Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Haskell: Inferenza dei Tipi
Algoritmo di Hindley/Milner

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **6**, 18 marzo 2021

Lezione 6a

Sistema Formale di Tipaggio

Sistema dei Tipi di Haskell

Cominciamo con il dare una versione astratta dei tipi di Haskell (**senza classi** e solo **costruttore freccia**).

Useremo la metavariabile τ (con eventuali pedici) per indicare tipi (o **monotipi**), le metavariabili $\alpha, \beta, \gamma, \ldots$ per indicare **variabili di tipo** e infine la variabile σ (con pedici) per indicare schemi di tipo (o **tipi polimorfi**).

$\tau ::=$	int | bool | …	(eventuali) tipi base
	α	variabile di tipo
	$\tau_1 \rightarrow \tau_2$	tipi freccia
$\sigma ::=$	τ	tipi sono schemi di tipo
	$\forall \alpha.\tau$	tipi polimorfi o politipi

Osservate che non è possibile annidare $\forall$ con $\rightarrow$: quindi il tipo **$(\forall \alpha.\ \alpha \rightarrow \alpha) \rightarrow (\forall \alpha.\ \alpha \rightarrow \alpha)$ non appartiene** a questa sintassi.

Se volessi tipi di questo genere dovrei aggiungere la regola sintattica $\sigma ::= \sigma_1 \rightarrow \sigma_2$, oppure rimuovere la distinzione tra monotipi e schemi di tipo.

Tipi, Sostituzioni, Unificazioni

Definiamo **sostituzione** θ una funzione finita da variabili di tipo in tipi. **Applicare una sostituzione** θ a un tipo τ, notazione $\tau\theta$ significa sostituire tutte le variabili $\alpha \in \text{dom}(\theta)$ in τ con $\theta(\alpha)$.

Esempio: Sia data la sostituzione $\theta=\{(\alpha, \beta\rightarrow\beta)\}$ e sia $\tau = \alpha\rightarrow\alpha$. Allora $\tau\theta=(\beta\rightarrow\beta)\rightarrow(\beta\rightarrow\beta)$.

Diciamo che due tipi τ_1 e τ_2 **unificano** se esiste una sostituzione θ tale che $\tau_1\theta = \tau_2\theta$.

Esempio $\tau_1 = \text{int}\rightarrow\beta$ e $\tau_2 = \alpha\rightarrow$ float unificano con la sostituzione $\{(\alpha, \text{int}), (\beta, \text{float})\}$. Ma anche $\alpha\rightarrow(\beta\rightarrow\beta)$ e $(\gamma\rightarrow\gamma)\rightarrow\delta$ unificano con la sostituzione $\{(\alpha, \gamma\rightarrow\gamma), (\delta, \beta\rightarrow\beta)\}$.

Diciamo che τ_1 **è più generale di** τ_2 (notazione $\tau_1 \leq \tau_2$) se esiste una sostituzione θ tale che $\tau_1\theta = \tau_2$

Esempio: Il tipo $\alpha\rightarrow\alpha$ è più generale del tipo $(\beta\rightarrow\beta)\rightarrow(\beta\rightarrow\beta)$. Viceversa per i tipi τ_1 e τ_2 dell'esempio precedente è falso sia che $\tau_1 \leq \tau_2$ sia che $\tau_2 \leq \tau_1$.

Frammento di Haskell (ML)

Ci preoccuperemo di dare un tipo a un frammento minimale di Haskell (questa è un'operazione usuale quando si fa teoria).

Tale mini-linguaggio è sostanzialmente un λ-calcolo tipato con una espressione particolare detta **let** (corrisponde alle definizioni di funzioni in Haskell, oppure al costrutto **`let`** e ha una parentele stretta con la clausola **`where`**).

M	::=	x	variabile
		$\lambda x.M$	astrazione
		let x=N **in** M	astrazione
		$(M\ N)$	applicazione

La semantica di **let** x = N **in** M è equivalente a $(\lambda x.\ M)\ N$ ma vedremo che ci sono importanti differenze nelle regole di tipaggio.

I **giudizi** del sistema dei tipi avranno la forma $\Gamma \vdash M : \sigma$ dove Γ è un ambiente e **dà un tipo alle variabili libere di** M (è una funzione finita da variabili in tipi) e σ è uno schema di tipo.

Sistema di Derivazione dei Tipi (1)

Vediamo le regole base per stabilire la correttezza di un tipo.

Quando conveniente, vedremo gli ambienti come **insiemi di associazioni** variabili/tipo nella forma $(x : \tau)$

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \quad (VAR)$$

I termini che consideriamo in Haskell sono **tutti chiusi**. Sono operatori di legame (o definizioni) i parametri a sinistra nelle equazioni ricorsive, i lambda ovviamente, le variabili dentro il pattern matching…

Tuttavia, il sistema di derivazione dei tipi è dato per **induzione strutturale** e per tipare $\lambda x.M$ devo saper tipare M dove la variabile x **occorre libera**.

Analogo discorso per le espressioni nella forma **let** x=M **in** N, dove x occorre libera in N.

Attenzione a dove occorre σ (politipo) e dove τ (monotipo)!

Sistema di Derivazione dei Tipi (2)

Applichiamo la seguente regola di tipo ogni qual volta applichiamo una funzione polimorfa a dei parametri.

$$\frac{\Gamma \vdash M : \sigma \quad \sigma \leq \sigma'}{\Gamma \vdash M : \sigma'} \quad (SPEC)$$

Esempi tipici possono essere $(\lambda x.x)\ 3$. Il tipo di $\lambda x.x$ va specializzato per essere applicato a un intero, sostituendo una variabile di tipo con il tipo int.

Ma posso specializzare i tipi anche con altri tipi polimorfi.

Ad esempio in una espressione come foldr $(\lambda x.x)\ 3$.

Sistema di Derivazione dei Tipi (2)

Applichiamo la seguente regola di tipo ogni volta che derivo un tipo per un termine e posso generalizzare **le variabili di tipo che non occorrono in tipi di variabili definite in λ più esterni**!

$$\frac{\Gamma \vdash M : \sigma \quad \alpha \notin free(\Gamma)}{\Gamma \vdash M : \forall \alpha . \sigma} \quad (GEN)$$

La side condition $\alpha \notin free(\Gamma)$ è fondamentale. Ad esempio:

$$\frac{\dfrac{x : \alpha \vdash x : \alpha}{x : \alpha \vdash x : \forall \alpha . \alpha}}{x : \alpha \vdash x : \beta}$$

che è chiaramente inconsistente. Oppure si riuscirebbe a tipare $\lambda x.x$ con $\forall\, \alpha\ \beta.\ \alpha \rightarrow \beta$: questo è il tipo di una funzione che prende un input di qualsiasi tipo e restituisce un valore di qualsiasi altro tipo… una funzione molto strana a ben vedere… E così via.

Sistema di Derivazione dei Tipi (3)

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \longrightarrow \tau_2} \quad (ABS)$$

$$\frac{\Gamma \vdash M : \tau_1 \longrightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2} \text{(APP)}$$

Osservazione: In queste due regole si applicano a monotipi. Se ho bisogno di generalizzare un politipo prima di una applicazione, preventivamente lo specializzo con la regola (SPEC).

Il fatto di limitarsi ai monotipi impedisce di tipare termini come $\lambda x.xx$: ovviamente nello scope di un λ per effetto della condizione vista prima, una variabile ***x* va tipata sempre con lo stesso tipo**!

Sistema di Derivazione dei Tipi (4)

$$\frac{\Gamma \vdash N : \sigma_1 \qquad \Gamma, \{x : \sigma_1\} \vdash M : \sigma_2}{\Gamma \vdash \textbf{\textit{let}}\ x = N\ \textbf{\textit{in}}\ M : \sigma_2} \quad (LET)$$

osservate ancora che il termine **let** $x = \lambda y.y$ **in** xx è tipabile mentre l'"equivalente" $(\lambda x.xx)(\lambda y.y)$ non lo sia.

Il termine **let** tuttavia **limita il polimorfismo** della variabile y alla struttura del tipo del termine $\lambda x.x$

Quindi sarebbe "pericoloso" tipare termini come $\lambda x.xx$, ma non lo è **se conosco già il tipo a cui verrà istanziato il tipo di x**, e questo è coerente con l'auto applicazione.

Derivazioni di tipo

Le derivazioni di tipo sono **alberi** i cui nodi interni sono istanze di regole. Alla radice ci sarà il giudizio da dimostrare, mentre sulle foglie ci saranno **i tipaggi delle variabili.**

Le regole sono syntax driven, ad eccezione di (GEN) e (SPEC) in cui modifico il tipo senza dipendere dalla sintassi del termine che tipo.

Esempio: (il linguaggio è un po' diverso con tipaggi espliciti sulle variabili astratte dai lambda)

$$\dfrac{\dfrac{\dfrac{x{:}\mathtt{Bool} \in x{:}\mathtt{Bool}}{x{:}\mathtt{Bool} \vdash x : \mathtt{Bool}}\,\text{T-Var}}{\vdash \lambda x{:}\mathtt{Bool}.x : \mathtt{Bool}\to\mathtt{Bool}}\,\text{T-Abs} \qquad \dfrac{}{\vdash \mathtt{true} : \mathtt{Bool}}\,\text{T-True}}{\vdash (\lambda x{:}\mathtt{Bool}.x)\ \mathtt{true} : \mathtt{Bool}}\,\text{T-App}$$

Lezione 6b

L'algoritmo di Inferenza di Tipo di Hindley-Milner

Sistema di Tipaggio e Algoritmo

Un termine M è tipabile se esiste un contesto Γ e un tipo τ tale che $\Gamma \vdash M : \tau$. Ma riusciamo a trovare Γ e τ?

Il sistema di derivazione dei tipi è molto utile per **specificare i termini tipabili**, ma pur essendo un calcolo formale non offre un vero e proprio algoritmo.

Infatti:

- devo "indovinare" i tipi delle variabili
- devo "indovinare" eventuali generalizzazioni e specializzazioni (regole non syntax-driven).

Vedremo nel seguito un algoritmo.

Algoritmo di Unificazione

Esiste un noto **algoritmo di unificazione** dovuto a Robinson che trova la sostituzione più generale (*most general unifier*, *mgu*) che unifica due termini.

L'algoritmo fu originariamente studiato per la soluzioni di vincoli/equaizioni in logica del prim'ordine: è usato in Prolog.

Lo vedremo brevemente, descritto sui tipi (senza $\forall$, in cui bisogna avere l'accortezza di **non far unificare** variabili legate).

Definizione: θ è l'unificatore più generale tra τ_1 e τ_2 se $\tau_1\theta = \tau_2\theta$ e per ogni altro unificatore θ' allora esiste θ'' tale che $\theta' = \theta \circ \theta''$.

$\mathcal{U}(\alpha, \tau) = \mathcal{U}(\tau, \alpha) = \{(\alpha, \tau)\}$ a patto che $\alpha \notin \mathit{free}(\tau)$ (**occurs check**)

$\mathcal{U}(\tau_1 \to \tau_2, \tau'_1 \to \tau'_2) = \theta_2 \circ \theta_1$ dove $\theta_1 = \mathcal{U}(\tau_1, \tau'_1)$ e $\theta_2 = \mathcal{U}(\tau_2\theta_1, \tau'_2\theta_1)$

L'algoritmo **fallisce in tutti gli altri casi** (ad esempio quando deve unificare due tipi basi diversi, o due tipi con un numero diverso di frecce, ma non ci sono variabili da istanziare per far combaciare il numero di frecce etc.)

Algoritmo $\mathcal{W}$ di Type Inference (1)

L'algoritmo $\mathcal{W}$ di Hindley-Milner trova il **tipo più generale** di un termine, usando l'algoritmo $\mathcal{U}$ come sub-routine.

L'idea è di assegnare alle variabili il tipo più generale possibile (una variabile di tipo fresca) e poi accumulare i vincoli (generati dalle unificazioni) imposte dai contesti in cui appaiono le variabili (ad esempio, dentro le applicazioni).

L'algoritmo $\mathcal{W}$ prende in input un termine M e un ambiente di tipo Γ e calcola simultaneamente un tipo τ e una sostituzione θ.

Nella descrizione di $\mathcal{W}$ indicheremo con:

- $\Gamma\theta$ l'ambiente $\{(x, \tau\theta) \mid (x, \tau) \in \Gamma\}$
- $\Gamma \setminus x$ l'ambiente in cui ho cancellato le associazioni per x.
- $\mathcal{J}(\sigma)$ è l'**istanziazione** delle variabili quantificate nello schema di tipo σ con variabili di tipo fresche
- $\mathcal{G}(\Gamma, \tau)$ è la generalizzazione di τ, cioè lo schema di tipo: $\forall \alpha_1 \ldots \alpha_n.\tau$ dove $\{\alpha_1, \ldots, \alpha_n\} = free(\Gamma) \setminus free(\tau)$

Algoritmo $\mathcal{W}$ di Type Inference (2)

$\mathcal{W}(\Gamma, x) = (\varnothing, x : \mathcal{J}(\sigma))$ **dove** $(x, \sigma) \in \Gamma$

$\mathcal{W}(\Gamma, M\ N) = (\theta_3 \circ\ \theta_2 \circ\ \theta_1, \beta\theta_3)$
dove $\mathcal{W}(\Gamma, M) = (\theta_1, \tau_1)$
$\mathcal{W}(\Gamma\theta_1\ , N)\ = (\theta_2, \tau_2)$
$\theta_3 = \mathcal{U}(\tau_1, \tau_2 \rightarrow \beta)$ con β variabile di tipo fresca

$\mathcal{W}(\Gamma, \lambda x.M) = (\theta_1, \beta\theta_1 \rightarrow \tau_1)$
dove $\mathcal{W}(\Gamma \setminus x \cup \{(x : \beta)\}, M) = (\theta_1, \tau_1)$ β fresca

$\mathcal{W}(\Gamma,$ **let** $x = M$ **in** $N) = (\theta_2 \circ\ \theta_1, \tau_2)$
dove $\mathcal{W}(\Gamma, M) = (\theta_1, \tau_1)$
$\mathcal{W}(\Gamma \setminus x \cup \{(x : \mathcal{G}(\Gamma\theta_1, \tau_1)\}, N) = (\theta_2, \tau_2)$

$\theta_2 \circ\ \theta_1 = \{(\alpha : \tau\ \theta_2)\ |\ (\alpha : \tau) \in \theta_1\}$

Correttezza e Completezza di $\mathcal{W}$

Teorema: Per ogni termine M e ambiente Γ ho che:

- (CORRETTEZZA) Se $\mathcal{W}(\Gamma, M)$ termina con successo restituendo (θ, τ) allora $\Gamma\theta \vdash M : \tau$
- (COMPLETEZZA) per ogni σ tale che $\Gamma \vdash M : \sigma$, allora $\mathcal{W}(\Gamma, M)$ termina con esito positivo restituento (θ, σ') con σ' più generale di σ.

```
\x y -> x y = \x (\y-> x y)**
  W((x:a),(\y-> x y))*
  W([(x:a), (y:b)], xy)
    X: W([(x:a), (y:b)], x) = <{},a>
    Y: W([(x:a), (y:b)], y) = <{},b>
    U(a,b->c) = <{(a, b->c)},c>
  * <{(a, b->c)},b->c>
  ** <{(a, b->c)},(b->c)->b->c>

\x y. x (x y)
```

Lezione 6

That's all Folks…

Grazie per l'attenzione…

…Domande?