THE SETL PROGRAMMING LANGUAGE

TAPASYA PATKI POOJA BHANDARI

Department of Computer Science
University of Arizona

Abstract
This paper presents a study on the language SETL, which is a very-high level, wide spectrum programming language based on the mathematical theory of sets, primarily used for transformational programming, compiler optimization, algorithm design and data processing. We describe the language features and delve into the issues from the compiler’s perspective.

1 HISTORY

In the late 1960s, researchers were addressing compiler optimization problems, and a need for an expressive, high-level language that supported set-intensive algorithm design was felt. SETL, introduced by Prof. Jacob Schwartz at the New York University (NYU) at the Courant Institute of Mathematical Sciences in 1970, was conceived to meet this purpose. Prof. Schwartz, who obtained his PhD from Yale University in 1951, is a renowned mathematician and a computer scientist, and currently holds professorship at NYU. Most of the research and developmental work related to SETL has been carried out at NYU.

Modern computer programming languages can be broadly grouped into Object-Oriented languages, Functional programming languages, and Scripting languages. Since SETL was developed during 1970s, SETL does not strictly belong to the above classification. While SETL provides no support for object-orientation, newer versions and dialects of SETL, like SETL2 and ISETL support object oriented and graphics programming.

SETL is a very powerful language, and has various application domains. The first ADA translator, ADA/Ed and the first implementation of AIML, the markup language for Artificial Intelligence, were written in it. Python’s predecessor ABC, is also heavily influenced by SETL.

2 KEY FEATURES

SETL is an interpreted, very high-level, dynamic language that provides for set-theoretic syntax and semantics. Derived from the Algol-family, it bears syntactic similarity with C and Perl. It offers the programmer a lot of flexibility by letting the code be independent of any data-structure implementation details, and thus strives to put the needs of the programmer ahead of those of the machine.

SETL is more-or-less an imperative, sequential language with assignment, and partial support for functional programming through backtracking. Program development in SETL is facilitated by the use of fewer and more abstract operations. Typically, a small SETL program can do a lot.

A variable in SETL is weakly and dynamically typed, and storage allocation is done at run time. SETL has value semantics, and does not allow pointer manipulations. In this respect, SETL realizes Hoare's ideal of programming without pointers. This brings with it a substantial measure of orthogonality and robustness.
SETL is called a wide spectrum language (WSL) as it allows all levels of abstraction from problem specifications down to hardware level implementations. This is the most important feature of language that makes the language suitable for rapid prototyping and transformational programming.

3 TYPES, OPERATORS AND SCOPING

The conventional mathematical and logical operations are supported in SETL [1,2,3]. An identifier is not statically associated with a particular data-type in this language. Instead, depending upon the last value assigned to a variable, the type of the variable is set. Also, the operator is applied depending upon the type-context of its operands, for example, a ‘+’ may mean an integer addition or a string concatenation.

Scalar Data Types

SETL primarily has two levels of data-typing - Simple/Scalar types, and Compound types. Simple types include atom, integer, real, string, and boolean. There are no limitations on the size and the range of these simple types (except in situations where there might be hardware constraints).

Compound Data Types

Compound data types in SETL are in the form of sets, tuples and maps. A set is an unordered collection of distinct elements of arbitrary types. Sets are represented by listing the elements within the curly braces { }. SETL does not allow sets of infinite size. Various set operators are listed in Table 1. The following example illustrates the basic set operations:

Ex 1: Set

```
a := {2, 3, 4};
b := {3, 5, 6};
c := a + b; $ c = {2, 3, 4, 5, 6}
x := c with 9; $ x = {2, 3, 4, 5, 6, 9}
x less:=9; $ x = {2, 3, 4, 5, 6}
p := {1,2}
e := arb p; $ e = 1 or 2
```

A tuple is an ordered sequence of arbitrary types, with support for direct indexing. Tuples are similar to one dimensional vectors, but possess greater degree of flexibility. The components of a tuple are stored at contiguous locations within a dynamically managed memory area. The elements of a tuple can be accessed using subscripting. If a previously defined value in a tuple is set to om, a hole is created in the tuple. Om, or omega, is a special type in SETL which represents an undefined value. Various tuple operators are listed in Table 1. The following example depicts tuple usage:

Ex 2: Tuple

```
tuple := [ 2, 'xyz', [1, 2, 3], 44 ];
a:=tuple(1); $ a := 2
tuple(1) := 7; $ tuple = [ 7, 'xyz', [1, 2, 3], 44 ]
tuple(3) := om; $ creates a hole in the tuple
```
A map is simply a set of tuples of length 2, where the first element of each tuple is said to belong to the domain of the map, and the second element to the range. More formally, given sets A and B, a map is a subset F of the Cartesian Product A x B, defined under the relation \( b = f(a) \), where \( a \in A \), \( b \in B \), and \( f \) is a function. Single valued maps are for functions and multi-valued maps are for relations. Maps provide simple and easy to use facility for representing data structures and databases [20]. SETL implements the domain and range of a map by calls. All the map operators are discussed in Table 1. The following example shows all the properties of maps:

\[
\text{map} := \{ [1,1], [4,2], [9,3], [16,4], [25,5] \}; \\
x := \text{domain map}; \quad x = \{ 1, 4, 9, 16, 25 \} \\
y := \text{range map}; \quad y = \{ 1, 2, 3, 4, 5 \} \\
\text{map lessf} := 1; \quad \text{$\text{map lessf}$ removes [1,1]}
\]

**Operators**

<table>
<thead>
<tr>
<th>Operators</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Set Union/ Tuple Concatenation</td>
</tr>
<tr>
<td>-</td>
<td>Set difference</td>
</tr>
<tr>
<td>*</td>
<td>Set Interception</td>
</tr>
<tr>
<td>less</td>
<td>Removes an element from a set</td>
</tr>
<tr>
<td>from</td>
<td>Removes an element and assigns remaining</td>
</tr>
<tr>
<td>arb</td>
<td>Selects an arbitrary element</td>
</tr>
<tr>
<td>#</td>
<td>Highest defined element</td>
</tr>
<tr>
<td>with</td>
<td>Adds a single element at end of tuple/set</td>
</tr>
<tr>
<td>fromb</td>
<td>Removes the first element of tuple</td>
</tr>
<tr>
<td>frome</td>
<td>Removes the last element of tuple</td>
</tr>
<tr>
<td>domain</td>
<td>Returns domain of a map</td>
</tr>
<tr>
<td>range</td>
<td>Returns range of a map</td>
</tr>
<tr>
<td>lessf</td>
<td>Removes pairs for one domain value</td>
</tr>
</tbody>
</table>

Table 1: Set, Tuple and Map operators

**Omega**

There is another special data type, called omega, or \( \text{om} \), which is used to represent an undefined value. This is a unique feature of SETL. It is possible to un-define a previously defined element using \( \text{om} \). It is convenient to regard \( \text{om} \) as being the value of an undefined variable, or element. \( \text{om} \) is used to facilitate exception handling in SETL. Identifiers which have not been set to any particular value are in this undefined state, and may be thought of as containing the value \( \text{om} \). Thus, \( \text{om} \) is not a value but the absence of any value. \( \text{om} \) can appear only as the right hand side of the assignment and in equality tests.

\[
a := 2; \\
a := \text{om}; \quad \text{$\text{value of a is undefined using om}$} \\
a := a + 4; \quad \text{$\text{causes an error}$}
\]

Ex 4: Omega & Error
SETL also supports error handling through *assertions*. Similar to Java and other languages, the `assert(expr)` module signals an error if the expression results in a fallacy after the evaluation of the expression `expr`.

**Data Structure Examples**

Elements can easily be inserted or deleted from tuples, which makes implementation of data structures easy. The following example depicts how tuples can be used as queues and stacks in a simple manner.

```
q := []; q with := 1; q with := 2; $ q = [1, 2]
elt from b q; $ elt = 1, q = [2]
elt from b q; $ elt = 2, q = []
elt from b q; $ e = om, q = []
```

Ex 5: Tuple used as a queue

```
s := []; s with := 1; s with := 2; $ s = [1, 2]
elt from s; $ elt = 2, s = [1]
elt from s; $ elt = 5, s = []
elt from s; $ e = om, s = []
```

Ex 6: Tuple used as Stack

Maps can be used to represent complex data-structures like graphs and trees easily. This helps the programmer to code at an abstract level, without bothering about the intricate implementation details. Figure 1 shows an example of this.

![Binary Search Tree](image)

**Type and Scope Rules**

SETL is dynamically typed, and depending upon the value assigned to a variable at run-time, the type is decided. Also, the current data-type of a variable is the type of the value last assigned to it. In SETL, variables have procedure scope therefore, any variable name `varX` is implicitly changed to `varX_n` where `n` is the procedure number. This way, name conflicts can be resolved.

In SETL the objects must be built explicitly before they can be used elsewhere and there is no lazy evaluation. Every object, except atoms and procedure values, can be converted to a string and back facilitating process-boundary transfers. Strings are termed as first-class citizens in SETL, making the language suitable for data-processing and management. Shared variables and concurrency do not exist in the language.
4 USEFUL CONSTRUCTS

SETL has tailored language constructs to support set-theoretic programming operations efficiently. These constructs are useful for handling discrete mathematics and logic programming in SETL.

Quantifiers

In SETL, there are two kinds of quantifiers used to test, exists and forall, which make implementation of sets very efficient, allowing subsets to be built in a compact way as each set requires a loop implicitly.

<table>
<thead>
<tr>
<th>Ex 7: Example with exists</th>
<th>Ex 8: Example with forall</th>
</tr>
</thead>
</table>
| set := \{ 2, 4, 6\};  
if exists a in set | set := \{ 2, 4, 6\};  
if forall a in set |
| then $ executed with a = 6 
else end;  
| then $ will not be executed 
else $ executed with x=om end;  
| |

Compound Operators

In SETL, there is another distinguishing form of loop called the compound operator. A compound operator is formed by adding a slash / to the name of a binary operator. This is use to apply the binary operator to the sequence of elements, which are either a set or a tuple. The following table shows how to use compound operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+/t$</td>
<td>Sum of elements in tuple</td>
</tr>
<tr>
<td>$0+/t$</td>
<td>Same but sum is 0 not om for empty tuple/set</td>
</tr>
<tr>
<td>$*/[x \text{ in set}</td>
<td>1 \text{ in } x]$</td>
</tr>
<tr>
<td>$&quot;+/t$</td>
<td>String from tuple of characters</td>
</tr>
</tbody>
</table>

Table 2: Compound Operators

5 DESIGN CONSIDERATIONS

Given the very-high level nature and flexible support for compound data types and control structures of SETL, the burden of the implementation and low-level system handling is delegated to the compiler. It becomes very important to consider the internal representations for the complex data types like sets, tuples and maps. The internal representations must be simple, and yet should provide for fast access. Compound iteration, which is integral to these compound types should be supported naturally by these internal representations.

The default mechanism used for representation of sets is a doubly-linked hash table. A system hash function is applied on the elements to obtain the locations. Thus hash function can be applied to any object in SETL. Each element is stored in what is called an ‘element-block’, which can be considered as the ‘data’ part of a typical linked-list node. To facilitate iteration, the linked nature of the list is exploited. With the use of a good hash function, the search time can be made more or less constant, i.e. of the order O(1). Thus, use of a hash-table allows us to have a simple, yet reasonably fast access. Figure 2 depicts how the doubly-linked hash table can be maintained.
Tuples are represented as dynamically-growing arrays, occupying contiguous memory locations. Reallocation of tuples may be needed in case the size grows beyond the available chunk of memory. This might lead to the hidden copy problem, in which the compiler has to implicitly trigger a copy operation based on the dynamic expansion or contraction of the tuple. The same may be applicable when we have even more complicatedly nested data structures, say tuples of sets of sets.

![Figure 2: Internal Representation of a Set](image)

The default representation for a map is similar to that of a set. The domain-value of each element of the map is used as a key to the hash function for determining the location in the hash table. The element-block in this case has two slots, one for the domain object and the other for the range object.

### 6 DATA REPRESENTATION SUBLANGUAGE (DRSL)

Programming in SETL typically comprises of two phases. In the first phase, the focus is on the high-level abstract development of the main algorithm. This phase lets the programmer’s code be independent of any data-structure implementation details, thus making the language orthogonal and suitable for rapid prototyping. Once the correctness and verifiability of the algorithm have been ensured, the second phase begins, where the focus is on improving the efficiency of the implementation. The representation language in SETL is used for this purpose.

DRSL is a unique feature of SETL which makes the language truly wide-spectrum, and gives the programmer the flexibility of overriding the default representations used by the language. SETL’s DRSL lets the programmer add a set of declarations to control the data-structure implementation of his/her previously developed program. In an ideal scenario, no re-writing of the existing code is required. The
program with the added set of declarations is called the supplemented program [22]. The supplemented program produces the same output as the base program.

To support DRSL, new program objects, to be deployed as universal sets of program values, called bases are introduced. Bases are auxiliary data structures which enable us to access related groups of program variables in an efficient manner [21]. Once the bases have been specified, the program variables can be represented as ‘belonging’ to a base. Bases are updated at run-time depending upon the associations specified by the program. Thus, we can declare a program variable \( x \) belonging to a base \( B \), by writing:

\[ x \in B \]

If such a declaration has been made, then a value assigned to \( x \) at run-time is automatically inserted into base \( B \), unless this value is found to exist already. Given this declaration, \( x \) is said to be located in \( B \). The variable \( x \) is represented internally as a pointer to some element-block in the base \( B \), thus, the value of \( x \) can be simply obtained by dereferencing this pointer.

This highlights the purpose of bases. The key idea is to reduce the number of hash operations required during program execution. Suppose we have sets <\( S_1, S_2, S_3 \ldots S_n \)> belonging to a universal base \( B \). Thus, each of the sets \( S_1 \) to \( S_n \) would comprise of elements of \( B \). Once all the base values have been hashed, possible re-hashing of overlapping subsets or elements can be avoided.

Three possible options for a base-association are possible. These are:

1. local association
2. indexed association
3. sparse association

Consider the declarations for a set \( s \):

\[
\begin{align*}
\text{s : local set} & \ (\epsilon \ B) \\
\text{s : indexed set} & \ (\epsilon \ B) \\
\text{s : sparse set} & \ (\epsilon \ B)
\end{align*}
\]

Local association reserves one-bit in each-element block of base \( B \). This bit is set if the element represented by the block belongs to set \( s \). This representation supports fast insertion and deletion. If the indexed representation is used, all the bits are grouped into a bit string, which is maintained and stored elsewhere. A unique index is assigned to each element block in \( B \) to provide access. The disadvantage of local and indexed representation is that to iterate over a structure, a membership test in \( s \) has to be carried out for all element-blocks in \( B \). This can prove to be very inefficient at times when compared to direct iteration over set \( s \), as in the un-based case. When iteration over such a ‘sparse’ set is to be performed more frequently, the Sparse association can be used. Sparse association maintains a doubly-linked hash table for the set \( s \), which is similar to the default format. These associations can also be applied to tuples and maps, or to other complicated, nested program variables.

Here is an example on how DRSL is used. The following set of declarations can be added to a program involving graphs and paths; say for example, a program that finds the shortest path in a given graph. Then, the map for \textit{graph} can be defined as having its domain and range located in the base \textit{nodes}. Similarly, a \textit{path} can be defined as a tuple being located in the same universal base.
One of the primary uses of DRSL is to support automatic data representation in SETL [21]. As mentioned earlier, the primary motive is to reduce the number of hash operations required. This involves the deployment of a selection algorithm to create provisional bases and corresponding based representations for variables otherwise requiring hashing. These provisional bases depict the ‘local’ information, and are later integrated into an overall basing structure by propagating these globally. During this propagation phase, individual bases that are logically similar may be equivalenced. The automatic data representation selection algorithm thus comprises of the following four phases.

- Base Generation
- Representation merging and base equivalencing
- Base pruning and representation adjustment
- Conversion Optimization

Program Transformation

In the SETL philosophy, Program Transformation is realized using Abstract Interpretation Model along with DRSL [8]. In abstract interpretation, the abstract program A is exercised at program P’s compile time to find out desired properties of objects in program P. Thus, a program P with well defined semantics can be projected onto a more abstract program A, capturing salient properties of objects in P; e.g. the sign of a product can be determined from the multiplicands sign, without knowing their specific values. Likewise, the result types for known operators can be fixed at compile time itself regardless of the actual run-time values of those operands. Program transformation is broadly carried out on the following situations:

- Finite differencing for reducing expensive set operations within loops (using program invariants)
- Real time simulation of an associative memory on a random access memory for base sets

7 GARBAGE COLLECTION IN SETL

SETL’s mathematical value semantics make the costly hidden copy operations a problem. The SETL optimizer resolved the hidden copy problem by adopting a conservative approach to dynamic set-updation [10]. Dynamically allocated aggregate data types with arbitrary levels of nesting resulted in difficult memory management. This results in higher runtime costs and inefficient implementation due to dynamic weak typing, massive overloading, and the prevalence of operations that rely on associative access [10].

Programs written in PASCAL, C and ADA generally do not make heavy use of the heap and the implementations of these languages typically do not perform garbage collection. Instead, functions to reclaim objects are provided to the user, for example, free and delete. Unlike the above approach, the SETL make heavy use of the heap and does implicit garbage collection to reclaim unused storage. The SETL system relies on compaction garbage collection for dynamic allocation (whereas subsequent implementation SETL2 used collections of stacks).
A garbage collection algorithm discovers and categorizes an object as garbage when it is no longer active, and reuses their storage. We define a small set of objects as root objects that are always active and other objects are active only if those objects can be reached via a path of pointers (containment by reference) or inclusions (containment by value). Reference counting, mark and sweep and copy-and-compact are widely used garbage collection methodologies. In copy-and-compact garbage collector, the cost of a single collection is proportional to the amount of memory in use (not to the amount of memory available) [13]. A compacting garbage collector changes pointer values as opposed to mark-and-sweep collector.

Since SETL does not provide the ability to return lexically scoped functions, compiler may allocate activation records on stack. SETL is a value language and not a pointer language and uses a garbage collector to reclaim unused storage. Non-atomic values are represented with pointers into the heap and values are assigned to variables by copying pointers. Sets and tuples are treated as values and reclaimed with a garbage collector. SETL uses a notation of ivariable and ovariable. While each occurrence of a variable x in which its value is used is called an ivariable, each occurrence of a variable x in which it is re-defined is called an ovariable. An ovariable o and ivariable i are said to be chained together if i and o are occurrences of the same variable x and a path exists in control flow graph from o to i, containing no assignments to x. Thus, i ∈ DU (o) and o ∈ UD (i), where UD and DU are the use-to-definition and definition-to-use maps of the traditional data-flow analysis. An ovariable o is said to be dead when DU (o) = Ø. The program is partitioned into basic blocks, which are the nodes in the program’s control flow graph G. Overwriting, Stack allocation and Area allocation is supported in the SETL garbage collector. In subsequent SETL compilers, the stack and area allocation are not indicated; but the copy optimization (overwriting update) is continued [13].

8 DIALECTS OF SETL

SETL has improved in the recent years through SETL2 and ISETL with several contemporary modifications and has potentials for future IT applications. Newer dialects of SETL also support Object oriented programming and packages. Most dialects of SETL support a sub-language to indicate how data structures are to be stored internally by the compiler. SETL2 adds to SETL syntax and name scoping closer to recent imperative languages. Application of SETL in teaching database has indicated that SETL notations seemed better than standard query language [11]. Using ISETLW, the laboratory oriented activities in academic coursework can be carried out. SETL has been also found useful in bioinformatics for the construction of customized analysis workflow from existing building blocks and public domain databases. SETL databases provide an alternative to standard SQL database [19]. Recently, the scope of potential applications of SETL for Internet Data processing has been brought out [8].

9 CONCLUSION

SETL and Pascal existed in about the same era. Though Pascal has support for simple set-theoretic representations like sets, it does not allow maps. Also, the control structures in Pascal are not as flexible as the ones in SETL. When compared to functional languages like PROLOG, SETL appears to be limited in its capabilities. Newer dialects of SETL like SETL2 can be compared to languages like Java at the object-orientation level.

SETL is a powerful, easy-to-learn language, and its highly orthogonal nature makes it suitable for rapid prototyping. SETL facilitates theorem proving, extensive data processing and transformational programming and extends support for artificial intelligence applications. Potentials of SETL in AIML open up new vistas for artificial intelligence domain [23, 24] especially with the recent advancements like object oriented support and GUI provisions in dialects of SETL.
ACKNOWLEDGEMENTS

We would like to extend our thanks to Prof. Christian Collberg, Department of Computer Science, University of Arizona, for his guidance and encouragement all through the semester.

REFERENCES

5. R K Dewar, The SETL Programming Language, 1979,
6. Various web resources at http://cs.nyu.edu/bacon/