

Typing, Parameter Passing, Lazy Evaluation

Dynamic and Static Typing (EPL 4.1-4.4, VRH 2.8.3)
Parameter Passing (VRH 6.1-6.4.4)
Lazy Evaluation (VRH 4.5)

Carlos Varela

Rensselaer Polytechnic Institute

May 2, 2011

Partially adapted with permission from:

Seif Haridi

KTH

Peter Van Roy

UCL

Data types

- A datatype is a set of values and an associated set of operations
- An abstract datatype is described by a set of operations
- These operations are the only thing that a user of the abstraction can assume
- Examples:
 - Numbers, Records, Lists,... (Oz basic data types)
 - Stacks, Dictionaries,... (user-defined secure data types)

Types of typing

- Languages can be *weakly typed*
 - Internal representation of types can be manipulated by a program
 - e.g., a string in C is an array of characters ending in ‘\0’.
- *Strongly typed* programming languages can be further subdivided into:
 - *Dynamically typed* languages
 - Variables can be bound to entities of any type, so in general the type is only known at **run-time**, e.g., Oz, SALSA.
 - *Statically typed* languages
 - Variable types are known at **compile-time**, e.g., C++, Java.

Type Checking and Inference

- *Type checking* is the process of ensuring a program is well-typed.
 - One strategy often used is *abstract interpretation*:
 - The principle of getting partial information about the answers from partial information about the inputs
 - Programmer supplies types of variables and type-checker deduces types of other expressions for consistency
- *Type inference* frees programmers from annotating variable types: types are inferred from variable usage, e.g. ML.

Example: The identity function

- In a dynamically typed language, e.g., Oz, it is possible to write a generic function, such as the identity combinator:

```
fun {Id X} X end
```

- In a statically typed language, it is necessary to assign types to variables, e.g. in a **statically typed variant of Oz** you would write:

```
fun {Id X:integer}:integer X end
```

These types are checked at compile-time to ensure the function is only passed proper arguments. `{Id 5}` is valid, while `{Id Id}` is not.

Example: Improper Operations

- In a dynamically typed language, it is possible to write an improper operation, such as passing a non-list as a parameter, e.g. in Oz:

```
declare fun {ShiftRight L} 0|L end
  {Browse {ShiftRight 4}}           % unintended misuse
  {Browse {ShiftRight [4]}}        % proper use
```

- In a statically typed language, the same code would produce a type error, e.g. **in a statically typed variant of Oz** you would write:

```
declare fun {ShiftRight L>List}:List 0|L end
  {Browse {ShiftRight 4}}           % compiler error!!
  {Browse {ShiftRight [4]}}        % proper use
```

Example: Type Inference

- In a statically typed language with type inference (e.g., ML), it is possible to write code without type annotations, e.g. using Oz syntax:

```
declare fun {Increment N} N+1 end
{Browse {Increment [4]}}           % compiler error!!
{Browse {Increment 4}}            % proper use
```

- The type inference system knows the type of ' + ' to be:

`<number> X <number> → <number>`

Therefore, **Increment** must always receive an argument of type `<number>` and it always returns a value of type `<number>`.

Static Typing Advantages

- Static typing restricts valid programs (i.e., reduces language's expressiveness) in return for:
 - Improving error-catching ability
 - Efficiency
 - Security
 - Partial program verification

Dynamic Typing Advantages

- Dynamic typing allows all syntactically legal programs to execute, providing for:
 - Faster prototyping (partial, incomplete programs can be tested)
 - Separate compilation---independently written modules can more easily interact--- which enables open software development
 - More expressiveness in language

Combining static and dynamic typing

- Programming language designers do not have to make an *all-or-nothing* decision on static vs dynamic typing.
 - e.g, Java has a root **Object** class which enables *polymorphism*
 - A variable declared to be an **Object** can hold an instance of any (non-primitive) class.
 - To enable static type-checking, programmers need to annotate expressions using these variables with *casting* operations, i.e., they instruct the type checker to pretend the type of the variable is different (more specific) than declared.
 - Run-time errors/exceptions can then occur if type conversion (casting) fails.
- Alice (Saarland U.) is a statically-typed variant of Oz.

Parameter Passing Mechanisms

- Operations on data types have arguments and results. Many mechanisms exist to pass these arguments and results between calling programs and abstractions, e.g.:
 - Call by reference
 - Call by variable
 - Call by value
 - Call by value-result
 - Call by name
 - Call by need
- We will show examples in Pascal-like syntax, with semantics given in Oz language.

Call by reference

```
procedure sqr(a:integer, var b:integer);  
begin  
    b:=a*a  
end
```

```
var i:integer;  
sqr(25, i);  
writeln(i);
```

```
proc {Sqr A ?B}  
    B=A*A  
end
```

```
local I in  
    {Sqr 25 I}  
    {Browse I}  
end
```

- The variable passed as an argument can be changed inside the procedure with visible effects outside after the call.
- The **B** inside **Sqr** is a synonym (an *alias*) of the **I** outside.
- The default mechanism in Oz is *call by reference*.

Call by variable

```
procedure sqr(var a:integer);  
begin  
    a:=a*a  
end
```

```
var i:integer;  
i:=25;  
sqr(i);  
writeln(i);
```

```
proc {Sqr A}  
    A:=@A*@A  
end
```

```
local I = {NewCell 0} in  
    I := 25  
    {Sqr I}  
    {Browse @I}  
end
```

- Special case of *call by reference*.
- The identity of the cell is passed to the procedure.
- The **A** inside **Sqr** is a synonym (an alias) of the **I** outside.

Call by value

```
procedure sqr(a:integer);  
begin  
    a:=a+1;  
    writeln(a*a)  
end  
var i:integer;  
i:=25;  
sqr(i);  
writeln(i);
```

```
proc {Sqr A}  
    C = {NewCell A}  
in  
    C := @C + 1  
    {Browse @C*@C}  
end  
  
{Sqr 25}
```

- A value is passed to the procedure. Any changes to the value inside the procedure are purely local, and therefore, **not** visible outside.
- The local cell **C** is initialized with the argument **A** of **Sqr**.
- Java uses call by value for both primitive values and object references.
- SALSA uses call by value in both local and remote message sending.

Call by value-result

```
procedure sqr_inc(inout a:integer);  
begin  
    a:=a*a  
    a:= a+1  
end
```

```
var i:integer;  
i:=25;  
sqr_inc(i);  
writeln(i);
```

```
proc {SqrInc A}  
    D = {NewCell @A}  
in  
    D := @D * @D  
    D := @D + 1  
    A := @D  
end  
local C = {NewCell 0} in  
    C := 25  
    {SqrInc C}  
    {Browse @C}  
end
```

- A modification of call by variable. Variable argument can be modified.
- There are two mutable variables: one inside **Sqr** (namely **D**) and one outside (namely **C**). Any *intermediate* changes to the variable inside the procedure are purely local, and therefore, **not** visible outside.
- **inout** is ADA terminology.

Call by name

```
procedure sqr(callbyname a:integer);  
begin  
    a:=a*a  
end
```

```
var i:integer;  
i:=25;  
sqr(i);  
writeln(i);
```

```
proc {Sqr A}  
    {A} := @{A} * @{A}  
end
```

```
local C = {NewCell 0} in  
    C := 25  
    {Sqr fun {$} C end}  
    {Browse @C}  
end
```

- Call by name creates a function for each argument (a *thunk*). Calling the function evaluates and returns the argument. Each time the argument is needed inside the procedure, the thunk is called.
- Thunks were originally invented for Algol 60.

Call by need

```
procedure sqr(callbyneed a:integer);  
begin  
    a:=a*a  
end
```

```
var i:integer;  
i:=25;  
sqr(i);  
writeln(i);
```

```
proc {Sqr A}  
    B = {A} % only if argument used!!  
in  
    B := @B * @B  
end
```

```
local C = {NewCell 0} in  
    C := 25  
    {Sqr fun {$} C end}  
    {Browse @C}  
end
```

- A modification of *call by name*. The thunk is evaluated **at most** once. The result is stored and used for subsequent evaluations.
- *Call by need* is the same as lazy evaluation. Haskell uses lazy evaluation.
- *Call by name* is lazy evaluation without memoization.

Which one is *right* or *best*?

- It can be argued that *call by reference* is the most primitive.
 - Indeed, we have coded different parameter passing styles using *call by reference* and a combination of cells and procedure values.
 - Arguably, *call by value* (along with cells and procedure values) is just as general. E.g., the example given for *call by variable* would also work in a *call by value* primitive mode. Exercise: Why?
- When designing a language, the question is: for which mechanism(s) to provide linguistic abstractions?
 - It largely depends on intended language use, e.g., *call by name* and *call by need* are integral to programming languages with lazy evaluation (e.g., Haskell and Miranda.)
 - For concurrent programs, *call by value-result* can be very useful (e.g. Ada.)
 - For distributed programs, *call by value* is best due to state encapsulation (e.g., SALSA).

More parameter passing styles

- Some languages for distributed computing have support for *call-by-move*.
 - Arguments to remote procedure calls are temporarily migrated to the remote location for the time of the remote procedure execution (e.g., Emerald).
 - A dual approach is to migrate the object whose method is to be invoked to the client side before method invocation (e.g., Oz).
- Java Remote Method Invocation (RMI) dynamically determines mechanism to use depending on argument types:
 - It uses *call by reference* in remote procedure calls, if and only if, arguments implement a special (**Remote**) interface
 - Otherwise, arguments are passed using *call by value*.
 - => Semantics of method invocation is different for local and remote method invocations!!
 - There is no language support for object migration in Java (as there is in other languages, e.g., SALSA, Oz, Emerald), so *call by move* is not possible.

Lazy evaluation

- The default functions in Oz are evaluated *eagerly* (as soon as they are called)
- Another way is lazy evaluation where a computation is done only when the result is needed

- Calculates the infinite list:
0 | 1 | 2 | 3 | ...

```
declare  
fun lazy {Ints N}  
  N|{Ints N+1}  
end
```

Lazy evaluation (2)

- Write a function that computes as many rows of Pascal's triangle as needed
- We do not know how many beforehand
- A function is *lazy* if it is evaluated only when its result is needed
- The function `PascalList` is evaluated when needed

```
fun lazy {PascalList Row}
  Row | {PascalList
        {AddList
         Row
         {ShiftRight Row}}}}
end
```

Lazy evaluation (3)

- Lazy evaluation will avoid redoing work if you decide first you need the 10th row and later the 11th row
- The function continues where it left off

`declare`

```
L = {PascalList [1]}  
{Browse L}  
{Browse L.1}  
{Browse L.2.1}
```

```
L<Future>  
[1]  
[1 1]
```

Lazy execution

- Without laziness, the execution order of each thread follows textual order, i.e., when a statement comes as the first in a sequence it will execute, whether or not its results are needed later
- This execution scheme is called *eager execution*, or *supply-driven* execution
- Another execution order is that a statement is executed only if its results are needed somewhere in the program
- This scheme is called *lazy evaluation*, or *demand-driven* evaluation (some languages use lazy evaluation by default, e.g., Haskell)

Example

$B = \{F1\ X\}$

$C = \{F2\ Y\}$

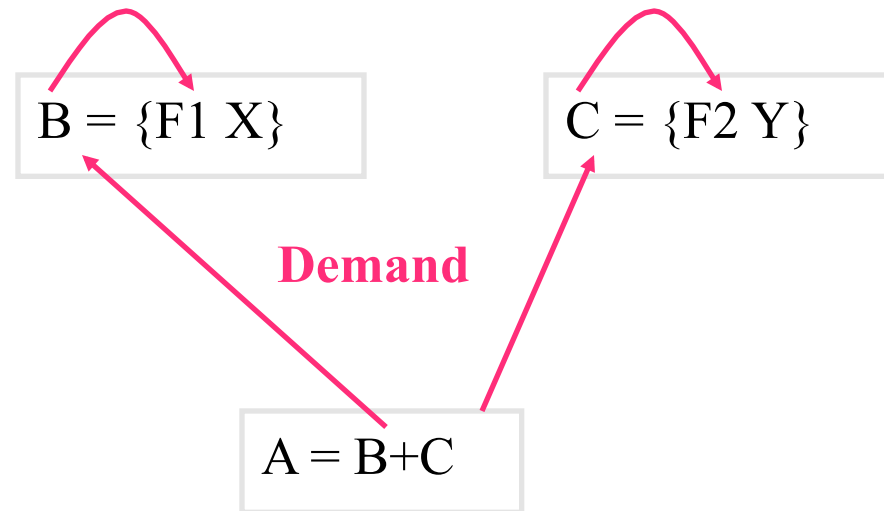
$D = \{F3\ Z\}$

$A = B+C$

- Assume F1, F2 and F3 are lazy functions
- $B = \{F1\ X\}$ and $C = \{F2\ Y\}$ are executed only if and when their results are needed in $A = B+C$
- $D = \{F3\ Z\}$ is not executed since it is not needed

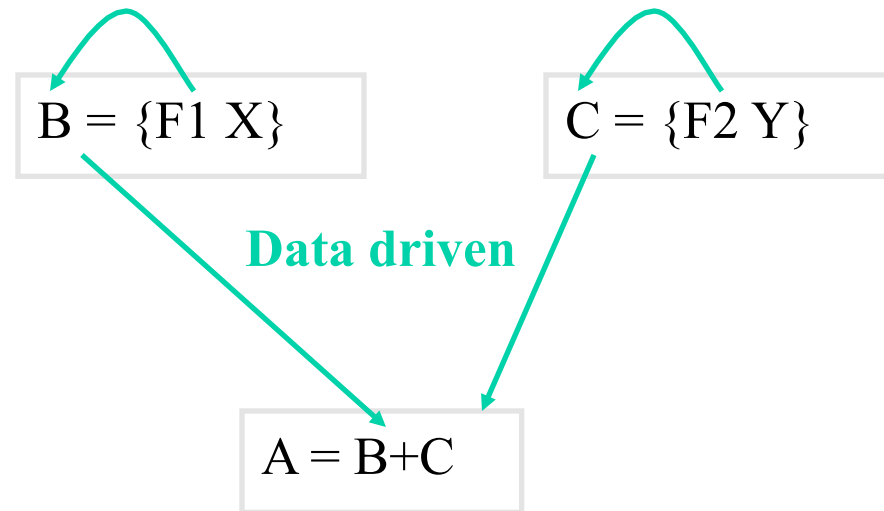
Example

- In lazy execution, an operation suspends until its result is needed
- The suspended operation is triggered when another operation needs the value for its arguments
- In general multiple suspended operations could start concurrently



Example II

- In data-driven execution, an operation suspends until the values of its arguments results are available
- In general the suspended computation could start concurrently



Using Lazy Streams

```
fun {Sum Xs A Limit}  
  if Limit>0 then  
    case Xs of X|Xr then  
      {Sum Xr A+X Limit-1}  
    end  
  else A end  
end
```

```
local Xs S in  
  Xs={Ints 0}  
  S={Sum Xs 0 1500}  
  {Browse S}  
end
```

How does it work?

```
fun {Sum Xs A Limit}
  if Limit>0 then
    case Xs of X|Xr then
      {Sum Xr A+X Limit-1}
    end
  else A end
end
```

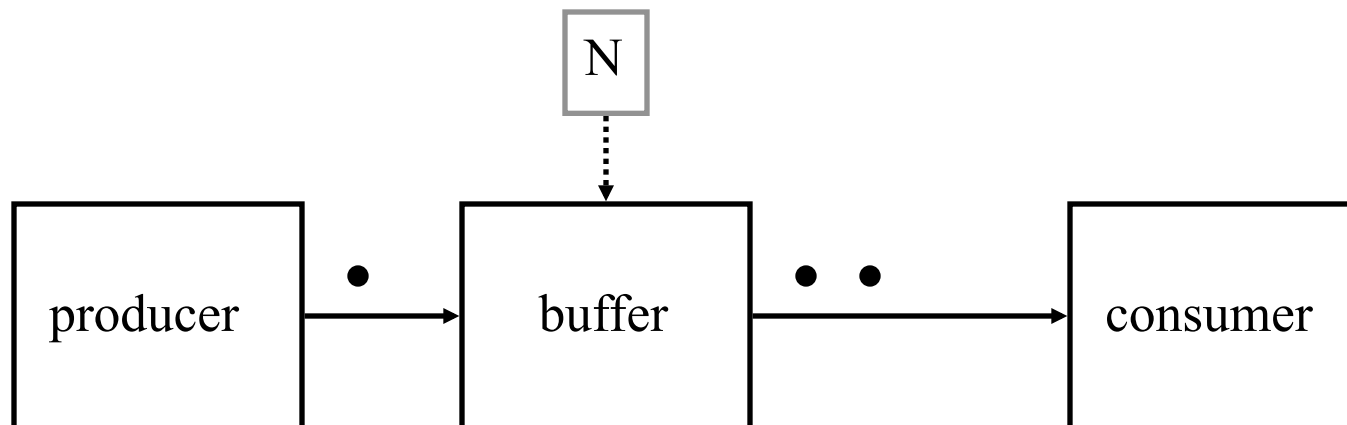
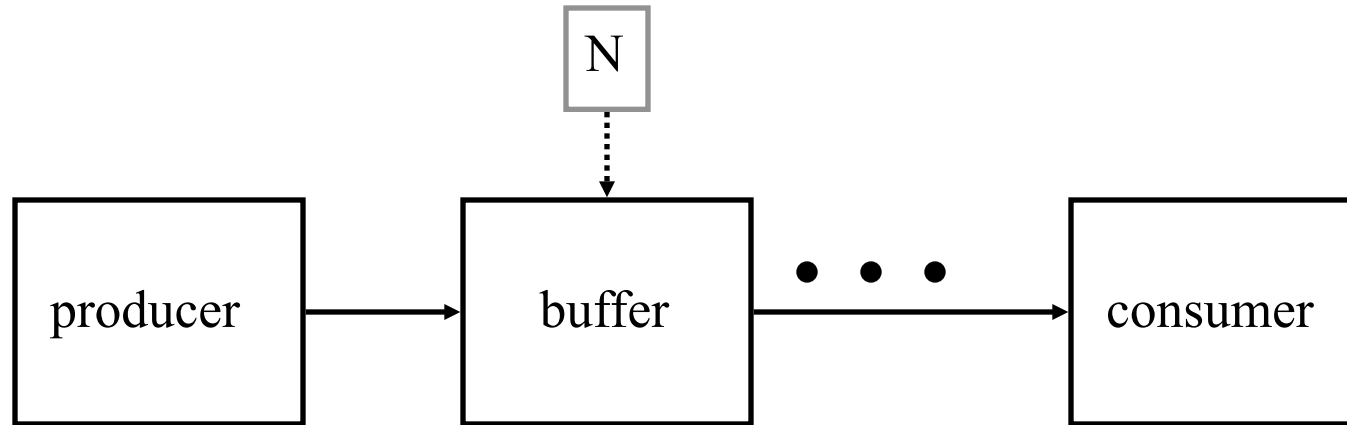
```
fun lazy {Ints N}
  N | {Ints N+1}
end

local Xs S in
  Xs = {Ints 0}
  S={Sum Xs 0 1500}
  {Browse S}
end
```

Improving throughput

- Use a lazy buffer
- It takes a lazy input stream `In` and an integer `N`, and returns a lazy output stream `Out`
- When it is first called, it first fills itself with `N` elements by asking the producer
- The buffer now has `N` elements filled
- Whenever the consumer asks for an element, the buffer in turn asks the producer for another element

The buffer example



The buffer

```
fun {Buffer1 In N}  
  End={List.drop In N}  
  
  fun lazy {Loop In End}  
    In.1|{Loop In.2 End.2}  
  end  
  
in  
  {Loop In End}  
end
```

Traversing the In stream, forces the producer to emit N elements

The buffer II

```
fun {Buffer2 In N}
  End = thread
    {List.drop In N}
  end
  fun lazy {Loop In End}
    In.1|{Loop In.2 End.2}
  end
in
  {Loop In End}
end
```

Traversing the In stream, forces the producer to emit N elements and at the same time serves the consumer

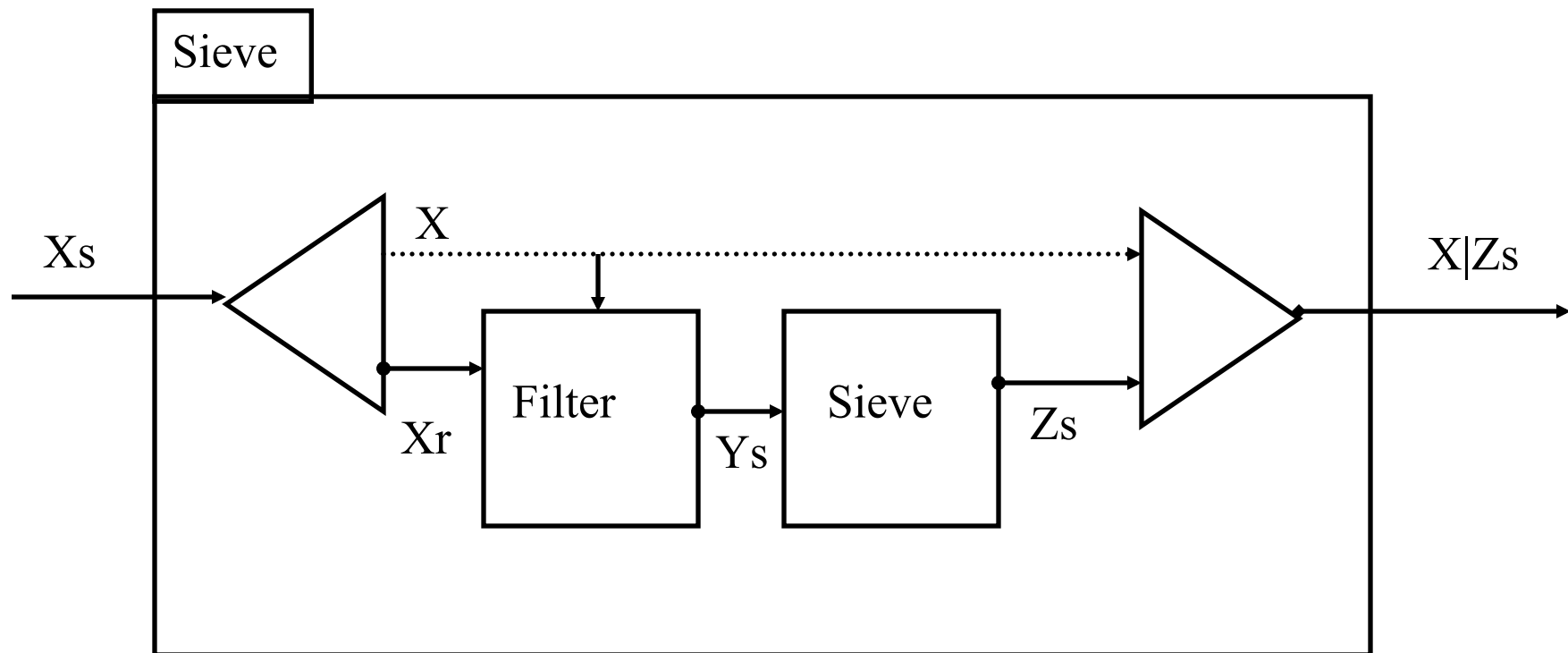
The buffer III

```
fun {Buffer3 In N}
  End = thread
    {List.drop In N}
  end
  fun lazy {Loop In End}
    E2 = thread End.2 end
    In.1|{Loop In.2 E2}
  end
in
  {Loop In End}
end
```

Traverse the In stream, forces the producer to emit N elements and at the same time serves the consumer, and requests the next element ahead

Larger Example: The Sieve of Eratosthenes

- Produces prime numbers
- It takes a stream $2\dots N$, peels off 2 from the rest of the stream
- Delivers the rest to the next sieve



Lazy Sieve

```
fun lazy {Sieve Xs}
  X|Xr = Xs in
  X | {Sieve {LFilter
    Xr
    fun {$ Y} Y mod X \= 0 end
  }}
end

fun {Primes} {Sieve {Ints 2}} end
```

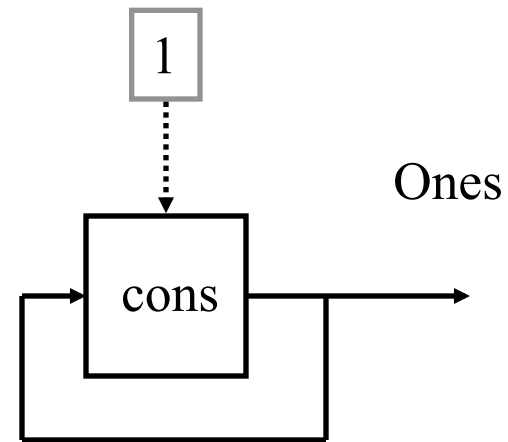
Lazy Filter

For the Sieve program we need a lazy filter

```
fun lazy {LFilter Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    if {F X} then X|{LFilter Xr F} else {LFilter Xr F} end
  end
end
```

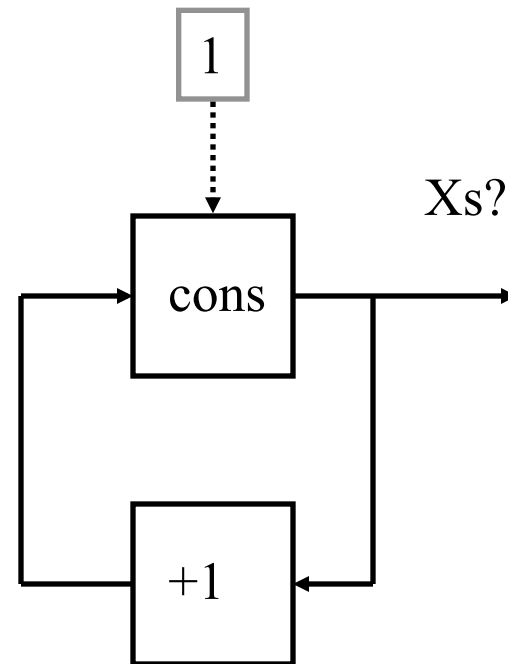
Define streams implicitly

- $\text{Ones} = 1 \mid \text{Ones}$
- Infinite stream of ones



Define streams implicitly

- $Xs = 1 | \{LMap\ Xs$
 `fun {$ X} X+1 end`
- What is Xs ?



The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with $a, b, c \geq 0$ (in ascending order)

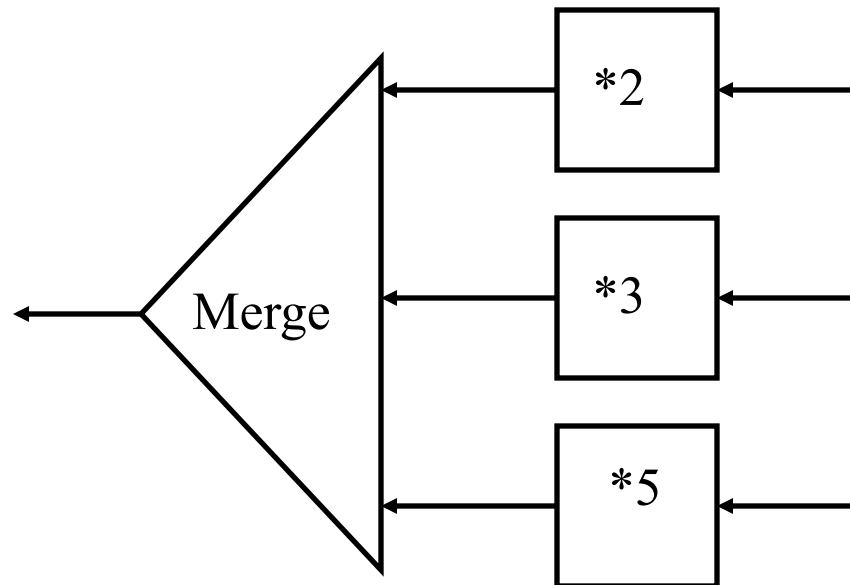
*2

*3

*5

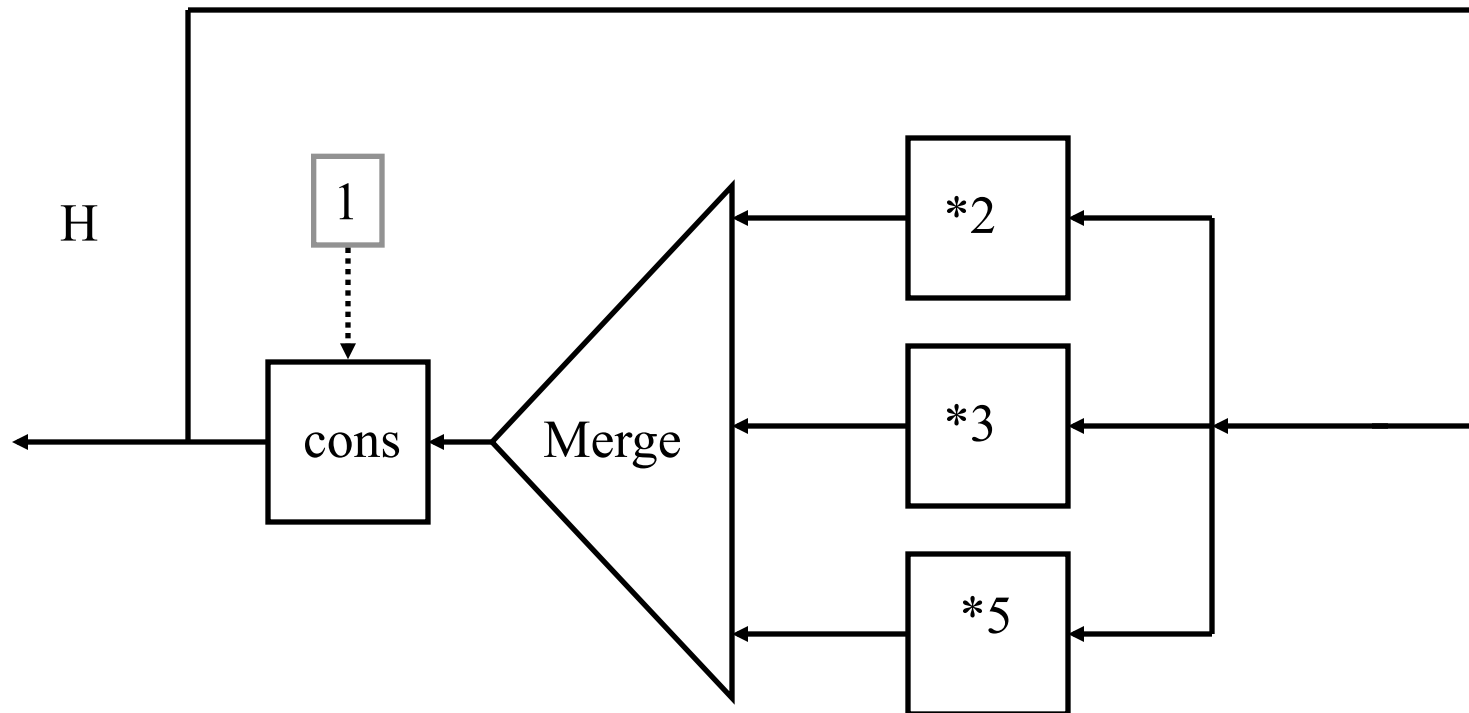
The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with $a, b, c \geq 0$ (in ascending order)



The Hamming problem

- Generate the first N elements of stream of integers of the form: $2^a 3^b 5^c$ with $a, b, c \geq 0$ (in ascending order)



Lazy File Reading

```
fun {ToList FO}
  fun lazy {LRead} L T in
    if {File.readBlock FO L T} then
      T = {LRead}
    else T = nil {File.close FO} end
  L
end
{LRead}
end
```

- This avoids reading the whole file in memory

List Comprehensions

- Abstraction provided in lazy functional languages that allows writing higher level set-like expressions
- In our context we produce lazy lists instead of sets
- The mathematical set expression
 - $\{x*y \mid 1 \leq x \leq 10, 1 \leq y \leq x\}$
- Equivalent List comprehension expression is
 - $[X*Y \mid X = 1..10 ; Y = 1..X]$
- Example:
 - $[1*1 \ 2*1 \ 2*2 \ 3*1 \ 3*2 \ 3*3 \ \dots \ 10*10]$

List Comprehensions

- The general form is
- $[f(x,y, \dots,z) \mid x \leftarrow \text{gen}(a_1, \dots, a_n) ; \text{guard}(x, \dots)$
 $y \leftarrow \text{gen}(x, a_1, \dots, a_n) ; \text{guard}(y, x, \dots)$

]
- No linguistic support in Mozart/Oz, but can be easily expressed

Example 1

- $z = [x\#x \mid x \leftarrow \text{from}(1,10)]$
- $Z = \{\text{LMap } \{\text{LFrom } 1 \ 10\} \text{ fun } \{\$ X\} X\#X \text{ end}\}$
- $z = [x\#y \mid x \leftarrow \text{from}(1,10), y \leftarrow \text{from}(1,x)]$
- $Z = \{\text{LFlatten}$
 $\{\text{LMap } \{\text{LFrom } 1 \ 10\}$
 $\text{fun } \{\$ X\} \{\text{LMap } \{\text{LFrom } 1 \ X\}$
 $\text{fun } \{\$ Y\} X\#Y \text{ end}$
 $\}$
 end
 $\}$
 $\}$

Example 2

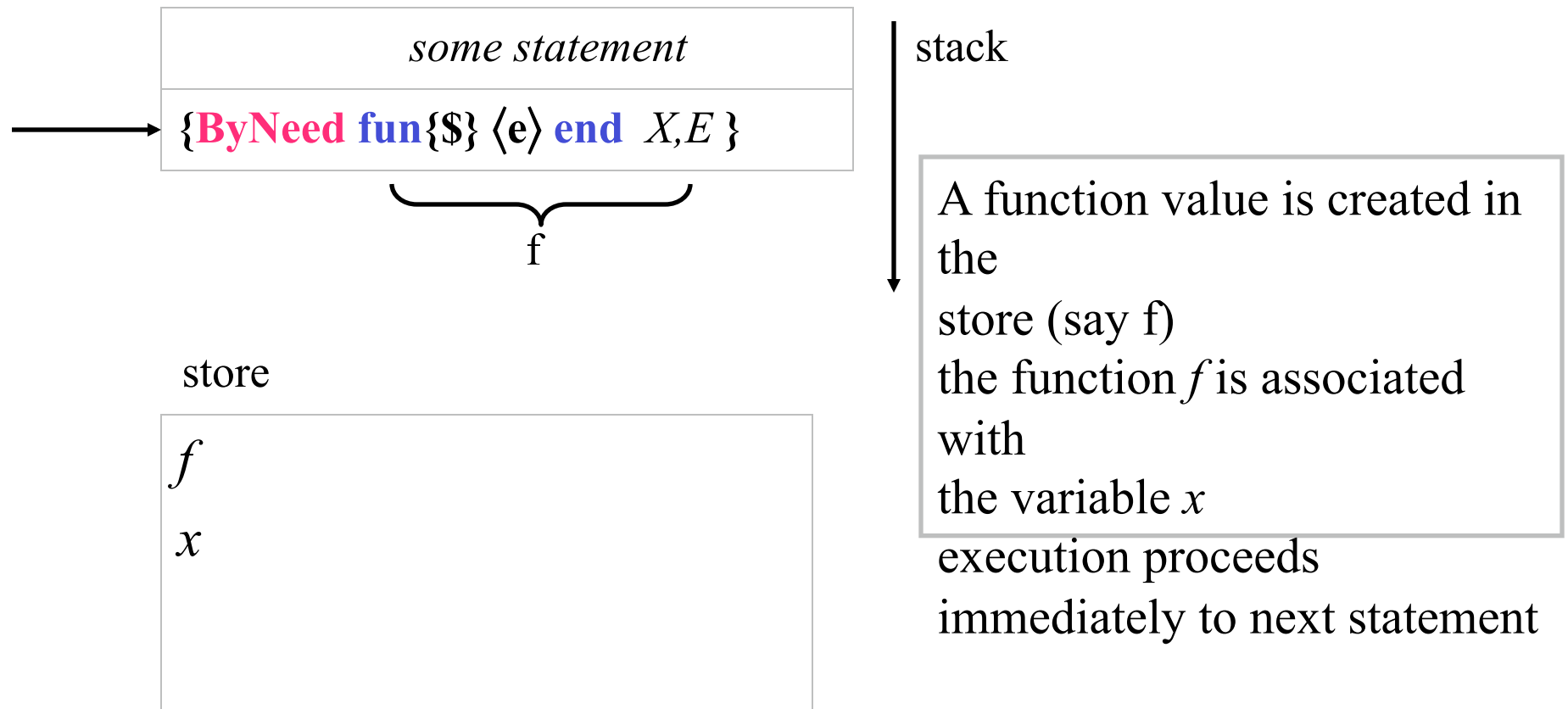
- $z = [x\#y \mid x \leftarrow \text{from}(1,10), y \leftarrow \text{from}(1,x), x+y \leq 10]$
- $Z = \{ \text{LFilter} \{ \text{LFlatten} \{ \text{LMap} \{ \text{LFrom } 1 \ 10 \} \text{ fun } \{ \$ X \} \{ \text{LMap} \{ \text{LFrom } 1 \ X \} \text{ fun } \{ \$ Y \} X\#Y \text{ end} \} \text{ end} \} \text{ fun } \{ \$ X\#Y \} X+Y \leq 10 \text{ end} \} \}$

Implementation of lazy execution

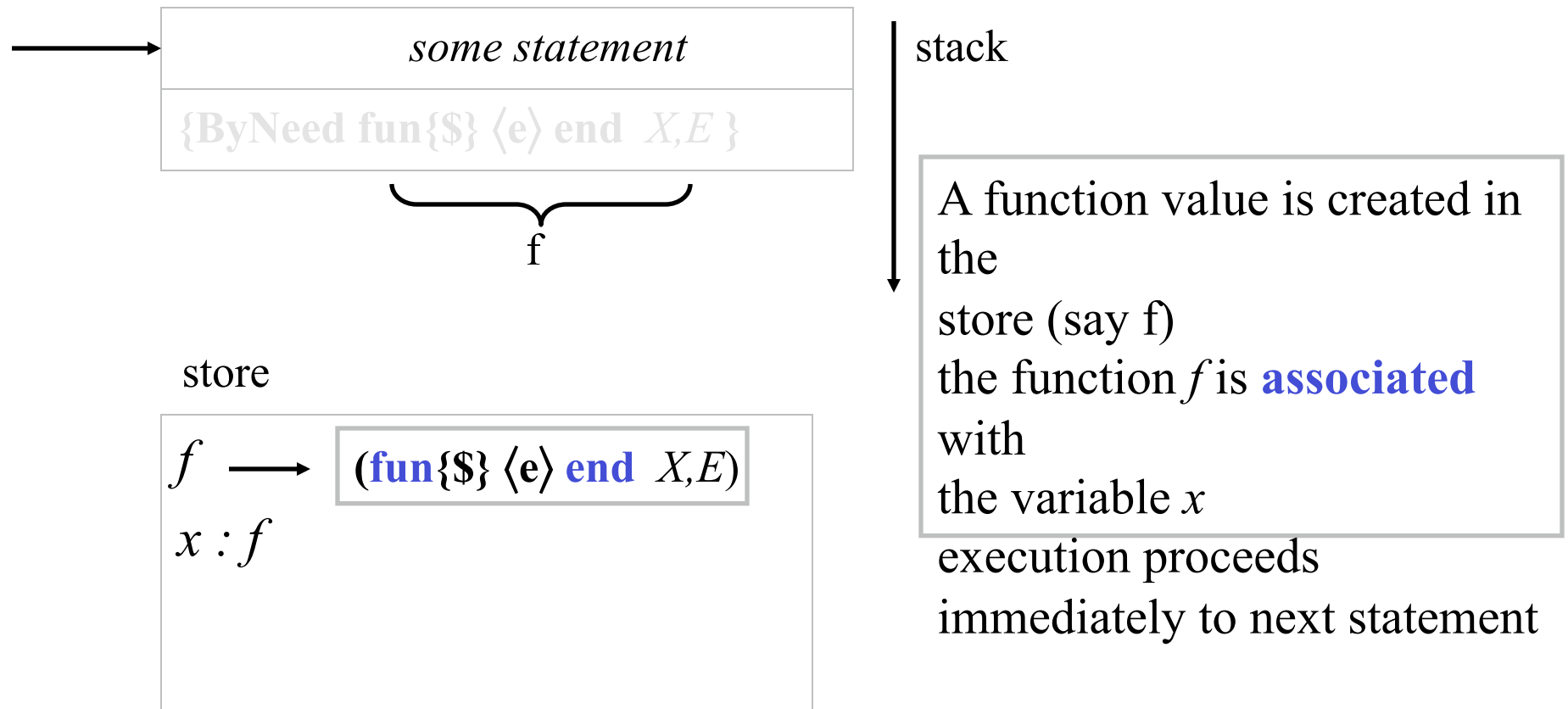
The following defines the syntax of a statement, $\langle s \rangle$ denotes a statement

$$\begin{array}{l}
 \langle s \rangle ::= \text{skip} \quad \text{empty statement} \\
 | \quad \dots \\
 | \text{thread } \langle s_1 \rangle \text{ end} \quad \text{thread creation} \\
 | \text{\{ByNeed fun\{ \$ \} } \langle e \rangle \text{ end } \langle x \rangle \} \quad \text{by need statement} \\
 \qquad \underbrace{\hspace{10em}}_{\text{zero arity}} \quad \underbrace{\hspace{2em}}_{\text{variable}} \\
 \qquad \text{function}
 \end{array}$$

Implementation



Implementation



Accessing the ByNeed variable

- $X = \{\text{ByNeed } \text{fun}\{\$ \} \ 111 * 111 \ \text{end}\}$ (by thread T0)
- Access by some thread T1
 - if $X > 1000$ then $\{\text{Browse hello}\#X\}$ end

or

- $\{\text{Wait } X\}$
- Causes X to be bound to 12321 (i.e. $111 * 111$)

Implementation

Thread T1

1. X is needed
2. start a thread T2 to execute F (the function)
3. only T2 is allowed to bind X

Thread T2

1. Evaluate $Y = \{F\}$
2. Bind X the value Y
3. Terminate T2

4. Allow access on X

Lazy functions

```
fun lazy {Ints N}  
  N | {Ints N+1}  
end
```



```
fun {Ints N}  
  fun {F} N | {Ints N+1} end  
in {ByNeed F}  
end
```

Exercises

90. VRH Exercise 6.10.2 (page 482).
91. Explain why the *call by variable* example given would also work over a *call by value* primitive parameter passing mechanism. Give an example for which this is not the case.
92. Explain why *call by need* cannot always be encoded as shown in the given example by producing a counter-example. (Hint: recall the difference between normal order evaluation and applicative order evaluation in termination of lambda calculus expression evaluations.)
93. Create a program in which *call by name* and *call by need* parameter passing styles result in different outputs.
94. Can type inference always deduce the type of an expression?
 - If not, give a counter-example. How would you design a language to help it statically infer types for non-trivial expressions?

Exercises

95. Write a lazy append list operation `LazyAppend`. Can you also write `LazyFoldL`? Why or why not?
96. Exercise VRH 4.11.10 (pg 341)
97. Exercise VRH 4.11.13 (pg 342)
98. Exercise VRH 4.11.17 (pg 342)