Data Analyzing for SetlX

Ahmad Aboulazm

June 24, 2013

Contents

Abstract

As SETLX is an untyped language, type errors are only caught at runtime. This is the same as in untyped languages like *Prolog* or *Python*. However, current *Prolog* implementations check for *singleton variables*, i.e. variables that are used only once. The reason is that many typos result in singleton variables. Therefore, checking singleton variables can uncover mistyped variable names. Our intention is to implement similar static analysis techniques for SETLX. In contrast to *Prolog* programs, where a predicate can both read a variable and define it, the direction of the data flow in SETLX programs is well defined. Therefore, it is possible to implement a static analysis for SETLX programs that is more precise than the corresponding analysis for *Prolog*: In particular, two different checks are possible.

1. We can check whether a variable is defined before it is used. This is commonly referred to as definite assignment analysis.
2. We can check whether a variable is read after it is written. If a variable is written but never read afterwards, the assignement to this variable is useless. This kind of data flow analysis is known as live variable analysis.

Along with some other checks such as

1. Checking if any used procedure is defined in the code or not.
2. Giving a warning in classes in case the user defined a new variable with the same name of a class variable due to not adding a prefix 'this.' before the variable's name.

and some other small checks which will be discussed later on. But first lets start by giving a brief introduction about SETLX

1 Introduction to setlX

New programming languages are being proposed every year, inventors are always asked the same question. Why is a new programming language when there are a lot others ? The answer is always that it consists of an economical, theological and practical arguments. The first claim has been discussed before thus there is no reason to repeat it. As for the second claim, it also won't be discussed here for the fact that the average computer scientist won't have the philosophical background to follow theological discussions. Nevertheless, throughout the examples that will be given, it is hoped that they will convince the reader with the third claim. One main reason for that is that SETLX programs are readable and concise which makes them very useful when writing complex algorithms, and also that it is very close to pseudocode but instead the user has actual running program.

In the following sub-chapters we are going to introduce the most important features of this language.

If you wish to see the official full tutorial and instructions to download SETLX kindly visit this following link :

`http://wwwlehre.dhbw-stuttgart.de/~stroetma/SetlX/setlX.php`

1.1 SetlX Data Types

SETLX is an interpreter that can be used to calculate simple calculations. For example by typing

```
=> 1/3 + 2/5;
```

and hitting return yields to this response

```
~< Result:  11/15 >~
```

This example shows that SETLX actually supports rational numbers, it also makes sure that the answer is in the simplest form. So after typing

```
=> 1/3 + 2/3;
```

The response would be

```
~< Result:  1 >~
```

as in this case, the simplest form of the result will have a denominator of 1, thus SETLX only prints the nominator.

The precision of any result computed in SETLX is unlimited unless there is no more memory space to take it, so if we compute the factorial of 50

```
=> 50!;
```

The result would be

```
~< Result:  30414093201713378043612608166064768844377641568960512000000000000 >~
```

To calculate floating point values in SETLX the simplest way is to add 0.0 to the calculated expression as in

```
=> 1/3 + 2/5 + 0.0;
```

which will yield to a result in this case as shown

```
~< Result:  0.7333333333333333 >~
```

To create a string in SETLX, you just need to put some characters between double quotations. Which makes the *hello world program* in SETLX's interactive mode as simple as

```
=> "Hello world!";
```

which outputs

```
~< Result:  "Hello world!" >~
```

To assign a value to a variable in SETLX, we use the ':=' operator which is the major common thing between SETLX and the programming language *C* so writing

```
=> "x := 2";
```

assigns the value '2' to the variable 'x'. Always keep in mind that in SETLX variables always have to start with a lower case.

SETLX provides some operators to create boolean expressions. Which are

1. `&&` as the logical and.
2. `||` as the logical or.
3. `!` as the logical not.

4. `=>` as the logical implication.

5. `<==>` as the logical equivalence which can be also replaced with `==`.

6. `<!=>` as the logical antivalence which can be also replaced with `!=`.

It also supports the universal and existential quantifiers *"forall"* and *"exists"*. So in order to evaluate the formula

$$\forall x \in \{1, \cdots, 10\} : x^2 \leq 2^x$$

we simply write

```
forall (x in {1..10} | x ** 2 <= 2 ** x);
```

and to evaluate

$$\exists x \in \{1, \cdots, 10\} : 2^x < x^2$$

we simply write

```
exists (x in {1..10} | 2 ** x < x ** 2);
```

The most interesting data type in SETLX is the *set* type. You can create a *set* by writing

```
=> {1, 2, 3};
```

which is exactly the same as writing

```
=> {2, 1, 3};
```

as order doesn't matter in *sets*. SETLX also provides other convenient ways of creating sets, for example writing

```
=> {1..15};
```

will create a *set* containing all elements counting from 1 till 15 with a step of 1. While writing

```
=> {a,b..c};
```

will create a *set* containing all elements counting from a till c with a step of $b - a$. The same also applies for descending orders.
There are some basic operators on sets which are

1. `"+"` to compute the union of 2 sets.

2. `"*"` to compute the intersection of 2 sets.

3. `"-"` to compute the difference of 2 sets.

4. `"><"` to compute the cartesian product of 2 sets.

5. `"** 2"` to compute the cartesian product of a set with itself.

6. `"2 **"` to compute the power set of a set.

7. `"%"` to compute the symmetric difference of 2 sets.

Another interesting thing about *sets* in SETLX is set comprehension which is used to build sets which has the general formula of

{ *expr* : x_1 `in` s_1, $\cdots$, x_n `in` s_n | *cond* }.

where *expr* is an expression that contains x_1, $\cdots$, x_n in it which are defined afterwards by s_1, $\cdots$, s_n while *cond* is an extra optional condition if needed. For example

```
{ a * b :  a in { 1 ..  3 }, b in { 1 ..  3 } };
```

computes the set

{1, 2, 3, 4, 6, 9}.

This is actually very powerful, as by very simple code like

```
s := {2..100};
s - { p * q :  p in s, q in s };
```

yields to an output containing all prime numbers between 1 and 100. SETLX also supports some set functions which make things easier like

`first(`s`)`

which returns the *first* element of the set s, and

`last(`s`)`

which returns the *last* element of the set s.

SETLX also supports lists which have 2 main differences with sets which are :

1. A syntactical difference : the curly braces "{" and "}" of sets are substituted with the square brackets "`[`" and "`]`" for lists.
2. A logical difference : Lists are an ordered collection of elements which can contain an element more than once unlike sets.

Away from these two points, lists are almost the same as sets and they support everything mentioned above concerning sets.

Alike SETL the old version of SETLX, SETLX pairs are supported. A pair is represented in SETLX as a list of 2. A binary relation can be represented as a set of pairs. So if we can consider r as a binary relation, then we have a domain and a range represented in the formulas

`domain(r) = { x :[x,y] in r }` and `range(r) = { y :[x,y] in r }`.

Furthermore, binary relations can be used as a map. In that case if r is the binary relation, we have a definition for $r[x]$

$$r[x] := \begin{cases} y & \text{if the set } \{y \mid [x,y] \in r\} \text{ contains exactly one element } y; \\ \Omega & \text{otherwise.} \end{cases}$$

Binary relations seem very helpful that they actually can be used as functions, but that won't be smart as that would consume a lot of memory and will compute even cases that aren't needed. Thus in SETLX *procedures* are used for that. Figure 1 for example defines a procedure that computes prime numbers starting 2 up to a given number n.

```
1  primes := procedure(n) {
2      s := { 2..n };
3      return s - { p*q : p in s, q in s };
4  };
```

Figure 1: A procedure to compute the prime numbers.

In figure 1, the block starts with assigning "`procedure(n) {`" to a variable `primes` which is the name of the procedure where n in this case is the parameter given to this *procedure*, then the procedure's block ends with a closing bracket "}" followed by a semi column `';'`. The procedure's block itself contains the logic that the procedure has to perform.
In SETLX procedures can be

1. assigned to a variable.
2. used as an argument to another procedure.
3. be returned from other procedures.

more about procedures will be discussed later on.

Strings in SETLX are a sequence of characters enclosed by double quotes. Here is a list of functions that can be applied to *Strings* in SETLX :

1. `s1 + s2`: $'+'$ to concatenate strings $s1$ and $s2$.
2. `s * n` : $'*'$ to concatenate multiple instances of the string s n times where n is a natural number.
3. `s[i]` : to get the i_{th} character of string s.

SETLX also provides *string interpolation* where any string containing substring of it enclosed between two "`$`" signs, SETLX evaluates the expression between them then substitute the result of it into the string. For example, if n has the value of 6

```
print("$n$!  = $n!$");
```

will actually print

```
6!  = 720.
```

If you wish to turn off such kinds of processing, you just have to use single quotes instead of double quotes. For example

```
print('$n$!  = $n!$');
```

will actually print

```
$n$!  = $n!$.
```

SETLX provides also *first order terms* similar to the one provided by the programming language *Prolog*. Terms consists of *functors* and *arguments*. The difference between a *functor* and a *function* is that *functors* start with a capital letter and doesn't evaluate anything. An example for using *Terms* is implementing *ordered binary trees*.

1. An empty tree is represented as

   `Nil`()

2. A non-empty tree has three components

   (a) The *root* node.

   (b) A left subtree.

   (c) A right subtree.

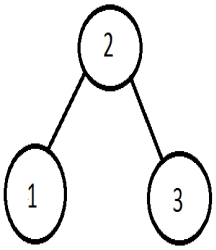
   represented as

   `Node`(k, l, r),

   where k is the element stored at the root, l is the left subtree and r is the right subtree. For example the term

   ```
   Node(2,Node(1,Nil(),Nil), Node(3,Nil(), Nil()))
   ```

   is actually representing the tree that looks as follows



SETLX supports 3 main functions for terms.

1. `fct(t)` which returns the functor of a term t.
2. `args(t)` which returns the arguments of a term t.
3. `makeTerm(f,l)` which creates a term of functor f and arguments l.

So executing `fct(Node(3,Nil(),Nil()))` yields a result of `"Node"`,
and executing `args(Node(3,Nil(),Nil()))` yields a result of `[3, Nil(), Nil()]`,
and executing `makeTerm("Node",[ makeTerm("Nil",[]), makeTerm("Nil",[]) ])` constructs the term `Node(3, Nil(), Nil())`.

Figure 2 actually shows how terms can be used in SETLX to implement binary trees.

```
insert := procedure(m, k1) {
    switch {
        case fct(m) == "Nil" :
             return Node(k1, Nil(), Nil());
        case fct(m) == "Node":
             [ k2, l, r ] := args(m);
             if (k1 == k2) {
                 return Node(k1, l, r);
             } else if (compare(k1, k2) < 0) {
                 return Node(k2, insert(l, k1), r);
             } else {
                 return Node(k2, l, insert(r, k1));
             }
    }
};
```

Figure 2: Inserting an element into a binary tree.

1.2 Statements in setlX

In this subsection we are going to talk about different kinds of statements in SETLX. The most basic statements are the assignment statements. In SETLX, you can have a single assignment like

```
x := 2;
```

which assigns the variable x to the value of 2, Chained assignments can also be used as

```
a := b := 2;
```

which assigns both variables a and b to the value of 2, you can also have simultaneous assignments at the same time like

```
[x, y, z] := [1, 2, 3];
```

which assigns variable x to a value of 1, variable y to a value of 2 and variable z to a value of 3. Simultaneous assignment can also be useful when swapping values is needed. For example, the statement

```
[x, y] := [y, x];
```

actually swaps the values between x and y.

Functions and their structure have been discussed before but lets see some more examples about them. Looking at figure 3, the first function 'factors' is a function that computes the factors of a given number while the second one 'primes' make use of the function 'factors' to compute all prime numbers upto a given number p. We can give a grammar rule for defining a function as

```
fctDef -> VAR ":=" "procedure" "(" paramList ")" "" block "" ";"
```

where the symbols in this grammar means the following

1. `VAR` is a variable which will be used as the name of the function further on.

2. `paramList` is the list of parameters used in the function separated by commas. Parameters can just be a variable name or a variable name preceded by "rw" which means that this variable is a "read-write" variable and after the procedure is executed, that variable's value might actually be changed.

3. `block` is the code holding the logic of the function itself.

```
1  factors := procedure(p) {
2      return { f in { 1 .. p } | p % f == 0 };
3  };
4  primes := procedure(n) {
5      return { p in { 2 .. n } | factors(p) == { 1, p } };
6  };
7  print(primes(100));
```

Figure 3: A naive program to compute primes.

There is actually a simpler way to define a procedure in SETLX if the procedure's logic is a single expression, which is called the lambda definition and has the grammar rule

```
fctDef -> ID ":=" "lambdaParams" "|->" exp ";"
```

where `lambdaParams` is either a single parameter or a set of parameters enclosed by square brackets. An example of using the lambda definition is

```
double := x > x*2;
```

SETLX supports *if-then-else statements*, *switch statements* and *match statements*. An example of the use of the *if-then-else statements* is shown in figure 4 while an example for the use of the *switch statements* is shown in figure 5.

```
1  toBin := procedure(n) {
2      if (n < 2) {
3          return str(n);
4      } else {
5          r := n % 2;
6          n := floor(n / 2);
7          return toBin(n) + toBin(r);
8      }
9  };
```

Figure 4: A function to compute the binary representation of a natural number.

```
1  sort3 := procedure(l) {
2      [ x, y, z ] := l;
3      if (x <= y) {
4          if (y <= z) {
5              return [ x, y, z ];
6          } else if (x <= z) {
7              return [ x, z, y ];
8          } else {
9              return [ z, x, y ];
10         }
11     } else if (z <= y) {
12         return [z, y, x];
13     } else if (x <= z) {
14         return [ y, x, z ];
15     } else {
16         return [ y, z, x ];
17     }
18 };
```

Figure 5: A function to sort a list of three elements.

As for match statements, there are different types of matching statements. One kind is called *string-matching*. One example for it is shown in figure 6

```
1  reverse := procedure(s) {
2      match (s) {
3          case []   : return s;
4          case [c|r]: return reverse(r) + c;
5          default   : abort("type error in reverse($s$)");
6      }
7  };
```

Figure 6: A function that reverses a string.

calling `reverse("abc");` will yield an output of "cba". There are other kinds of matching like *list-matching*, *set-matching* and *term-matching*. Term-Matching is the most elaborate form of matching, it's similar to the matching provided in programming languages *Prolog* and ML. An example for *term-matching* is shown in figure 7 which does the same work as the code shown in figure 2 discussed previously.

A more complex example is shown in figure 8 which computes the derivative of t relative to x. In order to understand this example better, we have to discuss some predefined functions which convert strings to terms which are the *canonical* and the *parse* functions. These functions are going to be discussed in the last sub-chapter entitled *predefined functions*.

```
1  insert := procedure(m, k1) {
2      match (m) {
3          case Nil() :
4              return Node(k1, Nil(), Nil());
5          case Node(k2, l, r):
6              if (k1 == k2) {
7                  return Node(k1, l, r);
8              } else if (compare(k1, k2) < 0) {
9                  return Node(k2, insert(l, k1), r);
10             } else {
11                 return Node(k2, l, insert(r, k1));
12             }
13         default: abort("Error in insert($m$, $k1$, $v1$)");
14     }
15 };
```

Figure 7: Inserting an element into a binary tree using matching.

SETLX offers two kinds of loops which are the *while loops* and the *for loops*. The while loops follows the grammar rule

```
statement -> "while" "(" boolExpr ")" "" block "" .
```

Figure 9 is an example for implementing *Collatz conjecture*: which has the recursive definition

1. $f(n) := 1 \qquad \text{if } n \leq 1,$
2. $f(n) := \begin{cases} f(n/2) & \text{if } n \,\%\, 2 = 0; \\ f(3 \cdot n + 1) & \text{otherwise.} \end{cases}$

using the while loop.

On the other hand *for-loops* follow the grammar rule

```
statement -> "for" "(" iterator("," iterator)* ")" "" block "" .
```

A simple iterator would be x in s where s is a set or a list or a string. An example for the usage of *for-loops* is represented in figure 10 and has an output represented in figure 11

```
diff := procedure(t, x) {
    match (t) {
        case t1 + t2 :
            return diff(t1, x) + diff(t2, x);
        case t1 - t2 :
            return diff(t1, x) - diff(t2, x);
        case t1 * t2 :
            return diff(t1, x) * t2 + t1 * diff(t2, x);
        case t1 / t2 :
            return ( diff(t1, x) * t2 - t1 * diff(t2, x) ) / t2 * t2;
        case f ** g :
            return diff( @exp(g * @ln(f)), x);
        case ln(a) :
            return diff(a, x) / a;
        case exp(a) :
            return diff(a, x) * @exp(a);
        case ^variable(x) : // x is defined above as second argument
            return 1;
        case ^variable(y) : // y is undefined, matches any other variable
            return 0;
        case n | isNumber(n):
            return 0;
    }
};
```

Figure 8: A function to perform symbolic differentiation.

```
f := procedure(n) {
    if (n == 0) {
        return 1;
    }
    while (n != 1) {
        if (n % 2 == 0) {
            n /= 2;
        } else {
            n := 3 * n + 1;
        }
    }
    return n;
};
```

Figure 9: A program to test the Ulam conjecture.

```
rightAdjust := procedure(n) {
    switch {
        case n < 10 : return "   " + n;
        case n < 100: return  "  " + n;
        default:      return   " " + n;
    }
};
for (i in [1 .. 10]) {
    for (j in [1 .. 10]) {
        nPrint(rightAdjust(i * j));
    }
    print();
}
```

Figure 10: A simple program to generate a multiplication table.

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Figure 11: Output of the program in Figure 10.

1.3 Regular Expressions

SETLX like most modern programming languages support *Regular expressions* which is a very powerful tool to process strings. Regular Expressions can be used in match statements. One example for that is shown in figure 12 which is a procedure that takes a certain string and recognize if it's a word, integer or just a white space.

Regular Expressions also can be used for extracting substrings. Consider the example given in figure 13 which extracts parts of a phone-code in the format of `+49-711-6673-4504` where 49 is the country code, 711 is the area code, 6673 is the company code and 4504 is the extension. In this example if an expression is matched, variable c will have the country code, variable a will have the area code, variable co will have the company code, variable x will have the extension, finally variable e will hold the string that matched the whole regular expression.

```
1  classify := procedure(s) {
2      match (s) {
3          regex '0|[1-9][0-9]*': print("found an integer");
4          regex '[a-zA-Z]+'    : print("found a word");
5          regex '\s+'          : // skip white space
6          default              : print("unkown: $s$");
7      }
8  };
```

Figure 12: A simple function to recognize numbers and words.

```
1  extractCountryArea := procedure(phone) {
2      match (phone) {
3          regex '\+([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)' as [e, c, a, co, x]:
4              return [c, a];
5          default: abort("The string $phone$ is not a phone number!");
6      }
7  };
```

Figure 13: A function to extract the country and area code of a phone number.

Regular Expressions can also be used with *scan-statements* as shown in figure 14. *Scan Statements* have the general form of

scan (s) {
  regex r_1 as l : b_1
  ⋮
  regex r_n as l : b_n
}

```
1  printComments := procedure(file) {
2      s := join(readFile(file), "\n");
3      scan (s) {
4          regex '//[^\n]*'                as c: print(c[1]);
5          regex '/\*([^*]|\*+[^*/])*\*+/' as c: print(c[1]);
6          regex '.|\n'                        : // skip every thing else
7      }
8  };
```

Figure 14: Extracting comments using the match statement.

Given bellow some examples for predefined functions for Regular Expressions.

1. `testRegexp('(a*)(a+)b',"aaab");` yields a result `["aaab", "aa", "a"]`. where every item of the list is defined by the order of the opening parenthesis.

2. `matches('(a*)(a+)b',"aaab", true);` with three arguments with the last one defined as a boolean *true* has the exact same effect as the `testRegexp` and has the same return of `["aaab", "aa", "a"]`.

3. `matches('(a*)(a+)b',"aaab");` with two arguments returns true if the string given matches the regular expression or false otherwise. In this case given in the example the return would be true.

4. `replace(s, r, t)` replaces every substring of s that matches the regular expression r by the string t.

1.4 Functional Programming and Closures