
Controllo dei tipi

Linguaggi e tipi

Il problema che ci poniamo ora è quello di comprendere come la struttura dei tipi del linguaggio possa influenzare il progetto e la struttura di implementazione dei linguaggi di programmazione.

L'importanza dei tipi anche nel Lambda Calcolo

Abbiamo visto che il lambda calcolo ha la potenza espressiva delle macchine di Turing
Possiamo codificare dati e strutture dati come espressioni del calcolo

$$True = \lambda t.\lambda f.t$$
$$False = \lambda t.\lambda f.f$$

$$0 = \lambda z.\lambda s.z$$
$$1 = \lambda z.\lambda s.s\ z$$

Dati e strutture dati possono essere codificati tramote opportune funzioni (i valori del calcolo)

L’importanza dei tipi per il lambda calcolo

Dato che nel lambda calcolo I programmi e i valori sono funzioni possiamo facilmente scrivere programmi che non corretti rispetto all’uso inteso dei valori

$$\boldsymbol{False\ 0 = (\lambda t.\lambda f.f)(\lambda z.\lambda s.z)}$$

Quale è l’errore?

L’importanza dei tipi per il lambda calcolo

Dato che nel lambda calcolo I programmi e i valori sono funzioni possiamo facilmente scrivre programmi che non corretti rispetto all’uso inteso dei valori

$$\boldsymbol{False\ 0 = (\lambda t.\lambda f.f)(\lambda z.\lambda s.z) \rightarrow \lambda f.f}$$

L’applicazione False 0 produce un valore!!!!

… è solo un problema del lambda calcolo?

La stessa cosa accade nel linguaggio macchina

Le **istruzioni** sono parole della macchina: **sequenze di bit**

I **dati** sono codificati come parole macchina: **sequenze di bit**

Le **operazioni** prendono in ingresso **sequenze di bit** e restituiscono come risultato **sequenze di bit**

Cosa sono i tipi?

- Uno dei principi centrali dell'ingegneria del software è quello di realizzare opportuni **meccanismi** che permettano di ***rilevare precocemente*** **errori di programmazione**.
- I linguaggi di programmazione supportano questa idea con i sistema di tipo (e più in generale con gli strumenti di analisi statica).
- Il tipo è un **attributo di un** dato che descrive come il linguaggio di programmazione permette di usare quel particolare dato.
  - I linguaggi di programmazione prevedono i tipi di dati di base come numeri interi (di varie dimensioni), numeri in virgola mobile (che approssimano i numeri reali), caratteri e booleani.
- **Un tipo limita i valori che un'espressione, come una variabile o una funzione, può assumere. Inoltre, definisce le operazioni che possono essere fatte sui dati e il modo in cui i valori di quel tipo possono essere memorizzati**.

Sistema dei tipi

- Un sistema dei 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**.

Sistema dei tipi

- Un sistema dei 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**.
- **Metodo sintattico:** la struttura sintattica guida il metodo di analisi del comportamento dei programmi

Sistema dei tipi

- Un sistema dei 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**.
- **Metodo sintattico:** la struttura sintattica guida il metodo di analisi del comportamento dei programmi
- **Effettivo**: si può definire un algoritmo che calcola una approssimazione **statica** dei comportamenti a **run-time** di un programma

Sistema dei tipi

- Un sistema dei 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**.
- **Metodo sintattico:** la struttura sintattica guida il metodo di analisi del comportamento dei programmi
- **Effettivo**: si può definire un algoritmo che calcola una approssimazione **statica** dei comportamenti a **run-time** di un programma
- **Strutturale**: i tipi assegnati alle component di un programma sono calcolati in modo *composizionale:* il tipo di un'espressione dipende solo dai tipi delle sue sottoespressioni

Sistema dei tipi

Un **sistema dei tipi** associa *tipi* ai valori calcolati

Esaminando il flusso dei valori calcolati, il sistema dei tipi tenta di dimostrare che non avvengano *errori di tipo*.

Il sistema stesso determina che cosa costituisce un errore di tipo, garantendo che le operazioni che si aspettano un certo tipo di valore non siano utilizzate con valori per i quali quell'operazione non ha senso.

Esempio

Supponiamo che l'espressione

```
if (condizione_complicata)
   { return 5/0; }
```

sia inserita all'interno di un programma di grosse dimensioni

Se i test del programma non forzano mai la valutazione <condizione_complicata>, l'errore nel ramo then (divisione per zero) non verrebbe mai individuato.
Tuttavia per un qualche input non considerate nella batteria dei test potrebbe accadere che venga eseguito il ramo then generando un errore di esecuzione.

Questo errore avrebbe potuto chiaramente essere evitato se il programmatore e/o l'implementazione del linguaggio avessero segnalato il valore 5/0 come qualcosa di inappropriate

Passiamo a JavaScript

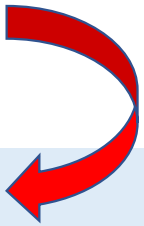
```
function addNumbers(x, y) {
  return x + y;}
// invocazione utente
console.log(addNumbers(3, "0"));
```

JavaScript Coercion:
l'operatore primitivo + può essere invocato con una coppia di valori (**number, string)**

Coercion: il valore numerico viene trasformato in un valore di tipo stringa.
viene stampato il valore "30" … una stringa.

Passiamo a TypeScript

```
function addNumbers(x: number, y: number) {
   return x + y;}
// invocazione utente
console.log(addNumbers(3, "0"));
```



TypeScript: segnala un errore di tipo rilevando questo bug

Type safety

La mancanza di **type safety** (correttezza dell'suo dei tipi) permette di scrivere programmi pieni di bug e … permette agli attaccanti di sfruttare bug per alterare maliziosamente il comportamento del programma o per prendere il pieno controllo del flusso di controllo.

I tipi sono essenziali nella programmazione moderna

- **Organizzare** i dati in strutture di alto livello
- **Documentare** il programma traminte informazioni di base sul significato delle variabili e delle funzioni
- **Informare** il compilatore di quanta memoria è necessaria per memorizzare un valore di un certo tipo
- **Specificare** le caratteristiche del comportamento delle funzioni
  - "tipi come specifiche" è un'idea importante della ingegneria del software

Controllo dei tipi

- **Statico** (in fase di compilazione) o **dinamico** (in fase di esecuzione),
  - Controllo statico trova gli errori prima di mandare il programma in esecuzione, non degrada le prestazioni
- Controllo di tipo (type checker) **verifica** che le intenzioni del programmatore (espresse dalle annotazioni di tipo) siano rispettate dal programma
- Un programma che supera il controllo dei tipi è garantito comportarsi bene in fase di esecuzione: non applica mai un'operazione ad un valore di tipo non corretto.
- Un programma che controlla i tipi rispetta le astrazioni tipo introdotte nel programma.
- **Controllo dei tipi: forma di correttezza parziale del programma (rispetto ai tipi)**

I tipi sono importanti per comprendere la struttura dei linguaggi di programmazione

Il sistema di tipi di un linguaggio ha un forte effetto sul modo in cui si sviluppano programmi in quel linguaggio

Esempi

Nel Pascal il risultato di una funzione non può essere un tipo di array.
In Java, un array è un oggetto e gli array possono essere usati ovunque.
In OCAML il risultato dell'invocazione di una funzione può essere una funzione

Per comprendere appieno un linguaggio di programmazione abbiamo bisogno di capire il suo sistema di tipi
Le nozioni sottostanti I sistemi di tipo appaiono in forme differenti nei diversi linguaggi e ci aiutano a confrontare e capire le caratteristiche del linguaggio.

Domande significative

Come fa un progettista di linguaggio (o un programmatore) a sapere che programmi che superano il controllo dei tipi a run-time esibiscono le proprietà desiderate?

Per rispondere a questa domanda dobbiamo comprendere due aspetti essenziali
(1) come specificare i sistemi di tipi
(2) come dimostrare che un sistema di tipi è corretto.

Per dimostrare la correttezza è indispensabile comprendere nel dettaglio il significato dei programmo (cosa succede quando vengono eseguiti
Comprendere i sistemi di tipo porterà ad una comprensione più profonda del significato dei programmi.


Un primo caso di studio
Sistema di tipo per espressioni

Espressioni e valori

Espressioni

E::=
- **true**
- **false**
- **if E then E else E**
- **0**
- **succ E**
- **pred E**
- **isZero E**

Valori

V ::=
- **true**
- **false**
- **NV**

Valori numerici

NV ::=
- **0**
- **successor NV**

Regole di esecuzione (interprete di espressioni)

$$if\ true\ then\ E1\ else\ E2 \rightarrow E1$$ **IF-TRUE**

$$if\ false\ then\ E1\ else\ E2 \rightarrow E2$$ **IF-FALSE**

$$\frac{E \rightarrow E'}{if\ E\ then\ E1\ else\ E2 \rightarrow if\ E'\ then\ E1\ else\ E2}$$ **IF-COND**

Regole di esecuzione

$$\frac{E \to E'}{pred\ E \to pred\ E'}$$

$$pred\ (succ\ NV1) \to NV1$$

$$isZero\ 0 \to true$$

$$\frac{E \to E'}{succ\ E \to succ\ E'}$$

$$pred\ 0 \to 0$$

$$isZero\ (succ\ NV1) \to false$$

$$\frac{E \to E'}{isZero\ E \to isZero\ E'}$$

Tipi per espressoni

Il linguaggio delle espressioni prevede solo due **forme** sintattiche per i tipi: **Bool** o **Nat**

TIPI

T ::=
Bool
Nat

Controllo di tipo (type checker)

Il controllo di tipo definisce una relazione binaria (E, T) che associa il tipo T all'espressione E

Useremo la notazione E:T per indicare una coppia della relazione di assegnamento dei tipi.

Controllo dei tipi

Metodo sintattico

Regole definite per induzione strutturale sulla sintassi del programma

Regole di controllo dei tipi

$$\mathbf{true} : \mathbf{Bool}$$

$$\mathbf{false} : \mathbf{Bool}$$

$$\mathbf{0} : \mathbf{Nat}$$

$$\frac{E : \mathbf{Nat}}{\mathbf{succ}\ E : \mathbf{Nat}}$$

$$\frac{E : \mathbf{Nat}}{\mathbf{pred}\ E : \mathbf{Nat}}$$

$$\frac{E : \mathbf{Nat}}{\mathbf{isZero}\ E : \mathbf{Bool}}$$

$$\frac{E : \mathbf{Bool}, E1 : T, E2 : T}{\mathbf{if}\ E\ \mathbf{then}\ E1\ \mathbf{else}\ E2 : T}$$

Controllo dei tipi: derivazione

$$\frac{\overline{? \; ? \; ? \; ? \; ? \; ? \; ?}}{\mathbf{if}\ \mathbf{isZero}\ \ \mathbf{0}\ \mathbf{then}\ \mathbf{0}\ \mathbf{else}\ \mathbf{pred}\ \ \mathbf{0} : \mathbf{Nat}}$$

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

Controllo dei tipi: derivazione

$$\frac{\dfrac{\mathbf{0}:\boldsymbol{Nat}}{\boldsymbol{isZero}\ \mathbf{0}:\boldsymbol{Bool}} \quad \mathbf{0}:\boldsymbol{Nat} \quad \dfrac{\boldsymbol{O}:\boldsymbol{Nat}}{\boldsymbol{pred}\ \mathbf{0}:\boldsymbol{Nat}}}{\boldsymbol{if}\ \boldsymbol{isZero}\ \ \mathbf{0}\ \boldsymbol{then}\ \mathbf{0}\ \boldsymbol{else}\ \boldsymbol{pred}\ \ \mathbf{0}:\boldsymbol{Nat}}$$

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

Precisione vs Approssimazione

I sistemi di tipo sono generalmente **imprecisi**: non definiscono esattamente quale tipo di valore sarà restituito da ogni programma, ma solo **un'approssimazione conservativa.**

Esempio

$$\frac{E : Bool, E1 : T, E2 : T}{if\ E\ then\ E1\ else\ E2 : T}$$

Utilizzando questa regola non siamo in grado di associare un tipo all'espressione

$$if\ true\ then\ 0\ else\ false$$

Questa espressione sicuramente restiuisce come risultato un valore numerico

Composizionalità

Osservazione:
La regola di controllo di tipo del condizionale richiede che le espressioni che costituiscono il "ramo then" e il "ramo else" abbiamo lo stesso tipo, e che quell tipo sia il tipo del condizionale.

$$\frac{E: Bool, E1: T, E2: T}{if\ E\ then\ E1\ else\ E2: T}$$

La regola garantisce la composizionalità.
Perché?

Composizionalità

Osservazione:
La regola di controllo di tipo del condizionale richiede che le espressioni che costituiscono il "ramo then" e il "ramo else" abbiamo lo stesso tipo, e che quell tipo sia il tipo del condizionale.

$$\frac{E: Bool, E1: T, E2: T}{if\ E\ then\ E1\ else\ E2: T}$$

La regola garantisce la composizionalità.
Quale condizione permette di associare il tipo alle espressione

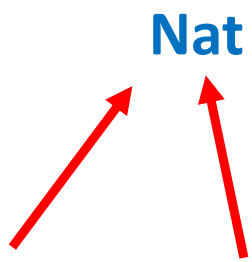
$$pred\ (IF\ E\ then\ E1\ else\ E2)$$

$$if\ (if\ E\ then\ e1\ else\ E2)\ then\ E3\ else\ E4$$

Composizionalità

Osservazione:
La regola di controllo di tipo del condizionale richiede che le espressioni che costituiscono il "ramo then" e il "ramo else" abbiamo lo stesso tipo, e che quell tipo sia il tipo del condizionale.

$$\frac{E: Bool, E1: T, E2: T}{if\ E\ then\ E1\ else\ E2: T}$$



La regola garantisce la composizionalità.
Quale condizione permette di associare il tipo alle espressione

$$pred\ (IF\ E\ then\ E1\ else\ E2)$$

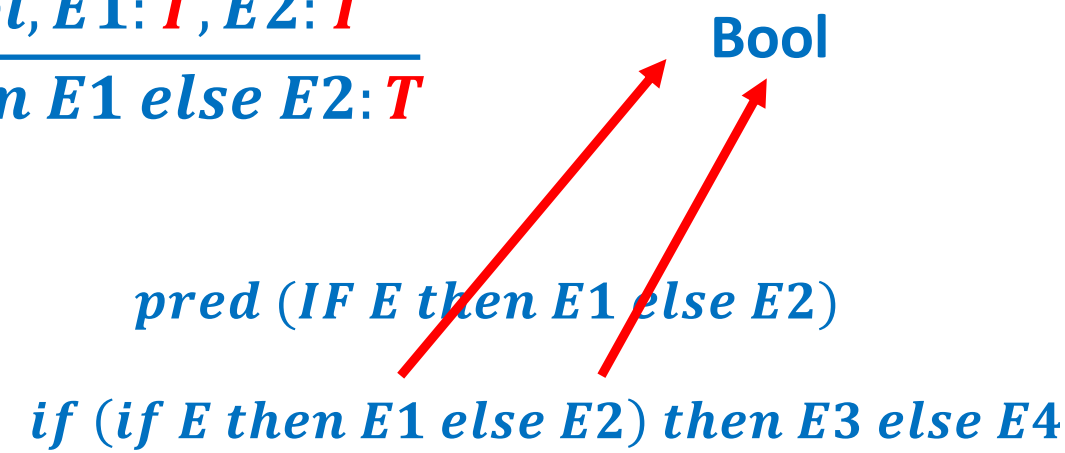
$$if\ (if\ E\ then\ E1\ else\ E2)\ then\ E3\ else\ E4$$

Composizionalità

Osservazione:
La regola di controllo di tipo del condizionale richiede che le espressioni che costituiscono il "ramo then" e il "ramo else" abbiamo lo stesso tipo, e che quell tipo sia il tipo del condizionale.

$$\frac{E: Bool, E1: T, E2: T}{if\ E\ then\ E1\ else\ E2: T}$$

La regola garantisce la composizionalità.
Quale condizione permette di associare il tipo alle espressione



Type Safety: Correttezza

La correttezza (type safety) del sistema di tipo
è espressa formalmente da queste due proprietà

Progresso: Se **E:T** allora **E** è un valore oppure $\boldsymbol{E \rightarrow E'}$
per una qualche espressione **E'**

Conservazione: Se **E:T** e $\boldsymbol{E \rightarrow E'}$ allora $\boldsymbol{E'}$**:T**

Type Safety: Correttezza

La correttezza (type safety) del sistema di tipo
è espressa formalmente da queste due proprietà

Progresso: Se **E:T** allora **E** è un valore oppure $E \rightarrow E'$
per una qualche espressione **E'**

Progresso: Una espressione ben tipata non si blocca a run-time

Conservazione: Se **E:T** e $E \rightarrow E'$ allora **E':T**

Conservazione: I tipi sono preservati dalle regole di esecuzione

Formalmente: Inversion lemmata

1. Se true:R allora R=Bool.
2. Se false:R allora R=Bool.
3. Se (if E then E1 else E2):R, allora E:Bool, E1:R, E2:R.
4. Se 0:R allora R=Nat.
5. Se succ E:R allora R=Nat e E:Nat.
6. Se pred E:R, allora R=Nat e E :Nat.
7. Se isZero E:R, allora R=Bool e E:Nat.

Dimostrazione: Per induzione strutturale sulla struttura delle regole.
Dimostrazione standard!!!

Dalla teoria all'algoritmo di typechecking

```
typeof(E) = if E = true then Bool
            else if E = false then Bool
            else if E = if E1 then E2 else E3 then
                 let T1 = typeof(E1) in
                 let T2 = typeof(E2) in
                 let T3 = typeof(E3) in
                        if T1 = Bool and T2=T3 then T2
                        else "not typable"
             else if E = 0 then Nat
             else if E= succ E1 then
                  let T1 = typeof(E1) in if T1 = Nat then Nat else "not typable"
             else if E = pred E1 then
                  let T1 = typeof(E1) in
                      if T1 = Nat then Nat else "not typable"
              else if t = iszero E1 then
                  leet T1 = typeof(E1) in +
                      if T1 = Nat then Bool else "not typable"
```

Forme canoniche

Se v è un valore di tipo Bool, allora v=true oppure v = false.

Se v è un valore di tipo Nat, alllora v è un valore numerico.

Progresso

Progresso: Se **E:T** allora **E** è un valore oppure $E \rightarrow E'$
per una qualche espressione **E'**

Dimostrazione

Per induzione sulla struttura della derivazione di E:T

Casi base

$true : Bool$

$false : Bool$

$0 : Nat$

Immediato: true, false e 0 sono valori

Progresso

Progresso: Se **E:T** allora **E** è un valore oppure $E \rightarrow E'$
per una qualche espressione **E'**

Dimostrazione

Per induzione sulla struttura della derivazione di E:T

Casi Induttivi

E = if E1 then E2 else E3:T

Applicando al regola di tipo per il condizionale otteniamo

$$\frac{E1: Bool, E2: T, E3: T}{E = if\ E1\ then\ E2\ else\ E3: T}$$

$$E1: Bool, E2: T, E3: T$$

Per ipostesi induttiva, E1 è un valore oppure esiste E4 tale che $E1 \rightarrow E4$
Nel caso sia un valore allora deve essere necessariamente true o false (lemma delle Forme normali). In questo caso applichiamo le regole IF per valori true e false.
Nel secondo caso otteniamo if E1 then E2 else E3 → if E4 then E2 else E3 e applichiamo l'induzione ad una dimostrazione più piccola.

Conservazione

Se **E:T** e $\boldsymbol{E \rightarrow E'}$ allora $\boldsymbol{E'}$**:T**

Dimostrazione. Solita induzione strutturale sulla derivazione di E:T.
Casi di base sono immediati (true, false e 0 sono valori)

Conservazione

Se **E:T** e $E \rightarrow E'$ allora E'**:T**

Dimostrazione. Solita induzione strutturale sulla derivazione di E:T.
Casi induttivi
Supponiamo che l'ultima regola applicata sia la regola del condizionale.
E = if E1 then E2 else E3:T

$$\frac{E: Bool, E2: T, E3: T}{if\ E1\ then\ E2\ else\ E3: T}$$

Pertanto abbiamo che

$$E1: Bool, E2: T, E3: T$$

E1 è un valore oppure, per ipotesi induttiva, esiste E4 tale che $E \rightarrow E4$
Nel caso sia un valore allora deve essere necessariamente true o false (lemma delle Forme normali). Nel caso sia true allora $E \rightarrow E2$ **e E2:T. Il caso falso è simmetrico.**
Nel secondo caso otteniamo if E1 then E2 else E3 $\rightarrow$ $E4$

Conservazione

Se **E:T** e $E \rightarrow E'$ allora **E':T**

Dimostrazione. Solita induzione strutturale sulla derivazione di E:T.
Casi induttivi
Supponiamo che l'ultima regola applicata sia la regole del condizionale.
E = if E1 then E2 else E3:T

$$\frac{E: Bool, E2: T, E3: T}{if\ E1\ then\ E2\ else\ E3: T}$$

Pertanto abbiamo che

$$E1: Bool, E2: T, E3: T$$

E1 è un valore oppure, per ipotesi induttiva, esiste E4 tale che $E \rightarrow E4$
Nel caso sia un valore allora deve essere necessariamente true o false (lemma delle Forme normali). Nel caso sia true allora $E \rightarrow E2$ e E2:T. Il caso falso è simmetrico.

Nel secondo caso otteniamo E= if E1 then E2 else E3 $\rightarrow$ *E4* e abbiamo tre posssibili casi per la derivazione

Consideriamo il caso in cui

E= if E1 then E2 else E3 → *E4 = if E1,1 the E2 else E3*

Apllicando l'ipotesi induttiva alla derivazione E1:Bool otteniamo che E1,1: Bool

Combinando questo risultato con le derivazioni di tipo (che valgono per ipotesi)

E2:T e E3:T

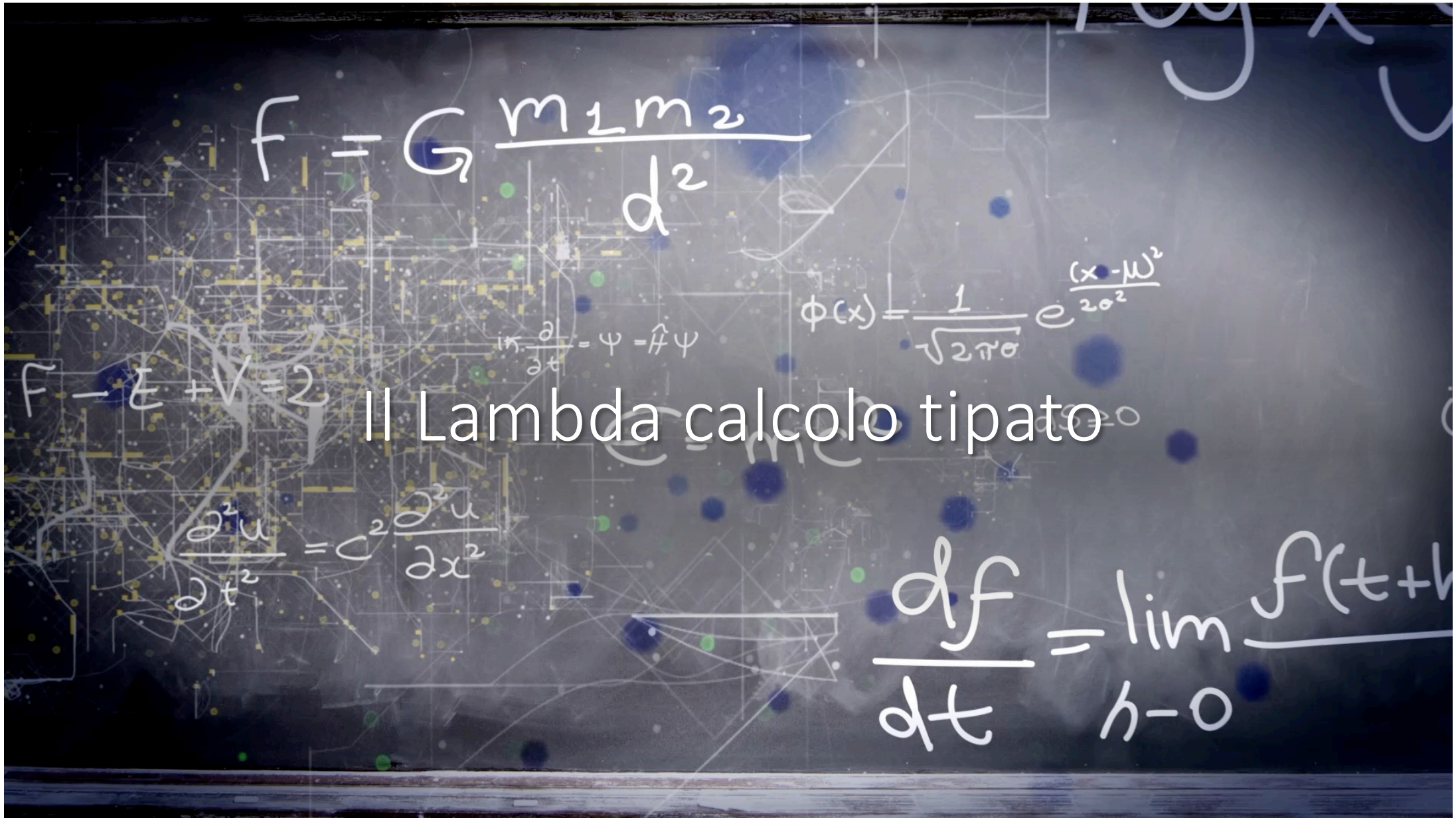
possiamo derivare applicando la regola IF

$$\frac{E1, 1 : Bool, E2 : T, E3 : T}{if\ E1, 1\ then\ E2\ else\ E3 : T}$$

Questo conclude il caso.
Gli altri casi sono simili

Conclusione

- Abbiamo visto come strutturare una strategia di definizione e di verifica di un sistema di tipo in un caso semplice: il linguaggio delle espressioni.
- Passiamo ora ad un secondo caso di studio.


Il Lambda calcolo tipato

Lambda calcolo tipato semplice

- Come abbiamo visto a lezione, Il lambda-calcolo nella sua forma"pura" non prevede tipi.
- Per comodità è più semplice avere un insieme di tipi di base, quindi, a rigore, nella letteratura, esistono numerose varianti del lambda calcolo tipato semplice a seconda della scelta dei tipi base.
- Noi consideriamo una variante costruita sui booleani.

Come progettare il sistema dei tipi per il lambda calcolo?

#1:

I tipi sono **nomi** che rappresentano **insiemi di valori**

Nel lambda calcolo abbiamo, oltre alle costanti, come insieme di valori le funzioni

$$\lambda x.\, x$$

Come progettare il sistema dei tipi per il lambda calcolo?

#2:

Gli **insiemi di valori di input validi** ogni operazione del programma devono essere descritti in termini di tipi

Nel lambda calcolo l'applicazione richiede che il primo argomento attuale, sia obbligatoriamente un valore funzionale.

Come progettare il sistema dei tipi per il lambda calcolo?

Idea #2:

Gli **insiemi di valori di input validi** ogni operazione del programma devono essere descritti in termini di tipi.

Ora dobbiamo fare una scelta: (1) annotiamo le lambda astrazioni con il tipo atteso dell'argomento

$$\lambda x{:}\,T\,.\,E$$

Come avviene nei linguaggi di programamzione con la notazione

fun x: T = E

Come progettare il sistema dei tipi per il lambda calcolo?

#2:

Gli **insiemi di valori di input validi** ogni operazione del programma devono essere descritti in termini di tipi.

Ora dobbiamo fare una scelta: (2) lasciamo le lambda astrazioni senza annotazione di tipo

$$\lambda x.\, E$$

e chiediamo alle regole di tipo di ”inferire" un'annotazione appropriata (come succeed in Ocaml o Haskell)?

Come progettare il sistema dei tipi per il lambda calcolo?

#3:

Il tipo associato ad una espressione in cui compaino variabili è determinato a partire dai tipi associato alle variabili

Strategia composizionale: Dimmi quali sono i tipi delle variabili in un programma e ti dirò quale è il tipo del programma.

Come progettare il sistema dei tipi per il lambda calcolo?

#4:

Dimostrare la correttezza del controllo dei tipi

Programmi che possano il controllo dei tipi (programmi ben tipati) non presenteranno a run-time comportanenti errati dovuti ad un uso non corretto dei tipi .

Nota

I passi **#1–#4** sono i passi metodologici da seguire ogniqualvolta si deve progettare un sistema di tipo per un linguaggio di programmazione.

#1: Sintassi dei tipi

Tipi semplici

τ ::=	**Tipi**
Bool	**Tipo dei booleani**
$\tau \to \tau$	**Tipo delle funzioni**

#2: Sintassi del linguaggio

FUN: Lambda calcolo tipato semplice con booleani

e ::=	**Espressioni**
x	**Variabili**
fun x: τ = e	**Funzioni**
Apply(e, e)	**Applicazione**
true	**Costante true**
false	**Costante false**
if e then e else e	**Condizionale**

#2: Sintassi del linguaggio (formato tradizionale)

FUN: Lambda calcolo tipato semplice con booleani

e ::=	**Espressioni**
x	**Variabili**
λ x:τ. e	**Funzioni**
e e	**Applicazione**
true	**Costante true**
false	**Costante false**
if e then e else e	**Condizionale**

#2: Valori

V::=	**Valori**
fun x:τ_1 = e: τ_2	**Funzioni**
true	**Valore true**
false	**Valore false**

#3: Composizionlità del type checker

Definamo le regole del type checker induttivamente sulla struttura sintattica del linguaggio

Problema:

$$\frac{x{:}\,\tau_1, ?\,?\,?\,?\,?}{fun\ x{:}\,\tau_1 = e{:}\,\tau_1 \rightarrow \tau_2}$$

Il tipo del corpo e della funzione dipende dal tipo del parametro formale x. Come teniamo conto di questa associazione?

#3: tipi delle variabili

Ambiente dei tipi.

L'ambiente dei tipi è una funzione (di dominio finito) che associa nomi a tipi. Noi scriveremo:

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2 \ \ldots xk : \tau_k$$

per indicare la funzione

$$\Gamma(x_i) = \tau_i$$

che associa il tipo τ_i al valore x_i

La notazione

$$\Gamma, x : \tau$$

Sarà usata per indicare l'estensione della funzione Γ con l'associazione $x : \tau$

$$(\Gamma, \mathrm{x} : \tau)(\mathrm{x}) = \tau \text{ e } (\Gamma, \mathrm{x} : \tau)(\mathrm{y}) = \Gamma(y) \ per \ y \ ! = x$$

Esempio

$$\Gamma = x{:}\,\tau, y{:}\,\tau'$$

$$\Gamma(x) = \tau$$

$$\Gamma(z) = \mathit{undefined}$$

Esempio

$$\Gamma = x{:}\,\tau, y{:}\,\tau'$$

$$\Gamma(x) = \tau$$

$$\Gamma' = \Gamma, z{:}\,\tau_2$$

$$\Gamma'(z) = \tau_2$$

Esempio

L'ambiente dei tipi vuoto $\emptyset$ non contiene alcun legame di tipo per le variabili

$$\emptyset(x) = \boldsymbol{undefined}$$
$$\boldsymbol{per\ tutti\ i\ nomi\ x}$$

#3: tipi delle variabili

Ambiente dei tipi.

L'ambiente dei tipi è una funzione (di dominio finito) che associa nomi a tipi. Noi scriveremo:

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2 \ \ldots xk : \tau_k$$

OSSERVAZIONE:
L'ambiente dei tipi è una funzione pertanto
è assicurata l'unicità dell'associazione tra nomi e tipi

#3: tipi delle variabili

Ambiente dei tipi.

L'ambiente dei tipi è una funzione (di dominio finito) che associa nomi a tipi. Noi scriveremo:

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

Definizione altermativa (equivalente)
Ambiente di tipo è una sequenza di coppie

$$x_1 : \tau_1, x_2 : \tau_2 \ \ldots x_k : \tau_k$$

in cui nomi x_i sono tutti distinti

$$x_i \neq x_j$$

Giudizio di tipo

Supponiamo che Γ sia un ambiente di tipo.
La notazione:

$$\boldsymbol{\Gamma \vdash e : \tau}$$

È usata per indicare che l'espressione e ha tipo τ
nell'ambiente di tipo Γ.

INTUIZIONE: L'ambiente dei tipi tiene conto dei legami tra i nomi che compaiono nel programma (l'espressione e) e il loro tipo

Giudizio di tipo

Supponiamo che Γ sia un ambiente di tipo.
La notazione:

$$\boldsymbol{\Gamma \vdash e : \tau}$$

È usata per indicare che l'espressione e ha tipo τ nell'ambiente di tipo Γ.

Le regole sono applicata dal compilatore in fase di analisi statica. L'ambiente dei tipi nel gergo dei compilatori è chiamato Tabella dei Simboli.

Regole di tipo

Definiamo il sistema per il controllo dei tipi (type checker) per il nostro Lambda calcolo tipato semplice

$$\Gamma \vdash true : Bool$$

$$\Gamma \vdash false : Bool$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Una variabile ha il tipo a lei associato nell'ambiente dei tipi.

Condizionale

$$\frac{\Gamma \vdash e : Bool, \Gamma \vdash e_1 : \tau, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \tau}$$

Tipi per le funzioni

- Quale è il tipo di una funzione?
- Il costruttore di tipo $\tau 1 \rightarrow \tau 2$ descrive il tipo della funzioni che prendono in ingresso un argomento di tipo $\tau 1$ e restituiscono un risultato di tipo $\tau 2$”

Funzioni

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

Dato che il tipo del parametro formale x (τ_1) è noto:
(i) il tipo delle occorrenze del parametro x nel corpo della funzione saranno associate a τ_1
(ii) il tipo del risultato della funzione sarà il tipo del corpo della funzione.

Intuizione
La funzione richiede un parametro di tipo τ_1 e restituisce come risultato un valore di tipo τ_2.
Notazione: $\tau_1 \to \tau_2$

Funzioni: regole di visibilità

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

Intuizione (informatica)
La definizione del parametro formale x e del suo tipo è una specie di dichiarazione dinamica.
La dichiarazione del parametro x sovrascrive e annulla le precedenti dichiarazionie per x.
La portata della dichiarazione del parametro x è il corpo della funzione

Esempio

$$\frac{x{:}\,Bool \vdash x{:}\,Bool}{\emptyset \vdash fun\; x{:}\,Bool = x{:}\,Bool \rightarrow\; Bool}$$

Funzioni: Controllo di tipo

DEFINIZIONE

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

INVOCAZIONE

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

Esempio

$$\dfrac{\dfrac{-}{\Gamma \vdash f: Bool \to Bool,} \quad \dfrac{\Gamma \vdash false: Bool, \Gamma \vdash true: Bool, \Gamma \vdash false: Bool}{\Gamma \vdash if\ false\ then\ true\ else\ false: Bool}}{\dfrac{\Gamma = f: Bool \to Bool \vdash Apply(f, if\ false\ then\ true\ else\ false): Bool}{\emptyset \vdash fun\ f: Bool \to Bool = \ Apply(f, if\ false\ then\ true\ else\ false): Bool \to Bool \to Bool}}$$

Esercizio

Dimostrare la deribazione di tipo seguente

$$f : Bool \to Bool \vdash fun\ x : Bool = Apply\big(f, (if\ x\ then\ true\ else\ false)\big) : Bool \to Bool$$

Type Safety

La correttezza (type safety) del sistema di tipo del lambda calcolo Tipato è espressa formalmente da queste due proprietà

Progresso: Se $\emptyset \vdash e : \tau$ allora **e** è un valore oppure $e \rightarrow e'$ per una qualche espressione **e'**

Progresso: Una espressione senza variabili libera ben tipata non si blocca a run-time

Conservazione: Se $\Gamma \vdash e : \tau$ e $e \rightarrow e'$ allora $\Gamma \vdash e' : \tau$

Conservazione: I tipi sono preservati dalle regole di esecuzione

Come si dimostra?

Passi soliti

Si parte con dimostrare il lemma di inversione e il lemma delle forme canoniche

Inversion lemmata

Se $\Gamma \vdash true : \tau$ allora $\tau = Bool$.

Se $\Gamma \vdash false : \tau$ allora $\tau = Bool$.

Se $\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 : \tau$ allora

$\Gamma \vdash e_1 : Bool$ e $\Gamma \vdash e_2 : \tau$ e $\Gamma \vdash e_3 : \tau$

Se $\Gamma \vdash x : \tau$ allora $\Gamma(x) = \tau$

Se $\Gamma \vdash fun\ x{:}\tau_1 = e : \tau$ allora

$\tau = \tau_1 \rightarrow \tau_2$ per qualche τ_2 tale che $\Gamma, x : \tau_1 \vdash e : \tau_2$

Se $\Gamma \vdash Apply(e_1, e_2) : \tau_2$ allora esite un tipo τ_1 tale che

$\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2$, $\Gamma \vdash e_2 : \tau_1$

Lemma: forme canoniche

Se V è un valore di tipo Bool allora V è true oppure false.

Se V è un valore di tipo $\tau_1 \rightarrow \tau_2$ allora V è della forma
fun x: τ_1 = e: τ_2

Teorema del progresso

Progresso: Se $\emptyset \vdash e : \tau$ allora **e** è un valore oppure $e \rightarrow e'$
per una qualche espressione **e'**

Come si dimostra?
Per induzione sulle derivazioni di tipo.

I casi di base (costanti booleane e condizionale) sono identici ai casi di base del sistema per le espressioni aritmetiche.
Il caso delle variabili è banale.
Il caso dell'astrazione funzionale è immediato, poiché le funzioni sono valori.

Teorema del progresso

Progresso: Se $\emptyset \vdash e: \tau$ allora **e** è un valore oppure $e \rightarrow e'$
per una qualche espressione **e'**

Casi induttivi (consideriamo solo il caso dell'applicazione)
Applicazione: e = Apply(e_1, e_2), $\emptyset \vdash e_1: \tau_1 \rightarrow \tau_2$ $\emptyset \vdash e_2: \tau_1$ **.**
Per l'ipotesi induttiva possiama affermare che,
e_1 è un valore o può fare un passo di valutazione, e così pure e_2.
Se le espressioni possono fare un passo applichiamo le regole di riduzione dell'applicazione e terminiamo.
Se invece sono entrambi valori, allora per il lemma delle forme canoniche abbiamo che e_1 deve essere della forma

$$fun\ x: \tau_1 = e': \tau_1 \rightarrow \tau_2$$

Pertanto applichiamo la regola di beta riduzione.

Teorema della conservazione

Conservazione: Se $\Gamma \vdash e : \tau$ e $e \to e'$ allora $\Gamma \vdash e' : \tau$

Si dimostra sempore per induzione strutturale sulle regole di tipo.
Quale è il caso difficile?

Teorema della conservazione

Conservazione: Se $\Gamma \vdash e : \tau$ e $e \to e'$ allora $\Gamma \vdash e' : \tau$

Si dimostra sempore per induzione strutturale sulle regole di tipo.
Quale è il caso difficile?

Applicazione: e = Apply(e1, e2). Quindi vale che

$$\Gamma \vdash e : \tau$$
$$\Gamma \vdash e_1 : \tau_1 \to \tau_2$$
$$\Gamma \vdash e_2 : \tau_2$$
$$\tau = \tau_2$$
$$e \to e'$$

Si deve pertanto dimostrare che

$$\Gamma \vdash e' : \tau_2$$

Teorema della conservazione

Applicazione: e = Apply(e1, e2). Quindi vale che

$$\Gamma \vdash e : \tau$$
$$\Gamma \vdash e_1 : \tau_1 \to \tau_2$$
$$\Gamma \vdash e_2 : \tau_2$$
$$\tau = \tau_2$$
$$e \to e'$$

Si deve pertanto dimostrare che

$$\Gamma \vdash e' : \tau_2$$

Applichiamo il lemma dell'inversione:. Questo comporta che:

$$e_1 = fun\ x : \tau_1 = e_3$$

$$e' = e_3\ \{x = e_2\}$$

Problema
non siamo più sintattici!!
Si deve gestire la sostituzione

Substitution lemma

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

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

Come si dimostra?
Per induzione sulla derivazione di

$$\Gamma, x{:}\,\tau_1 \vdash e{:}\ \tau$$

I casi più interessanti sono quelli per le variabili e le astrazioni funzionali.
Trovate la dimostrazione completa sulle note didattiche

Teorema della conservazione

Esercizio:

Conlcudere la dimostrazione del teorema.

Estensioni

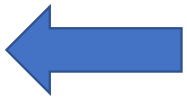
Abbiamo considerato un lambda calcolo tipo con costanti booleane e condizionale

Possiamo estendere il linguaggio con altri costrutti per farlo diventare un vero linguaggio di programmazione

Costanti numeriche

Tipi semplici

τ ::=	**Tipi**
Bool	**Tipo dei booleani**
Nat	**Tipo dei natutali**
$\tau \to \tau$	**Tipo delle funzioni**



Sintassi

FUN: Lambda calcolo tipato semplice con booleani, naturali e operazioni numeriche

e ::=	Espressioni
x	Variabili
fun x: τ = e	Funzioni
Apply(e, e)	Applicazione
true	Costante true
false	Costante false
n	Costanti numeriche
e $\oplus$ e	Operazioni binarie
if e then e else e	Condizionale

n ::= 0, 1, 2 ……

Assunzione: Sono noti i tipi delle operazioni primitive $\otimes: \tau_1 \times \tau_2 \rightarrow \tau$

Regole di tipo

$$\Gamma \vdash n : Nat$$

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

Esempio di derivazione

$$\dfrac{\dfrac{\Gamma \vdash x : int \quad \Gamma \vdash 3 : int}{<: int \times int \to bool}}{\Gamma \vdash x < 3 : bool} \qquad \dfrac{\dfrac{\Gamma \vdash x : int \quad \Gamma \vdash 1 : int}{+ : int \times int \to int}}{\Gamma \vdash x + 1 : int} \qquad \dfrac{\dfrac{\Gamma \vdash y : int \quad \Gamma \vdash 1 : int}{+ : int \times int \to int}}{\Gamma \vdash y + 1 : int}$$

$$\Gamma \vdash \textbf{if } x < 3 \textbf{ then } x + 1 \textbf{ else } y + 1 : int$$

$$\Gamma = x : int, y; int$$

Esempio derivazione con errore

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash x : \mathit{int} \quad \Gamma \vdash 3 : \mathit{int}}{<: \mathit{int} \times \mathit{int} \to \mathit{bool}}}{\Gamma \vdash x < 3 : \mathit{bool}} \quad \dfrac{\dfrac{\Gamma \vdash x : \mathit{int} \quad \Gamma \vdash 1 : \mathit{int}}{+ : \mathit{int} \times \mathit{int} \to \mathit{int}}}{\Gamma \vdash x + 1 : \mathit{int}} \quad \dfrac{\dfrac{\Gamma \vdash y : \mathit{real} \quad \Gamma \vdash 1.0 : \mathit{real}}{\boxplus : \mathit{real} \times \mathit{real} \to \mathit{real}}}{\Gamma \vdash y \boxplus 1.0 : \mathit{real}}}{\Gamma \vdash \textbf{if } x < 3 \textbf{ then } x + 1 \textbf{ else } y \boxplus 1.0 : \textbf{error}}$$

$$\Gamma = x : \mathit{int}, y; \mathit{real}$$

Nota: La regola di tipo del condizionale richiede che il ramo then e il ramo else abbiamo lo stesso tipo

Dichiarazioni locali

e ::= ….

let x = e_1 in e_2 : τ_2

Regole di valutazione

$$\frac{e_1 \to e'}{let\ x = e_1 in\ e_2 : \tau_2 \to let\ x = e' in\ e_2 : \tau\_2}$$

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

Dichiarazioni locali

e ::= ….

let x = e_1 in e_2 : τ_2

Regola di tipo

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

Ricorsione

Si può facilmente dimostrare che nel lambda calcolo tipato semplice che abbiamo introdotto tutti i programmi terminano. Ad esempio Il combinatore Ω che abbiamo introdotto non è tipabile

Il prossimo passo è estendere il linguaggio con un costrutto linguistico per gestire la ricorsione che sia tipabile (con tutte le proprietà che ci aspettiamo)

Verso la ricorsione con un esempio

Consideriamo la seguente funzione nel nostro lambda calcolo tipati semplice

```
aux  = fun f:Nat→Bool =
          fun x :Nat. =
            if (isZero x) then true else
              if (isZero (pred x)= then false else
                f (pred (pred x))

aux : (Nat→Bool ) → Nat→Bool
```

Intuizione:
aux è un *generatore: se viene*
applicata ad una funzione iE che approssima
Il comportamento di isEven fino a n,
iE k (con k <=n) restituisce
valore calcolato da isEven k, allora
aux iE k rstistuisce una migliore
Approssimazione di isEven fino a k+2

isEven = fix aux

Applicando fix a aux si ottiene il limite delle approssimazioni
Ovvero il punto fisso

I passi verso la definizione di fix

Sintassi espressioni
e::= …. | fix e

fix e : definizione ricorsiva

Regole di valutazione

$$\frac{}{\boldsymbol{fix}\ (\boldsymbol{fun}\ \boldsymbol{x}\colon \boldsymbol{\tau} = \boldsymbol{e}) \rightarrow \boldsymbol{e}[\boldsymbol{x} = (\boldsymbol{fix}\ (\boldsymbol{fun}\ \boldsymbol{x}\colon \boldsymbol{\tau} = \boldsymbol{e}))}$$

$$\frac{\boldsymbol{e} \rightarrow \boldsymbol{e}'}{\boldsymbol{fix}\ \boldsymbol{e} \rightarrow \boldsymbol{fix}\ \boldsymbol{e}'}$$

Regole di tipo

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{e}\colon\ \boldsymbol{\tau} \rightarrow \boldsymbol{\tau}}{\boldsymbol{\Gamma} \vdash \colon \boldsymbol{\tau}}$$

I passi verso la definizione di fix

Sintassi espressioni
e::= …. | fix e

fix e : definizione ricorsiva

Regole di valutazione

$$\frac{}{fix\ (fun\ x{:}\tau = e) \rightarrow e[x = (fix\ (fun\ x{:}\tau = e))}$$

Regole di tipo

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

$$\frac{e \rightarrow e'}{fix\ e \rightarrow fix\ e'}$$

Lambda calcolo tipato semplice + fix = PCF

Programming Computable Functions (PCF) è un linguaggio funzionale tipato introdotto da Gordon Plotkin in 1977

Una sintassi più semplice

$$\textbf{letrec}\ x{:}\tau = e\ \textbf{in}\ e' \equiv \textbf{let}\ x = \textbf{fix}\ (\textbf{fun}\ x{:}\tau = e)\ \textbf{in}\ e'$$

```
letrec isEven : Nat→Bool =
  fun x:Nat =
     if (isZero x) then true else
       if (iszero (pred x)) then false
           else isEven (pred (pred x)) in
  isEven 7;
```

Ricorsione in Ocaml: osservazione

- Nel linguaggio Ocaml come vedremo tra poco abbiamo due primitive per definire funzioni
- La primitiva **fun** permette di definire funzioni non ricorsive
  - (**fun** x -> x + 1)
  - let plusOne = fun x -> x+1;;
- La primitive **letrec** permette di definire funzioni ricorsive
  - **let rec** fact x =
    if x <= 1 then 1 else x * fact (x - 1)

Nota

- Funzioni anonime (come fun(x: τ)=e del nostro lambda calcolo tipato semplice) sono presenti in molti linguaggi di programmazione, ad esempio Java a partire dalla versione 8).
- Funzioni anonime in
  - Scala: (x: Type) => e
  - Java 8: x -> e (senza dichiarare il tipo)
  - Haskell: \x -> e oppure \x::Type -> e

Conclusioni

- Il calcolo lambda è un modello fondazionale essenziale per comprendere la teoria della computazione
- Utile per capire I linguaggi di programmazione: sistemi dei tipi,, terminazione, sistemi di verifica, etc. possono essere analizzati nel lambda calcolo per pi scalare a linguaggi di programmazione completi

CONSIDERAZIONI

- Tipi sono **specifiche di comportamento**: il sitema dei tipi che abbiamo considerato specifca il comportamento input-output delle funzioni e il typechecking verifca l'adeguatezza del comportamento input-output
- Tip e typechecking possono essere usati per verificare **proprietà di programmi**
  - proprietà di **segretezza e autenticità** dei protocolli di sicurezza
  - proprietà comportamentali (**assenza di deadlock**) in sistemi concorrenti

Tipi e Ingegneria del Software

- Typechecking fornisce delle garanzie di correttezza parziali e inevitabilmente rifiuta alcuni programmi corretti.
- La maggior parte delle proprietà interessanti **non possono essere automaticamente verificate** quindi i tipi possono solo dare un'approssimazione della correttezza.

Tipi e Ingegneria del Software

Perché l'analisi esatta di programmi è impossibile

- Alcuni problemi sono **indecidibili**: è impossibile costruire un algoritmo che risolva istanze arbitrarie del problema (teoria della calcolabilità)
- **Halting problem** (**problema della fermata**): è possible decidere se un dato programma si ferma (termina) quando gli viene presentato un certo input?
- **Turing** ha mostrato che il **problema della fermata non è decidibile**

Proprietà di programmi e decidibilità

- Alcune proprietà interessanti
  - Il programma termina su tutti in dati in ingresso?
  - Quanto grande può diventare la memoria dinamica (heap)?
  - Viene garantita la privacy dell'informazione?
- Le proprietà che coinvolgono la previsione esatta del comportamento del programma sono generalmente indecidibili (Teorema di Rice, 1953)
- Non possiamo semplicemente eseguire il programma e vedere cosa succede, perché non c'è un limite superiore al tempo di esecuzione dei programmi.

Un viaggio (non spericolato) nell'halting problem

Supponiamo che P sia una funzione scritta in un qualche linguaggio di programmazione:

```
void P(String S);
```

Sia H l'implementazione di un algoritmo A codificata come una opportune funzione

```
boolean H(String X, String S);
```

… Halting problem

Utilizzando H possiamo definire una funzione Q così definita
Q prende come parametro il nome di una funzione W
Q utilizza H per determinare se la funzione W termina quando viene invocata con il proprio nome
Se la funzione W termina allora Q esegue un ciclo infinito altrimenti Q termina

```
void Q(String W) {
     if (H(W,W)) {
          while (true) {}
     }
}
```

…. Halting problem

Mandiamo in esecuzione Q con il proprio "codice" come input.

`Q("void Q(String W) {…}")`

- Q invoca H(Q,Q) che restituisce un valore booleano (true o false)
- se H restituisce true allora:
  - per costruzione di H, Q termina con input Q.
  - per costruzione di Q, Q cicla con Q come input.

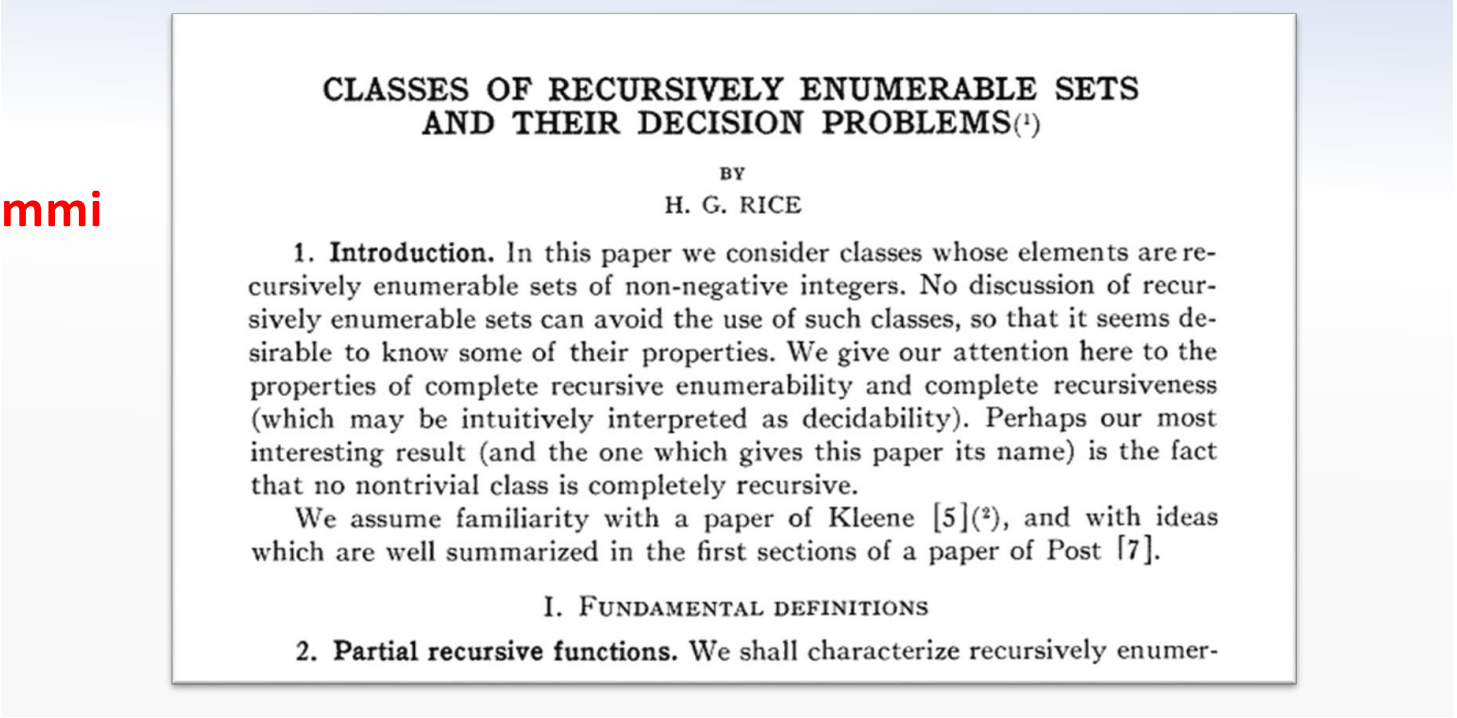
  CONTRADDIZIONE!!!!!
- Se H restituisce false allora:
  - per costruzione di H, Q cicla con Q come input
  - per costruzione di Q,, Q termina on input Q.

  CONTRADDIZIONE!!!!!

Q non è definibile!!!!.

Teorema di Rice

Proprietà non banali del
Comportamento di programmi
Non sono decidibili.



CLASSES OF RECURSIVELY ENUMERABLE SETS AND THEIR DECISION PROBLEMS(1)

BY

H. G. RICE

1. **Introduction.** In this paper we consider classes whose elements are recursively enumerable sets of non-negative integers. No discussion of recursively enumerable sets can avoid the use of such classes, so that it seems desirable to know some of their properties. We give our attention here to the properties of complete recursive enumerability and complete recursiveness (which may be intuitively interpreted as decidability). Perhaps our most interesting result (and the one which gives this paper its name) is the fact that no nontrivial class is completely recursive.

We assume familiarity with a paper of Kleene [5](2), and with ideas which are well summarized in the first sections of a paper of Post [7].

I. FUNDAMENTAL DEFINITIONS

2. **Partial recursive functions.** We shall characterize recursively enumer-

La soluzione adottata nei sistemi di tipo

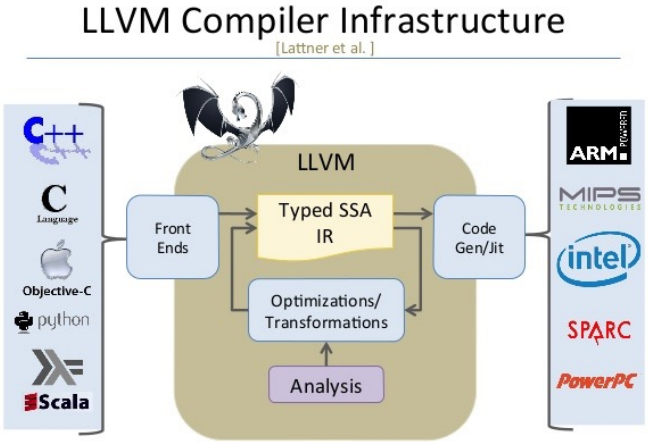
Le soluzioni approssimate el problema possono essere decidibili!

L'approssimazione deve preservare i comportamenti (teorema del progresso e della conservazione).

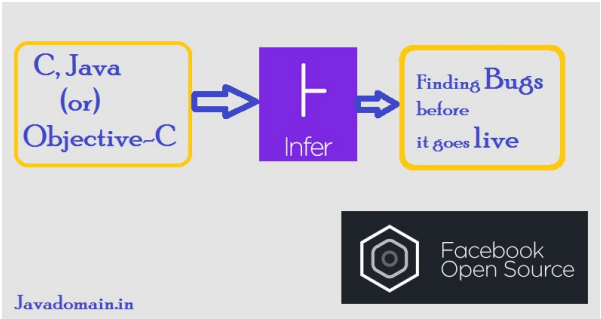
Non è tutto perduto!!!

La non decidibilià di proprietà di programmi è uno scoglio non superabile …

…. l'analisi statica (inclusi i sistemi di tipi) è un'area enorme e di successo



FLOWDROID



INFER

CODE ANALYZER

La ricerca nel campo dei linguaggi di programmazione

Una tendenza importante nello sviluppo dei linguaggi di programmazione è stata l'inclusione di sistemi di tipi più sofisticati nei linguaggi di programmazione

Lo studio dei sistemi di tipi ci permette di comprendere come potrebbe essere la prossima generazione di linguaggi di programmazione