Logica per l'Informatica

Astrazione e generalizzazione in matematica e programmazione

Claudio Sacerdoti Coen

<sacerdot@cs.unibo.it>

Universitá di Bologna

02/12/2020

Astrazione

"Processo tipico della matematica che permette di definire enti, concetti o procedure matematiche, estraendo e isolando alcune caratteristiche comuni a più oggetti e trascurandone altre"

Esempi di astrazione dai dettagli implementativi:

1. implementazioni concrete dei naturali in teoria degli insiemi
   0 := $\emptyset$, 1 := $\{\emptyset\}$, 2 := $\{\emptyset, \{\emptyset\}\}$, . . . , n+1 := $n \cup \{n\}$, . . .
   ove $2 + 3 = 5$ e $2 \in 5$ e $2 \subseteq 5$
   oppure
   0 := $\emptyset$, 1 := $\{\emptyset\}$, 2 := $\{\{\emptyset\}\}$, . . . , n+1 := $\{n\}$, . . .
   ove $2 + 3 = 5$ e $2 \notin 5$ e $2 \not\subseteq 5$
   vs concetto astratto di numero naturale

$$\exists \mathbb{N}, O, S.$$
$$\forall n.(n \in \mathbb{N} \iff n = O \vee \exists m.(m \in \mathbb{N} \wedge n = Sm)) \wedge$$
$$(\forall m.O \neq Sm) \wedge$$
$$(\forall n, m.Sn = Sm \Rightarrow n = m)$$

Tecniche comuni in programmazione/dimostrazione

Astrazione

"Processo tipico della matematica che permette di definire enti, concetti o procedure matematiche, estraendo e isolando alcune caratteristiche comuni a più oggetti e trascurandone altre"

Esempi di astrazione dai dettagli implementativi:

1. implementazioni concrete di liste di interi

```
struct node {int item; struct node* next;}
```

oppure

```
struct node {int item; struct node* next;
                       struct node* prev;}
```

vs concetto astratto (interfaccia) di una lista

```
node* empty();
insert(int n, node* l);
remove(int n, node* l);
...
```

Tecniche comuni in programmazione/dimostrazione

Generalizzazione

“Generalizzare significa prendere una definizione o una relazione che si applica a certi casi, e definirla in un nuovo modo tale che nei casi di prima dia gli stessi risultati, però si applichi anche ad altri casi.”

Vedremo fra poco alcuni esempi in matematica e in programmazione.

Benefici dell'astrazione e della generalizzazione

1. Riuso: le buone generalizzazioni si applicano moltissime volte
2. Chiarezza: le generalizzazioni catturano concetti di alto livello, introducono nomi comprensibili, chiarificano e minimizzano le assunzioni
3. Decoupling: attraverso astrazione e generalizzazione è possibile fare in modo che la correttezza di una parte non dipenda dall'implementazione di un'altra parte, permettendo modifiche indipendenti
4. Correttezza: l'alternativa al riuso è il cut&paste che può propagare errori, da correggere poi in tutte le copie, e introdurre errori quando il codice viene riusato ove non valgano gli stessi invarianti sui dati

Espressività dei linguaggi di programmazione

Tesi di Church-Turing

Qualunque problema di calcolo risolvibile da un linguaggio di programmazione general-purpose lo è da qualunque altro linguaggio general-purpose.

Quindi tutti i linguaggi di programmazione hanno lo stesso potere espressivo se si considera il solo calcolo.

In cosa differiscono quindi i linguaggi di programmazione?

Espressività dei linguaggi di programmazione

I linguaggi di programmazione differiscono enormemente nella capacità di generalizzazione e talvolta anche di astrazione.

1. Una funzione generica esprimibile in un linguaggio $\mathcal{A}$ potrebbe richiedere cut&paste nel linguaggio $\mathcal{B}$

   Esempio: in C le funzioni che operano su liste di interi e su liste di coppie di interi (struct) sono distinte
2. Alcuni linguaggi non hanno meccanismi di astrazione (è il programmatore che si deve impegnare a NON sfruttare i dettagli implementativi); altri permettono di astrarre solo sull'esistenza di certi dati/funzioni; altri ancora sull'implementazione di certi tipi; etc.

Informatica vs matematica

In matematica lo studio dell'astrazione e della generalizzazione ha dato vita all'algebra astratta (o moderna), ben sviluppata in tutto il XX secolo, e più recentemente a teorie ancora più potenti (es. teoria delle categorie).

In informatica non vi è un consenso sui meccanismi da adottare. Es.:

1. moduli (es. in OCaml, JavaScript)
2. classi e code inheritance (es. in C++, Java, ...)
3. polimorfismo ad-hoc (overloading, es. in C++, Julia)
4. polimorfismo generico (generics, templates, ...)
5. polimorfismo generico bounded (es. in Java)
6. funtori (es. in ML e OCaml)
7. traits (es. in Rust, Java)
8. type classes (es. in Haskell)
9. ...

Esempio di generalizzazione in matematica

Teorema T1: $\forall e \in \mathbb{N}.((\forall x.x + e = x) \Rightarrow e = 0)$
Dimostrazione: sia $e \in \mathbb{N}$ t.c. $\forall x.x + e = x$ (H). Per H, $0 + e = 0$. Poichè $e = 0 + e$ si ha $e = 0$. □

Teorema T2: $\forall e \in \mathbb{Z}.((\forall x.e * x = x) \Rightarrow e = 1)$
Dimostrazione: sia $e \in \mathbb{Z}$ t.c. $\forall x.e * x = x$ (H). Per H, $e * 1 = 1$. Poichè $e = e * 1$ si ha $e = 1$. □

Gli enunciati e le prove sono simili, ma differenti, così come i tipi di dato e le operazioni. C'è una generalizzazione comune? È istruttivo studiarla?

Esempio di generalizzazione in matematica

Teorema T1: $\forall e \in \mathbb{N}.((\forall x.x + e = x) \Rightarrow e = 0)$
Dimostrazione: sia $e \in \mathbb{N}$ t.c. $\forall x.x + e = x$ (H). Per H, $0 + e = 0$. Poichè $e = 0 + e$ si ha $e = 0$. □

Teorema T2: $\forall e \in \mathbb{Z}.((\forall x.e * x = x) \Rightarrow e = 1)$
Dimostrazione: sia $e \in \mathbb{Z}$ t.c. $\forall x.e * x = x$ (H). Per H, $e * 1 = 1$. Poichè $e = e * 1$ si ha $e = 1$. □

Teorema G:
$\forall \mathbb{A}.\forall \circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}.\forall e_r, e_l \in \mathbb{A}.$
$(\forall x.x \circ e_r = x) \wedge (\forall x.e_l \circ x = x) \Rightarrow$
$e_l = e_r$
Dimostrazione: siano $\mathbb{A}$ insieme, $\circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}$ e $e_l, e_r \in \mathbb{A}$ t.c. $\forall x.x \circ e_r = x$ (H1) e $\forall x.e_l \circ x = x$ (H2). Per H1 e H2 si ha $e_l = e_l \circ e_r = e_r$. □

Esempio di generalizzazione in matematica

Teorema T1: $\forall e \in \mathbb{N}.((\forall x.x + e = x) \Rightarrow e = 0)$
Teorema T2: $\forall e \in \mathbb{Z}.((\forall x.e * x = x) \Rightarrow e = 1)$
Teorema G:
$\forall \mathbb{A}.\forall \circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}.\forall e_r, e_l \in \mathbb{A}.$
$(\forall x.x \circ e_r = x) \wedge (\forall x.e_l \circ x = x) \Rightarrow$
$e_l = e_r$

T1 è G nel caso particolare $\mathbb{A} = \mathbb{N}$, $\circ = +$, $e_l = 0$.
T2 è G nel caso particolare $\mathbb{A} = \mathbb{Z}$, $\circ = *$, $e_r = 1$.
Altri esempi di istanze di G:

- $\forall e \in \mathbb{N}.((\forall x.e + x = x) \Rightarrow e = 0)$
- $\forall e \in \mathbb{B}.((\forall b.e \,\&\, b = e) \Rightarrow b = \mathit{true})$

G è una generalizzazione utile di T1 e T2 (ha altre istanze oltre a T1 e T2)

Esempio di generalizzazione in matematica

Teorema G:
$\forall \mathbb{A}.\forall \circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}.\forall e_r, e_l \in \mathbb{A}.$
$(\forall x.x \circ e_r = x) \wedge (\forall x.e_l \circ x = x) \Rightarrow$
$e_l = e_r$

Introduciamo due definizioni:

1. e è un elemento neutro a sinistra per $\circ$ sse $\forall x.e \circ x = x$
2. e è un elemento neutro a destra per $\circ$ sse $\forall x.x \circ e = x$
3. e è un elemento neutro per $\circ$ sse è neutro sia a sinistra che a destra

Teorema G: se un'operazione binaria su $\mathbb{A}$ ha sia un elemento neutro a sinistra che uno a destra, allora questi coincidono.

G ha un enunciato più comprensibile

Esempio di generalizzazione in matematica

1. e è un elemento neutro a sinistra per $\circ$ sse $\forall x.e \circ x = x$
2. e è un elemento neutro a destra per $\circ$ sse $\forall x.x \circ e = x$
3. e è un elemento neutro per $\circ$ sse è neutro sia a sinistra che a destra

Ci sono elementi neutri a destra/sinistra che non sono elementi neutri?
Sì. Es: 1 è un elemento neutro a destra per la divisione, ma non è neutro. ($4/1 = 4$ ma $1/4 \neq 4$).

Teorema H: se $\circ$ è un'operazione commutativa, allora se ha un elemento neutro a destra o a sinistra allora tale elemento è neutro.

G è una generalizzazione informativa: ci ha fatto scoprire nuovi concetti e comprendere meglio i precedenti

Dalle generalizzazioni alle teorie matematica

Una volta introdotti nuovi concetti è possibile continuarne lo studio in astratto.

Può un'operazione avere elementi neutri a sinistra distinti?
Sì. Es: $n \circ m := |n| * m$. Sia 1 che -1 sono elementi neutri a sinistra. L'operazione non ha elementi neutri a destra.

Teorema: l'elemento neutro, se esiste, è unico.
Dimostrazione: siano e_1, e_2 due elementi neutri. In particolare e_1 è neutro a sinistra e e_2 lo è a destra. Quindi, per G, essi coincidono. □
Corollario (per H): l'elemento neutro di un'operazione commutativa, se esiste, è unico.

Dalle teorie matematiche alle strutture algebriche

Se una teoria è interessante, vi sono molti teoremi dimostrabili a partire da essi.

Ripetere ogni volta in maniera esplicita tutte le assunzioni non è una buona idea. Esempio:

Corollario (per H): l'elemento neutro di un'operazione commutativa, se esiste, è unico.
Corollario (per H) (tesi esplicitata):
$\forall \mathbb{A}.\forall \circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}.\forall e \in \mathbb{A}.(e \text{ elemento neutro } \Rightarrow$
$\forall e' \in \mathbb{A}.(e' \text{ elemento neutro} \Rightarrow e = e'))$

Due teoremi hanno le stesse ipotesi?
(difficile capirlo a prima vista)
Manca un nome per tale "pacchetto" di assunzioni e ipotesi.
(un nome introduce un concetto)

Dalle teorie matematiche alle strutture algebriche

Una struttura algebrica è una tupla (o ennupla) formata da uno o più insiemi, zero o più elementi, zero o più funzioni (chiamate operazioni) su tali insiemi e zero o più assiomi (= proprietà) che devono essere soddisfatte.

Esempi:

- un left unital magma è una tripla $(\mathbb{A}, \circ, e)$ t.c. $\circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}$ e e è un elemento neutro a sinistra per $\circ$.
- un right unital magma è una tripla $(\mathbb{A}, \circ, e)$ t.c. $\circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}$ e e è un elemento neutro a destra per $\circ$.
- un unital magma è una tripla $(\mathbb{A}, \circ, e)$ t.c. $\circ \in \mathbb{A}^{\mathbb{A}\times\mathbb{A}}$ t.c. $(\mathbb{A}, \circ, e)$ è sia un left unital magma che un right unital magma

Ce ne sono centinaia, ne introdurremo alcuni. Interi scaffali di librerie sono dedicate allo studio di una sola di queste strutture algebriche!

Una volta introdotte, le strutture algebriche stesse diventano oggetto di studio. Esempio:

Teorema (prodotto cartesiano di left unital magmas): siano $(\mathbb{A}, \circ, a)$ e $(\mathbb{B}, \bullet, b)$ due left unital magma. Allora anche $(\mathbb{A}, \circ, a) \times (\mathbb{B}, \bullet, b) := (\mathbb{A} \times \mathbb{B}, \oplus, \langle a, b\rangle)$, chiamato prodotto cartesiano dei due left unital magmas, è un left unital magma dove $\langle x_1, y_1\rangle \oplus \langle x_2, y_b\rangle := \langle x_1 \circ x_2, y_1 \bullet y_2\rangle$.
Dimostrazione: sia $c \in \mathbb{A} \times \mathbb{B}$, ovvero $c = \langle x, y\rangle$ e dimostriamo $\langle a, b\rangle \oplus \langle x, y\rangle = \langle x, y\rangle$. Si ha
$\langle a, b\rangle \oplus \langle x, y\rangle = \langle a \circ x, b \bullet y\rangle = \langle x, y\rangle$. □

Esempio di istanza: poichè $(\mathbb{R}, +, 0)$ è un left unital magma, allora lo è anche $(\mathbb{C}, +, 0)$.

Strutture algebriche come astrazioni

Abbiamo introdotto i left unital magmas a partire da una generalizzazione di due teoremi.

Un altro modo di pensare una struttura algebrica è però come astrazione di tutte le sue istanze.

Esempio: un left unital magma astrae l'insieme dei numeri naturali con la somma nascondendone tutte le proprietà, tranne il fatto che lo 0 sia un elemento neutro a sinistra!

È un modo per dire che temporaneamente siamo interessati solamente a tale proprietà.

Pertanto, solitamente, siamo interessanti a che tale proprietà si preservi durante la trasformazione dei dati. Introduciamo a tal fine la nozione di morfismo.

Morfismi di strutture algebriche

Un morfismo fra due strutture algebriche dello stesso tipo è una funzione che mappi gli elementi della prima in quelli della seconda rispettandone le proprietà e il modo di manipolarli.

Esempio: siano $(A, \circ, a)$ e $(B, \bullet, b)$ due left unital magma. Un morfismo da $(A, \circ, a)$ a $(B, \bullet, b)$ è una funzione $f \in B^A$ t.c.

1. $f(a) = b$
2. $\forall x, y. f(x \circ y) = f(x) \bullet f(y)$

Esempio: $(\mathbb{N}, +, 0)$ e $(\mathbb{N}, *, 1)$ sono due left unital magma. La funzione $f(n) = 2^n$ è un morfismo dal primo al secondo in quanto:

1. $2^0 = 1$
2. $\forall x, y. 2^{(x+y)} = 2^x * 2^y$

Isomorfismi di strutture algebriche

Un isomorfismo di due strutture algebriche è un morfismo dalla prima alla seconda che sia una funzione biettiva la cui inversa sia anch'essa un morfismo.

Due strutture algebriche sono isomorfe se c'è almeno un isomorfismo fra di esse.

Tutte le manipolazioni su una struttura possono essere effettuate su quella isomorfa e viceversa, a seconda di cosa sia più conveniente.

Esempio: $(\mathbb{R}_0^+, +, 0)$ e $(\mathbb{R}^+, *, 1)$ sono isomorfe come testimoniato dal morfismo $e^{\cdot}$ e dal suo morfismo inverso $\log \cdot$. Invece di moltiplicare numeri grandi posso passare alla scala logaritmica e sommare numeri piccoli.

Generalizzazione in informatica

Problema: data una lista di interi, restituirne la somma

```
sum [] = 0
sum (n:l) = n + sum l
```

Problema: data una lista di booleani, restituire la loro congiunzione

```
conj [] = True
conj (b:l) = conj l && b
```

Problemi simili, soluzini simili (ma non uguali: b congiunto a destra, n sommato a sinistra).

C'è una generalizzazione utile, chiarificante e informativa?

Generalizzazione in informatica

Il codice è sempre accompagnato dalle proprietà che soddisfa!

Teorema: `sum (l1 ++ l2) = sum l1 + sum l2`
Dimostrazione: fisso `l2` e procedo per induzione su `l_1`.
caso []:
`sum ([] ++ l2)=sum l2=0 + sum l2=sum [] + sum l2`
caso `n:l`:
per ipotesi induttiva `sum (l ++ l2) = sum l + sum l2`
si ha `sum (n:l ++ l2) = sum (n:(l ++ l2))`
`= n + sum(l ++ l2) = n + (sum l + sum l2)`
`= (n + sum l) + sum l2 = sum (n:l) + sum l2` □

Generalizzazione in informatica

Problema: data una lista di interi, restituirne la somma

```
sum [] = 0
sum (n:l) = n + sum l
```

Problema generalizzato da sum: data una lista `[x1,...,xn]`, un valore `e` e un'operazione `op`, restituire `op(x1,op(x2,...op(xn,e)...))`

```
foldr op e [] = e
foldr op e (n:l) = op n (foldr op e l)
```

Ritroviamo `sum` e una variante di `conj` come istanze:

- `sum = foldr (+) 0`
- `conj' = foldr (&&) True`

Generalizzazione in informatica

Problema: data una lista di booleani, restituirne la congiunzione

```
conj [] = True
conj (b:l) = conj l && b
```

Problema generalizzato da conj: data una lista `[x1,...,xn]`, un valore `e` e un'operazione `op`, restituire `op(...op(op(e,xn),xn-1),...x1)`

```
rfoldl op e [] = e
rfoldl op e (n:l) = op (rfoldl op e l) n
```

Ritroviamo una variante di `sum` e `conj` come istanze:

- `sum' = foldr (+) 0`
- `conj = rfoldl (&&) True`

Generalizzazione in informatica

Il codice è sempre accompagnato dalle proprietà che soddisfa!

Teorema: per ogni `op,e` t.c. `op` è associativa e `e` il suo elemento neutro a sinistra si ha

```
foldr op e (l1++l2)=op(foldr op e l1)(foldr op e l2)
```

Dimostrazione: fisso `op,e,l2` t.c. soddisfino le proprietà richieste e procedo per induzione su `l_1`.

caso []: `foldr op e ([] ++ l2)=foldr op e l2`

`=op e (foldr op e l2)` (perchè `e` neutro a sinistra)

`=op (foldr op e []) (foldr op e l2)`

caso `n:l`:
per ipotesi induttiva

```
foldr op e (l++l2)=op (foldr op e l) (foldr op e l2
```

si ha

```
foldr op e (n:l ++ l2) = foldr op e (n:(l ++ l2))
= op n (foldr op e (l ++ l2))
= op n (op (foldr op e l) (foldr op e l2))
= op (op n (foldr op e l)) (foldr op e l2)
```

(perchè `op` è associativa)

```
= op (foldr op e (n:l)) (foldr op e l2)
```

□

Lo stesso teorema NON vale per la rfoldl!

Es: consideriamo `op a b = b`, operazione associativa di elemento neutro a sinistra 7. Si ha
`rfoldl op 7 [2,4] = op (op 7 4) 2 = 2`, ma
`op (rfoldl op 7 [2]) (rfoldl op 7 [4])`
`= op (op 7 2) (op 7 4) = op 2 4 = 4` $\neq$ `2`

Quindi le due generalizzazioni `foldr` e `rfoldl` NON sono equivalenti perchè non soddisfano le stesse proprietà!

Generalizzazione in informatica

Esercizio: dimostrare per induzione strutturale che:

Teorema: se `op` è commutativa allora
`foldl op e l = rfoldl op e l`

Quindi la `sum` e la sua variante `sum'` sono equivalenti in quando `+` è commutativa, così come la `conj` e la sua variante `conj'` in quanto `&&` è commutativa.

Tuttavia, generalizzando scegliendo "alla cieca" una delle due si rischia di ottenere codice errato una volta reinstanziato su operazioni non commutative.

Generalizzazione in informatica

L'astrazione `foldl` è:

1. utile: per calcolare la produttoria dei numeri in una lista posso chiamare `foldl (*) 1`, per il minimo `foldl min max_int`, etc.
2. chiarificante: l'enunciato del problema spiega in che ordine vengono processati gli elementi della lista (cfr con `rfoldl`)
3. informativa: tramite essa abbiamo esplicitato le ipotesi "nascoste" usate nella prova di commutazione `foldl`/(++) (es. passare un elemento che non sia neutro è una cattiva idea)

Generalizzazioni NON utili

La seguente generalizzazione è un esempio di generalizzazione inutile, non chiarificante (anzi!) e non informativa:

```
gen b [] = if b then 0 else True
gen b (n:l) = (if b then (+) else (&&)) n (gen b l)
```

Infatti:

1. `sum = gen True` e `conj' = gen False` pertanto `gen` generalizza le due
2. ma nessun altro problema può essere risolto con un'istanza di `gen`

Strutture algebriche in programmazione

```
foldr op e [] = e
foldr op e (n:l) = op n (foldr op e l)
```

Come nel caso della matematica, passare sempre in giro esplicitamente alla funzione `foldl` e similari tutti i parametri `op,e` e similari non è consigliabile:

- il codice viene offuscato
- si rischia di passare coppie non coerenti (es. una operazione e un elemento che non è neutro per essa)
- non si viene a creare una libreria di "strutture algebriche" componibili (es. la `foldr` lavora su una struttura (A, op, e) dove A è un tipo di dato, op un'operazione binaria su A consigliata essere associativa e e consigliato essere il suo elemento neutro; strutture di questo tipo si possono comporre, es. con prodotti cartesiani)

Strutture algebriche in programmazione

Ogni linguaggio di programmazione ha i suoi meccanismi per dichiarare tali strutture algebriche. Haskell ha le type classes. Esempio:

Un monoide è una tripla $(A, \circ, e)$ t.c. $\circ$ è associativo e $(A, \circ, e)$ è un unital magma (ovvero e è l'elemento neutro di $\circ$).
Un esempio è $(\mathbb{N}, +, 0)$.

In sintassi Haskell:

```
class Monoid a where
   op :: a -> a -> a
   e :: a
 -- dove op associativo, e elemento neutro

instance Monoid Integer where
   op = (+)
   e = 0
```

Programmare con le type classes

Possiamo scrivere le nostre funzioni astraendo su una type class:

```
foldr [] = e
foldr (n:l) = op n (foldr l)
```

Haskell capisce da solo il seguente tipo:
`Monoid a => [a] -> a` da leggersi come:

1. per ogni tipo `a` (la `a` è una variabile di tipo libera)
2. se (implicazione logica `=>`) `a` è un `Monoid` (scritto `Monoid a`)
3. allora il tipo è quello di una funzione di input "lista di `a`" (scritto `[a]`) e output `a`

Programmare con le type classes

Possiamo scrivere le nostre funzioni astraendo su una type class:

```
foldr [] = e
foldr (n:l) = op n (foldr l)
```

Haskell capisce da solo il seguente tipo:

```
Monoid a => [a] -> a
```

Per calcolare la somma della lista di numeri $[1, 2, 3]$ scrivo semplicemente `foldr [1,2,3]` (invece di quello che senza type class avrei scritto `foldr (+) 0 [1,2,3]`)

`[1,2,3]` ha tipo `[Integer]`, quindi `a = Integer`, quindi Haskell cerca una istanza di `Monoid Integer` e trova quella (vedi slides precedenti) con `op = (+)` e `e = 0`.

Programmare con le type classes

Come in matematica, una volta introdotta una struttura matematica sotto forma di type class, posso costruirci sopra una libreria.

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
   op (x1,y1) (x2,y2) = (op x1 y1, op x2 y2)
   e = (e,e)
```

Lettura: se `a` è un `Monoid` e `b` è un `Monoid` allora il prodotto cartesiano di `a` e `b` (scritto `(a,b)`) è un `Monoid` tale che ...

Eseguendo `foldr [(1,2),(3,4)]` ottengo `(4,6)`!
Eseguendo `foldr [(1,(1,1)),(2,(1,0))]` ottengo `(3,(2,1))`!

Morfismi in programmazione

Anche i morfismi giocano un ruolo importante in programmazione.

Tipiamo esplicitamente le nostre liste:

`[] :: [a]`

(la lista vuota è una lista di `a` per ogni tipo `a`)

`(:) :: a -> [a] -> [a]`

(`(:)` prende una testa di tipo `a` e una coda di tipo `[a]` per restituire `[a]`, per ogni tipo `a`)

Possiamo astrarre il tutto in una struttura algebrica (a, b, e, op), chiamata foldable, dove si ha $e \in b$ e $op \in b^{a \times b}$.

Le liste di `a` sono l'istanza di foldable $(a, [a], [], :)$.
Array, alberi, etc. sono altre istanze.

Morfismi in programmazione

Possiamo astrarre il tutto in una struttura algebrica (a, b, e, op), chiamata foldable, dove si ha $e \in b$ e $op \in b^{a \times b}$.

Le liste di `a` sono l'istanza di foldable $(a, [a], [], :)$.

Un morfismo di foldable da (a, b, e, op) a (a', b', e', op') è una coppia di funzioni $f_a \in a'^a$ e $f_b \in b'^b$ t.c.

1. $f_b(e) = e'$
2. $f_b(op(x, l)) = op'(f_a(x), f_b(l))$ (per ogni x, l)

Il caso particolare dove (a, b, e, op) è $(a, [a], [], :)$, $a' = a$ e $f_a(x) = x$ diventa:

1. $f_b([]) = e'$
2. $f_b(x : l) = op'(x, f_b(l))$ (per ogni x, l)

La f_b è esattamente la `foldr`!

Morfismi in programmazione

Mettendo in pratica quanto detto prima si riesce ad arrivare in Haskell a scrivere un mofrismo `foldr` super-generico che data una qualunque struttura dati che possa essere visitata elemento per elemento (ovvero un'istanza di foldable) ne sintetizza un risultato combinando via via gli elementi incontrati in base a certe funzioni passate dall'utente (un'altra istanza di foldable).

La libreria standard di Haskell prende a prestito a piene mani idee, definizioni e concetti provenienti dal mondo dell'algebra e della teoria delle categorie (semigruppi, monoidi, anelli, funtori, applicativi, monadi, lenti, . . .).

Contenuti del secondo modulo del corso

In questo secondo modulo del corso:

- introdurremo le prime nozioni di algebra astratta, usate in matematica (e talvolta applicate in informatica)
- vedremo semplici esempi di astrazione e generalizzazione in programmazione, ispirati dalla parte algebrica