Streams And Iteration In A Single Assignment (Functional) Language SISAL

Stream - data type
- Ordered sequence of values

Single Assignment - A name is assigned a value at most once during the execution of a program (actually within a given definitional scope)

Functional
- A program is a function
- There are no variables (memory objects) whose states are modified by operators
- Functions return arbitrary typed values
- Every atomic expression produces a value (for, if)

Single Assignment Variables Versus Mutable (normal) Variables

What parallelism can you identify in the following segment of code?

```
x = f_1(v);
y =  x+f_2(w);
x  = f_3(u);
z = x + f_4(s);
```

What parallelism can you identify in the following segment of code?

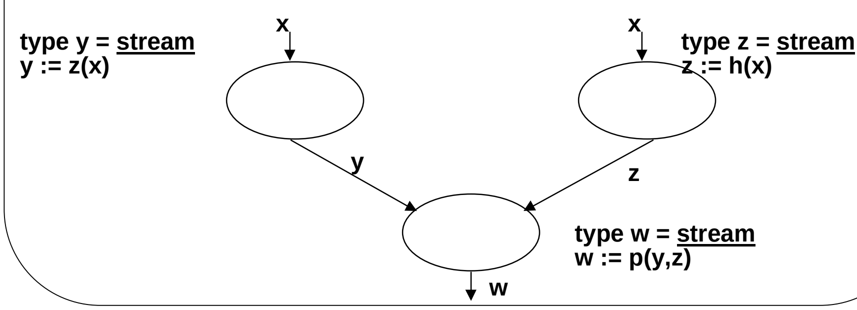
```
x = f_1(v);
y =  x+f_2(w);
t  = f_3(u);
z = t + f_4(s);
```

**WHAT DO THESE LANGUAGE CONSTRUCTS
HAVE TO DO WITH PARALLELISM?**

Single Assignment ==> Data Flow Graph

Streams ==>Overlapping Computations By Data Dependence Joined Nodes

Iteration - Construct Structured Objects



SISAL

**WHAT DO THESE LANGUAGE CONSTRUCTS
HAVE TO DO WITH PARALLELISM?**

Streams
- Pipeline (or vector) parallelism
- A function whose output is a "stream" can begin output before completing execution

Single assignment
- Order of execution is completely determined By data flow
- Independent blocks Of code can be unambiguously identified

Functional
- Local name scopes
- Unambiguous name resolution
- Guaranteed independence Of execution

COMPOUND DATA TYPES

ARRAY TYPE SPECIFICATION

array [type_spec]

type_spec :=basic_type_spec |
compound_type_spec |
type name

type info = array [integer]

type c = array [array [real]]

OPERATIONS ON ARRAYS

create - array real [ ]

set bounds - array_adjust (A, LO, HI)

create by element - array real [J : V]
- create single element array of element J with value set to V

select A[J] - result is value associated with A[J]

replace - A[J : V] - result is a new array with value at A[J] set to V and other values as previous

concatenate - A || B - result is a new array with B conceptually appended to A

set bounds - array_adjunct (A, LO, HI) - result is new array with size [LO, HI]

STREAM type specifications

stream [type_spec]

stream [integer]

stream [record[X : real; Y : real]]

create_stream type_name [ ]

append - stream_append (G, V)
G - stream
V - element of stream
- result is a new stream

select first - stream_first(G) - result is first element of stream

select all but first - stream_rest(G) - result is a stream of all elements except the first element of the previous stream

concatenate - G || H - result is a new stream which is the sequence of G followed by the sequence of values of H

UNION type specification

union [Fix : integer ; FLOAT : real]

i

tag test - is N (u) - result is true or false

Name Scopes

function F (X : integer returns real)
<expression>
end function
X is value name defined within F

let X : real := 3.0 ;
<other declaration/definitions>
in
<expression>

end let

Program Structures

All Structures Produce Values

All Structures May Produce Tuples Of Values

All Structures May Be Embedded In Arithmetic Operations When They Output 1-tuples

Conditional Expression

if <expression> then <expression>

[elseif <expression> then <expression>]

else <expression>

end if

Has Value Of Selected <expression>s

Let Construct

```
let
decldef_part
in
expression
end let

    let
        X : real
        T : real
        T := P + 3.7
        X := T + 2.4
    in
        X * T ; X/T
    end let
```

Has Two-tuple (X * T, X/T) As Value

Significance Is Name Scope Definition

For Construct

product form - produces an array or stream result

non-product form - produces a result based on updates of values. (Watch this carefully!)

Very complex syntax!

Non-Product for

for initial

decldef_part
iterator - terminator
returns return_exp_list
end for

Name scope is local to loop
Any names defined in loop are local to loop

return_exp_list

- value of - converts sequence of values to a single result, which may be a structure
  - reduction operators
    - sum
    - product
    - least
    - greatest
    - catenate
- array of - converts a sequence to an array for return
- stream of - delivers actual sequence (stream) of values

Functional Parallel Programming

```
define Bubble
type onedim = array[  integer  ]
%  Bubblesort A
function Bubble( n : integer ; Ain : onedim returns onedim)
        for initial
                A, limit := Ain, n
        while limit > 1 repeat
                limit := old limit - 1;
                A := for initial
                                j     := 0 ;
                                A1  := old A ;
                             while j < limit repeat
                                j     := old j + 1 ;
                                A1  := if old A1[j] > old A1[j+1] then
                                              old A1[j: old A1[j+1], old A1[j]]
                                          else
                                              old A1
                                          end if
                           returns
                                value of A1
                          end for
        returns
                value of A
        end for
```


Functional Parallel Programming

Product form of for construct

for in_exp_list
[decldef_part]
returns return_exp_list
end for

Primarily a shorthand form

All computations that can be expressed by the product form can also be expressed by the non-product form.

The reverse is not true.

in_exp_list - many forms, (too many!)

- I in 1, 10 dot J in 11, 20
- [1, 11] [2, 12] - - - [10, 20]
- I in 1, 10 cross J in 11, 20
- [I, J] 1 Š I Š 10, 11 Š J Š 20

Functional Parallel Programming

```
define multiply
type OneD = array[double_real] ;
type TwoD = array[OneD] ;
% Multiple A{1. . x, 1. . y] and B{1. .y, 1. .z] .
%
function multiply(x,y,z : integer ; A, B: TwoD returns TwoD)
  for i in 1, x cross j in 1, z
       Cij :=     for k in 1, y
                  returns value of sum A(i, k] * B[k, j]
                  end for
  returns array of Cij
  end for
end function % multiply
NOTES:  "sum" is a reduction operator
```


Functional Parallel Programming

```
% Gauss-Jordan elimination with no pivoting, based on the  routine  given
% in the book "Numerical Recipes-The Art of Scientific Computing" (Press,
% Flannery, Teukolsky, Vetterling) pages 28-29.
%
% A is an input matrix of NxN double_real floating point numbers and B is
% a vector of N double_real floating point numbers.   The program returns
% the vector x in the equation A x = B.  If A is singular, an error value
% is NOT generated.
%

define Main

type double = double_real;
type OneI   = array[integer];
type OneD   = array[double];
type TwoD   = array[OneD];
```

Functional Parallel Programming

```
function Reduce(pivot: integer; A: TwoD; B: OneD
          returns TwoD, OneD)

  for row in A at i
    mult    := A[i, pivot] / A[pivot, pivot];
    rrow, rB := if i = pivot then
                 for x in row returns
                   array of x / A[pivot, pivot]
                 end for,
                 B[i] / A[pivot, pivot]
               else
                 for x in row at j returns
                   array of x - mult * A[pivot, j]
                 end for,
                 B[i] - mult * B[pivot]
               end if
  returns array of rrow
      array of rb
  end for

end function % Reduce
```


Functional Parallel Programming

```
function Main(n: integer; Ain: TwoD; Bin: OneD  returns OneD)

  for initial

    i    := 0;
    A, B := Ain, Bin;

  while i < n repeat

    i    := old i + 1;
    A, B := Reduce(i, old A, old B);

  returns value of B
  end for

end function % Main
```

Functional Parallel Programming

```
% Uses the Sieve of Erotosthenes to compute the prime
% numbers <= Limit
%
define Main
type StrmInt = stream[ integer ];
global sqrt( a : double_real returns double_real )

function Filter( S: StrmInt; M: integer returns StrmInt )
   for I in S returns
       stream of I unless mod( I, M ) = 0
   end for
end function % Filter

function Integers( Limit: integer returns StrmInt )
   for initial
      I := 3;
   while I < Limit - 1 repeat
      I := old I + 2;
   returns stream of I
   end for
```


Functional Parallel Programming

```
function Integers( Limit: integer returns StrmInt )
   for initial
      I := 3;
   while I < Limit - 1 repeat
      I := old I + 2;
   returns stream of I
   end for

end function % Integers

function main( Limit: integer returns StrmInt )
for initial
     T    := 2;
     S    := Integers( Limit );
     Maxt := integer( sqrt( double_real( Limit ) ) )
  repeat
     T := stream_first( old S );
     S := if T <= Maxt then
             Filter( stream_rest( old S ), T )
          else
             stream_rest( old S )
          end if
  until stream_empty( S )
  returns stream of T
  end for
```