Giorno 1
Programmazione Funzionale

Sommario

La programmazione funzionale è basata su *computazioni senza stato*, che procedono invece mediante *riscritture* di espressioni. Senza stato non c'è iterazione, quindi l'unico costrutto per il controllo di sequenza è la ricorsione. Sono presentati gli aspetti generali della programmazione funzionale, il λ-calcolo e degli esempi di estensioni tipate di questo (λ-calcolo tipato semplice, PCF)

1 Aspetti generali

1.1 Dichiarazione e applicazione

1. Sia $f(x) = x^2$ la funzione che associa ad x il suo quadrato;
2. Allora $f(2) = 4$.

In (1) l'espressione sintattica $f(x)$ è usata per introdurre il nome della funzione *anonima* $x \mapsto x^2$, mentre in (2) indica il risultato dell'*applicazione* di f ad uno specifico valore.

Per distinguere tra nome e corpo della funzione, scriviamo (in OCaml):

- **Dichiarazione di funzione:**

  ```
  let f = (fun x -> x * x) in
  ```

  Dove `fun x -> x * x` si dice *astrazione*.

- **Applicazione di funzione**

  ```
  f(2);;   (* Stampa 4 *)
  ```

Si noti che le parentesi non sono necessarie, avrei potuto infatti scrivere semplicemente `f 2`.

1.2 Funzioni di ordine superiore, currying

Le funzioni possono prendere come argomenti, o restituire, altre funzioni. Ad esempio si possono dichiarare funzioni "di più parametri" (currying) scrivendo:

```
let f = (fun x -> fun y -> x*y);;
```

Ossia la funzione $f(x, y) = xy$, scritta come:

$$f : x \mapsto (y \mapsto (x \cdot y))$$

`f` si può applicare scrivendo, ad esempio:

- "`f 5 (6)`" (si ricordi che `f 5` è la funzione $y \mapsto 5y$)
- "`(f 5) 6`"
- "`(f 5) (6)`"
- "`f 5 6`"

1.3 Computazione come riduzione

Un linguaggio funzionale è costituito esclusivamente da espressioni.

La valutazione di un'espressione consiste in una riscrittura (= *riduzione*) di questa, che avviene sostituendo testualmente una sottoespressione del tipo "funzione applicata ad un argomento" con il corpo della funzione in cui si sostituiscono le occorrenze del parametro formale con il parametro attuale, e.g.

Sia f = `(fun x -> x * x)`   `f 5` $\rightarrow$ `(x * x) (5)` $\rightarrow$ `5 * 5`

1.3.1 Redex

Un Redex è una *espressione riducibile*, ossia un'applicazione della forma `(fun x -> corpo) arg`

- Il *ridotto* di un redex è l'espressione che si ottiene sostituendo i parametri `x` del corpo con `arg`.
- β-regola: Se in un'espressione `exp1` compare come sottoespressione un redex, `exp1` si riduce nell'espressione `exp2` in cui il redex è sostituito con il suo ridotto.

1.4 Valutazione

- **Valori**: Si dice *valore* un'espressione che non deve essere ulteriormente riscritta. Un valore può essere primitivo o funzionale.
- **Capture-avoiding substitution**: Lo stesso nome non è mai dato a variabili distinte (variabili "fresche", vd. esempio nella parte sul λ-calcolo)
- **Strategie di valutazione**:
  - **Valutazione eager** – *call by value*, un redex è valutato solo se la sua parte argomento è un valore: si valuta quindi l'espressione del parametro attuale*prima di essere associata al parametro formale.*
  - **Valutazione normale** – *call by name*, in "ordine normale"; Un redex è valutato prima della sua parte argomento: l'espressione del parametro attuale *non viene valutata prima di essere associata al parametro formale.*

2 λ-calcolo

2.1 Sintassi

Un programma è un'espressione:

$$\begin{aligned} e ::= {} & x && \text{Variabile} \\ & |\lambda x.e && \text{Astrazione (fun dec)} \\ & |ee && \text{Applicazione (fun call)} \end{aligned}$$

Regole d'inferenza:

Variabile:

$$\frac{x \in Var}{x \in Exp}$$

Astrazione:

$$\frac{x \in Var \quad e \in exp}{\lambda x.e \in Exp}$$

Applicazione:

$$\frac{e_1 \in Exp \quad e_2 \in Exp}{e_1 e_2 \in Exp}$$

Convenzioni sintattiche:

- Lo scope del λ si estende il più a destra possibile:

$$\lambda x.\lambda y.xy = \lambda x.(\lambda y.(xy))$$

- L'applicazione associa a sinistra:

$$e_1 e_2 e_3 = (e_1 e_2) e_3$$

Aggiungiamo le **operazioni aritmetiche** e le **costanti numeriche** al lambda calcolo per semplificare.

2.2 Variabili libere e legate

Il λ in $\lambda x.e$ agisce da operatore di *binding*, cioè lega la variabile x nell'espressione e. La variabile x può essere sostituita da un'altra variabile *fresca* (dove fresca = nuova, diversa in nome da tutte le altre che stiamo trattando) ed il programma rimane lo stesso.

Definizione: Una variabile introdotta (dichiarata) da un λ si dice **legata** da quel λ.

Definizione: Una variabile che non è associata a nessun λ si dice **libera**.

Definizione: Si dicono **α-equivalenti** due espressioni uguali a meno della sostituzione di una variabile legata con una variabile fresca., e.g. $\lambda a.(a+1) \equiv_\alpha \lambda b.(b+1)$.

2.2.1 Definizione formale

L'insieme FV delle variabili libere è definito da:

$$\begin{aligned}
&FV(x) = \{x\} \\
&FV(e_1e_2) = FV(e_1) \cup FV(e_2) \\
&FV(\lambda x.e) = FV(e) \setminus \{x\}
\end{aligned}$$

2.3 Alberi di sintassi astratta

Un albero di sintassi astratta (AST) rappresenta un programma.

- Le foglie sono variabili
- Un'astrazione $\lambda x.e$ è un nodo etichettato λx che ha come sottoalbero l'AST di e.
- Un'applicazione e_1e_2 consiste di un nodo etichettato @ con come sottoalberi sinistro e destro gli AST di e_1 ed e_2

2.4 Esecuzione

L'esecuzione avviene come descritto in 1.3; quando si valuta una λ-espressione $(\lambda x.e_1)e_2$ si sostituisce ogni occorrenza di x in e_1 con e_2 (valutata o meno, dipende se call-by-name o call-by-value).

Notazione: La notazione $e_1\{x := e_2\}$ descrive la λ-espressione e_1 in cui ogni occorrenza della variabile x è sostituita da e_2.

β-riduzione: È la regola che cattura precisamente la nozione di applicazione funzionale ($\sim \beta$-regola):

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

2.4.1 Capture Avoiding Substitution

Problema: Se e_2 contiene una variabile libera y che è legata in e_1, la variabile y potrebbe essere catturata, e l'espressione potrebbe cambiare di significato. E.g.

$$(\lambda x.(\lambda y.(x+y)))\underbrace{y}_{\text{libera}} \to (\lambda y.(x+y))\{x := y\} \to \underbrace{(\lambda y.(y+y))}_{y\text{ catturata}}$$

Soluzione: e_1 deve essere α-convertita:

$$(\lambda y.(x+y))\{x := y\} \underset{\alpha\text{-conv, } z \text{ fresh}}{\to} (\lambda z.(x+z))\{x := y\} \to \underbrace{(\lambda z.(y+z))}_{y\text{ libera}}$$

Definizione formale della Capture Avoiding Substitution

$$\begin{aligned}
x\{x := e\} &\equiv e \\
y\{x := e\} &\equiv y, \text{ se } y \neq x \\
(e_1e_2)\{x := e\} &\equiv (e_1\{x := e\})(e_2\{x := e\}) \\
(\lambda y.e_1)\{x := e\} &\equiv \begin{cases} \lambda y.(e_1\{x := e\}) & \text{se } y \neq x \text{ e } y \notin FV(e) \\ \lambda z.((e_1\{x := z\})\{x := e\}) & \text{se } y \neq x \text{ e } y \in FV(e)\text{, z fresca} \end{cases}
\end{aligned}$$

2.5 Interprete del λ-calcolo

2.5.1 β riduzione, formalmente

La β-riduzione ($\rightarrow$) è definita da:

$$(\lambda x.e_1)e_2 \rightarrow e_1\{x := e_2\}$$

$$\frac{e_1 \rightarrow e'}{e_1e_2 \rightarrow e'e_2} \qquad \frac{e_2 \rightarrow e'}{e_1e_2 \rightarrow e'_1} \qquad \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

Definizioni e notazioni

- Indichiamo con $\Longrightarrow$ la chiusura riflessiva e transitiva di $\rightarrow$
- Una λ-espressione e_1 è β*-riducibile* alla λ-espressione e_2 se $e_1 \Longrightarrow e_2$
- Due λ-espressioni e_1, e_2 si dicono β**-equivalenti** se:
  - Sono α-equivalenti;
  - Una delle due è β-riducibile all'altra ($e_1 \Longrightarrow e_2 \vee e_2 \Longrightarrow e_1$);
  - Sono entrambe β-riducibili alla stessa λ-espressione ($e_1 \Longrightarrow e \wedge e_2 \Longrightarrow e$).

  In altre parole, due λ-espressioni sono β-equivalenti quando sono indistinguibili dal punto di vista computazionale: calcolano gli stessi risultati.
- Una λ-espressione si dice in **forma normale beta** se non è ulteriormente riducibile.

Teorema di Church Rosser: L'ordine in cui vengono scelte le β-riduzioni non influisce sul risultato finale.

2.5.2 Non terminazione

La λ-espressione $\Omega = (\lambda x.xx)(\lambda x.xx)$ non può essere ridotta in forma normale, poiché la riduzione produce nuovamente Ω.

$$(\lambda x.xx)(\lambda x.xx) \rightarrow (xx)\{x := (\lambda x.xx)\} \rightarrow (\lambda x.xx)(\lambda x.xx)$$

2.5.3 Ricorsione: il combinatore Y

La ricorsione è implementata tramite la funzione:

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

che gode della proprietà:

$$YF \equiv_\beta F(YF)$$

Infatti:

$$\begin{aligned} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))F &\Longrightarrow (\lambda x.F(xx))(\lambda x.F(xx)) \\ &\Longrightarrow F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &\equiv_\beta F(YF) \end{aligned}$$

2.5.4 Costrutti nel λ-calcolo

- Booleani:
  - True: $\lambda t.\lambda f.t$

    (tra t e f seleziona il primo)
  - False: $\lambda t.\lambda f.f$

    (tra t e f seleziona il secondo)

- Condizionale:

  $$IF = \lambda \texttt{c}.\lambda \texttt{then}.\lambda \texttt{else.c then else}$$

  - Se c è true:
    $(\lambda t.\lambda f.t)$ `then else` $\Longrightarrow$ `then`
  - Se c è false:
    $(\lambda t.\lambda f.f)$ `then else` $\Longrightarrow$ `else`

Esempio di Condizionale:

$$\begin{aligned}
& (\lambda \texttt{c}.\lambda \texttt{then}.\lambda \texttt{else.c then else})\,\texttt{true}\, AB \\
& \Longrightarrow (\lambda \texttt{then}.\lambda \texttt{else}.(\lambda t.\lambda f.t)\texttt{then else})AB \\
\textit{(sosituisco A e B)} & \Longrightarrow (\lambda t.\lambda f.t)AB \Longrightarrow (\lambda f.A)B \Longrightarrow A
\end{aligned}$$

- Numerali di Church:
  - Si definisce $C_0 = \lambda z.\lambda s.z$ (dove $z \sim$ zero ed $s \sim$ *succ*)
  - $C_1 = \lambda z.\lambda s.sz$
  - $C_n = \lambda z.\lambda s.s(s(\dots(sz)))$
  - Funzione **plus** $= \lambda m.\lambda n.\lambda z.\lambda s.m(nzs)s$

    Idea: Facendo $plus(a)(b)$ voglio sostituire il corpo di a alla z nel corpo di b:

    $$plus(\lambda z.\lambda s.ssz)(\lambda z.\lambda s.sz) \Longrightarrow (\lambda z.\lambda s.sssz)$$

    La funzione *plus* fa proprio questo: $A = \lambda z.\lambda s.ssz \quad B = \lambda z.\lambda s.sz$

    $$\begin{aligned}
    & (\lambda m.\lambda n.\lambda z.\lambda s.m(nzs)s)AB \\
    & \Longrightarrow \lambda z.\lambda s.B(\underline{Azs})s \\
    & \Longrightarrow \lambda z.\lambda s.B(ssz)s \\
    & = \lambda z.\lambda s.\underline{(\lambda z.\lambda s.sz)(ssz)}s \\
    \textit{(idea)} & \Longrightarrow \lambda z.\lambda s.\underline{(\lambda s.sssz)s} \\
    & \Longrightarrow \lambda z.\lambda s.sssz
    \end{aligned}$$
  - Funzione **times** $= \lambda m.\lambda n.m\ C_0(Plus\ n)$
    Idea: Sommare n a zero m volte.

2.5.5 Scoping statico

Il lambda calcolo adotta un meccanismo di scoping statico per definire la visibilità delle variabili.

Ad esempio: in $(\lambda x.x(\lambda x.x))z$ la x più a destra della variabile si riferisce alla x introdotta nel secondo λ. Tramite l'α-conversione si ottiene una versione equivalente:

$$\lambda x.x(\lambda x.x))z \equiv_\alpha \lambda x.x(\lambda y.y))z$$

2.5.6 Dichiarazioni locali

Le dichiarazioni di variabili locali sono codificate tramite λ-astrazione e applicazione:

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \approx (\lambda x.e_2)e_1$$

2.5.7 Strutture dati

Si possono rappresentare anche strutture dati, e.g. una coppia (a, b) si può rappresentare come:

$$\lambda x.IF\ x\ a\ b$$

E posso accedere a e b utilizzando le due funzioni:

$$Fst = \lambda f.fTrue \qquad\qquad Snd = \lambda f.fFalse$$

2.5.8 Call-by-Value vs Call-by-Name

- Riduzione call-by-value:

  $$(\lambda x.e_1)v \to e_1\{x := v\}$$

  $$\frac{e_1 \to e'}{e_1e_2 \to e'e_2} \qquad\qquad \frac{e_2 \to e'}{ve_2 \to ve'}$$

  Idea: partendo da sinistra, seleziono la prima applicazione e_1e_2 e riduco l'espressione e_1 finché non è ridotta ad un valore (funzionale), poi valuto la parte argomento e_2 e infine applico la funzione.

- Riduzione call-by-name

  $$(\lambda x.e_1)e_2 \to e_1\{x := e_2\} \qquad\qquad \frac{e_1 \to e'}{e_1e_2 \to e'e_2}$$

  Idea: Partendo da sinistra, seleziono l'applicazione e_1e_2, riduco e_1 ad un valore funzionale ed applico la funzione prima di calcolare il valore dell'argomento $\Longrightarrow$ *applico la funzione appena possibile.*

- In alcuni casi la call-by-value e la call-by-name possono comportarsi in modo diverso, ad esempio la valutazione dell'espressione:

  $$\texttt{IF True True } (\Omega\Omega)$$

  Termina e restituisce `True` se call-by-name (non cerca di valutare $\Omega\Omega$), mentre non termina mai se call-by-value.

3 Controllo dei tipi

Nel λ-calcolo è possibile scrivere programmi che non sono corretti rispetto all'uso inteso dei valori:

$$False\ 0 = (\lambda t.\lambda f.f)(\lambda z.\lambda s.\lambda z) \to \lambda f.f$$

Questo programma produce un valore, ma non ha senso. Dobbiamo assegnare dei *tipi* ai dati, ossia degli attributi che descrivono come il linguaggio permette di usare quel particolare dato.

3.1 Sistema dei tipi

Un sistema di tipi è un metodo sintattico, effettivo per dimostrare l'assenza di comportamenti anomali del programma strutturando le operazioni del programma in base ai tipi di valori che calcolano.

- **Effettivo**: si può definire un algoritmo che calcola un'approssimazione statica dei comportamenti a runtime del programma
- **Strutturale**: I tipi delle componenti di un programma sono calcolati in modo composizionale, i.e. il tipo di un'espressione dipende solo dai tipi delle sue sottoespressioni

3.1.1 Esempi

[Divisione per zero, js $\to$ ts]

3.1.2 Controllo dei tipi

Il *type checker* verifica che le intenzioni del programmatore (espresse dalle annotazioni di tipo) siano rispettate dal programma.

Il controllo dei tipi può essere statico o dinamico. Se è statico trova gli errori prima dell'esecuzione del programma, e non degrada le prestazioni.

3.2 Case study: sistema di tipo per espressioni

Sintassi:

```
 E ::= V | if E then E else E
V ::= true | false | NV | isZero E
 NV ::= 0 | succ NV | pred NV
```

3.2.1 Esempi di regole di esecuzione:

- `IF-TRUE`

$$\texttt{if}\ \textbf{true}\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \to\ E_1$$

- `IF-FALSE`

$$\texttt{if}\ \textbf{false}\ \texttt{then}\ E_1\ \texttt{else}\ E_2\ \to\ E_2$$

- `IF-COND`

$$\frac{E \to E'}{\texttt{if}\ E\ \texttt{then}\ E1\ \texttt{else}\ E_2\ \to\ \texttt{if}\ E'\ \texttt{then}\ E1\ \texttt{else}\ E_2}$$

- Regole della forma:

$$\frac{E \to E'}{pred\ E \to pred\ E'} \qquad \frac{E \to E'}{succ,\ E \to succ\ E'} \qquad \text{... (isZero)}$$

- $pred(succ\ NV) \to NV$ e viceversa
- $pred\ 0 \to 0$, $isZero\ 0 \to true$, $isZero(succ\ NV) \to false$

3.2.2 Tipi per espressioni

Gli unici due tipi di questo linguaggio sono `Bool` e `Nat`
Le regole di controllo dei tipi sono molto intuitive:

- $true : Bool \qquad false : Bool \qquad 0 : Nat$
- Regole della forma:

$$\frac{E : Nat}{succ\ E : Nat} \qquad \frac{E : Nat}{pred,\ E : Nat} \qquad \frac{E : Nat}{isZero,\ E : Bool}$$

- $\dfrac{E : Bool \quad E_1 : T \quad E_2 : T}{if\ E\ then\ E_1\ else\ E_2 : T}$, che non permette ad esempio di associare un tipo all'espressione

  ```
  if true then 0 else false
  ```

  ...anche se questa assume sempre valore numerico. In questo modo si garantisce però la composizionalità (tipo delle espressioni dipende solo da tipo delle sottoespressioni)

  Ogni coppia espressione-tipo (E, T) della relazione di tipo è caratterizzata da un albero di derivazione costruito da istanze delle regole di inferenza.

3.2.3 Type Safety: Progresso e Conservazione

La correttezza (type safety) di un sistema di tipo è espressa formalmente dalle proprietà di *progresso e conservazione.*

- **Progresso**: se $E : T$ allora E è un valore oppure esiste E' tale che $E \to E'$

  (ossia: una espressione ben tipata non si blocca a run-time)
- **Conservazione**: Se $E : T$ e $E \to E'$ allora $E' : T$

  (ossia: i tipi sono preservati dalle regole di esecuzione)

3.2.4 Lemmi di inversione

I lemmi di inversione sono "le regole di tipo lette al contrario", e.g.:

- Dalla regola $true : Bool$ si ottiene il lemma di inversione: "Se $true : R$ allora $R = Bool$"
- Dalla regola sui tipi delle espressioni condizionali si ottiene:

$$\text{Se } (if\ E\ then\ E_1\ else\ E_2) : R \ \texttt{allora}\ E : Bool,\ E_1 : R,\ E_2 : R$$

Queste regole possono essere codificate in un algoritmo che restituisce il tipo di un'espressione (e sono utilizzate implicitamente al passo induttivo delle dimostrazioni di progresso e conservazione).

3.2.5 Forme canoniche

Le forme canoniche sono i valori che possono assumere le espressioni di un certo tipo:

- Se v è di tipo `Bool`, allora $v = true$ oppure $v = false$
- Se v è di tipo `Nat`, allora v è un valore numerico.

3.2.6 Progresso

Se $E : T$ allora E è un valore oppure esiste E' tale che $E \to E'$

Dimostrazione. Per induzione sulla struttura della derivazione di $E : T$

- **Casi base:**
$$true : Bool \qquad\qquad false : Bool \qquad\qquad 0 : Nat$$
  Immediato, poiché sono valori.
- **Casi induttivi:** (vediamo, come esempio, quello dell'If-Then-Else, gli altri sono analoghi.)
  Si consideri la regola:
$$\frac{E : Bool \quad E_1 : T \quad E_2 : T}{if\ E\ then\ E_1\ else\ E_2 : T}$$
  Per ipotesi induttiva vale il progresso per E, E_1, E_2, cioè o sono valori o si può fare un passo.
  - Se E è un valore allora per le forme canoniche deve valere $True$ o $False$. In questo caso si possono applicare le regole IF-TRUE e IF-FALSE per mostrare che l'espressione fa un passo e diventa E_1 o E_2, per cui vale la regola, per ipotesi induttiva.
  - Se E non è un valore si applica la regola IF-COND:
$$\frac{E \to E'}{\texttt{if } E \texttt{ then } E1 \texttt{ else } E_2 \to \texttt{if } E' \texttt{ then } E1 \texttt{ else } E_2}$$
    Quindi si fa un passo.

□

3.2.7 Conservazione

Se $E : T$ e $E \to E'$ allora $E' : T$

Dimostrazione. Analizziamo di nuovo solo il caso dell'If-Then-Else, per induzione.

$$\frac{E : Bool \quad E_1 : T \quad E_2 : T}{if\ E\ then\ E_1\ else\ E_2 : T}$$

(Caso base: I guess per vacuità vale sui valori(?))

Supponiamo che la regola valga per E, E_1, E_2.
Possiamo fare un passo utilizzando la regola IF-COND citata prima, e sapendo, per hp induttiva, che la conservazione vale per E il tipo rimarrà $Bool$, mentre i tipi di E_1 ed E_2 rimarranno gli stessi perché sono invariati. □

3.3 λ-calcolo tipato semplice

3.3.1 Estensione con i booleani

Estendiamo il lambda calcolo nel seguente modo:

$$\texttt{e ::= x | }\lambda\texttt{x:}\tau\texttt{.e | e e | true | false | if e then e else e}$$

E notiamo che nel parametro formale della lambda astrazione appare un'annotazione di tipo τ. La sintassi dei tipi è:

$$\tau\ \texttt{::= Bool | }\tau \to \tau$$

3.3.2 Sintassi simil-OCaml

Utilizziamo una nuova sintassi, analoga a quella del lamda-calcolo tipato appena introdotto:

$$\texttt{e ::= x | fun x:}\tau\texttt{ = e | Apply(e, e) | true | false | if e then e else e}$$

3.3.3 Ambiente

Le regole di tipatura si complicano rispetto al sistema di tipo per espressioni visto prima, poiché abbiamo delle variabili (la `x` in `fun x:`τ); ci serve perciò un *ambiente*.
L'ambiente è una funzione Γ che associa nomi a tipi. Per indicare $\Gamma(x_i) = \tau_i$ noi utilizzeremo la notazione:

$$\Gamma = x_1 : \tau_1, \quad x_2 : \tau_2, \quad \dots \quad x_k : \tau_k$$

Mentre utilizzeremo la notazione $\Gamma, x : \tau$ per indicare l'**estensione** di Γ con l'associazione $x : \tau$:

$$(\Gamma, x : \tau)(y) = \begin{cases} \tau & y = x \\ \Gamma(y) & y \neq x \end{cases}$$

Ovviamente questo vale solo se y è nell'ambiente: in caso contrario $\Gamma(y) = undefined$.

Nota: nel gergo dei compilatori l'ambiente è chiamato "tabella dei simboli".

3.3.4 Giudizio di tipo

Sia Γ un ambiente di tipo, si usa la notazione $\Gamma \vdash e : \tau$ per indicare che e ha tipo τ nell'ambiente Γ.

3.3.5 Regole di tipo

$$\Gamma \vdash true : Bool \qquad\qquad \Gamma \vdash false : Bool \qquad\qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Funzioni:

- *Definizione:*

$$\frac{(\Gamma, x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash (fun\ x : \tau_1 = e) : \tau_1 \to \tau_2}$$

  In questo modo implemento lo scoping; la dichiarazione del parametro x **sovrascrive** e annulla le precedenti dichiarazioni per x che sono in Γ.

- *Invocazione:*

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash Apply(e_1, e_2) : \tau_2}$$

3.3.6 Type safety

- **Progresso**: Se $\emptyset \vdash e : \tau$ allora e è un valore oppure esiste e' tale che $e \rightarrow e'$
- **Conservazione**: Se $\Gamma \vdash e : \tau$ e $e \rightarrow e'$ allora $\Gamma \vdash e' : \tau$

Le dimostrazioni sono simili a quelle fatte per le espressioni:

- Si deve fare una dimostrazione per ogni regola di derivazione
- Le regole che non hanno delle precondizioni sono usate come casi base, mentre quelle che ne hanno come casi induttivi

Progresso: Unico caso meno ovvio: $\mathit{Apply}(e_1, e_2), \quad \emptyset \vdash e_1 : \tau_1 \rightarrow \tau_2, \quad \emptyset \vdash e_2 : \tau_1$

Dimostrazione. Per ipotesi induttiva sappiamo che il progresso vale per e_1 ed e_2;

- Se le espressioni **possono fare un passo** allora si possono applicare le regole di riduzione dell'applicazione, quindi vale il progresso.
- Se **sono entrambe valori** allora per il lemma delle forme canoniche del lambda calcolo tipato (analogo a quelli per le espressioni e valido anche per valori funzionali) abbiamo che e_1 è della forma $(fun\ x : \tau_1 = e') : \tau_1 \rightarrow \tau_2$, quindi si può applicare la β-riduzione.

□

Conservazione: Con gli strumenti che abbiamo non possiamo dimostrare la conservazione della *Apply*, ci serve un nuovo lemma.

Lemma di sostituzione I tipi sono preservati dall'operazione di sostituzione:

$$\Gamma, x : \tau_1 \vdash e : \tau \quad \wedge \quad \Gamma \vdash e_1 : \tau_1 \implies \Gamma \vdash e\{x = e_1\} : \tau$$

Questo lemma si dimostra per induzione sulla derivazione di $\Gamma, x : \tau_1 \vdash e : \tau$, dimostrazione nelle note.

Possiamo adesso dimostrare la conservazione:

Dimostrazione. La regola di tipo per la apply è:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \mathit{Apply}(e_1, e_2) : \tau_2}$$

- **Caso:** $e_1 = (fun\ x : \tau_1 = e_3)$, allora fa un passo tramite la β-riduzione:

  $$\mathit{Apply}((fun\ x : \tau_1 = e_3), e_2) \rightarrow e_3\{x := e_2\}$$

  E per mostrare che vale la conservazione devo dimostrare che $e_3\{x := e_2\}$ ha lo stesso tipo di $\mathit{Apply}(e_1, e_2)$, ossia τ_2. La regola di tipo che applico a questo punto è:

  $$\frac{(\Gamma, x : \tau_1) \vdash e_3 : \tau_2}{\Gamma \vdash (fun\ x : \tau_1 = e_3) : \tau_1 \rightarrow \tau_2}$$

  che permette di dimostrare che $e_3 : \tau_2$, il che è diverso da $e_3\{x := e_2\} : \tau_2$, ma per il lemma di sostituzione sappiamo che, dato che il tipo di x ed il tipo di e_2 sono uguali, vale $e_3\{x := e_2\} : \tau_2$

□

3.3.7 Estensione con booleani e numeri

```
e ::= x | fun x:τ = e | Apply(e, e)
       | true | false | n | e⊕e | if e then e else e
```

Regole di tipo per i naturali:

$$\Gamma \vdash n : Nat \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \oplus : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

3.3.8 Dichiarazioni locali

Aggiungiamo alla sintassi il costrutto `let` x `=` e_1 `in` $e_2 : \tau$

Regole di valutazione:

$$\frac{e_1 \to e'}{let\ x = e_1\ in\ e_2 : \tau_2 \to let\ x = e'\ in\ e_2 : \tau_2}$$

$$let\ x = v\ in\ e_2 : \tau_2 \to e_2\{x = v\} : \tau_2$$

Queste regole mi obbligano a valutare completamente e_1 prima di sostituirla in e_2.

Regola di tipo:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{let\ x = e_1\ in\ e_2 : \tau_2}$$

3.3.9 Ricorsione

Si usa fix, che permette di ottenere il punto fisso di una funzione, come segue:

$$\frac{}{fix(fun\ x : \tau = e) \to e[x = fix(fun\ x : \tau = e)]} \qquad\qquad \frac{e \to e'}{fix\ e \to fix\ e'}$$

E adesso è possibile anche tipare funzioni che non terminano (cosa che non si poteva fare, e.g. si provi a tipare un'applicazione del combinatore omega):

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash fix\ e : \tau}$$

Il lambda calcolo tipato semplice + fix è noto in letteratura come PCF (Plotkin 1977).

3.3.10 Una sintassi più semplice per la ricorsione

Linguaggi come OCaml usano la sintassi `let rec`

```
let rec fact x =
        if x <= 1 then 1 else x * fact (x-1);;
```