Limitations of the Sequential Model of Computation

We have introduced the declarative sequential kernel language (DSKL) and explored some ways of using it, e.g.,

- list processing and declarative data structures,
- higher-order and functional programming, etc.

Some of the properties of this model may lead to problems, e.g.,

- a statement can only be executed if the execution of all preceding statements has been completed;\footnote{Statements can be ‘popped from the semantic statement stack’ using exceptions, but this does not solve the problem of sequential execution.}
- while executing nested operations on lists, each inner operation has to be completed before any outer operation can start traversing its input;
- statements that freeze (suspend) over an unbound variable cause the whole program to stop before the computation is completed.

Example (Sequential execution of statements)

\[
\begin{array}{|c|}
\hline
\text{Integers} = \{\text{Enumerate 1 30}\} & \% (1) \\
\text{Fibs} = \{\text{Map Integers Fib}\} & \% (2) \\
\text{EvenFibs} = \{\text{Filter Fibs IsEven}\} & \% (3) \\
\text{Browse EvenFibs}\} & \% (4) \\
\hline
\end{array}
\]

- (2) will only be executed when (1) has been completed (we first generate all integers from 1 to 30, then compute the corresponding Fibonacci numbers);
- (3) will only be executed when (2) has been completed (we first compute all Fibonacci numbers with indices from 1 to 30, then filter out the odd ones);
- in general, step (i + 1) will only be executed when step (i) has been completed.
Limitations of the Sequential Model of Computation contd

Example (Execution of nested list operations)

<table>
<thead>
<tr>
<th>Browse</th>
<th>% (4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>% (3)</td>
</tr>
<tr>
<td>Map</td>
<td>% (2)</td>
</tr>
<tr>
<td>Enumerate 1 30</td>
<td>% (1)</td>
</tr>
</tbody>
</table>
| F!b) | IsEven |}

The source code order of the calls is inversed wrt. the previous example. But the execution order stays the same, since Oz uses eager evaluation (meaning arguments are evaluated before a procedure is called) i.e.

- Filter will not be called before Map has completed, and
- Map will not be called before Enumerate has completed.

Limitations of the Sequential Model of Computation contd

Example (Freezable statements)

... case X of int(I) then I [] var(V) then {Lookup Environment V} [] exp(E) then {Evaluate Environment E} ...}

- If X happens to be an unbound variable, the case statement freezes, and the program never completes.

In the DSKL, freezable statements are of no use when they do freeze. In a concurrent model of computation, performance in cases where piped lists processors are involved can be improved by parallelizing the computation, and freezable statements are useful for synchronizing the threads.

Limitations of the Sequential Model of Computation contd

Consider the process of lexemizing, tokenizing, and parsing a program.²

Example (Lexemization, tokenization, and parsing)

```plaintext
Program = "local X in ... end" % [&l &o &c ... &e &n &d]
Lexemes = {Lexemize Program}
Tokens = {Tokenize Lexemes}
ASTree = {Parse Tokens}
```

- What if some lexemes (e.g., the first one) cannot be recognized by the tokenizer?
- What is some subsequence of tokens (e.g., the first three tokens) cannot be successfully parsed?

In such cases, an early processor (e.g., lexemizer or tokenizer) performs unnecessary computation over the whole program, while it could have been stopped by a later processor (e.g., tokenizer or parser).

²The first distinction is somewhat artificial, as in most cases tokenization is done simultaneously with lexemization; see earlier lectures.

Lecture Outline

Limitations of the Sequential Model of Computation

Chapter 4: Overview

Declarative Concurrency
- Programming with Declarative Concurrency
- The Declarative Concurrent Model of Computation
- Concurrency, Determinism, and Declarativeness
- Streams

Summary
Chapter 4: Overview

§ 4.1 The data-driven concurrent model.
§ 4.2 Basic thread programming techniques.
§ 4.3 Streams.
§ 4.4 Using the declarative concurrent model directly.
§ 4.5 Lazy execution.
§ 4.6 Soft real-time programming.
§ 4.7 The Haskell language.
§ 4.8 Limitations and extensions of declarative programming.
§ 4.9 Advanced topics.
§ 4.10 Historical notes.

Lecture Outline

Limitations of the Sequential Model of Computation

Chapter 4: Overview

Declarative Concurrency

Programming with Declarative Concurrency

The declarative concurrent model of computation (the declarative concurrent kernel language, DCKL) is an extension of the declarative sequential model. There are two ways in which we can extend DSKL with concurrency:

- **Data-driven concurrency**: a thread of computation can be executed as soon as it has all data it needs.
- **Demand-driven concurrency**: a thread of computation can be executed no sooner than when the result of its execution is needed by another thread.

We will examine both ways, and have them combined in one concurrent model.4

### Declarative Concurrency

What is concurrency?

Concurrency is the progressing of **two or more activities** (processes, tasks, threads) **in parallel**.

Concurrent programs may bring the benefits of:
- more efficient use of resources (e.g., processor time);
- more intuitive and realistic simulations of real-world systems;
- disseminating tasks among a number of processors or computer nodes in a distributed computation environment; etc.

Even if we run programs on non-concurrent machines, we can simulate concurrency, to some extent at least.3

---

3There is a distinction made between concurrency and parallelism; in the latter case there are processes running in parallel, the former may be a sequential simulation with interleaving.

4In fact, both the eager and the lazy models of computation are independent of concurrency, and can be included in a sequential language.
Suppose our program needs to do different things with a list of all non-negative integers (e.g., take the first \( n \) integers, take the \( n \)-th one, etc.)

**Example (Integers, the sequential way)**
code/integers-sequential.oz

```oz
Integers =
  local
    fun {Generate From}
      From|{Generate From+1} end in
    {Generate 0} end
  end

Let's now try to see the first \( N \) integers:

```{Browse {List.take Integers N}}

What happens?

- Nothing is displayed, since the list has not been completed yet.

We can solve the problem by moving the production to a separate thread.

**Example (Integers, the concurrent way)**
code/integers-concurrent.oz

```oz
Integers =
  local
    fun {Generate From}
      From|{Generate From+1} end in
    thread {Generate 0} end end

Let's now try to see the first \( N \) integers:

{Browse {List.take Integers N}}

What happens?

- We see the list \([0 1 2 3 4 5 6 7 8 9]\) displayed.
- But Integers is still being produced!^5

^5We shall later see how this can be solved with lazy functions.

We can use the same pattern to process finite lists while they grow.

**Example (Computing Fibonacci numbers\(^6\) with Map)**
code/fib-naive.oz

```oz
fun {Fib N}
  if N < 3 then 1
  else \{Fib N-1\} + \{Fib N-2\} end end
Indices = [38 3 36 16 29 33 1]
{Browse {Map Indices Fib}} % (1)
{Browse thread {Map Indices Fib} end} % (2)

In (1), we first see nothing. Only when the complete list has been constructed will it be displayed.

In (2), we first see \_\_ (corresponding to the record created in Map, just as in Generate), which is then iteratively extended when a Fibonacci number is computed.

\(^6\)Using a naive implementation of Fib precisely because it performs slowly, so that we can see things happen in our time scale.
Programming with Declarative Concurrency contd

How does it work? Recall the kernel language version of Map.

Example (Map, quasi-kernel version)
code/map-kernel.oz

```
Map = proc {$ List Procedure ?Result}
    case List of Head|Tail then
        local PartialResult NewHead in
            Result = NewHead|PartialResult
            {Procedure Head NewHead}
        {Map Tail Procedure PartialResult} end
    else Result = nil end end
```

▷ A new record is first constructed, then its head is bound to a transformation of the original head, and then its tail is bound in the recursive call.

The threaded solution is fine in that we can access elements of a list as soon as they appear, instead of waiting for the whole list to be produced first.

▷ We can ‘consume’ elements from a list while the list is still being produced.

But what if the elements are produced with unequal speed (like the Fibs above)? It could be useful to be able to have smaller Fibonacci numbers accessible on the list while the larger ones are still being produced.

▷ Instead of enclosing the whole mapping operation in a separate thread, we need to

▷ spawn a thread for each individual application of the transform to an item on the input list.

Sounds easy (?)

Programming with Declarative Concurrency contd

Example (Concurrent Map)
code/map-concurrent.oz

```
fun {CMap List Function}
    case List of Head|Tail then
        thread {Function Head} end|{CMap Tail Function}
    else nil end end
```

Indices = [3 38 36 16 29 33 1]
{Browse {CMap Indices Fib}}

▷ CMap immediately constructs the whole list, containing just unbound variables.

▷ In separate threads, each variable on the list is bound to a Fibonacci number, as soon as it is computed.

Try it! Execute the file code/fib-test.oz and compare the output from the three different calls—watch carefully what happens:

```
Indices = [3 38 36 16 29 33 1]
{Browse {CMap Indices Fib}}
{Browse thread {Map Indices Fib} end}
{Browse {Map Indices Fib}}
```

▷ Which version is most convenient?

▷ Why is the threaded application of Map not faster than the sequential one?

▷ Why is CMap considerably faster than the threaded application of Map, even though they have the same amount of work to do? Shouldn’t they differ just in the order in which items appear in the output?

(Here’s a ‘concurrency trick’: in the case of Fib-mapping, there is no workload reduction by using concurrency; however, with a fair scheduler, the more threads you create, the more you get done before others.)
The Declarative Concurrent Model of Computation

To accommodate for concurrency, we need to significantly extend our model of computation.

Syntax for computing with threads

\[
\langle \text{statement} \rangle ::= \ldots \\
| \quad \text{thread} \langle \text{statement} \rangle \text{ end}
\]

- At the syntax level, there is just one simple statement form added to the language.

The Declarative Concurrent Model of Computation contd

We need to extend the abstract machine . . .

Extension of the abstract machine

The abstract machine for the sequential model had the following structure:

Abstract machine

Semantic stack

\[
\begin{align*}
& s_1, E_1 \\
& s_2, E_2 \\
& s_3, E_3 \\
& s_4, E_4
\end{align*}
\]

Single assignment store

Extension of the abstract machine contd

The abstract machine has the following extension:

- The semantic stack is replaced by a multiset of semantic stacks.
- Each semantic stack in the multiset corresponds to one thread.
- Why a multiset? Because two threads may happen to have identical semantic statements on their corresponding stacks — their stacks can be identical (a set cannot include the same element twice).\footnote{In practice, we would like the threads to have uneven chances for execution, and organize them, e.g., using a priority queue.}

The Declarative Concurrent Model of Computation contd

... with multiple semantic stacks.

Extension of the abstract machine contd

The abstract machine for the concurrent model has the following structure:

Abstract machine

Multiset of semantic stacks

\[
\begin{align*}
\ldots & s_1, E_1 \\
\ldots & s_2, E_2 \\
\ldots & s_3, E_3 \\
\ldots & s_4, E_4
\end{align*}
\]

Single assignment store
The Declarative Concurrent Model of Computation contd

Extension of the abstract machine contd

The initial state of the sequential machine:

semantic statement
{ ( [ (statement), {}], {} ) }

multiset

The initial state of the concurrent machine:

semantic statement
{ { (statement), {} }, {} }

multiset

Semantics for computing with threads

◮ A semantic stack is executable if its top-most semantic statement is not frozen (is not in suspension).
◮ At each step of the computation an executable semantic stack is chosen, and its top-most statement is executed.
◮ The choice of a stack is not specified by the semantics (it is non-deterministic from this point of view). It is an implementational detail of the scheduler.

Semantics for computing with threads contd

◮ We assume that the scheduler is fair: each executable stack will eventually be chosen, and stacks are chosen with equal frequency (on average). *
◮ When a semantic stack is empty, it can be removed.
◮ When all semantic stacks are removed, the computation has finished.

*More precisely, we want each thread to have equal chance of execution among threads within its priority group.

Execution of the thread statement

The semantic statement is:

(thread (statement) end, E)

The rule of execution is:
1. Create a new semantic stack.
2. Put onto the new stack the semantic statement

((statement), E)
The Declarative Concurrent Model of Computation

Example (Execution of the thread statement)

```plaintext
proc {RunLikeCrazy N}
  if N > 0 then
    thread {RunLikeCrazy N-1}
    thread {RunLikeCrazy N-2} end
  end
end
{RunLikeCrazy 2}
```

Quiz: 1. How many threads will be started during an execution of the program? (five)
2. What is the maximal number of threads that can run concurrently during an execution of the program? (two)

The kernel semantics for computing with threads specifies how to execute concurrent programs on a sequential machine (concurrency without parallelism, simulated concurrency, SISD).

- In practice we may be interested in other sorts of concurrency, e.g., running multiple concurrent programs on multiple processors (concurrency with parallelism, MIMD) or in executing one instruction on multiple pieces of data at once (SIMD).

SISD: single instruction single data; MIMD: multiple instructions multiple data; SIMD: single instruction multiple data (from Flynn’s classification of concurrency).

Example (SISD and SIMD, Fortran)

SISD:

```fortran
do i=2, n-1
  output(i) = output(i-1) + output(i) + output(i+1)
end do
```

SIMD:

```fortran
forall (i=2:n-1)
  output(i) = output(i-1) + output(i) + output(i+1)
end forall
```

- The `do` loop computes the update for each element of the array separately; each change is immediately visible to the subsequent steps.
- The `forall` loop computes the update for each element of the array based on the original values, and all updates can be executed at the same time, unlike in the former case.

Try it! $ gfortran forall.f95 -o forall && ./forall

Concurrency, Determinism, and Declarativeness

With sequential programs, it was easy to predict the order of execution of statements within a program. With concurrency added, it is no longer easy.

Example (Concurrent execution)

```plaintext
local x in
  (Browse a)
  thread {Browse b} x=unit end
  (Browse c)
  case x of unit then skip end
  (Browse d) end
```

- What is the order of appearance of elements in the output?
Example (Concurrent execution contd)
There are the following dependencies:

\[ a \rightarrow c \rightarrow d \]
\[ b \]

The following orders of execution are possible:
- \[ a \rightarrow b \rightarrow c \rightarrow d \]
- \[ a \rightarrow c \rightarrow b \rightarrow d \]

What is non-determinism?
Non-determinism is a feature of computation which, at a certain point, can arbitrarily choose among two or more ways to proceed further.
- It is impossible to predict in advance which way will the computation proceed, although we may be able, in principle, to predict which are the ways the computation can proceed.
- The same program may lead to different computations at different executions.
- The analysis of concurrent programs with non-deterministic choices is difficult.

Is the data-driven concurrent model of computation declarative? To be declarative, a concurrent program must fulfil exactly one of the following conditions:
- All executions of the program must have the same result (= the results of all executions must be logically equivalent). \(^{10}\)
- All executions of the program must lead to a failure.

The data-driven concurrent model does not add to the declarative sequential model (without exceptions) any feature that would make it non-declarative.
- Irrespectively of the order of computation in different executions, the result must always be the same, or the program must always fail. \(^{11}\)
- A non-deterministic program can be declarative.

---

\(^{10}\)This condition, however, requires us to clearly specify what ‘result’ is taken to mean.

\(^{11}\)Again, it must be clear what qualifies as a result.
Concurrency, Determinism, and Declarativeness

If a program contains two (or more) concurrent statements that bind the same variable to two (or more) different, non-unifiable values, one of them will succeed, the others must fail.

- The program must fail in any case.
- However, in different executions, the variable will have been bound to different values at the time the unification failure occurs.
- If this binding is not a part of the result, then the program is declarative.
- If this binding is a part of the result, the program is not declarative\(^\text{12}\) — the program always fails over unification, but the variable has been bound, and its value varies from execution to execution.

\(^{12}\)And thus the data-driven concurrent model of computation is not declarative (see next footnote).

---

Example (Concurrency and exceptions)

(1)

```plaintext
local X in
  thread X=1 end
  thread X=2 end
  \{Browse X\} end
```

- In both programs, there will be a unification failure.
- In (1), one of the spawned threads will fail, but the main thread will not; the value of X will always be displayed (1 or 2, on different occasions).
- In (2), sometimes 2 will be displayed, on other occasions the program will fail with no output.

Oops! Is this declarative? Should we blame Browse for the apparently non-declarative behaviour?

---

Example (Concurrency and exceptions)

(2)

```plaintext
local X in
  thread X=1 end
  \{Browse X\} end
```

We can handwave the problem away by claiming that the behaviour above is a consequence of the implicitly involved exception handling mechanism:

- the unification failure propagates, as an exception, within the confines of the involved thread, causing the thread to fail while
- the main thread continues without even noticing the failure.\(^{14}\)

\(^{14}\)That is, it can be claimed that is the fault of the implementation of Oz in which exceptions are included, rather than of the conceptual model, may this convince you.

---

Example (Concurrency and exceptions, Java)

code/failure.java

```java
public class failure {
  public static void main(String[] args) {
    (new Thread() { public void run() { int x = 1/0; } }).start();
    System.out.println("I'm still alive!");
  }
}
```

---

Example (Concurrency and exceptions)

According to CTMCP, combining concurrency with exceptions (i.e., extending DSKL with exceptions and with concurrency) leads to a non-declarative model of computation.\(^{15}\)

---

\(^{15}\)As in the previous footnote, we can argue that it is not the case that combining concurrency with exceptions leads to non-declarativeness, but rather that exceptions (exception handling) just allow us to see the non-declarative behaviour of a concurrent program.
Concurrent, Determinism, and Declarativeness

How about this program?

Example (Concurrency)

```lisp
local X in
  thread if X==1 then skip else X=0 end end
  thread if X==0 then skip else X=1 end end
{Browse X} end
```

- There will be no output: both threads freeze over the comparison statement since \(X\) is unbound; the program never terminates.
- This is a simple example of a deadlock, where \(X\) is the resource and both threads wait for the resource to become bound.

The Thread module

You can find the following operations on threads useful:

- `{Thread.this}` returns the name of the current thread.
- `{Thread.suspend T}` stops thread \(T\); \(T\) remains stopped until it is explicitly resumed.
- `{Thread.resume T}` reactivates thread \(T\) if it was suspended.
- `{Thread.preempt T}` preempts thread \(T\); \(T\) can be resumed by the scheduler at any time with no explicit reactivation.
- `{Thread.terminate T}` terminates thread \(T\); \(T\) can no longer be run.

Note: These operations may lead to non-declarativeness. (They are not part of the declarative model!)

Concurrent, Determinism, and Declarativeness

What does the following program do?

Example (The Thread module)

```lisp
local T1 T2 X in
  thread T1 = {Thread.this} {Thread.terminate T2} X=1 end
  thread T2 = {Thread.this} {Thread.terminate T1} X=2 end
{Browse X} end
```

- One of the threads succeeds in terminating the other, and then binds \(X\).
- Either 1 or 2 will be displayed, non-deterministically.
Concurrency, Determinism, and Declarativeness

The thread statement creates and starts a thread. In some languages, you can create a thread and then start it.

Example (Creating a ready-to-run thread in Oz)

```oz
fun {MakeThread Thunk}
  This in thread
  This = {Thread.this}
  {Thread.suspend This}
  {Thunk} end
  proc {Message $ Message}
    try {Thread.Message This}
    catch _ then skip end end
end
```

T = {MakeThread proc {}} Result = {Map [10 20 30] Fib} end
(T resume)

* The returned value is a procedure which starts the new thread when passed resume as the message. Thunk is a procedure of no arguments (the thread's body).

Streams

What is a stream?

A stream is a potentially infinite data structure, where
- elements can be read from the beginning, and
- elements can be added to the end.

That is, a stream can be both read and extended.

A stream is an abstraction for the flow of data:
- one process (a producer) adds elements to the stream,
- another process (a consumer) reads elements from the stream. (There can be more than one consumer.)

The extending and reading of a stream can be performed concurrently.
- A stream can be produced and consumed at the same time, in separate threads.
- ... But a stream can also have the form of a sequence of delayed computations that are completed (enforced) when there is demand for the data.

Delayed computations can be realized by lazy execution, and do not require concurrency.¹⁵

Benefits of using a stream:
- The stream can be consumed without waiting for the producer to finish the production.
- It is not necessary to keep all elements of the stream in memory (elements can be garbage-collected after they have been read).
- The overall performance can be improved; e.g., if the production does not occupy 100% of the CPU time, or if production and consumption can be run on different CPU's.
- It is possible to perform operations on potentially infinite amounts of data (e.g., the list of all natural numbers).

¹⁵Lazy computation will be introduced later. In CTMCP, lazy computation is realized using threads.
Streams contd

In Oz, streams can be implemented using a list-like structure with an unbound end. 16

Example (Streams)

```
declare UnboundTail
Stream = a|b|c|UnboundTail
```

We can read from the stream by starting from the head:

```
Head|NewStream = Stream
```

We can extend the stream by binding the tail:

```
d|NewUnboundTail = UnboundTail
```

16This is just the same as we have been doing with diff lists. But diff lists represent lists; a stream is conceptually something else again, but implemented in Oz with list structures with an unbound tail.

Lecture 12: Declarative Concurrency. Streams. (49/63)

A stream can be read concurrently while it is being produced.

Example (Connecting a producer and a consumer)

```
local Integers Result in
  thread Integers = {Enumerate 0 1000} end
  thread Result = {Sum Integers} end
  {Browse Integers}
  {Browse Result}
end
```

◮ Enumerate is a producer (we have seen it before).
◮ Sum is a consumer (we have seen it before).
◮ Integers is a stream that flows from Enumerate to Sum.

Just like pipelines in list processing, only that we use concurrency and can thus handle potentially infinite streams.

Lecture 12: Declarative Concurrency. Streams. (50/63)

A stream can be read concurrently by more than one consumer.

Example (Connecting a producer with more than one consumer)

```
local Integers Result in
  thread Integers = {Enumerate 0 1000} end
  thread Summed = {Sum Integers} end
  thread Odd = {Filter Integers IsOdd} end
  thread SummedOdd = {Sum {Filter Integers IsOdd}} end
  ...
end
```

Just like pipelines in list processing, only that using concurrency we can have any number of readers performing concurrently on a single stream.

Lecture 12: Declarative Concurrency. Streams. (51/63)

We can rewrite many of our previous list processors to work with possibly infinite streams.17

◮ In some cases, it suffices to wrap the original list processor into a separate thread.

Example (Stream processors)

```
fun {StreamEnumerate Start Next Stop}
  if {Stop Start} then nil
  else Start|thread {StreamEnumerate {Next Start} Next Stop} end end
```

or (with just one new thread)

```
fun {StreamEnumerate Start Next Stop}
  thread {Enumerate Start Next Stop} end end
```

17Note: a list processor would never return if applied to an infinite stream; if wrapped into a separate thread, partial output from some, but not all, of the list processors is available while the rest is still being produced.
A list processor that is defined in terms of other list processors can be converted into a stream processor if the embedded list processors are also converted into stream processors.\(^{18}\)

**Example (Stream processors contd)**

this doesn't work (`Mux` is defined in terms of `Map` and `Transpose`):

```oz
fun {StreamMux Lists Combine}
  thread {Mux Lists Combine} end end
```

we need something like this:

```oz
fun {StreamMux Streams Combine}
  {StreamMap {StreamTranspose Streams} Combine} end
```

This must also be done with any list processors embedded in the embedded list processors, recursively.

What about `FoldRight`?

- `FoldRight` accumulates postponed computations until it reaches the end of the input, so that embedding it in a separate thread won't help with infinite streams.
- But we can implement a stream version so that it will work as a basis for stream transducers.

**Example (Limited FoldRight for streams)**

```oz
fun {StreamFoldRight Stream Null Transform Combine}
  case Stream of Head|Tail
    then {Combine {Transform Head}
      thread {StreamFoldRight Tail Null Transform Combine} end}
  else Null end end

fun {StreamMap Stream Function}
  {StreamFoldRight Stream nil Function Cons} end
```

How can we generate an infinite stream of all Fibonacci numbers?

- We could use `Enumerate` to generate an infinite stream of consecutive integers, and then stream-map `Fib` onto it.

**Example (Infinite stream of Fibonacci numbers)**

```oz
Fibs =
  {StreamMap
    thread {Enumerate 1 fun {$ N} N+1 end fun {$ _} false end} end
    Fib}
```

**Note:** This is a consequence of using dataflow variables. In list-processing languages without dataflow variables (e.g., Scheme), fold-right would not work this way.\(^{19}\)

---

\(^{18}\)This must also be done with any list processors embedded in the embedded list processors, recursively.

\(^{19}\)In Scheme, streams are implemented using lazy computation.
How can we generate an infinite stream of all Fibonacci numbers?

- We could use the more generic `Enumerate` (lecture 9) to generate an infinite stream of consecutive integers immediately transformed with `Fib`.

**Example (Infinite stream of Fibonacci numbers)**

```oz
Fibs = thread {Enumerate 1 fun {$ N} N+1 end fun {$ _} false end Fib} end
```

- But each Fibonacci number is computed from scratch—even if the efficient iterative version is used, there's still a lot of redundant computation done.

**Example (Infinite stream of Fibonacci numbers)**

```oz
Fibs = 1|1|{StreamAdd Fibs {StreamDrop Fibs 1}}
```

- `StreamAdd` adds two streams, creating another stream.
- `StreamDrop` skips the first `n` elements from a stream, returning the rest.

**Note:** There's just one stream under construction! (Both arguments to `StreamAdd` are different parts of the same stream `Fibs`, and the result forms the rest of that very stream). And we don't even use the function `Fib`...

**Try it!** Execute the file `code/fib-stream.oz`.

**Note:** `Fibs` is an infinite stream that is being produced once you start its production. We will soon see how to create an infinite stream of Fibonacci numbers that grows only if it is read.

How does it work? Here are a few initial steps:

```
1|1|{StreamAdd 1|1|_ (StreamDrop 1|1|_ 1)}
1|1|{StreamAdd 1|1|_ 1|_}
1|1|2|(StreamAdd 1|2|_ 2|_)
1|1|2|3|(StreamAdd 2|3|_ 3|_)
1|1|2|3|5|(StreamAdd 3|5|_ 5|_)
...
```

All we need is `StreamAdd` and `StreamDrop`.\(^{20}\)

\(^{20}\)Since we know the count passed to `StreamDrop` in advance, `(StreamDrop Fibs 1)` can be replaced with the equivalent `Fibs2`, for simplicity.
Lecture Outline

Limitations of the Sequential Model of Computation

Chapter 4: Overview

Declarative Concurrency
  Programming with Declarative Concurrency
  The Declarative Concurrent Model of Computation
  Concurrency, Determinism, and Declarativeness
  Streams

Summary

This time
  ◮ The declarative concurrent model of computation.
  ◮ Concurrency, determinism, and declarativeness.
  ◮ Streams.

Next time
  ◮ More on streams.
  ◮ Lazy execution.

Summary contd

Homework
  ◮ Examine and try out today's code, read Mozart/Oz documentation if necessary.
  ◮ Read Ch. 4 in CTMCP (Secs. 4.1–4.3).

Pensum
  ◮ All of today's slides, except for non-Oz code (if any).

Further reading
  ◮ SICP provides a very interesting discussion of streams as potentially infinite data structures (they use lazy execution, we'll come to that.)