ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εφαρμογές της Μαθηματικής Λογικής

(ΕΑΡΙΝΟ ΕΞΑΜΗΝΟ 2016-2017)

Εξαμηνιαία Εργασία με θέμα:

λ-Λογισμός και οι Εφαρμογές του στις Γλώσσες Προγραμματισμού

Ομάδα:

Πετρόπουλος-Τράκας Ευθύμης (031 11525)
Ροδίτης-Κουτσαντώνης Ορέστης (031 11052)

Επιβλέπων:

Στεφανέας Πέτρος

λ-Λογισμός (λ-calculus)[1] [2]

Ο λ-λογισμός είναι ένα τυπικό σύστημα της Μαθηματικής Λογικής για την έκφραση της υπολογισιμότητας με βάση την αφαίρεση και σύνθεση συναρτήσεων, με τη χρήση δέσμευσης μεταβλητών κι αντικατάστασης τους. Η δημιουργία του αποδίδεται στον Alonso Church, ως μέρος της έρευνας του για τη θεμελίωση των μαθηματικών το 1930 και τον Stephen Kleene, ο οποίος απέδειξε την λογική του ασυνέπεια το 1935. Το 1936, ο Church, απομονώνοντας το κομμάτι που αφορά την υπολογισιμότητα δημιούργησε αυτό που είναι τώρα γνωστό ως λ-λογισμός χωρίς τύπους και αργότερα, το 1940, τον λ-λογισμό με τύπους, ένα σύστημα υπολογιστικά πιο ασθενές. Ο Church χρησιμοποιώντας το λ-λογισμό έδωσε αρνητική απάντηση στο πρόβλημα απόφασης του David Hilbert και χρησιμοποιήθηκε για να ορίσει τις υπολογίσιμες συναρτήσεις. Πλέον αποτελεί ένα καθολικό μοντέλο υπολογισιμότητας που μπορεί να χρησιμοποιηθεί για να προσομοιώσει οποιαδήποτε μηχανή Τούρινγκ. Η δομή των εκφράσεων του λ-λογισμού στην απλούστερη περίπτωση του αποτελούνται από τρείς μορφές και δύο είδη λειτουργιών αναγωγής. Οι εκφράσεις αποτελούνται από:

- μεταβλητές $x_1 \, . \, x_2 \, \ldots \, x_n$
- τα σύμβολα αφαίρεσης λ και .
- παρενθέσεις ( )

Το σύνολο των εκφράσεων, Λ, μπορεί να οριστεί αναδρομικά:

1. Αν x είναι μεταβλητή, τότε x ∈ Λ
2. Αν x είναι μεταβλητή και M ∈ Λ, τότε (λ x . M) ∈ Λ
3. Αν M, N ∈ Λ, τότε (M N) ∈ Λ

Οι παραγόμενες από το 2 εκφράσεις ονομάζονται αφαιρέσεις και από το 3 εφαρμογές.

Αναλυτικότερα στον πίνακα που ακολουθεί:

Syntax	Name	Description
a	Variable	A character or string representing a parameter or mathematical/logical value
(λx.M)	Abstraction	Function definition (M is a lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

[1] "Lambda calculus - Wikipedia." https://en.wikipedia.org/wiki/Lambda_calculus. Accessed 26 Oct. 2017.

[2] "Εφαρμογές της λογικής στην πληροφορική" Πέτρος Στεφανέας, Γιώργος Κολέτσος

Operation	Name	Description
(λx.M[x]) → (λy.M[y])	α-conversion	Renaming the bound (formal) variables in the expression. Used to avoid name collisions.
((λx.M) E) → (M[x:=E])	β-reduction	Substituting the bound variable by the argument expression in the body of the abstraction

Σύμφωνα με το θεώρημα Church-Rosser, αν η επαναλαμβανόμενη εφαρμογή των βημάτων αναγωγής κάποια στιγμή τελειώσει, τότε θα έχουμε παράγει μια β-κανονική μορφή. Όπως αναφέραμε παραπάνω, ο λ-λογισμός είναι ισοδύναμος με ένα Turing πλήρες υπολογιστικό μοντέλο και αποτελεί τη μικρότερη δυνατή καθολική γλώσσα προγραμματισμού. Παρόλα αυτά, ο λ-λογισμός είναι μια τυποποίηση της υπολογισιμότητας που δίνει έμφαση στη χρήση κανόνων μετασχηματισμού, και δεν ενδιαφέρεται για τη μηχανή που τους υλοποιεί. Σχετίζεται περισσότερο με το λογισμικό παρά με το υλικό και ως τέτοια έχει αποτελέσει τη βάση του λεγόμενου συναρτησιακού προγραμματισμού και συγκεκριμένα με τον δηλωτικό συναρτησιακό προγραμματισμό.
Στον λ-λογισμό κάθε έκφραση αποτελεί μια ανώνυμη συνάρτηση, ενός μόνο ορίσματος και επιστρέφει μια μοναδική τιμή, το αποτέλεσμά της. Δεν υπάρχουν συναρτήσεις πολλών ορισμάτων, αλλά μερική εφαρμογή μιας συνάρτησης σε μια άλλη (Curry-ing) και σύνθεσή τους. Ως παράδειγμα η εκφράσεις που ακολουθούν είναι ισοδύναμες και η πρώτη αποτελεί συντομογραφία της δεύτερης τυπικής έκφρασης:

$$\lambda x_1 x_2 ... x_n \,.\, (expr) \equiv \lambda x_1 \,.\, \lambda x_2 \,...\, \lambda x_n \,.\, (expr)$$

Χρησιμοποιώντας τις εκφράσεις του λ-λογισμού μπορούμε να εκφράσουμε όλες τις μαθηματικές συναρτήσεις. παραδείγματος χάριν, η συνάρτηση $f(x) = 2 \cdot x$, γράφεται ως $\lambda x \,.\, x + 2$ (ή με τη χρήση οποιασδήποτε άλλης μεταβλητής αντί του x) και η εφαρμογή της συνάρτησης για $x = 5$ ως $(\lambda f \,.\, f\ 3)\ (\lambda x \,.\, 2 \cdot x)$ ή σε πιο “ανεπίσημη” μορφή $(\lambda x \,.\, 2 \cdot x)\ 3$.

Αντικατάσταση

Η αντικατάσταση (substitution), η οποία συμβολίζεται με $E[V := E']$, αντικαθιστά μία μεταβλητή V από την έκφραση E′ σε κάθε σημείο που η V είναι ελεύθερη στην E. Ο ακριβής ορισμός χρειάζεται προσοχή, για να αποφευχθεί η κατά λάθος δέσμευση άλλων μεταβλητών. Για παράδειγμα, δεν είναι σωστή η αντικατάσταση $(\lambda x \,.\, y)\ [y := x]$ να δώσει $(\lambda x \,.\, x)$, αφού η μεταβλητή x που αντικαθιστά το y θα έπρεπε να ήταν ελεύθερη, αλλά καταλήγει να είναι δεσμευμένη. Η σωστή λύση για αυτή την περίπτωση είναι η $(\lambda z \,.\, x)$, έως την α-ισοδυναμία.

Η αντικατάσταση σε όρους του λογισμού λάμδα ορίζεται αναδρομικά στη δομή των όρων, ως εξής:

- $x[x := N]$ ≡ N
- $y[x := N]$ ≡ y, αν $x \neq y$
- $(M_1\ M_2)[x := N]$ ≡ $(M_1[x := N])(M_2[x := N])$
- $(\lambda y.M)[x := N]$ ≡ $\lambda y\ (M[x := N])$, αν $x \neq y$ και $y \notin FV(N)$

Μπορεί κανείς να παρατηρήσει ότι η αντικατάσταση ορίζεται μοναδικά κατά προσέγγιση α-ισοδυναμίας.

α-μετατροπή

Η μετατροπή άλφα επιτρέπει την αλλαγή ονόματος σε δεσμευμένες μεταβλητές. Για παράδειγμα, μία μετατροπή άλφα της έκφρασης $\lambda x\,.\,x$ είναι η $\lambda y\,.\,y$. Συχνά, σε πολλές χρήσεις του λ-λογισμού, όροι που διαφέρουν μόνο κατά μία α-μετατροπή θεωρούνται ισοδύναμοι.
Οι ακριβείς κανόνες για τη μετατροπή άλφα δεν είναι εντελώς προφανείς. Πρώτον, κατά την α-μετατροπή μίας αφαίρεσης, μετονομάζονται μόνο εκείνες οι εμφανίσεις μεταβλητών που δεσμεύονται από την ίδια αφαίρεση. Για παράδειγμα, μία α-μετατροπή της έκφρασης $\lambda x\,.\,\lambda x\,.\,x$ θα ήταν η $\lambda y\,.\,\lambda x\,.\,x$, αλλά όχι η $\lambda y\,.\,\lambda x\,.\,y$. Η τελευταία έχει διαφορετικό νόημα από την αρχική.
Δεύτερον, η μετατροπή άλφα δεν είναι δυνατή αν θα προκαλούσε τη δέσμευση μίας μεταβλητής από διαφορετική αφαίρεση. Για παράδειγμα, αν αντικαταστήσουμε το x με y στην έκφραση $\lambda x\,.\,\lambda y\,.\,x$, παίρνουμε $\lambda y\,.\,\lambda y\,.\,y$, που δεν είναι καθόλου το ίδιο.

β-αναγωγή

Η αναγωγή βήτα εκφράζει την έννοια της εφαρμογής συνάρτησης. Η αναγωγή βήτα της έκφρασης $((\lambda V\ .\ E)E')$ είναι απλά $E[V := E']$.

η-μετατροπή

Η μετατροπή ήτα εκφράζει την έννοια της εκτατικότητας (extensionality), που σε αυτό το πλαίσιο είναι ότι δύο συναρτήσεις είναι ίδιες αν και μόνο αν δίνουν το ίδιο αποτέλεσμα για όλα τα ορίσματα. Η μετατροπή ήτα μετατρέπει την έκφραση $\lambda x\,.\,fx$ σε f και αντίστροφα, όταν η μεταβλητή x δεν εμφανίζεται ελεύθερη μέσα στην f.

β-κανονική Μορφή

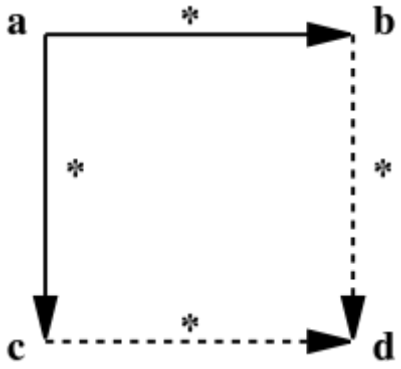
Στον λ-λογισμό, η β-κανονική μορφή είναι η μορφή μια έκφρασης στην οποία δεν μπορούν να εφαρμοστούν επιπλέον β-αναγωγές. Ένας όρος βρίσκεται σε βη-κανονική μορφή, αν δεν μπορούν να εφαρμοστούν ούτε β-αναγωγή ούτε

η-μετατροπή σε αυτόν. Στο λ-λογισμό ένας β-redex, είναι ένας όρος στη μορφή $(\lambda x \,.\, A)\, M$.

Θεώρημα Church-Rosser[3]

Σύμφωνα με το θεώρημα Church-Rosser, αν στο λ-λογισμό ακολουθούμε κανόνες απλοποίησης - ελάττωσης (reduction rules) σε όρους εκφράσεων (β-redex), η σειρά με την οποία θα εφαρμοστούν οι κανόνες αυτοί στην έκφραση, δεν παίζει ρόλο στην κανονική μορφή στην οποία θα μετασχηματιστεί τελικά, αν αυτή υπάρχει. Επομένως, σύμφωνα με αυτό το θεώρημα, κάθε έκφραση έχει το πολύ μία μοναδική κανονική μορφή.
Το θεώρημα αυτό ισχύει για αρκετές παραλλαγές του λ-λογισμού, όπως ο λ-λογισμός με απλούς τύπους, αλγεβρικά συστήματα με εξελιγμένα συστήματα τύπων και για την άλγεβρα του Gordon Plotkin.

Σχηματική αναπαράσταση του θεωρήματος Church-Rosser



Ισομορφισμός Curry-Howard

Ο ισομορφισμός Curry-Howard εκφράζει μια αντιστοιχία μεταξύ των αποδείξεων στον προτασιακό λογισμό και των όρων του λ-λογισμού με τύπους και πιο συγκεκριμένα μια αντιστοιχία στον τρόπο που μια έκφραση του προτασιακού λογισμού αποδεικνύεται από μια θεωρία, και στον τρόπο που μια έκφραση του λ-λογισμού προκύπτει ως β-ισοδύναμη από μια άλλη έκφραση του λ-λογισμού. Η ισοδυναμία αυτή σέβεται την έννοια της αναγωγής, δηλαδή τη μετάβαση από redex σε contractum. Ουσιαστικά, μια απόδειξη μιας φόρμουλας φ αντιστοιχεί σε έναν όρο τύπου φ του λ-λογισμού με τύπους, και η τελική φόρμουλα φ αντιστοιχεί στον τελικό τύπο φ της έκφρασης.
Θεωρούμε γνωστό το συντακτικό και τις έννοιες του προτασιακού λογισμού, οπότε θα δώσουμε μια περιγραφή του ισομορφισμού.
Αν ταυτίσουμε τους ατομικούς τύπους με τις προτασιακές μεταβλητές και στη συνέχεια κάθε φόρμουλα της μορφής (φ → ψ) την ταυτίσουμε με τον τύπο (φ → ψ)

[3] “Church–Rosser theorem" https://en.wikipedia.org/wiki/Church%E2%80%93Rosser_theorem Accessed 26 October 2017

και κάθε φόρμουλα (φ ∧ ψ) με τον τύπο (φ × ψ), τότε μπορούμε να θεωρήσουμε ότι οι φόρμουλες και οι τύποι ταυτίζονται.
Για παράδειγμα η προτασιακή φόρμουλα (A → B) ∧ (C → A) είναι ο τύπος (A → B) × (C → A) όπου A, B, C είναι προτασιακές μεταβλητές (ή ατομικοί τύποι).
Θα χρησιμοποιήσουμε τα γράμματα A,B,C για τις προτασιακές μεταβλητές ή αντίστοιχα για τους ατομικούς τύπος του λ-λογισμού, και τα γράμματα φ,ψ,... για τους προτασιακούς τύπους ή αντίστοιχα για τους τύπους του συστήματος Church.
Σε κάθε απόδειξη Π της φόρμουλας φ από τα πακέτα υποθέσεων $[\varphi_1]_{i1}, \ldots, [\varphi_k]_{ik}$ θα αντιστοιχήσουμε με μοναδικό τρόπο έναν όρο N τύπου φ (δηλ. N^{φ}) με ελεύθερες μεταβλητές $x_{i1}^{\varphi 1}, \ldots, x_{ik}^{\varphi k}$ του αρχικού συστήματος Church με γινόμενο τύπων, ως εξής:

1. Σε κάθε απόδειξη φ^i (από το πακέτο υποθέσεων $[\varphi]_i$) αντιστοιχούμε τη μεταβλητή x_i^{φ} (και αντιστρόφως).
2. Αν $^{\Pi 1}\varphi$ και $^{\Pi 2}\psi$ αντιστοιχούν στα N^{φ} και $M^{\psi.1}$ τότε η απόδειξη $^{\Pi 1}\varphi\ ^{\Pi 2}\psi\ \varphi\wedge\psi$ αντιστοιχεί στον όρο $\langle N^{\varphi}, M^{\psi}\rangle$ (τύπου φ ∧ ψ).
3. Αν η $^{\Pi}\varphi\wedge\psi$ αντιστοιχεί στον $M^{\varphi\wedge\psi}$ τότε η απόδειξη $^{\Pi}\varphi\wedge\psi\ \ \varphi$ αντιστοιχεί στον $\pi^1 M$ και η $^{\Pi}\varphi\wedge\psi\ \psi$ αντιστοιχεί στον $\pi^2 M$.
4. Αν $^{[\varphi]i\Pi}\psi$ αντιστοιχεί στον όρο M^{Ψ} τότε η απόδειξη $^{[\varphi]i'\Pi}\psi\ \varphi\rightarrow\psi$ αντιστοιχεί στον όρο $\lambda x_i^{\varphi} M^{\Psi}$
5. Αν $^{\Pi}\psi$ αντιστοιχεί στον όρο M^{Ψ} τότε $^{\Pi}\psi\ \varphi\rightarrow\psi$ αντιστοιχεί στον όρο $\lambda x^{\varphi}.M^{\Psi}$ τύπου (φ → ψ), (όπου x μπορεί να είναι το πρώτο νέο σύμβολο μεταβλητής). Εδώ στην απόδειξη δεν εκφορτίζουμε κανένα πακέτο υποθέσεων και αντίστοιχα στον όρο χρησιμοποιούμε τη μεταβλητή x^{φ} που δεν εμφανίζεται ελεύθερη στον M^{Ψ} .

Η αντιστοιχία που ορίσαμε είναι ισομορφισμός διότι σέβεται την «πράξη» της αναγωγής, δηλαδή αν μία απόδειξη Q προκύψει από την P με την αναγωγή ενός redex τότε ο αντίστοιχος της Q όρος N προκύπτει από τον αντίστοιχο της P όρο M με την αναγωγή του αντίστοιχου redex.

Church-Turing Υπόθεση[4]

Στην επιστήμη των υπολογιστών, η υπόθεση Church-Turing, αποτελεί μια υπόθεση για τη φύση των υπολογίσιμων συναρτήσεων. Αναφέρει ότι μια συνάρτηση στο πεδίο των φυσικών αριθμών είναι υπολογίσιμη από έναν άνθρωπο ακολουθώντας έναν αλγόριθμο, αγνοώντας περιορισμούς πόρων, αν και μόνο αν είναι υπολογίσιμη από μια μηχανή Turing. Ο Church και ο Turing, απέδειξαν ότι μια συνάρτηση είναι λ-υπολογίσιμη αν και μόνο αν είναι υπολογίσιμη από μια μηχανή Turing και είναι γενικά αναδρομική.

[4] “Church-Turing Thesis” https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis Accessed 26 October 2017

Λ-Λογισμός με τύπους

Ο λ-λογισμός με τύπους είναι ένας τυποποιημένος φορμαλισμός που χρησιμοποιεί το σύμβολο λ, για την ανώνυμη αφαίρεση συνάρτησης. Σε αυτό το πλαίσιο, οι τύποι συνήθως είναι αντικείμενα συντακτικής φύσης που αντιστοιχίζονται σε λ-όρους, η ακριβής φύση ενός τύπου εξαρτάται από τον εκάστοτε λογισμό. Οι λ-λογισμοί με τύπους μπορούν να θεωρηθούν εκλεπτυσμένες εκδόσεις του λ-λογισμού χωρίς τύπους αλλά μπορούν επίσης να θεωρηθούν και σαν βασικότερη θεωρία, με τον λ-λογισμό χωρίς τύπους να είναι ειδική περίπτωση που έχει μόνο έναν τύπο.
Οι λ-λογισμοί με τύπους είναι θεμελιώδεις γλώσσες προγραμματισμού και αποτελούν τη βάση των γλωσσών συναρτησιακού προγραμματισμού με τύπους όπως η ML και η Haskell, και κάπως γενικότερα των γλωσσών προστακτικού προγραμματισμού. Οι λ-λογισμοί με τύπους παίζουν σημαντικό ρόλο στη σχεδίαση των συστημάτων τύπων των γλωσσών προγραμματισμού, όπου η τυποποιησιμότητα συνήθως εκφράζει επιθυμητές ιδιότητες του προγράμματος, όπως π.χ. ότι το πρόγραμμα δε θα προκαλέσει κάποιο σφάλμα μνήμης.
Οι λ-λογισμοί με τύπους έχουν στενή σχέση με τη μαθηματική λογική και τη θεωρία αποδείξεων μέσω του ισομορφισμού Curry-Howard και μπορούν να θεωρηθούν η εσωτερική γλώσσα των κλάσεων των κατηγοριών, π.χ. ο λ-λογισμός με απλούς τύπους είναι η γλώσσα των καρτεσιανά κλειστών κατηγοριών.

Αναδρομή στον λ-λογισμό - Συνδυαστές σταθερού σημείου

Στην επιστήμη των υπολογιστών και στη συνδυαστική λογική, ένας συνδυαστής σταθερού σημείου (fixed-point combinator), αποτελεί μια συνάρτηση υψηλότερης τάξης *fix* τέτοια ώστε για κάθε συνάρτηση f γυρίζει ένα σταθερό σημείο x αυτής της συνάρτησης. Σταθερό σημείο συνάρτησης ονομάζουμε μια τιμή η οποία όταν εφαρμοσθεί σαν είσοδος στη συνάρτηση, γυρίζει τον εαυτό της σαν αποτέλεσμα. Με άλλα λόγια, ο συνδυαστής *fix*, όταν εφαρμοσθεί σε μια συνάρτηση f, επιστρέφει το ίδιο αποτέλεσμα όπως η εφαρμογή της f στο αποτέλεσμα της εφαρμογής της *fix* στην f. Απεικονίζει συνεπώς λύση στην εξίσωση σταθερού σημείου:

$$x = fx$$

Η σχέση που ικανοποιεί συνεπώς είναι η:

$$fix\, f = f\,(fix\, x),\ \forall\ f$$

Ο πιο γνωστός τέτοιος συνδυαστής, που μας ενδιαφέρει άμεσα καθώς ορίζει την αναδρομή στο λ-λογισμό είναι ο Y-Combinator του Curry ο οποίος ορίζεται ως εξής:

$$Y = \lambda f\,.\,(\lambda x\,.\,f(x\,x))\ (\lambda x\,.\,f(x\,x))$$

Σύστημα Τύπων

Σύστημα Τύπων *a la Curry*

Στον καθαρό λ-λογισμό δεν χρησιμοποιήθηκαν τύποι για τους όρους και τις εκφράσεις που κατασκευάζαμε. Οι συναρτήσεις μας δεν είχαν προκαθορισμένο πεδίο ορισμού και πεδίο τιμών, οπότε δεν υπήρχαν περιορισμοί στο τι ορίσματα δεχόταν μια συνάρτηση και στο τι τιμές παρήγαγε. Υπάρχουν όμως περιπτώσεις που θα θέλαμε το σύστημά μας να περιέχει τέτοιους περιορισμούς ώστε να ανταποκρίνεται σε περιπτώσεις του πραγματικού κόσμου και υπαρκτών προβλημάτων της τεχνολογίας λογισμικού, για παράδειγμα, όπου η χρήση σχετικών περιορισμών είναι υποχρεωτική.

Η επιβολή τέτοιων περιορισμών υλοποιείται με την εισαγωγή των τύπων. Συνεπώς, θα θέλαμε να εμπλουτίσουμε τον καθαρό λ-λογισμό με τύπους, κάτι που βέβαια έχει ως αποτέλεσμα νέες παραλλαγές του λ-λογισμού, χωρίς να “πετάμε” την “καθαρή” εκδοχή του.

Ένα πρώτο σύστημα που υλοποιεί αυτή την απαίτηση είναι λ-λογισμός με απλούς τύπους *a la Curry* και ορίζεται ως εξής. Έστω U ένα αριθμήσιμο πλήθος συμβόλων τα οποία θα ονομάζονται τύποι ή μεταβλητές τύπων. Το σύνολο Τ όλων των τύπων στο σύστημα ορίζεται επαγωγικά ως εξής:

1. Κάθε στοιχείο του U είναι τύπος.
2. Αν σ και τ τύποι τότε η έκφραση σ→ τ είναι τύπος

Περιβάλλον ή Βάση θα ονομάζουμε ένα σύνολο της μορφής $\{x_1:t_1,...,x_n:t_n\}$, όπου x_i είναι μεταβλητές-όροι του λ-λογισμού και t_i τύποι. Είναι δηλαδή, ένα σύνολο που απονέμει τύπους στις μεταβλητές του λ-λογισμού. Θεωρούμε ότι x_i=\=x_j για i=\=j, δηλαδή δεν μπορεί να απονεμηθεί πάνω από ένας τύπος σε κάθε όρο. Αν Γ ένα περιβάλλον, τότε dom(Γ) είναι το σύνολο των x_i του περιβάλλοντος.

Έστω Γ περιβάλλον. Ορίζουμε τη σχέση Γ ⊢ M:σ , ορίζοντας επαγωγικά τα αξιώματα και τους κανόνες απαγωγής, ως εξής:

Αξίωμα: Γ, x:τ ⊢ x:τ

Κανόνες:

$$\frac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x.M{:}\sigma \to \tau} \qquad \frac{\Gamma \vdash M{:}\sigma \to \tau \qquad \Gamma \vdash N{:}\sigma}{\Gamma \vdash M\ N{:}\tau}$$

Στον πρώτο κανόνα η μεταβλητή x μπορεί να θεωρηθεί ως ιδιομεταβλητή. Αν έχουμε αποδείξει ότι $\Gamma \vdash \lambda x.M : \sigma \to \tau$, θα θεωρούμε ότι η x δεν εμφανίζεται στο Γ (αλλιώς παίρνουμε ένα α-ισοδύναμο του λx.M) ώστε να μπορούμε να σχηματίσουμε το $\Gamma, x : \sigma \vdash M : \tau$ (π.χ. ως επαγωγική υπόθεση) το οποίο απαιτεί x:σ ∉ Γ.

Όταν γράφουμε ⊢ M : σ εννοούμε ότι ο όρος M τυποποιείται με κενό περιβάλλον.

Οι παραπάνω κανόνες δεν εξασφαλίζουν παρ' όλα αυτά ότι μπορεί να απονεμηθεί τύπος σε κάθε λ-όρο. Αν Μ λ-όρος και υπάρχει Γ και σ ώστε $\Gamma \vdash M : \sigma \to \tau$, τότε λέμε ότι ο Μ είναι τυποποιήσιμος.
Όπως μπορεί να επαχθεί από τον συμβολισμό και τους κανόνες, μπορούμε να θεωρήσουμε ότι ο τύπος τ → σ αναφέρεται σε μια συνάρτηση που δέχεται όρισμα τύπου τ και επιστρέφει αποτέλεσμα τύπου σ.

Έστω Γ ⊢ Μ:σ. Τότε:

1. Γ ⊆ Γ' ⇒ Γ' ⊢ Μ : σ
2. FV(M) ⊆ dom(Γ) , όπου FV οι ελεύθερες μεταβλητές της Μ
3. Αν Γ΄⊆Γ και dom(Γ') = FV(M) τότε Γ' ⊢Μ : σ , δηλαδή σε μια τυποποίηση ενός όρου είναι αρκετό να απονείμουμε τύπους μόνο στις ελεύθερες μεταβλητές του όρου.

Οι τύποι που αποδίδονται σε κάποιο όρο εξαρτώνται από τη μορφή του όρου, σύμφωνα με το παρακάτω Λήμμα Δημιουργίας, σύμφωνα με το οποίο ισχύει ότι:

1. Γ⊢x:σ ⇒ x:σ ∈ Γ
2. $\Gamma \vdash M\,N : \sigma$ ⇒ Υπάρχει τύπος τ ώστε Γ ⊢Μ : τ→ σ και Γ ⊢Ν : τ
3. $\Gamma \vdash \lambda x.M : \sigma$ ⇒ Υπάρχουν τύποι τ και ρ ώστε Γ, x : τ⊢Μ : ρ και σ = τ → ρ

Με σ[α := τ] συμβολίζουμε τον τύπο που προκύπτει αν στο σ αντικαταστήσουμε κάθε εμφάνιση του α με τον τύπο τ. Αντίστοιχα, αν Γ = $\{x_1:t_1,...,x_n:t_n\}$, τότε
Γ[α := τ] = $\{x_1:t_1[\alpha := \tau],...,x_n:t_n[\alpha := \tau]\}$.

Λήμμα Αντικατάστασης

Ισχύει ότι

1. Αν Γ ⊢ Μ : σ, τότε Γ[α := τ] ⊢Μ : σ[α := τ]
2. Αν Γ, x : τ ⊢ Μ : σ και Γ ⊢ Ν : τ , τότε Γ ⊢ Μ [x := Ν] : σ.

Το Λήμμα Αντικατάστασης έχει ως αποτέλεσμα την ακόλουθη πρόταση:
Subject Reduction
Εάν Γ ⊢ Μ : σ και $M_\beta N$ τότε Γ ⊢Ν : σ.

Church–Rosser για τους τυποποιήσιμους όρους

Έστω Γ ⊢ Ν : σ και έστω $M_\beta M'$ και $M_\beta M'$. Τότε υπάρχει όρος L ώστε
$M'_\beta L$ και $M''_\beta L$ και Γ ⊢ L : σ.

Σύστημα Τύπων *a la Church*

Στο σύστημα τύπων *a la Curry* υπήρχε ήδη ο όρος του λ-λογισμού, ο οποίος "εκ των υστέρων" τυποποιούνταν. Για αυτο και είναι γνωστό ως σύστημα απονομής τύπων.

Στο σύστημα Church ο όρος δημιουργείται ταυτόχρονα με την τυποποίησή του. Σε κάθε στάδιο της κατασκευής του “φέρει” τον τύπο του και μόνον όροι με τους σωστούς τύπους μπορούν να συνθέσουν ένα καινούριο όρο.

Τύπος

Οι τύποι είναι ίδιοι με το σύστημα του Curry.
Θα παρουσιάσουμε δύο τρόπους δημιουργίας λ-όρων με τύπους. Ο πρώτος είναι ο αρχικός τρόπος σχηματισμού που χρησιμοποίησε ο Church (αρχικό σύστημα Church) και ο δεύτερος μία προσαρμογή του πρώτου στον τρόπο τυποποίησης à la Curry. (Θα αναφέρεται απλά ως σύστημα à la Church).

Αρχικό σύστημα Church

Μεταβλητές όρων: Για κάθε τύπο σ υπάρχει ένα αριθμήσιμο πλήθος μεταβλητών τύπου σ. Οι μεταβλητές τύπου σ θα γράφονται ως x^{σ}, θα νοούνται δηλαδή ως ζεύγη (x, σ), όπου x είναι ένα σύμβολο μεταβλητής. Υποθέτουμε ότι αν $\sigma \neq \tau$ τότε στις x^{σ} και y^{τ} έχουμε $x \neq y$, δηλαδή χρησιμοποιούμε διαφορετικά σύμβολα μεταβλητών για τους διαφορετικούς τύπους. Αν λοιπόν V_{σ} είναι το σύνολο των συμβόλων μεταβλητών για κάθε τύπο σ [δηλαδή (i). $x \in V_{\sigma} \Rightarrow x^{\sigma}$ είναι μεταβλητή τύπου σ και (ii). $\sigma \neq \tau \Rightarrow V_{\sigma} \cap V_{\tau} = \varnothing$], τότε $V = \cup_{\sigma \in T} V_{\sigma}$ είναι το σύνολο όλων των συμβόλων μεταβλητών.
[Όροι]
Το σύνολο των όρων ορίζεται αναδρομικά ως εξής:

1. Κάθε μεταβλητή x^{σ} είναι όρος τύπου σ.
2. Αν M είναι όρος τύπου $\sigma \rightarrow \tau$ και N όρος τύπου σ τότε (M N) είναι όρος τύπου τ.
3. Αν M είναι όρος τύπου τ και x^{σ} μεταβλητή τύπου σ τότε (λx^{σ}. M) είναι όρος τύπου $\sigma \rightarrow \tau$.

Ένας όρος του αρχικού συστήματος Church είναι σαν ένας όρος του καθαρού λ-λογισμού με τη διαφορά ότι σε όλες τις μεταβλητές, ελεύθερες και δεσμευμένες, υπάρχει μία απονομή-αναγραφή τύπων. Αν «σβήσουμε» αυτές τις απονομές ο όρος αυτός μετατρέπεται σε όρο του λ-λογισμού χωρίς τύπους.

Έστω M όρος του αρχικού συστήματος Church. Ορίζουμε αναδρομικά το |M|, το σβήσιμο των τύπων του M , ως εξής:

1. $|x^{\sigma}| = x$
2. $|M\ N| = |M|\ |N|$
3. $|\lambda x^{\sigma}.\ M| = \lambda x.|M|$

Σχόλιο. Έστω $V = \cup_{\sigma \in T} V_{\sigma}$. Μπορούμε να θεωρήσουμε ότι το σύνολο V είναι το σύνολο των μεταβλητών στο σχηματισμό των όρων του καθαρού λ-λογισμού.

Δηλαδή, ουσιαστικά, έχουμε διαμερίσει τις μεταβλητές του καθαρού λ-λογισμού σε ένα αριθμήσιμο σύνολο (ένα για κάθε τύπο σ) από αριθμήσιμα σύνολα (τα σύμβολα μεταβλητών για τον τύπο σ). Με αυτήν την παραδοχή ο λ-λογισμός παραμένει ως έχει (ίδιοι όροι). Επιπλέον, μπορούμε να τροποποιήσουμε ελαφρώς την έννοια της α-ισοδυναμίας και να δεχόμαστε ότι οι α-ισοδύναμοι όροι παράγονται μόνο με αντικατάσταση των δεσμευμένων μεταβλητών, που αντιστοιχούν σε σύμβολα μεταβλητών του ίδιου τύπου, π.χ. λx.x είναι α-ισοδύναμος με το λy.y μόνο εάν x, y ∈ V_σ για κάποιο τύπο σ. Αυτή η παραδοχή έχει την έννοια ότι, αν για κάποιο όρο M του λ-λογισμού χρειαστεί να θεωρήσουμε έναν α-ισοδύναμό του M', τότε ο M' δημιουργείται από τον M εφαρμόζοντας την περιοριστική α-ισοδυναμία, δηλαδή αντικαθιστώντας δεσμευμένες μεταβλητές μόνο με σύμβολα μεταβλητών που αντιστοιχούν στον ίδιο τύπο. Όλες οι ιδιότητες του λ-λογισμού παραμένουν σε ισχύ με βάση αυτήν την παρατήρηση.

Το αποτέλεσμα της διαμέρισης $V=\cup_{\sigma\in T}V_\sigma$ είναι ότι ο όρος |M| «θυμάται» την απονομή τύπων στις μεταβλητές του. Υπάρχει πάντα μία συνάρτηση T : V → T που ορίζεται ως

$$T(x) = \text{ο μοναδικός } \sigma \text{ ώστε } x \in V_\sigma$$

Αν M όρος, ορίζουμε τον τυποποιημένο C(M) ως εξής:

$C(x) = x^{T(x)}$

C(M N) = C(M) C(N)

$C(\lambda x.M) = \lambda x^{T(x)}.\ C(M)$

Ο C(M) προκύπτει αν σε κάθε μεταβλητή x του M (ελεύθερη ή δεσμευμένη) προσθέσουμε τον αντίστοιχο τύπο σ (αν x ∈ V_σ). Για τυχόντα M ο C(M) δεν είναι όρος Church. Εύκολα βλέπουμε ότι, αν M και N είναι α-ισοδύναμοι με την περιοριστική έννοια που αναφέρθηκε πιο πάνω, τότε C(M) και C(N) είναι α-ισοδύναμοι ως όροι Church.

Αν M είναι όρος Church, τότε C(|M|) = M .

Αν M και N και L είναι όροι Church, τότε

1. $|M\ [x^\sigma := N]| = |M\ |[x := |N|]$
2. $M \to_\beta N$ συνεπάγεται $|M| \to_\beta |N|$
3. αν $|M| \to_\beta L$ τότε $M \to_\beta C(L)$ και C(L) είναι όρος Church
4. αν $|M|_\beta L$ τότε $M_\beta C(L)$

Αποδεικνύεται, επίσης, ότι η ιδιότητα Church-Rosser ικανοποιείται από τους όρους Church.

Ισχύουν οι παρακάτω προτάσεις:

1. Αν M είναι όρος Church τύπου σ και οι ελεύθερες μεταβλητές του |M| είναι μεταξύ των $x_1, \ldots x_k$ και αν $\Gamma = \{x_1 : T(x_1), \ldots x_k : T(x_k)\}$, τότε $\Gamma \vdash |M| : \sigma$.
2. Αν M είναι λ-όρος και $\Gamma \vdash M : \sigma$ (όπου υποθέτουμε ότι αν x : τ ∈ Γ τότε T (x) = τ) τότε C(M) είναι όρος Church τύπου σ.

Η πρόταση αυτή παρουσιάζει την σχέση μεταξύ των όρων Church και την τυποποίηση των λ-όρων στο σύστημα Curry.

Σύστημα Church

Μπορούμε να διαφοροποιήσουμε το αρχικό σύστημα Church και να το παρουσιάσουμε με τον τρόπο με τον οποίο τυποποιούνται οι λ-όροι στο σύστημα Curry. Θα χρειαστούμε την έννοια του προόρου.

Έστω Τ το σύνολο των τύπων. Ορίζουμε ως προόρους τις εκφράσεις Λ_T που σχηματίζονται από τον ακόλουθο ορισμό:

$\Lambda_T V \mid (\lambda x : T.\ \Lambda_T) \mid (\Lambda_T \Lambda_T)$

Η σχέση τυποποίησης $\vdash *$ ορίζεται από:

$$\Gamma, x : \sigma \vdash * x : \sigma$$

$$\frac{\Gamma, x : \sigma \vdash * x : \sigma}{\Gamma \vdash * \lambda x : \sigma.\ M : \sigma \to \tau} \qquad \frac{\Gamma \vdash * M : \sigma \to \tau \qquad \Gamma \vdash * N : \sigma}{\Gamma \vdash * M\ N : \tau}$$

Η σχέση $\Gamma \vdash * M : \sigma$ τυποποιεί τον προ-όρο Μ με τύπο σ στο περιβάλλον Γ. Η α-ισοδυναμία, η αντικατάσταση και η αναγωγή ορίζονται με προφανή τρόπο στους προ-όρους. Εάν Μ είναι προ-όρος, τότε μπορούμε να ρυθμίσουμε τα σύμβολα μεταβλητών ώστε, αν $\lambda x : \sigma. \ldots$ εμφανίζεται στο Μ , τότε $T(x) = \sigma$. Με αυτήν την προϋπόθεση μπορούμε, εάν έχουμε επίσης φροντίσει για το ακόλουθο:

$x : \tau \in \Gamma \rightarrow T(x) = \tau$

να μετατρέψουμε κάθε $\Gamma \vdash * M : \sigma$ σε έναν όρο Church M^c ως εξής:

Κάθε εμφάνιση $\lambda x : \sigma$ στον Μ αντικαθίσταται με λx^σ .

Κάθε άλλη εμφάνιση μεταβλητής x αντικαθίσταται από $x^{T(x)}$.

Αν $\Gamma \vdash * M : \sigma$ (κάτω από τις προϋποθέσεις μεταβλητών που αναπτύχθηκαν πιο πανω), τότε M^c είναι όρος Church. Το αντίστροφο ισχύει επίσης.

Αν Μ είναι όρος Church τύπου ρ και M^π είναι ο προόρος που αποκτάται από τον Μ αντικαθιστώντας κάθε εμφάνιση του λx^σ στον Μ με $\lambda x : \sigma$ και κάθε άλλη εμφάνιση μεταβλητής x^τ στον Μ με x και αν $x_1^{\sigma 1}, \ldots x_k^{\sigma k}$ περιέχουν τις ελεύθερες μεταβλητές του Μ , θα έχουμε ότι:

$x_1 : \sigma_1 , \ldots x_k : \sigma_k \vdash * M : \rho$

Για την τυποποίηση $\vdash *$ ισχύουν τα παρακάτω:

Αν $\Gamma \vdash * M : \sigma$, τότε:

1. $\Gamma \subseteq \Gamma' \Rightarrow \Gamma' \vdash * M : \sigma$
2. $FV(M) \subseteq dom(\Gamma)$
3. $\Gamma' \vdash * M : \sigma$, όπου $dom(\Gamma') = FV(M)$ και $\Gamma' \subseteq \Gamma$

Δημιουργία

Ισχύουν τα ακόλουθα:

1. $\Gamma \vdash * x : \sigma \Rightarrow x : \sigma \in \Gamma$
2. $\Gamma \vdash * M\ N : \sigma \Rightarrow$ υπάρχει τ ώστε $\Gamma \vdash * M : \tau \rightarrow \sigma$ και $\Gamma \vdash * N : \tau$
3. $\Gamma \vdash * \lambda x : \tau.\ M : \sigma \Rightarrow$ υπάρχει ρ ώστε $\sigma = \tau \rightarrow \rho$ και $\Gamma, x : \tau \vdash * M : \rho$

Αντικατάσταση

Ισχύουν τα ακόλουθα:

1. $\Gamma \vdash^* M : \sigma \Rightarrow \Gamma[\alpha := \tau\,] \vdash^* M : \sigma[\alpha := \tau\,]$
2. $\Gamma, x : \tau \vdash^* M : \sigma$ και $\Gamma \vdash^* N : \tau \Rightarrow \Gamma \vdash^* M\,[x := N\,] : \sigma$

Subject reduction

Αν $\Gamma \vdash^* M$ και $M \rightarrow_\beta N$ τότε $\Gamma \vdash^* N$.

Church–Rosser

Αν $\Gamma \vdash^* M : \sigma$ και $M_\beta M_1$ και $M_\beta M_2$, τότε υπάρχει L ώστε $M_{1\beta}L$ και $M_{2\beta}L$ και $\Gamma \vdash^* L : \sigma$.

Μοναδικότητα τυποποίησης

Ισχύουν τα ακόλουθα:

1. $\Gamma \vdash^* M : \sigma$ και $\Gamma \vdash^* M : \tau \Rightarrow \sigma = \tau$
2. $\Gamma \vdash^* M : \sigma$ και $\Gamma \vdash^* N : \tau$ και $M \equiv \beta N \Rightarrow \sigma = \tau$

Η σχέση των τυποποίησεων $\vdash^*$ και $\vdash$ θα είναι ανάλογη με αυτή μεταξύ των όρων Church M και των $\Gamma \vdash M : \sigma$.

Έστω $M, N \in \Lambda_T$.

1. Αν $M \rightarrow \beta\ N$, τότε $|M| \rightarrow \beta\ |N|$.
2. $\Gamma \vdash^* M : \sigma \Rightarrow \Gamma \vdash |M| : \sigma$

Ύψωση

Για όλα τα $M, N \in \Lambda$ (λ-όροι):

1. $|M'| \rightarrow_\beta N \Rightarrow$ υπάρχει $N' \in \Lambda_T$ με $|N'| = N$ ώστε $M' \rightarrow_\beta N'$
2. Αν $\Gamma \vdash M : \sigma$, τότε υπάρχει $M' \in \Lambda_T$ με $|M'| = M$ ώστε $\Gamma \vdash^* M' : \sigma$.

Επεκτάσεις του λ-λογισμού με απλούς τύπους

Πέρα από το σύστημα του λ-λογισμού με απλους τύπους που μόλις είδαμε, υπάρχουν διάφορες επεκτάσεις που μας επιτρέπουν ακόμη μεγαλύτερη εκφραστικότητα, όπως τον ορισμό σύνθετων τύπων που αποτελούν καρτεσιανό γινόμενο πιο απλών τύπων. Υπάρχει επιπλέον η δυνατότητα ορισμού ειδικών τύπων, όπως οι Boolean και Nat (Not a type) που είναι συνηθισμένοι στο πεδίο του προγραμματισμού και της προτασιακής/κατηγορηματικής λογικής ή ο ορισμός του αριθμητικού τύπου, που μαζί με τον ορισμό της συνάρτησης του διαδόχου αριθμού επιτρέπει την ανακατασκευή της αριθμητικής μέσω του λ-λογισμού.

Δηλωτικός Προγραμματισμός[5]

Στην επιστήμη των υπολογιστών ο δηλωτικός προγραμματισμός (declarative programming) αποτελεί ένα προγραμματιστικό υπόδειγμα (πρότυπο), στον οποίο αντίθετα με τον προστακτικό προγραμματισμό, ο υπολογισμός του ζητούμενου αποτελέσματος υπολογίζεται περιγράφοντας τις επιθυμητές ιδιότητες του. Σε αντίθεση με τον προστακτικό προγραμματισμό, που είναι πιο ευρέως διαδεδομένος, στον δηλωτικό προγραμματισμό μπορούμε να πούμε ότι περιγράφουμε το “τι” επιθυμούμε να υπολογίσουμε, ενώ στον προστακτικό το “πως” θέλουμε να υπολογιστεί κάτι. Γνωστές γλώσσες προγραμματισμού δηλωτικού υποδείγματος, είναι η SQL, η Haskell, η LISP, η SML, η Erlang, η Prolog, οι κανονικές εκφράσεις (regular expressions), η CSS και πολλές ακόμη. Οι γλώσσες Haskell και LISP (COMMON LISP) θα μελετηθούν πιο εκτενώς σε επόμενο κεφάλαιο. Ορισμένες γλώσσες λογικού προγραμματισμού όπως οι γλώσσες βάσεων (SQL) και η Prolog, παρότι είναι δηλωτικές κατά κύριο λόγο υποστηρίζουν ένα μείγμα συναρτησιακού και διαδικαστικού (procedural) προγραμματισμού.
Υποπαραδείγματα του υποδείγματος του δηλωτικού προγραμματισμού αποτελούν τα εξής πρότυπα:

- Constraint
    - Constraint Logic
        - Concurrent constraint logic
- Dataflow
    - Flow-based
    - Cell-oriented (spreadsheets)
    - Reactive
- Functional
    - Functional Logic
    - Purely Functional
- Logic
    - Abductive Logic
    - Answer Set
    - Concurrent Logic
    - Functional Logic
    - Inductive Logic

Ο όρος του δηλωτικού προγραμματισμού περιλαμβάνει αρκετά, πιο γνωστά, προγραμματιστικά μοντέλα. Παρακάτω μερικές λεπτομέρειες για μερικά από τα υποδείγματα τα οποία μας απασχολούν σε αυτή την εργασία.

---

[5] “Declarative Programming” https://en.wikipedia.org/wiki/Declarative_programming Accessed 26 October 2017

Constraint Programming

Σε αυτό το υπόδειγμα, για να προγραμματίσουμε, δηλώνουμε σχέσεις μεταξύ μεταβλητών σε μορφή περιορισμών. Οι περιορισμοί αυτοί δηλώνουν τις ιδιότητες της επιθυμητής λύσης - στόχου. Για να λύσουμε το σύνολο των περιορισμών, αποδίδουμε τιμή σε κάθε μεταβλητή, έτσι ώστε η λύση να είναι συνεπής με το μέγιστο αριθμό περιορισμών. Συνήθως ο προγραμματισμός μέσω περιορισμών συναντάται σε συνδυασμό με άλλα υποδείγματα όπως ο συναρτησιακός, ο λογικός ή ακόμη και μερικές φορές σε προστακτικές γλώσσες.

Domain-Specific Languages

Οι γλώσσες συγκεκριμένου τομέα (DSL), έχουν το πλεονέκτημα ότι είναι πολύ χρήσιμες για την επίλυση συγκεκριμένων προβλημάτων και καθώς δεν αποτελούν απαραίτητα Turing πλήρη υπολογιστικά μοντέλα, αυτό κάνει εύκολο να είναι αμιγώς δηλωτικές γλώσσες προγραμματισμού. Μερικές γνωστές γλώσσες αυτού του τύπου είναι οι κανονικές εκφράσεις, το Make (build specification language), η Yacc (Look ahead Left-to-Right parser generator) και αρκετές γλώσσες σήμανσης, όπως η QML, η HTML, η XAML και η XLST. Οι γλώσσες σήμανσης απλά δηλώνουν το τι πρέπει να φαίνεται - σχεδιαστεί (για παράδειγμα σε επίπεδο διεπαφής χρήστη ή στο web) και δεν περιέχουν καμία πληροφορία για το πως θα γίνει αυτό, για τον τρόπο που θα σχεδιαστεί ή για τον τρόπο αλληλεπίδρασης με το χρήστη.

Logic Programming

Στις γλώσσες λογικού προγραμματισμού δηλώνεις σχέσεις και κάνεις ερωτήματα σε αυτές με μορφή κατηγορημάτων. Ο τρόπος με τον οποίο αποθηκεύεται η βάση γνώσης και απαντώνται τα ερωτήματα μέσα σε αυτήν, αφήνεται στην υλοποίηση και στον αυτόματο αποδείκτη θεωρημάτων της γλώσσας, αλλά συνήθως γίνεται με τη χρήση ενοποίησης. Πολλές γλώσσες λογικού προγραμματισμού επιτρέπουν τις παρενέργειες (side effects) και ως αποτέλεσμα δεν είναι “καθαρές” λογικές γλώσσες.

Functional Reactive Programming

Ο συναρτησιακός αντιδραστικός προγραμματισμός, αποτελεί ένα υπόδειγμα αντιδραστικού προγραμματισμού (asynchronous dataflow programming), στον οποίο χρησιμοποιούνται τα δομικά στοιχεία του συναρτησιακού προγραμματισμού, όπως η σύνθεση συναρτήσεων, η εφαρμογή συναρτήσεων σε λίστα (map), η εφαρμογή συνάρτησης με ελάττωση (reduce) και η εφαρμογή συνάρτησης με φιλτράρισμα (filter). Εκτενής χρήση του υποδείγματος γίνεται στα γραφικά περιβάλλοντα, στη ρομποτική και στη μουσική, με σκοπό την πιο εύκολη υλοποίηση μέσω της ρητής χρονικής μοντελοποίησης του προβλήματος.

Hybrid Languages

Οι υβριδικές γλώσσες, αποτελούν μια κλάση γλωσσών προγραμματισμού που αποτελούνται από τη σύνθεση δύο ή περισσότερων υποδειγμάτων σε ένα κοινό πρότυπο. Τέτοιος γλώσσες αποτελούν τα Makefiles, που αφενός προσδιορίζουν τις εξαρτήσεις με δηλωτικό τρόπο, αφετέρου εμπεριέχουν μια προστακτική λίστα από ενέργειες να γίνουν.

Δηλωτικός Προγραμματισμός στη C++11[6] [7]

Εκτός όμως του καθαρά δηλωτικού υποδείγματος γλωσσών προγραμματισμού, υπάρχουν και προστακτικές υβριδικές γλώσσες, που εμπεριέχουν τεχνικές από τον δηλωτικό προγραμματισμό. Ένα τέτοιο παράδειγμα ευρέως χρησιμοποιούμενου προτύπου γλώσσας είναι το C++11 της C++, που με τη χρήση προτύπων μεταβλητού αριθμού παραμέτρων (variadic templates) μπορεί να επιτύχει κανείς δηλωτικό προγραμματισμό.
Επίσης με τη χρήση δηλωτικού προγραμματισμού και προτύπων της C++, είναι δυνατόν να υλοποιήσουμε μια Turing πλήρη μετα-γλώσσα προγραμματισμού, η οποία υπολογίζει όσα της βάζουμε κατά τη διαδικασία μεταγλώττισης. Αναλυτικά για να είναι μια τέτοια μετα-γλώσσα Turing πλήρης, πρέπει το σύστημα στο οποίο υλοποιείται και τρέχει να είναι Turing πλήρες (ο GCC φροντίζει γι' αυτό) και ότι τα ίδια τα πρότυπα μπορούν να υλοποιήσουν μια Turing μηχανή. Η Turing πληρότητα των προτύπων της C++, προκύπτει σαν ατύχημα από τη σύγκρουση κι συγχώνευση δύο χαρακτηριστικών: από την προτυποποίηση και την ειδίκευση προτύπων. Αυτά τα δύο χαρακτηριστικά επιτρέπουν στα πρότυπα της C++ να δρουν σαν μια μη τυποποιημένη γλώσσα επανεγγραφής.
Τα πρότυπα στη C++ δηλώνουν έναν τύπο παραμετροποιήσιμο από ορισμένες σταθερές και τύπους. Παραδείγματος χάριν:

```
template <typename A> struct ListNode {
  A data ;
  ListNode<A>* next ;
} ;
```

Η ειδίκευση προτύπων, επιτρέπει στον προγραμματιστή να καταπατά τον ορισμό ενός προτύπου, για ορισμένο συνδυασμό παραμέτρων στο πρότυπο. Θα μπορούσαμε να καταπατήσουμε τον ορισμό που δώσαμε παραπάνω για το ListNode όταν το πρότυπο χρησιμοποιείται με τύπο unsigned int ως εξής:

---

[6] "Lazy Declarative Programming in C++11"
https://blogs.kde.org/2014/08/30/lazy-declarative-programming-c11

[7] "Lambda-calculus in C++ templates - Matt Might - might.net."
http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/.

```
template <> struct ListNode<unsigned int> {
  /* Don't let them use unsigned ints. */
  int data ;
  ListNode<int>* next ;
}
```

Ένα κλασσικό παράδειγμα της χρησιμότητας αυτού του χαρακτηριστικού, της ειδίκευσης των προτύπων δηλαδή, είναι η υλοποίηση ενός space-efficient διανύσματος από boolean τιμές, δηλαδή ένα vector<bool> που να χρησιμοποιεί ένα bit ανά τιμή κι όχι μια λέξη.
Μπορούμε εύκολα να δούμε λοιπόν, ότι απλές συναρτήσεις όπως αυτή του παραγοντικού, είναι εύκολα υλοποιήσιμες με τη χρήση ειδικευμένων προτύπων.

```
template <int N> struct Factorial  {
  enum { value = N * Factorial<N-1>::value };
};

template <> struct Factorial<0>  {
  enum { value = 1 } ;
};
```

Χωρίς τη χρήση της ειδίκευσης των προτύπων, μια κλήση σε μορφή Factorial<5>::value, δεν θα τερμάτιζε ποτέ, καθώς θα έμπαινε σε ατέρμονη αναδρομή, επανεγγράφοντας τον εαυτό του συνεχώς.
Μια αναλυτική απόδειξη της Turing πληρότητας αυτού του συστήματος μπορείτε να βρείτε στις αναφορές της βιβλιογραφίας, στο τέλος του εγγράφου.
Ένα από τα πράγματα που αναδεικνύει αυτή η υλοποίηση της ενσωματωμένης μετα-γλώσσας, είναι ότι κάθε υπολογίσιμη συνάρτηση μπορεί να υπολογιστεί στο χρόνο μεταγλώττισης στη C++, με εκτενή χρήση δηλωτικού προγραμματισμού και των προτύπων της C++11.

Δηλωτικός προγραμματισμός με Python[8]

Μια ενδιαφέρουσα περίπτωση εφαρμογής δηλωτικού προγραμματισμού σε γλώσσες με προστακτικό χαρακτήρα είναι η χρήση της γλώσσας Python σε πραγματικές εφαρμογές. Η Python είναι μια πλήρης γλώσσα κατά βάση προστακτικού προγραμματισμού, αντικειμενοστραφής, με ισχυρά χαρακτηριστικά και δυναμικό έλεγχο τύπων. Τα προγράμματα της Python συνήθως τρέχουν σε έναν διερμηνευτή της γλώσσας. Για το λόγο αυτό, η απόδοση μεταξύ ενός προγράμματος που έχει γραφεί αντίστοιχα με τις βασικές δομές της Python, και του ίδιου προγράμματος που έχει γραφεί σε γλώσσα χαμηλότερου επιπέδου, διαφέρει σημαντικά, με τα

[8] “Python as a Declarative Programming Language”
http://www.benfrederickson.com/python-as-a-declarative-programming-language/

προγράμματα της Python να μπορούν να είναι δεκάδες φορές πιο αργά από μεταφρασμένα προγράμματα. Η Python όμως είναι εφοδιασμένη με ποικίλες βιβλιοθήκες, οι οποίες είναι γραμμένες σε γλώσσες χαμηλού επιπέδου και οι οποίες έχουν την ικανότητα να επικοινωνούν με προγράμματα γραμμένα σε καθαρή Python μέσω συγκεκριμένων προγραμματιστικών διεπαφών, με χαρακτηριστικό παράδειγμα τις βιβλιοθήκες που χρησιμοποιούνται τα τελευταία χρόνια για Data Analysis και Machine Learning. Οι βιβλιοθήκες αυτές, από τη στιγμή που θα λάβουν τα δεδομένα που χρειάζονται από το πρόγραμμα γραμμένο σε Python, έχουν απόδοση παρόμοια ή συγκρίσιμη με προγράμματα γραμμένα για παράδειγμα σε C++. Συνεπώς, σε τέτοια σενάρια χρήσης, ο κώδικας γραμμένος σε Python αποφεύγει να πραγματοποιεί υπολογισμούς και αρκεί να λειτουργεί σαν “κόλλα” ανάμεσα στα πιο γρήγορα κομμάτια της εφαρμογής, δηλώνοντας τους υπολογισμούς που επιθυμεί οι συνδεδεμένες βιβλιοθήκες να πραγματοποιήσουν, παρέχοντας τον τρόπο απόκτησης των δεδομένων και διαχειριζόμενος τη ροή των δεδομένων ανάμεσα στα συστατικά της εφαρμογής, όπου αυτό χρειάζεται. Με αυτό τον τρόπο το πρόγραμμα Python συμπεριφέρεται ως δηλωτική γλώσσα, εξαλείφοντας το μειονέκτημα της μειωμένης απόδοσης και εκμεταλλευόμενο το εύκολο συντακτικό της γλώσσας για την δημιουργία της συνολικής λογικής της εφαρμογής.

Συναρτησιακός προγραμματισμός[9]

Στην επιστήμη των υπολογιστών ο συναρτησιακός προγραμματισμός αποτελεί ένα προγραμματιστικό υπόδειγμα που θεωρεί τον υπολογισμό ως την αποτίμηση μαθηματικών συναρτήσεων και αποφεύγει τις έννοιες της μεταβλητής κατάστασης και των δεδομένων μεταβλητής τιμής. Είναι υποκατηγορία του δηλωτικού προγραμματισμού, δηλαδή τα προγράμματα αποτελούνται από εκφράσεις και δηλώσεις. Στον δηλωτικό προγραμματισμό η τιμή που επιστρέφει μια συνάρτηση στηρίζεται αποκλειστικά και μόνο στις τιμές των ορισμάτων της, συνεπώς είναι ανεξάρτητη από οποιαδήποτε καθολική κατάσταση του προγράμματος και δεν επηρεάζεται από παρενέργειες εκτέλεσης. Η αποφυγή των παρενεργειών και η καθαρότητα ως προς την τιμή που αποδίδει μια συνάρτηση διευκολύνει σημαντικά την κατανόηση και τον έλεγχο ορθότητας των προγραμμάτων, που είναι και από τα πιο σημαντικά πλεονεκτήματα του συναρτησιακού προγραμματισμού.

Η δημιουργία των συναρτησιακών γλωσσών προγραμματισμού στηρίχθηκε θεωρητικά στη μαθηματική θεωρία του λ-λογισμού. Από τις πιο σημαντικές έννοιες του συναρτησιακού προγραμματισμού που έχουν τη βάση τους στον λ-λογισμό είναι η έννοια των πρώτης και ανώτερης τάξης συναρτήσεων. Οι συναρτήσεις θεωρούνται δεδομένα όπως κάθε άλλος τύπος δεδομένου, μπορούν να περαστούν ως ορίσματα σε άλλες συναρτήσεις και μπορούν να επιστραφούν ως αποτελέσματα από άλλες συναρτήσεις. Ακολουθώντας τον λ-λογισμό, οι συναρτήσεις πρώτης τάξης είναι αυτές που δέχονται ένα μόνον όρισμα, ενώ οι συναρτήσεις ανώτερης τάξης, που φαινομενικά δέχονται “πολλά ορίσματα”, στην πραγματικότητα αναπαρίστανται και αντιμετωπίζονται ως συναρτήσεις μιας μεταβλητής, που δέχονται ένα όρισμα και επιστρέφουν αντίστοιχα μια συνάρτηση που δέχεται ένα όρισμα λιγότερο, με την τεχνική του currying. Η τεχνική αυτή επιτρέπει και την μερική εφαρμογή μιας συνάρτησης, δηλαδή μια συνάρτηση ανώτερης τάξης δε χρειάζεται να τροφοδοτηθεί με όλα της τα ορίσματα ώστε να αποτιμηθεί, αλλά ανάλογα με το ποια ορίσματα έχουν αποδοθεί επιστρέφεται ως αποτέλεσμα μια συνάρτηση η οποία δέχεται, με currying, τα υπόλοιπα ορίσματα.

Οι αγνές συναρτήσεις είναι οι συναρτήσεις που δεν έχουν καμία παρενέργεια, πέρα από το αποτέλεσμα που επιστρέφουν. Αυτό έχει ως αποτέλεσμα μια σειρά από χρήσιμες ιδιότητες των αγνών συναρτήσεων:

- Αν το αποτέλεσμα μιας καθαρής συνάρτησης δεν χρησιμοποιείται, μπορεί να αγνοηθεί χωρίς να επηρεαστούν άλλες εκφράσεις του προγράμματος
- Αν μια αγνή συνάρτηση κληθεί με ορίσματα τα οποία δεν προκαλούν παρενέργειες, το αποτέλεσμα της κλήσης για ορίσματα σταθερής τιμής θα είναι κι αυτό σταθερό. Αυτό είναι αληθές στις καθαρά συναρτησιακές

[9] “Functional Programming” https://en.wikipedia.org/wiki/Functional_programming Accessed 26 October 2017

γλώσσες, όπου δεν υπάρχουν εκφράσεις ανάθεσης τιμής, οπότε η τιμή μιας “μεταβλητής” δεν αλλάζει από τη στιγμή που ορίζεται, με συνέπεια η τιμή της να μπορεί να χρησιμοποιηθεί αντί της μεταβλητής σε κάθε σημείο εκτέλεσης. Η ιδιότητα αυτή ονομάζεται referential transparecy. Σε πρακτικό επίπεδο, πέραν του τι σημαίνει αυτό για την ορθότητα του προγράμματος, επιτρέπει επίσης και την αποθήκευση των αποτελεσμάτων εκτέλεσης μιας συνάρτησης στη μνήμη ώστε αν κληθεί ξανά με τα ίδια ορίσματα το αποτέλεσμα να είναι έτοιμο.

- Αν δύο αγνές εκφράσεις δεν έχουν μεταξύ τους εξαρτήσεις δεδομένων, η σειρά εκτέλεσής τους μπορεί να αναστραφεί, ή μπορούν να εκτελεστούν παράλληλα, χωρίς να είναι δυνατή η ανάμιξη των δεδομένων εκτέλεσής τους. Δηλαδή, οι αγνά συναρτησιακές εκφράσεις είναι ασφαλείς για παράλληλη εκτέλεση.
- Αν η γλώσσα στο σύνολό της δεν επιτρέπει παρενέργειες, κάθε στρατηγική αποτίμησης μπορεί να χρησιμοποιηθεί χωρίς να αλλάζει την ορθότητα του προγράμματος. Αυτό δίνει ελευθερία στον μεταφραστή της γλώσσας ώστε να επιλέξει τη βέλτιστη στρατηγική αναδιάρθωσης ή σύνθεσης των αποτιμήσεων των εκφράσεων.

Μιας και οι συναρτησιακές δεν ακολουθούν το μοντέλο του προστακτικού προγραμματισμού, η επανάληψη δεν υλοποιείται με τον τρόπο που υλοποιείται σε μια προστακτική γλώσσα. Αντ' αυτού, η επανάληψη υλοποιείται με τη χρήση αναδρομής, και σύμφωνα με τον λ-λογισμό η αναδρομή και η επανάληψη είναι πράγματι υπολογιστικά ισοδύναμες. Στη γενική περίπτωση η αναδρομή μπορεί να απαιτεί την διατήρηση στοίβας, αν όμως χρησιμοποιείται αναδρομή ουράς, οπότε σε κάθε βήμα μια συνάρτηση καλεί αναδρομικά κάποια άλλη και το τελικό αποτέλεσμα ισούται με το αποτέλεσμα της τελευταίας κλήσης,1 ο compiler μπορεί να το αναγνωρίσει και να πραγματοποιήσει βελτιώσεις, με κυριότερη τη μη τήρηση στοίβας πέρα από τα δεδομένα της τελευταίας κλήσης της αναδρομής.

Οι συνηθισμένες συναρτησιακές γλώσσες επιτρέπουν άπειρα επίπεδα αναδρομής και είναι Turing complete, με συνέπεια να διατηρούν την μη επιλυσιμότητα του προβλήματος τερματισμού, μπορεί να προκαλέσουν αβεβαιότητα σε εκφράσεις εξισωτικής λογικής, και στη γενική περίπτωση απαιτούν την εισαγωγή ασυνεπειών στο σύστημα τύπων της γλώσσας.

Σε αντίθεση, στην υποκατηγορία των γλωσσών του απολύτως συναρτησιακού προγραμματισμού (total functional programming) δεν επιτρέπεται η άπειρη αναδρομή και γενικά δεν είναι πλήρεις κατά Turing. Όμως, μπορούν ακόμη να εκφράσουν ένα μεγάλο εύρος υπολογισμών και αποφεύγουν τα προβλήματα της άπειρης αναδρομής, περιορίζοντας τα προγράμματά τους στο σύνολο των προγραμμάτων που αποδεδειγμένα τερματίζουν.

Οι συναρτησιακές γλώσσες χωρίζονται, ανάλογα με την στρατηγική αποτίμησής τους, χωρίζονται σε οκνηρή και αυστηρή (lazy/strict evaluation). Όσες ακολουθούν την οκνηρή αποτίμηση, αποτιμούν τις εκφράσεις τους μόνο όταν τα αποτελέσματά

τους ζητηθούν, και μόνο στο επίπεδο που χρειάζεται, αντίθετα με την αυστηρή αποτίμηση όπου οι εκφράσεις αποτιμώνται άμεσα και εξ'ολοκλήρου. Η στρατηγική μπορεί να κάνει διαφορά σε περίπτωση που υπάρχουν εκφράσεις που δεν μπορούν να αποτιμηθούν εξ' ολοκλήρου, με χαρακτηριστικό παράδειγμα τις άπειρες λίστες στη Haskell, και σε περιπτώσεις όπου η αποτίμηση μιας έκφρασης θα είχε ως αποτέλεσμα να προκύψει σφάλμα εκτέλεσης, αλλά η αποτίμηση τελικά δε συμβαίνει μιας και η τιμή της έκφρασης δεν ζητείται. Η οκνηρή αποτίμηση στηρίζεται στην τεχνικής της αναγωγής γράφων, όπου κόμβοι του γράφου είναι οι εκφράσεις του προγράμματος και οι ακμές εκφράζουν τις εξαρτήσεις δεδομένων ανάμεσά τους.

Ως θεωρητικό υπόβαθρο χρησιμοποιήθηκε στις συναρτησιακές γλώσσες και ο λ-λογισμός με τύπους και ο λ-λογισμός χωρίς τύπους, με πιο συνηθισμένη την πρώτη περίπτωση. Στην περίπτωση του λ-λογισμού με τύπους, τα μη έγκυρα προγράμματα αναγνωρίζονται κατά τη μετάφραση, έχοντας την πιθανότητα να "κόψει" και εγκυρα προγράμματα που όμως θεωρείται ότι έχουν πιθανότητα σφαλμάτων, ενώ στην περίπτωση του λ-λογισμού χωρίς τύπους, όλα τα έγκυρα προγράμματα γίνονται δεκτά κατά τη μετάφραση, με την πιθανότητα να προκύψουν σφάλματα εκτέλεσης λόγω ασυνέπειας τύπων.

Στις περισσότερες συναρτησιακές γλώσσες, χρησιμοποιείται εκτενώς ο συμπερασμός τύπων, που κάνει μη αναγκαία τη δήλωση τύπων από τον προγραμματιστή, ενώ επιτρέπει και πολυμορφισμό στον κώδικα.

Η χρήση καθαρά συναρτησιακού μοντέλου προγραμματισμού μπορεί να δημιουργήσει δυσκολίες λόγω της μη μεταβλητότητας των "μεταβλητών" και την έλλειψη κεντρικής κατάστασης. Οι γλώσσες που δεν είναι αμιγώς συναρτησιακές παρέχουν μη συναρτησιακούς τρόπους χρήσης κατάστασης με τρόπο παρόμοιο με των προστακτικών γλωσσών, παρέχοντας ταυτόχρονα τα εργαλείο για προγραμματισμό σε συναρτησιακό στυλ και προωθόντας το ως τον κύριο τρόπο προγραμματισμού σε αυτές. Οι καθαρά συναρτησιακές γλώσσες, μπορούν να χρησιμοποιήσουν μια δομή που να αποτελεί ουσιαστικά την κατάσταση εκτέλεσης που θέλουμε να περάσουμε από συνάρτηση σε συνάρτηση, και σε κάθε βήμα ένα αλλαγμένο αντίγραφο αυτής περνιέται στο επόμενο επίπεδο. Η καθαρά συναρτησιακή γλώσσα Haskell χρησιμοποιεί τα monads, μια έννοια από τη θεωρία κατηγοριών, ώστε να παρέχει μια αφαίρεση για τις λειτουργίες εισόδου-εξόδου και για τη διατήρηση κατάστασης εκτέλεσης, διατηρώντας τον καθαρά συναρτησιακό χαρακτήρα της. Περισσότερες λεπτομέρειες επί αυτού θα δούμε στη συνέχεια.

Μια υποκατηγορία γλωσσών, στις οποίες περιλαμβάνονται πολλές συναρτησιακές γλώσσες προγραμματισμού όπως και οι Haskell και LISP που θα δούμε στη συνέχεια, είναι οι γλώσσες προγραμματισμού βασισμένες σε εκφράσεις. Σε αυτές, κάθε οντότητα είναι μια έκφραση που συνεπώς έχει μια ορισμένη τιμή. Τυπικές εξαιρέσεις είναι οι μακροεντολές, οι εντολές προεπεξεργαστή, και οι δηλώσεις ορισμού ζευγαριού ονομάτων-τιμών, που δεν αντιμετωπίζονται ως εκφράσεις.

Σύντομη Εισαγωγή στη Θεωρία Κατηγοριών[10]

Κατηγορίες

Η Θεωρία Κατηγοριών είναι το πεδίο εκείνο των μαθηματικών που εξετάζει τις γενικές ιδιότητες και χαρακτηριστικά των διαφόρων μαθηματικών δομών μέσα από την μελέτη σχέσεων μεταξύ αντικειμένων αυτών των δομών.
Η θεωρία κατηγοριών χρησιμοποιείται για να τυποποιήσει τα μαθηματικά και τις έννοιές της ως συλλογή των αντικειμένων και των βελών. Η θεωρία κατηγοριών μπορεί να χρησιμοποιηθεί για να τυποποιήσει τις έννοιες άλλων υψηλού επιπέδου αφαιρέσεων όπως η καθορισμένη θεωρία, η θεωρία τομέων και η θεωρία ομάδας. Διάφοροι όροι που χρησιμοποιούνται στη θεωρία κατηγορίας, συμπεριλαμβανομένου του όρου “μορφισμός”, διαφέρουν από τις χρήσεις τους μέσα στα μαθηματικά τα ίδια. Στη θεωρία κατηγορίας ένας “μορφισμός” υπακούει ένα σύνολο όρων συγκεκριμένων για την θεωρία την ίδια. Κατά συνέπεια, πρέπει να ληφθεί προσοχή για να γίνει κατανοητό το πλαίσιο στο οποίο γίνονται δηλώσεις.
Μια κατηγορία Γ αποτελείται από τις ακόλουθες τρεις μαθηματικές οντότητες:

- Μια κατηγορία ob (Γ), τα της οποίας στοιχεία καλούνται αντικείμενα
- Μια κατηγορία hom (Γ), τα στοιχεία της οποίας καλούνται μορφισμοί ή χάρτες ή βέλη. Κάθε μορφισμός φ έχει ένα αντικείμενο πηγής α και το αντικείμενο στόχων β. Η έκφραση φ : a → b, θα δηλωνόταν προφορικά ως «το φ είναι ένας μορφισμός από το α στο β». Η έκφραση hom(a, b) - εκφράζεται εναλλακτικά ως homC(a, b), mor(a, b), or C(a, b)- δείχνει την hom-κατηγορία όλων των μορφισμών από το α στο β.
- Μια δυαδική λειτουργία ◦, καλείται σύνθεση μορφισμών, έτσι ώστε για οποιαδήποτε τρία αντικείμενα α, β, και γ, έχουμε

  $$\hom(\beta, \gamma) \times \hom(\alpha, \beta) \to \hom(\alpha, \gamma).$$

  Η σύνθεση του φ: α → β και g: β → γ γράφεται ως g ◦ φ ή gf, κυβερνημένος από δύο αξιώματα:
  - *Προσεταιριστικότητα*: Εάν φ: α → β, g: β → γ και h: γ → δ τότε h ◦ (g ◦ φ) = (h ◦ g) ◦ φ
  - *Ταυτότητα:* Για κάθε αντικείμενο X, υπάρχει ένας μορφισμός: X → X καλείται μορφισμός ταυτότητας για το X, έτσι ώστε για κάθε μορφισμό φ: α → β, έχουμε 1b ◦ φ = φ = φ ◦ 1a. Από τα αξιώματα, μπορεί να αποδειχθεί ότι υπάρχει ακριβώς ένας μορφισμός ταυτότητας για κάθε αντικείμενο. Μερικοί συγγραφείς παρεκκλίνουν από τον ορισμό που δίνεται ακριβώς με τον προσδιορισμό κάθε αντικειμένου με το μορφισμό ταυτότητάς του.

[10] "Category theory - Wikipedia." https://en.wikipedia.org/wiki/Category_theory. Accessed 26 Oct. 2017.

Μορφισμοί

Οι σχέσεις μεταξύ των μορφισμών (όπως το fg = χ) απεικονίζονται συχνά χρησιμοποιώντας τα μεταλλακτικά διαγράμματα, με «τα σημεία» (γωνίες) να αντιπροσωπεύουν τα αντικείμενα και «τα βέλη» να αντιπροσωπεύουν τους μορφισμούς.

Οι μορφισμοί μπορούν να έχουν οποιαδήποτε από τις ακόλουθες ιδιότητες. Ένας μορφισμός φ: α → β είναι :

- *μονομορφισμός* (ή αμφί) εάν το φ ◦ g1 = φ ◦ g2 συνεπάγεται g1 = g2 για όλους τους μορφισμούς g1, g2: X → a.
- *επιμορφισμός* (ή επί) εάν g1 ◦ φ = g2 ◦ φ συνεπάγεται g1 = g2 για όλους τους *μορφισμούς* g1, g2: β → x.
- *ομομορφισμός* εάν το φ είναι αμφί και επί
- *ισομορφισμός* εάν υπάρχει ένα μορφισμός g: β →a έτσι ώστε φ ◦ g = 1b και g ◦ φ = 1a. [4]
- *ενδομορφισμός* αν α=β. ενδ(α) δείχνει την κατηγορία ενδομορφισμού του α
- *αυτομορφισμός* εάν το φ είναι και ένας ενδομορφισμός και ένας ισομορφισμός.
- *retraction* εάν ένα σωστό αντίστροφο του φ υπάρχει, δηλ. εάν υπάρχει ένα μορφισμός g: β → α με το fg = 1b.
- *section* εάν ένα αριστερό αντίστροφο του φ υπάρχει, δηλ. εάν υπάρχει ένα μορφισμός g: β → α με το gf = 1a.

Κάθε retraction είναι ένας επιμορφισμός, και κάθε section είναι μονομορφισμός. Επιπλέον, οι ακόλουθες τρεις δηλώσεις είναι ισοδύναμες:

- f είναι μονομορφισμος και retraction
- f είναι επιμορφισμος και section
- f είναι ισομορφισμος

Ως κατηγορία ορίζουμε λοιπόν τη μαθηματική οντότητα η οποία έχει κάποιες επιθυμητές ιδιότητες και δομή:

i. Μια συλλογή αντικειμένων A,B,...
ii. Μια συλογή μορφισμών / βελών f,g,...
iii. $f: A \to B$, $\Pi \circ (f)$, $\Pi \, \mathrm{T} \, (f)$
iv. ∘: σύνθεσης: $f: A \to B$, $g: B \to C$, $g \circ f: A \to C$
    - προσεταιριστική ιδιότητα
    - $\forall \, A \, \varepsilon \, C, \, id_A : A \to A$

Ουσιαστικά, μπορούμε να πούμε απλοϊκά, ότι η θεωρία κατηγοριών είναι μια αφηρημένη θεωρία συναρτήσεων.

Συναρτητές (Functor)[11]

Οι συναρτητές αποτελούν κατηγορία μορφισμών μεταξύ κατηγοριών κι έχουν τις εξής ιδιότητες:

1. Διατηρούν τα ταυτοτικά βέλη (identity arrows)
2. Διατηρούν τη σύνθεση των σύνθετων βελών-μορφισμών (composible arrows)

$$A \to^{f} B \mapsto^{covariant} FA \to^{F(f)} FB$$

$$A \to^{f} B \mapsto^{contravariant} FB \to^{F(f)} FA$$

Τα Functors δομούν-συντηρούν τους χάρτες μεταξύ των κατηγοριών. Μπορούν να θεωρηθούν ως μορφισμοί στην κατηγορία όλων των κατηγοριών. Ένας Functor Φ από μια κατηγορία Γ σε μια κατηγορία Δ, γράφεται Φ: Γ → Δ, περιλαμβάνει: για κάθε αντικείμενο X στο Γ, ένα αντικείμενο Φ (X) στο Δ και για κάθε μορφισμό φ: X → Y στο Γ, ένα μορφισμό Φ (φ): Φ (X) → Φ (Y), έτσι ώστε οι ακόλουθες δύο ιδιότητες ισχύουν:

- Για κάθε αντικείμενο X στο Γ, Φ (1x) = 1F (X).
- Για όλους τους μορφισμούς φ: X → Y και γ: Y → ζ, Φ (γ ◦ φ) = Φ (γ) ◦ Φ (φ).
- Ένα contravariant functor Φ: Γ → Δ, είναι όπως ένα covariant functor, εκτός από το ότι «αντιστρέφει τους μορφισμούς» («αντιστρέφει όλα τα βέλη»). Πιο συγκεκριμένα, κάθε μορφισμός φ: X → Y στο Γ πρέπει να οριστεί σε ένα μορφισμό Φ (φ): Φ (Y) → Φ (X) στο Δ. Με άλλα λόγια, ένας contravariant functor λειτουργεί ως covariant functor από την αντίθετη κατηγορία Cop στο Δ.

Φυσικοί Μετασχηματισμοί[12]

Ένας φυσικός μετασχηματισμός είναι μια σχέση μεταξύ δύο functors. Το Functors περιγράφει συχνά τις “φυσικές κατασκευές” και οι φυσικοί μετασχηματισμοί κατόπιν περιγράφουν τους “φυσικούς ομομορφισμούς” μεταξύ δύο τέτοιων κατασκευών. Μερικές φορές δύο αρκετά διαφορετικές κατασκευές παράγουν «το ίδιο» αποτέλεσμα. Αυτό εκφράζεται από έναν φυσικό ισομορφισμό μεταξύ των δύο functors. Εάν Φ και G είναι (covariant) functors μεταξύ των κατηγοριών Γ και Δ, τότε ένας φυσικός μετασχηματισμός η από το Φ στο G συνδέει σε κάθε αντικείμενο X στο Γ ένα μορφισμό ηX: Φ (X) → Γ (X) στο Δ έτσι ώστε για κάθε μορφισμό φ: X → Y στο Γ, έχουμε ηY ◦ Φ (φ) = G (φ) ◦ ηX αυτό σημαίνει ότι το ακόλουθο διάγραμμα είναι μεταλλακτικό:
Τα δύο functors Φ και G καλούνται φυσικά ισομορφικά εάν υπάρχει ένας φυσικός μετασχηματισμός από το Φ στο G έτσι ώστε ηX είναι ένας ισομορφισμός για κάθε αντικείμενο X στο Γ.

---

[11] "Functor - Wikipedia." https://en.wikipedia.org/wiki/Functor. Accessed 26 Oct. 2017.
[12] "Natural transformation - Wikipedia." https://en.wikipedia.org/wiki/Natural_transformation. Accessed 26 Oct. 2017.

Μονοειδές (Monoid)[13]

Στη θεωρία κατηγοριών ένα μονοειδές σε μια κατηγορία μονοειδών, είναι ένα αντικείμενο M με δύο μορφισμούς:

- $\mu : \mathrm{M} \otimes \mathrm{M} \rightarrow \mathrm{M}$, που καλείται γινόμενο
- $\eta : \mathrm{I} \rightarrow \mathrm{M}$, που καλείται μονάδα (unit)

και ικανοποιούνται οι παρακάτω σχέσεις:

- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- $e \cdot a = a \cdot e = a, \ \forall \ (a, b, c) \ \varepsilon \ S$

Μονάδα (Monad)[14]

Στον συναρτησιακό προγραμματισμό ηλεκτρονικών υπολογιστών, μια μονάδα (monad) είναι μια δομή η οποία αναπαριστά τους υπολογισμούς οι οποίοι ορίζονται σε μια σειρά από βήματα. Ο τύπος της δομής μονάδας ορίζει αυτή την σειρά εφαρμογής υπολογισμών ή μια κλήση ενθυλακωμένων συναρτήσεων του ίδιου τύπου. Αυτή η δομή επιτρέπει στον προγραμματιστή να δημιουργεί, με τεχνικές σωλήνωσης (pipelining), επεξεργασίες οι οποίες επεξεργάζονται δεδομένα με βήματα. Σε κάθε ενέργεια υπάρχουν περαιτέρω κανόνες επεξεργασίας οι οποίες παρέχονται από την μονάδα. Παραθέτοντας τα λόγια του Mac Lane:
“All told, a monad in X is just a monoid in the category of endofunctors of X, with product x replaced by composition of endofunctors and unit set by the identity endofunctor”.

[13] "Monoid (category theory) - Wikipedia." https://en.wikipedia.org/wiki/Monoid_(category_theory). Accessed 26 Oct. 2017.
[14] "Monad (category theory) - Wikipedia." https://en.wikipedia.org/wiki/Monad_(category_theory). Accessed 26 Oct. 2017.

Γλώσσα Προγραμματισμού Haskell[15]

Εισαγωγή

Η Haskell είναι μια πρότυπη αμιγώς συναρτησιακή γλώσσα προγραμματισμού γενικής χρήσης με μη-αυστηρή σημασιολογία και ισχυρούς τύπους. Πήρε το όνομά της από τον επιστήμονα της λογικής Haskell Curry. Στη Haskell, "μια συνάρτηση είναι μέλος πρώτης τάξης" της γλώσσας προγραμματισμού. Ως συναρτησιακή γλώσσα προγραμματισμού, χρησιμοποιεί σαν κύρια δομή ελέγχου τη συνάρτηση. Η γλώσσα βασίζεται στις παρατηρήσεις του Haskell Curry και των πνευματικών του απογόνων, ότι "μια απόδειξη είναι ένα πρόγραμμα και ο μαθηματικός τύπος που αποδεικνύει είναι ο τύπος του προγράμματος".
Η Haskell χρησιμοποιεί οκνηρή αποτίμηση, ταίριασμα προτύπων (pattern matching), συμπερίληψη λιστών (list comprehensions), κλάσεις τύπων (typeclasses), και πολυμορφισμό τύπων. Είναι μια αμιγώς συναρτησιακή γλώσσα, που σημαίνει ότι οι συναρτήσεις της δεν έχουν παρενέργειες. Υπάρχει ένας ξεχωριστός τύπος που απεικονίζει τις παρενέργειες, ορθογώνιος προς τον τύπο των συναρτήσεων. Μια αμιγής συνάρτηση μπορεί να επιστρέψει μια παρενέργεια, η οποία στη συνέχεια εκτελείται, μοντελοποιώντας με αυτόν τον τρόπο τις μη-αμιγείς (impure) συναρτήσεις των άλλων γλωσσών.
Η Haskell έχει ένα ισχυρό σύστημα τύπων βασισμένο στην εξαγωγή τύπων. Η βασική συνεισφορά της Haskell σε αυτό το πεδίο είναι η προσθήκη των κλάσεων τύπων, οι οποίες αρχικά δημιουργήθηκαν σαν ένας αυστηρός τρόπος να προστεθεί η υπερφόρτωση στη γλώσσα, αλλά στη συνέχεια βρήκαν επιπλέον χρήσεις.
Ο τύπος που απεικονίζει τις παρενέργειες είναι ένα παράδειγμα μίας Μονάδας (Monad). Οι Μονάδες είναι ένα γενικό πλαίσιο που μπορεί να μοντελοποιήσει διαφορετικούς τρόπους υπολογισμού, όπως η διαχείριση λαθών, ο μη-ντετερμινισμός, η λεξιλογική ανάλυση και η μνήμη συναλλαγών σε λογισμικό (software transactional memory). Οι Μονάδες ορίζονται σαν απλοί τύποι δεδομένων αλλά η Haskell παρέχει κάποιες συντακτικές διευκολύνσεις (syntactic sugar) για τη χρήση τους.
Η βασική υλοποίηση της Haskell, ο GHC, είναι ταυτόχρονα ένας διερμηνέας και ένας μεταγλωττιστής σε κώδικα μηχανής για πολλές πλατφόρμες. Ο GHC διακρίνεται για την υψηλής απόδοσης υλοποίηση του ταυτοχρονισμού και του παραλληλισμού και για το πλούσιο σύστημα τύπων του που περιέχει πρωτοποριακά χαρακτηριστικά όπως οι γενικευμένοι αλγεβρικοί τύποι δεδομένων και οι Οικογένειες Τύπων (Type Classes/Families).

---

[15] "Haskell (programming language) - Wikipedia." https://en.wikipedia.org/wiki/Haskell_(programming_language). Accessed 26 Oct. 2017.

Συντακτικό

Το συντακτικό της Haskell είναι αρκετά επηρεασμένο από γλώσσες όπως η Clean, η FP, η Miranda και η Standard ML. Θυμίζει αρκετά το συντακτικό στα μαθηματικά και γι' αυτό είναι εύκολα κατανοητή από ανθρώπους με ισχυρό μαθηματικό υπόβαθρο ή ερευνητές στο χώρο του λ-λογισμού και της θεωρίας τύπων. Αποτελεί ένα πολύ εκφραστικό συντακτικό, με αποτέλεσμα με λίγο κώδικα να μπορεί να εκφράσει κανείς λύση σε πολύπλοκα και μεγάλα προβλήματα. Σε συνδυασμό με τη σύνθεση συναρτήσεων και την ικανότητα αφαίρεσης, λόγω της αντιμετώπισης των συναρτήσεων ως μέλη πρώτης τάξης, είναι πολύ εύκολο κανείς να γράψει ιδιωματικό κώδικα που να είναι πολύ περιεκτικός και υψηλού επιπέδου, δηλαδή ουσιαστικά κάποιος όταν τον διαβάζει να βλέπει απευθείας τον τρόπο λύσης. Μερικά παραδείγματα ακολουθούν.

Υλοποίηση του παραγοντικού με διάφορους τρόπους:

```
-- με αναδρομή
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- με αναδρομή και χρήση guards
factorial n
  | n < 2 = 1
  | otherwise = n * factorial (n - 1)

-- με λίστες
factorial n = product [1..n]

-- με αναδρομή αλλά χωρίς pattern matching
factorial n = if n > 0 then n * factorial (n-1) else 1
```

Επίσης, όπως είπαμε ήδη, η Haskell έχει αυστηρό σύστημα τύπων και διαθέτει μηχανισμό να δηλώνουμε τύπους συναρτήσεων και ορισμάτων. Σε περίπτωση που δεν δηλώσουμε ρητά τύπους, ο μεταγλωττιστής προσπαθεί να τους βρει μόνος του κι αν αποτύχει μας βγάζει μήνυμα λάθους. Το συντακτικό για τη δήλωση τύπων είναι:

```
factorial :: (Integral a) => a -> a
```

Αυτό διαβάζεται ως: Η συνάρτηση factorial είναι μια συνάρτηση που δέχεται ένα όρισμα a που ανήκει στην οικογένεια τύπων integral και γυρίζει ένα αποτέλεσμα ίδιου τύπου a.

Μια αποδοτική λύση της σειράς Fibonacci, τόσο σε μνήμη όσο και σε χρόνο, με χρήση άπειρης ακολουθίας, για την παραγωγή αριθμών. Αυτό ενώ δεν είναι δυνατό να γίνει σε άλλες γλώσσες τόσο εύκολα ή πολλές φορές και καθόλου, αλλά στη Haskell λόγω οκνηρής αποτίμησης είναι πολύ συχνό φαινόμενο και αποτελεί πολύ καλή υλοποίηση πολλές φορές, ειδικά αν δεν γνωρίζουμε πόσους όρους μπορεί να χρειαστούμε από μια ακολουθία. Γράφοντας την σε μορφή παραγωγής άπειρης

λίστας, μπορούμε να πάρουμε όσους όρους τελικά χρειαζόμαστε με χρήση συναρτήσεων όπως η take και η takeWhile.

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Άλλες λύσεις για τον υπολογισμό όρων της σειράς Fibonacci είναι:

```
-- Δήλωση τύπου
fib :: Int -> Integer

-- Με χρήση αυτοαναφερόμενων δεδομένων
fib n = fibs !! n
        where fibs = 0 : scanl (+) 1 fibs
        -- 0,1,1,2,3,5,...

-- Το ίδιο, ρητά γραμμένο
fib n = fibs !! n
        where fibs = 0 : 1 : next fibs
              next (a : t@(b:_)) = (a+b) : next t

-- Παρόμοια ιδέα, όμως με χρήση της συνάρτησης zipWith
fib n = fibs !! n
        where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- Με χρήση συνάρτησης γεννήτριας
fib n = fibs (0,1) !! n
        where fibs (a,b) = a : fibs (b,a+b)
```

Ιδιαιτερότητες

Η Haskell ως συναρτήσεις είπαμε ότι ορίζει τις ίδιες οντότητες όπως οι συναρτήσεις στα μαθηματικά. Αυτό σημαίνει πως η κλήση μια συναρτησης με τα ίδια ορίσματα θα έχει πάντα το ίδιο αποτέλεσμα. Λόγω της αμιγώς συναρτησιακής της φύσης, η Haskell κάνει πολύ εύκολη την υλοποίηση και τον υπολογισμό ορισμένων προβλημάτων. Επίσης λόγω των ιδιοτήτων της, μας δίνει τη δυνατότητα να επιχειρηματολογήσουμε για την ορθότητα του προγράμματος μας πιο εύκολα από ότι σε άλλες γλώσσες. Οι συναρτήσεις που κάνουν υπολογισμούς είναι χωρίς παρενέργειες και αυτό σημαίνει πως συνήθως είναι πολύ πιο εύκολο να υλοποιήσουμε παράλληλα προβλήματα στη Haskell, καθώς δεν υπάρχει ανάγκη για κλείδωμα κοινών δεδομένων και δομών. Επίσης μέσα από την ανάπτυξη αυτού που ονομάζουν στη Haskell ως Nested Data Parallelism και τη χρήση αυτού του μοντέλου προγραμματισμού, αποφεύγονται τελείως και τα race conditions, καθώς και διευκολύνεται ο παράλληλος προγραμματισμός πάνω σε αραιές δομές δεδομένων (sparse matrix).

Όμως, αυτός ο διαχωρισμός μεταξύ παρενεργειών και υπολογισμών, κάνει πολύ “ξένη” τη συμπεριφορά της γλώσσας όταν θέλουμε να εισάγουμε συναρτήσεις που έχουν παρενέργειες, όπως π.χ. το γράψιμο σε κάποιο αρχείο, το διάβασμα από μια

βάση δεδομένων ή το διαδίκτυο. Αυτό επιτυγχάνεται μέσω δομών που ονομάζονται Μονάδες (monads) και ουσιαστικά δένουν (bind) μια τιμή με την κλήση μια συνάρτησης. Αυτό όμως χρειάζεται κανείς χρόνο και εμπειρία για να το μάθει και να το χρησιμοποιεί αποδοτικά στα προγράμματα του. Επίσης για κάποια χρόνια στην αρχή της ανάπτυξης της γλώσσας, δεν υπήρχαν καθόλου ως έννοια, κάνοντας έτσι τη γλώσσα καθαρά ακαδημαϊκή, καθώς οι εφαρμογές στον πραγματικό κόσμο χρειάζεται να αλληλεπιδράσουν με το περιβάλλον τους. Μόλις εισήχθησαν οι Μονάδες, χρειάστηκε αρκετός χρόνος ώστε να μπορέσουν οι προγραμματιστές να καταλάβουν τη χρήση τους και το τρόπο διαχείρισής τους, καθώς ενώ είναι πράγματα γνωστά σε ανθρώπους που γνωρίζουν θεωρία κατηγοριών από τα μαθηματικά, δεν ήταν κάτι που ήταν γνωστό στους προγραμματιστές. Έτσι για αρκετά χρόνο υπήρξε σαν γλώσσα ακαδημαϊκή με σκοπό να διδάσκει στους μαθητές της επιστήμης των υπολογιστών την έννοια του αμιγώς συναρτησιακού προγραμματισμού.

Επίσης λόγω της οκνηρής αποτίμησης και του συλλέκτη σκουπιδιών (garbage collector) είναι πολύ δύσκολο κάποιος να επιχειρηματολογήσει για το χρόνο εκτέλεσης ενός προγράμματος και γι’ αυτό το λόγο δεν είναι γλώσσα που προτείνεται για υλοποίηση ενσωματωμένων συστημάτων με hard real-time προθεσμίες εκτέλεσης. Ακόμη αυτά τα χαρακτηριστικά κάνουν πολύ δύσκολη τη δουλειά του benchmarking του κώδικα και πολλές φορές είναι δύσκολο για τον προγραμματιστή να βρεί το λόγο για το οποίο η υλοποίηση του καθυστερεί.

Παρ’ όλες όμως όλες τις ιδιαιτερότητες της και της κριτικής που έχει δεχτεί όλα αυτά τα χρόνια, η Haskell παραμένει μια γλώσσα που κατά την άποψη πολλών δείχνει τη σωστή κατεύθυνση στο πως πρέπει να είναι και να σχεδιάζονται οι γλώσσες προγραμματισμού. Επίσης αποτελεί μια πολύ εξελιγμένη γλώσσα με πολύ καθαρές δομές και διαχωρισμό εννοιών, πράγμα που την κάνει ιδανική τόσο για τη διδασκαλία στην επιστήμη των υπολογιστών, όσο και σε εφαρμογές που χρειάζονται και επωφελούνται από τη δυνατότητα παραλληλοποίησης με μικρό κόστος και προσπάθεια.

Γλώσσα Προγραμματισμού LISP[16]

Εισαγωγή

Η γλώσσα LISP, που στην πραγματικότητα είναι μια οικογένεια γλωσσών που περικλείει έναν αριθμό διαλέκτων που επηρεάστηκε σημαντικά από τη θεωρία και τη σύνταξη του λ-λογισμού και υποστηρίζει πολλαπλά προγραμματιστικά υποδείγματα, όπως συναρτησιακό προγραμματισμό, προστακτικό προγραμματισμό και μεταπρογραμματισμό, με δυναμικό και αυστηρό προσδιορισμό τύπων με συμπερασμό. Είναι δεύτερη από γλώσσες προγραμματισμού υψηλού επιπέδου, με πρώτη σε χρονολογία δημιουργίας τη FORTRAN, και αρχικός της σκοπός ήταν η δημιουργία μιας γλώσσας για την έκφραση προγραμμάτων και αλγορίθμων με μαθηματική σύνταξη. Δημιουργήθηκε το 1958 στο MIT από τον John McCarthy. Σήμερα, οι πιο διαδεδομένες διάλεκτοι της LISP είναι η Common LISP (CLISP) και η Schema και η γλώσσα έχει πλέον ορισμένο ANSI standard. Από παλιά είχε συνδεθεί με εφαρμογές τεχνητής νοημοσύνης. Είναι επίσης η πρώτη γλώσσα που εισήγαγε την έννοια και την εφαρμογή του garbage collection κατά την εκτέλεση του προγράμματος.

Το όνομα LISP είναι ακρωνύμιο της φράσης “List Processor”. Οι λίστες είναι η κύρια συντακτική κατασκευή της γλώσσας, και κάθε έκφραση της γλώσσας είναι ένα S-expression, δηλαδή μια κατασκευή από εμφωλευμένες λίστες, οι οποίες μπορούν να αναχθούν με τη σειρά τους σε δυαδικά δέντρα - μια λίστα μπορεί να θεωρηθεί ως ένας κόμβος δέντρου με το αριστερό του παιδί να είναι το πρώτο στοιχείο της λίστας και το δεξί του παιδί να είναι η λίστα που περιλαμβάνει όλα τα επόμενα στοιχεία, ή μπορεί να είναι ένας κόμβος-φύλλο με μια ορισμένη τιμή. Αυτό έχει ως αποτέλεσμα ο κώδικας ενός προγράμματος LISP να μπορεί να αντιμετωπιστεί ως δεδομένα, οπότε αντίστοιχα δεδομένα που αποτελούν έγκυρο κώδικα LISP μπορούν να εκτελεστούν από τον διερμηνέα της γλώσσας, ανεξαρτήτως του τρόπου που έχουν δημιουργηθεί. Αυτό δίνει τη δύναμη σε ένα πρόγραμμα LISP να παράξει κώδικα που στη συνέχεια το ίδιο θα εκτελέσει, και άρα να αλλάξει τον εαυτό του ή άλλα προγράμματα.

Συντακτικό Γλώσσας και Ιδιότητες

Όπως αναφέρθηκε, κάθε έκφραση της LISP είναι ένα S-expression, και δε γίνεται διαχωρισμός ανάμεσα σε κώδικα και δεδομένα, παρά κάθε διαθέσιμη έκφραση

[16] “Lisp Tutorial” https://www.tutorialspoint.com/lisp/index.htm
“Lisp (Programming Language)” https://en.wikipedia.org/wiki/Lisp_(programming_language) Accessed 26 October 2017

γίνεται προσπάθεια να αποτιμηθεί. Η LISP δηλαδή δεν “εκτελεί κώδικα”, αλλά τυπικά “αποτιμά εκφράσεις” σε τιμές, κάτι που τονίζει τον έντονο συναρτησιακό χαρακτήρα της. Πρακτικά, ένα S-expression γράφεται σε μορφή λίστας, η οποία δηλώνεται με παρενθέσεις στα δύο άκρα της και τα στοιχεία της λίστας διαχωρίζονται μεταξύ τους με κενό. Τα στοιχεία της λίστας επιτρέπεται να έχουν διαφορετικούς τύπους μεταξύ τους. Κάθε στοιχείο που περιέχεται σε ένα s-expression είναι ένα atom ή μια λίστα. Ένα άτομο (atom) είναι είτε ένας αριθμός είτε ένα σύμβολο (όνομα). Για να εφαρμόσουμε μια παρένθεση σε ένα σύνολο ορισμάτων, δημιουργούμε μια λίστα, με το πρώτο στοιχείο της να είναι το όνομα (σύμβολο) της συνάρτησης και τα υπόλοιπα στοιχεία της λίστας να είναι τα ορίσματά της. Συνεπώς, η γλώσσα ακολουθεί το prefix notation, μιας και η συνάρτηση τοποθετείται συνακτικά πριν τα ορίσματά της.
Ως παράδειγμα, για να εκτελέσουμε την πρόσθεση 2 + 3 + 4, θα γράφαμε την εξής έκφραση:

```
(+ 2 3 4)
```

η οποία θα μας έδινε αποτέλεσμα 9.
Η LISP προσπαθεί σε κάθε S-expression που συναντά να αντιμετωπίσει το πρώτο στοιχείο ως σύμβολο συνάρτησης, εκτός αν ορίζουμε αλλιώς. Πιο συγκεκριμένα, με τον τελεστή (‘) ορίζουμε ότι δε θέλουμε το S-expression που ακολουθεί να αποτιμηθεί, αλλά να επιστραφεί ως έχει. Π.χ. η έκφραση `‘(print 5)` επιστρέφει μια λίστα με δύο στοιχεία, όπου το πρώτο είναι το σύμβολο της συνάρτησης `print` και το δεύτερο ο αριθμός 5. Αν είχε αποτιμηθεί, η συνάρτηση `print` θα εκτυπώσει ως παρενέργεια τον αριθμό 5 και θα επιστρέψει την τιμή του - δηλαδή επιστρέφει ως τιμή το όρισμα το οποίο εκτύπωσε - οπότε η τιμή της έκφρασης μετά την αποτίμηση θα είναι ο αριθμός 5.
Οι λίστες γράφονται ως παρενθέσεις που περιέχουν όλα τα στοιχεία της λίστας, για συντακτική ευκολία. Στην πραγματικότητα, και σε αντιστοιχία με τις υπόλοιπες συναρτησιακές γλώσσες προγραμματισμού, μια λίστα είναι μονά συνδεδεμένη και είναι το αποτέλεσμα της κλήσης της συνάρτησης `cons`, με το πρώτο όρισμα να είναι η κεφαλή της λίστας και το δεύτερο όρισμα να είναι η ουρά της λίστας, ενώ στην περίπτωση της λίστας με ένα στοιχείο το δεύτερο όρισμα είναι η κενή λίστα (), και το αποτέλεσμα είναι ένα ζευγάρι στοιχείου-ουράς ονομάζεται cons cell. Κάθε λίστα συνεπώς είναι μια σειρά από εμφωλευμένα cons, κάτι που κάνει φανερή τη δυνατότητα αναγωγής ενός S-expression σε δυαδικό δέντρο. Το πρώτο στοιχείο μπορεί να ανακτηθεί με τη συνάρτηση `car/first` και η ουρά με τη συνάρτηση `cdr/rest`. Η συνάρτηση `cons` μπορεί να αναπαρασταθεί συντακτικά και με ένα σύμβολο τελείας. Δηλαδή οι παρακάτω τρεις εκφράσεις είναι ισοδύναμες:

```
(a b c d) ≡ (cons a (cons b (conss c (cons d () ) ) ) ) ≡ (a . (b
. (c. d))) ≡ (a b c . d)
```

Η LISP δεν είναι αμιγώς συναρτησιακή γλώσσα, όπως για παράδειγμα η Haskell. Αν και αρχικά τα atoms ήταν μη μεταβλητά, πλέον είναι δυνατή η αλλαγή τιμών των

μεταβλητών καθώς και η διατήρηση καθολικής κατάστασης του προγράμματος και η αλλαγή τιμών μεταβλητών που έχουν ορισθεί εκτός του S-expression που αποτιμάται. Είναι επίσης δυνατή και για αυτόν τον λόγο η υϊοθέτηση ενός προστακτικού μοντέλου προγραμματισμού, παράλληλα με συναρτησιακό και δηλωτικό. Επιτρέπεται επιπλέον η εκτέλεση επανάληψης χωρίς να είναι υποχρεωτική η χρήση αναδρομής, με τις αντίστοιχες συντακτικές δομές να είναι διαθέσιμες (`do`,`dolist`,`loop`), μαζί με δομές για εκτέλεση κατά συνθήκη (`if`, `cond`). Οι παρενέργειες, όπως το input-output, είναι επιτρεπτές ως μέρος της γλώσσας και δεν υφίστανται ειδική διαχείριση, σε αντίθεση με τη Haskell.
Στη LISP επιτρέπεται ο ορισμός ανώνυμων λ-συναρτήσεων με την χρήση συντακτικού του λ λογισμού, οι οποίες ως δεδομένα μπορούν να χρησιμοποιηθούν μόνο στα σημεία όπου χρειάζεται.
Ένα σημαντικό χαρακτηριστικό της LISP σε σχέση με άλλες γλώσσες, είναι ότι το συντακτικό του κώδικα είναι μια ευθεία αναπαράσταση σε ανθρώπινα αναγνώσιμη μορφή της δομής του προγράμματος στο πιο χαμηλό επίπεδο. Αυτό είναι εξαιρετικά χρήσιμο για τη χρήση μακροεντολών, οι οποίες όμως δεν είναι ένα είδος μεταγλώσσας αλλά συναρτήσεις της ίδιας της αρχικής γλώσσας, συνεπώς μπορούν να χρησιμοποιήσουν όλες τις δυνατότητές της, αλλά και να παραχθούν από οποιαδήποτε δομή μπορεί να παράξει λίστες.
Ακολουθούν ενδεικτικά κομμάτια κώδικα για να καταδείξουν μερικές βασικές λειτουργίες της γλώσσας:

```
(list 1 2 'a 3)
```

Ορισμός στοιχείων λίστας με τη συνάρτηση list, με δήλωση να μην αποτιμηθεί το άτομο a

```
(lambda (arg) (+ arg 1))
```

Ορισμός συνάρτησης λ που δέχεται όρισμα arg και αυξάνει την τιμή του κατά 1

```
(defun foo (a b c d) (+ a b c d))
```

Ορισμός συνάρτησης με όνομα foo που δέχεται μια λίστα με ορίσματα και τα προσθέτει

```
(setf x 5)
```

Ορισμός της τιμής του atom x σε 5 σε καθολική έκταση (όλο το πρόγραμμα)

```
(defmacro setTo10(num) (setf num 10) (print num))
```

Ορισμός μακροεντολής που παίρνει όρισμα num θέτει την τιμή του σε 10 και την εκτυπώνει

```
(cond ((> a 20) (format t "~% a is greater than 20")) (t (format t
"~% value of a is ~d " a)))
```

Η συνάρτηση cond δέχεται μια λίστα από υπολίστες με δύο στοιχεία, όπου αν το πρώτο στοιχείο της υπολίστας αποτιμάτε σε t (True - αληθές) αποτιμάται το δεύτερο στοιχείο της υπολίστας και επιστρέφεται η τιμή του, και αν δεν αποτιμάται σε t δεν αποτιμάται το δεύτερο στοιχείο και η ίδια λειτουργία πραγματοποιείται στην επόμενη υπολίστα.

```
(if (> a 20) (format t "~% a is less than 20")) (format t "~%
value of a is ~d " a)
```

Το κλασσικό if then else σε LISP. Αν το όρισμα αποτιμάται σε t τότε αποτιμάται το πρώτο S-expression που ακολουθεί, αλλιώς το επόμενο.

```
(defun factorial (n)
   (if (= n 0) 1
       (* n (factorial (- n 1)))))
```

Ορισμός συνάρτησης παραγοντικού με αναδρομικό τρόπο.

Επίλογος - Συμπεράσματα

Σε αυτή την εργασία μελετήσαμε το λ-λογισμό και τη σχέση του με τις γλώσσες προγραμματισμού, την επιρροή του στη σύλληψη και στο σχεδιασμό τους τόσο στο κομμάτι της υπολογισιμότητας, όσο και στο κομμάτι του συστήματος τύπων τους. Είτε οι γλώσσες αναπτύχθηκαν εμπειρικά (LISP), είτε προσεκτικά και σταδιακά ώστε να πληρούν ορισμένες αυστηρές προϋποθέσεις και περιορισμούς (Haskell), είτε βρίσκονται κάπου ενδιάμεσα όσον αφορά το σχεδιασμό τους, είναι πολύ σημαντική η βαθύτερη κατανόηση των εννοιών που εκφράζουν για την υπολογισιμότητα και την Turing πληρότητα σαν υπολογιστικές μηχανές. Αποκτώντας μια πλήρη και μη επιφανειακή άποψη για το σχεδιασμό και την υλοποίηση των γλωσσών προγραμματισμού, αποτελεί αναπόσπαστο μέρος της εκπαίδευσης κάθε σύγχρονου προγραμματιστή και ερευνητή της επιστήμης των υπολογιστών, προκειμένου να μπορέσει να αναπτύξει ακόμη πιο αφηρημένα και προχωρημένα συστήματα. Έχοντας συνδέσει την υπολογισιμότητα και την Turing πληρότητα, μπορούμε πλέον να αποφανθούμε για την ικανότητα επίλυσης προβλημάτων των συστημάτων μας και την περαιτέρω ανάπτυξή τους.