A Comparitive Exploration of Three Unusual Languages

J Bolton

May 26, 2011

Contents

Abstract

This paper provides an introductory tutorial to three contrasting and unusual programming languages: Smalltalk, SETL, and Agda. Some prior experience in programming is assumed. The languages are highly abstract yet represent very different perspectives: the object oriented paradigm, functional programming with dependent types, and set-based imperative programming. Working with each of them at the same time will be an invigorating brain exercise and will hopefully provide some unique insights into the strengths and weaknesses of each style.

1 Introduction

Agda represents the extreme end of functional type theory based on the elegant syntax of Haskell. The language will introduce the very interesting world of dependent types and all the expressiveness that comes with them. Smalltalk is a much older language known for its pure implementation of the object oriented type system, made famous by such extremely popular languages as C++ and Java. Finally, SETL is the oldest of the three languages whose main feature is its flexible, built-in support for creating and manipulating sets and tuples.

We will start out with a very quick overview of each language, starting with the least difficult, SETL, and progressing to the most difficult, Agda. This paper will emphasize a discussion of the typing model of each language.

We will then examine three programs chosen to showcase the advantages of each of the languages. We will follow this with a larger, single program (an LL parser generator) written in all three languages to provide a direct, contrasting example among all three styles. The larger program, a top down parser, was chosen in the hope that none of the three paradigms would have a very big advantage in its implementation over the others

2 SETL

2.1 Squishing Trees of Numbers

SETL uses an imperative style of programming similar to the classic language Ada. The language has assignments, loops, conditionals, and arithmetic that shouldn't look all that foreign to those familiar with other imperative languages.

We will examine the language by walking through a couple problems that seem to particularly suit its style. We will end by examining a general

problem that we will also implement in Smalltalk and Agda later on.

The first of our problems comes from the puzzle website called Project Euler. It involves finding a maximal path through a large tree of random numbers. These aren't quite binary trees, but more like lattices, where every child node is shared to the left and right. This is the statement of the problem from the website:

> By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.
>
> ```
>    3
>   7 4
>  2 4 6
> 8 5 9 3
> ```
>
> That is, 3 + 7 + 4 + 9 = 23.
>
> Find the maximum total from top to bottom in [a triangle with 100 levels].

The strategy is to select the second highest level and 'collapse' each node by finding the largest adjacent node above it. Eventually, we will produce a single list consisting of the bottom row where each node is the maximum sum of some path that led to it.

This problem is made simpler by the fact that we only need to find the maximum sum, and we don't have to list the path.

```
   3
  7 4
 2 4 6
8 5 9 3
First, we select the row [7,4], and collapse the top level to make:
  10 7
 2  4 6
8  5 9 3
Now select the row [2,4,6] and collapse it with the top level again.
In the case of the middle node, 4, choose its maximum adjacent node (10).
 12  14 13
8  5   9  3
A final time:
```

```
20 19 23 16
Now, we simply need to fold the two-ary max function over this final list to
get our answer, which is 23.
```

Let's translate this to SETL. Our first step is to get our tree into the language. We can use SETL's read function, which takes a string of a certain format and returns a corresponding data type. For example, if we give it...

```
[[3]
[7 4]
[2 4 6]
[8 5 9 3]]
```

... we will get a tuple of tuples. In SETL, tuples are represented by brackets and can be heterogenous, but ours will contain only numbers. Tuples differ from sets in that they can contain duplicates and follow a specific sequence.

We will need to iterate over our tree, which is a tuple of tuples. We know we're done when there is only one level left in the tree. Our loop can then look like:

```
loop while #Tree /= 1
    ...
end loop;
```

The pound sign is the unary function 'length of'. Nice and terse, isn't it? Inside this loop, we need to manipulate the level that is second to the top, as planned. In SETL, we have really handy subtuple extraction similar to many other languages by simply inserting an index (or a range of indices) inside a pair of parens after the tuple.

```
Tree(1)
```

will return the top level of our tree, while Tree(2) will return the second to the top, as we need.

When we have a tuple of tuples, we can layer these subtuple extractions.

```
Tree(2)(1)
```

will return the first node of the second level of our tree. Tree(2)(#Tree(2)) will return the last node of the second level of our tree. In our loop, we will define what happens to the leftmost and rightmost nodes in the level first,

since they are a special case. For all the nodes in the middle, we have to find the maximum of the two nodes that are above and adjacent to the them. For nodes at the very beginning and end, we only need to add the node above, since they have only one adjacent node.

```
loop while \#Tree /= 1
    tree(2)(1) +:= tree(1)(1);
    tree(2)(#tree(2)) +:= tree(1)(#tree(1));
end loop;
```

Next, we need to loop through the nodes in the middle, adding the maximum of their adjacent nodes. In SETL, we can create lists with the form

```
[start..end]
```

We need to iterate from the second node in our level to the second to last node.

```
[2..#tree(2)-1]
```

will do just the thing. Let's put the loop together:

```
(for n in [2..#tree(2)-1])
  tree(2)(n) +:= tree(1)(n) max tree(1)(n-1);
end;
```

'max' is an infix binary operator. The head of the loop could also be written as 'loop for n in...', but surrounding it by parentheses means you can get rid of the 'loop' keyword.

All that's left to do is to remove the top row. Both sets and tuples in SETL are mutable, so you can remove stuff, add stuff, and monkey with them as much as you please. The binary operator 'fromb' will remove an element from the beginning of a tuple. [1] Let's put it all together.

```
read(Tree);
(while #Tree /= 1)
  tree(2)(1) +:= tree(2-1)(1);
  tree(2)(#tree(2)) +:= tree(1)(#tree(1));
```

[1] 'fromb' requires a left hand side variable to assign our discarded level to. We don't need to do anything with that variable, however, so it would be cleaner if we didn't do any assignment, but I have not seen any such function built in.

```
  (for n in [2..#tree(2)-1])
    tree(2)(n) +:= tree(1)(n) max tree(1)(n-1);
  end;
  q fromb tree;
end;
```

We have one last step. After the above loop finishes, we will be left with a single tuple. Now we need to fold 'max' over our final tuple. Fold in SETL is the wonderfully terse forward slash character.

```
print(max/Tree(1));
```