Sistemi di tipo

Sistemi di tipo e analisi statica

- le tecniche di analisi statica approssimano (a tempo di compilazione) il comportamento di un programma (a tempo di esecuzione) con l'obiettivo di scoprire eventuali errori e dimostrare particolari proprietà
  - l'analisi dei tipi è una delle tecniche più importanti
    - classifica le frasi di un programma in base al tipo di valori che calcolano e garantisce che ogni operazione sia applicata a dati corretti
  - i sistemi di tipo sono uno dei modi per specificare i tipi calcolati da un linguaggio

Sistemi di tipo

- un sistema di tipo è un *proof system* (logica)
  - i teoremi che si dimostrano riguardano i tipi di un programma
- un sistema di tipo si specifica attraverso i seguenti passi
  - definizione della sintassi dei tipi
  - definizione dell'ambiente dei tipi
  - definizione dei *judgments*
  - definizione delle regole di inferenza
  - dimostrazione della *soundness*

Sintassi dei tipi

- tipi di base
- costruttori utilizzabili per costruire nuovi tipi per composizione di quelli esistenti

- specificati da una grammatica libera il cui linguaggio corrisponde all'insieme di tutti i possibili tipi `TYPE`

- un esempio
  - $t ::= k \mid t_1 \rightarrow t_2 \mid t_1 * t_2$
  - $k \in$ {integer, boolean}

Ambiente dei tipi

- tutti i linguaggi di programmazione permettono di denotare entità varie con dei nomi (variabili?)
- l'ambiente contiene le associazioni tra nomi e valori denotati
- anche il tipo di una frase dipende da cosa è denotato dalle variabili che vi occorrono
- ambiente dei tipi
  - associazioni tra nomi e valori denotati (tipi)
  - rappresentato come una lista di coppie definita ricorsivamente

    $\Gamma ::= \Phi \mid \Gamma, x:t$

    - Φ ambiente vuoto
    - $\Gamma, x:t$ ambiente Γ esteso con una nuova associazione tra x e t
    - $\mathrm{dom}(\Gamma) = \{ x \mid \exists t \in \mathrm{TYPE}\ x:t \in \Gamma \}$
  - se le variabili in Γ sono distinte, Γ può essere visto come una funzione parziale
    - $\Gamma(x)$ denota il tipo di x

Judgments

- le formule ben formate della logica
  - affermazioni formali sul tipo di una frase
- tipico judgment

  `Γ |- A`

  - Γ "implica" A
  - A asserzione le cui variabili libere sono legate in Γ
  - relazione fra una frase e un tipo

    `Γ |- P: t`

  - relazione fra due tipi (se ci sono sottotipi)

    $\Gamma \vdash t_1 <: t_2$

Regole di inferenza

- come certi judgments derivano da altri judgments

premesse $\Gamma_1 \;|\!-\; A_1 \quad ..... \quad \Gamma_n \;|\!-\; A_n$

(nome della regola) --

conclusione $\Gamma \;|\!-\; A$

- ci sono sempre alcune regola senza premesse (*assiomi*, judgments validi)
- una derivazione (*dimostrazione*) è un albero di judgments, in cui le foglie sono assiomi ed ogni nodo interno è la conclusione di una regola che ha come premesse i nodi figli

assiomi $\Gamma_{11} \;|\!-\; A_{11} \quad ..... \quad \Gamma_{1n} \;|\!-\; A_{1n}$

... ...

(regola n) ----------------------------------

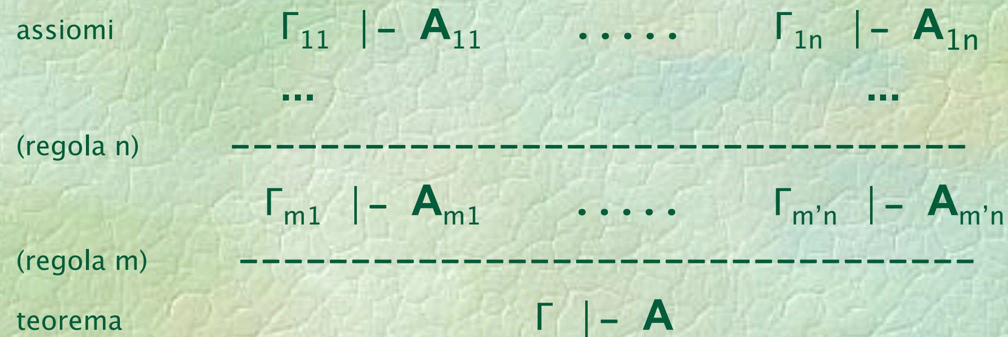
$\Gamma_{m1} \;|\!-\; A_{m1} \quad ..... \quad \Gamma_{m'n} \;|\!-\; A_{m'n}$

(regola m) ----------------------------------

teorema $\Gamma \;|\!-\; A$

Judgments validi

- i judgments validi coincidono con i teoremi
  - radici di derivazioni



- derivare un typing judgment della forma `Γ |- P: t` significa dimostrare che la frase `P` ha tipo `t`

Teorema di type soundness

- obiettivo di un sistema di tipo è garantire che non si verifichino certi errori a run time
- due famiglie di errori
  - errori trapped
    - catturati dal supporto a run time (arresto e generazione di eccezioni)
  - errori non trapped
    - non essendo rivelati quando si verificano, possono causare ogni genere di problema
- un buon sistema di tipo dovrebbe assicurare l'assenza di tutti gli errori non trapped e di buona parte di quelli trapped, scartando a priori tutti quei programmi che potenzialmente li potrebbero generare
- naturalmente il sistema di tipo deve essere dimostrato "corretto" (sound) rispetto alla semantica
- se `Γ |- P: t` ed il costrutto `P` ha come semantica il valore `V`, allora il tipo di `V` deve essere `t`

Proprietà di un sistema di tipo

- *soundness del sistema di tipo rispetto alla semantica*
- decidibilità
  - deve esistere un algoritmo (type checking) in grado di verificare se una frase `P` è ben tipata o meno
    - trovare una derivazione con radice `Γ |- P: t` per qualche `Γ` e `t`
  - nei linguaggi tipati staticamente, l'algoritmo può essere sviluppato con varie tecniche e fa parte del compilatore
  - deve soddisfare due importanti proprietà
    - *correttezza: se il tipo calcolato dall'algoritmo per* `P` *è* `t`*, il judgment* `Γ |- P: t` *è valido*
    - *completezza: se il judgment* `Γ |- P: t` *è valido, il tipo calcolato dall'algoritmo per* `P` *è* `t`
- trasparenza
  - deve essere immediato per l'utente del linguaggio prevedere se il programma supererà il controllo dei tipi
    - deve essere facile capire perché un programma viene rifiutato
- l'esistenza di un *algoritmo di type inference* è altra cosa ancora

Alcuni importanti sistemi di tipo

- l'articolo di Cardelli tratta quasi esclusivamente i sistemi di tipo utili per ragionare su linguaggi imperativi e orientati ad oggetti
  - dove, quando va bene, esiste un algoritmo di verifica dei tipi
- noi siamo interessati ai sistemi di tipo della programmazione funzionale (λ-calcolo)
  - ce ne sono molti
  - sono gli unici ad essere completamente formalizzati (relazione con la semantica, relazione con l'algoritmo)
  - si possono confrontare più facilmente con i risultati della interpretazione astratta
- e agli algoritmi di inferenza di tipi
- vedremo tre sistemi, in ordine crescente di complessità
  - i tipi monomorfi di Church-Curry
  - i monotipi con variabili di Hindley
  - i tipi polimorfi con ricorsione monomorfa (Damas-Milner, ML)
    - la sua generalizzazione, con ricorsione polimorfa (Damas-Milner-Mycroft)
      -

Il linguaggio

- λ-calcolo non tipato
  - `x, f, …` ∈ X: variabili
  - `e,` e_1`, …` ∈E: espressioni
- e::= int n |id x | e_1 + e_2 | λx.e |
  if e_1 then e_2 else e_3 | $e_1(e_2)$ | μf. λx. e
- essenzialmente il nostro linguaggio didattico (senza `boolean`, con poche operazioni e senza `let`)
  - il valore della guardia del condizionale è confrontato con 0
  - in linea di principio un'espressione come let i be e_1 in e_2 può essere vista come "zucchero sintattico" per l'espressione (λi.e_2)e_1
    - vedremo che non è vero (polimorfismo!)

Monotipi di Church-Curry 1

- i tipi (*monotipi*)
  - $m ::= int \mid m_1 \rightarrow m_2$
- il sistema **C** (Church-Curry)

```
H |-c int n : int          H |-c id x : H(x)

H |-c e1 : int   H |-c e2 : int      H [x ← m1] |-c e : m2
----------------------------------   ---------------------
    H |-c e1 + e2 : int               H |-c λx.e : m1 → m2

H |-c e1 : int   H |-c e2 : m   H |-c e3 : m      H |-c e1 :m1 → m2   H |-c e2 :m1
--------------------------------------------      ----------------------------------
     H |-c if e1 then e2 else e3 : m                     H |-c e1(e2) : m2

                    H [f ← m] |-c λx. e : m
                    ---------------------
                    H |-c μf. λx. e : m
```

Monotipi di Church-Curry 2

- secondo il sistema C il tipo di un'espressione `e` è un qualunque monotipo m_i per cui il judgment `H |-`$_c$ `e : `m_i è valido
  - un espressione può avere un numero qualunque (anche infinito) di tipi
  - per esempio, l'espressione `λx.x` ha i seguenti monotipi

    `int→int, (int→int) → (int→int), ...`

- l'algoritmo di verifica dei tipi è semplicemente l'uso delle regole di C (dalle conclusioni alle ipotesi, top-down), per ridurre il judgment da provare agli assiomi
- è impossibile usare direttamente C per l'inferenza di tipi
  - a parte il fatto che non esiste un unico tipo, come si inventano i monotipi m_1 ed `m` nelle regole per astrazione e ricorsione?

$$\frac{H\ [x \leftarrow m_1]\ \vdash_c e : m_2}{H\ \vdash_c \lambda x.e : m_1 \rightarrow m_2} \qquad \frac{H\ [f \leftarrow m]\ \vdash_c \lambda x.\ e : m}{H\ \vdash_c \mu f.\ \lambda x.\ e : m}$$

Politipi di Church-Curry e monotipi di Hindley

- nei politipi di Church-Curry, un tipo è un insieme di monotipi di Church-Curry
  - il sistema di tipo è relativamente complesso
  - ...e poco utile

- sono più interessanti dal punto di vista pratico generalizzazioni più semplici del sistema C
  - prima di tutto i monotipi di Hindley, il cui sistema H ci avvicina al sistema di tipo di ML
  - H è dotato di un algoritmo di inferenza che determina il tipo principale per il sistema C
    - una rappresentazione esatta di tutti i monotipi
  - usa la cosiddetta astrazione di Herbrand, cioè i tipi con variabili
  - H coincide con il sistema di tipo di ML ed il suo algoritmo di inferenza coincide con quello di Damas-Milner nel frammento senza `let`

Monotipi di Hindley

- i tipi (*monotipi con variabili*)

  τ := int | 'a | $\tau_1 \rightarrow \tau_2$

- i tipi sono termini, parzialmente ordinati con la relazione di istanziazione
  - `mgu` calcola l'unificatore più generale di un insieme di tipi
  - `gci` è la greatest common instance di un insieme di tipi (applicazione del loro mgu)
  - `mgu` e `gci` si estendono a qualunque tipo di struttura che contenga tipi (per esempio, l'ambiente dei tipi)
  -
- un monotipo con variabili, per esempio `'a` $\rightarrow$ `'a`, viene di solito (Java, ML) considerato un tipo polimorfo
- il vero *polimorfismo* nei sistemi di tipo è cosa molto più complessa

Sul polimorfismo 1

- i costrutti che sono coinvolti nelle varie forme di polimorfismo sono quelli che richiedono la valutazione di una espressione in un ambiente modificato con una nuova associazione, in particolare
  - `let x be` e_1 `in` e_2 valutazione di e_2 con una nuova associazione per `x`
  - `λx.e` valutazione di `e` con una nuova associazione per `x`
  - `μx. λy. e` valutazione di `λy. e` con una nuova associazione per `x`
- supponiamo che, nei tre casi, il nuovo nome `x` denoti il tipo τ nell'ambiente dei tipi e che τ contenga variabili di tipo
- polimorfismo vuol dire che diverse occorrenze di `x` nell'espressione possono usare diverse istanze di τ
- polimorfismo `let`
  - si applica solo ad associazioni create con il `let` (occorrenze di `x` in e_2)
- ricorsione polimorfa
  - si applica alle chiamate ricorsive di `x` in `λy.e`
- astrazione polimorfa
  - si applica alle occorrenze del parametro formale `x` in `e`

Sul polimorfismo 2

- `let x be` e_1 `in` e_2 valutazione di e_2 con una nuova associazione per `x`
- `λx.e` valutazione di `e` con una nuova associazione per `x`
- `μx. λy. e` valutazione di `λy. e` con una nuova associazione per `x`
- polimorfismo `let`
  - si applica solo ad associazioni create con il `let` (occorrenze di `x` in e_2)
- ricorsione polimorfa
  - si applica alle chiamate ricorsive di `x` in `λy. e`
- astrazione polimorfa
  - si applica alle occorrenze del parametro formale `x` in `e`
- ML permette soltanto il polimorfismo `let`
  - il `let` è stato introdotto proprio per avere una forma (semplice) di polimorfismo
  - `let i be` e_1 `in` $e_2 \neq (\lambda i.e_2)e_1$
    - provate in ML le espressioni

```
let x be (λy.y) in x x
(λx. x x) (λy.y)
```

- i monotipi di Hindley non permettono nessun tipo di polimorfismo

Torniamo ai monotipi di Hindley

- i tipi (*monotipi con variabili*)

  τ := int |'a | $\tau_1 \rightarrow \tau_2$

- il sistema H (Hindley): le regole sono le stesse di C, cambiano solo i tipi!

$$H \;|\!-_H \text{int } n : \text{int} \qquad H \;|\!-_H \text{id } x : H(x)$$

$$\frac{H \;|\!-_H e_1 : \text{int} \quad H \;|\!-_H e_2 : \text{int}}{H \;|\!-_H e_1 + e_2 : \text{int}} \qquad \frac{H\,[x \leftarrow \tau_1] \;|\!-_H e : \tau_2}{H \;|\!-_H \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \;|\!-_H e_1 : \text{int} \quad H \;|\!-_H e_2 : \tau \quad H \;|\!-_H e_3 : \tau}{H \;|\!-_H \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \qquad \frac{H \;|\!-_H e_1 : \tau_1 \rightarrow \tau_2 \quad H \;|\!-_H e_2 : \tau_1}{H \;|\!-_H e_1(e_2) : \tau_2}$$

$$\frac{H\,[f \leftarrow \tau_1 \rightarrow \tau_2] \;|\!-_H \lambda x.\ e : \tau_1 \rightarrow \tau_2}{H \;|\!-_H \mu f.\ \lambda x.\ e : \tau_1 \rightarrow \tau_2}$$

Un esempio di verifica di tipo in H

τ := int |'a | $\tau_1 \rightarrow \tau_2$

H $|-_H$ int n : int   H $|-_H$ id x : H(x)

```
H |-H e1 : int    H |-H e2 : int
--------------------------------
      H |-H e1 + e2 : int
```

```
H [x ← τ1] |-H e : τ2
---------------------
H |-H λx.e : τ1 → τ2
```

```
H |-H e1 : int    H |-H e2 : τ    H |-H e3 : τ
----------------------------------------------
      H |-H if e1 then e2 else e3 : τ
```

```
H |-H e1 : τ1 → τ2   H |-H e2 : τ1
----------------------------------
        H |-H e1(e2) : τ2
```

```
H [f ← τ1 → τ2] |-H λx. e : τ1 → τ2
------------------------------------
     H |-H μf. λx. e : τ1 → τ2
```

```
H [x ← 'a] |-H id x :'a
-----------------------
H [x ← 'a] |-H id x :'a
-----------------------
H |-H λx.id x : 'a → 'a
```

Dalla verifica all'inferenza di tipo in H

- H è dotato di un algoritmo di inferenza che determina il tipo principale per il sistema C
  - una rappresentazione esatta di tutti i monotipi
- l'algoritmo di inferenza di Hindley, data una espressione restituisce un *typing*, cioè una coppia ambiente, tipo

$$e \Rightarrow_{\mathtt{H}} <\mathtt{H}, \tau>$$

- l'algoritmo utilizza, oltre alle funzioni `gci` e `mgu`, l'ambiente di tipi più generale $\mathcal{H}$ in cui ogni nome è legato ad una variabile di tipo distinta.
- ne facciamo vedere una descrizione in stile "semantica operazionale", per permettere di confrontare le regole del sistema di tipo con quelle dell'algoritmo di inferenza

L'algoritmo di inferenza di Hindley 1

$$\texttt{int n} \Rightarrow_H \langle \mathcal{H}, \texttt{int} \rangle$$

$$\texttt{id x} \Rightarrow_H \langle \mathcal{H}, \mathcal{H}(\texttt{x}) \rangle$$

$$\frac{e_1 \Rightarrow_H \langle H_1, \tau_1 \rangle \qquad e_2 \Rightarrow_H \langle H_2, \tau_2 \rangle}{e_1 + e_2 \Rightarrow_H \texttt{gci}\{\langle \mathcal{H}, \texttt{int} \rangle, \langle H1, \tau 1 \rangle, \langle H2, \tau 2 \rangle\}}$$

$$\frac{e_1 \Rightarrow_H \langle H_1, \texttt{int} \rangle \quad e_2 \Rightarrow_H \langle H_2, \tau_2 \rangle \quad e_3 \Rightarrow_H \langle H_3, \tau_3 \rangle}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow_H \texttt{gci}\{\langle H_1, \texttt{'a} \rangle, \langle H_2, \tau_2 \rangle, \langle H_3, \tau_3 \rangle\}}$$

- sottoespressioni diverse si comunicano eventuali istanziazioni di variabili di tipo, attraverso gli ambienti e le operazioni di `gci`

L'algoritmo di inferenza di Hindley 2

$$\frac{e_1 \Rightarrow_H \langle H_1, \tau_1\rangle \qquad e_2 \Rightarrow_H \langle H_2, \tau_2\rangle \qquad gci\{\langle H_1, \tau_1\rangle, \langle H_2, \tau_2 \rightarrow 'a\rangle\} = \langle H, \tau_2 \rightarrow \tau\rangle}{e_1\ e_2 \Rightarrow_H \langle H, \tau\rangle}$$

$$\frac{e \Rightarrow_H \langle H, \tau\rangle}{\lambda x.e \Rightarrow_H \langle H[x \leftarrow 'a], H(x) \rightarrow \tau\rangle}$$

$$\frac{\lambda x.e \Rightarrow_H \langle H, \tau\rangle \qquad \sigma = mgu\{'a \rightarrow 'b, H(f), \tau\}}{\mu f.\lambda x.e \Rightarrow_H \langle \sigma\ H[f \leftarrow 'c], \sigma\tau\rangle}$$

- non banali da capire e soprattutto “da inventare” a partire dal semplice ed intuitivo sistema di tipo
- si noti che la regola per le funzioni ricorsive non prevede alcun calcolo di punto fisso

Un esempio di inferenza in H

$$\texttt{id x} \Rightarrow_H \quad \langle \mathcal{H}, \mathcal{H}(\texttt{x}) \rangle$$

$$\texttt{e} \Rightarrow_H \langle H, \tau \rangle$$

$$\lambda\texttt{x.e} \Rightarrow_H \langle H[\texttt{x} \leftarrow \texttt{'a}], H(\texttt{x}) \rightarrow \tau \rangle$$

$\lambda\texttt{x.id x} =$ valuto $\texttt{id x}$ $H = \mathcal{H}$, $\tau = \texttt{'x}$

$= \langle H[\texttt{x} \leftarrow \texttt{'a}], H(\texttt{x}) \rightarrow \tau \rangle =$

$= \langle \mathcal{H}[\texttt{x} \leftarrow \texttt{'a}], \texttt{'x} \rightarrow \texttt{'x} \rangle$

- il tipo inferito contiene la variabile di tipo inizialmente denotata da **x** in $\mathcal{H}$
- nell'ambiente restituito, **x** denota una variabile di tipo fresca

- l'uso dell'ambiente per modellare la "trasmissione all'indietro" di legami per le variabili porta ad un algoritmo poco trasparente
  - non ragioniamo così quando "immaginiamo" il tipo di una espressione ML
- vedremo una formulazione molto più intuitiva nel contesto dell'interpretazione astratta

Verso sistemi di tipo polimorfi

- i soliti monotipi con variabili

  τ := int |'a | $\tau_1 \rightarrow \tau_2$

- negli schemi di tipo di Milner (*politipi parametrici*) si distingue tra variabili libere e variabili legate (quantificate universalmente)

  π := τ |∀ 'b$_1$...,'b$_k$.τ, dove 'b$_1$...,'b$_k$ sono tutte le variabili libere in τ (ftv(τ), free type variables, variabili di tipo libere)

- l'ambiente dei tipi contiene politipi parametrici
- quando si vuole essere polimorfi, il monotipo deve essere inserito nell'ambiente come politipo parametrico, applicando l'operazione gen(τ) = ∀ 'b$_1$...,'b$_k$.τ, con {'b$_1$...,'b$_k$} = ftv(τ)
- se π = ∀ 'b$_1$...,'b$_k$.τ, l'operazione inst(π), restituisce una qualunque istanza del monotipo τ ottenuta rimpiazzando le variabili 'b$_1$...,'b$_k$ con monotipi qualunque contenenti variabili di tipo fresche
-
- aggiungiamo al linguaggio il costrutto `let x = e1 in e2`
-

Il sistema di tipo DM (Damas-Milner, ML)

- il sistema DM: cambia l'ambiente (politipi parametrici), cambia la regola per `id x`, nuova regola per il `let`

$$H \mid\!-_{DM} \texttt{int } n : \texttt{int}$$

$$\frac{\tau \in inst(H(x))}{H \mid\!-_{DM} \texttt{id } x : \tau}$$

$$\frac{H \mid\!-_{DM} e_1 : \texttt{int} \qquad H \mid\!-_{DM} e_2 : \texttt{int}}{H \mid\!-_{DM} e_1 + e_2 : \texttt{int}}$$

$$\frac{H[x \leftarrow \tau_1] \mid\!-_{DM} e : \tau_2}{H \mid\!-_{DM} \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \mid\!-_{DM} e_1 : \texttt{int} \qquad H \mid\!-_{DM} e_2 : \tau \qquad H \mid\!-_{DM} e_3 : \tau}{H \mid\!-_{DM} \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

$$\frac{H \mid\!-_{DM} e_1 : \tau_1 \rightarrow \tau_2 \qquad H \mid\!-_{DM} e_2 : \tau_1}{H \mid\!-_{DM} e_1(e_2) : \tau_2}$$

$$\frac{H[f \leftarrow \tau_1 \rightarrow \tau_2] \mid\!-_{DM} \lambda x.\ e : \tau_1 \rightarrow \tau_2}{H \mid\!-_{DM} \mu f.\ \lambda x.\ e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \mid\!-_{DM} e_1 : \tau_1 \qquad H[x \leftarrow gen(\tau_1)] \mid\!-_{DM} e2 : \tau}{H \mid\!-_{DM} \texttt{let } x = e1 \texttt{ in } e2 : \tau}$$

Polimorfismo in DM

- la regola per `id x`, assegna un qualunque tipo ottenuto istanziando le variabili generiche contenute nel politipo denotato da `x`
- il `let` è l'unico costrutto che inserisce nell'ambiente tipi generici
  - solo polimorfismo `let`

$$\frac{\tau \in inst(H(x))}{H \mid\!-_{DM}\ id\ x : \tau}$$

$$\frac{H\ [x \leftarrow \tau_1]\ \mid\!-_{DM}\ e : \tau_2}{H \mid\!-_{DM}\ \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H\ [f \leftarrow \tau_1 \rightarrow \tau_2]\ \mid\!-_{DM}\ \lambda x.\ e : \tau_1 \rightarrow \tau_2}{H \mid\!-_{DM}\ \mu f.\ \lambda x.\ e : \tau_1 \rightarrow \tau_2}$$

$$\frac{H \mid\!-_{DM}\ e_1 : \tau_1 \quad H\ [x \leftarrow gen(\tau_1)]\ \mid\!-_{DM}\ e2 : \tau}{H \mid\!-_{DM}\ let\ x = e1\ in\ e2 : \tau}$$

Una verifica di tipo in DM

- il sistema DM: cambia l'ambiente (politipi parametrici), cambia la regola per `id x`, nuova regola per il `let`

$$\frac{\tau \in \mathtt{inst(H(x))}}{\mathtt{H}\ |\!-_{DM}\ \mathtt{id\ x} : \tau} \qquad \frac{\mathtt{H}\ |\!-_{DM}\ e_1 : \tau_1 \quad \mathtt{H\ [x} \leftarrow \mathtt{gen}(\tau_1)\mathtt{]}\ |\!-_{DM}\ \mathtt{e2} : \tau}{\mathtt{H}\ |\!-_{DM}\ \mathtt{let\ x = e1\ in\ e2} : \tau}$$

$$\frac{\mathtt{H\ [x} \leftarrow \tau_1\mathtt{]}\ |\!-_{DM}\ \mathtt{e} : \tau_2}{\mathtt{H}\ |\!-_{DM}\ \lambda \mathtt{x.e} : \tau_1 \to \tau_2} \qquad \frac{\mathtt{H}\ |\!-_{DM}\ e_1 : \tau_1 \to \tau_2 \quad \mathtt{H}\ |\!-_{DM}\ e_2 : \tau_1}{\mathtt{H}\ |\!-_{DM}\ e_1(e_2) : \tau_2}$$

$H[f \leftarrow gen('c \to 'c)]\ |\!-_{DM}\ id\ f : ('e \to 'e) \to ('a \to 'a)$ $\quad$ $H[f \leftarrow gen('c \to 'c)]\ |\!-_{DM}\ id\ f : 'e \to 'e$

---- ------------------------------------ $\quad$ ------------------------------------

$H\ [x \leftarrow 'c]\ |\!-_{DM}\ id\ x : 'd$ $\quad$ $H\ [f \leftarrow gen('c \to 'c)]\ |\!-_{DM}\ id\ f(id\ f) : 'a \to 'a$

------------------------- $\quad$ --

$H\ |\!-_{DM}\ \lambda x.id\ x : 'c \to 'd$ $\quad$ $H\ [f \leftarrow gen('c \to 'd)]\ |\!-_{DM}\ id\ f(id\ f) : 'a \to 'a$

----------------------- $\quad$ --

$H\ |\!-_{DM}\ \lambda x.id\ x : 'b$ $\quad$ $H\ [f \leftarrow gen('b)]\ |\!-_{DM}\ id\ f(id\ f) : 'a \to 'a$

$H\ |\!-_{DM}\ let\ f = \lambda x.id\ x\ in\ id\ f(id\ f) : 'a \to 'a$

Un errore di tipo in DM

$$\frac{\tau \in inst(H(x))}{H \mid\!-_{DM}\ id\ x : \tau}$$

$$\frac{H\ [x \leftarrow \tau_1]\ \mid\!-_{DM}\ e : \tau_2}{H \mid\!-_{DM}\ \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{H \mid\!-_{DM}\ e_1 : \tau_1 \to \tau_2 \quad H \mid\!-_{DM}\ e_2 : \tau_1}{H \mid\!-_{DM}\ e_1(e_2) : \tau_2}$$

'c = 'a→'a, 'e='c

H [f ← 'c →'c] |-$_{DM}$ id f :'e→('a →'a)

fallimento

H [f ← 'c →'c] |-$_{DM}$ id f : 'c

H [f ← 'c →'d] |-$_{DM}$ id f(id f): 'a →'a

'd = 'c

H [x ← 'c] |-$_{DM}$ id x : 'd

H |-$_{DM}$ λf.id f(id f): 'b →('a →'a)

'b = 'c →'d

H |-$_{DM}$ λx.id x : 'b

H |-$_{DM}$ (λf.id f(id f))(λx.id x) : 'a →'a

L'algoritmo di inferenza di Damas-Milner 1

$$\texttt{int n} \Rightarrow_{DM} \langle \texttt{H, int} \rangle$$

$$\frac{\tau \in \texttt{inst(H(x))}}{\texttt{id x} \Rightarrow_{DM} \langle \texttt{H}, \tau \rangle}$$

$$\frac{e_1 \Rightarrow_{DM} \langle \texttt{H, int} \rangle \qquad e_2 \Rightarrow_{DM} \langle \texttt{H, int} \rangle}{e_1 + e_2 \Rightarrow_{DM} \langle \texttt{H, int} \rangle}$$

$$\frac{e_1 \Rightarrow_{DM} \langle \texttt{H, int} \rangle \quad e_2 \Rightarrow_{H} \langle \texttt{H}, \tau \rangle \quad e_3 \Rightarrow_{DM} \langle \texttt{H}, \tau \rangle}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Rightarrow_{DM} \langle \texttt{H}, \tau \rangle}$$

- le regole sono più semplici di quelle di H: niente $\mathcal{H}$, niente `gci` ma tutto passa da H
  - le unificazioni sono nascoste (vedi esempio dopo)
  - continua a non essere trasparente

L'algoritmo di inferenza di Damas-Milner 2

$$\frac{e_1 \Rightarrow_{DM} \langle H, \tau_2 \rightarrow \tau\rangle \qquad e_2 \Rightarrow_{DM} \langle H, \tau_2\rangle}{e_1\ e_2 \Rightarrow_{DM} \langle H, \tau\rangle}$$

$$\frac{e \Rightarrow_{DM} \langle H[x \leftarrow \tau_1], \tau_2\rangle}{\lambda x.e \Rightarrow_{DM} \langle H, \tau_1 \rightarrow \tau_2\rangle}$$

$$\frac{\lambda x.e \Rightarrow_{DM} \langle H[f \leftarrow \tau_1 \rightarrow \tau_2], \tau_1 \rightarrow \tau_2\rangle}{\mu f.\lambda x.e \Rightarrow_{DM} \langle \tau_1 \rightarrow \tau_2\rangle}$$

$$\frac{e_1 \Rightarrow_{DM} \langle H, \tau_1\rangle \qquad e_2 \Rightarrow_{H} \langle H[x \leftarrow gen(\tau_1)], \tau_2\rangle}{let\ x\ be\ e_1\ in\ e_2 \Rightarrow_{DM} \langle H, \tau_2\rangle}$$

- si noti che la regola per le funzioni ricorsive non prevede alcun calcolo di punto fisso

Un esempio di inferenza in DM

$$\frac{\tau \in \texttt{inst}(H(x))}{\texttt{id x} \Rightarrow_{DM} \langle H, \tau \rangle}$$

$$\frac{e_1 \Rightarrow_{DM} \langle H, \texttt{int} \rangle \qquad e_2 \Rightarrow_{DM} \langle H, \texttt{int} \rangle}{e_1 + e_2 \Rightarrow_{DM} \langle H, \texttt{int} \rangle}$$

$$\frac{e \Rightarrow_{DM} \langle H[x \leftarrow \tau_1], \tau_2 \rangle}{\lambda x.e \Rightarrow_{DM} \langle H, \tau_1 \rightarrow \tau_2 \rangle}$$

- $\lambda x.\lambda y.\texttt{id x + id y} \Rightarrow_{DM} \langle H, 'a \rightarrow 'b \rangle$
- $\lambda y.\texttt{id x + id y} \Rightarrow_{DM} \langle H[x \leftarrow 'a], 'b \rangle$ $\quad 'b = 'c \rightarrow 'd$
- $\texttt{id x + id y} \Rightarrow_{DM} \langle H[x \leftarrow 'a; y \leftarrow 'c], 'd \rangle$ $\quad 'd = \texttt{int}$
- $\texttt{id x} \Rightarrow_{DM} \langle H[x \leftarrow 'a; y \leftarrow \texttt{int}], \texttt{int} \rangle$ $\quad 'c = \texttt{int}$

$'a = \texttt{int}$

  - $\texttt{id y} \Rightarrow_{DM} \langle H[x \leftarrow \texttt{int}; y \leftarrow \texttt{int}], \texttt{int} \rangle$

  - $\lambda x.\lambda y.\texttt{id x + id y} \Rightarrow_{DM} \langle H, \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \rangle$

Damas-Milner-Mycroft

- diverso solo nella regola delle funzioni ricorsive, che contiene
  - un calcolo di punto fisso
  - la generalizzazione del tipo corrente inserito nell'ambiente per il nome della funzione ricorsiva
- di conseguenza, ricorsione polimorfa
  - ma non è un algoritmo!
- numerosi esempi di programmi che mostrano la differenza fra i vari sistemi di tipo nel capitolo "interprete astratto sui tipi"