6.001 Structure and Interpretation of Computer Programs, Spring 2005
Transcript – 6B: Streams, Part 2

PROFESSOR: OK, well, we've been looking at streams, this signal processing way of putting systems together. And remember, the key idea is that we decouple the apparent order of events in our programs from the actual order of events in the computer. And that means that we can start dealing with very long streams and only having to generate the elements on demand. That sort of on-demand computation is built into the stream's data structure. So if we have a very long stream, we only compute what we need. The things only get computed when we actually ask for them.

Well, what are examples? Are they actually asking for them? For instance, we might ask for the n-th element of a stream. Here's a procedure that computes the n-th element of a stream. An integer n, the n-th element of some stream s, and we just recursively walk down the stream. And the end of 0, we compute the head. Otherwise, it's the n-th the minus 1 element of the tail of the stream. Those two are just like for Lisp, but the difference is those elements aren't going to get computed until we walk down, taking successive n-ths. So that's one way that the stream elements might get forced.

And another way, here's a little procedure that prints a stream. We say print a stream, so to print a stream s. Well, what do we do? We print the head of the stream, and that will cause the head to be computed. And then we recursively print stream the tail of the stream. And if we're already done, maybe we have to return something about the message done. OK, and then so if you make a stream, you could say here's the stream, this very long stream. And then you say print the stream, and the elements of the stream will get computed successively as that print calls them. They won't get all computed initially. So in this way, we can deal with some very long streams. Well, how long can a stream be? Well, it can be infinitely long.

Let's look at an example here on the computer. I could walk up to this computer, and I could say-- how about we'll define the stream of integers starting with some number N, the stream of positive integers starting with some number n. And that's cons-stream of n onto the integers from one more. So there are the integers.

Then I could say let's get all the integers. define the stream of integers to be the integers starting with 1. And now if I say something like what's the what's the 20th integer. So it's 21 because we start counting at 0.

Or I can do more complicated things. Let me to define a little predicate here. How about define no-seven. It's going to test an integer, and it's going to say it's not. I take the remainder of x by 7, I don't get 0. And then I could say define the integers with no sevens to be, take all the integers and filter them to have no sevens.

So now I've got the stream of all the integers that are not divisible by seven. So if I say what's the 100th integer and the list not divisible by seven, I get 117. Or if I'd like to say well, gee, what are all of them? So I could say print stream all these integers with no seven, it goes off printing. You may have to wait a very long time to see them all.

Well, you can start asking, gee, is it really true that this data structure with the integers is really all the integers? And let me draw a picture of that program I just wrote. Here's the definition of the integers again that I just typed in, Right it's a cons of the first integer under the integer starting with the rest. Now, we can make a picture of that and see what it looks like.

Conceptually, what I have is a box that's the integer starting with n. It takes in some number n, and it's going to return a stream of-- this infinite stream of all integers starting with n. And what do I do? Well, this is an integers from box. What's it got in it? Well, it takes in this n, and it increments it. And then it puts the result into recursively another integer's from box. It takes the result of that and the original n and puts those together with a cons and forms a stream. So that's a picture of that program I wrote.

Let's see. These kind of diagrams we first saw drawn by Peter Henderson, the same guy who did the Escher language. We call them Henderson diagrams. And the convention here is that you put these things together. And the solid lines are things coming out are streams, and dotted lines are initial values going in. So this one has the shape of-- it takes in some integer, some initial value, and outputs a stream.

Again, you can ask. Is that data structure integers really all the integers? Or is it is something that's cleverly arranged so that whenever you look for an integer you find it there? That's sort of a philosophical question, right? If something is there whenever you look, is it really there or not? It's sort of the same sense in which the money in your savings account is in the bank.

Well, let me do another example. Gee, we started the course with an algorithm from Alexandria, which was Heron of Alexandria's algorithm for computing the square root. Let's take a look at another Alexandrian algorithm. This one is Eratosthenes method for computing all of the primes. It is called the Sieve of Eratosthenes. And what you do is you start out, and you list all the integers, say, starting with 2. And then you take the first integer, and you say, oh, that's prime. And then you go look at the rest, and you cross out all the things divisible by 2. So I cross out this and this and this. This takes a long time because I have to do it for all of the integers. So I go through the entire list of integers, crossing the ones divisible by 2.

And now when I finish with all of the integers, I go back and look and say what am I left with? Well, the first thing that starts there is 3. So 3 is a prime. And now I go back through what I'm left with, and I cross out all the things divisible by 3. So let's see, 9 and 15 and 21 and 27 and 33 and so on. I won't finish. Then I see what I'm left with. And the next one I have is 5. Now I can through the rest, and I find the first one that's divisible by 5. I cross out from the remainder all the ones that are divisible by 5. And I do that, and then I go through and find 7. Go through all the rest, cross out things divisible 7, and I keep doing that forever. And when I'm done, what I'm left with is a list of all the primes. So that's the Sieve of Eratosthenes.

Let's look at it as a computer program. It's a procedure called sieve. Now, I just write what I did. I'll say to sieve some stream s. I'm going to build a stream whose first element is the head of this. Remember, I always found the first thing I was left with, and the rest of it is the result of taking the tail of this, filtering it to throw away all the things that are divisible by the head of this, and now sieving the result. That's just what I did.

And now to get the infinite stream of times, we just sieve all the integers starting from 2. Let's try that. We can actually do it. I typed in the definition of sieve before, I hope, so I can say something like define the primes to be the result of sieving the integers starting with 2. So now I've got this list of primes. That's all of the primes, right? So, if for example, what's the 20th prime in that list? 73. See, and that little pause, it was only at the point when I started asking for the 20th prime is that it started computing. Or I can say here let's look at all of the primes. And there it goes computing all of the primes. Of course, it will take a while again if I want to look at all of them, so let's stop it.

Let me draw you a picture of that. Well, I've got a picture of that. What's that program really look like? Again, some practice with these diagrams, I have a sieve box. How does sieve work? It takes in a stream. It splits off the head from the tail. And the first thing that's going to come out of the sieve is the head of the original stream. Then it also takes the head and uses that. It takes the stream. It filters the tail and uses the head to filter for nondivisibility. It takes the result of nondivisibility and puts it through another sieve box and puts the result together. So you can think of this sieve a filter, but notice that it's an infinitely recursive filter. Because inside the sieve box is another sieve box, and inside that is another sieve box and another sieve box.

So you see we start getting some very powerful things. We're starting to mix this signal processing view of the world with things like recursion that come from computation. And there are all sorts of interesting things you can do that are like this. All right, any questions? OK, let's take a break.

Well, we've been looking at a couple of examples of stream programming. All the stream procedures that we've looked at so far have the same kind of character. We've been writing these recursive procedures that kind of generate these stream elements one at a time and put them together in cons-streams. So we've been thinking a lot about generators. There's another way to think about stream processing, and that's to focus not on programs that sort of process these elements as you walk down the stream, but on things that kind of process the streams all at once.

To show you what I mean, let me start by defining two procedures that will come in handy. The first one's called add streams. Add streams takes two streams: s1 and s2. and. It's going to produce a stream whose elements are the are the corresponding sums. We just sort of add them element-wise. If either stream is empty, we just return the other one. Otherwise, we're going to make a new stream whose head is the sum of the two heads and whose tail is the result of recursively adding the tails. So that will produce the element-wise sum of two streams.

And then another useful thing to have around is scale stream. Scale stream takes some constant number in a stream s and is going to produce the stream of elements of s multiplied by this constant. And that's easy, that's just a map of the function of an element that multiplies it by the constant, and we map that down the stream.

So given those two, let me show you what I mean by programs that operate on streams all at once. Let's look at this. Suppose I write this. I say define-- I'll call it ones-- to be cons-stream of 1 onto ones. What's that? That's going to be an infinite stream of ones because the first thing is 1. And the tail of it is a thing whose first thing is 1 and whose tail is a thing whose first thing is 1 and so on and so on and so on. So that's an infinite stream of ones.

And now using that, let me give you another definition of the integers. We can define the integers to be-- well, the first integer we'll take to be 1, this cons-stream of 1 onto the element-wise sum onto add streams of the integers to ones. The integers are a thing whose first element is 1, and the rest of them you get by taking those integers and incrementing each one by one. So the second element of the integers is the first element of the integers incremented by one. And the rest of that is the next one, and the third element of that is the same as the first element of the tail of the integers incremented by one, which is the same as the first element of the original integers incremented by one and incremented by one again and so on.

That looks pretty suspicious. See, notice that it works because of delay. See, this looks like-- let's take a look at ones. This looks like it couldn't even be processed because it's suddenly saying in order to know what ones is, I say it's cons-stream of something onto ones. The reason that works is because of that very sneaky hidden delay in there. Because what this really is, remember, cons-stream is just an abbreviation. This really is cons of 1 onto delay of ones.

So how does that work? You say I'm going to define ones. First I see what ones is supposed to be defined as. Well, ones is supposed to be defined as a cons whose first part is 1 and whose second part is, well, it's a promise to compute something that I don't worry about yet. So it doesn't bother me that at the point I do this definition, ones isn't defined. Having run the definition now, ones is defined. So that when I go and look at the tail of it, it's defined. It's very sneaky. And an integer is the same way. I can refer to integers here because hidden way down-- because of this cons-stream. It's the cons-stream of 1 onto something that I don't worry that yet. So I don't look at it, and I don't notice that integers isn't defined at the point where I try and run the definition.

OK, let me draw a picture of that integers thing because it still maybe seems a little bit shaky. What do I do? I've got the stream of ones, and that sort of comes in and goes into an adder that's going to be this add streams thing. And that goes in-- that's going to put out the integers. And the other thing that goes into the adder here is the integer, so there's a little feedback loop. And all I need to start it off is someplace I've got a stick that initial 1.

In a real signal processing thing, this might be a delay element with that was initialized to 1. But there's a picture of that ones program. And in fact, that looks a lot like-- if you've seen real signal block diagram things, that looks a lot like accumulators, finite state accumulators. And in fact, we can modify this a little bit to change this into something that integrates a stream or a finite state accumulator, however you like to think about it.

So instead of the ones coming in and getting out the integers, what we'll do is say there's a stream s coming in, and we're going to get out the integral of this, successive values of that, and it looks almost the same. The only thing we're going to do is when s comes in here, before we just add it in we're going to multiply it by some number dt. And now what we have here, this is exactly the same thing. We have a box, which is an integrator. And it takes in a stream s, and instead of 1 here, we can put the additional value for the integral.

And that one looks very much like a signal processing block diagram program. In fact, here's the procedure that looks exactly like that. Find the integral of a stream. So an integral's going to take a stream and produce a new stream, and it takes in an initial value and some time constant. And what do we do? Well, we internally define this thing int, and we make this internal name so we can feed it back, loop it around itself. And int is defined to be something that starts out at the initial value, and the rest of it is gotten by adding together. We take our input stream, scale it by dt, and add that to int. And now we'll return from all that the value of integral is this thing int. And we use this internal definition syntax so we could write a little internal definition that refers to itself.

Well, there are all sorts of things we can do. Let's try this one. how about the Fibonacci numbers. You can say define fibs. Well, what are the Fibonacci numbers? They're something that starts out with 0, and the next one is 1. And the rest of the Fibonacci numbers are gotten by adding the Fibonacci numbers to their own tail. There's a definition of the Fibonacci numbers.

How does that work? Well, we start off, and someone says compute for us the Fibonacci numbers, and we're going to tell you it starts out with 0 and 1. And everything after the 0 and 1 is gotten by summing two streams. One is the fibs themselves, and the other one is the tail of the fibs.

So if I know that these start out with 0 and 1, I know that the fibs now start out with 0 and 1, and the tail of the fibs start out with 1. So as soon as I know that, I know that the next one here is 0 plus 1 is 1, and that tells me that the next one here is 1 and the next one here is 1. And as soon as I know that, I know that the next one is 2. So the next one here is 2 and the next one here is 2. And this is 3. This one goes to 3, and this is 5. So it's a perfectly sensible definition. It's a one-line definition. And again, I could walk over to the computer and type that in, exactly that, and then say print stream the Fibonacci numbers, and they all come flying out.

See, this is a lot like learning about recursion again. Instead of thinking that recursive procedures, we have recursively defined data objects. But that shouldn't surprise you at all, because by now, you should be coming to really believe that there's no difference really between procedures and data. In fact, in some sense, the underlying streams are procedures sitting there, although we don't think of them that way. So the fact that we have recursive procedures, well, then it should be natural that we have recursive data, too.

OK, well, this is all pretty neat. Unfortunately, there are problems that streams aren't going to solve. Let me show you one of them. See, in the same way, let's imagine that we're building an analog computer to solve some differential equation like, say, we want to solve the equation y prime dy dt is y squared, and I'm going to give you some initial value. I'll tell you y of 0 equals 1. Let's say dt is equal to something.

Now, in the old days, people built analog computers to solve these kinds of things. And the way you do that is really simple. You get yourself an integrator, like that one, an integrator box. And we put in the initial value y of 0 is 1. And now if we feed something in and get something out, we'll say, gee, what we're getting out is the answer. And what we're going to feed in is the derivative, and the derivative is supposed to be the square of the answer. So if we take these values and map using square, and if I feed this around, that's how I build a block diagram for an analog computer that solves this differential equation.

Now, what we'd like to do is write a stream program that looks exactly like that. And what do I mean exactly like that? Well, I'd say define y to be the integral of dy starting at 1 with 0.001 as a time step. And I'd like to say that says this. And then I'd like to say, well, dy is gotten by mapping the square along y. So define dy to be map square along y. So there's a stream description of this analog computer, and unfortunately, it doesn't work.

And you can see why it doesn't work because when I come in and say define y to be the integral of dy, it says, oh, the integral of y-- huh? Oh, that's undefined. So I can't write this definition before I've written this one. On the other hand, if I try and write this one first, it says, oh, I define y to be the map of square along y? Oh, that's not defined yet. So I can't write this one first, and I can't write that one first. So I can't quite play this game.

Well, is there a way out? See, we can do that with ones. See, over here, we did this thing ones, and we were able to define ones in terms of ones because of this delay that was built inside because cons-stream had a delay. Now, why's it sensible? Why's it sensible for cons-stream to be built with this delay? The reason is that cons-stream can do a useful thing without looking at its tail. See, if I say this is cons-stream of 1 onto something without knowing anything about something, I know that the stream starts off with 1. That's why it was sensible to build something like cons-stream. So we put a delay in there, and that allows us to have this sort of self-referential definition.

Well, integral is a little bit the same way. See, notice for an integral, I can-- let's go back and look at integral for a second. See, notice integral, it makes sense to say what's the first thing in the integral without knowing the stream that you're integrating. Because the first thing in the integral is always going to be the initial value that you're handed. So integral could be a procedure like cons-stream. You could define it, and then even before it knows what it's supposed to be integrating, it knows enough to say what its initial value is.

So we can make a smarter integral, which is aha, you're going to give me a stream to integrate and an initial value, but I really don't have to look at that stream that I'm supposed to integrate until you ask me to work down the stream. In other words, integral can be like cons-stream, and you can expect that there's going to be a delay around its integrand. And we can write that. Here's a procedure that does that.

Another version of integral, and this is almost like the previous one, except the stream it's going to get in is going to expect to be a delayed object. And how does this integral work? Well, the little thing it's going to define inside of itself says on the cons-stream, the initial value is the initial value, but only inside of that cons-stream, and remember, there's going to be a hidden delay inside here. Only inside of that cons-stream will I start looking at what the actual delayed object is.

So my answer is the first thing's the initial value. If anybody now asks me for my tail, at that point, I'm going to force that delayed object-- and I'll call that s-- and I do the add streams. So this is an integral which is sort of like cons-stream. It's not going to actually try and see what you handed it as the thing to integrate until you look past the first element. And if we do that and we can make this work, all we have to do here is say define y to the integral of delay of y, of delay of dy. So y is going to be the integral of delay of dy starting at 1, and now this will work. Because I type in the definition of y, and that says, oh, I'm supposed to use the integral of something I don't care about right now because it's a delay.

And these things, now you define dy. Now, y is defined. So when I define dy, it can see that definition for y. Everything is now started up. Both streams have their first element. And then when I start mapping down, looking at successive elements, both y and dy are defined. So there's a little game you can play that goes a little bit beyond just using the delay that's hidden inside streams. Questions? OK, let's take a break.

Well, just before the break, I'm not sure if you noticed it, but something nasty started to happen. We've been going along with the streams and divorcing time in the programs from time in the computers, and all that divorcing got hidden inside the streams. And then at the very end, we saw that sometimes in order to really take advantage of this method, you have to pull out other delays. You have to write some explicit delays that are not hidden inside that cons-stream.

And I did a very simple example with differential equations, but if you have some very complicated system with all kinds of self-loops, it becomes very, very difficult to see where you need those delays. And if you leave them out by mistake, it becomes very, very difficult to see why the thing maybe isn't working. So that's kind of mess, that by getting this power and allowing us to use delay, we end up with some very complicated programming sometimes, because it can't all be hidden inside the streams.

Well, is there a way out of that? Yeah, there is a way out of that. We could change the language so that all procedures acted like cons-stream, so that every procedure automatically has an implicit delay around its arguments. And what would that mean? That would mean when you call a procedure, the arguments wouldn't get evaluated. Instead, they'd only be evaluated when you need them, so they might be passed off to some other procedure, which wouldn't evaluate them either.

So all these procedures would be passing promises around. And then finally maybe when you finally got down to having to look at the value of something that was handed to a primitive operator would you actually start calling in all those promises. If we did that, since everything would have a uniform delay, then you wouldn't have to write any explicit delays, because it would be automatically built into the way the language works.

Or another way to say that, technically what I'm describing is what's called-- if we did that, our language would be so-called normal-order evaluation language versus what we've actually been working with, which is called applicative order-- versus applicative-order evaluation.

And remember the substitution model for applicative order. It says when you go and evaluate a combination, you find the values of all the pieces. You evaluate the arguments and then you substitute them in the body of the procedure. Normal order says no, don't do that. What you do is effectively substitute in the body of the procedure, but instead of evaluating the arguments, you just put a promise to compute them there. Or another way to say that is you take the expressions for the arguments, if you like, and substitute them in the body of the procedure and go on, and never really simplify anything until you get down to a primitive operator. So that would be a normal-order language.

Well, why don't we do that? Because if we did, we'd get all the advantages of delayed evaluation with none of the mess. In fact, if we did that and cons was just a delayed procedure, that would make cons the same as cons-stream. We wouldn't need streams of all because lists would automatically be streams. That's how lists would behave, and data structures would behave that way. Everything would behave that way, right? You'd never really do any computation until you actually needed the answer. You wouldn't have to worry about all these explicit annoying delays. Well, why don't we do that?

First of all, I should say people do do that. There's some very beautiful languages. One of the very nicest is a language called Miranda, which is developed by David Turner at the University of Kent. And that's how this language works. It's a normal-order language and its data structures, which look like lists, are actually streams. And you write ordinary procedures in Miranda, and they do these prime things and eight queens things, just without anything special. It's all built in there. But there's a price.

Remember how we got here. We're decoupling time in the programs from time in the machines. And if we put delay, that sort of decouples it everywhere, not just in streams. Remember what we're trying to do. We're trying to think about programming as a way to specify processes. And if we give up too much time, our language becomes more elegant, but it becomes a little bit less expressive. There are certain distinctions that we can't draw.

One of them, for instance, is iteration. Remember this old procedure, iterative factorial, that we looked at quite a long time ago. Iterative factorial had a thing, and it said there was an internal procedure, and there was a state which was a product and a counter, and we iterate that going around the loop. And we said that was an iterative procedure because it didn't build up state. And the reason it didn't build up state is because this iter that's called is just passing these things around to itself. Or in the substitution model, you could see in the substitution model that Jerry did, that in an iterative procedure, that state doesn't have to grow. And in fact, we said it doesn't, so this is an iteration.

But now think about this exact same text if we had a normal-order language. What would happen is this would no longer be an iterative procedure? And if you really think about the details of the substitution model, which I'm not going to do here, this expression would grow. Why would it grow? It's because when iter calls itself, it calls itself with this product. If it's a normal-order language, that multiplication is not going to get done. That's going to say I'm to call myself with a promise to compute this product. And now iter goes around again. And I'm going to call myself with a promise to compute this product where now one of the one factors is a promise. And I call myself again. And if you write out the substitution model for that iterative process, you'll see exactly the same growth in state, all those promises that are getting remembered that have to get called in at the very end.

So one of the disadvantages is that you can't really express iteration. Maybe that's a little theoretical reason why not, but in fact, people who are trying to write real operating systems in these languages are running into exactly these types of problems. Like it's perfectly possible to implement a text editor in languages like these. But after you work a while, you suddenly have 3 megabytes of stuff, which is-- I guess they call them the dragging tail problem of people who are looking at these, of promises that sort of haven't been called in because you couldn't quite express an iteration. And one of the research questions in these kinds of languages are figuring out the right compiler technology to get rid of the so-called dragging tails. It's not simple.

But there's another kind of more striking issue about why you just don't go ahead and make your language normal order. And the reason is that normal-order evaluation and side effects just don't mix. They just don't go together very well. Somehow, you can't-- it's sort of you can't simultaneously go around trying to model objects with local state and change and at the same time do these normal-order tricks of de-coupling time. Let me just show you a really simple example, very, very simple.

Suppose we had a normal-order language. And I'm going to start out in this language. This is now normal order. I'm going to define x to be 0. It's just some variable I'll initialize. And now I'm going to define this little funny function, which is an identity function. And what it does, it keeps track of the last time you called it using x. So the identity of n just returns n, but it sets x to be n. And now I'll define a little increment function, which is a very little, simple scenario.

Now, imagine I'm interacting with this in the normal-order language, and I type the following. I say define y to be increment the identity function of 3, so y is going to be 4. Now, I say what's x? Well, x should have been the value that was remembered last when I called the identity function. So you'd expect to say, well, x is 3 at this point, but it's not. Because when I defined y here, what I really defined y to be increment of a promise to do this thing. So I didn't look at y, so that identity function didn't get run. So if I type in this definition and look at x, I'm going to get 0.

Now, if I go look at y and say what's y, say y is 4, looking at y, that very active looking at y caused the identity function to be run. And now x will get remembered as 3. So here x will be 0. Here, x will be 3. That's a tiny, little, simple scenario, but you can see what kind of a mess that's going to make for debugging interactive programs when you have normal-order evaluation.

It's very confusing. But it's very confusing for a very deep reason, which is that the whole idea of putting in delays is that you throw away time. That's why we can have these infinite processes. Since we've thrown away time, we don't have to wait for them to run, right? We decouple the order of events in the computer from what we write in our programs. But when we talk about state and set and change, that's exactly what we do want control of. So it's almost as if there's this fundamental contradiction in what you want.

And that brings us back to these sort of philosophical mutterings about what is it that you're trying to model and how do you look at the world. Or sometimes this is called the debate over functional programming. A so-called purely functional language is one that just doesn't have any side effects. Since you have no side effects, there's no assignment operator, so there are no terrible consequences of it. You can use a substitution-like thing. Programs really are like mathematics and not like models in the real world, not like objects in the real world.

There are a lot of wonderful things about functional languages. Since there's no time, you never have any synchronization problems. And if you want to put something into a parallel algorithm, you can run the pieces of that parallel processing any way you want. There's just never any synchronization to worry that, and it's a very congenial environment for doing this. The price is you give up assignment. So an advocate of a functional language would say, gee, that's just a tiny price to pay. You probably shouldn't use assignment most of the time anyway. And if you just give up assignment, you can be in this much, much nicer world than this place with objects.

Well, what's the rejoinder to that? Remember how we got into this mess. We started trying to model things that had local state. So remember Jerry's random number generator. There was this random number generator that had some little state in it to compute the next random number and the next random number and the next random number. And we wanted to hide that state away from the Cesaro compute part process, and that's why we needed set. We wanted to package that stated modularly.

Well, a functional programming person would say, well, you're just all wet. I mean, you can write a perfectly good modular program. It's just you're thinking about modularity wrong. You're hung up in this next random number and the next random number and the next random number. Why don't you just say let's write a program. Let's write an enumerator which just generates an infinite stream of random numbers. We can sort of have that stream all at once, and that's going to be our source of random numbers. And then if you like, you can put that through some sort of processor, which is-- I don't know-- a Cesaro test, and that can do what it wants.

And what would come out of there would be a stream of successive approximations to pi. So as we looked further down this stream, we'd tug on this Cesaro thing, and it would pull out more and more random numbers. And the further and further we look down the stream, the better an approximation we'd get to pi. And it would do exactly the same as the other computation, except we're thinking about the modularity different. We're saying imagine we had all those infinite streams of random numbers all at once. You can see the details of this procedure in the book.

Similarly, there are other things that we tend to get locked into on this one and that one and the next one and the next one, which don't have to be that way. Like you might think about like a banking system, which is a very simple idea. Imagine we have a program that sort of represents a bank account. The bank account might have in it-- if we looked at this in a sort of message-passing view of the world, we'd say a bank account is an object that has some local state in there, which is the balance, say.

And a user using this system comes and sends a transaction request. So the user sends a transaction request, like deposit some money, and the bank account maybe-- let's say the bank account always responds with what the current balance is. The user says let's deposits some money, and the bank account sends back a message which is the balance. And the user says deposit some more, and the bank account sends back a message. And just like the random number generator, you'd say, gee, we would like to use set. We'd like to have balance be a piece of local state inside this bank account because we want to separate the state of the user from the state of the bank account.

Well, that's the message-processing view. There's a stream view with that thing, which does the same thing without any set or side effects. And the idea is again we don't think about anything having local state. We think about the bank account as something that's going to process a stream of transaction requests. So think about this bank account not as something that goes message by message, but something that takes in a stream of transaction requests like maybe successive deposit announced. 1, 2, 2, 4, those might be successive amounts to deposit. And then coming out of it is the successive balances 1, 3, 5, 9.

So we think of the bank account not as something that has state, but something that acts sort of on the infinite stream of requests. But remember, we've thrown away time. So what we can do is if the user's here, we can have this infinite stream of requests being generated one at a time coming from the user and this transaction stream coming back on a printer being printed one at a time. And if we drew a little line here, right there to the user, the user couldn't tell that this system doesn't have state. It looks just like the other one, but there's no state in there.

And by the way, just to show you, here's an actual implementation of this-- we'll call it make deposit account because you can only deposit. It takes an initial balance and then a stream of deposits you might make. And what is it? Well, it's just cons-stream of the balance onto make a new account stream whose initial balance is the old balance plus the first thing in the deposit stream and make deposit account works on the rest of which is the tail of the deposit stream. So there's sort of a very typical message-passing, object-oriented thing that's done without side effects at all. There are very many things you can do this way.

Well, can you do everything without assignment? Can everybody go over to purely functional languages? Well, we don't know, but there seem to be places where purely functional programming breaks down. Where it starts hurting is when you have things like this, but you also mix it up with the other things that we had to worry that, which are objects and sharing and two independent agents being the same.

So under a typical one, suppose you want to extend this bank account. So here's a bank account. Bank accounts take in a stream of transaction requests and put out streams of, say, balances or responses to that. But suppose you want to model the fact that this is a joint bank account between two independent people. So suppose there are two people, say, Bill and Dave, who have a joint bank account. How would you model this?

Well, Bill puts out a stream of transaction requests, and Dave puts out a stream of transaction requests, and somehow, they have to merge into this bank account. So what you might do is write a little stream processing thing called merge, which sort of takes these, merges them together, produces a single stream for the bank account. Now they're both talking to the same bank account. That's all great, but how do you write merge? What's this procedure merge? You want to do something that's reasonable.

Your first guess might be to say, well, we'll take alternate requests from Bill and Dave. But what happens if suddenly in the middle of this thing, Dave goes away on vacation for two years? Then Bill's sort of stuck. So what you want to do is-- well, it's hard to describe. What you want to do is what people call fair merge. The idea of fair merge is it sort of should do them alternately, but if there's nothing waiting here, it should take one twice.

Notice I can't even say that without talking about time. So one of the other active researcher areas in functional languages is inventing little things like fair merge and maybe some others, which will take the places where I used to need side effects and objects and sort of hide them away in some very well-defined modules of the system so that all the problems of assignment don't sort of leak out all over the system but are captured in some fairly well-understood things.

More generally, I think what you're seeing is that we're running across what I think is a very basic problem in computer science, which is how to define languages that somehow can talk about delayed evaluation, but also be able to reflect this view that there are objects in the world. How do we somehow get both? And I think that's a very hard problem. And it may be that it's a very hard problem that has almost nothing to do with computer science, that it really is a problem having to do with two very incompatible ways of looking at the world. OK, questions?

AUDIENCE: You mentioned earlier that once you introduce assignment, the general rule for using the substitution model is you can't. Unless you're very careful, you can't.

PROFESSOR: Right.

AUDIENCE: Is there a set of techniques or a set of guidelines for localizing the effects of assignment so that the very careful becomes defined?

PROFESSOR: I don't know. Let me think. Well, certainly, there was an assignment inside memo proc, but that was sort of hidden away. It ended up not making any difference. Part of the reason for that is once this thing triggered that it had run and gotten an answer, that answer will never change. So that was sort of a one-time assignment. So one very general thing you can do is if you only do what's called a one-time assignment and never change anything, then you can do better.

One of the problems in this merge thing, people have-- let me see if this is right. I think it's true that with fair merge, with just fair merge, you can begin effectively simulating assignment in the rest of the language. It seems like anything you do to go outside-- I'm not quite sure that's true for fair merge, but it's true of a little bit more general things that people have been doing. So it might be that any little bit you put in, suddenly if they allow you to build arbitrary stuff, it's almost as bad as having assignment altogether. But that's an area that people are thinking about now.

AUDIENCE: I guess I don't see the problem here with merge if I call Bill, if Bill is a procedure, then Bill is going to increment the bank account or build the list that 's going to put in the next element. If I call Dave twice in a row, that will do that. I'm not sure where fair merge has to be involved.

PROFESSOR: The problem is imagine these really as people. See, here I have the user who's interacting with this bank account. Put in a request, get an answer. Put in a request, get an answer.

AUDIENCE: Right.

PROFESSOR: But if the only way I can process request is to alternate them from two people--

AUDIENCE: Well, why would you alternate them?

PROFESSOR: Why don't I?

AUDIENCE: Yes. Why do you?

PROFESSOR: Think of them as real people, right? This guy might go away for a year. And you're sitting here at the bank account window, and you can't put in two requests because it's waiting for this guy.

AUDIENCE: Why does it have to be waiting for one?

PROFESSOR: Because it's trying to compute a function. I have to define a function. Another way to say that is the answer to what comes out of this merge box is not a function of what goes in. Because, see, what would the function be? Suppose he puts in 1, 1, 1, 1, and he puts in 2, 2, 2, 2. What's the answer supposed to be? It's not good enough to say it's 1, 2, 1, 2, 1, 2.

AUDIENCE: I understand. But when Bill puts in 1, 1 goes in. When Dave puts in 2 twice, 2 goes in twice. When Bill puts in--

PROFESSOR: Right.

AUDIENCE: Why can't it be hooked to the time of the input-- the actual procedural--

PROFESSOR: Because I don't have time. See, all I can say is I'm going to define a function. I don't have time. There's no concept if it's going to alternate, except if nobody's there, it's going to wait a while for him. It's just going to say I have the stream of requests, the timeless infinite streams of all the requests that Dave would have made, right? And the timeless infinite stream of all the requests Bill would have made, and I want to operate on them. See, that's how this bank account is working.

And the problem is that these poor people who are sitting at the bank account windows have the misfortune to exist in time. They don't see their infinite stream of all the requests they would have ever made. They're waiting now, and they want an answer. So if you're sitting there-- if this is the screen operation on some time-sharing system and it's working functionally, you want an answer then when you talk the character. You don't want it to have to wait for everybody in the whole system to have typed one character before it can get around to service you. So that's the problem. I mean, the fact that people live in time, apparently. If they didn't, it wouldn't be a problem.

AUDIENCE: I'm afraid I miss the point of having no time in this banking transaction. Isn't time very important? For instance, the sequence of events. If Dave take out $100, then the timing sequence should be important. How do you treat transactions as streams?

PROFESSOR: Well, that's the thing I'm saying. This is an example where you can't. You can't. The point is what comes out of here is simply not a function of the stream going in here and the stream going in here. It's a function of the stream going in here and the stream going in here and some kind of information about time, which is precisely what a normal-order language won't let you say.

AUDIENCE: In order to brings this back into a more functional perspective, could we just explicitly time stamp all the inputs from Bill and Dave and define fair merge to just be the sort on those time stamps?

PROFESSOR: Yeah, you can do that. You can do that sort of thing. Another thing you could say is imagine that really what this function is, is that it does a read every microsecond, and then if there's none there, that's considered an empty one. That's about equivalent to what you said. And yes, you can do that, but that's a clg. So it's not quite only implementation we're worried about. We're worried about expressive power in the language, and what we're running across is a real mismatch between what we can say easily and what we'd like to say.

AUDIENCE: It sounds like where we're getting hung up with that is the fact it expects one input from both Bill and Dave at the same time.

PROFESSOR: It's not quite one, but it's anything you define. So you can say Dave can go twice as often, but if anything you predefine, it's not the right thing. You can't decide at some particular function of their input requests. Worse yet, I mean, worse yet, there are things that even merge can't do. One thing you might want to do that's even more general is suddenly you add somebody else to this bank account system. You go and you add John to this bank account system. And now there's yet another stream that's going to come into the picture at some time which we haven't prespecified.

So that's something even fair merge can't do, and they're things called-- I forget-- natagers or something. That's a generalization of fair merge to allow that. There's a whole sort of research discipline saying how far can you push this functional perspective by adding more and more mechanism? And how far does that go before the whole thing breaks down and you might as well been using set anyway.

AUDIENCE: You need to set him up on automatic deposit.

[LAUGHTER]

PROFESSOR: OK, thank you.