο επίπεδο των τύπων, και είναι εμφανές από το περιβάλλον που χρησιμοποιούνται σε ποια αναφερόμαστε. Η αποτίμηση αυτών των συναρτήσεων περιλαμβάνεται στο σχήμα 4.4, στους κανόνες Q-Abs και Q-Beta.

$d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x$
$c = x(\overline{T})$

Χρήσιμες συναρτήσεις

$$
\begin{array}{lcl}
\mathsf{branchTy}(c, R) & = & \overline{T} \to R \\
\mathsf{dataTy}(d) & = & \lambda(\overline{Y :: K}).\forall R.(\overline{\mathsf{branchTy}(c, R)}) \to R \\
\mathsf{dataKind}(d) & = & \overline{K} \Rightarrow * \\
\mathsf{constrTy}(d, c) & = & \forall(\overline{Y :: K}).\overline{T} \to X\ \overline{Y} \\
\mathsf{matchTy}(d) & = & \forall(\overline{Y :: K}).(X\ \overline{Y}) \to (\mathsf{dataTy}(d)\ \overline{Y})
\end{array}
$$

Binder functions

$$
\begin{array}{lcl}
\mathsf{dataBind}(d) & = & X :: \mathsf{dataKind}(d) \\
\mathsf{constrBind}(d, c) & = & c : \mathsf{constrTy}(c, X\ \overline{Y}) \\
\mathsf{constrBinds}(d) & = & \overline{\mathsf{constrBind}(d, c)} \\
\mathsf{matchBind}(d) & = & x : \mathsf{matchTy}(d) \\
\mathsf{binds}(x : T = t) & = & x : T \\
\mathsf{binds}(X : K = T) & = & X : K \\
\mathsf{binds}(d) & = & \mathsf{dataBind}(d), \mathsf{constrBinds}(d), \mathsf{matchBind}(d) \\
 & = & x : \mathsf{matchTy}(d)
\end{array}
$$

Σχήμα 4.2: Βοηθητικοί ορισμοί

$$\text{W-Con}\ \frac{c = x(\overline{T}) \qquad \overline{\Gamma \vdash T :: *}}{\Gamma \vdash_{\mathsf{ok}} c}$$

$$\text{W-Term}\ \frac{\Gamma \vdash T :: * \qquad \Gamma \vdash t : T}{\Gamma \vdash_{\mathsf{ok}} x : T = t} \qquad \text{W-Type}\ \frac{\Gamma \vdash T :: K}{\Gamma \vdash_{\mathsf{ok}} X : K = T}$$

$$\text{W-Data}\ \frac{\begin{array}{c} d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x \\ \Gamma' = \Gamma, \overline{Y :: K} \qquad \overline{\Gamma' \vdash_{\mathsf{ok}} c} \end{array}}{\Gamma \vdash_{\mathsf{ok}} d}$$

Σχήμα 4.3: Ορθή κατασκευή των δηλώσεων `let`

$$\text{Q-Refl}\ \frac{}{T \equiv T} \qquad \text{Q-Symm}\ \frac{T \equiv S}{S \equiv T}$$

$$\text{Q-Trans}\ \frac{S \equiv U \qquad U \equiv T}{S \equiv T} \qquad \text{Q-Arrow}\ \frac{S_1 \equiv S_2 \qquad T_1 \equiv T_2}{(S_1 \to T_1) \equiv (S_2 \to T_2)}$$

$$\text{Q-All}\ \frac{S \equiv T}{(\forall X :: K.S) \equiv (\forall X :: K.T)} \qquad \text{Q-Abs}\ \frac{S \equiv T}{(\lambda X :: K.S) \equiv (\lambda X :: K.T)}$$

$$\text{Q-App}\ \frac{S_1 \equiv S_2 \qquad T_1 \equiv T_2}{S_1 T_1 \equiv S_2 T_2} \qquad \text{Q-Beta}\ \frac{}{(\lambda X :: K.T_1) T_2 \equiv [X \mapsto T_2] T_1}$$

Σχήμα 4.4: Ισοδυναμία τύπων της FIR

4.2.4 Kinding της System F^{μ}_{ω}

Η μόνη προσθήκη της System F^{μ}_{ω} και FIR στα kinds της System F_{ω} είναι ο κανόνας για το `ifix`. Ο τελεστής σταθερού σημείου είναι πλήρως εφαρμοσμένος, δέχεται τον pattern functor και τις παρα-

$$\text{T-Var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-Abs}\ \frac{\Gamma, x : T_1 \vdash t : T_2 \quad \Gamma \vdash T_1 :: *}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2}$$

$$\text{T-App}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash (t_1\ t_2) : T_2} \qquad \text{T-TAbs}\ \frac{\Gamma, X :: K \vdash t : T}{\Gamma \vdash (\Lambda X :: K.t) : (\forall X :: K.T)}$$

$$\text{T-TApp}\ \frac{\Gamma \vdash t_1 : \forall X :: K_2.T_1 \quad \Gamma \vdash T_2 :: K_2}{\Gamma \vdash (t_1\ \{T_2\}) : [X \mapsto T_2]T_1} \qquad \text{T-Eq}\ \frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T}$$

$$\text{T-Wrap}\ \frac{\Gamma \vdash M : (F\ (\lambda(X :: K).\,\texttt{ifix}\,F\ X))\ T \quad \Gamma \vdash T :: K \quad \Gamma \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *)}{\Gamma \vdash \texttt{wrap}\ F\ T\ M : \texttt{ifix}\,F\ T}$$

$$\text{T-Unwrap}\ \frac{\Gamma \vdash M : \texttt{ifix}\,F\ T \quad \Gamma \vdash T :: K}{\Gamma \vdash \texttt{unwrap}\,M : (F\ (\lambda(X :: K).\,\texttt{ifix}\,F\ X))\ T}$$

$$\text{T-Let}\ \frac{\Gamma \vdash_{\mathsf{ok}} \overline{b} \quad \Gamma \vdash T :: * \quad \Gamma, \overline{\mathsf{binds}(b)} \vdash t : T}{\Gamma \vdash (\texttt{let}\ \overline{b}\ \texttt{in}\ t) : T}$$

$$\text{T-LetRec}\ \frac{\Gamma, \overline{\mathsf{binds}(b)} \vdash_{\mathsf{ok}} \overline{b} \quad \Gamma \vdash T :: * \quad \Gamma, \overline{\mathsf{binds}(b)} \vdash t : T}{\Gamma \vdash (\texttt{let rec}\ \overline{b}\ \texttt{in}\ t) : T}$$

Σχήμα 4.5: Σύνθεση τύπων της FIR

μέτρους του, "δένει την θηλιά" αναδρομικά και επιστρέφει έναν όρο με αυτόν τον τύπο, για αυτό και το αποτέλεσμα έχει kind $*$. Οι υπόλοιποι κανόνες είναι αναμενόμενοι και αντίστοιχοι του λ-λογισμού με απλούς τύπους, υποστηρίζοντας εφαρμογή, δημιουργία συνάρτησης και καθολικούς τύπους.

4.3 Όροι Let

Στο σχήμα 4.2 υπάρχουν οι βοηθητικές συναρτήσεις που θα χρησιμοποιήσουμε κατά την μελέτη των datatypes και για την μεταγλώττιση τους από FIR σε System F^{μ}_{ω}, στο επόμενο κεφάλαιο.

Το επιπλέον στοιχείο που προσθέτει η FIR είναι οι όροι `let` με τους οποίους μπορούμε να ορίσουμε bindings όρων και τύπων δεδομένων, για χρήση στο σώμα του `let`. Οι κανόνες που υπαγορεύουν την σωστή κατασκευή των όρων `let` φαίνονται στο σχήμα 4.3. Οι κανόνες τύπου που δίνουμε για τους όρους αυτούς (σχήμα 4.4, κανόνες T-Let και T-LetRec)

Διαισθητικά, υποθέτουμε ότι οι datatypes έχουν τους τύπους που δόθηκαν, και πραγματοποιούμε τον έλεγχο τύπων με αυτούς.

4.3.1 Datatypes

Η FIR υποστηρίζει δηλώσεις *τύπων δεδομένων*. Ένας ορισμός τύπου δεδομένων στην FIR περιλαμβάνει αρχικά μία δήλωση τύπου, μαζί με τις παραμέτρους τους, όπου οι δηλώσεις να συνοδεύονται με τα kinds τους. Στο δεξί μέλος έχουμε τους *κατασκευαστές* (constructors) και το όνομα της συνάρτησης που καταστρέφει τον τύπο δεδομένων, *συνάρτηση ταιριάσματος*. Η σύνταξη των δηλώσεων τύπων δεδομένων μοιάζει επομένως αρκετά με το πως μπορεί ο προγραμματιστής να δηλώσει τύπους δεδομένων στην Haskell.

Η συνάρτηση ταιριάσματος είναι ο τρόπος που χρησιμοποιούμε τους τύπους δεδομένων στην γλώσσα μας, και πραγματοποιεί το ταίριασμα προτύπου (pattern matching). Άλλη εναλλακτική θα ήταν να εισάγουμε έναν `typecase` τελεστή, όπως έχει γίνει σε άλλες υλοποιήσεις της System F^{μ}_{ω} που ξέρουμε από την βιβλιογραφία [Cai16].

Παρακάτω βλέπουμε την δήλωση του κλασσικού τύπου δεδομένων Maybe, που "τυλίγει" μια τιμή οποιουδήποτε τύπου, που μπορεί να είναι κενή. Οι κατασκευαστές του είναι οι Nothing, χωρίς ορίσματα, και Just με ένα όρισμα, και το όνομα της συνάρτησης ταιριάσματος είναι `matchMaybe`.

$$\texttt{data Maybe } (A :: *) = (\mathsf{Nothing}(), \mathsf{Just}(A)) \texttt{ with matchMaybe}$$

Ο τύπος της συνάρτησης `matchMaybe` είναι Maybe $A \to \forall R.R \to (A \to R) \to R$ και υλοποιεί το ταίριασμα σε τιμές του τύπου Maybe , όπως είδαμε στην παράγραφο 3.4 όπου γίνεται λόγος για την κωδικοποίηση Scott. Η συνάρτηση ταιριάσματος μετατρέπει τον αφηρημένο τύπο δεδομένων (abstract data type) στον κωδικοποιημένο κατά Scott τύπο που μπορεί να κάνει χρήση των τιμών του. Η ακριβής μορφή της συνάρτησης φαίνεται στην παράγραφο 5.2 και ο τύπος της δίνεται από την βοηθητική συνάρτηση matchTy(Maybe) του σχήματος 5.1

Τέλος, η υποστήριξη αναδρομικών δηλώσεων τύπων στην FIR, κάνει τα `ifix`, `wrap` και `unwrap` περιττά από πλευρά εκφραστικότητας. Για πρακτικούς λόγους όμως είναι προτιμότερο να παραμείνουν στην γλώσσα, παρά την επικάλυψη τους, καθώς έτσι η FIR αποτελεί γνήσιο υπερσύνολο της System F^{μ}_{ω} που απλοποιεί την παρουσίαση της.

$$\text{K-TVar}\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \qquad \text{K-Abs}\frac{\Gamma, X :: K_1 \vdash T :: K_2}{\Gamma \vdash (\lambda X :: K_1.T) :: K_1 \Rightarrow K_2}$$

$$\text{K-App}\frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash (T_1\ T_2) :: K_2} \qquad \text{K-Arrow}\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash (T_1 \to T_2) :: *}$$

$$\text{K-All}\frac{\Gamma, X :: K \vdash T :: *}{\Gamma \vdash (\forall X :: K.T) :: *} \qquad \text{K-Ifix}\frac{\Gamma \vdash T :: K \quad \Gamma \vdash F :: (K \Rightarrow *) \Rightarrow (K \Rightarrow *)}{\Gamma \vdash (\texttt{ifix } F\ T) :: *}$$

Σχήμα 4.6: Kinding της FIR

4.3.2 Ορθή κατασκευή των bindings & και κατασκευαστών

Ο έλεγχος τύπων απαιτεί οι όροι let να έχουν την σωστή μορφή, η οποία περιγράφεται στο σχήμα 4.3. Ουσιαστικά αυτό σημαίνει ότι θέλουμε οι κατασκευαστές των τύπων να παίρνουν ορίσματα μεταβλητές με ground types, δηλαδή ορίσματα με τύπο $*$ (κανόνας W-Con). Στη συνέχεια ο κανόνας W-Data ελέγχει ότι μία δήλωση ενός datatype είναι ορθή, όταν οι κατασκευαστές έχουν τον τύπο με τον οποίο έχουν δηλωθεί, στο περιβάλλον που περιέχει τα ονόματα και τα ορίσματα του τύπου δεδομένων που ορίζεται.

4.3.3 Αμοιβαία αναδρομικά let στο επίπεδο των όρων

Η FIR υποστηρίζει και την δήλωση αμοιβαία αναδρομικών όρων. Σε αυτή την διπλωματική δεν θα ασχοληθούμε με την μεταγλώττιση (αμοιβαία) αναδρομικών όρων, αλλά αποτελεί σημαντικό χαρακτηριστικό μίας γλώσσας, όχι μόνο σε πρακτικό επίπεδο, αλλά και σε θεωρητικό, καθώς η υποστήριξη αμοιβαία αναδρομικών όρων σε μοντέλο πρόθυμης αποτίμησης δεν χει εξερευνηθεί από την βιβλιογραφία. Για τον χειρισμό των αναδρομικών όρων Η αντιμετώπιση των αναδρομικών όρων αποτελεί σημαντικό μέρος της ευρύτερης εργασίας, της οποίας μέρος είναι η παρούσα διπλωματική.

Κεφάλαιο 5

Μεταγλώττιση τύπων δεδομένων

Στο κεφάλαιο αυτό θα δούμε πως μπορούμε να μεταγλωττίσουμε τις υψηλού-επιπέδου δηλώσεις τύπων δεδομένων που υποστηρίζει η FIR στην System F^{μ}_{ω}, εφαρμόζοντας δηλαδή desugaring σε μία γλώσσα που έχει μελετηθεί θεωρητικά και έχουν αποδειχτεί θεωρήματα ορθότητας για αυτή. Δεν θα αναφερθούμε στην μεταγλώττιση στο επίπεδο των όρων, αλλά ούτε και στην μετάφραση αναδρομικών `let` τύπων, καθώς η περίπτωση των τύπων δεδομένων είναι όχι μόνο πιο πλούσια, αλλά και υπερσύνολο των αναδρομικών δηλώσεων τύπων.

Κατά την μετάφραση από FIR σε System F^{μ}_{ω}, θα φανεί χρήσιμο το ότι η πρώτη είναι υπερσύνολο της δεύτερης, επομένως μπορούμε να "απαλείψουμε" κάθε στοιχείο της FIR σταδιακά, μέχρι να φτάσουμε σε καθαρή System F^{μ}_{ω}. Για παράδειγμα στην μεταγλώττιση των αναδρομικών datatypes θα χρησιμοποιήσουμε την μεταγλώττιση των μη-αναδρομικών `let`.

Καθώς θα ασχοληθούμε με τα bindings τύπων, και συγκεκριμένα τύπων δεδομένων δεν εξετάζουμε "ετερογενή" `let` που εμπεριέχουν τόσο δηλώσεις όρων όσο και τύπων. Από τους κανόνες γραμματικής της FIR βλέπουμε ότι τέτοια δήλωση είναι δυνατή. Το γεγονός αυτό δεν αποτελεί εμπόδιο στην περίπτωση μη αναδρομικών δεσιμάτων, καθώς οι νέοι ορισμοί δεν μπορούν να βρεθούν στο δεξιό μέλος μίας δήλωσης.

Βέβαια δεν μπορούμε να απαιτήσουμε ο χειρισμός των δεσιμάτων να είναι τόσο απρόσκοπτος και στην αναδρομική περίπτωση, και για αυτόν τον λόγο θα θεωρήσουμε *ομογενή* `let`, που εμπεριέχουν δηλώσεις μόνο μίας κατηγορίας, στην παρούσα εργασία μόνο τύπων δεδομένων.

Επιπροσθέτως, δεν βρίσκεται στο άμεσο ενδιαφέρον μας η υποστήριξη αναδρομικών τύπων σε `let` δηλώσεις, άρα οι τύποι δεν εξαρτώνται από datatypes ή αντίστροφα. Η έλλειψη εξαρτημένων τύπων σημαίνει ότι οι τύποι δεν εξαρτώνται από όρους άρα μπορούμε με ασφάλεια να χωρίσουμε τις κατηγορίες των `let`-bindings.

5.1 Μη-αναδρομικά `let`

Η "μεταγλώττιση" των μη-αναδρομικών δεσιμάτων `let` είναι τετριμμένη, και αποτελεί στην ουσία desugaring του `let` σε εφαρμογή συναρτήσεων. Μεταφράζονται σε δημιουργία συνάρτησης, ακολουθούμενη από άμεση εφαρμογή της στο δεξί μέλος του δεσίματος. Η μεταβλητή (τύπου ή όρου) που δένεται στο `let` δεν εμφανίζεται στο δεξί μέλος της ισότητας b, και άρα ο b δεν περιέχει ελεύθερη εμφάνιση του x, καθιστώντας την αφαίρεση και εφαρμογή ασφαλείς.

$$\begin{aligned} \mathbb{C}_{\text{term}}(\texttt{let}\ x : t = b\ \texttt{in}\ v) &= (\lambda(x : t).v)\ b \\ \mathbb{C}_{\text{type}}(\texttt{let}\ t :: k = b\ \texttt{in}\ v) &= (\Lambda(t :: k).v)\ \{b\} \end{aligned}$$

5.2 Μη-αναδρομικοί τύποι δεδομένων

Η μεταγλώττιση των μη-αναδρομικών τύπων δεδομένων είναι αρκετά απλή. Η στρατηγική μεταγλώττισης είναι συγκεντρωμένη στο σχήμα 5.1. Χρησιμοποιεί τις βοηθητικές συναρτήσεις του προηγούμενου κεφαλαίου (4.2) και ακολουθεί την κωδικοποίηση Scott (3.4). Στη συνέχεια της παραγράφου, ως παράδειγμα της μεθόδου θα μετατραπέι η FIR δήλωση του Maybe σε καθαρή System F^{μ}_{ω}.

$$d := \texttt{data Maybe } A = (\mathsf{Nothing}(), \mathsf{Just}(A)) \texttt{ with match}$$

- $\mathsf{branchTy}(c, R)$ υπολογίζει τον τύπο του constructor του συγκεκριμένου branch, δηλαδή τον τύπο που παίρνει τα ορίσματα του αντίστοιχου branch, και επιστρέφει έναν τύπο εξόδου R.

$$\mathsf{branchTy}(\mathsf{Nothing}(), R) = R$$
$$\mathsf{branchTy}(\mathsf{Just}\ A, R) = A \to R$$

- $\mathsf{dataKind}(d)$ υπολογίζει το kind του datatype. Πρόκειται για το kind arrow μεταξύ όλων των παραμέτρων που καταλήγει στο $*$.

$$\mathsf{dataKind}(\mathsf{Maybe}) = * \Rightarrow *$$

- $\mathsf{dataTy}(d)$ υπολογίζει τον Scott τύπο του datatype. Δεσμεύει τα ονόματα των παραμέτρων και υλοποιεί το ταίριασμα προτύπων χρησιμοποιώντας τους τύπους των branch.

$$\mathsf{dataTy}(d) = \lambda A.\forall R.R \to (A \to R) \to R$$

- $\mathsf{constrTy}(c, T)$ υπολογίζει τον τύπο ενός constructor d.

$$\mathsf{constrTy}(\mathsf{Nothing}(), \mathsf{Maybe}) = \forall A.\ \mathsf{Maybe}\ A$$
$$\mathsf{constrTy}(\mathsf{Just}\ A, \mathsf{Maybe}) = \forall A.A \to \mathsf{Maybe}\ A$$

- $\mathsf{constr}(d, c)$ υπολογίζει τον constructor στο επίπεδο των όρων. Ο τύπος R χρησιμοποιείται στη θέση του αφηρημένου τύπου δεδομένων, αφού έχουν "δεσμευτεί" μέσω type abstraction τα ορίσματά του. Στη συνέχεια κατασκευάζει την συνάρτηση ταιριάσματος που παίρνει όλες τις επιλογές και εφαρμόζει το i-οστό branch στα κατάλληλα στα ορίσματα.

$$\mathsf{constr}(d, \mathsf{Nothing}()) = \Lambda A.\Lambda R.\lambda(b_1 : R)(b_2 : A \to R).b_1$$
$$\mathsf{constr}(d, \mathsf{Just}(A)) = \Lambda A.\lambda(v : A).\Lambda R.\lambda(b_1 : R)(b_2 : A \to R).b_2\ v$$
$$\mathsf{constr}(d, \mathsf{Nothing}()) = \Lambda A.\Lambda R.\lambda(b_1 : R)(b_2 : A \to R).b_1$$

- $\mathsf{matchTy}(d)$ είναι ο τύπος της συνάρτησης ταιριάσματος

$$\mathsf{matchTy}(d) = \forall A.\ \mathsf{Maybe}\ A \to (\forall R.R \to (A \to R) \to R)$$

- $\mathsf{match}(d)$ δίνει τον ορισμό της συνάρτησης ταιριάσματος, που ουσιαστικά είναι η ταυτοτική συνάρτηση στον Scott τύπο.

$$\mathsf{match}(d) = \Lambda A.\lambda(v : \mathsf{Maybe}\ A).v$$

- $\mathsf{unveil}(d, t)$ αντικαθιστά τον αφηρημένο τύπο δεδομένων μέσα σε έναν όρο, με τον "ωμό" Scott τύπο.

$$\mathsf{unveil}(d, t) = [\mathsf{Maybe} \mapsto \lambda A.\forall R.R \to (A \to R) \to R]t$$

Η διαίσθηση πίσω από την μεταγλώττιση είναι απλή, με χρήση της αφαίρεσης τύπων σε όρους (Λ-συναρτήσεις)δεσμεύουμε τα ονόματα του datatype, των constructors και της συνάρτησης ταιριάσματος.

Οι αφηρημένοι τύποι δεδομένων μπορούν να εκφραστούν με την χρήση *υπαρξιακών* τύπων (existential types [Pier02]). Οι υπαρξιακοί τύποι δεν αναφέρονται στην παρούσα διπλωματική, αλλά αποτελούν

$d = \texttt{data}\, X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x$
$c = x(\overline{T})$

Βοηθητικές συναρτήσεις

$\mathsf{constr}(d, c) = \Lambda(\overline{Y :: K}).\lambda(\overline{a : T}).\Lambda R.\lambda(\overline{b : \mathsf{branchTy}(c, R)})\ b_i\ \overline{a}$
$\quad \textbf{where}\ c = c_i$
$\mathsf{constrs}(d) \quad = \overline{\mathsf{constr}(d, c)}$
$\mathsf{match}(d) \quad = \Lambda(\overline{Y :: K}).\lambda(x : (\mathsf{dataTy}(d)\ \overline{Y}).x$
$\mathsf{unveil}(d, t) \ = [X \mapsto \mathsf{dataTy}(d)]t$

Συνάρτηση μεταγλώττισης

$\mathbb{C}_{\mathsf{data}}(\texttt{let}\ d\ \texttt{in}\ t)$
$\quad = (\Lambda(\mathsf{dataBind}(d)).\lambda(\mathsf{constrBinds}(d)).\lambda(\mathsf{matchBind}(d)).t)$
$\quad \{\mathsf{dataTy}(d)\}$
$\quad \overline{\mathsf{unveil}(d, \mathsf{constrs}(d))}$
$\quad \mathsf{match}(d)$

Σχήμα 5.1: Μεταγλώττιση μη αναδρομικών τύπων δεδομένων

το δυικό ανάλογο των καθολικών ποσοδεικτών. Κάθε υπαρξιακός τύπος μπορεί επομένως να εκφραστεί με την χρήση των καθολικών ποσοδεικτώνν, και η χρήση του $\mathsf{unveil}(d, t)$ στους κατασκευαστές του αφηρημένου τύπου εξυπηρετεί αυτήν την λειτουργία.

Όπως είδαμε και στην αρχή του κεφαλαίου, ενώ κατά την μετάφραση των όρων `let` δημιουργούμε μία αφαίρεση στο επίπεδο τύπων ή όρων που εφαρμόζεται άμεσα, εδώ εναλλάσσουμε το abstraction με την εφαρμογή. Αυτό γίνεται γιατί ο τύπος δεδομένος πρέπει να παραμείνει abstract στους κατασκευαστές, ώστε να πάρουν τον κατάλληλο τύπο. Για αυτό χρειάζεται να αντικαταστήσουμε τον αφηρημένο τύπο, με την concrete μορφή του, ενέργεια που επιτελεί η συνάρτηση $\mathsf{unveil}(d, t)$. Για αυτό και η χρήση ενός ακόμα abstraction/let δεν είναι σωστή, καθώς θα δημιουργούσαν ακόμα έναν αφηρημένο τύπο.

Παρακάτω βλέπουμε συγκεντρωμένη την μεταγλώττιση του Maybe με χρήση των παραπάνω συναρτήσεων.

$\mathbb{C}_{\mathsf{data}}(\texttt{let data}\ \mathsf{Maybe}\ A = (\mathsf{Nothing}(), \mathsf{Just}(A))\ \texttt{with}\ \mathsf{match}$
$\quad \texttt{in}\ \mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\{\mathsf{Int}\}1)\ 0\ (\lambda x : \mathsf{Int}.x + 1))$

$= (\Lambda(\mathsf{Maybe} :: * \Rightarrow *).$	(τύπους του Maybe)
$\lambda(\mathsf{Nothing} : \forall A.\ \mathsf{Maybe}\ A).$	(τύπος του Nothing)
$\lambda(\mathsf{Just} : \forall A.A \to \mathsf{Maybe}\ A).$	(τύπους του Just)
$\lambda(\mathsf{match} : \forall A.\ \mathsf{Maybe}\ A \to \forall R.R \to (A \to R) \to R).$	(τύπους του match)
$\mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\{\mathsf{Int}\}1)\ 0\ (\lambda x : \mathsf{Int}.x + 1))$	(body of the let)
$(\lambda A.\forall R.R \to (A \to R) \to R)$	(ορισμός του Maybe)
$(\Lambda A.\Lambda R.\lambda(b_1 : R)\ (b_2 : A \to R).b_1)$	(ορισμός του Nothing)
$(\Lambda A.\lambda(v_1 : A).\Lambda R.\lambda(b_1 : R)\ (b_2 : A \to R).b_2\ v_1)$	(ορισμός του Just)
$(\Lambda A.\lambda(v : \forall R.R \to (A \to R) \to R).v)$	(ορισμός του match)

Από τα παραπάνω παρατηρούμε:

- Ο κατασκευαστής Just πρέπει να δώσει τον αφηρημένο τύπο στο σώμα του $tlet$, αλλιώς η εφαμοργή του match δεν θα συνοδεύεται από τον σωστό τύπο.

- Ο Scott τύπους παράγεται στον ορισμό Just.

- Η συνάρτηση ταιριάσματος match μετατρέπει τον αφηρημένο τύπο στην "ωμή " Scott εκδοχή του. Υπολογιστικά είναι απλά η ταυτοτική συνάρτηση.

5.3 Μεταγλώττιση αναδρομικών τύπων δεδομένων

Στην παράγραφο αυτή θα εξεταστεί η μεταγλώτιση των αναδρομικών τύπων δεδομένων από την FIR σε System F^{μ}_{ω}. Το πέρασμα αυτό θα χρησιμοποιήσει την δυνατότητα μεταγλώτισης των μη-αναδρομικών τύπων δεδομένων, όπως το είδαμε στο προηγούμενο κεφάλαιο. Η μεταγλώτιση των αναδρομικών τύπων δεδομένων μαζί με βοηθητικές συναρτήσεις για ευκολότερη παρουσίαση που είδαμε και στην περίπτωση των μη-αναδρομικών datatypes (4.2).

Η υποστήριξη απλά αναδρομικών τύπων είναι αρκετά απλή. Αρκεί να εκφράσουμε τον αναδρομικό τύπο ως type-level συνάρτηση, συγκεκριμένα ως pattern functor (4.2.1), αντικαθιστώντας κάθε αναδρομική εμφάνιση με ένα όρισμα που χρησιμοποιείται αναδρομικά στην συνάρτηση. Στη συνέχεια εφαρμόζωντας τον τελεστή σταθερού σημείου παράγουμε τον τελικό datatype.

Η ύπαρξη γινομένων στο επίπεδο των kinds θα έκανε τετριμμένη την κωδικοποίηση. Στο κεφάλαιο 4 όμως, είδαμε την γραμματική και τους κανόνες της System F^{μ}_{ω} και FIR, που δεν υποστηρίζουν τέτοιου τύπου γινόμενα.

Επομένως θα χρειαστεί να *κωδικοποιήσουμε* τα γινόμενα στο επίπεδο των τύπων με τις τεχνικές που έχουμε στην διάθεσή μας. Σε ένα περιβάλλον με ισχυρούς τύπους, το γινόμενο n kinds μπορεί να εκφραστεί ως συνάρτηση από έναν δείκτη σε μία τιμή. Ο δείκτης αναφέρεται στην θέση του στοιχείου του γινομένου. Αντί για φυσικό αριθμό, θα χρησιμοποιήσουμε μία διαφορετική ετικέτα που δέχεται παραμέτρους τύπων, ένα χαρακτηριστικό που έλειπε από την αρχική δουλειά [Yaku09] και συζητήθηκε μόνο με χρήση εξαρτημένων τύπων στο [Loh11]. Ο "δείκτης" που χρησιμοποιείται.

Η μεταγλώττιση της αναδρομικής περίπτωσης στην FIR είναι αρκετά κοντά με την μη-αναδρομική που παρουσιάστηκε στην προηγούμενη παράγραφο. Η ειδοποιός διαφορά είναι ότι "δένουμε" τα ονόματα των constructors, των συναρτήσεων ταιριάσματος, των ονομάτων των τύπων και των ορισμάτων μαζί, και κατασκευάζουμε τον τύπο της οικογένειας κάνοντας χρήση του πλήρους αναδρομικού τύπου, που γνωρίζει για όλα τα datatypes-μέλη της οικογένειας.

Θα γίνει χρήση των συναρτήσεων από το σχήμα fig. 5.1, δίνοντας τις αναγκαίες παραλλαγές για την αναδρομική περίπτωση όπου αυτό χρειάζεται.

Στην παράγραφο που ακολουθεί θα παρουσιαστεί βήμα-βήμα η μεταγλώττιση των αμοιβαία αναδρομικών ορισμών των τύπων δεδομένων Tree και Forest.

$$d_1 := \texttt{data Tree } A = (\mathsf{Node}(A, \mathsf{Forest}\, A))\ \texttt{with matchTree}$$
$$d_2 := \texttt{data Forest } A = (\mathrm{Nil}(), \mathrm{Cons}(\mathsf{Tree}\, A, \mathsf{Forest}\, A))\ \texttt{with matchForest}$$

- $\mathsf{tagKind}(l)$ εκφράζει το kind των datatypes της οικογένειας l.. Η διάκριση μεταξύ των περιπτώσεων των μελών της οικογένειας γίνεται μέσω της δομής του tag. Η χρήση tags για κωδικοποίηση τύπων δεδομένων είναι γνωστή από την βιβλιγραφία γύρω από τον *γενικευμένο προγραμματισμό* (generic programming). Τα tags που θα χρησιμοποιήσουμε μιμούνται την κωδικοποίηση Scott των ν-άδων (tuples). Η $\mathsf{tagKind}(l)$ συγκεντρώνει τα kinds των διαφορετικών περιπτώσεων.
$$\mathsf{tagKind}(l) = (* \Rightarrow *) \Rightarrow (* \Rightarrow *) \Rightarrow *$$

- $\mathsf{tag}(l, d)$ δίνει τον ορισμό του τύπου του tag για κάθε τύπο δεδομένων d της οικογένειας. Τα tags ζούν στο επίπεδο των τύπων, και δέχονται σαν ορίσματα τις παραμέτρους των datatypes καθώς και συναρτήσεις που αντιστοιχούν στην κάθε περίπτωση, και το εκάστοτε tag εφαρμόζει τις παραμέτρους στην σωστή περίπτωση. Έτσι υποστηρίζονται οι παραμετροποιημένοι datatypes, σημείο που υστερούσε από την βιβλιογραφία και έχει εξερευνηθεί κυρίως στο [Loh11]. Εδώ

$l = \texttt{let rec}\ \overline{d}\ \texttt{in}\ t$
$d = \texttt{data}\ X\ (\overline{Y :: K}) = (\overline{c})\ \texttt{with}\ x$
$c = x(\overline{T})$

Χρήσιμες συναρτήσεις

$$
\begin{array}{ll}
\mathsf{tagKind}(l) & = \overline{\mathsf{dataKind}(d)} \Rightarrow * \\
\mathsf{tag}(l,d) & = \lambda(\overline{Y :: K}).\lambda(\overline{X :: \mathsf{dataKind}(d)}).X_i\ \overline{Y} \\
 & \quad \textbf{where}\ d = d_i \\
\mathsf{inst}(f,l,d) & = \lambda(\overline{Y :: K}).f\ (\mathsf{tag}(l,d)\ \overline{Y}) \\
\mathsf{family}(l) & = \lambda(r :: \overline{\mathsf{dataKind}(d)} \Rightarrow *)\ .\lambda(t :: \mathsf{tagKind}(l)).\ \texttt{let}\ \overline{X = \mathsf{inst}(r,l,d)}\ \texttt{in}\ t\ \overline{\mathsf{dataTy}(d)} \\
\mathsf{instFamily}(l,d) & = \lambda(\overline{Y :: K}).\ \texttt{ifix}\ \mathsf{family}(l)\ (\mathsf{tag}(l,d)\ \overline{Y}) \\
\mathsf{constr}^{\mathtt{rec}}(l,d,c) & = \Lambda(\overline{Y :: K}).\lambda(\overline{a : T}).\ \mathtt{wrap}\ \mathsf{family}(l)\ (\mathsf{tag}(l,d)\ \overline{Y})\ (\Lambda R.\lambda(\overline{b : \mathsf{branchTy}(c,R)}).\ b_k\ \overline{a}) \\
 & \quad \textbf{where}\ d = d_i, c = c_k \\
\mathsf{constrs}^{\mathtt{rec}}(l,d) & = \overline{\mathsf{constr}^{\mathtt{rec}}(l,d,c)} \\
\mathsf{match}^{\mathtt{rec}}(l,d) & = \Lambda(\overline{Y :: K}).\lambda(x : \mathsf{instFamily}(l,d)\ \overline{Y}).\ \mathtt{unwrap}\ x \\
\mathsf{unveil}^{\mathtt{rec}}(l,t) & = [X_1 \mapsto \mathsf{instFamily}(l,1)] \ldots [X_n \mapsto \mathsf{instFamily}(l,d_n)]t
\end{array}
$$

Συνάρτηση μεγαγλώττισης

$$
\begin{array}{ll}
\mathbb{C}_{\mathrm{datarec}}(l) & = (\Lambda(\overline{\mathsf{dataBind}(d)}).\lambda(\overline{\mathsf{constrBinds}(d)}).\lambda(\overline{\mathsf{matchBind}(d)}).t) \\
 & \quad \overline{\{\mathsf{instFamily}(l,d)\}} \\
 & \quad \overline{\mathsf{unveil}^{\mathtt{rec}}(l,\mathsf{constrs}^{\mathtt{rec}}(l,d))} \\
 & \quad \overline{\mathsf{match}^{\mathtt{rec}}(l,d)}
\end{array}
$$

Σχήμα 5.2: Μεταγλώττιση αναδρομικών τύπων δεδομένων

αντιθέτως με την προηγούμενη δουλειά δεν χρησιμοποιούμε περιβάλλον με *εξαρτημένους τύπους* (dependent types).

$$
\begin{aligned}
\mathsf{tag}(l, \mathsf{Tree}) &= \lambda A.\lambda(v_1 :: * \Rightarrow *)(v_2 :: * \Rightarrow *).v_1\ A \\
\mathsf{tag}(l, \mathsf{Forest}) &= \lambda A.\lambda(v_1 :: * \Rightarrow *)(v_2 :: * \Rightarrow *).v_2\ A
\end{aligned}
$$

- Η βοηθητική συνάρτηση $\mathsf{inst}(f,l,d)$ εφαρμόζει τον τύπο f στο tag του datatype d της οικογένειας. Ο τύπος f για παράδειγμα θα μπορούσε να αναφέρεται στον τύπο *TreeForest*, της αμοιβαία αναδρομικής οικογένειας, με τους τύπους-μέλη Tree και Forest. Η εφαρμογή εδώ αντιστοιχεί στο ταίριασμα προτύπων, η έκφραση της μορφής $\mathsf{inst}(f,l,\mathsf{Tree})$ αναφέρεται στην περίπτωση f του τύπου Tree.

$$
\begin{aligned}
\mathsf{inst}(f,l,\mathsf{Tree}) &= \lambda A.f\ (\mathsf{tag}(\overline{d},\mathsf{Tree})\ A) \\
\mathsf{inst}(f,l,\mathsf{Forest}) &= \lambda A.f\ (\mathsf{tag}(\overline{d},\mathsf{Forest})\ A)
\end{aligned}
$$

- Η συνάρτηση $\mathsf{family}(l)$ ορίζει τον τύπο της αμοιβαία αναδρομικής οικογένειας. Πρόκειται για μία συνάρτηση που δέχεται δύο ορίσματα, το αναδρομικό που θα χρησιμοποιηθεί από τον τελεστή `ifix` για να "δέσει τον κόμπο" της αναδρομής, και το tag που δηλώνει την περίπτωση του τύπου. Στη συνέχεια εφαρμόζει στο tag τους κατά Scott κωδικοποιημένους τύπους της οικογένειας, οι οποίοι δηλώνονται στο `let` και έχουν με την σειρά τους αρχικοποιηθεί από το αναδρομικό όρισμα.

$$
\begin{aligned}
\mathsf{family}(l) = \lambda r\ t.\ &\texttt{let} \\
&\mathsf{Tree} = \mathsf{inst}(r, l, \mathsf{Tree}) \\
&\mathsf{Forest} = \mathsf{inst}(r, l, \mathsf{Forest}) \\
&\texttt{in}\ t\ \mathsf{dataTy}(d_1)\ \mathsf{dataTy}(d_2) \\
\mathsf{dataTy}(d_1) = \lambda A.\forall R.(A \to \mathsf{Forest}\ A \to R) \to R \\
\mathsf{dataTy}(d_2) = \lambda A.\forall R.R \to (\mathsf{Tree}\ A \to \mathsf{Forest}\ A \to R) \to R
\end{aligned}
$$

- Στην $\mathsf{instFamily}(l, d)$ χρησιμοποιείται για την αρχικοποίηση της οικογένειας ο πλήρης αναδρομικός τύπος, δηλαδή η εφαρμογή του `ifix` στον $\mathsf{family}(l)$ που ορίστηκε προηγουμένως.

$$\mathsf{instFamily}(l, \mathsf{Tree}) = \lambda A.\ \texttt{ifix}\ \mathsf{family}(l)\ (\mathsf{tag}(l, \mathsf{Tree})\ A)$$

- Η συνάρτηση $\mathsf{constr}^{\mathtt{rec}}(l, d, c)$ κατασκευάζει τον constructor c του τύπου δεδομένων d της οικογένειας. Έχουμε την ίδια συμπεριφορά με την μη-αναδρομική περίπτωση, εδώ όμως γίνται χρήση του `wrap` που πακετάρει τα ορίσματα και τον τύπο αποτελέσματος και επιστρέφει το κατάλληλο κλαδί (branch).

$$
\begin{aligned}
\mathsf{constr}^{\mathtt{rec}}(l, \mathsf{Tree}, \mathsf{Node}) = \Lambda A.\lambda(v_1 : A)(v_2 : \mathsf{Forest}\ A). \\
&\texttt{wrap}\ \mathsf{instFamily}(l, \mathsf{Tree})\ A \\
&(\Lambda R.\lambda(b_1 : A \to \mathsf{Forest}\ A \to R).b_1\ v_1\ v_2) \\
\mathsf{constr}^{\mathtt{rec}}(l, \mathsf{Forest}, \mathrm{Nil}) = \Lambda A. \\
&\texttt{wrap}\ \mathsf{instFamily}(l, \mathsf{Forest})\ A \\
&(\Lambda R.\lambda(b_1 : R)(b_2 : \mathsf{Tree}\ A \to \mathsf{Forest}\ A \to R).b_1) \\
\mathsf{constr}^{\mathtt{rec}}(l, \mathsf{Forest}, \mathrm{Cons}) = \Lambda A.\lambda(v_1 : \mathsf{Tree}\ A)(v_2 : \mathsf{Forest}\ A). \\
&\texttt{wrap}\ \mathsf{instFamily}(l, \mathsf{Forest})\ A \\
&(\Lambda R.\lambda(b_1 : R)(b_2 : \mathsf{Tree}\ A \to \mathsf{Forest}\ A \to R).b_2\ v_1\ v_2)
\end{aligned}
$$

- Η $\mathsf{match}^{\mathtt{rec}}(l, d)$ μας δίνει τον τύπο της συνάρτησης ταιριάσματος για τον datatype d. Παρόμοια με την μη αναδρομική περίπτωση, η συνάρτηση ταιριάσματος είναι η ταυτοτική συνάρτηση στον αφηρημένο τύπο δεδομένων (Algebraic Datatype). Το ίδιο συμβαίνει και στην αναδρομική περίπτωση, με την προσθήκη του `unwrap` που "αναλύει" τον ADT.

$$
\begin{aligned}
\mathsf{match}^{\mathtt{rec}}(l, \mathsf{Tree}) = \Lambda A.\lambda(v : \mathsf{Tree}\ A).\ \texttt{unwrap}\ v \\
\mathsf{match}^{\mathtt{rec}}(l, \mathsf{Forest}) = \Lambda A.\lambda(v : \mathsf{Forest}\ A).\ \texttt{unwrap}\ v
\end{aligned}
$$

- $\mathsf{unveil}^{\mathtt{rec}}(l, t)$ "ξετυλίγει" τους datatypes όπως πριν αντικαθιστώντας τις εμφανίσεις των δεσμεύσεων των ονομάτων τους με τον κατά Scott κωδικοποιημένο τύπο.

Κεφάλαιο 6

Συμπεράσματα & Μελλοντικές Κατευθύνσεις

6.1 Συνεισφορά

Στην παραπάνω εργασία ορίσαμε μια μικρή αλλά και ταυτόχρονα εκφραστική γλώσσα την οποία μεταγλωττίσαμε με σταδιακά περάσματα σε γνωστά και μελετημένα συστήματα του λ-λογισμού. Η γλώσσα αυτή, FIR προσθέτει let-bindings αμοιβαία αναδρομικές δηλώσεις όρων και τύπων δεδομένων στην System F^{μ}_{ω}, που αποτελεί το σύστημα του λ-λογισμού που χρησιμοποιούμε. Οι αναδρομικοί τύποι υποστηρίζονται μέσω μιας παραλλαγής του fixpoint τελεστή και οι τύποι δεδομένων εκφράζονται μέσω της κωδικοποίησης Scott.

Παρόλο που τα επιμέρους συστατικά έχουν αναφερθεί στο παρελθόν στην βιβλιογραφία, δεν έχουν συνδυαστεί κατά αυτό τον τρόπο, και αποτελούν μια άσκηση στην σχεδίαση μιας συναρτησιακής γλώσσας που μπορεί να λειτουργήσει σαν ενδιάμεση αναπαράσταση, με κύριο χαρακτηριστικό της την μεταγλώττιση αναδρομικών χαρακτηριστικών από υψηλού επιπέδου δηλώσεις, σε εφαρμογή του fixpoint τελεστή, ομοιόμορφα τόσο στο επίπεδο των όρων, όσο και στο επίπεδο των τύπων.

Η δουλειά που υποβλήθηκε στο συνέδριο Mathematics of Program Construction 2019 αποτελεί υπερσύνολο της παρούσας διπλωματικής. Συνοπτικά εξετάστηκαν:

- μεταγλώττιση αμοιβαία αναδρομικών τύπων σε τύπους της System F^{μ}_{ω}.
- μεταγλώττιση αμοιβαία αναδρομικών όρων σε όρους της System F^{μ}_{ω}.
- υλοποίηση του μεταγλωττιστή από FIR σε System F^{μ}_{ω} στην γλώσσα Agda, που υποστηρίζει εξαρτημένους τύπους.

Η υλοποίηση του μεταγλωττιστή σε Agda είναι εγγενώς συνοδευόμενη από τύπους, και οι όροι συνοδεύονται με την προέλευση των τύπων τους ([Alte99]). Το γεγονός αυτό αποδεικνύει ότι η μεταγλώττιση διατηρεί τα *kindx* και τους τύπους.

6.1.1 Μειονεκτήματα

Η κωδικοποίηση που χρησιμοποιήθηκε προσθέτει αισθητό φόρτο, εφαρμογή και χρήση συναρτήσεων, ειδικά σε σύγκριση με μία υλοποίηση αναδρομικών τύπων όπου τα δεδομένων συνοδεύονται με επιπλέον πεδία που δείχνουν στους αναδρομικούς ορισμούς που χρησιμοποιούν. Στις γλώσσες που εξετάσαμε, οι τύποι δεδομένων είναι καθαρές συναρτήσεις στο επίπεδο των τύπων, η γλώσσα δεν περιέχει πρόσθετα χαρακτηριστικά για την υποστήριξη τους.

Σε μία γλώσσα που στοχεύει σε ανταγωνιστική απόδοση, συγκρίσιμη με αυτή των mainstream γλωσσών προγραμματισμού, ο επιπλέον φόρτος που προστίθεται με την κωδικοποίηση αυτή είναι απαγορευτικός. Στη περίπτωση όμως που η ορθότητα μας ενδιαφέρει περισσότερο, είναι ωφέλιμο να έχουμε μια μικρή, αυτοτελή και πολύ εκφραστική γλώσσα-πυρήνα (core language) που είναι ενδελεχώς μελετημένη.

6.1.2 Βελτιστοποιήσεις

Έχουμε αναφέρει ότι η δυνατότητα υποστήριξης `let`-δεσιμάτων "ξεκλειδώνει" και απλοποιεί βελτιστοποιήσεις του μεταγλωττιστή. Θα αναφέρουμε δύο τέτοιες βελτιστοποιήσεις.

6.1.3 Εξάλειψη αδρανών let-binding

Σε μία γλώσσα με όρους `let` είναι αρκετά εύκολη η εφαρμογή της βελτιστοποίησης που απαλείφει τα αδρανή let. Συγκεκριμένα το πέρασμα της βελτιστοποίησης εντοπίζει τους όρους/τύπους/datatypes που έχουν δεσμευτεί στην κορυφή του `let`, αλλά δεν χρησιμοποιείται στο σώμα του. Τέτοια bindings μπορούν να απαλειφθούν.

Αντιθέτως στην System F^{μ}_{ω}, οι όροι `let` μεταφράζονται σε δημιουργία και εφαρμογή συναρτήσεων, περιπλέκοντας την απλή "προστακτική" φύση των όρων που κάνουν την εξάλειψη των αδρανών δεσιμάτων τόσο φυσική στην FIR. Η εφαρμογή της ίδιας βελτιστοποίησης στην System F^{μ}_{ω} θα απαιτούσε πιο λεπτομερή μελέτη και υλοποίηση της δομής του προγράμματος.

Ειδικά η κωδικοποίηση των αμοιβαία αναδρομικών τύπων δημιουργεί αρκετούς επιπρόσθετους όρους στην System F^{μ}_{ω}, καθιστώντας αρκετά δύσκολη ως αδύνατη την εξάλειψή τους, σε αντίθεση με την περίπτωση της FIR που είναι αρκετά απλή.

6.1.4 Βελτιστοποίησης γνωστού κατασκευστή

Η βελτιστοποίηση της *περίπτωσης γνωστού κατασκευαστή* (case-of-known-constructor optimization) είναι σημαντική για τις συναρτησιακές γλώσσες ([Jone98]). Συχνά όταν πραγματοποιούμε ένα ταίριασμα προτύπου ξέρουμε την δομή της τιμής στην οποία θα πραγματοποιηθεί το ταίριασμα, και μπορούμε να απαλείψουμε τις περιττές ενδιάμεσες κατασκευές και να δουλέψουμε κατευθείαν στην τιμή. Ένα παράδειγμα της βελτιστοποίησης στην γλώσσα FIR είναι:

$$\mathsf{match}\ \{\mathsf{Int}\}\ (\mathsf{Just}\ \{\mathsf{Int}\}\ 1)\ 0\ (\lambda x.x+1) \Longrightarrow (\lambda x.x+1)\ 1$$

Η βελτιστοποίηση αυτή είναι εύκολο να υλοποιηθεί στην FIR, όπου έχουμε γνώση των συναρτήσεων ταιριάσματος και των constructors του κάθε τύπου δεδομένων. Η βελτιστοποίηση αυτή δεν είναι εφαρμόσιμη όταν οι όροι `let` μεταγλωττιστούν σε System F^{μ}_{ω}.

6.2 Μελλοντικές Κατευθύνσεις

Περισσότερη δουλειά μπορεί να γίνει στην σημασιολογία της FIR. Στα προηγούμενα κεφάλαια δεν ορίστηκε ξεχωριστά η σημασιολογία της FIR, αντιθέτως η σημασιολογία της υπαγορεύεται από την μετάφραση σε System F^{μ}_{ω}. Δηλαδή η "σημασία" ενός προγράμματος FIR ορίζεται ως η "σημασία" του προγράμματος System F^{μ}_{ω} στο οποίο μεταφράζεται.

Ένας άλλος άξονας στον οποίον θα μπορούσε να βελτιωθεί η εργασία είναι στην ορθότητα της μεταγλώττισης . Η μεταγλώττιση από την FIR στην System F^{μ}_{ω} είναι ορθή αν οι λειτουργίες της αποτίμησης και της μεταγλώττισης αντιμετατίθενται. Δηλαδή η αποτίμηση ενός προγράμματος FIR και η μεταγλώττιση σε System F^{μ}_{ω}, ακολουθούμενη από την αποτίμηση πρέπει να δίνουν το ίδιο αποτέλεσμα για όλα τα προγράμματα.

Επιπρόσθετα, μία πραγματική γλώσσα προγραμματισμού με χρήση στο blockchain εμπεριέχει και θεωρητικές ατέλειες, όπως υλοποίηση κρυπτογραφικών συναρτήσεων, και built-in τύπους. Η θεωρητική ανάλυση χρειάζεται να συμπεριλάβει και αυτές της θεωρητικά βαρετές, αλλά πρακτικά σημαντικές πτυχές της γλώσσας. Συγκεκριμένα η Plutus Core, η πραγματική εκδοχή της System F^{μ}_{ω} υποστηρίζει sized integers, που κουβαλάνε μαζί το μέγεθός τους. Αυτό το χαρακτηριστικό στοχεύει στην ευκολότερη ανίχνευση λαθών που προκύπτουν από αριθμητικά overflows. Τα στοιχεία αυτά αποτελούν παραδείγματα των εμποδίων που πρέπει να λυθούν από μία γλώσσα που στοχεύει στην χρήση της σε πραγματικά συστήματα στο blockchain.

Βιβλιογραφία

[Ahme17] Amal Ahmed, Dustin Jamner, Jeremy G. Siek and Philip Wadler, "Theorems for Free for Free: Parametricity, with and Without Types", *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 39:1–39:28, August 2017.

[Alte99] Thorsten Altenkirch and Bernhard Reus, "Monadic Presentations of Lambda Terms Using Generalized Inductive Types", in *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, pp. 453–468, 1999.

[augu18] "Augur: A Decentralized Oracle and Prediction Market Platform", 2018.

[Back99] R. Backhouse, P. Jansson, J. Jeuring and L. Meertens, "Generic Programming — An Introduction", in *LNCS*, vol. 1608, pp. 28–115, Springer-Verlag, 1999. Revised version of lecture notes for AFP'98.

[Bare91] Henk Barendregt, "Introduction to Generalized Type Systems", *J. Funct. Program.*, vol. 1, no. 2, pp. 125–154, 1991.

[Biry17] Alex Biryukov, Dmitry Khovratovich and Sergei Tikhomirov, "Findel: Secure Derivative Contracts for Ethereum", in Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore and Markus Jakobsson, editors, *Financial Cryptography and Data Security*, pp. 453–467, Cham, 2017, Springer International Publishing.

[Brow17] Matt Brown and Jens Palsberg, "Typed self-evaluation via intensional type functions", in Castagna and Gordon [Cast17], pp. 415–428.

[Cai16] Yufei Cai, Paolo G. Giarrusso and Klaus Ostermann, "System F-omega with Equirecursive Types for Datatype-generic Programming", *SIGPLAN Not.*, vol. 51, no. 1, pp. 30–43, January 2016.

[Care09] Jacques Carette, Oleg Kiselyov and Chung-chieh Shan, "Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages", *J. Funct. Program.*, vol. 19, no. 5, pp. 509–543, September 2009.

[Cast17] Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, ACM, 2017.

[Cunh11] Alcino Cunha and Hugo Pacheco, "Algebraic Specialization of Generic Functions for Recursive Types", *Electr. Notes Theor. Comput. Sci.*, vol. 229, no. 5, pp. 57–74, 2011.

[Drey04] Derek Dreyer, "A Type System for Well-founded Recursion", *SIGPLAN Not.*, vol. 39, no. 1, pp. 293–305, January 2004.

[Drey05] Derek Dreyer, "Understanding and evolving the ML module system", Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 2005.

[Drey07] Derek Dreyer, "A Type System for Recursive Modules", *SIGPLAN Not.*, vol. 42, no. 9, pp. 289–302, October 2007.

[dVri14] Edsko de Vries and Andres Löh, "True sums of products", in Magalhães and Rompf [Maga14], pp. 83–94.

[Fell86] Matthias Felleisen and Daniel P. Friedman, "Control operators, the SECD-machine, and the lambda-calculus", in *3rd Working Conference on the Formal Description of Programming Concepts*, August 1986.

[Fell92] Matthias Felleisen and Robert Hieb, "The Revised Report on the Syntactic Theories of Sequential Control and State", *Theor. Comput. Sci.*, vol. 103, no. 2, pp. 235–271, September 1992.

[Frie07] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek and Onnie Lynn Winebarger, "Improving the Lazy Krivine Machine", *Higher Order Symbol. Comput.*, vol. 20, no. 3, pp. 271–293, September 2007.

[Geuv14] Herman Geuvers, "The Church-Scott representation of inductive and coinductive data", Types 2014, Paris, Draft. , 2014.

[Gibb12] Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, vol. 7470 of *Lecture Notes in Computer Science*, Springer, 2012.

[Gira72] Jean-Yves Girard, *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Thèse d'état, Université Paris 7, June 1972.

[Harp12] Robert Harper, *Practical Foundations for Programming Languages*, Cambridge University Press, New York, NY, USA, 2012.

[Harz18] Dominik Harz and William J. Knottenbelt, "Towards Safer Smart Contracts: A Survey of Languages and Verification Methods", *CoRR*, vol. abs/1809.09805, 2018.

[Hild18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu and Grigore Rosu, "KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine", in *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pp. 204–217, 2018.

[Hold06] Stefan Holdermans, Johan Jeuring, Andres Löh and Alexey Rodriguez, "Generic Views on Data Types", in Uustalu [Uust06], pp. 209–234.

[Jans00] Patrik Jansson and Johan Jeuring, "A Framework for Polytypic Programming on Terms, with an Application to Rewriting", in *Proceedings of the Workshop on Generic Programming (WGP2000)*, Utrecht University, 2000. UU-CS-2000-19.

[Joak11] Michael Flænø Werk Joakim Ahnfelt-Rønne, *Pricing composable contracts on the GP-GPU*, Ph.D. thesis, University of Copenhagen, 2011.

[Jone97] Simon Peyton Jones and Erik Meijer, "Henk: A Typed Intermediate Language", in *In Proc. First Int'l Workshop on Types in Compilation*, 1997.

[Jone98] Simon L Peyton Jones and AndréL M Santos, "A transformation-based optimiser for Haskell", *Science of computer programming*, vol. 32, no. 1-3, pp. 3–47, 1998.

[Jone00] Simon Peyton Jones, Jean marc Eber and Julian Seward, "Composing contracts: an adventure in financial engineering - Functional Pearl", 2000.

[Jone03] S. L. Peyton Jones and J-M. Eber, "How to Write a Financial Contract", 2003.

[Katz17] Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, vol. 10401 of *Lecture Notes in Computer Science*, Springer, 2017.

[Kiay17] Aggelos Kiayias, Alexander Russell, Bernardo David and Roman Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol", in Katz and Shacham [Katz17], pp. 357–388.

[Kise10] Oleg Kiselyov, "Typed Tagless Final Interpreters", in Gibbons [Gibb12], pp. 130–174.

[Koop14] Pieter Koopman, Rinus Plasmeijer and Jan Martin Jansen, "Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl", in *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages*, IFL '14, pp. 4:1–4:12, New York, NY, USA, 2014, ACM.

[Loh11] Andres Löh and José Pedro Magalhães, "Generic Programming with Indexed Functors", in *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, pp. 1–12, New York, NY, USA, 2011, ACM.

[Maga14] José Pedro Magalhães and Tiark Rompf, editors, *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, ACM, 2014.

[Mira18] Victor Cacciari Miraldo and Alejandro Serrano, "Sums of Products for Mutually Recursive Datatypes: The Appropriationist's View on Generic Programming", in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2018, pp. 65–77, New York, NY, USA, 2018, ACM.

[Naka08] Satoshi Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf", 2008.

[OCon17] Russell O'Connor, "Simplicity: A New Language for Blockchains", in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pp. 107–120, New York, NY, USA, 2017, ACM.

[Park18] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian and Grigore Rosu, "A formal verification tool for Ethereum VM bytecode", in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pp. 912–915, 2018.

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

[Pier05] Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.

[Reyn74] John C. Reynolds, "Towards a theory of type structure", in *Colloque sur la Programmation*, vol. 19 of *Lecture Notes in Computer Science*, pp. 408–425, Springer, April 1974.

[Scot63] Dana Scott, "A System of Functional Abstraction". Unpublished lecture notes, 1963.

[soli19] "Solidity v0.5.8 documentation", 2019.

[Stee52] Norman Steenrod and Samuel Eilenberg, *Foundations of Algebraic Topology*, Princeton University Press, 1952.

[Stir13] Colin Stirling, "Proof Systems for Retracts in the Simply Typed Lambda Calculus", in *ICALP 2013: Automata, Languages, and Programming*, pp. 398–409, 2013.

[Szab97] Nick Szabo, "Formalizing and Securing Relationships on Public Networks", *First Monday*, vol. 2, no. 9, 1997.

[Szab02a] Szabo, "A formal language for analyzing smart contracts", 2002.

[Szab02b] Nick Szabo, "A Formal Language for Analyzing Contracts", 2002.

[Thom18] Simon Thompson and Pablo Lamela Seijas, "Marlowe: Financial Contracts on Blockchain", in *ISoLA 2018: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Lecture Notes in Computer Science, Switzerland, October 2018, Springer-Verlag Berlin.

[Uust06] Tarmo Uustalu, editor, *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings*, vol. 4014 of *Lecture Notes in Computer Science*, Springer, 2006.

[Vand03] Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper and Perry Cheng, "Typed Compilation of Recursive Datatypes", *SIGPLAN Not.*, vol. 38, no. 3, pp. 98–108, January 2003.

[Wadl90] Philip Wadler, "Recursive types for free!", 1990.

[Wood14] Gavin Wood, "Ethereum: A secure decentralised generalised transaction ledger", *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[Yaku09] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh and Johan Jeuring, "Generic Programming with Fixed Points for Mutually Recursive Datatypes", *SIGPLAN Not.*, vol. 44, no. 9, pp. 233–244, August 2009.

[Yall19] Jeremy Yallop and Oleg Kiselyov, "Generating Mutually Recursive Definitions", in *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pp. 75–81, New York, NY, USA, 2019, ACM.

[Zahn18a] Joachim Zahnentferner, "An Abstract Model of UTxO-based Cryptocurrencies with Scripts", *IACR Cryptology ePrint Archive*, vol. 2018, p. 469, 2018.

[Zahn18b] Joachim Zahnentferner, "Chimeric Ledgers: Translating and Unifying UTXO-based and Account-based Cryptocurrencies", *IACR Cryptology ePrint Archive*, vol. 2018, p. 262, 2018.