```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Binary Methods with Functional Decomposition

Binary operations

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  - Can arise in original program or after extension

- Function decomposition deals with this much more simply…

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

ML Approach

Addition is different for most **`Int`**, **`String`**, **`Rational`** combinations

- Run-time error for non-value expressions

Natural approach: pattern-match on the pair of values

- For *commutative* possibilities, can re-call with **`(v2,v1)`**

```
fun add_values (v1,v2) =
  case (v1,v2) of
     (Int i, Int j) => Int (i+j)
   | (Int i, String s) => String (Int.toString i ^ s)
   | (Int i, Rational(j,k)) => Rational (i*k+j,k)
   | (Rational _, Int _) => add_values (v2,v1)
   | … (* 5 more cases (3*3 total): see the code *)

fun eval e =
  case e of
     …
   | Add(e1,e2) => add_values (eval e1, eval e2)
```