

Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

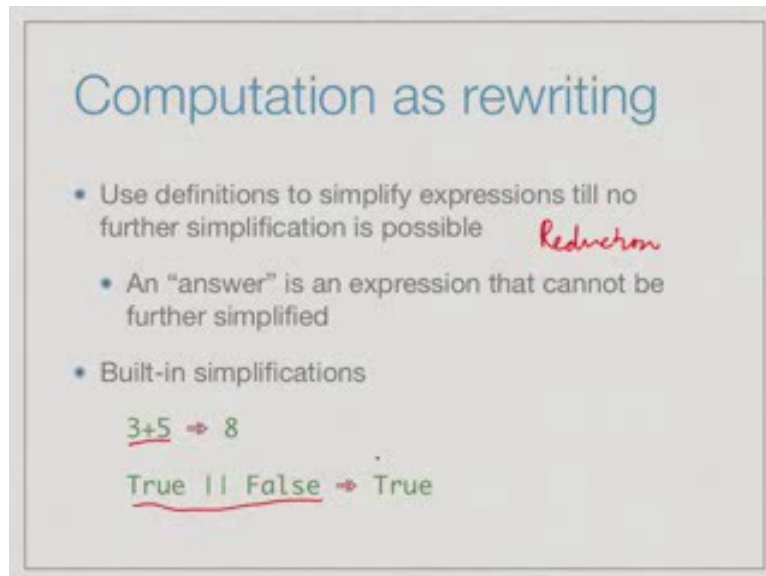
Module # 03

Lecture - 01

Computation as Rewriting

In order to explain some of the unique features that are available in Haskell, we need to understand how Haskell computations, how Haskell programs are executed; in other words, how computation progresses in Haskell.

(Refer Slide Time: 00:13)



Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible *Reduction*
- An "answer" is an expression that cannot be further simplified
- Built-in simplifications

$3+5 \Rightarrow 8$

$\text{True} \parallel \text{False} \Rightarrow \text{True}$

In general, in functional programming languages like Haskell, computation is a mechanical process of rewriting definitions. So, we have definitions which allow us to replace expressions on the left hand side by expressions on the right hand side and the process of computation is just to keep on doing this, until no further reduction is possible. So, this process of replacing left hand side by right hand side is often also called reduction.

So, when no further reduction is possible we have potentially an answer. Now, of course, if this expression is of the type that we want, if it is an Int or Bool or something, then it is a valid answer, if it is of a form which is not in the type that you like then we will get an error saying that some definition is missing.

To start with Haskell has built in definitions for the built in types. So, when we have an

arithmetic expression like $3 + 5$, then Haskell knows that it can be simplified to 8. It is not important for us how these definitions are built into Haskell, but these are obvious things that we all know and this is what it means to say that these are available to us to start with. So, similarly Boolean expressions like `True || False` would be rewritten as `True`.

(Refer Slide Time: 01:25)

Computation as rewriting

- Simplifications based on user defined functions

```

power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))

```

$x^0 = 1$
 $x^n = x \cdot x^{n-1}$

Now, in addition; obviously, we provide new definitions through our Haskell function definitions or our Haskell programs. So, here is a definition of the exponentiation function, so it says that anything raised to the power 0 is 1. So, this is the base case, this says x^0 is always 1, then it says, x^n , if it is not zero, it is going to be $x \cdot x^{n-1}$. So, this is just a basic inductive definition of the exponentiation function.

(Refer Slide Time: 01:58)

Computation as rewriting

$3^2 = 9$

- power 3 2

→ 3 * (power 3 (2-1))	user definition
→ 3 * (power 3 1)	built in simplification
→ 3 * (3 * (power 3 (1-1)))	user definition
→ 3 * (3 * (power 3 0))	built in simplification
→ 3 * (3 * 1)	user definition
→ 3 * 3	built in simplification
→ 9	built in simplification

Now, suppose Haskell is confronted with an expression of the form 3^2 that is, we are supposed to compute 3^2 . So, at this stage the only thing that can be simplified is the definition of power itself. So, using the function that we have defined we get the definition that $\text{power } x \ n$ is $x * x^{(n-1)}$. So, now, this $n-1$, $2-1$ is now an arithmetic expression that can be internally simplified. So, using the built in simplification we replace $2-1$ by 1 , so we get $3 * (\text{power } 3 \ 1)$.

Now, once again we cannot evaluate this multiplication, because the right hand side is not yet a value that can be multiplied by 3 . So, we have to again apply the definition of power and expand $\text{power } 3 \ 1$ as $3 * (\text{power } 3 \ (1-1))$. This is just a blind substitution, this is what you should remember, it just says that $\text{power } x \ n$ should be replaced by $x * (\text{power } x \ (n \text{ minus } 1))$. So, this is just a plain substitution, it does not require any intelligence on the part of the system.

It just says if I see a pattern which looks like this I can replace it with a pattern that looks like this, where the corresponding values will be transported to the current one, corresponding places. So, now, again $1-1$ is a built in expression, so we can make it 0 , now fortunately $\text{power } 3 \ 0$ has a simple form which is 1 . So, using the user definition for $\text{power } 3 \ 0$, I get to 1 and now I can start a plain arithmetic internally. So, I get $3 * 1$ is 3 , so I get $3 * 3$ and then $3 * 3$ is 9 .

And so by this process of simplifying using a combination of the definition of power which we provide and the arithmetic expressions for subtraction and multiplication which are built in to Haskell, we reach the conclusion that 3^2 is 9 . This is not because Haskell knows arithmetic or it knows anything about exponentiation, it is just that the way we have defined the computation rules, it ensures that the answer is meaningful.

(Refer Slide Time: 04:16)

Order of evaluation

- $(8+3)*(5-3) \Rightarrow 11*(5-3) \Rightarrow 11*2 \Rightarrow 22$
 $(8+3)*(5-3) \Rightarrow (8+3)*2 \Rightarrow 11*2 \Rightarrow 22$
- $\text{power } (5+2) (4-4) \Rightarrow \text{power } 7 (4-4)$
 $\Rightarrow \text{power } 7 \ 0 = 1$
 $\text{power } (5+2) (4-4) \Rightarrow \text{power } (5+2) \ 0 \Rightarrow 1$
- What would $\text{power } (\text{div } 3 \ 0) \ 0$ return?
division by zero

Now, there may be situations where more than one thing is possible, in our previous example at every stage more or less we could do only one thing, maybe at this point we could do two things, where we could multiply 3, with the bracketing we can do one thing. We do $3*1$ with the 3 but, we did not have brackets we could possibly do this in different orders. But, there will be situations where it is ambiguous, so here for instance we have an arithmetic expression which has two sub expressions which need to be evaluated.

So, I have $(8 + 3) * (5 - 3)$, so if I choose to go left to right, then I first replace $(8 + 3)$ by 11 and then replace $(5 - 3)$ by 2 and finally, we have something where we can multiply and get 22. But, we can of course, just as well do it the other way, we could start with $(5 - 3)$ and make it 2 and then move to $(8 + 3)$ and make it 11 and still we get the same answer. So, we are familiar with the fact in arithmetic that this will not matter, it does not matter with me, whether we simplify $(8 + 3)$ first or $(5 - 3)$ first.

So, either order will reach the same result, now the same could happen when we invoke a function like power. For instance, suppose we invoke power with the arguments $(5 + 2)$ and $(4 - 4)$, then if we choose to start with $(5 + 2)$ then this becomes power of 7 $(4 - 4)$. Then, I replace $(4 - 4)$ like 0 and then the power definition of 0 gives me 1, but if we do it the other way something interesting happens.

So, now, we start with $(4 - 4)$ and replace it by 0 and at this stage now there are actually two definitions that we can use. Because, the definition of power for 0's, since $\text{power } x \ 0$ is 1, in other words we could just as well have written $\text{power } _ \ 0$ is 1 because the x plays no role

here, so the `x` is not used. So, we said in the beginning that if we have a pattern in which the variable that we are matching is not used in the answer, we can as well make it an underscore. So, even though we have said `x`, the value of `x` is not used.

So, in particular here Haskell can look at the definition of `power` and that the second argument is `0` and just ignore this argument and give `1`. So, we have evaluated `power` in two ways, but the second way we have done something which is somewhat unexpected, which is that we have reached the final answer while not evaluating something in between. So, this is perhaps a curiosity, but what would be it, something like this, so we know now that `power anything 0` evaluates to `1` without looking at anything.

So, what do we replace that anything by a division by `0`, so logically if I write `div 3 0` then I should get an error, but if I say `power div 3 0 0` would Haskell expose that error or would it blindly apply the definition for `power anything 0` and give me a `1`. So, we will come back in this very shortly in the same lecture.

(Refer Slide Time: 07:33)

The slide is titled "Lazy Evaluation" in a large blue font. It contains a bulleted list explaining Haskell's evaluation strategy. The first bullet states that any Haskell expression is of the form `f e`, where `f` is the outermost function and `e` is the expression to which it is applied. The second bullet uses the example `head (2:reverse [1..5])`, where `head` is underlined in red and `(2:reverse [1..5])` is circled in green. It then lists that `f` is `head` and `e` is `(2:reverse [1..5])`. The third bullet states that when `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`.

- Any Haskell expression is of the form `f e` where
 - `f` is the **outermost** function
 - `e` is the expression to which it is applied.
- In `head (2:reverse [1..5])`
 - `f` is `head`
 - `e` is `(2:reverse [1..5])`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`.

So, Haskell therefore, has to make a choice about what order to evaluate expressions. So, we have seen that Haskell expressions are of the form `f` applied to `e` written as `f e`. So, we call `f` the outermost function and `e` the expression to which it is applied, `e` in turn could of course, contain more functions and so on, but they are inside. So, here is an example, so we have an expression which says take the head of the list, `2`, append it to reverse of `[1..5]`. So, there is an inner function `reverse` and then, there is an inner operator colon, but the outer function in this is `head`.

So, in our terminology if we say f of e , this is head of something, so that something is e and f is head. Now, what Haskell does is it prefers to use a definition which applies to f rather than definitions to apply inside e . So, in particular here if I have a choice between expanding `reverse [1..5]` and taking head of the list, then I would first try to use the expansion for head. So, if Haskell goes outermost, it tries to use the outermost definition that it can use in the current expression to reduce.

(Refer Slide Time: 08:45)

Lazy evaluation ...

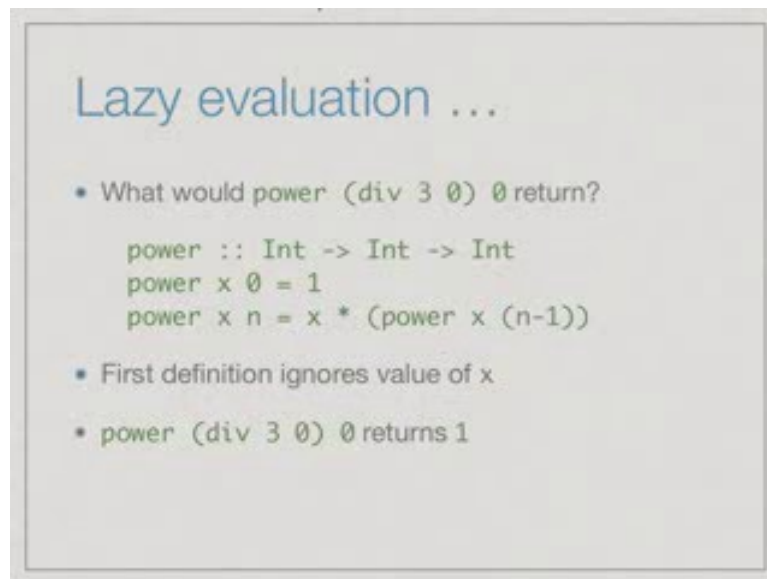
- The argument is not evaluated if the function definition does not force it to be evaluated.
 $\text{head } (x : xs) \rightsquigarrow x$
 $\text{head } (2 : \text{reverse } [1..5]) \rightsquigarrow 2$
- Argument is evaluated if needed
 $\text{last } (2 : \text{reverse } [1..5]) \rightsquigarrow$
 $\text{last } (2 : \underline{[5,4,3,2,1]}) \rightsquigarrow 1$

So, in this case it turns out for example that you can take head of this list, now `head(x:xs)` gives me x we know this. So, this is a kind of built in rule in Haskell, so in this expression the head value is known, it is 2. So, without evaluating `reverse` you can extract the head, so this is what Haskell would do, it will try to get the answer with as little work as possible, if you like to think of it that way and hence this is sometimes called lazy evaluation.

So, the argument to a function in Haskell is not evaluated, if the definition of the function does not force it to be evaluated. So, I have no interest in knowing what the result to `reverse[1..5]` is, because it is not going to be part of my answer. So, `head(2 : reverse[1..5])` just gives me 2. On the other hand if I want the last value of the list, then I do need to know what happens to the end. So, if I say `last (2 : reverse[1..5])`, then I must reverse this list and get 5, 4, 3, 2, 1.

So that, I can then conclude that last of that definition of list is 1. So, Haskell's lazy reductions starts from the outermost thing and if it finds, it has enough information to process the answer, it does not need to look at an argument, it will just ignore that argument.

(Refer Slide Time: 10:01)



Lazy evaluation ...

- What would `power (div 3 0) 0` return?

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```

- First definition ignores value of x
- `power (div 3 0) 0` returns 1

So, if we come back to the question about what happens to `power` when we give it a nonsensical argument in the first position, but 0 in the second position. So, Haskell's strategy would be to first look for a matching definition for `power`. So, it finds a matching definition of `power`, this matching definition does not require me to evaluate `x`, so it ignores `x`, so this would actually give me 1.

So, let's see how this works, so just to convince you that this is real, so here is a Haskell file which contains that exact same definition of `power`, it says `power` of `x 0` is 1, `power` of `x n` is `x` times `power` of `x n` minus 1. So, now, if I run this, I say `power` say for example `3 2` we have seen before `power 7 5` works correctly. Now, if I say `power (div 3 0) 0`, I get 1 and this is not because `div 3 0` is a sensible expression, because `div 3 0` on its own gives me as I would expect a divide by 0 error. So, lazy rewriting allows me to compute some values even though the arguments are not fully defined.

(Refer Slide Time: 11:29)

Lazy evaluation ...

power (dw 3 0) 0

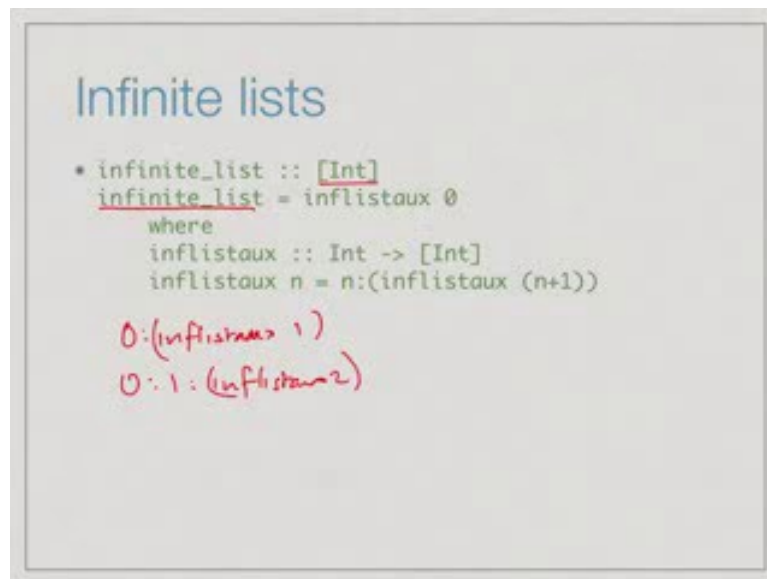
- If all simplifications are possible, order of evaluation does not matter, same answer
- One order may terminate, another may not
- Lazy evaluation expands arguments by "need"
- Can terminate with an undefined sub-expression if that expression is not used

Fortunately in general this order of evaluation given in a functional expression does not matter, just as it does for arithmetic, it does not matter. If all the simplifications are possible which ever order we choose, then the order does not matter we get the same answer. But, as we have seen if we choose one order, for example in this power example if I choose the order in which I try to evaluate the argument before the function, then I will reach a situation where the computation does not terminate or it terminates to an error, whereas if I use the outermost things, it does terminate with sensible answer.

Now, this is not to say, I mean we are not claiming that arithmetically something like infinity to the power 0 should be 1. So, its just saying that as per our rules it is possible to conclude from this expression that the answer is 1, it does not mean that it is arithmetically a sensible answer. So, the point is that using outermost reduction in this case, you get some expression which terminates, whereas if you use innermost reduction that is you start from the arguments and move outwards, then you will get an situation where your computation either does not terminate or gives some error.

So, lazy evaluation is also called call by need, it uses the arguments only if they are needed and therefore, as we have seen it can terminate even though there are undefined sub expressions which are not used in the computation.

(Refer Slide Time: 12:56)



So, why is this useful other than curiosities like power? Well, let us look at this definition, so here is a definition which creates an infinite list. So, infinite list, it says is of type [Int] and what it, how you get it is by calling an auxiliary function with an argument 0 and what does this function do, if it takes an n it sticks in front and then calls itself with n+1. So, if I call inflistaux with 0, this becomes 0 : (inflistaux 1) which in turn becomes 0 : 1 : (inflistaux 2) and so on.

So, in other words this is going, the 0 will produce a 1, the 1 will produce a 2 and so on. So, this is going to produce a infinite list starting with 0. So, we can verify that, so here we have a again the definition that we just wrote. So, we have inflist.hs, so if I put this into ghci and I ask for what inflist returns , indeed it does return infinite list, that is it just keeps going. So, Haskell is able to somehow generate this infinite list and show it to us because it generates it one at a time.

(Refer Slide Time: 14:33)

Infinite lists

- `infinite_list :: [Int]`
 `infinite_list = inflistaux 0`
 where
 `inflistaux :: Int -> [Int]`
 `inflistaux n = n:(inflistaux (n+1))`
- `infinite_list` → `[0,1,2,3,4,5,6,7,8,9,10,12,...]`
- `head (infinite_list)` → `head(0:inflistaux 1)` → `0`
- `take 2 (infinite_list)` →
 `take 2 (0:inflistaux 1)` →
 `0:(take 1 (inflistaux 1))` →
 `0:(take 1 (1:inflistaux 2))` → `[0,1]`

So, as we have seen infinite list the definition I have given above produces this infinite list and if we just run it in the interpreter and ask for the value of infinite list, we just get an unending stream of numbers. On the other hand, if we ask for some function on this list which does not use the entire stream, then we can get a sensible answer. for instance, if we ask for the head of this list then as we saw before what Haskell will do, it will try to apply a definition for head, but head will ask that I need some values in the list in order to compute the head.

So, then I will expand infinite list just enough, so infinite list will now be expanded as `inflistaux 0` which will in turn become `0 : inflistaux 1` and at this point we have a value. So, head of this will come out as `0`, we could also have a function which requires more of the infinite list, for instance suppose we wanted first two elements we say take two of this infinite list, then in order to get anywhere we first have to get the first element and then we take it out and then we have to recursively take one of the remaining.

So, again we have to get this one value out and then once we have got two values out then we can stop and say the take 2 of this list is `0,1`. So, in other words by using lazy evaluation, we can take infinite list and productively use them in our code.

(Refer Slide Time: 15:58)

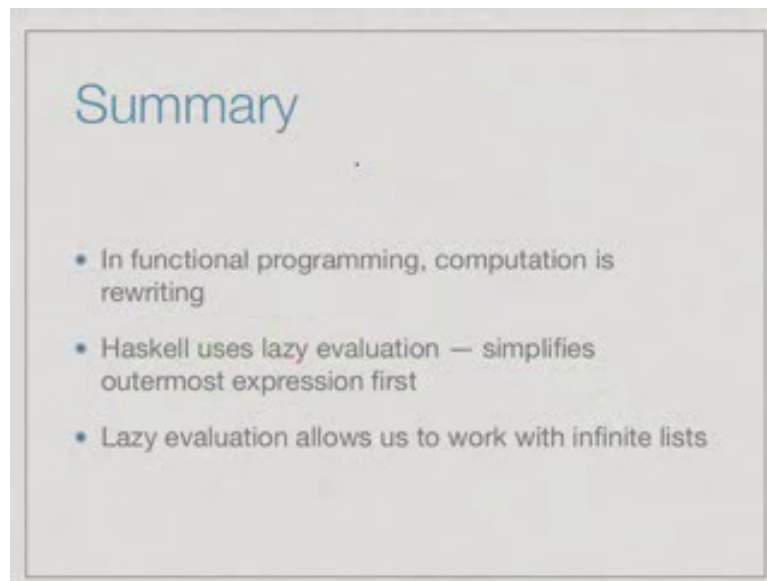
The slide is titled "Infinite lists" in blue text. Below the title, the notation $[m..n]$ is written in red. A bulleted list follows:

- Range notation extends to infinite lists
 - $[m..]$ $\Rightarrow [m, m+1, m+2, \dots]$
 - $[m, m+d..]$ $\Rightarrow [m, m+d, m+2d, m+3d, \dots]$
- Sometimes infinite lists simplify function definition

So, in fact, Haskell allows us to define infinite list directly using the range notation, if you remember we had this notation from m to n , $[m..n]$. So, now, if we leave out the upper end then we get a list which does not have a terminating point. So, $[m..]$ is just $[m, m+1, m+2, \dots]$. we can also write an infinite list with some arithmetic progression like thing. So, $[m, m+d, m+2d, \dots]$ and so on, so you might ask why this is useful well it will turn out we will see later examples, where allowing infinite list in the definition of a function makes it simpler to argue about it.

Because, if we don't know in advance what is the upper bound of the numbers we are looking for then we can just provide an infinite list and let Haskell figure out how many of these numbers it needs.

(Refer Slide Time: 16:47)



So to summarize, the process of computation in a functional programming language is rewriting or reduction and this applies rules from left to right to reduce expressions using the definition which are either built into the language or provided by the user. Now, this allows multiple orders of reduction, different sub expressions in the same expression may be available for reduction. So, Haskell uses what is called lazy evaluation, it simplifies the outermost expression first and one of the consequences of lazy evaluation is that we can now work with infinite list in Haskell.