Università degli Studi di Pisa
Corso di Laurea Magistrale in Informatica
Anno Accademico 2021-2022

Insegnamento di

Foundation of Computing

Pagina del corso: `http:/pages.di.unipi.it/montanari/FOC.html`

Note di

Tipi Ricorsivi, Adeguatezza Computazionale e Tipi Polimorfi

Libro di Testo: J. Mitchell, Foundations for Programming Languages, MIT Press, 1996.

a cura di
Ugo Montanari
Dipartimento di Informatica
Università degli Studi di Pisa
`ugo@di.unipi.it`

I tipi ricorsivi

Si aggiungono variabili alla sintassi dei tipi e una nuova espressione di tipo:

$\sigma ::= \dots \mid t \mid \mu t.\sigma$ si considerano però solo termini di tipo chiusi

Normalmente l'espressione $t = \mu t.\sigma$ viene presa come isomorfismo (dei corrispondenti insiemi)

$$\mu t.\sigma \cong [\mu t.\sigma / t]\sigma$$

Associati ad ogni tipo ricorsivo $\mu t.\sigma$ sono due funzioni:

$$dn^{\mu t.\sigma} : \mu t.\sigma \to [\mu t.\sigma/t]\sigma$$

$$up^{\mu t.\sigma} : [\mu t.\sigma/t]\sigma \to \mu t.\sigma$$

con i seguenti assiomi equazionali:

$$dn^{\mu t.\sigma}(up^{\mu t.\sigma} M) = M \qquad up^{\mu t.\sigma}(dn^{\mu t.\sigma} M) = M \qquad (up/dn)$$

Esempio I numeri naturali, succ, pred e zero?

$$nat \stackrel{def}{=} \mu t.\, unit + t$$

$$0 \stackrel{def}{=} up^{nat}(Inleft^{unit,nat}\, *) : nat \qquad Inleft^{unit,nat}\, \overset{*:unit}{*} : unit + nat$$

$$1 \stackrel{def}{=} up(Inright(up(Inleft\, *))) : nat \qquad up^{nat}(Inleft^{unit,nat}\, *) : nat$$

$\vdots$

$$n \stackrel{def}{=} \underbrace{up(Inright\, up(Inright \dots up}_{n \text{ times}}(Inleft\, *) \dots)) : nat$$

qui si vede bene come funziona il Case!

$succ \stackrel{def}{=} \lambda x{:}nat.\ up\,(Inright\ x)$

$zero? \stackrel{def}{=} \lambda x{:}nat.\ Case^{unit,nat,bool}\ (dn\ x)\,(\lambda y{:}unit.\ true)\,(\lambda z{:}nat.\ false)$

$pred \stackrel{def}{=} \lambda x{:}nat.\ Case^{unit,nat,nat}\ (dn\ x)\,(\lambda y{:}unit.\ 0)\,(\lambda z{:}nat.\ z)$

Per i booleani si può porre:

$bool \stackrel{def}{=} unit + unit$

$true \stackrel{def}{=} Inleft\ *$

$false \stackrel{def}{=} Inright\ *$

$cond_\sigma\ M\ N\ P \stackrel{def}{=} Case^{unit,unit,\sigma}\ M\,(K_{\sigma,unit}\ N)\,(K_{\sigma,unit}\ P)$

$cond_\sigma : bool \to \sigma \to \sigma \to \sigma$

$K_{\sigma,unit} \stackrel{def}{=} \lambda x{:}\sigma.\ \lambda y{:}unit.\ x : \sigma \to unit \to \sigma$

Si può verificare che $\boxed{cond_\sigma\ true\ N\,P = N}$ e $\boxed{cond_\sigma\ false\ N\,P = P}$

Si può verificare:

$zero?\ 0 = true$

$zero?\ n = false \qquad n \neq 0$

$pred\,(succ\ x) = x$

$\left(\lambda x{:}nat.\ Case^{unit,nat,bool}\ (dn\ x)\,(\lambda y{:}unit.\ true)\,(\lambda z{:}nat.\ false)\right)\left(up^{nat}\,(Inleft^{unit,nat}\ *)\right) : bool =$

$Case^{unit,nat,bool}\ (Inleft^{unit,nat}\ *)\,(\lambda y{:}unit.\ true)\,(\lambda z{:}nat.\ false) =$

$(\lambda y{:}unit.\ true)\ * \quad = true$

Ex. 2.6.7 p. 131 <u>Untyped</u> λ-calculus

$$U ::= x \mid UU \mid \lambda x.U$$

assiomi

$$\lambda x.U = \lambda y[y/x]U \qquad y \notin FV(U) \qquad (\alpha)$$

$$(\lambda x.U_1)U_2 = [U_2/x]U_1 \qquad (\beta)$$

$$\lambda x.Ux = U \qquad x \notin FV(U) \qquad (\eta)$$

Sia $\bullet = \mu t.\, t \to t$ l'unico tipo

$dn^\bullet : \bullet \to \bullet \to \bullet \qquad up^\bullet : (\bullet \to \bullet) \to \bullet \qquad dn^\bullet(up^\bullet M) = M \qquad up^\bullet(dn^\bullet M) = M$

Sia $[\![U]\!] = x_1 : \bullet, \dots, x_n : \bullet \rhd M : \bullet$ con $\{x_1, \dots, x_n\} = FV(U)$

per induzione:

$$[\![x]\!] = x : \bullet \rhd x : \bullet$$

$$\frac{[\![U_1]\!] = \Gamma_1 \rhd M_1 : \bullet \qquad [\![U_2]\!] = \Gamma_2 \rhd M_2 : \bullet \qquad FV(U_1U_2) = \{x_1, \dots x_n\}}{[\![U_1U_2]\!] = x_1 : \bullet, \dots, x_n : \bullet \rhd \underbrace{dn^\bullet M_1}_{\bullet \to \bullet} \underbrace{M_2}_{\bullet} : \bullet}$$

$$\frac{[\![U]\!] = \Gamma \rhd M \qquad FV(\lambda x.U) = \{x_1, \dots, x_n\}}{[\![\lambda x.U]\!] = x_1 : \bullet, \dots, x_n : \bullet \rhd up^\bullet \lambda x : \bullet . M : \bullet}$$

Dimostriamo (β)

$$[\![(\lambda x.U_1)U_2]\!] = \Gamma \rhd dn^\bullet(up^\bullet \lambda x : \bullet . M)N = (\lambda x : \bullet . M)N$$

$$= [M/x]N : \bullet = [\![[U_2/x]U_1]\!]$$

where $[\![U_1]\!] = \Gamma_1 \rhd N \qquad [\![U_2]\!] = \Gamma_2 \rhd M$

$\Gamma = \{x_1 : \bullet, \dots, x_n : \bullet\}$ with $\{x_1 \dots x_n\} = FV((\lambda x.U_1)U_2)$

E' possibile avere termini senza forma normale!

$I = \lambda x.xx$

$$[II] = \phi \triangleright \; dn^{\circ}(up^{\circ}\lambda x:\circ.dn^{\circ}xx)(up^{\circ}\lambda x:\circ.dn^{\circ}xx) \xrightarrow{ax}$$

$$(\lambda x:\circ.dn^{\circ}xx)(up^{\circ}\lambda x:\circ.dn^{\circ}xx) \to_{\beta}$$

$$dn^{\circ}(up^{\circ}\lambda x:\circ.dn^{\circ}xx)(up^{\circ}\lambda x:\circ.dn^{\circ}xx) \xrightarrow{ax} \to_{\beta} \cdots$$

Esempio di programma ALGOL nella pagina successiva

Esiste un operatore di punto fisso nel λ-calcolo non tipato

$$Y \stackrel{def}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Abbiamo

$$Yf = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))f \to_{\beta}$$

$$M_f \stackrel{def}{=} (\lambda x.f(xx))\;\lambda x.f(xx)) \to_{\beta}$$

$$f((\lambda x.f(xx))\lambda x.f(xx)) = f\,M_f$$

Esiste anche un altro operatore

$$\Theta = (\lambda x.\lambda f.f(xxf))(\lambda x.\lambda f.f(xxf))$$

direttamente con la proprietà

$$\Theta \to_{\beta} \lambda f.f(\Theta f) \quad \text{cioè} \quad \Theta f \to_{\beta} f(\Theta f)$$

La traduzione funziona, ma noi vogliamo un op. punto fisso che vada bene non soltanto per il tipo $\circ$, ma anche per ogni altro tipo σ, cioè $fix_{\sigma} : (\sigma \to \sigma) \to \sigma$

Ricorsione via tipi ricorsivi

Esempio di partenza:

```
begin
  function fact(f, n)
      if n=0 then 1 else n*f(f, n-1)
  fact(fact, 3)
end
```

si calcola il fattoriale senza ricorsione! Quasi incredibile!

Però fact può ricevere sè stessa come argomento!

Con la nostra struttura di tipi MM non può essere ben tipato

Però dnMM lo può essere!

Se poniamo

$fix_\sigma \stackrel{def}{=} dn\,MM : (\sigma \to \sigma) \to \sigma$

vogliamo la proprietà:

$fix_\sigma \to \lambda f{:}\sigma\to\sigma.\, f(fix_\sigma f)$ che caratterizza l'operatore di ricorsione

cioè

$dn\,MM \to \lambda f{:}\sigma\to\sigma.\, f(dn\,MM\,f) : (\sigma\to\sigma)\to\sigma$

da cui

$M \stackrel{def}{=} up(\lambda x{:}T.\lambda f{:}\sigma\to\sigma.\, f(dn\,x\,x\,f))$

$T \stackrel{def}{=} \mu t.\, t \to (\sigma\to\sigma)\to\sigma$

$fix_\sigma \stackrel{def}{=} (\lambda x{:}T.\lambda f{:}\sigma\to\sigma.\, f(dn\,x\,x\,f))(up(\lambda x{:}T.\lambda f{:}\sigma\to\sigma.\, f(dn\,x\,x\,f)))$

in cui vale $fix_\sigma \to_\beta \lambda f{:}\sigma\to\sigma.\, f(fix_\sigma f) : (\sigma\to\sigma)\to\sigma$

Computational Adequacy of a model A

Program : ∘ no free variables (closed)
∘ observable type : bool, int per PCF

Result : ∘ closed normal form
∘ observable type

diversa da Hoare qui il linguaggio logico è ristretto

Tre tipi di equivalenza tra termini tipati $\Gamma \triangleright M : \tau$

i) assiomatica : $\mathcal{E} \vdash \Gamma \triangleright M = N : \tau \iff (\Gamma \triangleright M : \tau) =_{ax} (\Gamma \triangleright N : \tau)$

ii) denotazionale : $\forall \eta \models \Gamma .\ [\![\Gamma \triangleright M : \tau]\!]\eta = [\![\Gamma \triangleright N : \tau]\!]\eta \overset{def}{\iff} A \models \Gamma \triangleright M = N$

iii) operazionale : è definita eval : programs $\longrightarrow$ results parziale

basica : $eval(P) = R \iff P =_{op} R$ diretto $eval(P) = R \Rightarrow \mathcal{E} \vdash P = R$

di Kleene : $(eval(P_1) = R =_{op} eval(P_2)$ oppure $eval(P_1)\uparrow, eval(P_2)\uparrow) \iff P_1 =_{op} P_2$

congr. osservaz. : $M =_{op} N \iff \forall C .\ C(M) =_{op} C(N)$ programmi

Requisito minimo : computational adequacy tra programmi

$$\forall P . \forall R .\ P =_{ax} R \iff P =_{den} R \iff P =_{op} R$$

Requisito per generici termini

$$=_{ax} \subseteq =_{den} \subseteq =_{op}$$

soundness of denotational semantics

compositionalità della semantica denotazionale

per assurdo : $M =_{den} N$, $M \neq_{op} N$
implica $\exists C .\ C(M) \neq_{op} C(N)$
ma $C(M) =_{den} C(N)$ ric. strutt
cade il requisito minimo.

Di solito

$$=_{ax} \subsetneq =_{den}$$

$\varphi_i = [\![\phi \vdash M_i : int]\!]\eta$

se equiv MdT : $M_i =_{den} M_j \Rightarrow \varphi_i = \varphi_j$
allora $=_{den}$ non r.e.
mentre $=_{ax}$ r.e.

Idealmente

$$=_{den} = =_{op}$$ full abstractness.

// introdurre term. gen. e dimostrazione

$=_{op}$ non uso η_{red} e surjective pairing

3 interpreti: deterministic, nondeterministic, parallel

Computational Adequacy per il PCF

Dati A_{PCF} abbiamo $=_{ax}$ $\subseteq$ $=_{den}$ appena visti, come conseguenza del teorema generale di soundness dei modelli di Plotkin, e da Turing

Per dimostrare $P =_{den} R \iff P =_{op} R$ per M program N result

usiamo un teorema di approssimazione.

Approximate Normal Forms di un termine M

$N \in anf(M)$ se è ottenuto applicando alcune riduzioni ad M e poi sostituendo qualche termine non in nf con $\bot_\sigma$

Quindi se N non contiene $\bot$, allora $M \overset{\text{sotto-}}{\twoheadrightarrow} N$ normal.

Se M ha forma normale $\Rightarrow$ $anf(M)$ è finito

Se $anf(M)$ è infinito $\Longrightarrow$ Tutte approssimazioni di una struttura sintattica infinita detta il "Böhm Tree".

Poniamo

$$[\![anf(M)]\!]\eta = \{ [\![\Gamma \triangleright N : \sigma]\!]\eta \mid N \in anf(M) \}$$ dove $[\![\bot]\!] = \bot$

Lemma $d \in [\![anf(M)]\!]\eta \Longrightarrow d \leq [\![\Gamma \triangleright M : \sigma]\!]\eta$. la semantica corretta è un maggiorante di quelle approssimate

Diamo un limite al numero di volte che si applica $fix. \to fix^n_\sigma$.

$lab(M)$ è l'insieme dei termini derivati a M se si sostituisce

labeled fixpoint fix^n_σ con fix_σ.

$fix^{n+1}_\sigma \to \lambda f : \sigma \to \sigma . f(fix^n_\sigma f)$

$fix^0_\sigma \to \lambda f : \sigma \to \sigma . \bot_\sigma$

lab-reduction: si usano queste due regole invece di quella del fix

Theorem: lab-reduction è confluent & terminating

<u>Lemma</u> $P \in lab(M)$ $\quad P \to_{lab} N$ normal $\Longrightarrow N \in anf(M)$

<u>Approximation theorem</u> : $[\![\Gamma \triangleright M : \sigma]\!]\eta = \bigvee [\![lab(M)]\!]\eta$ (prima)

$= \bigvee [\![anf(M)]\!]\eta$ (poi)

$\Leftrightarrow [\![P]\!]\eta = [\![R]\!]\eta$

<u>Corollario</u> $A_{PCF} \models P = R$ $\quad$ P program R result

ovvio per soundness $\Uparrow$ $\quad \Downarrow$ $\quad$ cioè A_{PCF} è computationally adequate per il PCF

$P \to R \Leftrightarrow \mathcal{E}_{PCF} \vdash P = R$ $\quad P \to\!\!\!\to R$

<u>Prova</u> Se $A_{PCF} \models P = R$, essendo i tipi osservabili nat e bool detti di un dominio piatto, deve esistere (per l'approx. theorem) $N \in anf(P)$ con $A_{PCF} \models N = R$. Anzi, addirittura $N \equiv R$ (sintacticamente!). Infatti se una forma normale chiusa C di tipo nat o bool contiene $\bot$, allora $[\![C]\!] = \bot$ (ad es. if mettere 3 e/o $\bot$ non è in forma normale!); ma $[\![N]\!] \neq \bot$ essendo $[\![N]\!] = [\![R]\!]$, R result; quindi N è una forma normale non approssimata; quindi N è un result; quindi $N \equiv R$ dato che $[\![N]\!] = [\![R]\!]$ e per il teorema di forma normale unica. Quindi abbiamo $P \to\!\!\!\to N \equiv R$.

Non-full abstractness del PCF : $=_{den} \subsetneq =_{op}$

Esiste in A_{PCF}

por : $B_\perp \to B_\perp \to B_\perp$

con

por true x = true
por x true = true
por false false = false

por

$x \backslash y$	T	F	$\perp$
T	T	T	T
F	T	F	$\perp$
$\perp$	T	$\perp$	$\perp$

$\leq$ per riga e per colonna monotone, quindi continua

<u>Teorema</u> : non esiste un termine PCF POR : bool → bool → bool

POR M N $\twoheadrightarrow$ true if M $\twoheadrightarrow$ true or N $\twoheadrightarrow$ true
false if M $\twoheadrightarrow$ false and N $\twoheadrightarrow$ false
no normal form otherwise

<u>Hint</u> si opera con riduzione lazy (deterministica) che è completa per bool/nat. Si dimostra che, dato un qualunque contesto con k holes $C[-,-,\dots,-]$, o si arriva a una forma normale che non dipende dagli argomenti, oppure esiste un buon precisato componente che deve essere valutato

dall'interprete. Ma in POR[ ]$_1$[ ]$_2$ questo non basta: infatti il risultato dipende dagli argomenti: se l'argomento è [ ]$_1$ allora

POR $diverge_{bool}$ true $\not\Rightarrow$ se è [ ]$_2$ allora POR true $diverge_{bool}$ $\not\Rightarrow$

con $diverge_\sigma = fix_\sigma\ (\lambda x{:}\sigma.\ x) : \sigma$. $\square$

In PCF possiamo però scrivere:

Is_POR? $\stackrel{def}{=}$ $\lambda P{:}bool \to bool \to bool.$
  if P true diverge then
    if P diverge true then
      if P false false then diverge else true
    else diverge
  else diverge : $(bool \to bool \to bool) \to bool$

Is_POR? M diverge per ogni M!

<u>Teorema</u>

Is_POR? $=_{op}$ $\lambda P{:}bool \to bool \to bool.\ diverge : (bool \to bool \to bool) \to bool$

<u>Proof</u>

Non esiste nessun contesto-programma che li distingue. Per assurdo, sia C[Is_POR?] e C[λP: diverge] uno diverge e uno no (se entrambi convergono a risultati diversi si può modificare C). Ma allora C[_] deve guardare il suo argomento, e quindi entrambi divergono. $\square$

Invece [[Is_POR?]] $\neq_{den}$ [[$\lambda P.bool \to bool \to bool.\ diverge_{bool}$]] in A_{PCF} dato che [[Is_POR?]] por = true mentre

[[$\lambda P.bool \to bool \to bool.\ diverge_{bool}$]] por = $\perp_{bool}$

PCF+por

Si aggiunge la costante

por : bool → bool → bool

con i seguenti assiomi di riduzione:

por true M → true
por M true → true
por false false → false

e $[\![por]\!] = por$

Si vede che:

- reduction for PCF+por è confluente
- lab-reduction è confluente e terminating

→ approximation theorem & adequacy result

↓

Teorema (Plotkin):

Per PCF+por abbiamo: $=_{den} = =_{op}$.

Polimorfismo

Una funzione polimorfa può essere applicata ad argomenti di "molti" tipi diversi.
Utile per programmare algoritmi che non dipendono (o dipendono solo in parte) dal tipo di certi argomenti.
Di solito "parametric" polymorphism (Strachey), invece di "ad hoc" polymorphism (ad es. con test esplicito sui tipi, come usando il case).

Esempio: composizione di funzioni

- $\text{compose}_{nat,nat,nat} \stackrel{def}{=} \lambda f{:}\underbrace{nat \to nat}_{\text{ripetuto solo qui!}}.\ \lambda g{:}\underbrace{nat \to nat}.\ \lambda x{:}\underbrace{nat}.\ f(g(x))$

versione polimorfa

- $\text{compose}_{r,s,t} \stackrel{def}{=} \lambda f{:}s \to t.\ \lambda g{:}r \to s.\ \lambda x{:}r.\ f(g(x))$

cioè esistono variabili che denotano tipi.
Si vuole fare astrazione/applicazione su queste variabili.

- $\text{compose} \stackrel{def}{=} \lambda r{:}T.\ \lambda s{:}T.\ \lambda t{:}T.\ \text{compose}_{rst}$

Un esempio ancora più semplice:

- $Id \stackrel{def}{=} \lambda t{:}T.\ \lambda x{:}t.\ x$

Qual'è però il tipo di compose?
↓
polymorphic types

Quale è il tipo di Id?

$Id : T \to t \to t$ non va bene, perché t non è legato!

Si scrive

$Id : \Pi t{:}T.\ t \to t$ oppure $\forall t{:}T.\ t \to t$

prodotto sul teoretico su un insieme infinito di indici

$I = \{1,2,3\}$ $\quad \prod_{i \in I} A_i = A_1 \times A_2 \times A_3$

$= \{ f : \{1,2,3\} \to A_1 \cup A_2 \cup A_3 \mid f(i) \in A_i \}$

Infatti

$$\prod_{i \in I} A_i = \{ f : I \to \bigcup_{i \in I} A_i \mid f(i) \in A_i \}$$

prodotto cartesiano esteso a un insieme infinito di indici

L'idea è che si può fare β-induzione sui tipi a compile time.

Abbiamo:

$$\mathrm{Id}\,\tau : (\Pi t : T.\, t \to t)(\tau : T) \xrightarrow{c} \mathrm{Id}\,\tau : \tau \to \tau$$

Resta ancora da capire cos'è T! Tre opzioni.

(A) T non contiene tipi polimorfi, ma solo costanti e costruttori tipo $\times$, $\to$, etc. $\to$ polimorfismo predicativo.

(B) T contiene tipi polimorfi $\to$ polim. impredicativo (definizione globale di T). Avete

System F (Girard 71-72)

Reynolds second-order λ-calculus (74)

(C) T un tipo, che può contenere se stesso: type:type

A: interessante minore, ma OK anche per modularità e type inference

B: Molto più complicato, molto importante e studiato

Esempio: Id può essere applicato al suo stesso tipo

$$(\lambda t:T.\lambda x:t.x)(\Pi t:T.t\to t) \twoheadrightarrow \lambda x:(\Pi t:T.t\to t).x$$

ottenendo una funzione che può essere applicata a Id!

$$(\lambda x:(\Pi t:T.t\to t).x)(\lambda t:T.\lambda x:t.x) \twoheadrightarrow x:(\lambda t:T.\lambda x:t.x).$$

Una specie di paradosso di Russell: non si possono trovare modelli set-theoretic del second-order λ-calculus.

C: Si tende a confondere tipi e valori (e quindi compile-time e run-time).

$(\lambda t:T.\lambda x:t.x)T$ è OK, dato che $T:T$

$\twoheadrightarrow \lambda x:T.x$

Quindi la funzione identità vale anche per i tipi.

$$(\lambda x:T.x)(t':T) \twoheadrightarrow t':T.$$

A, B: strongly normalizing con algoritmi efficienti

C: anche semplici $\Rightarrow$ compile type-checking indecidibile

facilmente inconsistenti (Martin-Löf 73, Girard)

Polimorfismo predicativo : $\lambda^{\to,\Pi}$

Due universi : U_1 e U_2 ← of large types : anche polimorfi.

di espressioni di tipo o types

U_1 ↑ universo di small types : tipi base e espressioni con $\to$ e variabili

Syntactic judgements : $\Gamma \triangleright A : B$ | A termo, B tipo 2 possib. / A tipo, B universo

$\Gamma = v_1 : A_1, \ldots, v_k : A_k$ $\quad$ A_i tipi o universi

Vanno definiti i contesti ben formati : $t : U_1, x : t$ è ok

$x : t, t : U_1$ non lo è!

- ϕ context (empty context)
- $\dfrac{\Gamma \text{ context}}{\Gamma, t : U_1 \text{ context}}$ t not in Γ $\quad$ (U_1 context) $\quad$ niente variabili di tipo U_2! Non servono! (ma ovviamente ci sono $\sigma : U_2$)
- $\dfrac{\Gamma \triangleright \sigma : U_i}{\Gamma, x : \sigma \text{ context}}$ x not in Γ $\quad$ (U_i type context)
- $\dfrac{\Gamma, x : A \text{ context}}{\Gamma, x : A \triangleright x : A}$ (var)
- $\dfrac{\Gamma \triangleright A : B \quad \Gamma, x : C \text{ context}}{\Gamma, x : C \triangleright A : B}$ (add var)

$\Sigma = (B, C)$ consiste di coppie $c : \tau$ $\quad$ τ chiuso $\quad$ $\tau \in U_1$

B ↑ base types $\quad$ c ↑ termo constant

- $\phi \triangleright b : U_1$ (cst U_1)
- $\dfrac{\Gamma \triangleright \tau : U_1 \quad \Gamma \triangleright \tau' : U_1}{\Gamma \triangleright \tau \to \tau' : U_1}$ ($\to U_1$)
- $\dfrac{\Gamma \triangleright \tau : U_1}{\Gamma \triangleright \tau : U_2}$ ($U_1 \subseteq U_2$)
- $\dfrac{\Gamma, t : U_1 \triangleright \sigma : U_2}{\Gamma \triangleright \Pi t : U_1 . \sigma : U_2}$ ($\Pi\, U_2$)

metanotazione :

x è variabile/termo

t è variabile/U_1

τ è tipo/U_1

σ è tipo/U_2

Pre-Termini di $\lambda^{\to,\Pi}$

$$M ::= x \mid \lambda x:\tau.M \mid MM \mid \lambda t:U_1.M \mid M\tau$$

- $\emptyset \triangleright c:\sigma$ (cst)
- $\dfrac{\Gamma, x:\tau \triangleright M:\tau' \quad \Gamma \triangleright \tau:U_1 \quad \Gamma \triangleright \tau':U_1}{\Gamma \triangleright (\lambda x:\tau.M):\tau \to \tau'}$ ($\to$ Intro)
- $\dfrac{\Gamma \triangleright M:\tau \to \tau' \quad \Gamma \triangleright N:\tau}{\Gamma \triangleright MN:\tau'}$ ($\to$ Elim)
- $\dfrac{\Gamma, t:U_1 \triangleright M:\sigma}{\Gamma \triangleright (\lambda t:U_1.M):\Pi t.\sigma}$ (Π Intro)
- $\dfrac{\Gamma \triangleright M:\Pi t:U_1.\sigma \quad \Gamma \triangleright \tau:U_1}{\Gamma \triangleright M\tau:[\tau/t]\sigma}$ (Π Elim)
- $\dfrac{\Gamma \triangleright M:\sigma_1 \quad \Gamma \triangleright \sigma_1 = \sigma_2 : U_i}{\Gamma \triangleright M:\sigma_2}$ (type eq) ← nel calcolo potremmo le uguaglianze di tipo sono provocate da α-conversioni
- $\Gamma \triangleright \Pi t:\sigma_1 = \Pi s:[s/t]\sigma_1 : U_2$ (α poli)

Second-order λ-calculus: aggiungere

$$\frac{\Gamma \triangleright \sigma:U_2}{\Gamma \triangleright \sigma:U_1} \quad (U_2 \subseteq U_1)$$

Type : type : aggiungere anche

$$\emptyset \triangleright U_1 : U_2 \quad (U_1 : U_2)$$

<u>Equational proof system</u> anche tutti gli assiomi di $\lambda\to$

$$\Gamma \rhd \lambda t.M = \lambda s.[s/t]M : \Pi t.\sigma \qquad (\alpha)_{\Pi}$$

$$\Gamma \rhd (\lambda t.M)\tau = [\tau/t]M : [\tau/t]\sigma \qquad (\beta)_{\Pi}$$

$:\Pi t.\sigma$

$$\Gamma \rhd \lambda t.Mt = M : \Pi t.\sigma \quad t \text{ not free in } M \qquad (\eta)_{\Pi}$$

$$\frac{\Gamma, t:\Pi, \rhd M = N : \sigma}{\Gamma \rhd \lambda t.M = \lambda t.N : \Pi t.\sigma} \qquad (\xi)_{\Pi}$$

$$\frac{\Gamma \rhd M = N : \Pi t.\sigma}{\Gamma \rhd M\tau = N\tau : [\tau/t]\sigma} \quad \Gamma \rhd \tau : \tau \qquad (\rho)_{\Pi}$$

Ancora <u>formulas-as-types correspondence</u>

logic formula ~ type

proof ~ lambda term

proof normalizat. ~ reduction

Curry-Howard homomorphism

$$\frac{\Gamma, t:U_1 \rhd M:\sigma}{\Gamma \rhd (\lambda t:U_1.M) : \Pi t.\sigma}$$

diventa

$$\frac{\Gamma, P:\text{proposizione} \vdash_M \phi}{\Gamma \vdash_{\lambda P.M} \forall P.\phi}$$

logica del 2° ordine con quantificazione solo sulle proposizioni

$\lambda P.M$ è una funzione che, data una prova Θ di P ritorna la prova $M\Theta$ di $[\Theta/P]\phi$

$$\frac{\Gamma \rhd M : \Pi t:U_1.\sigma \qquad \Gamma \rhd \tau : U_1}{\Gamma \rhd M\tau : [\tau/t]\sigma}$$

$$\frac{\Gamma \vdash_M \forall P.\text{propos.}\ \phi \qquad \Gamma \vdash \Theta : \text{formula senza quantif.}}{\Gamma \vdash_{M\Theta} [\Theta/P]\phi}$$

corrisponde a dare prova di tipo

Si passa all'ordine superiore con il polimorfismo impredicativo: non solo $\forall P$: propos.

Ma allora, a cosa corrispondono

- quantificazione universale primo ordine o generale

  $\Downarrow$

  tipi dipendenti !

- quantificazione esistenziale

  $\Downarrow$

  tipi astratti !

Infatti nel prodotto cartesiano generale

$$\prod_{i \in I} A_i = \{ f : I \to \bigcup_{i \in I} A_i \mid f(i) \in A_i \}$$

per il polimorfismo predicativo abbiamo assunto

$$I = U_1$$

cioè gli indici sono tipi, cioè insiemi.

Si potrebbe però prendere anche $I = \mathrm{Dom}\,\sigma$

$$\frac{\Gamma, \tau : U_1 \triangleright \sigma : U_2}{\Gamma \triangleright \Pi \tau : U_1 . \sigma : U_2} \quad \text{diventa} \quad \frac{\Gamma \triangleright \sigma : U_2 \quad \Gamma, x : \sigma \triangleright \sigma' : U_2}{\Gamma \triangleright \Pi x : \sigma . \sigma' : U_2}$$

cioè si astrae un tipo anche da una variabile qualsiasi

$\downarrow$

tipi che dipendono da un intero: es. array[n]

molto utili, ma in generale non verificabili staticamente.

Tipi astratti e quantificazione esistenziale

Facciamoci guidare da Curry-Howard

$$\frac{\Gamma \vdash M : [\Theta/P]\phi}{\Gamma \vdash \langle P = \Theta, M \rangle : \exists P.\phi}$$

con $\sum_{x \in A} B_x = \{ \langle a,b \rangle \mid a \in A, b \in B_a \}$

$\exists$ intro

$$\frac{\Gamma \rhd M : [\tau/t]\sigma}{\Gamma \rhd \langle t = \tau, M : \sigma \rangle : \exists t.\sigma}$$

$\Gamma \rhd \langle t = \tau, M : \sigma \rangle = \langle s = \tau, M : [s/t]\sigma \rangle : \exists t.\sigma$

s not free in σ

binder per σ → α-conversione

($\exists$ elim)

$$\frac{\Gamma \rhd N : \exists t.\sigma \qquad \Gamma, t : U_1, x : \sigma \rhd O : \rho}{\Gamma \rhd \text{abstype } t \text{ with } x : \sigma \text{ is } N \text{ in } O : \rho}$$

t not free in ρ

$\Gamma \rhd$ abstype t with $x : \sigma$ is N in O = abstype s with $y : [s/t]\sigma$ is N in $[y/x][s/t]O$

$\Gamma \rhd$ abstype t with $x : \sigma$ is $\langle t = \tau, M : \sigma \rangle$ in O $= [M/x][\tau/t]O : \rho$

similmente η

si fa la sostituzione di valore e di tipo contemporaneamente diverso in $\lambda^{\to, \Pi, \Sigma}$ più avanti

Un esempio di tipi astratti

abstype t with $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ is M in $N \stackrel{def}{=}$

abstype t with $y : \sigma_1 \times \ldots \times \sigma_n$ is M in $[Proj_1^n y, \ldots, Proj_n^n y / x_1, \ldots, x_n] N$

abstype cmplx with

create : real × real → cmplx
plus : cmplx × cmplx → cmplx
re : cmplx → real
im : cmplx → real

is

$$\langle t = real \times real, \langle C, P, R, I \rangle : (real \times real \to t) \times (t \times t \to t) \times (t \to real) \times (t \to real) \rangle$$

in

$$O = \lambda \langle x, y \rangle . (\lambda z : cmplx . re\ z)(create(x, y)) : real \times real \to re$$

implementazione: $\langle C', P', R', I' \rangle = M : [\sigma/t]\sigma = \langle C, P, R, I \rangle [real \times real / t]$

$C = \lambda p : real \times real . p$

$P = \lambda(z : real \times real, w : real \times real) . \langle Proj_1 z + Proj_1 w, Proj_2 z + Proj_2 w \rangle$

$R = \lambda z : real \times real . Proj_1 z$

$I = \lambda z : real \times real . Proj_2 z$

β conversione:

$= [P/plus][real \times real / t] O : real \times real \to real$ (No)

Moduli dei linguaggi di programmazione

Tipicamente: SML (MacQueen)

Informazioni su tipi, valori e strutture

Le signatures danno il "tipo" o l'interfaccia delle strutture

Così come per $\lambda^{\rightarrow}$ nella versione con tipi fissati o con tipi derivati via unificazione, qui si obbliga una struttura ad avere un'unica signature

Functors: sono funzioni da strutture a strutture

```
structure S =
  struct
    type t = int
    val x : t = 3
  end
```

```
signature SIG
  sig
    type t
    val x : t
  end
```

```
signature SIG'
  sig
    type t
    val x : int
  end
```

```
functor F (S : SIG) : SIG =
  struct
    type t = S.t * S.t
    val x : t = (S.x, S.x)
  end
```

Non esistono funtori "higher order"

Esempio di geometry program, p. 688-691

Calcolo predicativo con (generali) somme e prodotti

$\lambda^{\to}, \Pi, \Sigma$

È un'estensione di $\lambda^{\to}, \Pi$.

Sicuramente complicata, dato che una struttura può apparire in una type expression.

$$M ::= U_1 \mid U_2 \mid b \mid M \to M \mid \Pi x:M.M \mid \Sigma x:M.M$$
$$\mid x \mid c \mid \lambda x:M.M \mid MM$$
$$\mid \langle x:M = M, M:M \rangle \mid first(M) \mid second(M)$$

$\langle x:\sigma = M, N:\sigma \rangle$

x può essere usata sia in N sia in σ

diversamente dai tipi esistenziali, first e second permettono di accedere alle due componenti separatamente.

Tipificazione: regole AGGIUNTIVE rispetto al $\lambda^{\to}, \Pi$

$$\frac{\Gamma \rhd \sigma : U_2 \quad \Gamma, x:\sigma \rhd \sigma' : U_2}{\Gamma \rhd \Pi x:\sigma.\sigma' : U_2} \quad (\Pi\ U_2)$$

si può Π-astrarre anche con $x:\sigma'$!

$$\frac{\Gamma \rhd \sigma : U_2 \quad \Gamma, x:\sigma \rhd \sigma' : U_2 \quad \Gamma, x:\sigma \rhd M : \sigma'}{\Gamma \rhd \lambda x:\sigma.M : \Pi x:\sigma.\sigma'} \quad (\Pi\ \text{Intro})$$

$$\frac{\Gamma \rhd M : \Pi x:\sigma.\sigma' \quad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : [N/x]\sigma'} \quad (\Pi\ \text{Elim})$$

quando si scrive $x:\sigma$, può anche essere $t:U_2$!

$$\frac{\Gamma \triangleright \sigma : U_2 \qquad \Gamma, x : \sigma \triangleright \sigma' : U_2}{\Gamma \triangleright \Sigma x : \sigma . \sigma' : U_2} \quad (\Sigma\, U_2)$$

$$\frac{\Gamma \triangleright M : \sigma \qquad \Gamma \triangleright [M/x]N : [M/x]\sigma' \qquad \Gamma, x : \sigma \triangleright \sigma' : U_2}{\Gamma \triangleright \langle x : \sigma = M, N : \sigma' \rangle : \Sigma x : \sigma . \sigma'} \quad (\Sigma \text{ Intro})$$

$$\frac{\Gamma \triangleright M : \Sigma x : \sigma . \sigma'}{\Gamma \triangleright \text{first}(M) : \sigma} \quad (\Sigma \text{ Elim } 1)$$

$$\frac{\Gamma \triangleright M : \Sigma x : \sigma . \sigma'}{\Gamma \triangleright \text{second}(M) : [\text{first}(M)/x]\sigma'} \quad (\Sigma \text{ Elim } 2)$$

<u>Equational proof system</u>

In aggiunta agli assiomi di $\lambda^{\to, \Pi}$:

$$\Gamma \triangleright \text{first} \langle x : \sigma = M, N : \sigma' \rangle = M : \sigma \quad (\Sigma \text{ first})$$

$$\Gamma \triangleright \text{second} \langle x : \sigma = M, N : \sigma' \rangle = [M/x]N : [M/x]\sigma' \quad (\Sigma \text{ second})$$

$$\Gamma \triangleright \langle x : \sigma = \text{first } M, \text{second } M : \sigma' \rangle = M : \Sigma x : \sigma . \sigma' \quad (\Sigma \text{ surj. pair.})$$

Congruenze sui tipi

$$\frac{\Gamma \triangleright \sigma_1 = \sigma_1' : U_1 \qquad \Gamma \triangleright \sigma_2 = \sigma_2' : U_1}{\Gamma \triangleright \sigma_1 \to \sigma_2 = \sigma_1' \to \sigma_2' : U_1} \quad (\to \text{ Cong})$$

$$\frac{\Gamma \triangleright \sigma_1 = \sigma_1' : U_2 \qquad \Gamma, x : \sigma_1 \triangleright \sigma_2 = \sigma_2' : U_2}{\Gamma \triangleright \Pi x : \sigma_1 . \sigma_2 = \Pi x : \sigma_1' . \sigma_2' : U_2} \quad (\Pi \text{ Cong})$$

$$\frac{\Gamma \triangleright \sigma_1 = \sigma_1' : U_2 \qquad \Gamma, x : \sigma_1 \triangleright \sigma_2 = \sigma_2' : U_2}{\Gamma \triangleright \Sigma x : \sigma_1 . \sigma_2 = \Sigma x : \sigma_1' . \sigma_2' : U_2} \quad (\Sigma \text{ Cong})$$

Regole di congruenza per i termini

$$\frac{\Gamma, x:\sigma \rhd M = M' : \sigma' \qquad \Gamma \rhd \sigma = \sigma'' : U_i}{\Gamma \rhd \lambda x:\sigma.M = \lambda x:\sigma''.M' : \Pi x:\sigma.\sigma'}$$ (cong ξ) (per Π-intro)

$$\frac{\Gamma \rhd M = M' : \Pi x:\sigma.\sigma' \qquad \Gamma \rhd N = N' : \sigma}{\Gamma \rhd MN = M'N' : [N/x]\sigma'}$$ (cong ξ) (per Π-elim)

$$\frac{\Gamma \rhd M = M' : \sigma \quad \Gamma \rhd [M/x]N = [M'/x]N' : [M/x]\sigma' \quad \Gamma \rhd \sigma = \sigma'' : U_i \quad \Gamma \rhd [M/x]\sigma' = [M'/x]\sigma''' : U_i}{\Gamma \rhd \langle x:\sigma = M, N:\sigma' \rangle = \langle x:\sigma'' = M', N':\sigma''' \rangle : \Sigma x:\sigma.\sigma'}$$

(per Σ-intro)

$$\frac{\Gamma \rhd M = M' : \Sigma x:\sigma.\sigma'}{\Gamma \rhd first(M) = first(M') : \sigma}$$ (per Σ-Elim 1)

$$\frac{\Gamma \rhd M = M' : \Sigma x:\sigma.\sigma'}{\Gamma \rhd second(M) = second(M') : [first(M)/x]\sigma'}$$ (per Σ-Elim 2)

$$\frac{\Gamma \rhd M = N : \sigma \qquad \Gamma, x:A \text{ context}}{\Gamma, x:A \rhd M = N : \sigma}$$ (add var)

S = struct type t = int val x : t = 3 end $\equiv \langle t:U_1 = int, 3:t \rangle$

S.x $\equiv$ second (S)

sig type t val x : t end $\equiv \Sigma t:U_1.t$

functor F(S : SIG) : SIG = struct type t = S.t * S.t
val x : t = (S.x, S.x) end

$\equiv$

$\lambda S:(\Sigma t:U_1.t).\langle s:U_1 = first(S) \times first(S),$
$\langle second(S), second(S) \rangle : s \rangle : \Pi S:(\Sigma t:U_1.t).(\Sigma s:U_1.s$

Accettabili:

1. Polimorfismo impredicativo & dati astratti; oppure
2. Polimorfismo predicativo & somme generali

Se si mette insieme 2. con $U_1 = U_2$ si ottiene type:type con
- nontermination without explicit recursion
- undecidable type checking

1. e 2. sono strongly normalizing

Se si aggiunge type:type non lo è più

Anche type inference completeness dipende da $U_1 \neq U_2$.