SETL — A Parallel Specification Language

SETL — A Language for Prototyping Programs

- Essentials of SETL
- Expressing concurrency in Parallel SETL

Based on

- *Programming with Sets: An Introductionto the SETL Programming Language,* by R.B.K Dewar, E. Dubinsky, E. Schonberg, and J.T. Schwartz, Springer-Verlag, 1986.
- A Set-based language for prototyping parallel algorithms, by R. Hummel, R. Kelly, and S. Flynn Hummel, *Proceedings of CAMP-91*, Paris.

Basic Features of SETL

- **Primitives include sets and tuples**
- **Mathematical syntax**
- **Weakly typed**
- **Dynamic typing**
- **A variable can change type within the program**
- **Rich operator set, overloaded**
- **Complete loop construct**
- **Inefficient**

An Introductory Example

```
program  primes; -- This program prints
            -- out a list of prime numbers which
            -- includes all primes less than a parameter
            -- value which is specified as input data.
read (n); -- read input parameter
primes := {}; -- set of primes
p := 2; -- initial value to test
while p < n loop -- loop as long as p < n
if notexists t in primes | p mod t =0 then
 print (p); -- no divisors, it's a prime
 primes with:= p; -- add it to set of primes
end if;
p := p + 1; -- move to next test value
end loop;
end primes;
```

Basic features

The ' -- ' acts as a comment delimiter

Semicolons ' ; ' teminate statements

Variable types:

- **Integers, reals, booleans, strings**
- **Tuples and sets**

Assignments and expressions:

- **identifier := expression;**
- **identifier binop:= expression;**

  E.g., `x +:= 1;`

  Same as `x := x + 1;`

  In general:

  identifier := identifier binop expression;

Operators on Simple Types

Some binary operators

Binary Operator	*Functions*
+	Addition, string concatination
–	Subtaction
*	Multiplication
/	Division real result
div	Integer division, integer result
mod	Integer remainder
**	Exponentiation
and	Boolean and
or	Boolean or

Unary Operators

type	Returns string
float	Converts integer to real
floor	Floor
ceil	Ceiling
fix	Truncates real to integer
str	String representation of a value
val	Integer or real equivalent of a string
not	Boolean negation
#	Num of chars in a string

Substrings

- abc(4..7) extracts substring
- Can appear on left hand side

om

- Undefined state
- All identifiers are initialized to om

Tuples

- Ordered sequence of zero or more values
- Each element can have its own type
- Tuples can contain tuples, sets, strings, etc.
- Subtuples allowed similar to substrings
- Examples

```
a := [1,2,3];
b := [1,9,"abc","def",a];
c := b(1..3); -- c = [1,9,"abc"]
d := b(4..); -- d = ['def',[1,2,3]]
b(3..) := [ ]; -- b = [1,9]
x := [1..10]; -- x =
                          -- [1,2,3,4,5,6,7,8,9,10]
y := [1,3..11]; -- y = [1,3,5,7,9,11]
```

- Note special form [first, next..last]

Tuple Operators

Binary Operators

Operator	*Function*
+	**Tuple concatination**
with	**x with t appends element x to tuple t**

Unary Operators

Operator	*Function*
#	**Index of highest defined element**
type	**Returns string 'tuple'**

Binary operators with side effects

x fromb a	**Removes first element from tuple a, assigns to x**
x frome a	**Removes last element from tuple a, assigns to x**
a with:= x	**Appends x to tuple a**

- Stacks and queues are easily implemented using fromb, frome, and with.

SETS

- Like a tuple, but unordered
- Repeated values ignored
- Examples:

```
a := {1,2,"abc"};
b := {1,"abc",2};
c := {2,1,"abc",2};  – a = b = c
d := {a,"def",[a,b] };
e := {a,{a}};  – #e = 2
f := { }; f1 = {{ }};  – #f = 0, #f1 = 1
g := {1..10};  – g = {1,2,3,4,5,6,7,8,9,10}
```

Binary set operators

+	Set union
–	Set difference
*	Set intersection
with	Add one element to a set
less	Remove an elt from set
x from s	Remove arbitrary elt from set s and assign to x

Unary set operators

#	Number of elements
type	Returns the string "set"
arb	Select arbitrary element

Maps

- A set whose elements are all tuples of length 2
- Map notation

```
sqrt := { [1,1],[4,2],[9,3],[16,4] }
x := sqrt(9); -- x = 3
sqrt(25) := 5; – Adds [25,5] to set sqrt
```

- Multivalued maps

```
f := {["a",1],["b",2],["c",3],["a",10]};
x := f{"a"}; -- x = {1,10}
y := f{"b"}; -- y = {2}
y1 := f("b");  --  y1 = 2
y2 := f("a"); -- y2 =  om
z := f{"d"}; -- z =  om
f{"d"} := {4,5}; --  Adds ["d",4] and
                     -- ["d",5] to f
```

- Maps are the most important construct in SETL
- Example applications
  - Map from student names to test scores
  - Databases: maps from keys to data

Map operators

Binary Operators

a **lessf** x	Removes pairs for one domain element, i.e., removes all pairs of form [x,?] from a .

Unary map operators

domain	Yields domain of a map
range	Yields range of a map

Maps permit associative access to data

- `firstname("Lincoln") := "Abraham";`
- `x := firstname("Kennedy");`

Atoms

- **Type atom is usually used as a domain element**
- **Created by:**

  ```
  x := newat();
  ```
- **Guaranteed to be unique**

Conditional Statements

The if statement

```
if test then
    statement; statement; statement;
 else
    statement; statement; statement;
 end if;
```

The case statement

```
case
when test1 => block1
when test2 => block2
otherwise => blocke
end case ;
```

There is a *case expression* statement

Conditional blocks (in *if* and *case* statements) can return expressions instead

Tests

Binary test operators

= , /= , < , > , <= , >=	As expected
in	Left operand is an element of right set or tuple
notin	Left operand is not an elt in right set or tuple
subset	Left set is a subset of right set
incs	Left set includes set

Unary test operators

even , odd	integer even or odd test
is_integer , is_real , is_tuple , is_set , is_map , etc.	Type tests

See also quantified tests, below

Loops

Several forms

- **for form**

```
for iterator loop
  block
end loop;
```

- ***Iterators*:**

for `x` **in** `set` **loop**	unordered iteration
for `x` **in** `tuple` **loop**	ordered iteration
for `x` **in** `s	test` **loop**	conditional test
for `x` **in** `primes	x < 100` **loop**	Example
for `y=f(x)` **loop**	Loops for `[x,y]` in `f`, setting both `x` and `y`
loop	Infinite loop

Test forms

while *test* **loop**	loop while test succeeds
until *test* **loop**	end loop if test succeeds at end

Blocks can contain continue or quit

Implicit loops

Quantified tests

- Can be used in place of any test
- Are general expressions returning true or false

```
exists iterator |test
forall iterator |test
if exists x in s | x < 10 then
   return forall x in t |
     exists y in s | x = y;
```

Compound operators

- *binop / set_or_tuple*
- Examples

`+/ s`	Computes sum of elements in s
`+/ t`	Computes sum of elements in t
`*/{a in s	3 in a}`	Computes intersection over all sets in s (a set of sets) that contain 3

Set and tuple formers

- `{ expression : iterator }` — Set former
- `{ expression : iterator}` — Tuple former

- Examples

`{[x**2,x] : x in {1..100}}`	sqrt map for first 100 squares
`{a : a in tuples	a(1) = "Fred"}`	Selected elements in tuples
`{a in tuples	a(1) = "Fred"}`	Abbreviation of above
`[i**2 : i in [1..100]]`	Ordered tuple of 100 squares

Other features

- List-directed I/O
- File I/O
- Declarations for initializing
- Program form, procedures, packages
- Block structured, nested procedures & vbls
- Procedure parameters can be rw , rd , or wr
- Procedure name variables
- Recursion
- Compound types using tuples and selectors
- Command line processing

FTP Availability

SETL2 Interpretter

- ftp cs.nyu.edu
- cd pub/setl2 ; binary
- get appropriate_version
- MS-DOS, Macintosh, Sun3, Sun4, etc.
- Documentation included with distribution

SETL2 Description

- ftp cs.nyu.edu
- cd pub/local/hummel/setl2 ; binary
- get setl2.ps.Z

Parallel SETL paper

- ftp cs.nyu.edu
- cd pub/local/hummel/papers ; binary
- get parsetl.ps.Z

Expressing Parallelism using SETL

Three basic constructs:

- Data (Set) parallelism
- Queue (Farm) parallelism
- Queue parallelism:

Data Parallelism

Parallel iterators

- Keyword **over** or **pairs**
- Not necessarily SIMD
- `for parallel_iterator loop`

  `for x over set parloop`
- May iterate over tuples or strings also

Serializability

- Code is written assuming one or more processors
- Must be correct even if only one processor
- I.e., if *over* is replaced with *in*, still correct

Implicit data parallelism

- Parallel iterator in implicit loop denotes concurrency
- Examples:

  `{x**2 : x over set | T(x)}`

  `exists x over set | condition`

Static Concurrency

MIMD Processing

- `for x over set parloop`
- `for pairs y=f(x) parloop`
- Iterate variables are replicated, so each has separate values
- Semantics:

  Set or map or controling structure is evaluated

  As many processors as are available are assigned independently to iterates

  Loop is repeated until all iterates are completed

  Synchronization at the end, unless —

  End may be: `unsynchonized end loop;`
- Parallel iterators in implicit loops
- Compound operators will generally denote concurrency

SIMD Processing

- Keywords `simd_over` and `simd_pairs`
- Synchronization after every statement

SIMD Iterates

Code

```
•    for x simd_over nodes parloop
•       statement1;  statement2;  ...
•    end parloop;
```

Is equivalent to:

```
•    for x over nodes parloop
•       statement1;
•    end parloop;
•    for x over nodes parloop
•       statement2;
•    end parloop; ...
```

Block statements (if, case)

- Subblocks may execute concurrently
- Synchonization between statements within subblocks

Dynamic Parallelism

Queue Parallism, Farm model, Task Queues

```
•    while queue /=  [ ] parloop
•        x fromb queue;
•        ....
•    end parloop;
```

Semantics

- Queue parallelism is dynamic
- Changes to `queue` are permitted
- Queue may be a tuple, bag, or set
- Available processors extract from queue
- Continue beyond end only when everyone is waiting for more work
- Concurrent inserts and deletes into the queue may take before and during the loop
- withident

  Unique identifier assigned (as with `newat( )`)

  ```
  while queue /= [ ]  parloop
     withident t ;
     x fromb queue; ...
  ```

Pipeline Parallelism

Syntax

- **for** x **pipe** nodes **loop**
- **for pipe** y=f(x) **loop**
- Example:

```
for x pipe tuple loop
    statement1;
    statement2; ...
end loop;
```

Semantics

- Statements operate concurrently
- One instance of each statement for each iterate
- For any given iterate, statements execute in order
- Same as if executed sequentially
- Successive iterates may follow one statement behind previous iterate

Slide operator

Used for pipeline concurrency

- Window of data
- Fixed length fifo queue:
  - t := mknullfq(k);
  - Makes fixed length queue of length k
- t slide:= e ;
  - Appends e to queue t,
  - Pops first element off t if too long

Example code

```
t1 := mknullfq(3); t2 := mknullfq(3); t3
:=mknullfq(2);
out := [ ]
for e  pipe x loop
  localize t1, t2, t3, t4;
  t1 slide:= e;
  if ready(t1) then t2 slide:= ( +/ t1 ) / 3.0 end
if;
  if ready(t2) then t3 slide:= median(t2); end if;
  if ready(t3) then t4 := max/ t2 ; end if;
  if t4 /= om then out with:= e - t4;
end loop;
```

Localized variables

Iterate variables

- Iterate variables are replicated, one per iterate
- For static loops, variables are considered *localized*

Map evaluates on iterate variables

- `for x over set parloop`
- `   f(x) :=  func(x);`
- `... end parloop;`
- Then f(x) is localized, i.e., stored locally to x
- If enclosing parallel loops, then must be constrained by all enclosing loop variables
- For dynamic loops, must be constrained by withident variables

Declared localized

- `localize x;`
- Then x is replicated, as if dependent on all enclosing iterate variables

The Advantage of Maps

All variables are considered shared

- Random access using accesses to other localized map evaluates
- If f(x) is localized, then it will exist after termination of the loop
- Declared localized variables are scoped only to the loop (and embedded loops)
- The set of indices must be stored, so withident variables will need to be inserted into a set

Data parallel control

- Each element in the set of nodes defines a thread of control

Dynamic parallelism

- Each queue item defines a thread
- Localized variables are maps over the withident variable

Atomic Operations

Atomicity not guaranteed

- Within concurrent processing,
- Updating a complex data structure might not be atomic:

```
S := S less x;
```

S might be updated between access and write

Atomicity is guaranteed for certain accesses

- Concurrent readers always allowed
- Concurrent writes are atomic
- Assignment operators

```
x +:= value ;
```

- Concurrent inserts, deletes, unions, etc., into (e.g.) sets, using assignment operations
- Concurrent operations on shared tuples using with, fromb, frome
- Implicit set formers using parallel iterates
- Compound operators (reductions)

```
+/ set
```

Added atomic operations

Fetch and Φ

- Fetch&Op(variable, increment)
- Same as atomically performing:

```
procedure Fetch&Op( variable increment )
   var x ;
   x := variable;
   variable Op:= increment;
   return x;
end Fetch&Op;
```

- Hardware support is common for many Φ's

Parallel Prefix

- Φ `// nodes;`
- E.g., `+ // tuple`
- Equivalent to:

```
[+/tuple(1..i) : i in [1..#tuple ] ]
```

Concurrent assignments

- `[ a , b ] := [ Ptr(x) , Ptr(y) ];`

Comments

Parallel SETL can be executed with simple keyword substitutions

Permits verification of the algorithm

Doesn't exercise the concurrency
I.e., Won't find race conditions

Loops do not guarantee concurrency, but permit concurrent operation

As much parallelism as available

Parallel SETL is a prototyping language, for expressing parallel algorithms,
much as SETL is for expressing and prototyping sequential algorithms