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**
e $\oplus$ e	**Operazioni binarie**
if e then e else e	**Condizionale**

n ::= 0, 1, 2 ……

$\oplus$::= + - * / %

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

Regole di tipo

$$\Gamma \vdash n : Nat$$

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

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 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 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

Tipi e Ingegneria del Software

- 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.

Tipi e Ingegneria del Software

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.

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-

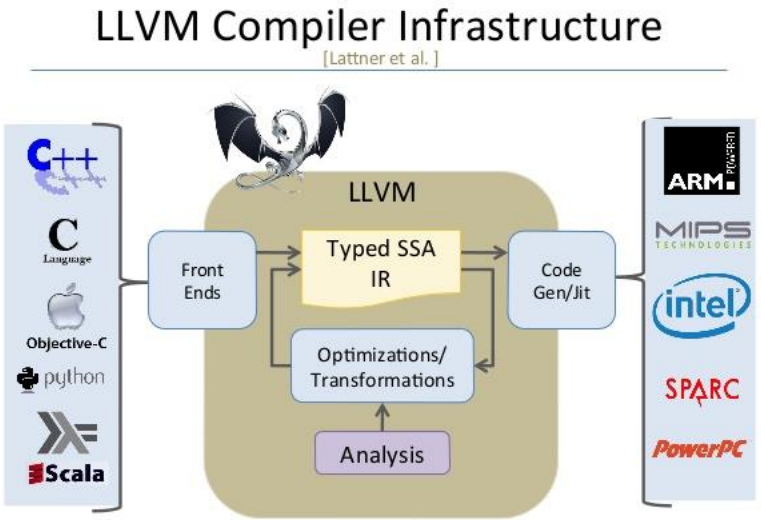
La soluzione adottata nei sistemi di tipo

Le soluzioni approssimate el problema possono essere decidibili!

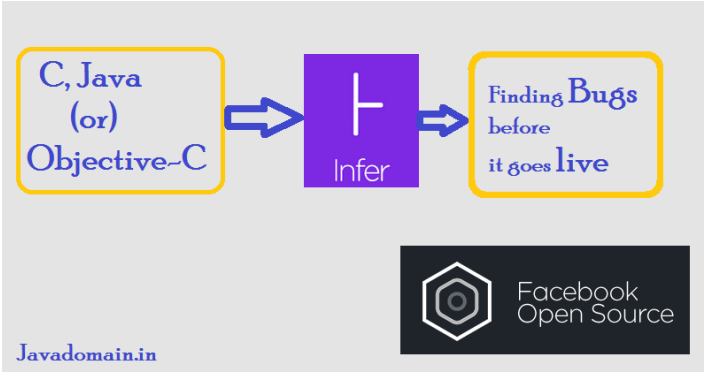
-> si possono definire sistemi di tipi per le proprietà da studiare

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

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



FLOWDROID



INFER

ROSLYN CODE ANALYZER