Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

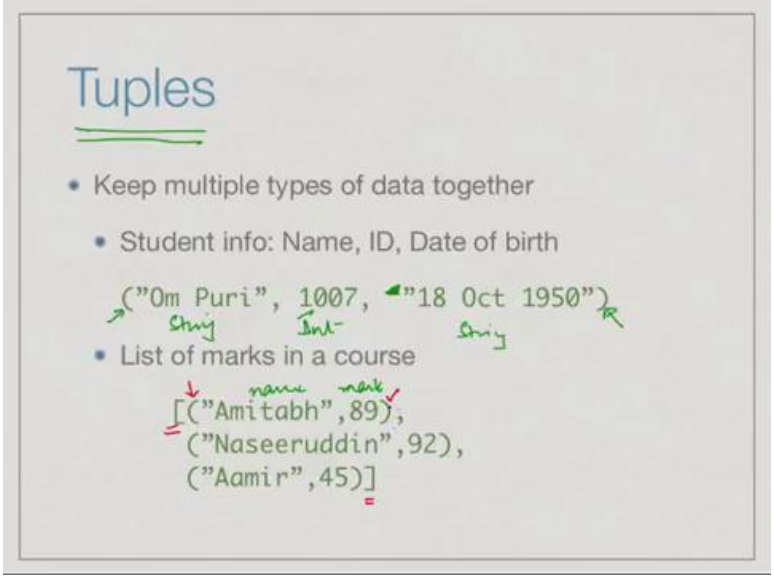
Chennai Mathematical Institute

Module # 02

Lecture - 04

Tuples

(Refer Slide Time: 00:02)



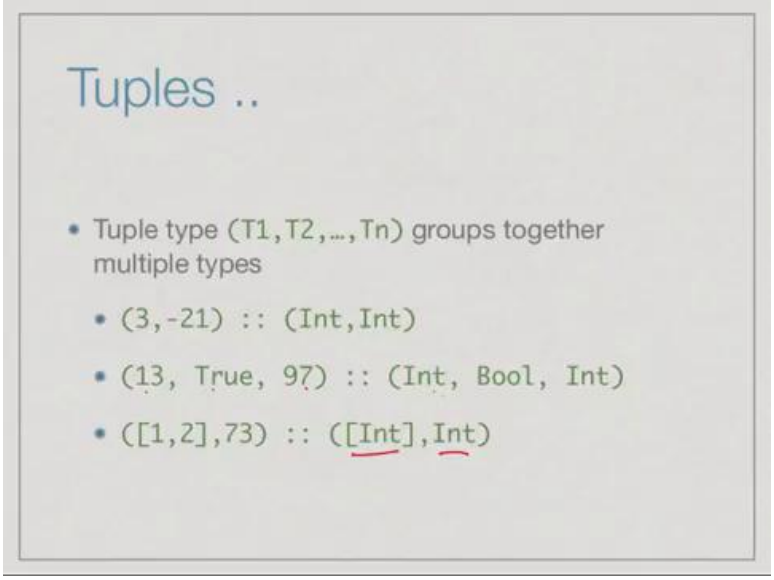
So, we have seen lists, which are sequences of uniform type and in particular, we also looked at Strings, which are the special case of lists, where the uniform type is character. But, very often we need keep multiple types of data together as a unit. So, this is what in a language like C or C plus,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

For example, if you want to deep 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 the list, because this is not a uniform type. Now, we might want to take group of such things and make it into a bigger item.

For instance, you might want to keep the list out marks for a number of students, where you keep the name and the marks together for a given students and then, we have a list of such items. So, Tuples had Haskell way of doing this, so notice that, we have use 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.

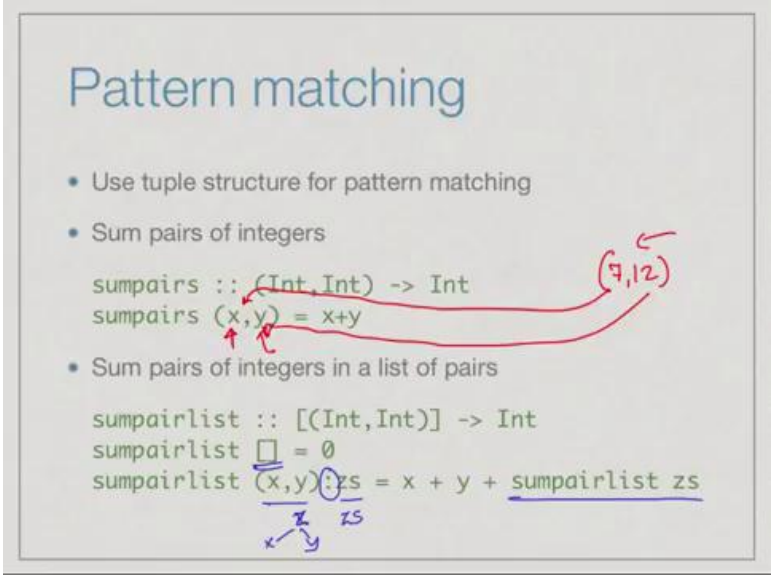
(Refer Slide Time: 01:26)



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

Where, if I have three values like this with round brackets, then the first is the Int, second is the Bool, third is an Int, each type is tuple of Int comma Bool comma 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 integers as the second component. So, here we have a tuple, which consists of lists of Int and Int.

(Refer Slide Time: 02:18)

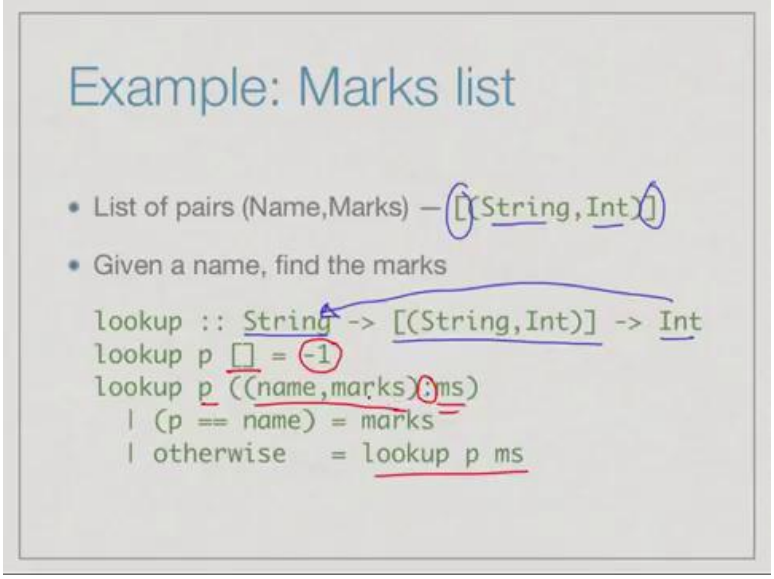


So, because the syntax is very structured, there is only one way to decompose the tuple, which is to use the comma to separate it out, so we can directly use pattern matching to extract each component. So, if I want to take pairs of integers and add them up, then I can just directly say that the first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

So, when I give it as an input says 7 comma 12, then 7 will get map to X, 12 will get map to Y, because of the pattern matching and then, the answer will be 7 plus 12, which is 19. Now, this pattern matching can be combined with another pattern matching like for lists. So, if you want to take the list of pairs and add them up across entire list, so I want X 1 plus Y 1, X 2 plus Y 2 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 the first value and a second value. So, this is the usual list pattern, it says colon use, colon to separate the X from the Z from the Z s and the Z itself is the pair. So, use X comma Y to split this Z two values X comma Y and now, I just take X plus Y as the sum of the first pair and add whatever I get by doing the list. So, tuples are particularly easy to manipulate in terms of a programs, because it is very easy to split them using patterns to get the integer components.

(Refer Slide Time: 03:48)

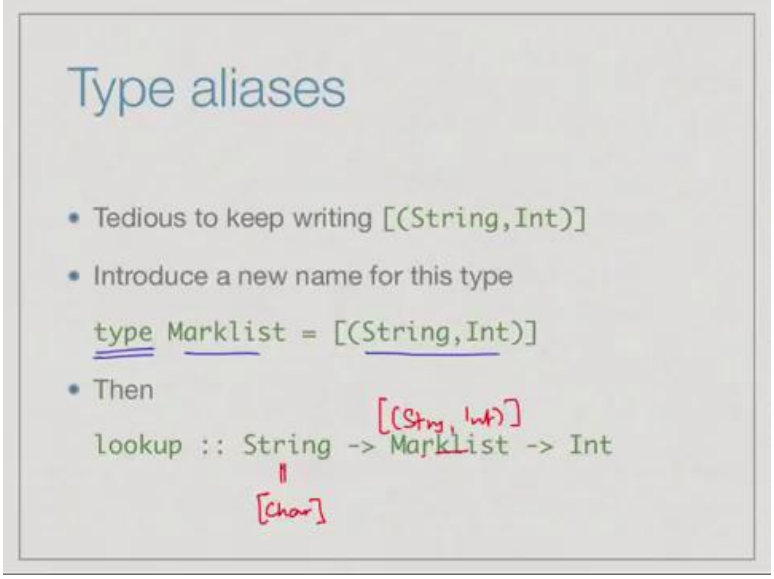


So, for examples, supposing as before we had this list of marks of students, where each pair consists of a name and the marks of that name, so we have a pair which consists of a String and an Int and we have a list of such pair. So, this is the type of our input and now you want to look up the name, 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 entire class and we want to written those marks for this particular student call.

So, we use our usual list induction, so if we have no names, then there is or we have not found this name in this list, then we have to written something. So, let us assume that everybody gets a positive mark, so as a default value, we can get minus 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)

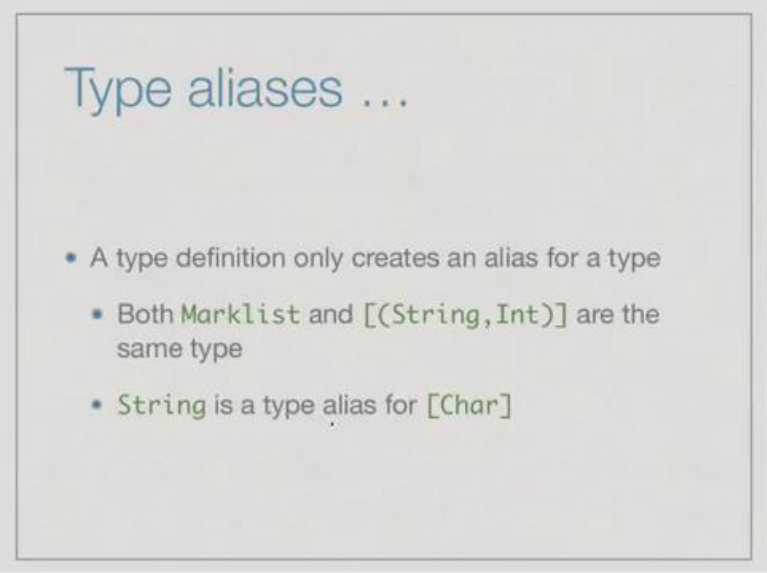


So, Haskell give us a little bit of flexibility in how to refer to these kinds of types. Very often, because we are grouping together these types as a unit, we want to maybe give it a name, which makes sense for that unit. So, we might want to say instead of writing this brace String comma Int, we might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

So, maybe, we want to give it a name like a mark list. So, Haskell as this declaration call type, which says that the name mark list is a synonym for the name String comma Int. So, this means, if we can now change or the earlier definitions. So, instead of writing here list of String comma Int has we are done earlier, once we had this type definition in our program, you can use that, instead of this type.

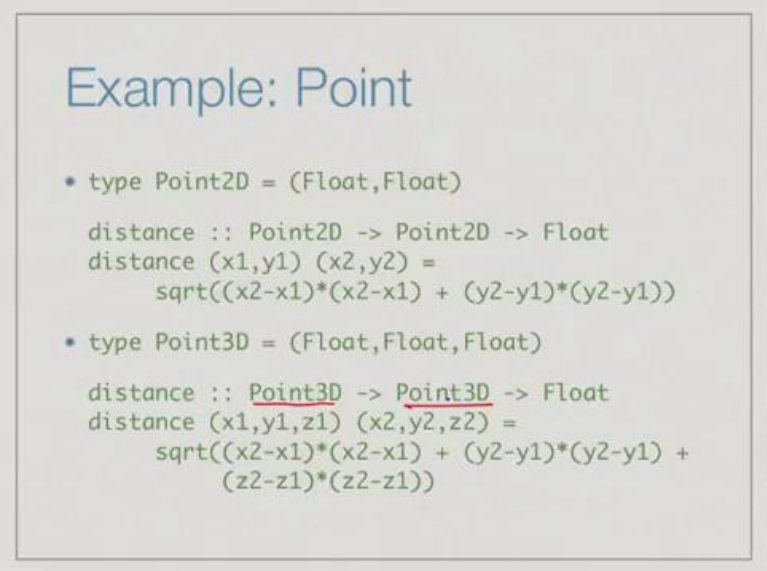
Now, this is just the synonym in the same way, that String is the same as list of char, there is no difference, there is no different between writing mark list and String of it, list of String comma Int. But, this is just very useful for us make our program more legible and also of course, cut down the unwanted steps we have to write and now, a nice thing is that is we change this slightly, then everywhere the type remains the same, provided of course, the functions are so called updated to use the new type.

(Refer Slide Time: 06:29)



So, both mark list and list of String comma Int are exactly the same type and the String is type alias, so this is not like is creating a new type. So, it is only creating a new name for a complex type.

(Refer Slide Time: 06:44)



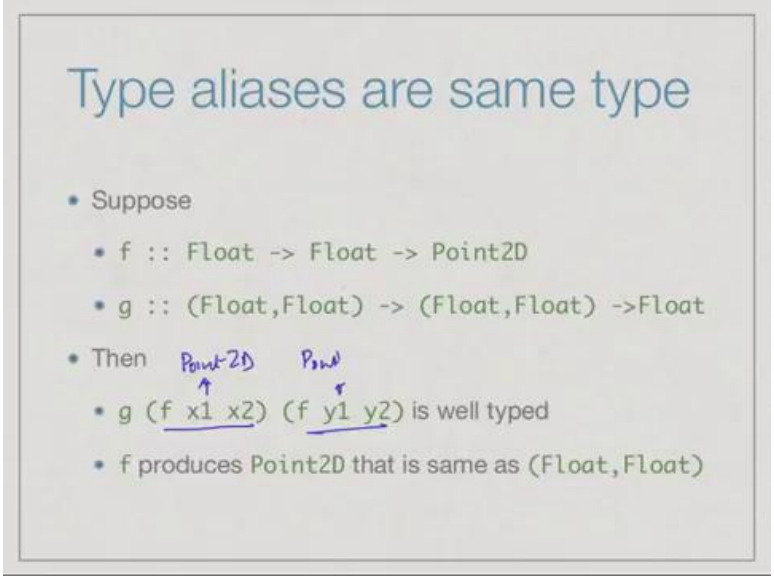
So, here is another example, so supposing we want to represent points in two dimension space by x, y coordinate, then the typical x, y coordinate will be of type Float comma Float, would be a pair of real numbers. We might want to encapsulate this Float comma Float at a single name, call it 2D point, let us use Point 2D to refer to this. then, we can compute the usual utility distance between x 1, y 1 and x 2, y 2 using Pythagoras formula, you take x 2 minus x 1 the whole square, y 2 minus y 1 whole square and take

this square root.

So, the list is between x 1, y 1 and x 2, y 2, so now, we know that the underline type is type is point 2D, Point 2D itself is Tuples. So, we can use this pattern matching for extract the coordinates without adding 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 to 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 use the Tuples to make your type of your program more meaningful. Now, you know that taking the distance between points, so it is make you program more legible.

(Refer Slide Time: 08:03)



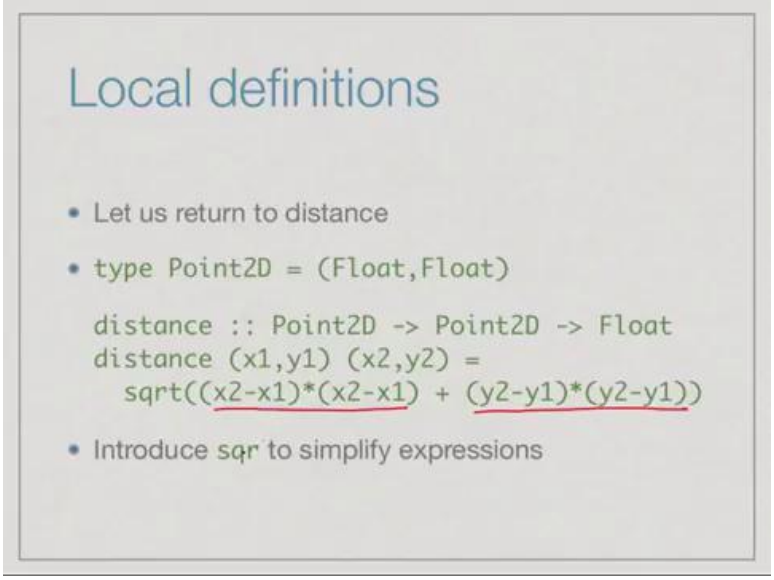
Now, to illustrate the fact that these are only type aliases, now suppose that we have a function f, which takes two Floats and produces a point. Now, remember that this is just by an earlier definition is same as Float comma Float, so we add this thing we said type Point 2D into Float comma Float. So, this definition gives us and alias, now I have an other function g, which takes Float comma Float, Float comma Float and produce a Float.

So, the whole point is this that should be the same as Point 2D and there is no difference we claim between Point 2D and Float comma Float and indeed, if I take the value of f produce for sum, so if I take f of x 1, x 2 and this produces a Point 2D. Again, if y 1 and y 2 produces a Point 2D, now I can feel these to g, g is expecting only Float comma Float

and Float comma not expecting Point 2D. But, because Point 2D and Float comma Float are exactly the same thing, Haskell will not complain about the type, this is say, this well typed.

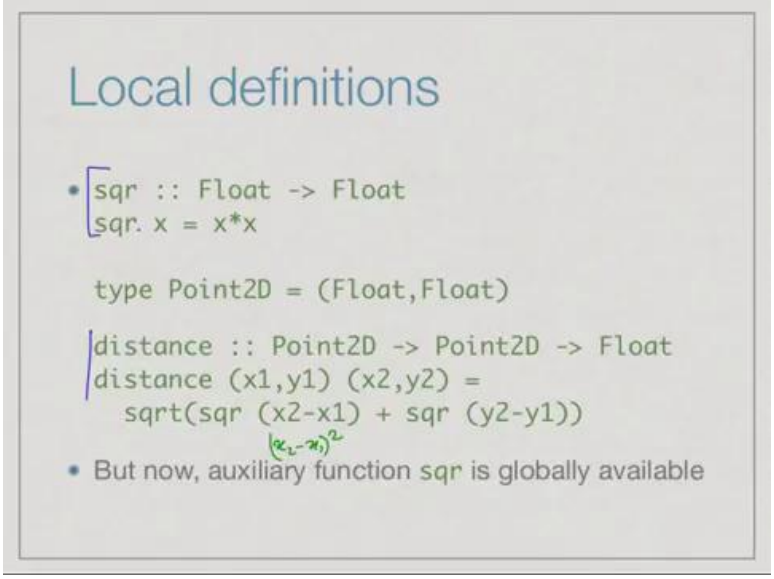
So, in every context, if I have a value of Point 2D, it is exactly compatible with any expectation of a value of tuple Float comma Float. So, these are just different names to the same thing, they are not different things about.

(Refer Slide Time: 09:29)



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

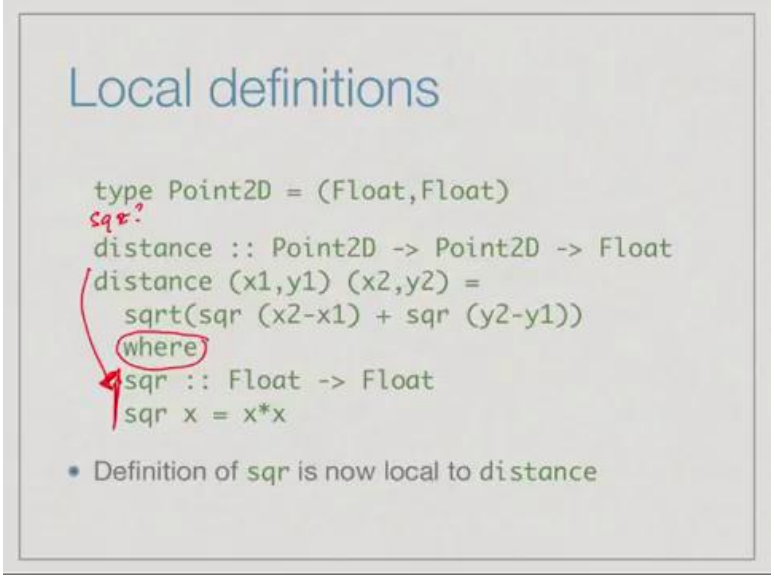
(Refer Slide Time: 09:51)



So, we can say square of x takes a Float and produces a Float and it will just written x times x, then this is the much more legible way of writing the same function. So, it says take the square root of x 2 minus x 1 square plus square plus y 2 minus y 1 square. So, this is much clearer this is x 2 minus x 1 the whole square, then the long thing the multiplication.

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 defined it separately. So, what we would like is to localize the auxiliary function and push it, so that, this is available only within distance and does not affect rest of the program, I can or cannot use, may or may not want to want to use sqr. So, if I want to I can write it again, but it does not conflict with this definition list.

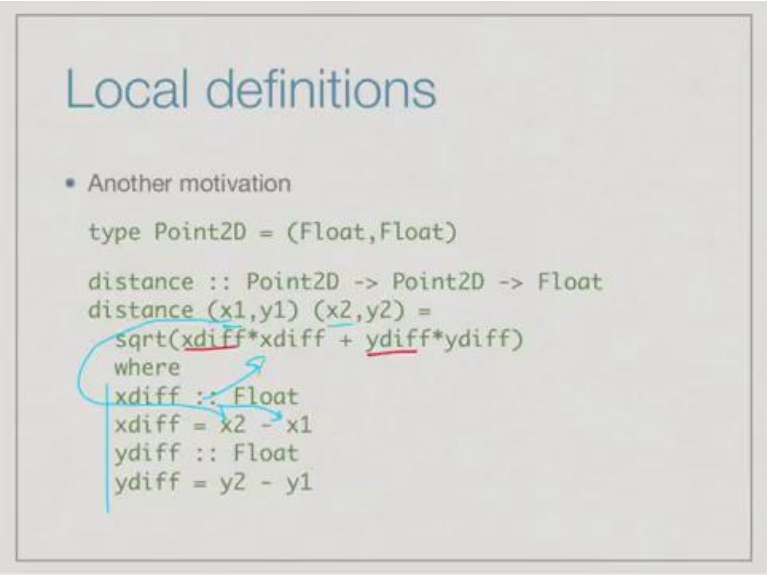
(Refer Slide Time: 10:47)



So, Haskell has this other key word, which you are not seeing before called where. So, Haskell says, you can write square root of x 2 minus x 1 whole square plus y 2 minus y 1 whole square and then, say where square is a function given then. So, this is the localized definition, now square is available within distance, but it is not available outside. So, distance can make you as a square and the function square is defined inside.

So, it is local scope is only within distance, outside, if I says square, there is no square. So, this is an obvious advantage of having local definitions, which is I mean using this square comma, which allows you the localized definition.

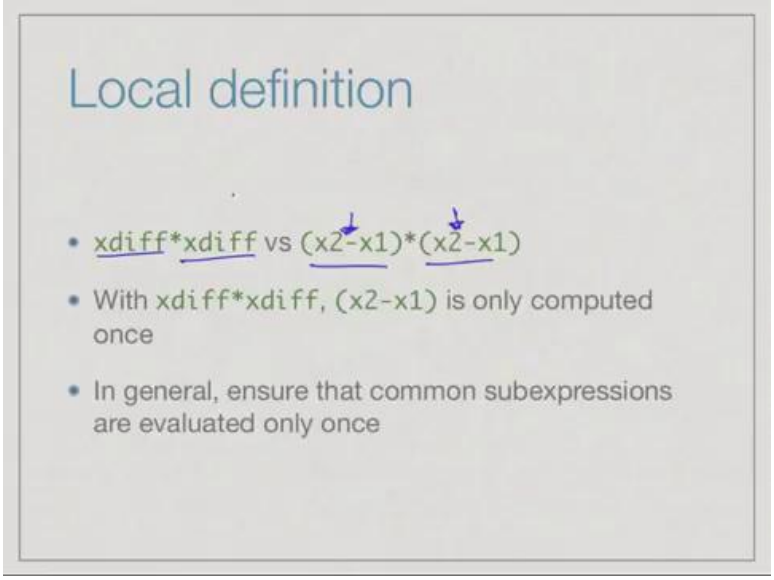
(Refer Slide Time: 11:35)



A slightly more settle point is that, we might want to avoid the duplicating a comprehension. So, here is the other use of where. So, we are not put square one back to the multiplied version, but instead of x 2 minus x 1, we have written xdiff, if instead of y 2 minus y 1 in ydiff and then, we are said xdiff minus x 1, ydiff is y 2 minus Y 1. So, notice one other feature here which is that the values at came into the function are available inside the where.

So, right x 2 and x 1 coming from the function called and they are available inside the where. So, I can refer to the argument of the function inside the where, I can refer to the values defined at the where and in the function.

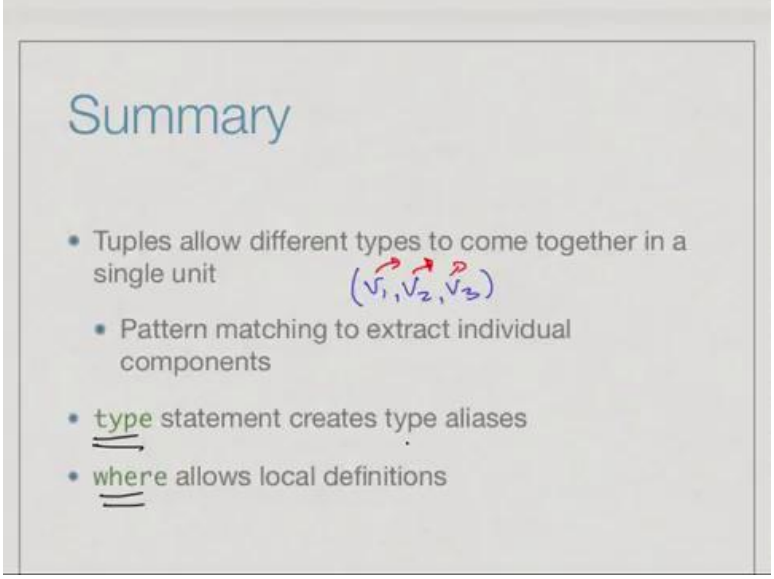
(Refer Slide Time: 12:24)



So, what is this give us while is says that, if I write xdiff times xdiff as a post to x 2 minus x 1. Here, Haskell will actually not recognized, there is no way Haskell will recognize this x 2 minus x 1 is the same as this x 2 minus x 1, it does not try to optimize any calculation, so it would actually do it twice. Now, in a subtraction cannot matter by this will complicate function had to be called, then if you use the where and give it a new name and use the same expression of 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 expression, which occurs multiple parts of the program are evaluated only one spherical set of arguments.

(Refer Slide Time: 13:12)



So, to summarize what we have seen is that list have sequences of uniform types and if you want to combines values, which are different types into single units and we use to 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 and we write the function definitions to process tuple data.

Along the way, we processing new Haskell construct type and where, type allows us to create new names for whole types. So, this is the convenient for writing programs and for making program more legible, where allows local definition and it also helps us to avoid wasteful recomputation expressions.