Type classes

Haskell appartiene ad una famiglia di linguaggi funzionali fortemente tipati, dove però il programmatore non è forzato a specificare il tipo in maniera esplicita. Sarà il compilatore che andrà ad inferire il tipo in base all'utilizzo dell'entità (si parla di **type inference**).

Ovviamente il programma deve essere scritto correttamente, altrimenti il compilatore non sarà capace di inferire alcun tipo (peggio ancora, un tipo diverso da quello desiderato).

Questo avviene anche in altri linguaggi convenzionali, come in Java e C++, ma vengono considerate eccezzioni. In linguaggi come Haskell questa caratteristica è innata.

Un possibile esempio di type inference in linguaggio Standard ML può essere:

```
val a = 3 * 3
val b = 3.14 * 3.14
```

In questo caso sarà il compilatore ad inferire il tipo delle due variabili.
Nel caso di `a` il tipo scelto sarà `int`, mentre per `b` sarà `float`.

Si osservi ora un esempio di codice rifiutato dal compilatore Standard ML:

```
fun square x = x * x
val a = square 3
val b = square 3.14
```

Il compilatore Standard ML restituirà un errore di tipo per una delle due variabili, causato dal fatto che `square` può restituire due tipi differenti. Se il compilatore scegliesse il tipo `int` come tipo da restituire, il compilatore restituirebbe un errore per la variabile `b`.

In OCaml la situazione non è migliore. Riprendendo un esempio simile al precedente:

```
let a = 3 * 3
let b = 3.14 *. 3.14
```

Per evitare questo tipo di problemi, in OCaml viene utilizzato un nuovo operatore aritmetico nel caso delle operazioni con floating point (`*.`).

```
let squareI x = x * x
let squareF x = x *. x
let a = squareI 3
let b = squareF 3.14
```

Questo approccio non è scalabile, volendo introdurre altri tipi numerici si dovrebbe creare un nuovo operatore aritmetico ogni volta.

Un altro caso spinoso è quello dell'uguaglianza. Si osservi un esempio in OCaml:

```
let rec member x = function
        [] -> false
    | (y :: ys) -> x == y || member x ys
;;
val member : 'a -> 'a list -> bool = <fun>
```

Usare `member` su una lista di funzioni (o di altri elementi che non supportano una nozione di uguaglianza) causa un **errore a run-time**.

Riprendendo lo stesso esempio in Standard ML:

```
fun member (x, [])     = false
    member (x, h :: y) = x == h orelse member x ys
;
val member : ''a * ''a list -> bool = <fun>
```

In questo caso, le variabili di tipo `''a` possono essere istanziate solo con tipi per i quali è definita una nozione di uguaglianza.

In Standard ML, se una variabile viene preceduta da un singolo apice (`'`), la variabile potrà essere istanziata con un qualsiasi tipo arbitrario. Se invece è preceduta da un doppio apice (`''`), la variabile potrà essere istanziata solo con tipi per i quali è definita una nozione di uguaglianza.

Questa soluzione risulta essere migliore rispetto a quella di OCaml ma resta incompleta, non comprendendo altre proprietà possibili dei tipi.

Haskell cerca di risolvere questi problemi cercando di introdurre meccanismi più generici, e non ad hoc come quelli discussi precedentemente.

In Haskell, i tipi sono divisi in **classi**, non necessariamente disgiunte. Ogni classe è definita dalle **operazioni** supportate da quei tipi.