Static Program Analysis for SetlX

Ahmad Aboulazm

July 9, 2013

Contents

Chapter 1

Introduction

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, five main 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.
3. Checking if any used procedure or class are defined in the code or not.
4. Checking that any procedure or class are called with the correct number of parameters.
5. 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.

The following chapters are going to be as follows

1. Chapter 2 : is going to be a brief introduction about the main language setlX.
2. Chapter 3 : is going to discuss the implementation of the analyzer.
3. Chapter 4 : is going to cover the part concerning the regression tests.
4. Chapter 5 : is going to be about further implementation that can be done in the future.

Chapter 2

Introducing setlX

setlX has been upgraded lately from the old untyped language *SETL*. It has been invented mainly for educational purposes as it is well suited to familiarize students with set theory, terms and functional programming.

In the following sections 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`

2.1 SetlX Data Types

setlX is an interpreter that can be used to evaluate simple expressions. 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 difference 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 combine 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 provides some operators to create boolean expressions as well such as `>,<,==,!=,<=,>=`.

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 bound to the values inside $s_1, \cdots, s_n$ 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 : A list is an ordered collection of elements which can contain an element more than once unlike sets.

Apart 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 length two. 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 $r[x]$

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

The symbol Ω in SETLX appears when there is no solution or result to a certain expression.

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 2.1 for example defines a procedure that computes prime numbers starting 2 up to a given number n.

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

Figure 2.1: A procedure to compute the prime numbers.

In Figure 2.1, 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 don'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.

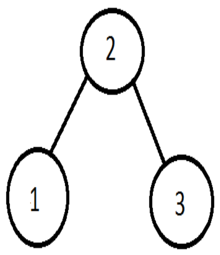
   represented as

   `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 2.2 actually shows how terms can be used in SETLX to implement ordered binary trees.

```
1   insert := procedure(m, k1) {
2       switch {
3           case fct(m) == "Nil" :
4                return Node(k1, Nil(), Nil());
5           case fct(m) == "Node":
6                [ k2, l, r ] := args(m);
7                if (k1 == k2) {
8                    return Node(k1, l, r);
9                } else if (compare(k1, k2) < 0) {
10                   return Node(k2, insert(l, k1), r);
11               } else {
12                   return Node(k2, l, insert(r, k1));
13               }
14      }
15  };
```

Figure 2.2: Inserting an element into a binary tree.

In an ordered binary tree any node will have nodes with lower values inserted on left of it and any nodes with higher values inserted on right of it, and in case the inserted node has the same value of an existing node then it won't be inserted into the tree. This is exactly what happens in the example shown in Figure 2.2 where m represents the tree and $k1$ represents the value of the node to be inserted. The tree's parent node is first checked if it's empty by checking if the functor of this node is `Nil` which will match the first case of the switch statement. If that condition is true then the returned tree will just be the newly inserted node, otherwise if it's a non-empty `Node` it will then match the second statement, in this case at line '6' $k2$ will have the value of this node while l and r will hold the left and right sub-trees of this node respectively. After that three nested *if-conditions* are executed. If the inserted node's value has the same value of the current node being checked then it just returns the original tree m, otherwise if it has a lower value then a recursion happens only this time checking the inserted node with the left sub-tree, the same goes for the case if the inserted node is higher only this time with the right sub-tree.

2.2 Statements in setlX

In this section 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 2.3, 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.

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

Figure 2.3: 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 2.4 while an example for the use of the *switch statements* is shown in Figure 2.5.

```
toBin := procedure(n) {
    if (n < 2) {
        return str(n);
    } else {
        r := n % 2;
        n := floor(n / 2);
        return toBin(n) + toBin(r);
    }
};
```

Figure 2.4: A function to compute the binary representation of a natural number.

```
sort3 := procedure(l) {
    [ x, y, z ] := l;
    if (x <= y) {
        if (y <= z) {
            return [ x, y, z ];
        } else if (x <= z) {
            return [ x, z, y ];
        } else {
            return [ z, x, y ];
        }
    } else if (z <= y) {
        return [z, y, x];
    } else if (x <= z) {
        return [ y, x, z ];
    } else {
        return [ y, z, x ];
    }
};
```

Figure 2.5: 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 2.6

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 2.7 which does the same

```
reverse := procedure(s) {
    match (s) {
        case []   : return s;
        case [c|r]: return reverse(r) + c;
        default   : abort("type error in reverse($s$)");
    }
};
```

Figure 2.6: A function that reverses a string.

work as the code shown in Figure 2.2 discussed previously.

```
insert := procedure(m, k1) {
    match (m) {
        case Nil() :
             return Node(k1, Nil(), Nil());
        case Node(k2, l, r):
             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));
             }
        default: abort("Error in insert($m$, $k1$, $v1$)");
    }
};
```

Figure 2.7: Inserting an element into a binary tree using matching.

A more complex example is shown in Figure 2.8 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 section entitled *predefined functions*.

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 2.9 is an example for implementing *Collatz conjecture*: which is represented as the recursive function f with definition

1. $f(n) := 1$ if $n \leq 1$,

```
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 2.8: A function to perform symbolic differentiation.

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 2.10 and has an output represented in Figure 2.11.

```
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 2.9: A program to test the Collatz 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 2.10: A simple program to generate a multiplication table.

2.3 Regular Expressions

SETLX like most modern programming languages supports *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 2.12 which is a procedure that takes a string and recognizes 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 2.13 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

```
  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 2.11: Output of the program in Figure 2.10.

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

Figure 2.12: A simple function to recognize numbers and words.

x will have the extension, finally variable e will hold the string that matched the whole regular expression.

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

Figure 2.13: 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 2.14. *Scan statements* have the general form of

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

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

Figure 2.14: Extracting comments using the match statement.

Given bellow are 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 value 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 result would be true.

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

2.4 Functional Programming

It has been stated previously that SETLX is a full-fledged functional language. Functions can be used as arguments to other functions and as return values. There is no big difference between the type of a function and any other type that can be assigned to a variable.

Using functions as arguments to other functions is not that commonly seen in conventional programming languages. Figure 2.15 shows an example for that. In this example the function *reduce* takes two arguments which are l as a list and f as a function. What this function do is that it reduces the list to one single value at the end by combining all its elements by the function f, so if we could say that f was an add function then the function *reduce* computes the sum of all its elements. In case the list was empty then the function returns the value om(Ω).

```
reduce := procedure(l, f) {
    match (l) {
        case []     : return;
        case [x]    : return x;
        case [x,y|r]: return reduce([f(x,y) | r], f);
    }
};
add      := procedure(a, b) { return a + b; };
multiply := procedure(a, b) { return a * b; };

l := [1 .. 36];
x := reduce(l, add     );
y := reduce(l, multiply);
```

Figure 2.15: Implementing a second order function.

Another example for second order functions is shown in Figure 2.16.

In this example the function *sort* takes two arguments. The first argument l is a list and the second argument *cmp* is a function that compares two list elements and returns either true or false. What this function does is that it orders the list according to the second order function given as an argument. If for example the function `less` defined in line 20 was given as an argument, then the resulting list would be ordered ascendingly. If instead the function `greater` was the argument given, then the resulting list would be ordered descendingly.

The previous two examples showed how functions can be used as arguments to other functions. As mentioned before functions can also be used as return values from other functions. Shown in Figure 2.17 is an example that computes the *discrete-derivative* of a certain function f. This example shows how functions can actually be used as a return value from other functions.

```
sort := procedure(l, cmp) {
    if (#l < 2) { return l; }
    m := #l \ 2;
    [l1, l2] := [l[.. m], l[m+1 ..]];
    [s1, s2] := [sort(l1, cmp), sort(l2, cmp)];
    return merge(s1, s2, cmp);
};
merge := procedure(l1, l2, cmp) {
    match ([l1, l2]) {
        case [[], _] : return l2;
        case [_, []] : return l1;
        case [[x1|r1], [x2|r2]] :
             if (cmp(x1, x2)) {
                 return [x1 | merge(r1, l2, cmp)];
             } else {
                 return [x2 | merge(l1, r2, cmp)];
             }
    }
};
less    := procedure(x, y) { return x < y; };
greater := procedure(x, y) { return y < x; };
l  := [1,3,5,4,2];
s1 := sort(l, less   );
s2 := sort(l, greater);
```

Figure 2.16: A generic sort function.

```
delta := procedure(f) {
    return n |-> f(n+1) - f(n);
};
g := n |-> n;
h := n |-> 2 ** n;
deltaG := delta(g);
deltaH := delta(h);

print([ deltaG(n) : n in [1 .. 10]]);
print([ deltaH(n) : n in [1 .. 10]]);
```

Figure 2.17: Computing the discrete derivative of a given function.

2.5 Exceptions and Backtracking

SETLX supports means of exception handling and backtracking. But since they are irrelevant to the project itself it won't be discussed here. To know more about them kindly check the official tutorial referenced at the beginning of this chapter.

2.6 Objects and Classes

SETLX supports some basic means of *object-oriented programming* most importantly classes. A *class* is a collection of *member-variables* and *methods* where *member-variables* are some defined variables and the *methods* are some defined functions. Figure 2.18 shows an example for creating a class *point* which represents a point in a plane.

```
class point(x, y) {
    mX := x;
    mY := y;

    getX  := procedure()  { return mX;            };
    getY  := procedure()  { return mY;            };
    setX  := procedure(x) { this.mX := x;         };
    setY  := procedure(y) { this.mY := y;         };
    toStr := procedure()  { return "<$mX$, $mY$>"; };

    distance := procedure(p) {
        return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
    };
}
```

Figure 2.18: The class `point`.

The first line of this example defines the class along with the constructor of this class where the constructor takes two arguments x and y. The general form of a class definition is as follows

```
class name(x_1, ..., x_n) {
    member-and-method-definitions
}
```

where *name* is the name of the class, $x_1, \cdots, x_n$ are the formal parameters of the constructor and *member-and-method-definitions* are the definitions of member variables and methods. In this example *mX and mY* are the defined member variables while *getX, getY, setX, setY, toStr and distance* are the class methods.

We can define an object *origin* of this class `point` by writing

```
origin := point(0, 0);
```

then if we would print the object itself by writing

```
print(origin);
```

that would yield the output shown in Figure 2.19. This output has been formatted this way to make it readable. Note that the constructor of the class `point` is part of the object origin.

```
object<
    { distance := procedure(p) {
                      return sqrt((mX-p.getX()) ** 2 + (mY-p.getY()) ** 2);
                  };
      toStr := procedure() { return "<$mX$, $mY$>"; };
      setX  := procedure(x) { this.mX := x; };
      getY  := procedure() { return mY; };
      mX    := 0;
      setY  := procedure(y) { this.mY := y; };
      getX  := procedure()  { return mX; };
      mY    := 0;
      class (x, y) {
          mX := x;
          mY := y;
          distance := procedure(p) {
              return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
          };
          getX := procedure() { return mX; };
          getY := procedure() { return mY; };
          setX := procedure(x) { this.mX := x; };
          setY := procedure(y) { this.mY := y; };
          toStr := procedure() { return "<$mX$, $mY$>"; };
      }
    }
>
```

Figure 2.19: The output of the command "`print(origin);`".

The fact that all methods are stored as part of the object may seem redundant but in fact it's useful in case we needed to disable a method of the class from being used for a certain object. For example to disable the setter methods for the object *origin* by simply writing

```
origin.setX := om;
origin.setY := om;
```

If we don't want to change methods on a per-object basis, we just need to declare these methods to be static as shown in Figure 2.20.

Now in this case the command

```
print(origin);
```

```
class point(x, y) {
    mX := x;
    mY := y;

  static {
    getX  := procedure()  { return mX;             };
    getY  := procedure()  { return mY;             };
    setX  := procedure(x) { this.mX := x;          };
    setY  := procedure(y) { this.mY := y;          };
    toStr := procedure()  { return "<$mX$, $mY$>"; };

    distance := procedure(p) {
        return sqrt((mX - p.getX()) ** 2 + (mY - p.getY()) ** 2);
    };
  }
}
```

Figure 2.20: The class `point` implemented using static methods.

will yield the output shown in Figure 2.21.

```
object<
    { mX := 0;
      mY := 0;
      class (x, y) {
          mX := x;
          mY := y;
        static {
          distance := procedure(p) {
                          return sqrt((mX-p.getX()) ** 2 + (mY-p.getY()) ** 2);
                      };
          getX  := procedure()  { return mX;             };
          getY  := procedure()  { return mY;             };
          setX  := procedure(x) { this.mX := x;          };
          setY  := procedure(y) { this.mY := y;          };
          toStr := procedure()  { return "<$mX$, $mY$>"; };
        }
      }
    }
>
```

Figure 2.21: Output of “`print(origin);`”.

2.7 Predefined Functions

SETLX provides a lot of predefined functions for

1. dealing with sets and strings.
2. dealing with strings.
3. working with terms.
4. working with mathematical functions.
5. dealing with objects.
6. supporting interactive debugging.
7. dealing with I/O.

Some of the predefined functions have been mentioned earlier. Since there are too many predefined functions, we are only going to discuss the most important of them that are used in the implementation of the static analysis.

1. `+`: computes the union of its arguments which are either sets or lists.
2. `first`: The function `first`(s) picks the first element from the sequence s. The argument s can either be a set, a list, or a string.
3. `fromB`: The function `fromB`(s) picks the first element from the sequence s. The argument s can either be a set, a list, or a string. This element is removed from s and returned. This function returns the same element as the function `first` discussed previously.
4. `fromE`: The function `fromB`(s) picks the last element from the sequence s. The argument s can either be a set, a string, or a list. This element is removed from s and returned. This function returns the same element as the function `last` discussed previously.
5. `domain`: If r is a binary relation, then the equality

   ```
   domain(r) = { x :[x,y] in R }
   ```

   holds. For example, we have

   ```
   domain({[1,2],[1,3],[5,7]}) = {1,5}.
   ```

6. `execute`: The function `execute` is called as

   `execute`(s).

   Here, s has to be a string that can be parsed as a SETLX statement. This statement is then executed in the current variable context and the result of this evaluation is returned. Note that s can describe a SETLX expression of arbitrary complexity. For example, the statement

   ```
   execute("f := procedure(x) { return x * x; };");
   ```

has the same effect as the following statement:

```
f := procedure(x) { return x * x; };
```

7. `split`: The function `split` is called as

   `split`(s, t).

   Here, s and t have to be strings. t can either be a single character or a regular expression. The call `split`(s, t) splits the string s at all occurrences of t. The resulting parts of s are collected into a list. If t is the empty string, the string s is split into all of its characters. For example, the expression

   ```
   split("abc", "");
   ```

   returns the list `["a", "b", "c"]`. As another example,

   ```
   split("abc xy z", " +");
   ```

   yields the list

   ```
   ["abc", "xy", "z"].
   ```

   Note that we have used the regular expression "`+`" to specify one or more blank characters.

   Certain *magic* characters, i.e. all those characters that serve as operator symbols in regular expressions have to be escaped if they are intended as split characters. Escaping is done by prefixing two backslash symbols to the respective character as in the following example:

   ```
   split("abc|xyz", "\\|");
   ```

   The function `split` is very handy when processing comma separated values from CVS files.

8. `str`: The function `str` is called as

   `str`(a)

   where the argument a can be anything. This function computes the string representation of a. For example, after defining the function `f` as

   ```
   f := procedure(n) { return n * n; };
   ```

   the expression `str(f)` evaluates to the string

   ```
   "procedure(n) { return n * n; }".
   ```

9. `canonical`: Given a term t, the expression `canonical`(t) returns a string that is the canonical representation of the term t. The point is, that all operators in t are replaced by functors that denote these operators internally. For example, the expression

   ```
   canonical(parse("x+2*y"));
   ```

   yields the string

   ```
   ^sum(^variable("x"), ^product(2, ^variable("y"))).
   ```

This shows that, internally, variables are represented using the functor `^variable` and that the operator “+” is represented by the functor `^sum`.

10. `parseStatements`: Given a string s, the expression

    `parseStatements`(s)

    tries to parse the string s as a sequence of SETLX statements. In order to visualize the structure of this term, the function `canonical` disussed above can be used. For example, the expression

    ```
    canonical(parseStatements("x := 1; y := 2; z := x + y;"));
    ```

    yields the following term (which has been formatted for easier readability):

    ```
    ^block([^assignment(^variable("x"), 1),
            ^assignment(^variable("y"), 2),
            ^assignment(^variable("z"), ^sum(^variable("x"), ^variable("y")))
           ])
    ```

Chapter 3

Implementation

In this chapter we are going to discuss the implementation of the analyzer itself. Since the code is too long, we are going only to discuss the focal points of each part of it.

First of all, in order to run the analyzer you have to type

```
setlx analyzeCode.stlx --params fileName.stlx
```

in the command line prompt for windows users or in the terminal for Mac or Unix users after setting up SETLX on your computer, where `fileName` is either a file or a list of files separated by commas to be analyzed.

In the upcoming sections we will discuss each part of the code separately. Other parts will be referenced but won't be discussed at the time but instead in their own specified sections.

3.1 Main Part

The main part of the code is shown in figure 3.1.

1. The main part consists of a *for-loop* that loops through the files to be checked printing the beginning and the end of it to notify the user which file is being checked at the moment.

2. After reading the file into the variable `l`, in *line 5* a new line "\n" in then added to the end of each line in the code in order to separate them. The reason for that is to avoid the commenting sign "\\" at any line from commenting all code lines followed by it.

3. After that in *line 6* the code is then parsed and saved in the same variable `l`.

4. In lines 8 to 11, four empty sets are created to be used later for checking the code.

5. In *line 14* and *line 15* all existing procedures and classes in the code being checked are saved by a certain format in the variables called *allProceduresIdenteties* and *allClassesIdenteties*.

```
for(directory in params){
    print("NOW CHECKING $directory$");
    directory := str(directory);
    l := readFile(directory);
    l := +/ [ g + "\n" : g in l ];
    l := parseStatements(l);

    definedVars := {};
    allUsedVars := {};
    allProceduresIdenteties := {};
    allClassesIdenteties := {};


    identifyAllProcedures(l, allProceduresIdenteties);
    identifyAllClasses(l, allClassesIdenteties);
    analyzeAllLoadedFiles(l, definedVars, allUsedVars,
     allProceduresIdenteties, allClassesIdenteties);

    checkCode(l, definedVars, allUsedVars, false, "",
    {}, {}, allProceduresIdenteties, false, {}, {}, {},
     false, "", allClassesIdenteties);

    unUsedVars := definedVars-allUsedVars;
    if(#unUsedVars != 0)
    {
            if(#unUsedVars == 1){
               print("Variable " + unUsedVars + " is defined
                but never used in the whole code! unless
                 it(they) was(were) assigned to a returned
                  procedure");
            }
            else{
               print("Variables " + unUsedVars + " are
                defined but never used in the whole code!
                 unless it(they) was(were) assigned to a
                  returned procedure");

            }
    }
    print("DONE CHECKING $directory$");
}
```

Figure 3.1: The main part of the analyzer's code.

6. After that in *line 16* the analyzer checks all files being loaded by the analyzed file using the *analyzeAllLoadedFiles* function so that any important information in them would be available further on when checking the file being analyzed.

7. The most important part then comes at *line 19* which is checking if there is any warning mentioned in the first chapter should appear to the user concerning his/her code using the *checkCode* function.

8. Then starting *line 23* upto *line 39* the analyzer checks if there were any variables defined throughout the whole code that were never used and in such case it prints to the user a notification concerning these variables.

9. This process is then repeated in case there were more files to be checked.

3.2 checkCode Function

The *checkcode* function takes 15 arguments which are :

1. `l` : which holds the code to be analyzed after being parsed in the main part.

2. `rw definedVars` : which is a set containing all defined variables at the point.

3. `rw allUsedVars` : which is set containing all used variables at the point.

4. `isProcedure` : which is a boolean value that indicates if the current code being checked belongs to a procedure or not.

5. `procName` : which holds the name of the procedure currently being checked. If the current code being checked didn't belong to a procedure then it's just an empty string.

6. `procParameters` : which is the set of parameters of the current procedure being checked. If the current code being checked didn't belong to a procedure then this argument is just an empty set.

7. `rw procEmbeddedProcedures` : which is a set that holds all procedures defined inside the procedure being checked in case they existed, if not then it is just the empty set.

8. `rw allProceduresIdenteties` : which is a set containing the identities of all existing procedures in the code in a certain format which is discussed in the section entitled *`identifyAllProcedures Function`*.

9. `forLoopCase` : which is a boolean value that indicates if the current code being checked belonged to a for loop or not.

10. `rw forLoopTmpDefVars` : which is a set containing the temporary defined variables in the for-loop case which are the iterators of the loop itself.

11. `exceptionTmp` : which is a set containing the temporary defined variables for an exception in case an exception block was being checked. Otherwise, it's just an empty set.

12. `matchTmp` : which is a set containing the temporary defined variables in a match case branch in case a match statement was being checked at the moment. Otherwise, it's just the empty set.
13. `isClass` : which is a boolean value that indicates if the current code being checked belonged to a class or not.
14. `className` : which is the name of the class in case the current code being checked belonged to a class. Otherwise it's just an empty string.
15. `rw allClassesIdenteties` : which is a set containing the identities of all existing classes in the code in a certain format which is discussed in the section entitled *"identifyAllClasses Function"*.

The *checkCode* function is composed of a big match statement. Its functionality is actually to decompose any block of code to simple statements in order to extract from every statement the defined and used variables in it. An example for that is the if-condition block shown below.

```
case ^ifThenBranch(cond, then):
                    checkCode(cond, ...);
                    checkCode(then, ...);
                    return;
```

where `cond` is the condition of the if-statement and the `then` is the block of code executed in case the condition was satisfied. The *checkCode* function does the same with all other constructors that contain blocks such as the *while-loops*, *for-loops*, *try-catch*, *check* and *switch* statements. However the *match statements* and the *scan statements* are held kind of differently by the *checkCode* function where they are only decomposed at the beginning as shown below

```
case ^scan(x, y, z):
                checkCode(x, ...);
                checkCode(y, ...);
                for(w in z){
                      checkCode(w, ...);
                }
```

```
case ^match(x, y):
                checkCode(x, ...);
                for(z in y){
                      checkCode(z, ...);
                }
```

Then checking their cases' first part is shown in Figure 3.2

```
case ^matchCaseBranch([v],cond,do) :
            getMatchedVariables(v, matchTmp);
            usedVarsInMatch := {};
            checkCode(cond, definedVars, usedVarsInMatch, ...);
            checkCode(do, definedVars, usedVarsInMatch, ...);
            allVariablesAreDefined := true;
            undefinedVariables := {};
            for(v in usedVarsInMatch){
                 v := str(v);
                 if((v in definedVars) && (!((v in
                    forLoopTmpDefVars)|| (v in exceptionTmp)
                    || (v in matchTmp)))){
                       allUsedVars := allUsedVars + {v};
                 }
                 if(!(v in definedVars || v in forLoopTmpDefVars
                 || v in exceptionTmp || v in matchTmp)){
                   allVariablesAreDefined := false;
                   undefinedVariables := undefinedVariables + {v};
                 }
            }
```

Figure 3.2: Match case branch-Part 1