**Introduction To Haskell Programming**

**Prof. S. P. Suresh**

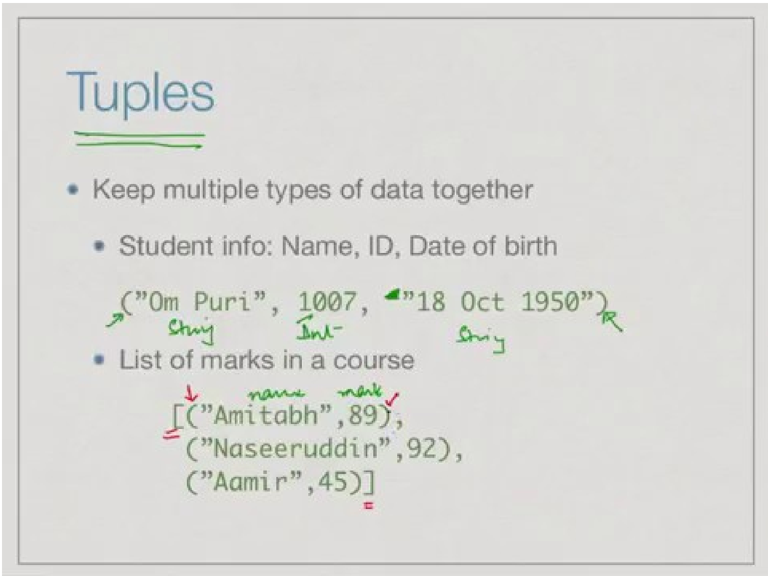
**Chennai Mathematical Institute**

**Module # 02**

**Lecture - 04**

**Tuples**

(Refer Slide Time: 00:02)

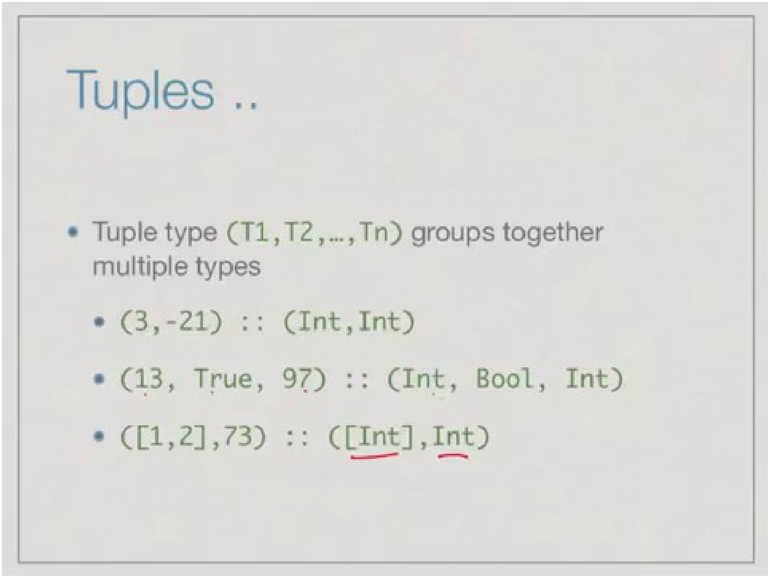



So, we have seen lists, which are sequences of uniform type and in particular, we also looked at Strings, which are the special case of lists, where the uniform type is character. But, very often we need to keep multiple types of data together as a unit. So, this is what in a language like C or C++ would be a Struct or it could be basically a unit of information.

For example, if you want to keep information about the student, you might want to record the name, may be the roll number and date of birth together in one place. So, if you do that, then you would have a String and then, you have say an Int and may be another String. Now, you cannot make this a list, because this is not of a uniform type. Now, you might want to take a group of such things and make it into a bigger item.

For instance, you might want to keep a list of marks for a number of students, where you keep the name and the marks together for a given student and then, we have a list of such items. So, Tuples are Haskell's way of doing this, so notice that, we have used here this round bracket, not the square bracket of this. So, we have the square bracket, which applies to list and then a round bracket which allows us to collect together values of different types.

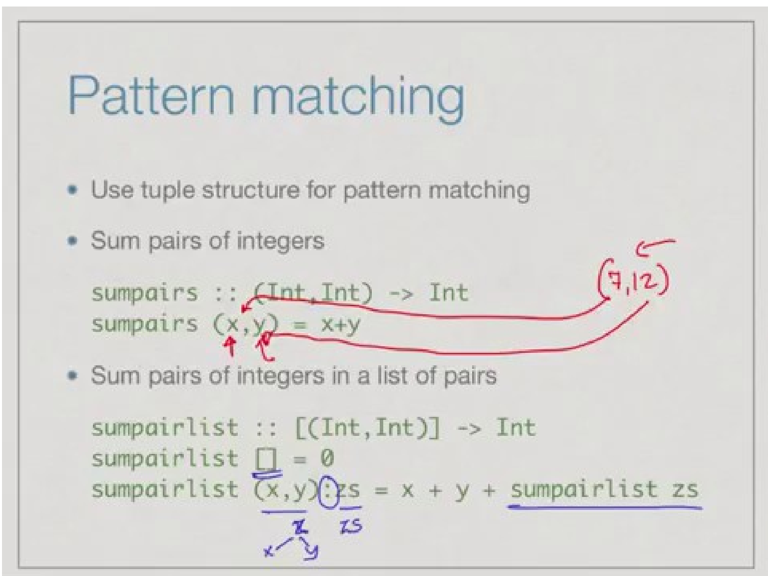
(Refer Slide Time: 01:26)




So, Tuple basically takes some n types and groups it together as a single unit, where we use this round bracket and comma to separate the given parts of the unit and the type of the overall Tuple is inherited from the underlying types. So, if I have a pair of integers (3,-21) written in this way, then its type is the tuple or the pair in this case, (Int, Int)

Well, if I have three values like this (13, True, 97) with round brackets, then the first is an Int, second is a Bool, third is an Int, so its type is tuple of (Int, Bool, Int). Now, because these types need not be uniform, we could have different structures, for instance, we can have a list as the first component and integer as the second component. So, here we have a tuple ([1,2],73) which consists of list of Int and Int.

(Refer Slide Time: 02:18)

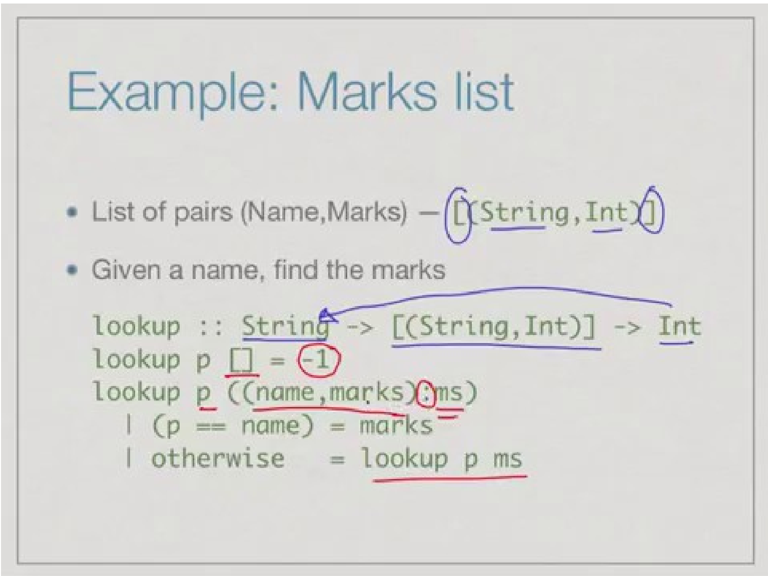


So, because the syntax is very structured, there is only one way to decompose the tuple, which is to use the comma to separate it out, so we can directly use pattern matching to extract each component. So, if I want to take pairs of integers and add them up, then I can just directly say that the first element of the pair will come out as x and the second, so this is the pattern which matches the tuples.

So, when I give it as an input say (7,12), then 7 will get mapped to x, 12 will get mapped to y, because of the pattern matching and then, the answer will be 7+12, which is 19. Now, this pattern matching can be combined with other pattern matching like for lists. So, if you want to take the pair, a list of pairs and add them up across the entire list, so I want x1+y1, x2+y2 and so on.

Then, we use list induction to say that, if I have an empty list of course, the sum is 0, if I have a non empty list, then I have a first value and a second value. So, this is the usual list pattern, it says use colon to separate the z from the zs and the z itself is the pair. So, use x , y to split this as two values x , y and now, I just take x + y as the sum of the first pair and add whatever I get by doing the rest. So, tuples are particularly easy to manipulate in terms of programs, because it is very easy to split them using patterns to get the integer components.

(Refer Slide Time: 03:48)

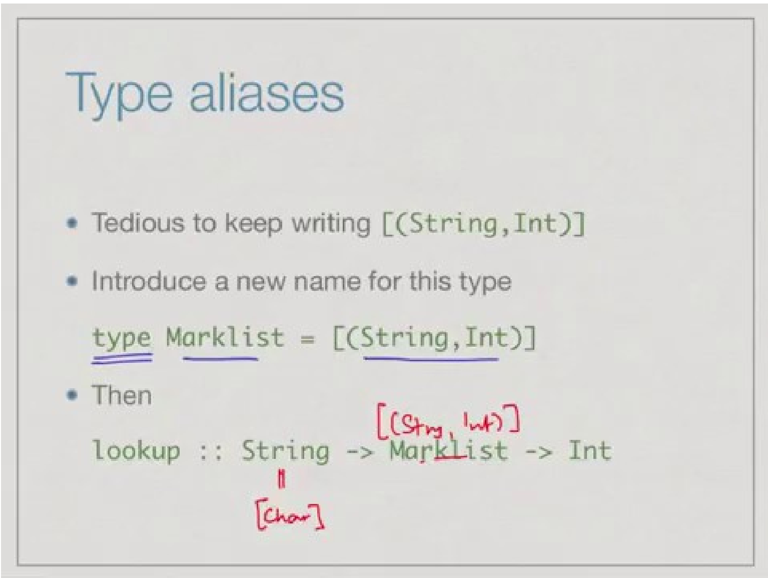



So, for example, supposing as before we had this list of marks of students, where each pair consists of a name and the marks for that name, so we have a pair which consists of a String and an Int and we have a list of such pairs. So, String-> [(String, Int)] is the type of our input and now you want to look up the name, the marks for a given student. So, we have given a name which is the String and we have given this list of marks for the entire class [(String, Int)], and we want to return those marks that this particular student got.

So, we use our usual list induction, so if we have no names or we have not found this name in this list, then we have to return something. So, let us assume that everybody gets a positive mark, so as a default value, we can get a -1 saying that, it was not found. But, as if we still have marks to process, then we break up the marks list into the rest and the first element.

First element again because of the structure will make it up into the name and marks, we match the given argument p with name, if it matches we return the marks, if it does not match we skip this value and look up this. So, this is the familiar list induction and this is this double level pattern matching, one is to do the list pattern with the colon and then, the tuple pattern with the comma.

(Refer Slide Time: 05:12)

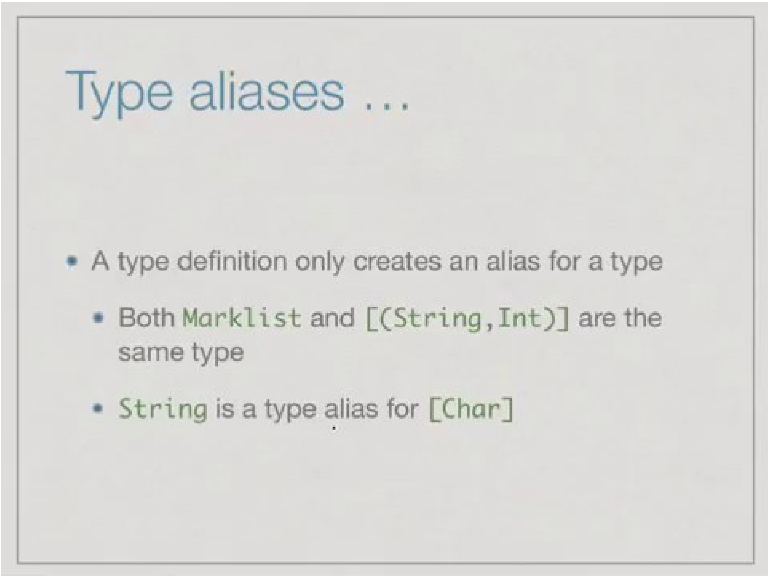



So, Haskell gives us a little bit of flexibility in how to refer to these kinds of types. Very often, because we are grouping together these types as a unit, we want to maybe give it a name, which makes sense for that unit. So, we might want to say instead of writing this stuff as [(String,Int)] we might want to indicate in our program that this particular unit has some meaningful association for us.

So, maybe, we want to give it a name like a marklist. So, Haskell has this declaration called type, which says that the name marklist is a synonym for the name [(String, Int)]. So, this means, we can now change our earlier definitions. So, instead of writing here [(String, Int)] as we have done earlier, once we had this type definition in our program, you can use that, instead of this type.

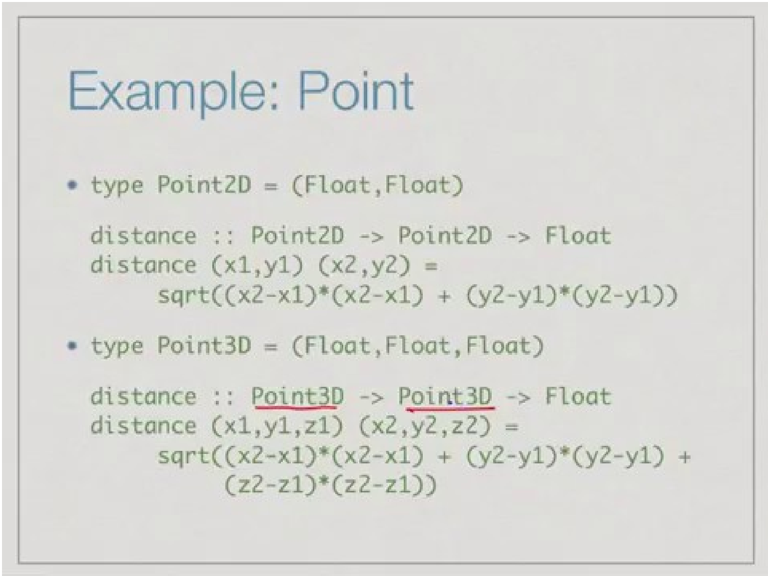
Now, this is just a synonym. In the same way, that String is the same as [Char], there is no difference, there is no difference between writing marklist and [(String, Int)]. But, this is just very useful for us to make our program more legible and also of course, to cut down the amount of steps we have to write and now, a nice thing is that if we change this slightly, then everywhere the type remains the same, provided of course, the functions are so updated to use the new type.

(Refer Slide Time: 06:29)




So, both marklist and [(String, Int)] are exactly the same type and the String is type alias of [Char], so this is not like it is creating a new type. So, it is only creating a new name for a complex type.

(Refer Slide Time: 06:44)

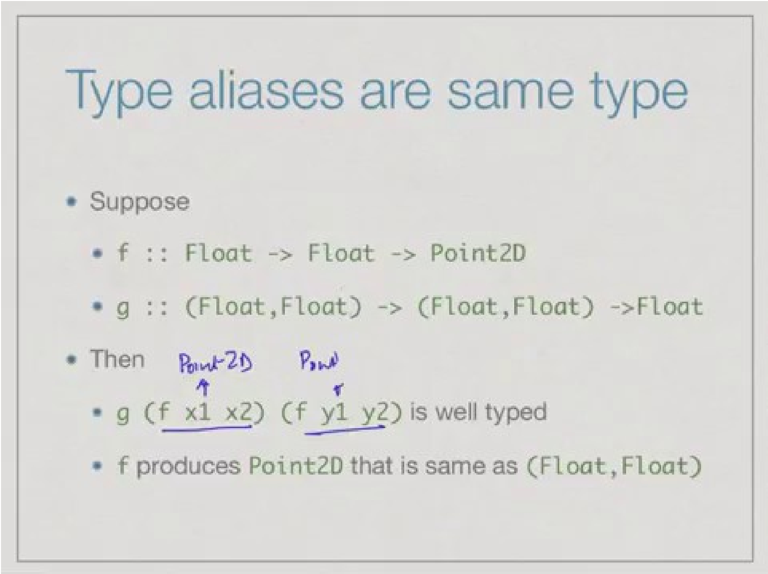



So, here is another example, so supposing we want to represent points in two dimensional space by x, y coordinates, then a typical x, y coordinate will be of type (Float,Float), would be a pair of real numbers. We might want to encapsulate this (Float,Float) as a single name, and call it say 2D point, let us use Point2D to refer to this. Then, we can compute the usual Euclidean distance between (x1, y1) and (x2, y2) using Pythagoras formula, you take (x2-x1)2 + (y2 - y1)2 and take its square root.

So, the distance between (x1, y1) and (x2, y2), so now, we know that the underlying type is Point2D. Point2D is itself a tuple. So, we can use this pattern matching to extract the coordinates without having to ask, what is the first coordinate, what is the second coordinate and then, use these values directly in the function. So, this is an example, now you could correspondingly expand this to a 3D point and have the three dimensional version of the distance formula, which includes the z coordinate and so on.

So, this is an example of how we could use Tuples to make your type of your program more meaningful. Now, you know that you are taking the distance between points, so it makes your program more legible.

(Refer Slide Time: 08:03)

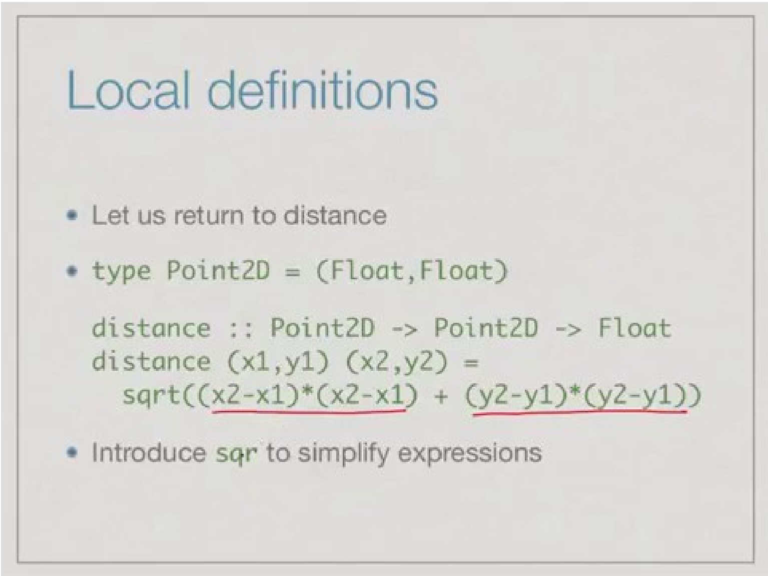



Now, to illustrate the fact that these are only type aliases, now suppose that we have a function f, which takes two Floats and produces a point. Now, remember that this is just by our earlier definition the same as (Float,Float), so we had this thing we said type Point2D equal to (Float,Float). So, this definition gives us an alias, now I have another function g, which takes (Float,Float), (Float,Float) and produce a Float.

So, the whole point is that this should be the same as Point2D and there is no difference we claim between Point2D and (Float,Float) and indeed, if I take the value of f produced for some, so if I take f of x1, x 2 then this produces a Point2D. Again, if y1 and y2 produces a Point2D, now I can feed these to g, with g is expecting only (Float,Float) and (Float,Float), it is not expecting Point 2D. But, because Point 2D and (Float,Float) are exactly the same thing, Haskell will not complain about the type, it will say, this is well typed.

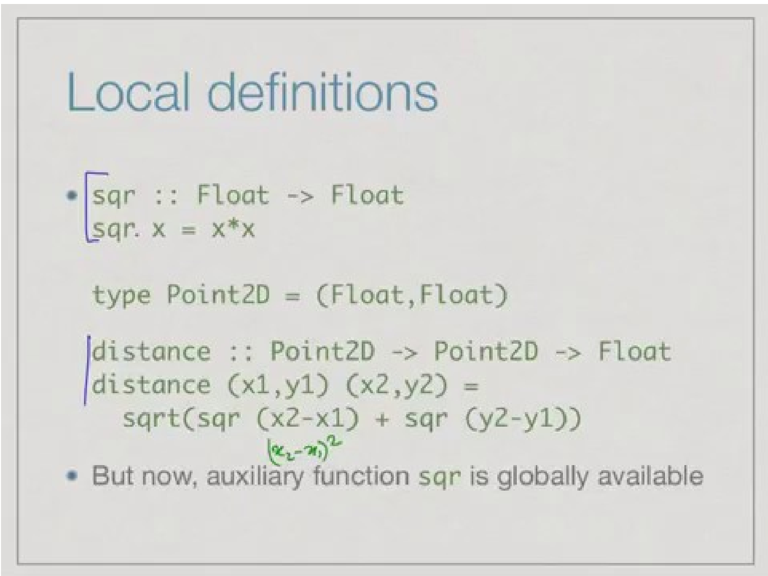
So, in every context, if I have a value of Point2D, it is exactly compatible with any expectation of a value of tuple (Float,Float). So, these are just different names for the same thing, they are not different things about.

(Refer Slide Time: 09:29)




So, let us get back to this distance function. So, this particular expression is rather tedious. So, let us say that, we want to write a function which actually takes an argument and squares it. So, we can introduce the function sqr.

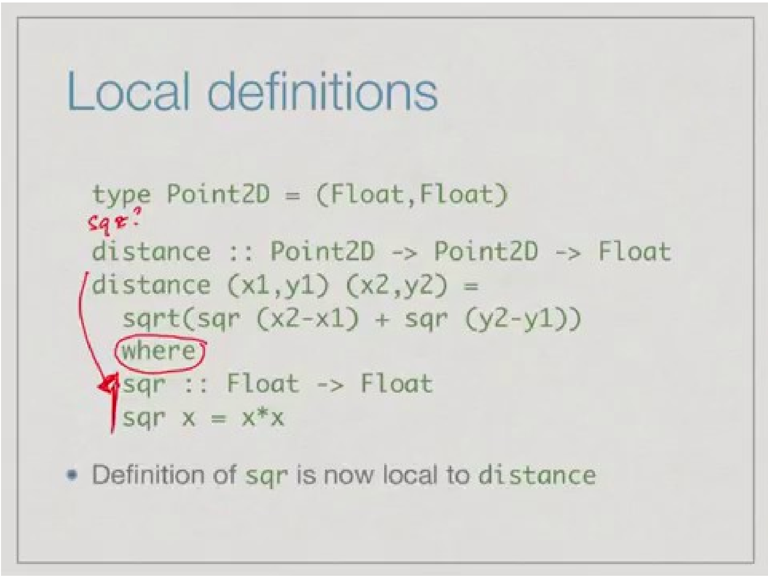
(Refer Slide Time: 09:51)




So, we can say square of x takes a Float and produces a Float and it just returns x * x, then this is a much more legible way of writing the same function. So, it says take the square root of sqr(x2- x1) + sqr(y2 - y1). So, this is much clearer that sqr(x2-x1) is (x2- x1)2, than the long thing of the multiplication as in (x2-x1)*(x2-x1).

But, now the only problem will this formulation is, that we have produced a function sqr, which is used in distance, but now becomes available everywhere in our program, because I have defined it separately. So, what we would like is to localize this auxiliary function and push it, so that, this is available only within distance and does not affect the rest of the program, I can or cannot use, may or may not want to use sqr elsewhere. So, if I want to I can write it again, but it does not conflict with this definition of sqr.

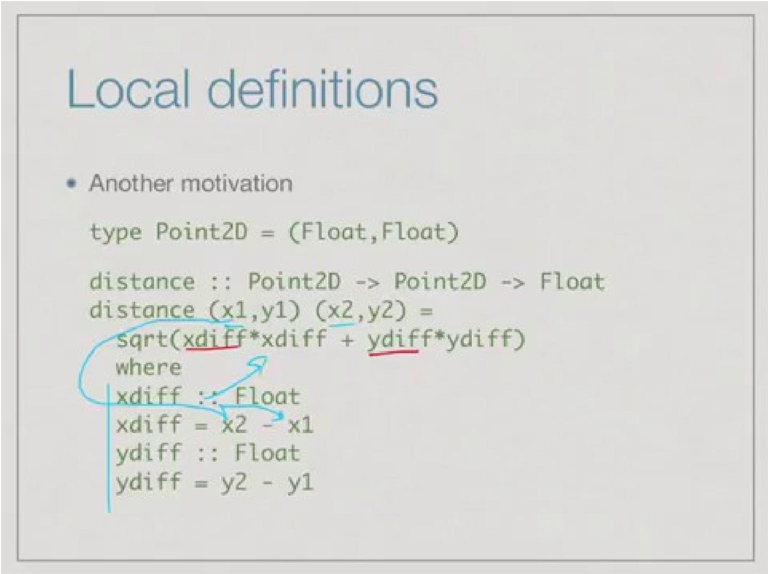
(Refer Slide Time: 10:47)




So, Haskell has this other key word, which we have not seen before called 'where'. So, Haskell says, you can write sqrt of (sqr(x2- x1) + sqr(y2- y1)) and then, say 'where' sqr is a function given then. So, this is the localized definition, now sqr is available within distance, but it is not available outside. So, distance can make use of sqr and the function sqr is defined inside.

So, it is local, its scope is only within distance. Outside, if I said sqr, there is no sqr. So, this is an obvious advantage of having local definitions, which is I mean using this where command which allows you to localize the definition.

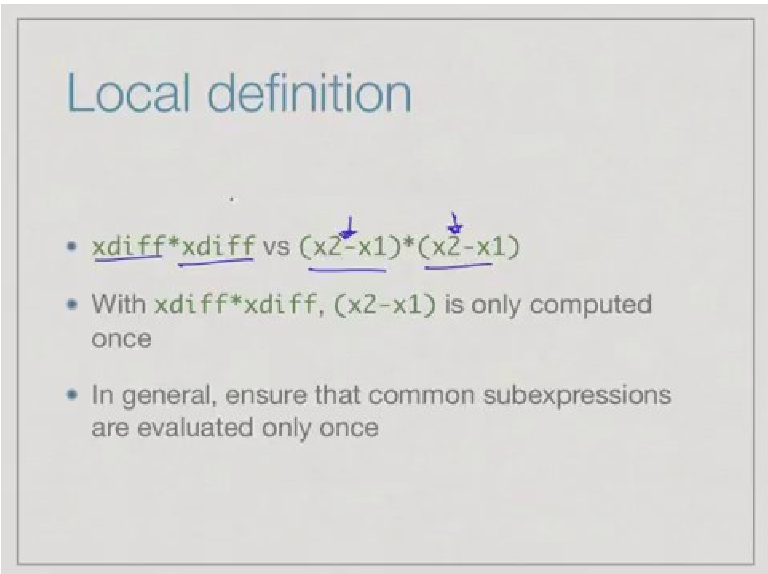
(Refer Slide Time: 11:35)




A slightly more subtle point is that, we might want to avoid duplicating a complication. So, here is another use of 'where'. So, we have not put square, we have gone back to the multiplied version, but instead of x2-x1, we have written xdiff, instead of y2-y1 we have written ydiff and then, we have said xdiff = x2-x 1, ydiff=y2-y1. So, notice one other feature here which is that the values that came into the function are available inside the where.

So, I have x2 and x1 coming from the function call and they are available inside the 'where'. So, I can refer to the arguments of the function inside the 'where', I can refer to the values defined in the 'where' in the function.

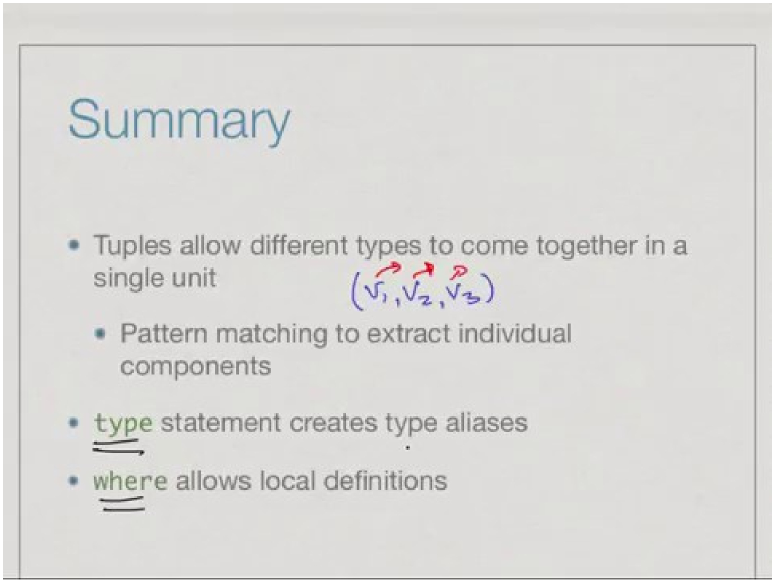
(Refer Slide Time: 12:24)



So, what does this give us, what it says is that, if I write xdiff*xdiff as opposed to x2-x1 Here, Haskell will actually not recognize, there is no way Haskell will recognize this x2-x1 is the same as this x2-x1, it does not try to optimize any calculation, so it would actually do it twice. Now, in a subtraction it might not matter but if this were a complicated function that had to be called, then if you use the 'where' and give it a new name and use the same expression with the name twice. It knows that it has to compute xdiff, but having you computed xdiff is the same xdiff here.

So, this is one way in which you can control the efficiency of the program by ensuring that the common expressions, which occur in multiple parts of your program are evaluated only once for a given set of arguments.

(Refer Slide Time: 13:12)




So, to summarize what we have seen is that lists have sequences of uniform type and if you want to combine values which are of different types into single units then we use tuples. So, tuples are written using this notation of round brackets and commas and then, we can use pattern matching to get the individual components when we write function definitions to process tuple data.

Along the way, we have also seen two new Haskell constructs, type and where. 'type' allows us to create new names for whole types. So, this is a convenience for writing programs and for making programs more legible, 'where' allows local definitions and it also helps us to avoid wasteful recomputation expressions.