CHARLES E. LEISERSON: Hey, everybody. Let's get going. Who here has heard of the FFT? That's most of you. So I first met Steve Johnson when he worked with one of my graduate students, now former graduate student, Matteo Frigo. And they came up with a really spectacular piece of performance engineering for the FFT, a system they call FFTW for the Fastest Fourier Transform in the West. And it has, for over years and years been a staple of anybody doing signal processing will know FFTW.

So anyway, it's a great pleasure to welcome Steve Johnson, who is going to talk about some of the work that he's been doing on dynamic languages, such as Julia and Python.

STEVEN JOHNSON: Yeah. Thanks.

CHARLES E. LEISERSON: Is that pretty actuate?

STEVEN JOHNSON: Yeah. Yeah, so I'm going to talk, as I said, about high level dynamic languages and how you get performance in these. And so most of you have probably used Python, or R, and Matlab. And so these are really popular for people doing in technical computing, statistics, and anything where you want kind of interactive exploration. You'd like to have a dynamically typed language where you can just type x equals 3. And then three lines later, you said, oh, x is an array. Because you're doing things interactively. You don't have to be stuck with a particular set of types. And there's a lot of choices for these.

But they usually hit a wall when it comes to writing performance critical code in these languages. And so traditionally, people doing some serious computing in these languages have a two-language solution. So they do high level exploration, and productivity and so forth in Python or whatever. But when they need to write performance critical code, then you drop down to a lower level language, Fortran, or C, or Cython, or one of these things. And you use Python as the glue for these low level kernels.

And the problem-- and this is workable. I've done this myself. Many of you have probably done this. But when you drop down from Python to C, or even to Cython, there there's a huge discontinuous jump in the complexity of the coding. And there's usually a lot of generality. When you write code in C or something like that, it's specific to a very small set of types, whereas the nice thing about high level languages is you can write generic code that works for a lot of different types.

So at this point, there's often someone who pops up and says, oh, well, I did performance programming in Python. And everyone knows you just need to vectorize your code, right? So basically, what they mean is you rely on mature external libraries that you pass on a big block of data. It does a huge amount of computation and comes back. And so you never write your own loops.

And this is great. If there's someone who's already written the code that you need, you should try and leverage that as much as possible. But somebody has to write those. And eventually, that person will be you. And because eventually if you do scientific computing, you run into a problem inevitably that you just can't express in terms of existing libraries very easily or at all.

So this was the state of affairs for a long time. And a few years ago, starting in Alan Edelman's group at MIT, there was a proposal for a new language called Julia, which tries to be as high level and interactive as-- it's a dynamically typed language, you know, as Matlab, or Python, and so forth. But general purpose language like Python, very productive for technical work, so really oriented towards scientific numerical computing.

But you can write a loop, and you write low level code in that that's as fast as C. So that was the goal. The first release was in 2013. So it's a pretty young language. The 1.0 release was in August of this year. So before that point every year there was a new release, 0.1, 0.2, Point3. And every year, it would break all your old code, and you'd have to update everything to keep it working.

So now they said, OK, it's stable. We'll add new features. We'll make it faster. But from this point onwards, for least until 2.0, many years in the future it will be backwards compatible. So there's lots of-- in my experience, this pretty much holds up. I haven't found any problem where there was a nice highly optimized C or Fortran code where I couldn't write equivalent code or equivalent performance, equivalently performing code in Julia given enough time,

right?

Obviously, if something is-- there's a library with 100,000 lines of code. It takes quite a long time to rewrite that in any other language. So there are lots of benchmarks that illustrate this. The goal of Julia is usually to stay within a factor of 2 of C. In my experience, it's usually within a factor of a few percent if you know what you're doing.

So there's a very simple example that I like to use, which is generating a Vandermonde matrix. So giving a vector a value as alpha 1 alpha 2 to alpha n. And you want to make an n by m matrix whose columns are just those entries to 0 with power, first power squared, cubed, and so forth element-wise. All right, so this kind of matrix shows up in a lot of problems. So most matrix and vector libraries have a built in function to do this and Python.

In NumPy, there is a function called numpy.vander to do this. And if you look at-- it's generating a big matrix. It could be performance critical. So they can implement it in Python. So if you look at the NumPy implementation, it's a little Python shim that calls immediately to C. And then if you look at the C code-- I won't scroll through it-- but it's several hundred lines of code. It's quite long and complicated.

And all that several hundred lines of code is doing is just figuring out what types to work with, like what kernels to dispatch to. And at the end of that, it dispatches to a kernel that does the actual work. And that kernel is also C code, but that C code was generated by a special purpose code generation. So it's quite involved to get good performance for this while still being somewhat type generic.

So their goal is to have something that works for basically any NumPy array and any NumPy type, which there's a handful, like maybe a dozen scalar types that it should work with, all right? So if you're implementing this in C, it's really trivial to write 20 lines of code that implements this but only for double precision, a point or two double position array, all right? So the difficulty is getting type generic in C.

So in Julia. Here is the implementation in Julia. It looks at first glance much like what roughly what a C or Fourier implementation would look like. It's just implemented the most simple way. It's just two nested loops. So just basically, you loop across. And as you go across, you accumulate powers by multiplying repeatedly by x. That's all it is. And it just fills in the array.

The performance of that graph here is the time for the NumPy implementation divided by the

time for the Julie implementation as a function of n for an n by n matrix. The first data point, I think there's something funny going on that's not 10,000 times slower. But for a 10 by 10, 20 by 20 array, the NumPy version is actually 10 times slower because it's basically the overhead that's imposed by all going through all those layers.

Once you get to 100 by 100 matrix, the overhead doesn't matter. And then it's all this optimized C code, generation and so forth is pretty much the same speed as the Julia code. Except the Julia code there, as I said, it looks much like C code would, except there's no types. It's Vander x. There's no type declaration. x can be anything.

And, in fact, this works with any container type as long as it has an indexing operation. And any numeric type-- it could be real numbers. It could be complex numbers. It could be quarternians, anything that supports the times operation. And there's also a call to 1. So 1 returns the multiplicative identity for whatever, so whatever group you're in you need to have a 1, right? That's the first column.

That might be a different type of 1 for a different object, right? It might be an array of matrices, for example. And then the 1 is the identity matrix. So, in fact. There are even cases where you can do. Get significantly faster than optimize C and Fortran codes. So I found this when I was implementing special functions, so things like the error function, or polygamma function, or the inverse of the error function.

I've consistently found that I can get often two to three times faster than optimized C and Fortran libraries out there, partly because I'm smarter than people who wrote those libraries, but-- no-- mainly because in Julia, I'm using basically the same expansions, the same series, rational functions that everyone else is using. The difference is then in Julia, it has built-in techniques for what's called metaprogramming or co-generation.

So usually, the special functions involved lots of polynomial evaluations. That's what they boil down to. And you can basically write co-generation that generates very optimized inline evaluation of the specific polynomials for these functions that would be really awkward to write in Fortran. You'd either have to write it all by hand or write a separate routine, a separate program that wrote Fortran code for you. So you can do this. It's a high level languages allow you to do tricks for performance that it would be really hard to do in low level languages.

So mainly what I wanted to talk about is give some idea of why Julia can be fast. And to understand this, you also need to understand why is Python slow. And in general, what's going

on in determining the performance in a language like this? What do you need in the language to enable you to compile it to fast code while still, still being completely generic like this Vander function, which works on any type. Even user-defined, numeric type, user-defined container type will be just as fast. There's no privileged-- in fact, if you look at Julia, almost all of Julia is implemented in Julia.

Integer operations and things like that, the really basic types, most of that is implemented in Julia, right? Obviously, if you're multiplying two 32-bit integers. At some point, it's calling an assembly language instruction. But even that, calling out to the assembly is actually in Julia. So at this point, I want to switch over to sort of a live calculation.

So this is from a notebook that I developed as part of a short course with Alan Edelman, who's sitting over there, [INAUDIBLE] on performance optimization high level languages. And so I want to go through just a very simple calculation. Of course, you would never-- in any language, usually you would have a built-- often have a built-in function for this. But it's just a sum function just written up there.

So we need to have a list, an array, of n numbers. We're just going to add them up. And if we can't make this fast, then we have real problems. And we're not going to be able to do anything in this list. So this is the simple sort of thing where if someone doesn't provide this for you, you're going to have to write a loop to do this.

So I'm going to look at it not just in Julia but also in Python, in C, and Python with NumPy and so forth. So this document that I'm showing you here is a Jupyter Notebook. Some of you may have seen this kind of thing. So Jupyter is this really nice-- they provide this really nice browser-based front end when I can put in equations, and text, and code, and results, and graphs all in one Mathematical notebook document.

And you can plug in different languages. So initially, it was for Python. But we plugged in Julia. And now there's R, and there's 30 different-- like 100 different languages that you can plug in to the same front end. OK, so I'll start with the C implementation of this.

So this is a Julia notebook, but I can easily compile and call out to C. So I just made a string that has just-- you know there's 10 lines C implementation. It's just the most obvious function that just takes in a pointer to an array of doubles and it's length. And it just loops over them and sums them up, just what you would do.

And then I'll compile it with GCC dash 03 and link it to a shared library, and load that shared library in Julia and just call it. So there's a function called C call in Julia where I can just call out to a C library with a 0 overhead, basically. So it's nice because you have lots of existing C libraries out there. You don't want to lose them. So I just say C call, and we call this c_sum function in my library. It returns a flow 64. It takes two parameters, a size t and a flow 64.

And I'm going to pass the length of my array. And the array-- and it'll automatically-- a Julia array, of course, is just a bunch of numbers. And it'll pass a pointer to that under the hood. So do that. And I wrote a little function to call relerr that computes the relative error between the fractional difference between x and y.

And I'll just check it. I'll just generate 10 to the 7 random numbers in 01 and compare that to the Julia because Julia has a built-in function called sum, that sums and array. And it's giving the same answer to 13 decimal places, so not quite machine precision, but there's 10 to the 7 numbers. So the error is kind of accumulative when you add it across.

OK so, as I'm calling it, it's giving the right answer. And now I want to just benchmark the C implementation, use that as kind of a baseline for this. This should be pretty fast for an array of floating point values. So there's a Julia package called benchmark tools. As you probably know from this class, benchmarking is a little bit tricky. So this will take something, run it lots of times, collect some statistics, return the minimum time, or you can also get the variance and other things.

So I'm going to get that number. B time is something called a macro in Julia. So it takes an expression, rewrites it into something that basically has a loop, and times it, and does all that stuff. OK, so it takes 11 milliseconds to sum 10 to the 7 numbers with a straight C, C loop compiled with jcc-03, no special tricks, OK? And so that's 1 gigaflop, basically, billion operations per second. So it's not hitting the peak rate of the CPU.

But, of course, there's additional calculations. This array doesn't fit in cache, and so forth. OK. So now let's-- before I do anything in Julia, let's do some Python. But I'll do a trick. I can call Python from Julia, so that way I can just do everything from one notebook using a package I wrote called PyCall. And PyCall just calls directly out to lib Python.

So with no virtually no overhead, so it's just like calling Python from within Python. I'm calling directly out to lib Python functions to call. And I can pass any type I want, and call any function, and do conversions back and forth. OK, so I'm going to take that array. I'll convert it to a

Python list object. So I don't want to time the overhead of converting my array to a Python array. So I'll just convert ahead of time.

And just start with a built-- Python has a built-in function called sum. So I'll use the built-in sum function. And I'll get this Py object for it. I'll call it PySum on this list and make sure it's giving the right answer OK is the difference is 10 to the minus 13 again. And now let's benchmark it. Oops. There we go.

So it takes a few seconds because it has to run it a few times and catch up with statistics. OK, so it takes 40 milliseconds. That's not bad. It's actually it's four or five times slower than C, but it's pretty good, OK? So and why is it five times slower than C? Is it is it because-- the glib answer is, oh, well, Python is slow because it's interpreted.

But the sum function is actually written in C. Here's the C implementation of the sum function that I'm calling. And I'm just linking to the GitHub code. There's a whole bunch of boilerplate that just checks with the type of the object, and then has some loops and so forth. And so if you look carefully, it turns out it's actually doing really well. And the reason it does really well is it has a fast path. If you have a list where everything is a number type, so then it has an optimized implementation for that case. But it's still five times slower than C.

And they've spent a lot of work on it. It used to be 10 times slower than C a couple of years ago. So they do a lot of work on optimizing this. And so why aren't they able to get C speed? Since they have a C implementation of a sum, are they just dumb, you know? No. It's because the semantics of the data type prevent them from getting anything faster than that.

And this is one of the things you learn when you do high level performance in high level languages. You have to think about data types, and you have to think about what the semantics are. And that that greatly constrains what any conceivable compiler can do. And if the language doesn't provide you with the ability to express the semantics you want, then you're dead. And that's one of the basic things that Julia does. So what does a Python list?

Right, so you have-- you can have three, four, right? A Python list is a bunch of objects, Python objects. But the Python numbers can be anything. They can be any type. So it's completely heterogeneous types. So, of course, a particular list like in this case can be homogeneous. But the data structure has to be heterogeneous because, in fact, I can take that homogeneous thing. At any point, I can assign the third element to a string, right? And it has to support that.

. .

So think about what that means for how it has to be implemented in memory. So what this has to do-- so this is a list of-- in this case, three items. But what are those items? So if they can be an item of any type, they could be things that-- they could be another array, right? It could be of different sizes and so forth. You don't want to have an array where everything is a different size, first of all. So it has to be this an array of pointers where the first pointer is 3, turned out to be 3. The next one is four. The next one is 2, all right?

But it can't just be that. It can't just be pointer to-- if this is a you 64-bit number, it's can't just be pointer to one 64-bit value in memory, because it has to know. It has to somehow store what type the subject is. So there has to be a type tag this says this is an integer. And this one has to have a type tag that says it's a string. So this is sometimes called the box.

So you have a value, you have a type tag plus a value. And so imagine what even the most optimized C imitation has to do given this kind of data structure. OK, here's the first element. It has to chase the pointer and then ask what type of object is it, OK? Then depending on what type of object is it, so I initialize my sum to that. Then I read the next object. I have to chase the second pointer, read the type tag, figure out the type. This is all done at run time, right?

And then, oh, this is another integer that tells me I want to use the plus function for two integers, OK? And then I read the next value, which maybe-- which plus function it's using depends upon the type of the object. It's an to object-oriented language. I can define my own type. If it has its own plus function, it should work with sum.

So it's looking up the types of the objects at runtime. It's looking at the plus function at runtime. And not only that, but each time it does a loop iteration it has to add two things and allocate a result. That result in general might be another-- it has to be a box because the type might change as you're summing through. If you start with integers, and then you get a floating point value, and then you get an array, the type will change. So each time you do a loop iteration, it allocates another box.

So what happens is the C implementation is a fast path. If they're all integer types, I think it doesn't reallocate that box for the sum it's accumulating all the time. And it caches the value of the plus function it's using. So it's a little bit faster. But still, it has to inspect every type tag and chase all these pointers for every element of the array, whereas the C implementation of sum, if you imagine what this is this compiles down to, for each loop iteration, what does it do?

It increments a pointer to the next element. At compile time, the types are all flow 64. So it flushes flow 64 value into a register. And then it has a running sum in another register, calls one machine instruction to add that running sum, and then it checks to see if we're done, an if statement there, and then goes on, all right? So just a few instructions and in a very tight loop here, whereas each loop iteration here has to be lots and lots of instructions to chase all these pointers to get the type tag-- and that's in the fast case where they're all the same type and it's optimized for that.

So where was I? So wrong thing. So that's the Python sum function. Now most many of you, you've used Python know that there is another type of array and there's a whole library called NumPy for working with numerics. So what problem is that addressing? So the basic problem is this data structure. This data structure, as soon as you have a list of items-- it can be any type-- you're dead, right? There's no way to make this as fast as a C loop over a double pointer.

So to make it fast, what you need to have is a way to say, oh, every element of this array is the same type. So I don't need to store type tags for every element. I can store a type tag once for the whole array. So there, there is a tag. There is a type, which is, say, float 64, OK? There is maybe a length of the array. And then there's just a bunch of values one after the other.

So this is just 1.0, 3.7, 8.9. And each of these are just 8 bytes, an 8-byte double in C notation, right? So it's just one after the other. So it reads this once, reads the length, and then it dispatches to code that says, OK, now-- basically dispatches to the equivalent of my C code, all right? So now once it knows the type and the length, then OK, it says it runs this, OK?

And that can be quite fast. And the only problem is you cannot write, implement this in Python. So Python doesn't provide you a way to have that semantics to have a list of objects where you say they all have to be the same type. There is no way to enforce that, or to inform the language of that in Python. And then to tell it-- oh, for this, since these are all the same type, you can throw away the boxes.

Every Python object looks like this. So there's no way to tell Python. Oh, well, these are all the same types. You don't need to store the type tags. You don't need to have pointers. You don't need to have reference counting. You can just slam the values into memory one after the other. It doesn't provide you with that facility, and so there's no way to-- you can make a fast Python compiler that will do this.

So NumPy is implemented in C. Even with-- some of you are familiar with PyPy, which is an attempt to make a fast-- like a tracing jit for Python. So when they poured into NumPy to PyPy, or they attempted to, even then they could implement more of it in Python. But they had to implement the core in C.

OK, but given that, I can do this. I can import the NumPy module into Julia, get its sum function, and benchmark the NumPy sum function, and-- OK, again, it takes a few seconds to run. OK, and it takes 3.8 milliseconds. So the C was 10 milliseconds. So it's actually doing faster than the C code, almost a little over twice as fast, actually.

And what's going on is their C code is better than my C code. Their C code is using SIMD instructions. So and at this point, I'm sure that you guys all know about these things where you can read in two numbers or four numbers into one giant register and one instruction add all four numbers at once.

OK, so what about if we go in the other direction? We write our own Python sum function. So we don't use the Python sum implemented in C. I write our own Python. So here is a little my sum function in Python. Only works for floating point. Oh, I initialize S to 0.0. So really, it only accumulates floating point values. But that's OK.

And then I just loop for x and a, s equals s plus x, return s is the most obvious thing you would write in Python. OK, and checked that it works. Yeah, errors 10 the minus 13th. It's giving the right answer. And now let's time it.

So remember that C was 10 milliseconds. NumPy was 5 milliseconds, and then built-in Python was like, the sum was 50 milliseconds operating on this list. So now we have C code operating on this list with 50 milliseconds. And now we have Python code operating on this list. And that is 230 milliseconds, all right? So it's quite a bit slower.

And it's because, basically, in Python, there's-- in the pure Python code, there's no way to implement this fast path that checks-- oh they're all the same type, so I can cash the plus function and so forth. I don't think it's feasible to implement that. And so basically then on every loop iteration has to look up the plus function dynamically and allocate a new box for the result and do that 10 to 7 times.

Now, so there's a built-in sum function in Julia. So [INAUDIBLE] benchmark that as a-- it's actually implemented in Julia. It's not implemented in C. I won't show you the code for the

built-in one because it's a little messy because it's actually computing the sum more accurately than the loop that have done. So that's 3.9 milliseconds. So it's comparable to the NumPy code, OK? So it's also using SIMD, but so this is also fast.

So now so why can Julie do that? So it has to be that the array type, first of all, has the type attached to it. So you can see the type of the array is an array of low 64. So there's a type tag attached to the array itself. So somehow, that's involved. So it looks more like an NumPy ray in memory than a Python list.

You can make the equivalent of a Python list that's called an array of any. So if I convert this to an array of any, so an array of any is something where the elements types can be any Julia type. And so then it has to be stored as something like this as an array of pointers to boxes. And when I do that-- let's see-- [INAUDIBLE] there it is-- then it's 355 milliseconds. So it's actually even worse than Python.

So the Julia-- the Python, they spent a lot of time optimizing their code past for things that had to allocate lots of boxes all the time. So in Julia, it's usually understood that if you're writing optimized code you're going to do it not on arrays of pointers to boxes. You're going to write on homogeneous arrays or things where the types are known at compile time. OK, so let's write our own Julia sum function. So this is a a Julia sum function. There's no type declaration. It works on any container type.

I initialize s for this function called 0 for the element type of a. So it initializes into the additive identity. So it will work on any container of anything that supports a plus function that has an additive identity. So it's completely generic. It looks a lot like the Python code except for there's no 0 function in Python. And let's make sure it gives the right answer. It does.

And let's benchmark it. So this is the code you'd like to be able to right. You'd like to be able to write high level code that's a straight level straight loop. Unlike the C code, it's completely generic, right? It works on any container, anything you can loop over, and anything that has a plus function, so an array of quarternians or whatever. And a benchmark, it's 11 milliseconds. It's the same as the C code I wrote in the beginning.

It's not using SIMD. So the instructions are-- that's where the additional factors of 2 come. But it's the same as the non SIMD C code. And, in fact, if I want to use SIMD, there is a little tag you can put on a loop to tell-- it says compile this with llvm. Tell llvm to try and vectorize the loop. Sometimes it can. Sometimes it can't. But something like this, it should be able to

vectorize it simple enough. You don't need to hand code SIMD instructions for a loop this simple. Yeah?

AUDIENCE: Why don't you always put the [INAUDIBLE]?

STEVEN JOHNSON: So a yeah, why isn't it the default? Because most code, the compiler cannot autovectorize. So it increases the completion time and often blows to the code size for no benefit. So it's only really-- it's really only relatively simple loops on doing simple operations and arrays that benefit from SIMD. So you don't want it to be there.

Yeah, so now it's 4.3 milliseconds. So it's about the same as the NumPy and so forth. It's a little slower than the NumPy. It's interesting. A year ago when I tried this, it was almost exactly the same speed as the NumPy. And then since then, both the NumPy and the Julia have gotten better. But the NumPy got better more. So there's something going on with basically how well the compiler can use AVX instructions by-- it seems like we're still investigating what that is. But it's an llvm limitation looks like.

So as it's still completely type generic. So it I make a random array of complex numbers, and then I sum them-- which one am I calling now? Am I calling the vector? My sum is the vectorized one, right? So complex numbers-- each complex number is two floating point numbers. So it should take about twice the time to naively think, right? So twice the number of operations to add, the same number of complex numbers, the same number of real numbers.

Yeah, and it does. It takes about 11 milliseconds. So 10 to the 7, which is about twice the 5 milliseconds it took for the same number of real numbers. And the code works for everything. So why? OK, so what's going on here? So-- OK, so we saw this my sum function. I'll just take out the SIMD for now. And we did all that. OK. OK, and it works for any type. It doesn't even have to be an array.

So, for example, there is another container type called a set in Julia, which is just an unordered collection of unique elements. But you can also loop over it. If it's a set of integers, you can also summit. And I'm waiting for the benchmarks to complete. So let me allocate a set. It says there's no type declarations here. Mysum a-- there was no a-- has to be an array of particular type or it doesn't even have to be an array, right? So set is a different data structure. And so it's a set of integers.

The sets are unique. If I add something to already the set-- it's in there-- it won't add it twice.

And it supports the fast checking. Is 2 in the set? Is 3 in the set? It doesn't have to look through all the elements. It's a hash internally. But I can call my mysum function on it, and it sums up 2 plus 17 plus 6.24, which is hopefully 49, right? So OK, so what's going on here?

So suppose you define-- the key, one of the keys, there's several things that are going on in to make Julia fast. One key thing is that when you have a function like this mysum, or even here's a simpler function-- f of x equals x plus 1-- when I call it with a particular type of argument, like an integer, or an array of integers, or whatever, then it compiles a specialized version of that for that type.

So here's f of x equals x plus 1. It works on any type supporting plus. So if I call f of 3-- so here I'm passing a 64-bit integer. When it did that, it says, OK, x is a 64-bit integer. I'm going to compile a specialized version of f with that knowledge. And then when I call with a different type, 3.1, now x is a floating point number. It will compile a specialized version with that value.

If I call it with another integer, it says, oh, that version was already compiled. [INAUDIBLE] I'll re-use it. So it only compiles it the first time you call it with a particular type. If I call it with a string, it'll give an error because it doesn't know how to add plus to a string. So it's a particular-- OK, so what is going on?

So we can actually look at the compiled code. So the function is called code, these macros called code llvm and code native. They say, OK, when I call f of 1, what's the-- do people know what llvm is? You guys-- OK, so llvm compiles to byte code first and then it goes to machine codes. So you can see the llvm bit code or byte code, or whatever it's called. And you can see the native machine code. So here's the llvm byte code that it compiles to [INAUDIBLE].

So it's a bit called add i640 basically one llvm instruction, which turns into one machine instruction, load effective address is actually a 64-bit edition function instruction. So let's think about what had to happen there. So you have f of x equals x plus 1.

Now you want to compile for x is a int 64, so 64-bit integer, or in Julia, we'd say x colon colon int 64. So colon colon means that this is of this type, OK? So this is 64-bit integer type. So what does it have to do? It first has to figure out what plus function to call. So plus, it has lots of-- there is a plus for two matrices, a plus for lots of different things. So depending on the types of the arguments, it decides on which plus function to call.

So it first realizes, oh, this is an integer. Oh, this is an integer. This is also a 64-bit integer. So

that means I'm going to call the plus function for two integers. So I'm going to look into that function. And then, oh, that one returns an int 64. So that's a return value for my function. And oh, by the way, this function is so simple that I'm going to inline it. So it's type specializing.

And this process of going from x is an integer to that, to figure out the type of the output, is called type inference, OK? So. in general, for type inference, it is given the types of the inputs, it tries to infer the types of the outputs and, in fact, all intermediate values as well.

Now what makes it a dynamic language is this can fail. So in some languages like ml or some other languages, you don't really declare types. But they're designed so they could give the types the inputs. So you can figure out everything. And if it can't figure out everything, it gives an error, basically. It has to infer everything. So Julia is a dynamic language. This can fail and have a fallback. If it doesn't know the type, it can stick things in a box.

But obviously, the fast path is when it succeeds. And one of the key things is you have to try and make this kind of thing work in a language. You have to design the language so that at least for all the built-in constructs, the standard library, in general, in the culture for people designing packages and so forth, to design things so that this type inference can succeed.

And I'll give a counter example to that in a minute, right? So and this works recursively. So it's not suppose I define a function g of x equals f of x times 2, OK? And then I called g of 1. So it's going to say, OK, x here is an integer, OK? I'm going to call f with an integer argument. Oh, I should compile f for an integer argument, figure out its return type, use its returned type to figure out what time's function to call and do all of this at compile time, right? Not at runtime, ideally.

So we can look at the llvm code for this. And, in fact, so remember, f of x adds 1 to x. And then we're multiplying by 2. So the result computes 2x plus 2. And llvm is smart enough. So f is so simple that it inlines it. And then llvm is smart enough that I don't have to-- well, I know it does it by one shift instruction to multiply x by 2. And then it adds 2. So it actually combines the times 2 and the plus 1. So it does constant folding, OK?

And it can continue on. If you look at h of x equals g of x times 2, then that compiles to 1 shift instruction to multiply x by 4 and then adding 4. So you want the-- so this process cascades. So you can even do it for recursive function. So here's a stupid implementation of the Fibonacci number and calculation of recursive limitation, right? So this is given n. It's an integer.

OK, if n is less than 3, returns 1. Otherwise, it adds the previous two numbers. I can compute the first call, listen to the first 10 integers. Here's the first 10 Fibonacci numbers. There's also an acute notation in Julia. You can say fib dot. So if you do f dot arguments, it calls the function element Y's on a collection and returns a collection. So F dot 1 to 10 returns the first 10 Fibonacci numbers.

And I can call-- there's a function called code 1 type that'll tell me what-- it'll tell me the output of type inference. N is a 1. And it goes through-- this is kind of a hard to read format. But this is like the-- after one of the compiler passes called lowering, but yeah. It's figure out the types of every intermediate calls.

So here it's invoking main dot fib. It's recursively. And it's figured out that the return type is also int 64. So it knows everything, OK? So you'll notice that here I declared a type. I've said that this is an integer, OK? I don't have to do that for type inference. This doesn't help the compiler at all because it does type inference depending on what I pass. So what this is is more like a filter.

It says that if I pass-- this function only accepts integers. If you pass something else, you should throw an error. Because if I don't want this function, because if I pass 3.7, if I fill out any number, if you look at the 3.7, I can check whether it's less than three. You can call it recursively. I mean, the function would run. It would just give nonsense, right? So I want to prevent someone from passing nonsense for this.

So that's one reason to do a type declaration. But another reason is to do something called dispatch. So what we can do is we can define different versions of the function for different arguments. So. for example. another nicer version of that is a factorial function. So here is a stupid recursive implementation of a factorial function that takes an integer argument and just recursively calls itself an n minus 1. You can call it a 10 factorial, OK?

If I want 100 factorial, I need to use a different type, not 64-bit integers. I need some arbitrary precision integer. And since I said it was an integer, if I call in 3.7, it'll given an error. So that's good. But now I can find a different version of this. So actually, there is a generalization of factorial to arbitrary real, in fact, even complex numbers called a gamma function. And so I can define a fallback that works for any type of number that calls a gamma function from someplace else.

And then if I can pass it to floating point value, I can-- if you take the factorial minus 1/2, it turns out that's actually a square root of pi. So if I square it, it gives pi, all right? So now I have one function and I have two methods, all right? So these types here, so there's a hierarchy of types. So this is what's called an abstract type, which most of you have probably seen.

So there's a type called number. And underneath, there's a class of subtypes called integer. And underneath, there is, for example, int 64 or int 8 for 64-bit integers. And underneath number there's actually another subtype called real. And underneath that there's a couple of subtypes. And then there's say flow 64 or float 32 for a single precision 32-bit floating point number and so forth. So there's a hierarchy of these things.

When I specify something can take integer I'm just saying so this type is not help the compiler. It's to provide a filter. So this method only works for these types. And this other method only works-- my second method works for any number type. So I have one thing that works for any number. Whoops. Here it is-- One that works for any number type, and one method that only works for integers. So when I call it for 3, which ones does it call, because it actually called both methods.

And what it does is it calls the most specific one. It calls the one that sort of farthest down the tree. So if I have a method defined for number and one defined for integer, if I pass an integer, it'll do this. If I have one that's defined for number, one that's defined by integer, and one that's defined specifically for int 8, and I call it a --pass an 8 bit integer, it'll call that version, all right? So it gives you a filter.

But, in general, you can do this on multiple arguments. So this is like the key abstraction in Julia, something called multiple dispatch. So this was not invented by Julia. I guess it was present in Small Talk, and Dylan. It's been in a bunch of languages. It's been floating around for a while. But it's not been in a lot of mainstream languages, not in a high performance way. And you can think of it as a generalization of advertising and programming.

So I'm sure all of you have done object oriented programming in Python or C++ or something like this. So in object-oriented programming typically the way you think of it is this is you save an object. It's usually spelled object dot method xy for example, right? And what it does, is this type, the object type determines the method, right?

So you can have a method called plus. But it would actually call a different class function for a complex number or a real number, something like that, or a method called length, which for a

Python list would call a different function than for an NumPy array, OK? In Julia, the way you would spell the same thing would you'd say method. And you wouldn't say object dot method.

So you don't think of the-- here, you think of the object as sort of owning the method. all right? And Julia-- the object would just be maybe the first argument. In fact, under the hood, if you looking in Python, for example, the object is passed as an implicit first argument called self, all right? So it actually is doing this. It's just different spelling of the same thing. But as soon as you read it this way, you realized what Python and what op languages are doing is they're looking at the type of the first argument to determine the method. But why just the first argument?

In a multiple dispatch language, you look at all the types. So this is sometimes-- in Julia, this will sometimes be called single dispatch because determining the method is called dispatch, figuring out which function is spelled length, which function actually are you calling this dispatching to the right function. So this is called multiple dispatch.

And it's clearest if you look at something like a plus function. So a plus function, if you do a plus b, which plus you do really should depend on both a and b, right? It shouldn't depend on just a or just b. And so it's actually quite awkward in languages, in o op languages like Python or especially C++ to overload a plus operation that operates on sort of mixed types. As a consequence, for example, in C++ there's a built-in complex number type.

So you can have a complex float, or complex double, complex and complex with different real types. But you can't add a complex float to a complex double. You can't add a single-precision complex number to a double-precision complex number or any mixed operation because-- any mixed complex operation because it can't figure out who owns the method. It doesn't have a way of doing that kind of promotion, all right?

So in Julia, so now you can have a method for adding a float 32 to a float 32, but also a method for adding a-- I don't know-- let's see, adding a complex number to a real number, for example. You want to specialize-- or a real number to a complex number. You want to specialize things. In fact, we can click on the link here and see the code.

So the complex number to a real number in Julia looks like this. It's the most obvious thing. It's implemented in Julia. Plus complex real creates new complex number. But you only have to add the real part. You can leave the imaginary part alone. And this works on any complex type. OK, so there's too many methods for-- OK, I can shrink that. Let's see.

So but there's another type inference thing called-- I'll just mention it briefly. So one of the things you've to do to make this type inference process work is given the types of the arguments you have to figure out the type of the return value, OK? So that means when you assign a function, it has to be what's called type stable. The type of the result should depend on the types of the arguments and not on the values of the arguments because the types are known at compile time. The values are only known at runtime.

And it turns out if you don't have this in mind, in C, you have no choice but to obey this. But in something like Python and dynamic language, like Python and Matlab, if you're not thinking about this, it's really easy to design things so that it doesn't work, so that it's not true. So a classic example is a square root function. All right, so suppose I pass an integer to it, OK? So the square root of-- let's do square root of 5, all right? The result has to be floating point number, right? It's 2.23-- whatever.

So if I do square root of 4, of course, that square root is an integer. But if I return an integer for that type, then it wouldn't be type stable anymore. Than the return value type would depend on the value of the input, whether it's a perfect square or not, all right? So it was returned to floating point value, even if the input is an integer. Yes?

AUDIENCE: If you have enough methods to find for a bunch of different types of lookup function? Can that become really slow?

STEVEN JOHNSON: Well, so the lookup comes-- it comes at compile time. So it's really kind of irrelevant. At least if type inference succeeds, if type inference fails, then it's runtime, it's slower. But it's not like-- it's like a tree surge. So it's not-- it's not as slow as you might think. But most of the time you don't worry about that because if you care about performance you want to arrange your code so that type inference succeeds.

So you prototype maybe to-- this is something that you do in performance optimization. Like when you're prototyping, you don't care about types, you say x equals 3. And the next line you say x equals an array-- whatever. But when you're optimizing your code, then, OK, you tweak it a little bit to make sure that things don't change types willy nilly and that the types of function depend on the types of the arguments not on the values.

So, as I mentioned, square root is what really confuses people at first, is if you take square root of minus 1, you might think you should get a complex value. And instead, it gives you an

error, right? And basically, what are the choices here? It could give you an error. It could give you a complex value. But if it gave you a complex value, then the return type of square root would depend upon the value of the input, not just the type.

So Matlab, for example, if you take square root of minus 1, it will happily give you a complex number. But as a result, if you have Matlab, Matlab has a compiler, right? But it has many, many challenges/ but one simple thing to understand is if the Matlab compiler suite sees a square root function, anywhere in your function, even if it knows the inputs that are real, it doesn't know if the outputs are complex or real unless it can prove that the inputs were positive or non-negative, right?

And that means it could then compile two code pass for the output, all right? But then suppose it calls square root again or some other function, right? You quickly get a combinatorial explosion of possible code paths because of possible types. And so at some point, you just give up and put things in a box. But as soon as you put things in a box, and you're looking up types at runtime, you're dead from a performance perspective.

So Python, actually-- s if you want a complex result from square root, you have to give it a complex input. So in Julia, a complex number, the I is actually m. They decide I is too useful for loop variables. So I and J. So m is the complex unit. And if you take square root of a complex input, it gives you a complex output.

So Python actually does the same thing. So if in Python if it takes square root of a negative value, it gives an error unless you give it a complex input. But Python made other mistakes. So, for example, in Python, an integer is guaranteed never to overflow. If you add 11 plus 1 plus 1 over and over again in Python, eventually overflow the size of a 64-bit integer, and Python will just switch under the hood to an arbitrary position in an integer, which seem like a nice idea probably at the time.

And the rest in Python is so slow that the performance cost of this test makes no difference in a typical Python code. But it makes it very difficult to compile. Because that means if you have integer inputs and you see x plus 1 in Python, the compiler cannot just compile that to one instruction, because unless you can somehow prove that x is sufficiently small.

So in Julia, integer arithmetic will overflow. But the default integer arithmetic it's 64 bits. So in practice that never overflows unless you're doing number theory. And you usually know if you're doing number theory, and then use arbitrary precision integers. It was much worse in

the days-- this is something people worried a lot before you were born when there were 16-bit machines, right? And integers, it's really, really easy to overflow 16 bits because the biggest sine value is then 32,767, right?

So it's really easy to overflow it. So you're constantly worrying about overflow. And even 32 bits, the biggest sign value is 2 billion. It's really easy to overflow that, even just counting bytes, right? You can have files that are bigger than 2 gigabytes easily nowadays. So some people worried about this all the time. There were 64-bit integers.

Basically, 64-bit integer will never overflow if it's counting objects that exist in the real universe, like bytes, or loop iterations, or something like that. So you just-- then you just say, OK, it's either 64 bits, or you're doing number theory. You should big ints. So OK. So let me talk about-- the final thing I want to talk about-- let's see, how much [INAUDIBLE] good-- is defining our own types.

So this is the real test of the language, right? So it's easy to make a language where there is a certain built-in set of functions and built-in types, and those things are fast. So, for example, for Python, there actually is a compiler called numba that does exactly what Julia does. It looks at the arguments, type specializes things, and then calls llvm and compiles it to fast code.

But it only works if you're only container type as an NumPy array and you're only scalar type is one of the 12 scalar types that NumPy supports. If you have your own user defined number type or your own user defined container type, then it doesn't work. And user-defined container types, it's probably easy to understand why that's useful. User defined number types are extremely useful as well.

So, for example, there's a package in Julia that provides the number type called dual numbers. And those have the property that if you pass them into the function they compute the function in its derivative. And just a slightly different. It basically carries around function an derivative values and has a slightly different plus and times and so forth that just do the product rule and so forth. And it just propagates derivatives. And then if you have Julia code, like that Vandermonde function, it will just compute its derivative as well.

OK, so I want to be able to find my own type. So a very simple type that I might want to add would be points 2D vectors in two space, right? So, of course, I could have an array of two values. But an array is a really heavyweight object for just two values, right? If I know at

compile time there's two values that I don't need to have a pointer to-- I can actually store them in registers. I I can unroll the loop over these and everything should be faster.

You can get an order of magnitude and speed by specializing on the number of elements for small arrays compared to just a general ray data structure. So let's make a point, OK? So this is-- and I'm going to go through several iterations, starting with a slow iteration. I'm going to define a mutable struct. OK, so this will be a mutable object where I can add a [INAUDIBLE] point that has two values x and y. It can be of any type.

I'll define a plus function that can add them. And it does the most obvious thing. It adds the x components, adds the y components. I'll define a 0 function that's the additive identity that just returns the point 0, 0. And then I can construct an object Point34. I can say 034 plus 0.56. It works. It can hold actually-- right now it's very generic, and probably too generic.

So they act like the real part can be a floating point number and the imaginary. The x can be a floating point number here and the y is a complex number of two integers, or even I can make a string and an array. It doesn't even make sense. So I probably should have restricted the types of x and y a little bit just to prevent the user from putting in something that makes no sense at all, OK?

So these things, they can be anything. So this type is not ideal in several ways. So let's think about how this has to be stored in memory. So this is a 0.1. 0.11, 3.7, right? So in memory it's-- there is an x and there is a y. But x and y can be of any type. So that means they have to be pointers to boxes. There's pointer to int 1 and there's a pointer to a float 64, in this case, 3.7.

So oh, this already, we know this is not going to be good news for performance. And it's mutable. So that mutable struct means if I take p equals a point, I can then say p dot x equals 7. I can change the value, which seems like a harmless thing to do, but actually is a big problem because, for example, if I make an array of a three piece, and then I say p dot y equals 8, and I look at that array, it has to change the y component, OK?

So if I have a p, or in general, if I have a p that's looking at that object, if this is an object you can mutate, it means that if I have another element, a q, it's also pointing the same object. And I say p, p dot x equals 4, then q dot x had better also before at that point. So to have mutable semantics, to have the semantics of something you can change, and other references can see that change, that means that this object has to be stored in memory on the heap as a pointer to two objects so that you have other pointer to the same object, and I mutate it, and the other

references see it.

It can't just be stuck in a register or something like that. It has to be something that other references can see. So this is bad. So if I have, so I can call 0.1 dot aa calls the constructor element-wise. A is this array of 10 to the 7 random numbers. I was benchmarking them before. That was taking 10 milliseconds, OK?

And I can sum it. I can call the built-in some function on this. I can even call my sum function on this because it supports a 0 function. And so it supports a plus. So here, I have an array. If i just go back up, I have an array here of 10 to the 7 values of type 0.1. So the type of 0.1 is attached to the array.

So the array and memory, so I have an array of 0.1, the one here means it's a one-dimensional array. There's also 2D, 3D, and so forth. That looks like a 0.1 value, a 0.1 value, a 0.1 value. But each one of those now-- sorry-- has to be a pointer to an x and a y, which themselves are pointers to boxes. All right, so summing is going to be really slow because there's a lot of pointer chasing. It has to run time, check what's the type of x, what's the type of y.

And, in fact, it was. It took instead of 10 milliseconds, it took 500 or 600 milliseconds. So to do better, we need to do two things. So, first of all, x and y, we have to be able to see what type they are, OK? It can't be just any arbitrary old thing that has to be a pointer to a box, OK? And the point object has to be mutable. It has to be something where if I have p equals something, q equals something, I can't change p and expect q to see it. Otherwise, if it's mutable, those semantics have to be implemented as some pointer to an object someplace. Then you're dead.

So I can just say struct. So struct now is not mutable. It doesn't have the mutable keyword. And I can give the arguments types. I can say they're both flow 64. And x and y are both the same type. They're both 64. But floating point numbers, I'll define plus the same way, 0 the same way, and now I can add them and so forth.

But now if I make an array of these things, and if I say p dot x equals 6, it will give an error. It says you can't actually mutate. Don't even try to mutate it because we can't support those semantics on this type. But that means so that type is actually-- if you look at look at that in memory, what the compiler is allowed to do and what it does do for this is if you have an array of points 0.21, then it looks like just the x value, the y value, The value, the y value, and so

forth. But each of these are exactly one 8 byte flow 64.

And all the types are known at compile time. And so if I sum them, it should take about 0.20 milliseconds, right? Because summing real numbers was 10 milliseconds. And this is twice as many because you have to sum the x's, sum the y's. And let's benchmark it. And let's see.

Oh, actually, some of the real numbers took 5 milliseconds. So something should take about 10. Let's see if that's still true. Yeah, it took about 10. So actually, the compiler is smart enough. So, first of all, it stores this in line as one big block, consecutive block of memory.

And then when you sum them, remember our sum function. Well, this is the built-in sum. But our sum function will work in the same way. The compiler-- llvm will be smart enough to say, oh, I can load this into a register, load y into a register, call, have a tight loop where I basically call one instruction to sum the x's, one instruction to sum the y's, and then repeat. And so it's about as good as you could get.

But you paid a big price. We've lost all generality, right? These can only be two 64-bit floating point numbers. I can't have two single-precision numbers or-- this is like a struct of two doubles in C. So if I have to do this to get performance in Julia, then it would suck. Basically, I'm basically writing C code in a slightly higher level syntax. I'm losing that benefit of using a high level language.

So the way you get around this is you define-- what you want is to define something like this 0.2 type. But you don't want to define just one type. You want to define a whole family of types. You want to define this for two float 64s, two float 32s. In fact, you want to define an infinite family of types, at two things of any type you want as long as they're two real numbers, two real types.

And so the way you do that in Julia is a parametrized type. This is called parametric polymorphism. It's similar to what you see in C++ templates. So now I have a struct. It's not mutable-- Point3. But the curly braces t. It says it's parametrized by type t. So x and y-- I've restricted it slightly here. I've said x and y had to be the same type. I didn't have to do that, but I could have had two parameters, one for the type of x, one to the type of y.

But most the time you'd be doing something like this, you'd want them both to be the same type. But they could be both 64s, both float 32s, both integers, whatever. So t is any type that less than colon means is a subtype of. So t is any subtype of real. It could be float 64. It could

be int 64. It could be int 8. It could be big float. It could be a user defined type. It doesn't care.

So this is really not-- it's a Point3 here. It is a whole hierarchy. So I'm not defining one type. I'm defining a whole set of types. So Point3 is a set of types. There's a point Point3 int 64. There is a Point3 float 32, a float 64 and so on. Infinitely, many types, as many as you want, and basically, it'll create more types on the fly just by instantiating.

So, for example, otherwise it looks the same. The plus function is basically the same. I add the x components, the y components. The 0 function is the same. Except now I make sure there's zeros of type t, whatever that type is. And now if I say Point34, now I'm instantiating a particular instance of this. And now that particular instance of Point3 we'll have-- this is an abstract type. We'll have one of these concrete types. And the concrete type it has in this case is a Point3 of two int 64s, two 64-bit integers, OK?

And I can add them. And actually, adding mixed types will already work because the plus, the addition function here, it works for any 2.3s. I didn't say there had to Point3s of the same type. Any two of these, they don't have to be two of these. They could be one of these and one of these. And then it determines the type of the result by the type of the [INAUDIBLE] it does type inference.

So if you have a point Point3 of two int 64s and Point3 of two float 32s, it says, oh, p dot x is an int 64. Q dot x is a full 64. Oh, which plus function do I call? There is a plus function from that mixing. And it promotes the result of flow 64. So that means that that sum is flow 64. The other sum is flow 64. Oh, then I'm creating a Point3 of flow 64s. So this kind of mixed promotion is done automatically. You can actually define your own promotion rules in Julia as well. And I can make an array.

And so now if I have an array of Point3 float 64, so this type is attached to the whole array. And this is not an arbitrary Point3. It's a Point3 of two float 64s. So it gets stored again as just 10 to the 7 elements of xy, xy where each one is 8 bytes 8 byes, 8 bytes, one after the other. The compiler knows all the types. And when you submit, it knows everything at compile time. And it will sum to these things. But I loaded this into a register, load this into a register called one instruction-- add them.

And so the sum function should be fast. So we can call the built-in sum function. We can call our own sum function. Our own some function, I didn't put SIMD here, so it's going to be twice as slow. But Yeah. Yeah?

AUDIENCE: Will this work with SIMD?

STEVEN JOHNSON: Yeah. Yeah. In fact, if you look, the built-in sum function, the built-in sum function is implemented in Julia. It just hasn't [INAUDIBLE] SIMD on the sum. So yeah. llvm is smart enough that if you give it a struct of two values and load them-- and if you tell it that you're adding these two values to these two values these two values to these two values, it will actually use SIMD instructions, I think. Oh, maybe not.

No, wait. Did my sum use SIMD? I'm confused. I thought it did.

AUDIENCE: [INAUDIBLE] removed it.

STEVEN JOHNSON: I thought I removed it. Yeah, so maybe it's not smart enough to use SIMD. I've seen in some cases where it's smart enough to use-- huh, yeah. OK, so they're the same speed. OK, no. I take it back. So maybe llvm is not smart enough in this case to use SIMD automatically. We could try putting the SIMD annotation there and try it again. But I thought it was, but maybe not.

Let's see. Let's put SIMD. So redefine that. And then just rerun this. So it'll notice that I've changed the definition. It'll recompile it. But the B time, since it times at multiple times, the first time it calls it, it's slow because it's compiling it. But it takes the minimum over several times. So let's see.

Yeah, this is the problem in general with vectorizing compilers if they're not that smart if you're using anything other than just an array of an elementary data type. Yeah, no. It didn't make any difference. So I took it back. So for more complicated data structures, you often have to use SIMD structure explicitly. And there is a way to do that in Julia. And there is a higher level library on top of that. You can basically credit a tuple and then add things and it will do SIMD acceleration. But yeah.

So anyway, so that's the upside here. There's a whole bunch of-- like the story of why Julia can be compiled with fast code, it's a combination of lots of little things. But there are a few big things. One is that its specialized thing is compile times. But, of course, you could do that in any language. So that relies on designing the language so that you can do type inference.

It relies on having these kind of parametrized types and giving you a way to talk about types and attach types to other types. So the array you notice probably-- let's see-- and now that

you understand what these little braces mean, you can see that the array is defined in Julia as another parametrized type. It's parametrized by the type of the element and also by the dimensionality. So it uses the same mechanism to attach types to an array. And you can have your own-- the array type in Julia is implemented mostly in Julia. And there are other packages that implement their own types of arrays that have the same performance.

One of the goals of Julia is to build in as little as possible so that there's not some set of privileged types that the compiler knows about and everything else is second class. It's like user code is just as good as the built-in code. And, in fact, the built-in code is mostly just implemented in Julia. There's a small core that's implemented in C for bootstrapping, basically. Yeah. So having parametrized types, having another technicalities, having all concrete types are final in Julia.

A concrete type is something you can actually store in memory. So Point3864 is something you can actually have, right? An object of two integers is that type. So it is concrete, as opposed to this thing. This is an abstract type. You can't actually have one of these. You can only have one of the instances of the concrete types. So but there are no-- this is final.

It's not possible to have a subtype of this because if you could, then you'd be dead because this is an array of these things. If the compiler has to know it's actually these things and not some subtype of this, all right, whereas in other languages, like Python, you can have subtypes of concrete types. And so then even if you said this is an array of a particular Python type, it wouldn't really know it's that type, or it might be some subtype of that. That's one of the reasons why you can't implement NumPy in Python. This is-- there's no way to say this is really that type and nothing else in the language level. Yeah?

AUDIENCE: Will this compilation in Julia work?

STEVEN JOHNSON: Oh, yeah. So and it's calling llvm. So basically, the stage is you call-- so there's a few passes. OK, so and one of the fun things is you can actually inspect all the passes and almost intercept all of them practically. So, of course, typing code like this, first, it gets parsed, OK? And you can macro those things [INAUDIBLE] actually are functions that are called right after parsing. They can just take the parse code and rewrite it arbitrarily.

So they can extend the language that way. But then it parsed, maybe rewritten by a macro. And then you get an abstract syntax tree. And then when you call it, let's say f of 3, then says,

oh, x is an integer. Int 64, it runs a type inference pass. It tries to figure out what's the type of everything, which version of plus to call and so forth.

Then it decides whether to inline some things. And then once it's done all that, it spits out llvm byte code, then calls llvm, and compiles it to machine code. And then it caches that some place for the next time you call, you call f of 4, f with another integer. It doesn't repeat the same processes. Notice it's cached. So that's-- so yeah. At the lowest level, it's just the llvm.

So then there's tons of things I haven't showed you. So I haven't showed you-- I mentioned metaprogramming. So it has this macro facility. So you can basically write syntax that rewrites other syntax, which is really cool for code generation. You can also intercept it after the type inference phase. You can write something called the generated function that basically takes-- because at parse time, it knows how things are spelled. And you can rewrite how they're spelled. But it doesn't know what anything actually means. It does knows x is a symbol. It doesn't know x as an integer-- whatever. It just knows the spelling.

So when you actually compile f of x, at that point, it knows x is an integer. And so you can write something called a generator or a stage function that basically runs at that time and says, oh, you told me x is an integer. Now I'll rewrite the code based on that. And so this is really useful for-- there's some cool facilities for multidimensional arrays. Because the dimensionality of the array is actually part of the type.

So you can say, oh, this is a three-dimensional array. I'll write three loops. Oh, you have a four-dimensional array. I'll write four loops. And it can rewrite the code depending on the dimensionality with code generation. So you can have code that basically generates any number of nested loops depending on the types of the arguments. And all the generation is done in compiled time after type inference. So it knows the dimensionality of the array.

And yeah, so there's lots of fun things like that. Of course, it has parallel facilities. They're not quite as advanced as Cilk at this point, but that's the direction there they're heading. There's no global interpreter lock like in Python. There's no interpreter. So there's a threading facility. And there's a pool of workers. And you can thread a loop. And the garbage collection is threading aware. So that's safe.

And they're gradually having more and more powerful run times, hopefully, eventually hooking into some of Professor Leiserson's advanced threading compiler, taper compiler, or whatever it is. And there's also-- most of what I do in my research is more coarse grained distributed

memory parallelism, so running on supercomputers and stuff like that.

And there's MPI. There is a remote procedure call library. There's different flavors of that. But yeah. So any other questions? Yeah?

AUDIENCE: How do you implement the big number type?

STEVEN JOHNSON: The big num type in Julia is actually calling GIMP. So that's one of those things. Let me just-- let me make a new notebook. So if I say I know big int 3, 3,000, and then I'd say that to the, say, factorial. I think there's a built-in factorial of that. All right, so this is called the big num type, right? It's something where the number of digits changes at run time.

So, of course, these are orders of magnitude slower than hardware things. Basically, it has to implement it as a loop of digits in some base. And when you add or multiply, you have to loop over those at runtime. These big num libraries, they are quite large and heavily optimized. And so nobody has reimplemented one in Julia. They're just calling out to a C library called the GNU multi-precision library.

And for floating point values, there is something called big float. So big float of pi is that I can actually-- let's set precision. Big float to 1000. That's 1,000 binary digits. And then say big float of pi. And [INAUDIBLE] more. By the way, you might have-- so I can have a variable alpha-- oops-- alpha hat sub 2 equals 17. That's allowed.

All that's happening here is that Julia allows almost arbitrary unicode things for identifiers. So I can have-- make it bigger so we can have an identifier Koala, right? So there's two issues here. So one is just you have a language that allows those things as identifiers. So Python 3 also allows Unicode identifiers, although I think Julia out of all the existing-- the common languages-- it's probably the widest unicode support.

Most languages only allow a very narrow range of unicode characters for identifiers. So Python would allow the koala, but Python 3 would not allow with alpha hat sub 2 because the numeric subscript unicode characters it doesn't allow. The other thing is how do you type these things. And that's more of an editor thing.

And so in Julia, we implemented initially in the repl and in Jupiter. And now all the editors support, you can just to tab completion of latex. So I can type in gamma, tab, and the tab completes to the unicode character. I can say dot. And it puts a dot over it and backslash

superscript 4. And it puts a 4. And that's allowed. So it's quite nice.

So when I'm typing emails, and I put equations in emails, I go to the Julia rappel and tab complete all my LaTeX characters so that I can put equations in emails because It's the easiest way to type these Unicode math characters. But yeah. So IPython borrowed this. So now do the same thing in the IPython notebooks as well.

So it's really fun. Because if you read old math codes, especially old Fortran codes or things, you see lots of variables that are named alpha hat or something like that, alpha hat underscore 3. It's so much nicer to have a variable that's actually the alpha hat sub 3. So that's cute.

CHARLES E. LEISERSON: Steve, thanks very much. Thanks. This was great.

[APPLAUSE]

We are, as Steve mentioned, looking actually at a project to merge the Julia technology with the Cilk technology. And so we're right now in the process of putting together the grant proposal. And if that gets funded, there may be some UROPS.