University of Southampton
Electronics and Computer Science

COMP2212 Programming Language Concepts

COBRA USER MANUAL

Contents

1 Introduction

This manual describes COBRA, an interpreted programming language. It is a domain specific programming language that performs certain simple computations on potentially unbounded integer sequences, i.e. Integer Streams. This manual introduces the user to the syntax and features of the COBRA language, giving an outline of how to interpret programs in the language, error messages that may arise, how to write programs and additional features such as type checking.

2 Using the COBRA interpreter

The COBRA interpreter is composed of a lexer, parser and evaluator. The lexer is written in Alex, the parser is written in Happy and the evaluator is written in Haskell. It can be compiled on a Linux machine using the make command, which produces an executable myinterpreter file. Users can run their programs on a Linux terminal by passing the program file and standard input to the interpreter as arguments. The syntax to run the program is as follows

```
myinterpreter program_name < input.txt
```

3 Overview of Syntax

3.1 Data Types

- COBRA has two main data types:

    1. **StreamType** - A structure with the type StreamType represents just a single stream, or a single, finite sequence of integers. The simplest structure with the type StreamType is called a PureStream. A PureStream is named as such to reflect that it is purely a single finite sequence of integers, and not an abstract syntax tree consisting of an operation on one or more expressions which eventually evaluates to a PureStream, even if the entire expression may have the type StreamType.
    An appropriate analogy for this is that, for example, the expressions 8 and 9*7-5 are both of type Int, however, 8 is a pure integer whereas 9*7-5 is not as it consists of operations on expressions, even though it will eventually evaluate to a pure integer (58) and hence has type Int. In the same way, in this language, a PureStream has type StreamType, but there can be other structures which aren't a PureStream which have type StreamType because they eventually evaluate to a PureStream.
    2. **MultiStreamType** - A structure with the type MultiStreamType represents a collection of one or more streams of finite integers in a specified order. The simplest structure of type MultiStreamType is a PureMultiStream, which is just a collection of one or more PureStreams. Just like for StreamType, there are more complex structures which are not a PureMultiStream, which have type MultiStreamType because it eventually evaluates to a PureMultiStream.

3.2 Syntax Constructs*

1. empty_stm – this represents an empty stream
2. empty – this represents a multi-stream (a list of single streams), but currently has no streams in it
3. <: – an operator which puts an integer at the front of a stream to give a new longer stream
4. +++ – an operator which places a stream into a multi-stream as the first stream in the multi-stream, giving a new longer multi-stream
5. M+ – an operator which appends a second multi-stream to a first multi-stream to create a larger multi-stream
6. S+ – an operator which combines two single streams into a multi-stream
7. :: – an operator which places a single stream at the front of a multi-stream, to give a multi-stream
8. +-> – an operator which puts an integer to the front of a stream (this also deletes the last integer which was originally in the stream) to give a new stream
9. <-+ – an operator which puts an integer to the back of a stream (this also deletes the first integer which was originally in the stream) to give a new stream
10. access_stm – an operator which returns the stream corresponding to a particular index in a multi-stream
11. Operator – an 'Operator' has possible values of +, -, *, /, % and ^
12. single_func – performs the same function on every integer in a stream, the function is specified by the integer and 'Operator' supplied
13. double_func – this function takes two streams, multiplies every element in each stream by a specified coefficient (integer) for that stream, then combines the two streams in the way specified by the 'Operator' given to give one stream
14. Sequences – This is a structure used to represent a general sequence of numbers (not a stream).
    Sequences have the following possible forms:
    (a) $[n] = i – represents a sequence i, i, i, i, ... where i is an integer
    (b) $[0] = i, $[1] = j, $[n] = $[n-2] *op* $[n-1] – represents a sequence where i and j are integers representing the first and second values of the sequence, and every subsequent nth value is the result of an operation on the (n-2)th and (n-1)th value, specified by the 'Operator' *op*
    (c) $[0] = i, $[n] = j * $[n-1] – represents a sequence where i is an integer representing the first value in the sequence, and j is an integer representing the value used to multiply each number in the sequence to get the next number in the sequence
    (d) $[0] = i, $[n] = j + $[n-1] – represents a sequence where i is an integer representing the first value in the sequence, and j is an integer representing the value added to each number in the sequence to get the next number in the sequence

15. accumulate – this is a function which takes a stream and a sequence and returns a stream whose integers are an accumulation of the integers in the original stream, with the coefficients used for the integers being accumulated derived from the sequence supplied to the function

16. Let blocks

    - let (x : *ty*) = *exp1* in *exp2* – represents a normal let block where x is a variable String, *ty* is the type of *x* (StreamType or MultiStreamType), *exp1* is an expression which can be of StreamType or MultiStreamType and *exp2* is an expression that may contain the free variable *x*

17. Input

    - There is a keyword "inp" which is used as a placeholder for the input to the program. When it is evaluated, it is simply replaced with the input which was read in for that program. The input for all programs will be of the form of a PureMultiStream.

18. Brackets

    - Finally, the COBRA language includes bracketing, which can be used to give certain smaller expressions within larger expressions higher precedence. Bracketing is used in its conventional way.

4 Additional Features*

4.1 Comments

To increase understanding of code, comments can be added to the code to explain what certain blocks of a program do. The interpreter is designed to ignore all words after "//".

4.2 Informative Error Messages

The lexer written in Alex uses the posn wrapper, so in case of any parse errors, the line and column position of the parse error are shown to the user. On unparsing, if the expression to be unparsed is not a PureStream or PureMultiStream, it cannot be unparsed, and an error is outputted which says so.

4.3 Type Checking

The COBRA language contains a type checker, which is run on every program before evaluation of the program begins. When it detects a type error, it outputs an error which describes which language structure is facing the type error, which argument(s) have the wrong type, which type(s) the argument(s) have and which type(s) the argument(s) should have.

4.4 Illegal Input

The COBRA language contains an illegal input checker, which is run on every input before it is supplied to the program. When it detects an illegal character in the input (for example, a symbol or an alphabet), it outputs an error along with the illegal character.

*Examples are included in the Appendix

5 Appendix

5.1 Programs

5.1.1 Program 1

```
let (x:StreamType) = access_stm 0 inp in ((0 +-> x) :: empty)
```

5.1.2 Program 2

```
let (x:StreamType) = (access_stm 0 inp) in (x S+ x)
```

5.1.3 Program 3

```
let (x:StreamType) = (access_stm 0 inp) in (let (y:StreamType
    ) = (access_stm 1 inp) in ((double_func: 1*x + 3*y)) ::
    empty)
```

5.1.4 Program 4

```
let (x:StreamType) = (access_stm 0 inp) in ((accumulate x:
    coeffs <- ($[n] = 1)) :: empty)
```

5.1.5 Program 5

```
let (x:StreamType) = (access_stm 0 inp) in ((accumulate x:
    coeffs <- ($[0]=1,$[1]=1,$[n]=$[n-2]+$[n-1])) :: empty)
```

5.1.6 Program 6

```
let (x:StreamType) = access_stm 0 inp in (x S+ ((0 +-> x)))
```

5.1.7 Program 7

```
let (x:StreamType) = (access_stm 0 inp) in (let (y:StreamType
    ) = (access_stm 1 inp) in ((double_func: 1*x - 1*y)) S+ x)
```

5.1.8 Program 8

```
let (x:StreamType) = access_stm 0 inp in ((double_func : 1 *
    x + 1 * (0 +-> x)) :: empty)
```

5.1.9 Program 9

```
let (x:StreamType) = (access_stm 0 inp) in ((accumulate x :
    coeffs <- ($[0] = 1, $[n] = 1 + $[n-1])) :: empty)
```

5.1.10 Program 10

```
let (x:StreamType) = (access_stm 0 inp) in ((accumulate x :
    coeffs <- ($[0] = 1, $[1] = 0, $[n] = $[n-2] ^ $[n-1])) ::
     empty)
```

5.2 Syntax Usage Examples

1. empty_stm and <:

```
0 <: 1 <: 2 <: 3 <: empty_stm
```

This will evaluate to:

```
0
1
2
3
```

2. empty and +++

```
(0 <: 1 <: 2 <: 3 <: empty_stm) +++ (4 <: 5 <:  6 <: 7 <:
    empty_stm) +++ empty
```

This will evaluate to:

```
0 4
1 5
2 6
3 7
```

3. M+

```
((0 <: 1 <: 2 <: 3 <: empty_stm) +++ empty) M+ ((4 <: 5
   <: 6 <: 7 <: empty_stm) +++ empty)
```

This will evaluate to:

```
0 4
1 5
2 6
3 7
```

4. S+

```
(0 <: 1 <: 2 <: 3 <: empty_stm) S+ (4 <: 5 <: 6 <: 7 <:
   empty_stm)
```

This will evaluate to:

```
0 4
1 5
2 6
3 7
```

5. :: – The distinction between :: and +++ is that +++ is only used to add a PureStream to a PureMultiStream, whereas :: can be used to add non-pure expressions of StreamType to a non-pure expression of MultiStreamType

```
(0 <: 1 <: 2 <: 3 <: empty_stm) :: ((4 <: 5 <: 6 <: 7 <:
   empty_stm) +++ empty)
```

This will evaluate to:

```
0 4
1 5
2 6
3 7
```

6. +->

```
0 +-> (1 <: 2 <: 3 <: 4 <: empty_stm)
```

This will evaluate to:

```
0
1
2
3
```

7. <-+

```
0 <-+ (1 <: 2 <: 3 <: 4 <: empty_stm)
```

This will evaluate to:

```
2
3
4
0
```

8. access_stm

```
access_stm 1 ((0 <: 1 : empty_stm) S+ (2 <: 3 <:
   empty_stm))
```

This will evaluate to:

```
2
3
```

9. single_func – The syntax takes the form "single_func : *s op i*" where *s* is the stream, *op* is the 'Operator' and *i* is an integer

```
single_func : (0 <: 1 <: 2 <: empty_stm) + 5
```

This will evaluate to:

```
5     // as 0+5=5
6     // as 1+5=6
7     // as 2+5=7
```

10. double_func – The syntax takes the form "double_func : *i * s op j * t*", where *i* and *j* are integers, *s* and *t* are streams and *op* is an 'Operator'

```
double_func : 5 * (0 <: 1 <: 2 <: empty_stm) + 3 * (3 <:
   4 <: 5 <: empty_stm)
```

This will evaluate to:

```
9          // as 5*0 + 3*3 = 9
17        // as 5*1 + 3*4 = 17
25        // as 5*2 + 3*5 = 25
```

11. Sequences and accumulate – They are used when an accumulate function is used on a stream: accumulate *s* : coeffs <- *seq* – represents an accumulate function on a stream where *s* is a stream with type StreamType (so is either a PureStream or will evaluate fully to a PureStream) and *seq* is a sequence given in the form described above. The accumulate function takes a stream and returns a stream where in every position, all the previous integers and the integer at that position in the original stream is multiplied by a coefficient, which is derived from the sequence provided with the accumulate function. The resultant values are then added up to give the value at that position in the resulting stream.

```
accumulate (1 <: 2 <: 3 <: empty_stm) : coeffs <- ($[0] =
    1, $[n] = 3 * $[n-1])
```

This will evaluate to:

```
1        // as 1*1 = 1
5        // as 3*1 + 1*2 = 5
18      // as 9*1 + 3*2 + 1*3 = 18
```

12. Let blocks

```
let (x : StreamType) = (1 <: 2 <: 3 <: empty_stm) in 0 +-
   > x
```

This will evaluate to:

```
0
1
2
```

13. Input

```
let (x : MultiStreamType) = inp in x
```

Assume the input for the above program is:

```
2 5
3 5
6 8
```

This will evaluate to:

```
2 5
3 5
6 8
```

5.3 Additional Features Examples

1. **Comments**
   For example:

```
let (x:StreamType) = access_stm 0 inp in ((0 +-> x) ::
   empty)  // Add 0 to the beginning of integer stream
```

2. **Type Checker**

```
(0 <: 1 <: 2 <: empty_stm) :: (3 <: 4 <: 5 <: empty_stm)
```

The error message output:

```
Type error related to the arguments to StreamCons.
Specifically, the second argument PureS (PartOf 3 (PartOf
    4 (PartOf 5 EmptyStream))).
This should have type MultiStreamType but instead has
   type StreamType.
```

3. **Illegal Input**
   For example, on input:

```
1
2
a
4
```

   The error message output:

```
Illegal alphabet in input: 'a'
CallStack (from HasCallStack):
  error, called at myinterpreter.hs:54:35 in main:Main
```