Operational Semantics for A Declarative Sequential Kernel Language

top

Table of Contents

1 Semantics - Synopsis

should be defined in a simple mathematical model helps us reason abt correctness, execution time, memory usage etc.

4 widely used approaches.

Model	Description	Suitable for	Example
Operational	how to execute	Imperative	
	a statement in terms of an	Declarative	OZ
	abstract machine	Logical	

Model	Description	Suitable for	Example
Axiomatic	Statement = **relation**	statement	Hoare
	b/w input and output states	sequence	Logic
	relation is a logical	stateful	
		models	
Denotational	statement = **function** over	Declarative	
	an abstract domain		
Logical	statement = **model** for		
	logical theorems	Declarative	
		Relational	

Two step approach: (1) Translate every program into an equivalent one in a simple core language called the**kernel language**. (2) Define *operational semantics* for the kernel language.

Other approaches: Translator Approach (1) Foundational Calculus approach: translate into a mathematical calculus. Example, the lambda calculus, first-order logic, or pi-calculus. (2) Translation directly into instructions on an abstract machine, or a virtual machine.

Interpreter Approach o Language Semantics is defined by an interpreter. o Linguistic Abstraction == extending the interpreter. o Interpreter is a program in L1 that accepts programs in L2 and executes them. o Metacircular Evaluator : L1 = L2

- Self-contained implementation of linguistic abstractions
- does not preserve execution-time complexity
- basic concepts interact inside the interpreter.

---

2 Single-Assignment Store

set of variables that are initially uninitialized. can be assigned to one value They are called **declarative variables**(dataflow variables).

A declarative variable is indistinguishable from its value.

Value Store: persistent mapping of variables to values.

- We can compute with partial values. e.g. Bind an unbound argument as the output of a procedure

2.1 Value Creation (Binding)

```
x = 314
```

(1) Construct value in the store (2) Bind x to this value.

If x is already bound, '=' tests whether the operands are compatible. Incompatible operands cause an exception.

2.2 Variable Identifier

textual name that refers to a store entity from outside the store.

Environment `{<X> -> x}`

A variable identifier can refer to a **value** or another **bound variable**.

Following the links of bound variables is called **dereferencing**. This is invisible to the programmer.

2.3 Partial Values

Data structure that may contain unbound variables.

2.4 Variable-to-variable binding

```
X = Y              =>  forms an equivalence class in the store
X = [1 2 3]            {x1, x2}
```

We can even have circular chain of references:

```
X = [1 2 X]
```

2.5 Use before binding

Ways of handling:

1. No error in execution: Garbage Output. e.g. C/C++
2. No error in execution, initialized to default value.
3. Exception in Runtime : Java
4. Compiler Error
5. Wait until possibly another path binds the variable.

(3) and (4) are good strategies for sequential programs. (5) is a good strategy for concurrent programs. Could it be bad for sequential programs? top

---

3 Kernel Language

3.1 Backus Naur Form (Above Kernel Language)

3.2 Language

Let <x> and <y> denote any variable identifier, <s> any statement, <v> any value, and <pattern> some pattern.

Statement	Description
s::=	
skip	empty statement
<s>1 <s>2	Statement sequence
local <x> in <s> end	Variable creation
<x>1 = <x>2	Variable-Variable binding

Statement	Description
<x> = <v>	Value creation
if <x>	Conditional
then <s>1	
else <s>2 end	
case <x>	Pattern matching
of <pattern> then <s>1	
else <s>2 end	
{<x> <y>1 … <y>m}	Procedure application

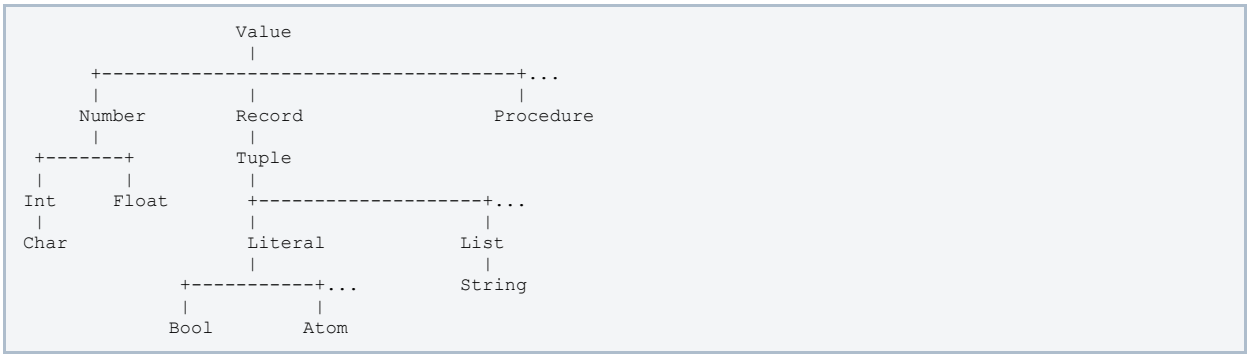
The following are the value expressions in the language (the <v> above.)

Nonterminal	Expression	Comment
<v>	<number> OR	Note that procedures
	<record> OR	are values
	<procedure>	
<number>	<int> OR	Integers are
	<float>	arbitrarily large
		Floats are of
		arbitrary precision
<record>, <pattern>	<literal> OR	nil, 'fine day'
	<literal>(<feature>1: <x>1	are patterns
	…	
	<feature>n: <x>n)	
<procedure>	proc{$ <x>1 … <x>m}	Procedure values are
	<s> end	called **closures**
<literal>	<atom> OR <bool>	
<feature>	<atom> OR <bool> OR <int>	integers are default
		features if labels are
		omitted.

Nonterminal	Expression	Comment
<bool>	true OR false	

A type is a value together with operations defined on it.

The system has a well-defined set of types called the basic types. For example, the following is a fragment of the hierarchy of the basic types. This hierarchy is ordered by set inclusion (for example, every list is a tuple)



Abstract Data Types: User-defined types which are not basic.

Oz is dynamically typed. Type errors in the (basic) declarative model cause immediate termination. If we extend the model with exceptions, type errors can be handled within the system.

Notes on basic types: Numbers: Unary minus is written with a tilde prefix: e.g. ~10 Strings: A string "ex" is a list of character codes [101 121] Procedures: <x> = proc {$ <y>1 … <y>m} <s> end has the syntactic shortcut proc {<x> <y>1 … <y>m} <s> end

This has two operations which are distinct in the elaborate version: (1) creating the procedure value (2) Binding it to the identifier <x>

Operations (on basic types)

Operation	Description	Argument Type
A==B		Value
A\=B	Not Equals	"
{IsProcedure B}		"
A=<B	!!!!	Number or Atom
A<B		"
A>=B	!!!!	"
A>B		"
A+B		Number
A-B		"
A*B		"
A div B	Integer Division	Int

Operation	Description	Argument Type
A mod B	Modulo	Int
A/B	Integer Division	Float
{Arity R}	Arity	Record
{Label R}		"
R.F	Field selection	"

top

---

4 Rationale

Can the kernel language be further reduced while keeping the expressivity unchanged?

Why are some of the above features considered basic?

4.1 Records

basic way to structure data. easy to compose, and decompose. decompose using patterns several utility methods: to find arity, to select a field etc.

Building block for (1) Object-Oriented Programming (see PlanePoint.oz) (2) Component-based Programming

4.2 Procedures

Why not objects? or functions?

Functions in Oz return a value.

Procedures can define entities that are not necessarily like mathematical functions. top

---

5 Kernel Language Semantics

Consists in evaluating functions over partial values.

Operational Semantics with respect to an abstract machine.

5.1 Basic Concepts

5.1.1 Simple execution

```
local A B in                :  create 2 variables in the store, make
                               A, B point to them
  A = 11                    :  Bind A to 11.
  B = A + A                 :  Add A to itself, bind the result to B
end
```

5.1.2 Variable identifiers and Static Scoping

```
local X in
  X = 1
  local X in
    X = 2
    {Browse X}              : value of an identifier is determined by
  end                         its innermost local declaration.
  {Browse X}                  Can be determined from the *text* of
end                           the program.
```

How else can this be done? What could be dynamic scoping?

5.1.3 Procedures

Parameters are passed by reference. Does not return a value. Produces effects by binding its unbound arguments.

```
proc {Max X Y ?Z}                 ? is an annotation for o/p [No effect]
  if X>Y then Z=X else Z=Y end  : Z is unbound when input, bound inside
end                              the procedure.
proc {LB X ?Z}
  if X>Y then Z=X else Z=Y end  : Y is a free variable.
end
```

Free variables take the values they have in the **text**, when the procedure is **defined** (and not the value it has when it is invoked).

```
local Y in
  Y=0
  proc {LB X ?Z}
    if X>Y then Z=X else Z=Y end  : Y is 0, and not 15 (Static scoping)
  end
  local Y = 15 Z in
    {LB 5 Z}
  end
end
```

When could dynamic scoping be useful? e.g. locally configurable parameters deeply passed-on arguments.

```
 e.g.     fn(Z,X1) : X1's value is used here
          ...
          f2(Y,X1)
          f1(X,X1) : X1's value is determined here
/can be simplified with dynamic scoping
```

When could dynamic scoping be harmful? Harder to reason about programs from the text! Does not support**modular** reasoning. A procedure that was correct may behave incorrectly in other contexts.

5.1.4 Procedural Abstraction

Any statement can be made into the body of a procedure. The variables used in the statement can be made either into formal parameters of the procedure, or into free variables. The free variables are scoped statically.

5.1.5 Dataflow behavior

In a single assignment store, variables can be unbound.

When a statement needs a variable which is as yet unbound, it waits till the object is bound. (This could happen in a different thread of execution.) This is called dataflow behavior.

Dataflow behavior enables concurrency

Can be implemented in the abstract machine in a simple way.

5.2 Abstract Machine

5.2.1 Definitions

A **single assignment store** \s: a set of variables partitioned into (1) Equal but unbound variables (e.g. after a variable-to-variable assignment) (2) Variables bound to a a value - number, record or procedure.

```
e.g. {x1, x2=x3, x1=a|x2}
```

A stored variable bound to a value is indistiguishable from that value.

An **environment** E: mapping from variable identifiers to entities in the single assignment store.

```
e.g. E is { X->a, Y-y}
```

A **semantic statement** is a pair (<s>,E) where

<s> is a statement

E is an environment

Semantic statement *relates* the statement to what it references (many-to-many)

```
e.g. (local X in X=10 end, {X->x})
```

An **execution state** is a pair (ST, \s) where ST is a semantic stack: a stack of semantic statements \s is a single-assignment store.

Semantic Stack	Single-Assignment Store
(local X in X=10 end, {X->x})	x->10

A **computation** is a sequence of execution states starting from the initial.

A single transition in a computation is called a **computation step**. A computation step is atomic, there are no visible intermediate states.

Questions:

(A) Why is an environment associated with every statement? Why not just have a global store?

(B) Why do we separate the identifiers and the value store variables?

5.2.2 Program Execution

A program is a statement <s>.

The initial execution state s

(<s>, O)	O

o At each step, the first element of ST is popped and execution proceeds according to the form of the statement.

o The final execution state, if the program terminates, is one in which the semantic state is empty.

A semantic stack can be in

- Runnable State

- Terminated State
- Suspended state.

5.2.3 Calculating with Environments

E(<x>): entity associated with variable <x> in the store.

Adjunction: E+{<x>->x} overrides any existing mapping of <x> e.g. in lexical scoping, not reassignment.

Restriction: defines a subdomain of an existing one. e.g. $E|_{<x>1, \ldots, <x>n}$ used in defining closures.

5.3 Nonsuspendable statements

Statement at	
the top of the	**Operational Semantic Actions**
semantic stack	
(skip, E)	execution is complete after popping this
	statement from the semantic stack.
($<s_1>$ $<s_2>$, E)	(1) Push ($<s_2>$;,E) on to the stack.
	(2) Push ($<s_1>$;,E) on to the stack.
	Q: Why doesn't the environment change?
(local <x>	(1) Create a new variable x in the store
in <s> end,E)	(2) E' = E + {<x> -> x}
	(3) Push (<s>,E')
($<x_1>$;=$<x_2>$;,E)	Bind E($<x_1>$;) to E($<x_2>$;) in the store.
(<x>=<v>,E)	(1) Create a new variable x in the store.
	(2) Construct the value represented by <v>
	(3) Let <x> refer to it.
	(4) All identifiers in <v> are replaced by
	contents as given by E.
	(5) Bind E(<x>) and <x> in the store.

In the last case, the values can be numbers, records or procedures.

N.B. Procedure values are also called **closures**.

The procedure body <s> can have free variable identifiers. formal parameters external references

The procedure value is a pair (proc {$ <y>1 … <y>m} <s> end, CE) where CE is $E|_{<z>1 \ldots <z>k}$ is the**contextual environment**.

5.4 Suspendable statements.

If a variable <x> is unbound, the execution is suspended until **activation** : E(<x>) must be determined.

In the declarative sequential model, if an execution is suspended, it will never continue.

```
| Statement at the top of semantic | Operational Semantic action           |
| stack                            |                                       |
|----------------------------------+---------------------------------------|
| (if <x> then <s>1 else <s>2 end, |  If E(<x>) is determined              |
| E)                               |   If E(<x>) is not a boolean          |
|                                  |      raise an error condition         |
|                                  |   If E(<x>) is true                   |
|                                  |      push(<s>1, E)                    |
|                                  |   Else                                |
|                                  |      push(<s>2, E)                    |
|                                  |  Else
|                                  |     Suspend execution                 |
|----------------------------------+---------------------------------------|
| ({<x> <y>1 ... <y>n},            | If E(<x>) is determined               |
| E)                               | If E(<x>) not a procedure value       |
|                                  |  or does not have n arguments         |
|                                  |  raise an error condition             |
|                                  | If E(<x>)::(proc {$ <z>1   <z>n}      |
|                                  |             <s> end, CE)              |
|                                  |  push(<s>, CE+{<z>1 -> E(<y>1),       |
|                                  |             ..., <z>n -> E(<y>n)})    |
|                                  | Else                                  |
|                                  |  Suspend execution                    |
|----------------------------------+---------------------------------------|
| (case <x> of <lit>(<f>1:<x>1     | If E(<x>) is determined               |
| ... <f>n = <x>n)                 |  If the label of E(<x>) is <lit> and  |
| then <s>1 else <s>2 end, E)      |   its arity is [<f>1   <f>n]          |
|                                  |     push(<s>1, E+{<x>1 -> E(<x>).f1,  |
|                                  |                ..., <x>n = E(<x>).fn}) |
|                                  |  Else                                 |
|                                  |    push(<s>2, E)                      |
|----------------------------------+---------------------------------------|
```

top

6 Execution Examples

6.1 Variable Identifiers and Static Scoping

```
--------------------------
local X in          ------+
  X=1                     |
  local X in        --+   <s>
    X=2               |   |
    {Browse X}      <s>1  |
  end               --+   |
  {Browse Y}        <s>2  |
end                 ------+
--------------------------
```

Sequence of Execution states.

1. The initial state of the semantic stack and the Store are

```
2.  (s,emptyset)    emptyset
```

3. After executing the local statement, and binding X=1

```
4.  (<s>1 <s>2 , {X->x})         {x->1}
```

5. After executing the sequential composition

```
6.  (<s>1, {X->x})               {x->1}
7.  (<s>2, {X->x})
```

8. Executing local statement in <s>1

```
9.  (X=2 {Browse X}, {X->x'})    {x', x->1}
10. (<s>2, {X->x})
```

11. After the binding X=2

```
12. ({Browse X}, {X->x'})        {x'->2,
13. ({Browse X}, {X->x})          x->1}
```

6.2 Procedure Definition and Calls

First, a procedure with no external references in its body.

```
1. local Copy in
2.   local B in
3.       local A in
4.            Copy =   proc {$ X ?Y}
5.                        Y=X
6.                     end
7.            B = 1
8.            {Copy B A}
9.            skip
10.       end
11.   end
12. end
```

1. The initial execution state is

```
2.  (1-12:: emptyset)              emptyset
```

3. After executing the three local declarations,

```
4.  (4-6::{Copy->c,B->b,A->a})        {c,b,a}
5.  (7::{Copy->c, B->b, A->a})
6.  (8::{Copy->c, B->b, A->a})
7.  (9::{Copy->c, B->b, A->a})
```

8. After executing the two bindings,

```
9.  (8::{Copy->c, B->b, A->a})        {c->(proc...end,emptyset),
10. (9::{Copy->c, B->b, A->a})         b->1,
11.                                    a}
```

12. The procedure invocation involves the following step which binds the formal parameters X and Y to the**store variables** corresponding to the actual parameters A and B. This is the pass-by-value mechanism in the current semantics.

```
13. (5::{X->b, Y->a})                  {c->(proc...end,emptyset),
14. (9::{Copy->c, A->a, B->b})                 {b->1},
15.                                           a}
```

16. After executing the procedure application,

```
17. (9::{Copy->c, B->b, A->a})        {c->(proc...end,emptyset),
18.                                    {a,b}->1}
```

The environment is unchanged since there are no external references in the body of the procedure.

19. After executing `skip`, the statement stack is empty, and a is bound to 1.

top

Now, we consider a procedure with an external reference.

```
1. local Y in
2.      Y=2
3.      local CopyConstant in
4.                  local A in
5.                        CopyConstant = proc {$ B ?A}
6.                                          A=Y
7.                                       end
8.                        {CopyConstant 1 A}
9.          end
10.       end
```

```
11. end
```

The procedure value in this case is

```
(proc {$ B A} A = Y end,  {Y->y})
                          ======
```

where the store contains

`y->2`

Procedure application has the following change: the new environment is computed starting from the contextual environment.

```
(6-6::{Y->y, B->b, A->a})             {CopyConstant->proc...end,
                                       a,
                                       b->1,
                                       y->2}
```

After the statement is executed, the stack is empty, with a bound to 2.

top

7 Unification

7.1 Introduction

The algorithm behind the binding operator ('=') and the logical operators entailment ('==') and disentailment ('\=').

The unification <X>=<Y> tries to make the partial values <X> and <Y> "equal" by adding 0 or more bindings. "Equal" means that after successful unification, <X>==<Y> would be `true`.

Informally, unification works by "adding binding information to the store". We can clarify this by first considering some specific cases.

Consider some examples.

1. The binding `X=Y` adds the information that `X` and `Y` are equal.

   Suppose `X` and `Y` were already bound before this. Then some additional bindings may be added to the store, e.g. `X = r(1 2) Y=r(A B) X=Y` would bind `A` to 1 and `B` to 2 in the store.

2. Unification is symmetric: `2=X` and `X=2` would both bind 2 to `X` in the store. Similarly `r(A 2) = r(1 B)` would bind `A` to 1 and `B` to 2 in the store.
3. If two partial values are already equal, unification does nothing: e.g. `2=2`, `X=X`.
4. If the partial values are not compatible in type, then unification fails, producing a `failure` exception. e.g.`10=12`, `r(A 2) = r(1 A)`.

   Executing the `fail` statement causes the unification to fail.

5. **Cyclic structures** can be unified. e.g. `X = 1|X`.

   Unification may result in new cyclic structures being formed. e.g. `X = f(a:X b:_) Y=f(a:_ b:Y) X=Y` creates a cyclic structure `X = f(a:X b:X)` with 2 cycles.

7.2 Algorithm

The store σ is partitioned into

1. Sets of unbound variables which are equal. (Each such set is called an **equivalence set**). e.g. If X=Y, and X,Y,Z are unbound variables, then the store contains the two equivalence sets `{x,y}` and `{z}`.
2. Variables bound to values.

For the purpose of the algorithm that follows, note that a set refers always to the first case of a set of equivalent unbound variables.

7.2.1 Primitive Bind

The basic operation of unification. It binds all variables in an equivalence set.

bind (S, <v>)
: binds all variables in the equivalence set S to the number or record <v>. This function is typically called when an unbound variable x ∈ S is unified with <v>. e.g.

```
bind ( {x, y},  r(a:x) )
```

removes `x` and `y` from the equivalence set `{x,y}` and binds each of them separately to `r(a:x)`.

bind(S, T)
: Merges the equivalence sets S and T. This is typically called when an unbound variable x ∈ S is unified with some unbound variable y ∈ T.

7.2.2 Unify(x,y)

Let `x`, `y` be two store variables. The algorithm Unify(`x`, `y`) proceeds as follows. The cases are treated separately.

Both unbound variables
: If x ∈ S and y ∈ T, then bind(S,T). Note that T is not necessarily distinct from S.

One variable is bound to a value
: If x ∈ S and y is bound to a value, then bind(S,y). If y ∈ T and x is bound to a value, then bind(T,x). (Note that a bound variable is indistinguishable from its value.)

Incompatible Record types
: If `x` is bound to $r(f_1: x_1 \ldots f_n: x_n)$ and `y` is bound to $r'(f_1': y_1 \ldots f_m': y_m)$, where $r \neq r'$ or $\{f_1, f_2, \ldots, f_n\} \neq \{f_1', f_2', \ldots, f_m'\}$ then raise a `failure` exception.

Compatible Record types
: If `x` is bound to $r(f_1: x_1 \ldots f_n: x_n)$ and `y` is bound to $r(f_1: y_1 \ldots f_n: y_n)$, then

    1. Do for `i` from 1 to n
    2. Unify(x_i, y_i)

top

Date: 2012-08-29 16:57:59 India Standard Time

Author:

Org version 7.8.11 with Emacs version 24

Validate XHTML 1.0

Introduction to Functional Programming in Oz

Table of Contents

1 Techniques

This is a brief introduction to functional programming in Oz.

1.1 Recursion

```
local Factorial in
   fun {Factorial N}
      if N == 0 then 1 else N*{Factorial N-1} end
   end
   {Browse {Factorial 5}}
end
```

Upper-case names denote variables. Recall that variables can be bound to a value only once, and afterwards they cannot be reassigned to a different value.

The **`local`** block introduces a variable **`Factorial`**, which happens to be a function. This function is the familiar recursive definition of Factorial: It consists of an**`if`** block which evaluates to the expression 1 if the input argument is 0, and evaluates to **`N*{Factorial N-1}`** otherwise.

Please note the syntax of function application - it is **`{function-name arg}`**. The first token in the curly braces is taken to be the function's name, and the remaining are the arguments to the function, if any.

The execution stack of **`{Factorial 4}`** will look as follows.

```
{Factorial 4} = 4 * {Factorial 3}
  {Factorial 3} = 3 * {Factorial 2}
    {Factorial 2} = 2 * {Factorial 1}
      {Factorial 1} = 1 * {Factorial 0}
        {Factorial 0} = 1
```

1.2 Tail Recursion

Typically, even though recursion is a more elegant way of expressing code, there is an overhead incurred due to the maintenance of a deep stack. In the above example, the call to **`{Factorial 4}`** resulted in an execution stack that was 5 layers deep. Note that this stack has to be maintained, since multiplication with **N** is done afterthe recursive call returns.

We will see a technique where *certain kinds of iteration* can be converted automatically into recursive code, which can execute without maintaining a deep stack. The technique is called **Tail recursion**.

Consider an iterative factorial program in C.

```
1:  long factorial(int n)
2:  {
3:     int product = 1;
4:     while(n >= 0){
5:         product = product*n;
6:         n = n-1;
7:     }
8:  }
```

Here, we modify the variables **`n`** and **`product`** to produce the factorial. These variables are called the accumulator variables. **We cannot directly adopt the style in Oz, since Oz variables can be bound only once.** We will imitate this style in Oz. The two main ideas are the following:

1. Create an auxiliary function (helper function), which has the *accumulator variables* as additional arguments. This function is roughly the **`while`** loop in the C code.
2. Then, the main function calls this auxiliary function, initializing the accumulator variables.

```
 1:  declare
 2:  %==========
 3:  % Returns the factorial of N
 4:  %==========
 5:  fun {Factorial N}
 6:     local FactorialAux in
 7:         %===============
 8:         % Auxiliary function - similar to while loop
 9:         % in the C code. N is the number whose factorial
10:         % we need. Product is the cumulative product
11:         %===============
12:         fun {FactorialAux N Product}
13:             if N == 0
14:             then Product
15:             else {FactorialAux N-1 N*Product}
16:         end
17:         %==============
18:         % Main function calls auxiliary with proper initial
19:         % values. Corresponds to Line 3 of the C code
20:         %=============
21:         {FactorialAux N 1}
22:  end
```

Please note the difference of **Lines 13-14** from the recursive definition of Factorial. The execution stack of **`{Factorial 4}`** now looks like

```
{Factorial 4} = {FactorialAux 4 1}
              = {FactorialAux 3 4*1}
              = {FactorialAux 2 3*4*1}
              = {FactorialAux 1 2*3*4*1}
              = {FactorialAux 0 1*2*3*4*1}
              = 1*2*3*4*1
```

In contrast with the recursive code, there is nothing that remains to be done in the calling function, after the recursive call returns - that is, the recursive call is the "last thing" in the calling function - such calls are called tail calls.

If the call is a tail call, an intelligent compiler can just remove the calling stack immediately after the recursive call. Thus in principle, the stackframe can be just 1 layer deep while executing the above. This is called tail call optimization.

Not all recursions are tail calls - a good example is an in-order traversal on a binary tree (Why?)

1.3 Lists! - nil, first, tail

One of the elementary data structures one encounters in FP is, the list. For an introduction to Oz Lists, please consult the introduction.

Briefly, a list is a sequence of elements. The *representation* of a list **L** in Oz is either **nil** or a *tuple* with label **|** (the vertical bar, or the pipe symbol) with two fields - **L.1** is the first element of the list, and **L.2** is itself a list.

Question: Suppose **L** is a list. What is the difference between **{Width L}** and **{Length L}**?

The following illustrates three valid notations for the same list in Oz.

```
[A B C]      = A|B|C|nil       = '|'(1:A
                                    2:'|'(1:B
                                          2:'|'(1:C 2:nil)))
```

1.4 Programming with Lists - Pattern-Matching

A list is recursively defined as either **nil** or a tuple, with the first element being a value, and the second being a list.

When writing programs over lists, the functions we define have to handle both these cases. We can handle this using an **if...then...else** expression, but Oz provides an elegant feature to deal with such situations: this is called pattern-matching, done using a **case** statement.

We now consider some basic functions over lists, as illustration.

- Length of a list

```
declare
fun {ListLength L}
   case L
   of nil then 0
   else 1+{Length L.2}
   end
end
```

- k-element of a list, for a fixed k.

```
declare
fun {Elt L K}
   case L
   of nil then nil
   else
      if K==0 then L.1 else {Elt L.2 K-1} end
   end
end
```

- Concatenation of two lists

```
declare
fun {Concat L M}
   case L
   of nil then M
   [] H|T then H|{Concat T M}
   end
end
```

- (*) Cross-product of two lists

```
%=========
% Input E - a value, and L = [L1 L2 ... Ln], a list.
% Returns a list [[E L1] [E L2] ... [E Ln]]
%========
fun {Preface E L}
   case L
   of nil then nil
   [] H|T then [E H]|{Preface E T}
   end
end
%===========
% Input - L, M lists
% Returns the cross product of the two lists
%===========
fun {CrossProduct L M}
   case L
   of nil then nil
   [] H|T then {Concat {Preface H M}
                       {CrossProduct T M}}
   end
end
```

- Reverse of a List, $O(n^2)$ time

```
declare
fun {ReverseList_1 L}
   case L
   of nil then nil
   [] H|T then {Append {Reverse T} [H]}
   end
end
{Browse {ReverseList_1 [1 2 3]}}
```

- Reverse of a List, O(n)

Here we cleverly use the observation that a function call stack essentially gives the last-in-first-out behaviour we want the reverse to accomplish.

```
declare
fun {ReverseList2 L}
   local ReverseAux in
      fun {ReverseAux Remainder Partial}
         case Remainder
         of nil then Partial
         [] H|T then {ReverseAux T H|Partial}
         end
      end
      {ReverseAux L nil}
   end
end
```

- Reverse of a List, without using helper functions (exponential-time!)

```
declare
fun {Reverse3 L}
   if {Length L} < 2
   then L
   else
      local R = {Reverse3 L.2} in
         R.1 | {Reverse3 L.1|{Reverse3 R.2}}
      end
   end
end
```

For example,

```
{Reverse3 [a b c d]} = d|{Reverse3 a|{Reverse3 [c b]}   //R = [d c b]
                     = d|{Reverse3 a|[b c]}
                     = d|[c b a]
                     = [d c b a]
```

1.5 Recursive Data Types - Follow the definition!

For recursive data-types, we can write functions which "follow" the recursive definition.

For example, a Binary Tree is either empty, or is a root with a left sub-tree and a right sub-tree, both of which are binary trees.

Then, the following is an in-order traversal of a binary-tree represented as a nested record.

```
declare
proc {InOrder BinTree}
   case BinTree
   of nil then skip
   else
      {InOrder BinTree.left}
      {Browse BinTree.root}
      {InOrder BinTree.right}
   end
end
%===== Example Usage ========
T = tree(root:3
         left:tree(root:2
                   left:tree(root:1
                             left:nil right:nil)
                   right:nil)
         right:tree(root:4
                    left:nil
                    right:tree(root:5
                               left:nil right:nil)))
{InOrder T}
```

1.6 Higher-Order Programming

In functional programming, functions are first-class values. This means that functions can be used wherever a value can be given. This includes the following three conditions.

1. It is possible to assign functions as values of variables.

```
local X Double in
   fun {Double Y} 2 * Y end
   X = Double
   {Browse {X 2}}
end
```

7. It is possible to pass functions as arguments.

```
local Accumulate Product in
   fun {Product X Y}
      X * Y
   end
   %==============
   % Function to calculate the cumulative result of
   % iterative application of BinOp over the list L.
   % BinOp is assumed to be right-associative.
   % Identity is the identity element of BinOp.
   %==============
   fun {Accumulate L BinOp Identity}
      case L
      of nil then Identity
      [] H|T then {BinOp H {Accumulate T BinOp Identity}}
      end
   end
   %======== Example Usage ======
   {Browse {Accumulate [1 2 3] Product 1}}
end
```

27. Return values can be functions.

```
28. local AddX F in
29.    %===========
30.    % Input X
31.    % Returns a function which adds X to its argument
32.    %==========
33.    fun {AddX X}
34.       fun {$ Y}  % $ sign denotes "anonymous" function
35.          X+Y
36.       end
37.    end
38.    F = {AddX 2}
39.    {Browse {F 3}}
40. end
```

Higher-order programming enables us to write *many* programs over lists using the following functions.

Map

Map is a higher-order function which takes a unary function F, and a list of elements [L1 L2 … Ln], and returns a list [F(L1) F(L2) … F(Ln)]

```
local Map in
   fun {Map L F}
      case L
      of nil then nil
      [] H|T then {F H}|{Map T F}
      end
   end
end
```

FoldR

A higher order function which takes a (right-associative) binary function B, the identity and a list [L1 … Ln] and returns the value {B L1 {B L2 {… {B Ln Identity}…}}}.

```
local FoldR in
   fun {FoldR L B I}
      case L
      of nil then I
      [] H|T then {B H {FoldR T B I}}
      end
   end
end
```

1.6.1 Programming using Map and Fold

Expression which calculates the sum of squares of the elements in a list.

```
{Browse {FoldR {Map [1 2 3] fun {$ X} X*X end}
               fun {$ X Y} X+Y end
               0}}
```

1.7 Lazy Evaluation

An evaluation strategy where a value is evaluated only when it is needed. Note that the **&&** and **||** operators in C follow a lazy evaluation strategy. We will now introduce this style of programming in Oz.

```
declare
fun lazy {ListOfIntsFrom N}
   N|{ListOfIntsFrom N+1}
end
{Browse {ListOfIntsFrom 0}}
fun lazy {LAppend Xs Ys}
   case Xs
   of X|Xr then X|{LAppend Xr Ys}
   [] nil then Ys
   end
end
```

1.7.1 Lazy Quick Sort

Modified snippet of code taken from Peter Van Roy's site: http://www.info.ucl.ac.be/~pvr/ds/lazyQuicksort.oz

```
proc {Partition Xs Pivot Left Right}
   case Xs
   of X|Xr then
      if X < Pivot
      then Ln in
         Left = X | Ln
         {Partition Xr Pivot Ln Right}
      else Rn in
         Right = X | Rn
         {Partition Xr Pivot Left Rn}
      end
   [] nil then Left=nil Right=nil
   end
end

fun lazy {LQuickSort Xs}
   case Xs of X|Xr then Left Right SortedLeft SortedRight in
      {Partition Xr X Left Right}
      {LAppend {LQuickSort Left} X|{LQuickSort Right}}
   [] nil then nil
   end
end
```

1.8 Advanced - Difference Lists

1.9 Advanced - Continuation-Passing Style

declarative-concurrency

Table of Contents

1 Declarative Concurrency - Introduction

Extend the declarative sequential model to deal with concurrency. The result of execution is the same as a sequential execution. The program results can be calculated incrementally.

```
declare Gen A B X Y
fun {Gen L N}
    {Delay 100}
    if L == N then nil else L|{Gen L+1 N} end
end

A = {Gen 1 10}
B = {Map A fun {$ X} X * X end}
{Browse A B}

thread X={Gen 1 10} end
thread Y={Map X fun {$ X} X*X end} end
{Browse Y}
```

We will extend the declarative sequential model in two steps.

(1) Add threads:

thread <s> end

This gives us one form of declarative concurrency.

that

(2) Extend the model with another execution order

{ByNeed P X}

This adds the possibility of doing ``demand-driven'' or lazy computation. This gives us the other form of declarative concurrency.

We will also deal with soft real-time programming, that is, program with time constraints. The term "soft" refers to the fact that missing a deadline might not be catastrophic, but the quality will decline.

The resulting **declarative concurrent model** will not have Exceptions…

---

2 Basic Concepts

The new model allows more than one executing statement to access the store. As if all of those are executing "at the same time".

2.1 Interleaving and Simultaneity

What execution order do we mean when we say threads execute simultaneously? There are two aspects to concurrency: interleaving among steps, and overlapping of two computation steps.

2.1.1 Language viewpoint:

There is one global sequence of computation steps, and threads take turns executing it. Computation steps do not overlap.

2.1.2 Implementation Viewpoint:

Threads may not be executing interleaved - for example, on a multiprocessor system, they could be executing concurrently. This is parallel execution.

Important Observation: Whatever the parallel execution of a program is, there is at least one interleaved execution that is observationally equivalent to it - that is, the evolution of the **store** during both the parallel and the interleaved execution is the same. It is hence sufficient to study interleaving semantics.

2.2 Causal Order

All computation steps form a global order called a partial order. This is called the **causal order**.

```
Example:

Causal Order:

      ------>---->
      ^ a      b
      |
----->------>----->
  I     J      K


Valid Execution Orders:
I a b J K
I a J b K
I a J K b
I J a b K
I J a K b
I J K a b
```

2.3 Nondeterminism

There is an execution state in which there is a choice of what thread to execute next.

In a declarative model, the nondeterminism is not visible to the programmer. This is due to:

(1) The single assignment store (2) Any operation has to wait until all its variables are bound. If we allow operations which the programmer can use to determine whether to wait or proceed, then the nondeterminism will be visible.

Browsing sometimes exhibits nondeterministic output - does this violate the above statement?

2.4 Thread Scheduling

The Mozart system has a thread scheduler, which decides which thread to run next. A thread can be in the following states: (1) Ready/Runnable : (2) Suspended : e.g. blocked on an unbound variable (3) Terminated

Fair Thread Scheduler: Every runnable thread will eventually execute.

An interesting observation: fairness is related to modularity. Each thread's effect can be studied independent of the existence of other threads (is this possible in fair, nondeclarative models?)

---

3 Semantics of Threads

Abstract machine now has several semantic stacks. Each semantic stack captures the notion of a ``thread".

All threads share the single assignment store. Threads communicate with each other through this store.

We extend the abstract machine in Section 2. An execution state is a pair (MST, s) where MST is a multiset of semantic stacks, and s is a single-assignment store. (Why a multiset?) A computation is a sequence of such execution states. No other concept is changed in the abstract machine.

3.1 Program Execution

The initial execution state is

Semantic Stack	SAS
{[<s>,0]}	0

Note that the semantic stack is a multiset.

At each step, one runnable semantic stack ST is selected from MST, leaving MST'.

MST = {ST} \multisetunion MST'.

One computation step is done in ST according to the semantics.

(ST, s) ——> (ST', s')

The choice of which stack to pick is up to the scheduler. The scheduler ensures fairness.

What happens when the currently executing statement blocks? In the sequential model, the interpreter would just hang. In the concurrent model, the interpreter has to try and run other stacks…

If there are no runnable stacks, and if every stack is terminated, then the computation terminates. If there are no runnable stacks, but at least one stack is suspended, then the computation blocks.

3.2 Semantics of the thread statement

The syntax of a thread statement: thread <s> end

Step: Create a new stack! What should be its contents?

Suppose the multiset is MST. if the selected stack ST is of the form

```
[thread <s> end, E]     % Stack Top
ST'                     % Rest of stack
```

then the new multi set is:

```
[<s>, E]  \multisetunion     ST'          \multisetunion    MST'
```

3.3 Memory Management

A terminated semantic stack can be deallocated.

A blocked semantic stack: Reclaimed if its activation condition depends on an unreachable variable. In this case, the thread will never be ready again, so execution is unchanged by removing this thread.

3.4 Example Execution

```
local B in
  thread B=true end
  if B then true end
```

```
end
```

When the local statement has executed, the semantic multistack looks thus:

Stack	Store
([thread…end, if…end])	{b}

After executing the thread statement, we get

Stack	Store
([b=true], [if b=true then true end]),	{b}

Now, there are two stacks. The second is blocked since its activation depends on b, which is unbound. So the scheduler picks the other thread. After executing b=tre, the stack is now

Stack	Store
([],[if b=true then true end])	{b=true}

Now the empty stack is removed.

Stack	Store
([if b=true then true end])	{b=true}

The scheduler then executes the single thread containing the if statement.

---

4 Declarative Concurrency

There are two issues:

(1) The inputs and the outputs are not necessarily values, since they may contain unbound variables. (2) The computation might not terminate.

4.1 Partial Termination

---

fun {Double L} case L of nil then nil else H|T then 2*H | {Double T} end end Y = {Double X}

If X keeps growing, then so does Y. But if X does not change, then Y stops growing and the program remains unchanged.

If the inputs do not change, the program stops executing any further. This is called partial termination.

4.2 Logical Equivalence

A set of store bindings: constraint.

For each variable x and constraint c, values (x,c) is the set of all possible values x can have, given that c holds.

Two constraints c1 and c2 are logically equivalent if (1) They contain the same variables and (2) For each variable x, values(x, c1) = values(x,c2)

For example,

x = rec(a b) and a = d is

equivalent to

x = rec(d b) and a = d

4.3 Declarative Concurrency - Definition

A program is said to be declaratively concurrent if for all inputs, the program does the following. All the executions either

(1) fail to terminate (2) terminate and are logically equivalent.

4.4 Failure and Confinement

If a declarative concurrent program results in a failure for a given set of inputs, then all possible executions with that set will fail.

For example, the bindings in

```
thread X=1 end
thread Y=X+1 end
thread X=Y end
```

will eventually conflict. The program terminates.

One way to handle this is using exceptions. But, with these, the executions are no longer declarative, since the store after different execution sequences could be different. For example, Y might be bound to 1 or to 2 before the execution fails in the example above.

Hiding the nondeterminism in case of failure: This is up to the programmer.

```
declare X Y
local XTrial YTrial StatusX StatusY StatusV in
  thread
    try
      XTrial = 1
      StatusX = success
    catch _ then StatusX=error end
  end
  thread
    try
      YTrial = XTrial+1
      StatusX = success
    catch _ then StatusX=error end
  end
  thread
    try
      XTrial = YTrial
      StatusX = success
    catch _ then StatusV=error end
  end
  if StatusX==error orelse
     StatusY==error orelse
     StatusV==error then
    X=1
    Y=1
  else
    X=XTrial
    Y=YTrial
  end
end
% Rest of the code using X and Y
```

5 Programming with threads

```
proc {ForAll Xs P}
     case Xs of nil then skip
     []   X|Xr then {P X} {ForAll Xr P} end
end

declare L in
thread {ForAll L Browse} end
end
```

Now, bind L

```
declare L1 L2 in
thread L = 1|L1 end
thread L1 = 2|L2 end
thread L2 = 3|nil end
```

The effect is the same as sequential execution, but the results can be computed incrementally.

5.1 Time Slices

Allowing each process to run for one step can increase the overload on the scheduler, or in other cases, hog the processor for too long executing an intensive step. The Mozart scheduler uses a preemptive scheduling based on timers.

Allowing one whole computation step leads to a deterministic scheduler. Running the threads multiple times preempts the threads exactly at the same steps.

The timer based-approach is hardware-implemented, hence more efficient. This is however, nondeterministic, and depends on external events in the system as well.

Either the thread itself, or the scheduler can keep track of the time a thread has run. The second is easier to implement.

A short time slice -

- responsive system.
- large overhead for thread switches.

Large number of threads with long timeslices (~10 ms)? == if the threads are interdependent, then long timeslices are ok. == if the threads are independent, long timeslices are not ok. You need a hard-real-time OS.

5.2 Priority Levels

Provides more control over how processor time is shared between threads. Three priority levels in Mozart: high, medium, low. Each has a guaranteed lower bound on processor time share. 100:10:1

When a thread creates a child thread, then child thread inherits the parent's priority. This is important for threading high-priority applications.

5.3 Cooperative vs. Competitive Concurrency

Threads are cooperative in completing a global task. On the other hand, you can have a competitive model where entities compete to complete a task. Usually those are OS processes.

Competitive Concurrency in Mozart - distributed computation model and Remote module. The distributed model is network-transparent.

---

6 Streams

A potentially unbounded list of messages. Its **tail** is an unbound dataflow variable (not a general partial value).

Sending a message: Send one element to the stream. Bind the tail to a list pair consisting of the element, and a new unbound tail.

Receiving a message: read one stream element.

A thread communicating through streams : **Stream Object**.

No locking or mutual exclusion is necessary since each variable is bound by only one thread.

Stream-based programming: e.g. Unix Pipes

```
            Summary--what's most important.
    To put my strongest concerns into a nutshell:
1. We should have some ways of coupling programs like
garden hose--screw in another segment when it becomes when
it becomes necessary to massage data in another way.
This is the way of IO also.
2. Our loader should be able to do link-loading and
controlled establishment.
3. Our library filing scheme should allow for rather
general indexing, responsibility, generations, data path
switching.
4. It should be possible to get private system components
(all routines are system components) for buggering around with.

                                                M. D. McIlroy
                                                October 11, 1964
```

6.1 Basic Producer/Consumer

Recall the list of random bits in Homework 1. The function which produced the stream of random bits is like a producer, and the function which computed running averages is like the consumer. This shows how to form a basic producer/consumer style program in the declarative model with threads.

You can also have multiple consumers without affecting the execution in any way.

```
local Xs S1 S2 in
    thread Xs={GenerateRandom} end
    thread S1={Average Count1} end
    thread S2={Average Count2} end
end
```

6.2 Transducers

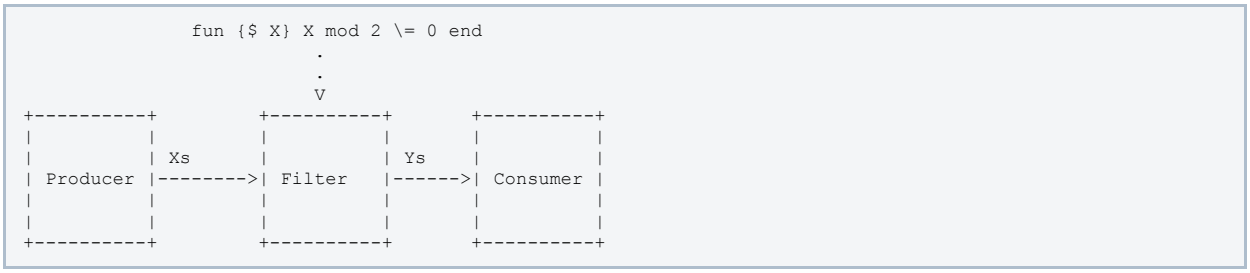
We can put many other stream objects between a producer and a consumer. These are called **Transducers**. A sequence of stream objects each of which feeds the next is called a **pipeline**. The simplest stream is a filter.

6.2.1 Sum of odd numbers in a stream

```
%-----------
% Producer
%-----------
fun {Generate N Limit}
   if N<Limit
   then N|{Generate N+1 Limit}
   end
end
%------------
% Consumer
%------------
fun {Sum Xs Accumulator}
   case Xs
   of X|Xr then {Sum Xr X+Accumulator}
   [] nil then Accumulator
   end
end
%-----------
```

```
% A filter in between producer and consumer
%-----------
local Xs Ys S in
   thread Xs={Generate 0 1000000} end
   thread Ys={Filter Xs
                    fun {$ X} X mod 2\=0 end} end % Take odd numbers
   thread S={Sum Ys 0} end
end
```

This can be represented graphically using the following diagram (sometimes called a Henderson Diagram.) The dashed arrows represent stream inputs, and the dotted arrow represents non-stream inputs.



6.2.2 Eratosthenes Sieve

We implement the Eratosthenes' Sieve as a stream-based Oz program. The Sieve generates prime numbers, starting from 2. The algorithm works as follows. First, we consider all consecutive numbers from 2, and remove multiples of 2 from this. At any stage, we pick the first number ~n~from the list whose multiples have not yet been removed, and remove all its multiples from the list, except for **n** itself. This process is iterated. The ultimate stream will consist of prime numbers only.

We can program this in the stream model as follows.

```
%---------------
% Returns an infinite stream of integers starting from N
%---------------
fun {IntsFrom N}
    N|{IntsFrom N+1}
end
%--------------------
% Returns a function which checks whether its argument is a multiple
% of N
%--------------------
fun {NonMultiple N}
   fun {$ M}
       M mod N \= 0
   end
end
%--------------------
% Sieve of Eratosthenes
%--------------------
fun {Sieve Xs}
    case Xs
    of X|Xr then
        thread
           Ys = {Filter Xr {NonMultiple X}} % Remove multiples of X
        end
        X|{Sieve Ys}
    [] nil then nil
    end
end
%---------------------
% Example Execution
%---------------------
local Xs Ys in
   thread Xs={IntsFrom 2} end
   thread Ys={Sieve Xs} end
   {Browse Ys}
end

% Think about why the usage code was not simply
% local Ys in
%    Ys = {Sieve {IntsFrom 2}} % Eager version
% end
```

6.3 Stream Objects

As an abstraction, we can introduce a new concept called a **Stream Object**. A stream object is a recursive procedure, which executes in its own thread, communicates with other stream objects via input and output streams, and finally, maintains a state.

```
proc {StreamObject InStream CurrentState ?OutStream}
   case InStream
   of InMsg|InStreamTail then
      NextState OutMsg OutStreamTail
   in
      % NexStateProcedure is the transition procedure
      {NextStateProcedure InMsg CurrentState OutMsg NextState}
      OutStream = OutMsg|OutStreamTail % Why is this done before recursing?
      {StreamObject InStreamTail NextState OutStreamTail}
   end
end
```

---

7 Using the declarative Concurrent model directly

We can use the declarative concurrent model directly without relying on to Streams or Stream Objects. The ingredients are partial values, and threads. We consider a few examples.

7.1 Concurrent Composition

We can create new threads using a **`thread`** statement. In a system with threading, the two important operations that are usually supported are thread creation, and thread join. Joining is the process whereby the creator of the new thread waits until the newly created thread is destroyed, before it continues.

The basic idea is for the creator thread to **`Wait`** for the binding of an unbound variable, which is bound by the newly created thread just before it terminates. The procedure **`{Wait X}`** blocks until **`X`** is bound, and gets unblocked when **`X`** is bound.

```
local X in
   thread {P1} X=unit end
   {Wait X} % Creator thread blocks until X is bound
end
```

This is easily extended to multiple processes.

```
local X1 X2 X3 in
   thread {P1} X1=unit end
   thread {P2} X2=X1   end
   thread {P3} X3=X2   end
   {Wait X3} % blocks until X3 is bound,
             % which happens only when all three new threads terminate.
end
```

We can abstract this into a procedure **`{Barrier Procs}`**, which takes in a list **`Procs`** of procedures. This executes each procedure in the list in its own thread, and blocks until all the threads have terminated.

The procedure merely generalizes the methodology described above.

```
proc {Barrier Procs}
   %------------------------
   % Takes a finite list of procedures, and a variable L used for
   % synchronizing.
   %
   % Runs each process in its own thread,
   % creates a new variable M, binds it to L, and passes M recursively
   % to run other procedures.
   %
   % Return value is L if there is no procedure to execute,
   % and the result of the recursive call otherwise.
   %------------------------
   fun {BarrierLoop Procs L}
```

```
      case Procs
      of  Proc|Procr then
          M
      in
          thread {Proc} M=L end
          {BarrierLoop Procr M}
      [] nil then L
   end


   S = {BarrierLoop Procs unit}
in
   {Wait S}
end

%------------ Example Usage-------------------
{Barrier [ proc {$} X = 1 end
           proc {$} {Browse X} end]}
```

---

mpc

Table of Contents

1 Semantics of Ports

We add a mutable store. So we have three kinds of stores:

1. Single Assignment Store: S
2. Trigger Store: T
3. Mutable Store: M

M contains pairs of the form **`X:Y`** where X and Y are variables in the Single Assignment Store.

The mutable store is initially empty.

Invariant: X is always bound to a **name** value that represents a port and Y is unbound. Names are unique, unforgeable constants.

Execution State is now (MST, S, T, M).

1.1 NewPort

({NewPort <s> <p>}, E)

does the following:

1. Create a fresh port name n.
2. Bind E(<p>) and n in the single assignment store.
3. If the binding is successful, then (1) Add the pair E(<s>) : E(<s>) to the mutable store M.
4. Else (1) Raise an error condition.

1.2 Send Operation

({Send <p> <y>), E)

1. If E(<p>) is determined (1) Uf E(<p>) is not bound to a port, then raise error (2) If the mutable store contains E(<p>):z
    1. Create a new variable z' in the store
    2. Update the mutable store to be E(<p>):z'
    3. Create a new list pair E(<y>)|z' and bind z with it in the single assignment store.
2. Else, suspend execution.

In a correct port, the end of the stream should be read-only. We have not implemented this. This can be implemented with "Secure ADT"s.

1.3 Memory Management

1. Extend the definition of reachability. If x:y is in the mutable store, and x is reachable, then y is reachable as well.
2. If the port variable x becomes unreachable, and the mutable store contains x:y, then y becomes unreachable as well.

2 Port Objects

A combination of one or more ports and a stream object.

Extends stream objects with many-to-one communication. Also, port objects can be embedded inside data structures.

Similar ideas in other languages: Agent Erlang Process Active Object

Message Passing Model: Program consists of a graph of interacting port objects.

```
declare P1 P2 ... Pn
local S1 S2 ... Sn in
  {NewPort S1 P1}
  {NewPort S2 P2}
  ...
  {NewPort Sn Pn}
  thread {RecrsveProc S1 ... Sn} end
end
```

RecrsveProc is a recursive procedure that reads the port streams and performs some actions for each message received.

Sending a message to a port objects: Send a message to any of the streams.

```
declare Port
local Stream in
   {NewPort Stream Port}
   thread for M in Stream do {Browse M} end end
end
% Only Port is visible outside
thread
```

```
   {Send Port hello}
   {Delay 1000}
   {Send Port world}
end
thread
   {Send Port hi}
end
```

2.1 Port Object abstraction

If there is no internal state, then the new port is simple.

```
fun {NewPortObject2 P}
Sin in
  thread
    for Msg in Sin do
      {P Msg}
    end
  end
  {NewPort Sin}
end
```

However, we can also consider ports which have an internal state, similar to that in the case of stream objects.

```
fun {NewPortObject InitState F}
Sin Sout in
  thread {FoldL Sin F InitState Sout}
  {NewPort Sin}
end
```

Using this, we can program simple message passing protocol.

---

e.g. from the text

3 players tossing a ball to each other. Each recepient tosses the ball to one of the other two, chosen at random.

Each player is a stateless port object.

Code: Players.oz

2.2 Reasoning with Port Objects

Proving a program correct consists of two parts.

1. Show that the port object is correct. Each port object defines a data abstraction. The abstraction should have an invariant assertion. Since the inside of a port is declarative, we can use the techniques for declarative programs.

   Each port object has a single thread. Hence it's operations are executed sequentially. Hence mathematical induction provides a way to assert invariants.

   - When the port object is first created, the invariant is satisfied.
   - If the assertion is satisfied before a message is handled, then the assertion is satisfied after it is handled.
2. Show that the program using the port objects is correct. This involves interacting ports. We have to determine the possible sequences of messages that each port can receive. First, we classify the events - message sends, message receives, internal state changes.

   Then we can consider the program as a state transition system. Reasoning about the correctness of such transition systems is complicated.

3 Simple Message Protocols

3.1 RMI

```
|\ |       Method is port's message-handling procedure.
| \|       A client sends a request to a server and waits for a reply.
|  |
|  |       ---------Server---------
| /|       proc {ServerProcedure Msg}
|/ |         case Msg
|  |           of append(X Y Z) then {Append X Y Z}
|\ |         end
| \|       end
|  |       Server = {NewPort2 ServerProcedure}
| /|       ------------------------
|/ |
|  |       ---------Client---------
C  S       proc {ClientProcedure Msg}
            case Msg
              of work(X Y Z R) then
                local W in
                  {Send Server append(X Y W)}
                  {Wait W}
                  {Send Server append(W Z R)}
                end
              end
            end
            Client = {NewPort2 ClientProcedure}
            declare R
            {Browse {Send Client work([1 2] [3] [4] R)}
            ------------------------
```

The client refers directly to the server, but not vice versa.

This is a synchronous call.

3.2 Asynchronous RMI

```
|     |
|\    |       The client proceeds immediately after sending the request,
| \   |       without waiting  for the server response.
|  \  |
|\  \ |
| \  \|
|  \  |
|   \/|
|   /\|
|  /  |
| /  /|
|/  / |
|  /  |
| /   |
|/    |
|     |
C     S
```

3.3 RMI with callback (using threads)

---

4 Using the Message Passing Model directly

4.1 Port Objects that share one thread

TBD

4.2 A thread abstraction with termination detection

TBD

4.3 Eliminating Sequential Dependencies

```
proc {ConcFilter L F ?L2}
   Send
in
   {NewPort L2 Send}
   {Barrier
       {Map L
            fun {$ X}
               proc {$}
                    if {F X} then {Send X} end
               end
            end}}
end
```

---

5 Semantics of Erlang Receive

Erlang is a concurrent, functional programming language developed by Ericsson. It supports efficient threading, fault tolerance, and "hot code replacement". The computation model has the following features.

The computational model consists of communicating entities called "**processes**". Each process has a **port**, and a**mailbox**.

1. A Functional core: Erlang is a dynamically typed, strict language. (Arguments are evaluated eagerly.) Each process has a port object defined by a recursive function. A process is spawned by specifying the initial function to execute.
2. Message-passing extension. Processes communicate by sending messages to other processes, asynchronously. The messages are delivered in FIFO order. Messages can contain any value, including functions. Each process has a unique process identifier, PID.

   Message receiving can be either blocking or non-blocking. The reception works using pattern-matching. Messages can be removed out-of-order, without removing the others in the mailbox.

Processes are independent by default (what other model is there?). This implies that messages are always copied from one process to another, never shared. Independence is better for reliability.

We are concerned only with these features of Erlang for the present. The complete language has features for distribution, fault-tolerance via *linking* of processes, and persistence.

```
-module (helloserver).
-export ([start/0, receiverloop/0]). % functions exported publicly,
                                     % other functions are 'private'.
                                     % foo/n denotes that foo is a
                                     % function with n arguments.

% spawn a process with the module helloserver, executing the function
% receiverloop. The argument list to receiverloop is empty.
start()->
   spawn(helloserver, receiverloop, [])


receiverloop()->
   receive
       {From, hello}->          % Message format : {Pid, hello}
           From!hi,             % Send hi to process with Pid 'From'
           receiverloop()
   end.
```

5.1 The `receive` operation

In this section, we will try to describe the semantics of the Erlang **`receive`** operation, in terms of equivalent Oz statements.

explicit-state

Table of Contents

1 Introduction

Declarative Programming: describing "what" to compute, without describing how to compute it. Usually stateless. Imperative Programming: describing "how" to compute. Usually includes state.

Advantages of declarative programming include

- Easier to build abstractions (procedural abstractions, higher order functions)
- Easier to test (referential transparency - a function always returns the same values for the same arguments). In programming with state, we have to test **sequences of execution**.
- Reasoning is easier (algebraic reasoning is possible).

1.1 State

We will deal with explicit state, in a sequential setting.

Principle of abstraction: Programs can be separated into specification and implementation. Specification: A contract defining how the system should behave Implementation: How the specification is realized.

Declarative and imperative programming both support this. However, when a system "grows", declarative programs have to be changed. The only way for a declarative program to know more about the external world, is by having extra arguments (for example, more **accumulators**). This means that not only the implementation, but the specification also changes.

Having internal state within the program helps us to model internal state as extra information within the program while keeping the specification the same as before. So imperative program supports this kind of abstraction better.

A system that supports the principle of abstraction has

1.1.1 Encapsulation

It should be possible to hide the internal details of a part.

E.g. A complex number data type can represent its data either in polar coordinate system or in a rectangular coordinate system, if its operations are correct.

Example of a feature that supports encapsulation: Lexical scoping: variables inside a lexical scope are inaccessible by entities outside the scope.

1.1.2 Compositionality

Possible to combine parts to make a new part.

e.g. The programmer should be able to build more complex data structures where their components themselves are user-defined types.

1.1.3 Instantiation/Invocation

Multiple instances of the same definition can be created. These instances may be different based on the environment in which they were instantiated/invoked.

A concrete example: Classes and objects. The objects, when initialized could take arguments based on the point where they are created.

A feature that supports instantiation: higher-order programming.

1.1.4 Invariant

With states, execution can have side effects. This makes reasoning about the correctness of programs, difficult. One way to alleviate this is by using encapsulation. Each encapsulation can be defined to have a property called an **invariant**, which is always true when viewed from the outside.

1.1.5 Progress

In addition to invariants (what is constant), we need a proof that the system is progressing towards a goal (what changes).

1.2 Component-based Programming

Characterized by encapsulation, compositionality and instantiation.

Each component is a part of the system.

Specifies an interface: how the component is seen from the "outside".

In component-based programming, we can compose components: build a new component that contains the original one.

2 Names

For the discussion, we will need a new concept in Oz called names. Names are constants which are unique in the system. Only two operations are allowed on them.

```
{NewName} % returns a fresh name
N1==N2    % Compares N1 and N2
```

The name created by a call to NewName is guaranteed to be unique in the system.

It is not possible, for example, to convert a name to a printable string which is displayed using Browse. The only way to know a name is to be passed it as a reference.

We will use names to define cells, and to secure data in the following discussion.

3 Explicit State

In declarative model, a component's behaviour is dependent only on its arguments. With explicit state, the behaviour depends on the internal state of the component as well.

Advantage of explicit state - can lead to shorter argument lists for components. Advantage of declarative model - we have substitute functions with their values without changing the behaviour of programs (usually called**referential transparency**)

Definition A **state** is a sequence of values in time that contains the intermediate values of a computation.

3.1 Implicit State

We can model state, even in the declarative style.

```
fun {SumList Xs S}
    case Xs
    of nil then S
    [] X|Xr then {SumList Xr X+S}
end
```

The code above is a *declarative* code which sums up the elements in a list. Consider the call {SumList [1 2 3] 0}. The last argument to the recursive calls takes the sequence of values 0, 1, 3, 6. This sequence can be seen as a state as per the definition.

3.2 Explicit State

Definition An **explicit state** in a procedure is a state whose lifetime extends over more than one procedure call, without being one of its argumnents.

Explicit state cannot be expressed in the declarative model. For that, we need a new concept called a *cell*.

```
local
   C = {NewCell 0}
   SumList
in
    proc {SumList Xs}
        case Xs
        of nil then skip
        [] X|Xr then
           C:=@C+X
           {SumList Xr}
        end
    end
    {SumList [1 2 3 4]}
    {Browse @C}
end
```

There are two operations allowed on cells

- := sets the content of the cell
- @ accesses the content of the cell

4 Declarative Model with explicit state

Concurrent components like stream objects and ports already introduce state into the declarative model. We will now add state explicitly into the model. The program model is sequential.

Explicit state is a *pair* of language entities: first, the state's identity and the second is the state's current content.

There is a mapping from the state's identity to its content. This mapping can be modified - this is what introduces explicit state. Interestingly, neither the identity nor the current content can be modified. Modifying the mapping produces the same effect as the modification of the state's content.

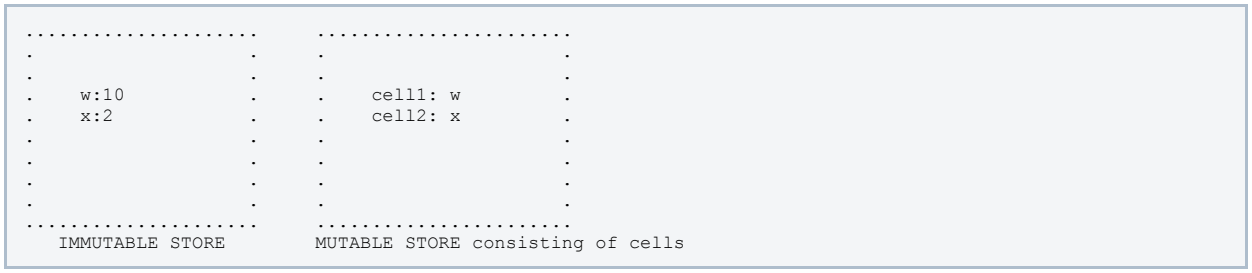
4.1 Cells

Definition A **cell** is a pair of a name, and a reference into the SAS.

Names are unforgeable, hence a cell is a secure ADT.

We introduce a new store called the mutable store. The cells are part of the mutable store.

There are now two stores: the SAS, which is immutable, and the mutable store of the cells.



We add two new statements to the declarative sequential model of Chapter 2, to handle cells.

4.2 NewCell

The statement that creates a cell is

```
{NewCell Cell InitialValue}
```

Semantics

In addition to the single assignment store (and the trigger store if we consider lazy evaluation), we create a mutable store. This store is initially empty. The mutable store contains cells which are pairs of the form x:y where x and y are variables of the SAS. x is always a name. y can be any partial value.

The execution state is now a tripe (Stack, SAS, Mutable).

The semantics for the ({NewCell Cell InitialValue}, Env) statement is:

- Create a new name n in SAS
- Bind E(Cell) and n in SAS
- If binding succeeds, add the pair E(Cell):E(InitialValue) in the Mutable store
- Otherwise, raise an error.

4.3 Exchange

The statement that modifies the value of a cell is:

```
{Exchange Cell OldValue NewValue}

(OR)

OldValue=(Cell:=NewValue)
```

This statement atomically binds OldValue to the old content of Cell, and sets the new content of Cell to be NewValue. Another syntax for the exchange statement is using the := operator. Usually we do not care about the old value of the cell, so the syntax simply reads

```
Cell:=NewValue
```

Semantics

The semantics of ({Exchange Cell OldValue NewValue},E) is

+If the activation condition, E(Cell) being determined, is false, suspend execution. Otherwise, we do the following.

1. If E(Cell) is not bound to the name of a cell, then raise an error
2. If the mutable store contains E(Cell):w, then do the following
    - Update the mutable store to E(Cell):E(NewValue)
    - Bind E(OldValue) and w in the store.

4.4 Memory Management

We just have to modify the definition of reachability to deal with the new operations. In addition to the previous cases, a variable y is reachable if the mutable store contains x:y and x is reachable.

If a variable x becomes unreachable, and the mutable store contains x:y, we remove this pair from the MS. (Note that y is not necessarily reclaimed in SAS.)

Reversing a list with cells:

```
fun {Reverse Xs}
   R = {NewCell nil}
in
   for X in Xs do Rs := X|@Rs end
   @Rs
end
```

4.5 Sharing and Equality

A cell is semantically, a pair of the form x:y. Hence we now have to distinguish the equality of cells from the equality of their contents.

The concept of **sharing** or **aliasing** is when two identifiers refer to the same cell.

```
X = {NewCell 0}
Y = X            % Y is an alias of X
Y := 10
{Browse @X}      % displays 10
```

When any alias changes the content of the cell, all the aliases see the changed content. This makes reasoning about the program difficult (In the previous example, if you just consider the statements in which X appears, you cannot compute the current value of X.)

4.5.1 Token and Structure Equality

Two values are equal iff they have the same structure. For example, the variables in the following are equal.

```
X = rec(f:1)
Y = rec(f:2)
{Browse X==Y}    % displays true
```

This is **structure equality**. With cells, we also introduce the concept of **token equality** - two cells are equal iff they are the same cell.

```
X = {NewCell 0}
Y = {NewCell 0}
{Browse X==Y}    % displays false
{Browse @X==@Y}  % displays true. However, this may change subsequently
Y:=1
{Browse @X==@Y}  % displays false now

Z = X
{Browse X==Z}    % displays true
```

*

5 Data Abstraction

A data abstraction is a way to define the abstract behaviour of data independent of its specific implementation. This leads to better modularity in the overall program, making it easier to reason about and to make changes in speicific parts of the program without affecting other parts.

The book talks about three axes of designing data structures.

- Open vs. Secure
- Unbundled (ADT) vs. Bundled (e.g. Classes)
- Declarative vs. using Explicit State.

All choices among these are possible, leading to eight different ways of abstracting data. We will consider a stack implementation, and consider some ways of doing this.

5.1 Open Unbundled Declarative stack

Suppose we implement a stack as a list, which supports operations for pushing, popping, and to check emptiness. We might implement it in the following manner.

```
local
  fun {NewStack} nil end
  fun {Push E S} E|S end
  fun {Pop S} if {isEmpty S}==false then S.1 end
  fun {IsEmpty S} S==nil end
in
  local X Y in
    X = {NewStack}
    {Browse {Pop {Push 2 {Push 1 X}}}} % displays 2
end
```

This implementation, where there is no encapsulation of the data and the procedures acting on it, is called an**unbundled** implementation or an **Abstract Data Type** (ADT).

The disadvantage is that if a programmer guesses that the stack is implemented as a list, (s)he can call {Length S}, which is not an operation that is meant to be performed on a stack.

5.2 Secure Unbundled Declarative stack

How can we ensure that a user goes only through the procedures that are provided in the ADT? We can use names for this purpose.

We will first introduce the notion of a wrapper/unwrapper pair of functions. To secure some value X, what we will do is to "lock" X inside a function, with a name - we call this they "Key". This function expects a key as input. If the given key is correct, we get the value of X.

Unwrap is a function which can unlock the wrapped version of X by giving the correct key.

Two things together ensure security: NewName returns a unique name, and the Key thus obtained is hidden from the outside by lexical scoping.

```
declare

proc {NewWrapper ?Wrap ?Unwrap}
    Key={NewName} % Unique key.
in
    % returns a function, which serves as
    % the "wrapped version" of X
    fun {Wrap X}
       fun {$ K}
         if K==Key then X end
       end
    end

    % function which takes a wrapped input and
```

```
    % returns the unwrapped value
    fun {Unwrap W}
       {W Key} % Calls wrapped version with the correct key
    end
end

% Usage
S = [1 2 3]
SecureS = {Wrap S}    % Note: SecureS is a function!
T = {Unwrap SecureS}  % T = [1 2 3]
                      % {Unwrap {Wrap S}}==S
% {Length SecureS} will fail
```

Using the wrap/unwrap pair, we can now implement a secure unbundled stack.

```
local Wrap Unwrap in
    {NewWrapper Wrap Unwrap}

    % Note that the wrapped stack is a function, not a list.

    fun {NewStack} {Wrap nil} end

    fun {Push E S} {Wrap E|{Unwrap S}} end

    % Pop binds the top of the stack in E and returns the smaller
    % stack.
    fun {Pop S ?E}
        case {Unwrap S}
        of H|T then
            E=H
            {Wrap S1}
        end
    end

    fun {IsEmpty S} {Unwrap S}==nil end
end
```

Now, the only way a user can mainpulate the stack is through the set of four functions above. A call of the form {Length {NewStack}} will fail.

5.3 Secure Bundled Stateful stack

The above two implementations were "unbundled" implementations. We can implement declarative bundled implementations as well. We will change two design choices at the same time and consider a secure, bundled, stateful stack. We will see that we do not need names here: security is ensured just by lexical scoping.

```
fun {NewStack}
    C={NewCell nil}
    proc {Push E} C:=E|@C end
    fun {Pop} case @C of X|S1 then C:=X1 S end end
    fun {IsEmpty} @C==nil end
in
    stack(push:Push pop:Pop isEmpty:IsEmpty}
end
```

6 Polymorphism

An procedure is **polymorphic** if it can operate on arguments of different types. There are two main kinds

- Generic : The same procedure body can operate on different argument types.
- Ad hoc : There are different procedure bodies for different kinds of arguments. This is also called*overloading* in Object-oriented languages.

Though polymorphism can be supported in any paradigm, it is easier to define for object-oriented languages.

We give two examples of polymorphic code, one with the secure unbundled implementation of a collection, and the other with an open bundled declaration.

TBD

* Polymorphism - Introduction
An operation is *polymorphic* if it works correctly for arguments of different types.

Two different kinds of polymorphism:
    Universal - the operation can handle different types using the same code.
    Ad hoc   - Different types may be handled using different code.

Examples of polymorphic operations:
    + operation in C can take integer arguments or float arguments.

All reasonable data abstractions can provide polymorphism : if the operation works correctly with one particular interface, then it can work with any other data abstraction with the same interface.

* Bundled and Unbundled Data Abstractions - Introduction

We will discuss two particular styles of data abstraction, and how polymorphism works in these:
        1. Unbundled implementation: an ADT where the Data and the functions on the ADT are stored separately.

        2. Bundled data abstractions, as in objects, where both the ADT and the operations are members of the data abstraction. A bundled data type provides two kinds of entities:
                1. Value
                2. Operation.

** The problem with unbundled data

Suppose we implement a stack in the following unbundled form.

```
--------------------------
fun {NewStack} nil end
fun {Push X S} X|S end
fun {Pop S ?X}
    case S
    of H|T then
        X=H
        T
    end
end
fun {IsEmpty S} S==nil end
--------------------------
```

A programmer who realizes that the stack is implemented as a list, can do S.1.2.1 to get the second element on the stack, which violates the data abstraction. This is the basic problem: how do we ensure that programmers do not manipulate the stack directly except through the functions that we provide?

** Secure Data Types
We can introduce protection boundaries with the help of *Secure data types*. We introduce the notion of a *name*. A name is a literal that is unique in the system. There are exactly two operations supported by names:

1. {NewName} : create a new name. This is a stateful function, since repeated calls must return different values.
2. N1==N2

A name cannot be printed.

** Secure Unbundled ADT

The basic idea to secure an unbundled data type is to wrap the data inside secure functions. These functions will allow access to the internal data only if you give it a correct key.

```
--------------------------
proc {NewWrapper ?Wrap ?Unwrap}
   Key = {NewName}
in
   fun {Wrap X}
      fun {$ K} if K==Key then X end end
   end
   fun {Unwrap WrappedObject}
      {WrappedObject Key}
   end
end
% {Unwrap {Wrap 2}} = {{Wrap 2} Key} = 2
--------------------------
```

Using this, we can implement a secure stack as follows.

```
--------------------------
declare NewStack Push Pop IsEmpty
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap nil} end
   fun {Push X S} {Wrap X|{Unwrap S}} end
   fun {Pop S ?X}
      case {Unwrap S}
      of H|T then
         X=H
         {Wrap T}
      end
   end
   fun {IsEmpty S} {Unwrap S}==nil end
end
--------------------------
```

We can now implement a stateful version, in an encapsulated manner.

```
--------------------------
declare NewStack Push Pop IsEmpty
local Wrap Unwrap in
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   fun {Push X S}
      C={Unwrap S} C:=X|@C {Wrap C}
   end
```

```
   fun {Pop S ?X}
      C={Unwrap S}
      case @C
      of H|T then
         X=H C:=S1 {Wrap C}
      end
   end
   fun {IsEmpty S} @{Unwrap S}==nil end
end
--------------------------
```

Using this, we can now discuss polymorphism.

* Polymorphism in Bundled and Unbundled abstractions

Polymorphism in the object style is easier. In the unbundled style, it requires first-class modules.

** Collection Type

Unbundled:

```
--------------------------
declare Collection
local Wrap Unwrap
    {NewWrapper Wrap Unwrap}
    fun {NewCollection}
        {Wrap {Stack.new}}
    end
    fun {Put C X}
        S = {Unwrap C}
    in
        {Stack.push X S}
    end
    fun {Get C ?X}
        S={Unwrap C}
    in
        {Stack.pop S X}
    end
    fun {IsEmpty C}
        {Stack.isEmpty {Unwrap C}}
    end
in
    Collection =
     collection(new:NewCollection
                put:Put
                get:Get
                isEmpty:IsEmpty)
end
--------------------------
```

Bundled:

```
-------------------
fun {NewCollection}
    S={NewStack}
    fun {Put X}
                {S.put X}
    end

    fun {Get}
        {S.pop}
    end


    fun {IsEmpty}
        {S.isEmpty}
    end
in
   collection(put:Put
              get:Get
              isEmpty:IsEmpty)
end
  ------------------------------
```

Example Use:

```
--------------------------
C={Collection.new}
{Collection.put C 1}
{Collection.put C 2}
X={Collection.get C}
{Browse X}
--------------------------
```

```
 ------------------------------
```