Crafting Provers in Racket

absolvent: Adrian Manea
coordonator: Traian Șerbănuță

510, SLA

Introducere și motivație

Ideea: Scurtă introducere în **teoria tipurilor** (MLTT) și verificări folosind **Racket**

MLTT pentru că folosește logica intuiționistă și stă la baza HoTT

Racket pentru că este pornit din Lisp (preferință personală) și creat expres pentru cercetări în limbaje de programare

Plan

1. Elemente de bază pentru MLTT: λ-calcul și tipuri dependente
2. Proust, *"a nano proof assistant"*
3. Pie pentru tipuri dependente

Referințe principale:

1. MLTT: [Martin-Löf, 1980], [Group, 2013], [Nordström and Petersson, 1990];
2. Proust: [Ragde, 2016];
3. Pie: [Friedman and Christiansen, 2018];
4. Racket: [rac, 2020], [Felleisen and Findler, 2014]

Teoria tipurilor în formularea P. Martin-Löf ($\sim$1980)

Dezvoltarea istorică ([Collins, 2012]) începe în jurul paradoxului lui Russell, atinge și probleme de limbaj (Frege, Carnap), atinge maturitatea prin corespondența Curry-Howard(-Lambek)

Distincții clare față de teoria mulțimilor $\Rightarrow$ intenții fundamentale, accentuate de *teoria toposurilor* (categorii + logică)

Abordare intuiționistă: existențialii trebuie instanțiați finitist, terțul exclus dispare, demonstrații bazate pe judecăți și reguli de inferență

Teoria tipurilor în formularea P. Martin-Löf (∼1980)

Formulare (sintaxă) abstractă, interpretări (semantici) diverse:

- propoziții ca tipuri (Curry-Howard, Wadler et al.)
- categorii cartezian închise (Lambek)
- probleme și soluții (Brouwer-Heyting-Kolmogorov)

A mulțime	$a \in A$	implică
A mulțime	a element al A	A nevidă
A propoziție	a demonstrație constructivă a A	A este adevărată
A intenție	a metodă de a îndeplini A	A realizabilă
A problemă	a metodă de rezolvare a A	A are soluție

Ilustrație: Interpretări diverse ale $a \in A$, [Martin-Löf, 1980, p. 4]

Lambda calcul fără tipuri ($\sim$1930)

Intuitiv: un formalism pentru „funcții anonime”.

Gramatica BNF:

$$t ::= x \mid \lambda x.t \mid tt$$

- variabile libere x;
- lambda-abstracții (leagă variabila x în termenul t);
- aplicări ale unui termen pe alt termen.

Exemple: x, $\lambda x.5$, $(\lambda x.5)z$, $(\lambda x.5)(\lambda z.(z+1))$ etc.

Astăzi, majoritatea limbajelor de programare au o formă de a declara funcții anonime cu expresii `lambda`.

Tipuri dependente

Apărute pentru interpretarea cuantificatorilor
Utilizate pentru *expresii parametrizate*, i.e. expresii „la fel” dpdv sintactic, dar cu interpretări diferite.

Judecăți primitive (stil Martin-Löf):

- A este un tip corect format în contextul Γ: $\Gamma \vdash A$ type;
- A și B sînt tipuri egale în contextul Γ: $\Gamma \vdash A \equiv B$ type;
- A este un termen corect format, de tip A, în contextul Γ: $\Gamma \vdash a : A$;
- a și b sînt termeni egali de tip A în contextul Γ: $\Gamma \vdash a \equiv b : A$.

Context = asumpții de tipizare, de forma:

$$\Gamma = x_1 : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1})$$

Tipuri funcționale

Intuitiv, date două tipuri A și B corect formate și populate, regulile de asociere ale elementelor formează *tipul funcțional* $A \to B$.
Inferențele pentru tipuri funcționale se fac cu *lambda calcul*.

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.b(x) : A \to B} \quad (\lambda)$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B} \quad (\text{ev})$$

$$\frac{\Gamma \vdash B \text{ type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \equiv b(x) : B} \quad (\beta)$$

Tipuri Π (produs)

Ideea: *familii de tipuri*, parametrizate de un tip anume.
Pentru fiecare $x : A$, asociem *cîte un tip* $B(x)$.
Notația $\prod_{(x:A)} B(x)$, numite și *funcții dependente*.

Exemplu: **numerele naturale**. Pornim cu $0 : \mathbb{N}$, $S : \mathbb{N} \to \mathbb{N}$.
Principiul de inducție, ca regulă de inferență:

$$\frac{\Gamma, n : \mathbb{N} \vdash P \text{ type} \qquad \Gamma \vdash p_0 : P(0) \qquad \Gamma \vdash p_S : \prod_{(n:\mathbb{N})} P(n) \to P(S(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S) : \prod_{(n:\mathbb{N})} P(n)}(\mathbb{N} - \mathrm{ind}).$$

Proust

```
;; Tipurile cu care lucrăm
(struct Lam (var body) #:transparent)          ; lambda abstracție
(struct App (func arg) #:transparent)          ; aplicare de funcție
(struct Ann (expr type) #:transparent)         ; tipizare explicită
(struct Arrow (domain codomain) #:transparent) ; tip funcțional

;; Exemplu de înțelegerea tipurilor simple
(define (parse-type t)
  (match t                                     ; pattern match pentru tip
[`(,t1 -> ,t2)                                 ; este funcțional?
 (Arrow (parse-type t1)                        ; atunci este Arrow
(parse-type t2))]
(`(,t1 -> ,t2 -> ,r ...)                       ; este "multifunc"?
 (Arrow (parse-type t1)                        ; atunci este "multi-Arrow"
(parse-type `(,t1 -> ,@r))))
[(? symbol? X) X]                              ; este simplu? returnează
[else (error "can't parse this type")]))
```

Proust

```
;; Pentru expresii lambda
;; type-check : Context Expr Type -> Boolean
;; returnează true dacă expr este de tip t în contextul ctx
(define (type-check ctx expr type)
  (match expr
[(Lam x t)                          ; este lambda?
 (match type                        ; atunci type este Arrow
   [(Arrow tt tw) (type-check (cons `(,x ,tt) ctx) t tw)]
   [else (cannot-check ctx expr type)])]
[else (if (equal? (type-infer ctx expr) type) true    ; eroare pentru alte tipuri
  (cannot-check ctx expr type))]))
```

Proust

```
;; Inferența altor tipuri
(define (type-infer ctx expr)
  (match expr
[(Lam _ _) (cannot-infer ctx expr)]          ; lambda separat
[(Ann e t) (type-check ctx e t) t]           ; pentru tipizare explicită
[(App f a)                                   ; aplicarea funcțiilor
 (define tf (type-infer ctx f))
 (match tf                                   ; trebuie să fie de tip Arrow
   [(Arrow tt tw) #:when (type-check ctx a tt) tw]  ; cînd restul merge
   [else (cannot-infer ctx expr)])]
[(? symbol? x)                               ; simboluri
 (cond
  [(assoc x ctx) => second]            ; pentru liste de tip (ctx), tipul e second
   [else (cannot-infer ctx expr)])]))        ; altfel, eroare
```

DEMO (Proust, Pie)

Bibliografie și lecturi suplimentare


(2020).
Racket.
https://racket-lang.org/.
Accessed: March 2020.


Collins, J. (2012).
A History of the Theory of Types.
Lambert Academic Publishing.


Felleisen, M. and Findler, B. (2014).
How to Design Programs.
MIT Press.


Friedman, D. and Christiansen, D. (2018).
The Little Typer.
MIT Press.


Group, T. H. (2013).
Homotopy Type Theory.
https://homotopytypetheory.org/.
Accessed March 2020.


Martin-Löf, P. (1980).
Intuitionistic Type Theory.
Lectures in Padua.
Notes by Giovanni Sambin.


Nordström, B. and Petersson, K. (1990).
Programming in Martin-Löf Type Theory.
Oxford University Press.
Freely available at
http://www.cse.chalmers.se/research/group/logic/book/.


Ragde, P. (2016).
Proust: A Nano Proof Assistant.
In Jeuring, J. and McCarthy, J., editors, *Trends in Functional Programming in Education*, pages 63–75. arXiv/cs.PL.