

Semantica di P_0

Definizione 3.2

Un enunciamento B è una funzione:

$$B : \text{veri/falsi} \rightarrow \{0,1\}$$

proposizionali

Definizione 3.3

Un enunciamento proposizionale B è esteso per induzione ad una valutazione \bar{B} del linguaggio P_0 nel modo seguente

- $\bar{B}(p) = B(p) \quad \forall p$
- $\bar{B}(d \rightarrow \beta) = \begin{cases} 0 & \text{se e solo se } \bar{B}(d) = 1 \text{ e } \bar{B}(\beta) = 0 \\ 1 & \text{altrimenti} \end{cases}$
- $\bar{B}(\neg d) = 1 - \bar{B}(d)$

Definizione 3.4

Une flif d di P_0 è detta vera:

- Tautologie: $\forall B \Rightarrow \bar{B}(d) = 1$
- Soddisfacibile: $\exists B | \bar{B}(d) = 1$
- Contraddittorio: $\exists B | \bar{B}(B) = 1 \quad \forall B \in \Gamma$

Proposizione 3.13

Une flif d di P_0 è una tautologia se e solo se $\bar{B}(d) = 1$ è contraddittorio

Dimostrazione

Se utilizziamo le def 3.3 notiamo che $\bar{B}(\neg d) = 1 - \bar{B}(d) = 0$

Proposizione 3.14

Se d e $d \rightarrow \beta$ sono tautologie $\Rightarrow B$ è una tautologia

$$\bar{B}(d) = 1 \quad \bar{B}(d \rightarrow \beta) = 1 \Rightarrow 1 \Rightarrow \bar{B}(\beta) = 1$$

Osservazione

$$\bar{B}(\beta) = 1 \text{ sempre se } \bar{B}(d \rightarrow \beta) = 1$$

Lemme 3.1

Dato un fbf λ di P_0 , $\bar{B}(\lambda)$ dipende solo dal valore assegnato da B alle variabili proposizionali in λ (perché B è una funzione).

Teorema

Dato fbf λ di P_0 , è decidibile se λ è una tautologia \Rightarrow

Dimostrazione

Si verifica facendo le tavole di verità di λ

Definizione 3.5 (conseguenze tautologiche)

Dato uno fbf λ di P_0 , è un insieme Γ di fbf di P_0 si dice che λ è conseguenza tautologica di Γ se e solo se per ogni assegnamento proposizionale B si ha che:

Se per ogni fbf $\beta \in \Gamma$ vale $\bar{B}(\beta) = 1$ allora $\bar{B}(\lambda) = 1$

$\forall \beta \in \Gamma \mid \bar{B}(\beta) = 1 \Rightarrow \bar{B}(\lambda) = 1$ e scriviamo $\Gamma \models \lambda$

Teorema di correttezza di P_0 (3.5)

Siamo Γ un insieme di fbf e λ uno fbf di P_0

Se $\Gamma \vdash_{P_0} \lambda$ allora $\Gamma \models \lambda$

"Se Γ deduce λ allora λ è una conseguenza tautologica"

Corollario 3.3

P_0 è consistente

Proposizione 3.15

Se Γ è un insieme di fbf di P_0 , se Γ è soddisfacibile allora Γ è consistente

Lemme 3.3 (soddisfacibilità)

Dato Γ un insieme di fbf di P_0 , se Γ è consistente allora

Γ è soddisfacibile

Γ soddisfacibile $\Leftrightarrow \Gamma$ consistente

Teorema 3.6 (di completezza)

Sia Γ un insieme di filf di P_0 ed una filf di P_0
se $\Gamma \models d$ segue $\Gamma +_{P_0} d$

Verso appunto del teorema di correttezza

Quindi teoreme 3.5 + 3.6 ottieniamo il teoreme di
correttezza e completezza

Corollario

Dato uno filf d di P_0 , è decidibile il problema di
dire se d è un teoreme di P_0 (Problema di decisione)

Questo è ricducibile al problema di dire se d
è una tautologia (che è a sua volta decidibile)

Introduzione di modelli Computazionali e modelli fondamentali delle programmazione funzionale

Cose significano computer?

trasformare un'informazione da una forma implicita ad una forma esplicita

$$\begin{array}{ccc} \text{Forme} & & \text{Forme} \\ \text{implicite} & & \text{esplicite} \\ (3+9)2 & \Rightarrow & 14 \end{array}$$

Un modello computazionale è una specifica formalizzazione matematica del concetto di "computazione" e delle mansioni ad esso collegate come per esempio informazioni, trasformazioni, etc...

I linguaggi di programmazione sono ispirati direttamente o indirettamente ad uno o più modelli computazionali.

Linguaggi di tipo imperativo (c, pascal, Fortran)

Sono quelli basati su modelli come la MT e la URM
URM = unlimited register machine è un modello computazionale basato su un numero illimitato di registri numerati, ognuno dei quali puo' contenere un numero naturale

Questo tipo di modello utilizza istruzioni semplici (incremento, decremento...) per manipolare i registri e definire algoritmi

L'URM è equivalente alle macchine di Turing in termini di potenza computazionale e viene usata per lo studio delle completezza e delle computabilità

Le mansioni fondamentali di tali linguaggi sono:

- Memorie
- variabile (qualsiasi che si può leggere o modificare)
- esperimento
- istruzione
- iterazione

Linguaggi di programmazione funzionali (Haskell, Scheme, Erlang)

tali linguaggi fanno riferimento al modello computazionale del λ-CALCOLO.

Nozioni del tipo

- Espressioni entità
- Variabili < scendente
- Valutazione astrazione di un valore concreto
- Ricorsione

Linguaggi di programmazione logici (Prolog)

Questi linguaggi si basano su delle proprietà di sottosistemi delle logiche del primo ordine.

Il concetto di computazione coincide con il concetto di ricerca di una deduzione per une formule logiche

Le programmazione funzionale

Le funzioni che restituisce l'insieme di une stringhe $x \in \Sigma^*$ sono definite come: $x = \begin{cases} \epsilon & \text{se } x = \epsilon \\ \gamma e \mid x = oy \text{ con } \epsilon \in \Sigma \text{ e } y \in \Sigma^* \end{cases}$

Questa definizione, oltre ad identificare unicamente una funzione sulle stringhe dell'alfabeto Σ , descrive implicitamente un metodo per calcolarla.

$$\text{clos} = \overset{2}{\text{bd}} \circ \overset{2}{\text{c}} = \overset{1}{\text{dfe}} = \epsilon \text{ dfe} = \text{dfe}$$

$$\text{mr} [\] = []$$

$$\text{mr} [e : y] = (\text{mr } y) ++ [e]$$

Quando chiediamo al Haskell di calcolare l'insieme delle stringhe clos

> mr "clos"
> "dfe" (risposta)

Esercizio

non ho capito

Sistema di equazioni differenziali

!!

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

$$g : \mathbb{R} \rightarrow \mathbb{R}$$

$$f'' - 6g' = 6 \sin(x)$$

$$6g'' + 2f' = 6 \cos(x)$$

$$f(0) = 0, f'(0) = 0, f(0) = 1, f'(0) = 1$$

Esempio per cui non tutte le definizioni di funzioni descrivono anche un procedimento per il loro calcolo

Una versione equivalente di `mr` del linguaggio Haskell:

`mr = \y → if (null y) then y
else (mr (tail y)) ++ [head y]`

`null`: restituisce true se l'esponente è la lista vuota
falsa altrimenti

`tail`: prende come esponente e restituisce una lista
uguale a quelle che abbiamo ma senza il primo elemento

`head`: prende una lista come esponente e restituisce il
sempre elemento

`++` prende due liste come esponenti e restituisce la
loro concatenazione

`if ... then ... else` espressioni condizionali

"`\y →`" queste è una funzione che ne prende un input
un esponente `y` e restituisce quello che c'è dopo le
parentesi

"`mr =`" `mr` è il nome

`square x = x * x` forme composite per definire il quadrato
di `x` in Haskell

`square :: X → X · X`

Definiamo il fattoriale:

$$\text{fact } 0 = 1$$

$$\text{fact } n = n \cdot (\text{fact}(n-1))$$

forme composite

$$\text{fact} = \lambda n \rightarrow \text{if } (n == 0) \text{ then } 1 \text{ else } n \cdot (\text{fact}(n-1))$$

Definiamo come definire una funzione in un linguaggio di prog funzionale

1) Costruire un'espressione matematica:

variabili, costanti, funz predefinite e inserire nomi di funz già esistenti \Rightarrow Applicazione funzionale

2) Costruire una funzione anonima

definire tale espressione chiamata corpo delle funzioni identificando una variabile (che chiama nome parametro formale) e utilizziamo l'operatore funzione

$\lambda \rightarrow$ operatore di astrazione funzionale

3) Assegnare la funzione anonima ad un nome utilizzando l'operatore =

Composizione nei linguaggi funzionali:

> nr "esol"

Indichiamo con nro A nro B nro C differenti passi

di composizione

• Primo passo computazionale

nro A sostituire un nome con l'espressione omologa col suo

• \rightsquigarrow B sostituire il parametro attuale al posto del parametro = suo formale

• \rightsquigarrow C calcolo di una funzione predefinita e cui abbiamo applicato un valore esplicito

"(f_e)

$$f(x) = x + 3$$

$$"f(x) = x + 3"$$

\downarrow parametro funzione f

ogni passo computazionale \rightarrow trasformare una sottossequenza

Pagine 8 studiare codice funzione m^j

Vedere come sono correlati no^A , no^B , no^C per fare queste cose

Linguaggio imperativo: schemi

Boolean even := true; // variabile globale

:

PROCEDURE calculate (INTEGER value): Integer;

begin

even := not even; // variabile globale dell'ambiente

if even

then Return (value + 1)

else Return (value + 2)

end if

end

:

print (calculate (6));

:

print (calculate (6));

Il valore restituito dipende dal numero di volte che viene eseguito il codice

Side effect: che è proprio dei linguaggi imperativi e non dei linguaggi funzionali

- le due print sono volentieri che con queste promesse produrre i risultati diversi $4 = (6+1)$
 $8 = (6+2)$
non dipendono dall'esperimento che do alle funzioni
Questo valore dipende dalla variabile globale even
le cui modifiche rappresentano un side effect
Insieme nelle programmazione funzionale poiché le funzioni definite sono delle vere e proprie funzioni

motomotiche il valore di una rotazione si moltiplica stessa indipendentemente da dove si trova e da quante volte viene moltiplicata. Queste proprietà si chiamano **referential properties**.