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 inizialmente una variante costruita su valori booleani (oltre che su valori di tipo funzione, già visti).

#1: Sintassi dei tipi

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

Esempi di tipi sintatticamente corretti:

Bool   Bool → Bool   (Bool → Bool) → Bool

(Bool → Bool) → (Bool → Bool)

Attenzione: → associa a destra!! quindi:

Bool → Bool → Bool = Bool → (Bool → Bool)

#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	**Funzioni**
true	**Valore true**
false	**Valore false**

#3: Semantica (cose già viste… riscritte con la nuova sintassi)

$$Apply\ (fun\ x{:}\,\tau = e_1\ ,\ e_2) \rightarrow e_1\{x \coloneqq e_2\}$$

$$\frac{e_1 \rightarrow e'}{Apply(e_1, e_2) \rightarrow Apply(e', e_2)} \qquad \frac{e_2 \rightarrow e'}{Apply(e_1, e_2) \rightarrow Apply(e_1, e')}$$

$$\frac{e \rightarrow e'}{fun\ x{:}\,\tau = e \rightarrow fun\ x{:}\,\tau.\,e'}$$

$$\frac{e_1 \rightarrow e_4}{if\ e_1\ then\ e_2\ else\ e_3 \rightarrow if\ e_4\ then\ e_2\ else\ e_3}$$

$$if\ true\ then\ e_2\ else\ e_3 \rightarrow e_2$$

$$if\ false\ then\ e_2\ else\ e_3 \rightarrow e_3$$

#3: Semantica (cose già viste… riscritte con la nuova sintassi)

$$Apply\ (fun \dots{:}\tau = e_1\ ,\ e_2\ ) \to e_1\{x \coloneqq e_2\}$$

β-riduzione senza strategie (non deterministica)

$$\frac{e_1 \to e'}{Apply(e_1, e_2) \to Apply(e', e_2)} \qquad \frac{e_2 \to e'}{\dots e_2) \to Apply(e_1, e')}$$

$$\frac{e \to e'}{fun\ x{:}\tau = e \to fun\ x{:}\tau . e'}$$

espressione condizionale

$$\frac{e_1 \to e\dots}{if\ e_1\ then\ e_2\ else\ e_3 \to i\dots}$$

$$\dots ue\ then\ e_2\ else\ e_3 \to e_2$$

$$\dots alse\ then\ e_2\ else\ e_3 \to e_3$$

#4: Composizionalità del type checker

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

Problema:

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

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:

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

per indicare la funzione

$$\boldsymbol{\Gamma(x_i) = \tau_i}$$

che associa il tipo τ_i al valore x_i

La notazione

$$\boldsymbol{\Gamma, x : \tau}$$

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

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

Esempio

$$\mathbf{\Gamma} = \boldsymbol{x}\colon \boldsymbol{\tau}, \boldsymbol{y}\colon \boldsymbol{\tau}'$$

$$\mathbf{\Gamma}(\boldsymbol{x}) = \boldsymbol{\tau}$$

$$\mathbf{\Gamma}(\boldsymbol{z}) = \boldsymbol{undefined}$$

Esempio

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

$$\boldsymbol{\Gamma(x) = \tau}$$

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

$$\boldsymbol{\Gamma'(z) = \tau_2}$$

Esempio

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

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

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 (Symbol Table)

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

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

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

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

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 (nell'ambiente esteso $\Gamma, x: \tau_1$) le precedenti dichiarazioni per x (che sono in Γ). La portata (scope) della dichiarazione del parametro x è il corpo della funzione. Infatti $\Gamma, x: \tau_1$ viene usato solo per tipare il corpo della funzione.

Esempio

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

Chiamata di funzione

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

Esempio

$$\dfrac{\dfrac{\dfrac{\Gamma(f) = Bool \rightarrow Bool}{\Gamma \vdash f : Bool \rightarrow Bool} \qquad \dfrac{\Gamma \vdash false : Bool \quad \Gamma \vdash true : Bool \quad \Gamma \vdash false : Bool}{\Gamma \vdash if\ false\ then\ true\ else\ false : Bool}}{\Gamma = f : Bool \rightarrow Bool \vdash Apply(f, if\ false\ then\ true\ else\ false) : Bool}}{\emptyset \vdash {\color{red} fun\ f : Bool \rightarrow Bool = Apply(f, if\ false\ then\ true\ else\ false)} : (Bool \rightarrow Bool) \rightarrow Bool}$$

Type Safety

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

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

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

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

Conservazione: I tipi sono preservati dalle regole di esecuzione

Teorema del progresso

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

Come si dimostra?
Per induzione sulle derivazioni di tipo.

I casi di base (costanti booleane) 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 $e \to e'$ per una qualche espressione **e’**

Casi induttivi (consideriamo solo il caso dell’applicazione)
Applicazione: e = Apply(e_1, e_2), $\emptyset \vdash e_1{:}\,\tau_1 \to \tau_2$ $\emptyset \vdash e_2{:}\,\tau_1$.
Per l'ipotesi induttiva possiamo 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 abbiamo che e_1 deve essere della forma

$$fun\ x{:}\,\tau_1 = e'{:}\,\tau_1 \to \tau_2$$

Pertanto applichiamo la regola di beta riduzione.

Teorema della conservazione

Conservazione: Se $\boldsymbol{\Gamma} \vdash \boldsymbol{e}\colon \boldsymbol{\tau}$ e $\boldsymbol{e} \rightarrow e'$ allora $\boldsymbol{\Gamma} \vdash \boldsymbol{e}'\colon \boldsymbol{\tau}$

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

Teorema della conservazione

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

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

Applicazione: e = Apply(e_1, e_2). Quindi vale che

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

Si deve pertanto dimostrare che

$$\boldsymbol{\Gamma \vdash e' : \tau_2}$$

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

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

$\Gamma \vdash e_1 : \tau_1 \to \tau_2$ comporta che:

$$e_1 = fun\ x : \tau_1 = e_3 \qquad \Gamma, x : \tau_1 \vdash e_3 : \tau_2$$

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

Problema: abbiamo dimostrato $e_3 : \tau_2$ e invece la semantica dell'applicazione ci porta in $e_3\ \{x = e_2\}$, non in e_3
… 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

Estensioni (cenni)

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

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**	**n ::= 0, 1, 2 ……**
e $\bigoplus$ e	**Operazioni binarie**	**$\bigoplus$::= + - * / %**
if e then e else e	**Condizionale**	

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

Regole di tipo

$$\boldsymbol{\Gamma \vdash n : Nat}$$

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

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

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

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 tipato semplice

let 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 restistuisce una migliore Approssimazione di isEven fino a k+2*

Aggiungiamo al linguaggio un costrutto fix tale che:

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

$$\frac{e \to e'}{fix\, e \to fix\, e'}$$

$$\frac{}{fix\,(fun\, f{:}\,\tau = e) \to e[f = (fix\,(fun\, f{:}\,\tau = e))}$$

Regole di tipo

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

I passi verso la definizione di fix

Lambda calcolo tipato semplice + fix = PCF

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

Sintassi espressioni
e::= …. | fix e

Regole di valutazione

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

$$\frac{}{fix\ (fun\ f{:}\tau = e) \to e[f = (fix\ (fun\ f{:}\tau = e))}$$

Regole di tipo

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

Una sintassi più semplice (zucchero sintattico)

$\boldsymbol{let\ rec}\ x : \tau = e\ \boldsymbol{in}\ e'$

corrisponde a: $\boldsymbol{let}\ x = \boldsymbol{fix}\ (\boldsymbol{fun}\ x : \tau = e)\ \boldsymbol{in}\ e'$

```
let rec 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 **let rec** permette di definire funzioni ricorsive
  - **let rec** fact x =
    if x <= 1 then 1 else x * fact (x - 1)

CONSIDERAZIONI

Conclusioni

- Il lambda calcolo 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 poi scalare a linguaggi di programmazione completi

Tipi e Ingegneria del Software

- 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
- Tipi 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

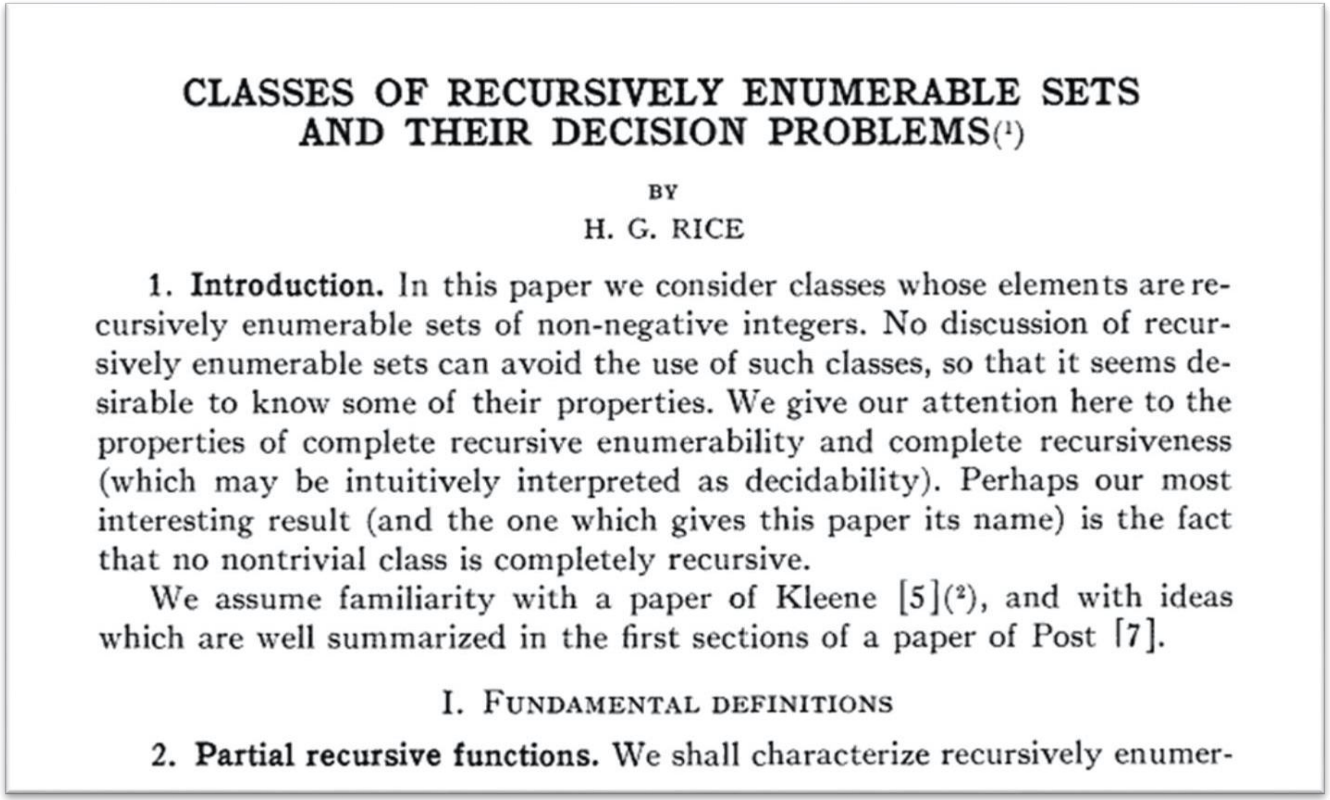
- Typechecking fornisce delle garanzie di correttezza parziale e inevitabilmente rifiuta alcuni programmi corretti.
- La maggior parte delle proprietà interessanti **non possono essere automaticamente verificate** quindi i tipi possono solo dare un'approssimazione della correttezza.

Proprietà di programmi e decidibilità

- Alcune proprietà interessanti:
  - Il programma termina su tutti in dati in ingresso?
  - Quanto grande puà 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.

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-

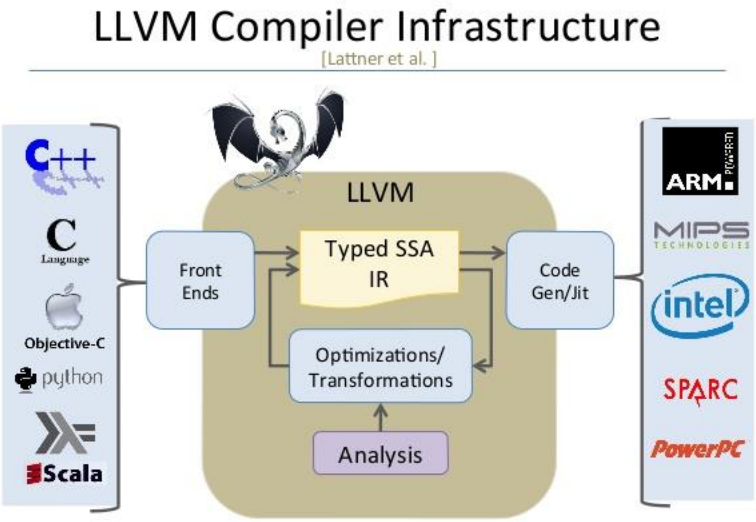
La soluzione adottata nei sistemi di tipo

Le soluzioni approssimate el problema possono essere decidibili!

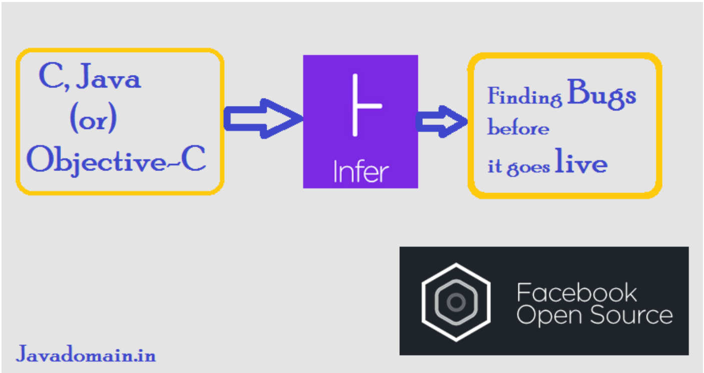
-> si possono definire sistemi di tipi per le proprietà da studiare

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

ESEMPI DI ANALIZZATORI STATICI (BASATI SU SISTEMI DI TIPI E APPROCCI SIMILI):



F
L
O
W
D
R
O
I
D



INFER

ROSLYN CODE ANALYZER