Haskell oferă deasemenea şi noţiunea de **clase derivate**. Ca exemplu, uneori vrem să definim o clasă Ord care moşteneşte toate operaţiile din Eq, dar care are în plus un set de operatori de comparaţie şi funcţii care calculează minimul şi maximul .

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Să-l observăm pe **(Eq a)**, contextul / ipoteza / precondiţia din declaraţia clasei. De aceea spunem că Eq este o *superclasă* a lui Ord (invers, Ord este *subclasă* a lui Eq) şi orice tip ce foloseşte o instanţă a lui Ord va trebui deasemenea să fie o instanţă a lui Eq. (în secţiunea următoare oferim o definiţie mai completă a lui Ord luată din Prelude.)

Un avantaj al folosirii unei astfel de clase cu incluziune este un context mai scurt: o expresie tip pentru o astfel de funcţie care foloseşte operatori din ambele clase Eq şi Ord , poate folosi mai degrabă contexul scurt (Ord a) decât (Eq a , Ord a) din moment ce Ord ‘implică ’ sau altfel spus 'derivează' din Eq .

Mai important, metodele sau operaţiile din subclase pot să folosească, să se bazeze, pe existenţa metodelor (operaţiilor) din superclase. De exemplu, declaraţia clasei Ord în Standard Prelude conţine această metodă pentru calculul rezultatului comparaţiei (<):

```
x < y  =  x <= y && x /= y
```

Un exemplu de folosire a clasei Ord exista în scrierea acelui algoritm quicksort definit în secţiunea 2.4.1**:**

```
quicksort :: (Ord a) => [a] -> [a]
```

Cu alte cuvinte quicksort operează doar pe liste de valori din tipurile ordonate. Această restricţie pentru quiksort a luat naştere datorită folosirii operatorilor de comparaţie < şi >= în definiţiile care descriu algoritmul. Haskell permite deasemenea moşteniri multiple (***multiple inheritance)*** din moment ce clasele pot avea una sau mai multe superclase . De exemplu declaraţia :

```
class (Eq a, Show a) => C a where ...
```

crează o clasa **C** care moşteneşte operatori din ambele clase Eq si Show.

Metodele clasei sunt tratate ca declaraţii principale în Haskell. Ele au în comun acelaşi spatiu de nume cu variabilele comune; prin urmare numele nu mai poate fi folosit pentru a indica atât metoda clasei cât şi a variabilei sau alte metode din diferite clase.

Contextele sunt deasemenea permise în declaraţii de date; vezi § 4.2.1

Metodele clasei pot avea anumite restricţii pentru orice variabilă de tip cu excepţia celei care defineşte clasa curentă. Spre exemplu, în această clasă:

```
class C a where
  m :: Show b => a -> b
```

metoda m necesită ca tipul b să fie în clasa Show. Oricum metoda m nu poate adăuga alte restricţii de clase pentru tipul a. Aceasta, dacă există, ar trebui dimpotrivă să facă parte din contextul declaraţiei de clasă.

Până acum am folosit tipuri de ordinul intâi (eng. “first-order”). De exemplu constructorul Tree a avut un argument, ca şi în Tree Integer (arbore ce conţine valori de tip intreg) sau Tree a (reprezentând familia de arbori ce conţine a – valori). Dar **Tree** fără argument este un constructor de tip, şi astfel ca argument primeşte un tip şi returnează ca

rezultat un tip. Nu există valori în Haskell care să conțină acest tip, dar astfel de tipuri "higher – order" pot fi folosite în declarații de clase.

Pentru început considerăm următoarea clasa **Functor** (preluată din Prelude)

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Funcția fmap generalizează funcția map folosită anterior. Observați că ea aplică un constructor de tip f (variabil de la o clasa la alta) altor tipuri a şi b. Ne aşteptăm ca f să fie legată (înlocuită) de un constructor de tip cum ar fi Tree căruia i se poate da un argument. Instanța clasei Functor de Tree va fi deci:

```
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Această declarație de instanță dovedeşte că *Tree* spre deosebire de *Tree a* este o instanță a lui Functor. Această capacitate este de folos. Ea demonstrează abilitatea Haskell-ului de a descrie generic procesarea tipurilor-container permițând funcțiilor cum este f să lucreze similar asupra arborilor, listelor, sau altor tipuri de date compuse.

Constructorii de tip sunt tratați la fel ca şi funcțiile. Tipul **T a b** este parsat ca (**T a**) **b . [...]**

Pentru funcții (->) este un constructor de tip; tipurile f->g şi (->) f g sunt echivalente. În mod similar, tipurile [a] şi [] a sunt echivalente. Pentru t-upluri, constructorii de tip (deasemenea şi constructorii de date) sunt (,) , (, ,) etc.

Aşa cum ştim, sistemul de tipuri determină erorile de scriere în expresii, dar cum rămâne cu erorile cauzate de *expresiile de tip* construite greşit? Expresia (+) 1 2 3 va da o eroare de scriere din moment ce (+) acceptă doar două argumente. În mod similar tipul Tree Int Int ar trebui să producă o eroare din moment ce tipul Tree acceptă un singur argument. Dar cum poate Haskell să determine expresiile de tip scrise greşit? Răspunsul constă în prezenta unui al doilea sistem de tipuri (n.n. un fel de supratipuri) ce asigură corectitudinea tipurilor. Fiecare tip are asociat lui un supratip, din care face parte, (eng. kind), care ajuta ca tipul să fie folosit corect.

Expresiile de tip sunt clasificate în supratipuri diferite care au una din cele doua forme posibile :

- Simbolul * reprezintă orice fel de tip cu date concrete. Adică dacă valoarea v are tipul t, supratipul lui t trebuie să fie * .
- Dacă t1 şi t2 sunt de acelaşi supratip simplu *, atunci * - > * este scrierea supratipului acelor tipuri care iau un tip t1 şi returnează un tip t2

Constructorul de tip care transforma un tip in altul are supratipul *->* ; tipul Tree Int are supratipul *. Elementele clasei Functor trebuie să aibă supratipul *->*. Erori cauzate de supratipuri pot rezulta dintr-o declarație ca: **instance Functor Integer where ...** din moment ce Integer are supratipul * in loc de * ->* .

Supratipurile nu apar direct în programele Haskell. Compilatorul le deduce fără a avea nevoie de vreo "declarație de supratip" . Supratipurile nu sunt vizibele în Haskell până când o eroare de program semnalată nu conduce la o eroare de supratip. Supratipurile sunt atât de simple încat compilatoarele ar trebui să fie capabile să asigure descrierea erorilor printr-un mesaj simplu, atunci când are loc o eroare de supratipuri. Vezi § 4.1.1 si § 4.6 pentru mai multe informații despre supratipuri.