YASPL - Yet Another Stream Programming Language

Hieu Trung Luu - thl1g15 , Sam Jones - sj2g17

May 2019

1 Introduction

YASPL is a domain specific programming language built for processing streams of integers. Programs written in YASPL take as input an arbitrary number of streams, which each contain an equal number of integers. The program will then output to stdout one or more streams which are the same length as the input streams.

2 Syntax

YASPL programs are divided into blocks. Each block is composed of the block name followed by the contents of the block enclosed in curly braces. The first block is the "start" block which is executed before the input starts being read. This is used for initialisation purposes. All subsequent blocks are named according to which lines of the input they should be executed on. They can be assigned a single line (X), an inclusive range of lines (X-Y), or a starting line (X+), in which case the block is executed on that line and all lines after that until the end of input. For example, a block named '4-6' would execute on lines 4, 5 and 6, and a block named '7+' would execute on line 7 and then on every line until after that. It is important to note that line numbering starts at 0. For a simple program in which the same code should be executed on every line, the only block necessary would be '0+'. Examples of blocks can be seen in listing 1 and 2 below.

Listing 1: example of start and 0+

```
start{
  buffer = 0;
}

0+{
  return buffer;
  buffer = $0;
}
```

Listing 2: example of 0-1 and 1+ (print first element of input stream

```
0-1{
  return $0;

}
1+{
  return 1;
}
```

Each block contains some number of lines of code, with each line being either an assignment/reassignment of a variable or a return statement, and ending in a semicolon. Before I talk about the structure of these, I will explain what makes a valid expression in YASPL as these are the core building blocks of the language.
Both assignments and return statements are built using expressions, which are the core building

blocks of the language. Section B of the Appendices details the syntax of expressions. YASPL expressions are mostly similar to Haskell expressions, so those familiar with Haskell should have no trouble understanding the syntax.

Listing 3: Example of lambda expression. The function is used in another block outside of start block. This example also shows the different scopes of the program (scope in lambda expression and global scope)

```
start{
  x = 100;
  a = \(x : int) -> (x==1);
}
0+{
  pred = a $0;
  out = if pred then 1 else 0;
  return out;
}
// if the input is a stream of
    1,2,3,4,5, output is a stream of
    1,0,0,0,0
```

Listing 4: example of list comprehension expression and built in function used to model Fibonacci sequence in problem 5

```
start{
  fib = [1, 1];
  seen = [];
}
0-1{
  return $0;
  seen = [$0];
}
0+{
  seen = seen++[$0];
  test = reverse fib;
  x = zip seen test;
  x = {(fst z) * (snd z) | z <- x};
  r = sum x;
  return r;
  a = last fib;
  b = head (tail (reverse fib));
  fib = fib++[(a+b)];
}
```

The return statement is responsible for output. It should be followed by the elements which should be added to the output streams, separated by spaces. For example, to output variable X in output stream 1 and variable Y in output stream 2, one would write 'return X Y'. The start block cannot contain a return statement, as there can only be output when there is input. Every other block must contain exactly one return statement. and all return statements in a program must be given the same number of arguments. The arguments of a return statement do not have to be variables, they can be any valid expression. For example, 'return X*2' is perfectly valid.

Assignments and reassignments are where most of the logic in a YASPL program takes places. A basic assignment is a variable name. followed by '=', followed by some expression. For example, "x = 10". This assigns the value of 10 to the variable x. If x had been assigned previously, then it is reassigned. Once assigned, a variable can be used in any expression. Aside from =, there are other reassignment operators available for utility. These include +=, -=, *= and /=. These are fairly standard operators which should be familiar from other languages. For reference, 'x += 1' is equivalent to 'x = x + 1', and the other operators mentioned work similarly.

Listing 5: example of return multiple stream in pr2.spl

```
0+{
  return $0 $0;
}
```

Listing 6: example of returning a stream which every element in the output is the double of it's element in the input)

```
0+{
  x = $0;
  return x*2;
}
```

3 Additional Features

3.1 Type system

YASPL has two primitive data types, Int and Bool, as well as two data structures, List and Pair. Lists contain any number of elements of the same type, and Pairs contain exactly two elements of potentially different types. These structures can be nested, for example a List of Pairs of Lists of Ints is valid.
In addition to these, functional types (e.g. Int -> Bool, a function from Int to Bool) are supported, and is the type a variable will have if it is assigned a lambda abstraction. Variables used on the left hand side of a function application must have a functional type, and the arguments must be of the type accepted by that function.
A type checker is built into the interpreter, which will not allow a program to execute if it contains type errors. This throws reasonably informative errors, such as 'Error: Int '+' Bool is not defined'.

3.2 Library Functions

There are a number of predefined haskell-like functions built into the YASPL interpreter. These are common functions for dealing with lists and pairs, and can serve as building blocks for writing more complex functions using lambda abstraction. The functions built in to the language include zip, reverse, sum, product, and various others. An example of using built-in functions can be seen in listing 4. The full list of built-in functions can be found in Appendix C.

3.3 Scope

The program has one global scope for every blocks. However the scope of lambda expression and list comprehension is separated from the global scope. Variables used in the lambda expression and list comprehension that have the same name with variables in the global scope will not use the value declared in global scope (Example program can be found at figure of Listing 3). Hence, the scope in YASPL can help user improve the speed of naming variables while programming.

3.4 User experience

Single line comments are supported by writing '//' at the start of the line. Nothing on this line will be processed by the interpreter.
Various types of error will be thrown to help the programmer with debugging. Type errors are thrown before program execution, and describe exactly what the type mismatch causing the problem is. Lexing and parsing errors will also be thrown if there are syntax problems, and these give a description of the error and the location of it in the code.
There are various mechanisms in YASPL which help keep programs concise. These the useful reassignment operators +=, -=, *= and /=; library functions which would otherwise need to be re-declared in many programs; syntactic sugar which allows a list to be written as [1, 2, 3] as opposed to 1:2:3:[]; and list comprehension, which is extremely useful in a wide range of

scenarios.

Appendices

A Problems 1 to 10

A.1 Problem 1

```
start{
  buffer = 0;
}

0+{
  return buffer;
  buffer = $0;
}
```

A.2 Problem 2

```
0+{
  return $0 $0;
}
```

A.3 Problem 3

```
0+{
  return ($0+($1*3));
}
```

A.4 Problem 4

```
start{
  acc = 0;
}

0+{
  acc += $0;
  return acc;
```

```
}
```

A.5 Problem 5

```
start{
  fib = [1, 1];
  seen = [];

}
0-1{
  return $0;
  seen = [$0];
}

0+{
  seen = seen++[$0];
  test = reverse fib;
  x = zip seen test;
  x = {(fst z) * (snd z) | z <- x};
  r = sum x;
  return r;
  a = last fib;
  b = head (tail (reverse fib));
  fib = fib++[(a+b)];
}
```

A.6 Problem 6

```
start{
  buffer = 0;
}

0+{
  return $0 buffer;
  buffer = $0;
}
```

A.7 Problem 7

```
0+{
  return ($0 - $1) $0;
}
```

A.8 Problem 8

```
start{
  buffer = 0;
}

0+{
  return (buffer + $0);
  buffer = $0;
}
```

A.9 Problem 9

```
start{
  seen = [];
  index = [];

}
0-1{
  return $0;
  seen = [$0];
  index = [1];
}

1+{
  seen = seen++[$0];
  buffer = (last index) +1;
  index = index ++ [buffer];

  x = zip seen (reverse index);
  x = {(fst z) * (snd z) | z <- x};
  r = sum x;
  return r;
}
```

A.10 Problem 10

```
start{
  lastTwo = [];
}

1-2{
  return $0;
  lastTwo = lastTwo++[$0];
}

2+{
  out = $0 + head lastTwo;
```

```
  lastTwo = [last lastTwo, out];
  return out;
}
```

B List of valid expressions

- Integer - A primitive integer
- Boolean - 'true' or 'false'
- Variable reference - A variable name evaluates to the value of that variable so long as the variable has been assigned.
- Arithmetic - Basic arithmetic expressions using the operators +, -, *, /, % (modulo), ^(exponent)
- Comparison - Comparison of integers using operators <, >, <=, >=, ==, !=
- Boolean logic - A Boolean logic expression using operators && (and), || (or)
- Pair - A pair of expressions in the form (A, B) where A and B are 2 expressions of any type.
- List - A list of expressions in the form [e1, e2.. en] where e1-en are expressions of the same type.
- Lambda Abstraction - A lambda calculus abstraction, where lambda is replaced with '\' and a type must be declared. The general form is \(x : T) -> E, where x is new variable name, T is a type and E is an expression. This is essentially a function which takes an argument of type T and returns the expression E. We will refer to lambda abstractions as functions.
- Function application - A function followed by one or more arguments, separated by spaces, such as 'f x', where f is a variable defined to be a function and x is some expression.
- List constructor - Adds an element to the head of a list. Has the form 'e:L', where e is an element and L is a list.
- List append - Appends one list to the end of another, has the form 'L1++L2' where L1 and L2 are lists.
- If statement - Basic conditional expression, has the form 'if B then e1 else e2', where B is a boolean expression and e1 and e2 are some other expressions of the same type. Evaluates to e1 if B is true, or e2 otherwise.
- Ident - This is how the input streams are accessed. During any block other than start, the program will have one line of the input loaded. If this is line 0, then this corresponds to element 0 of each of the input streams, likewise for other lines. An ident, written '$X' where X is a natural number, references the currently loaded element in input stream X.
- List comprehension - Iteratively compiles a list based on predicates. Written as '{ E | p1, p2.. pn }', where p1-pn are predicates, which I will explain shortly, and E is an expression

which represents what will be added to the list. A predicate can either be a membership declaration or a property. A membership declaration looks like 'x <- L' where x is a new variable name and L is a list. List L will be iterated over, and at each iteration x will reference the current element. A property is any boolean expression, and only on iterations where this holds property holds will the expression E be added to the list. One example usage of list comprehension would be 'x | x<-l, x>5'. This would evaluate to the list of all elements in l which are greater than 5. Another would be 'x*2 | x <- l', which would be the list of every element in l doubled.

C

List of built in function

- zip - Combines two lists into a list of pairs
- reverse - Reverse a list
- head - Take the first element of a list
- tail - Take all but the first element of a list
- last - Take the last element of a list
- init - Take all but the last element of a list
- fst - Take the first element of a pair
- snd - Take the second element of a pair
- sum - Calculate the sum of all elements in a list
- product - Calculate the product of all elements in a list
- length - Take the number of elements in a list
- elem - Return true if an element is in a list, or false otherwise
- take - Take a specified number of elements from a list
- drop - Drop a specified number of elements from the start of a list.