SubScript: A White Paper

Andre van Delft
February 2011

SubScript is an extension to the Scala programming language, aimed to ease developing interactive applications. Read this white paper to:

- understand the conceptual problem with the specification of control flow in currently used programming languages
- see that illustrated by an implementation of a simple application in Java
- see how easy the implementation is in SubScript
- find out the other benefits of SubScript
- learn about the technological and scientific developments leading to SubScript

As of this writing, SubScript has not been implemented, and the language definition is being finished. There is an implementation though of a predecessor language, named Scriptic, which shows the feasibility of the main SubScript constructs.

The conceptual gap

"Our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."

These wise words are from the famous paper "Goto statement considered harmful" that Edsger Dijkstra wrote in 1968[1]. In those days many programmers tended to use the "goto" statement, which too often obscured the control flow. The resulting programs were described as "spaghetti code"[2]. In the years after this paper, the use of goto statements decreased, also by improving choice and iteration constructs in programming languages. By 1980, "structured programming" had become widely accepted.

But spaghetti code returned, although this was not immediately recognized. Programs got graphical user interfaces, and program behaviour started to be driven by user generated events. At first, a "main event loop" would handle these events. Programming such an event loop was tedious, but the code was still understandable. As the user interfaces became more advanced, the coding complexity increased. For instance, time consuming tasks required special care so that they would not block user input.

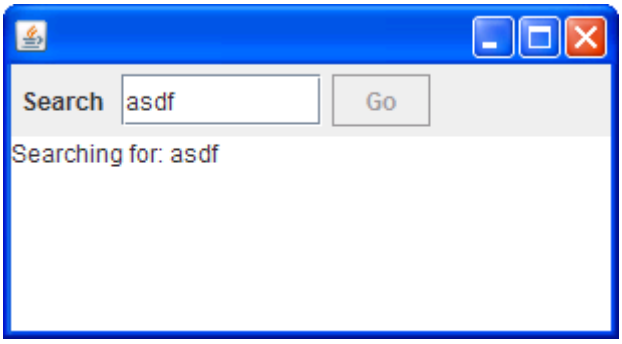
Around the year 2000 programming interactive applications had become really hard. Spaghetti code was needed to handle all kinds of input events; the screen needed to be updated in a specific thread, and background threads had to keep the application responsive. As a result, too many applications, even professional ones, freeze these days the GUI time and again.

The main cause for the trouble is that the applied programming languages offer inappropriate support for event-driven and parallel programming. Event handling and multithreading are usually implemented using dynamically created objects. Manipulating these data items largely determines the flow of control. This is much less clear than the use of explicit control flow constructs, that programming languages offer for concepts such as choice and iteration.

As Dijkstra wrote: we should *shorten the conceptual gap between the static program and the dynamic process*. This would require "first class citizens" in a programming language for event handling and parallelism.

A simple event-driven program

To view the problems with current programming languages, consider for instance a simple Java-based program to lookup items in a database, based on a search string. This example has been taken from an article about event-driven GUI programming[3].



The user can enter a search string in the text field and then press the Go button. This will at first put a "Searching…" message in the text area at the lower part. Then the actual search will be performed at a database, which may take a few seconds. Finally the results from the database are shown in the text area.
In intuitively clear pseudo-code, a search sequence could simply be described as:

```
searchSequence = searchCommand
                 showSearchingText
                 searchInDatabase
                 showSearchResults
```

The activity `searchCommand` would refer to the user pressing the Go button.

Some technical constraints hold for the implementation of such behaviour.

1. When the search button (labelled "Go") is pressed, that event must be handled.
2. Putting a text string in the text area requires executing some Java code in the so-called "swing thread": this is a special thread, present in all Java applications based on the Swing framework. It is responsible for updating the graphical user interface.
3. The search action on the database must be done in a background thread; otherwise that search would block the user interface. That would be inconvenient for the user, who would be annoyed by the seconds lasting freeze of the entire application.

These constraints would typically lead to a Java program like:

```
private void searchButton_actionPerformed() {
    showSearchingText();
    new Thread() {
        public void run() {
            searchInDatabase();
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        showSearchResults();
                    }
                }
            );
        }
    }.start();
}
```

This is very technical code, and much less clear than the previous pseudo-code. The quoted article gives an alternative, but that is still complicated and not intuitive.
In SubScript, so called scripts would implement a search sequence as follows:

```
scripts

  searchSequence    = searchCommand
                      showSearchingText
                      searchInDatabase
                      showSearchResults

  searchCommand     = searchButton
  showSearchingText = @swing: showSearchingText
  showSearchResults = @swing: showSearchResults
  searchInDatabase  = {* searchInDatabase *}
```

The first script is exactly equal to the intuitive pseudo code for the search sequence (which is not entirely coincidental). Then four refinements implement the three technical constraints:

1. The mentioning of the variable `searchButton` implies a call to a default script for a button. In this case, the default script is `action` imported from a utility library, which represents a click on the button. As a bonus, `action` will enable or disable the button whenever the application is ready to handle the related event; that functionality is not offered by the earlier presented Java code.
2. Braces `{` and `}` enclose fragments of Scala code. For a simple method call such as `showSearchingText` these braces may be omitted
3. The annotation `@swing:` ensure that the Scala methods `showSearchingText` and `showSearchingText` are executed as required in the swing thread.
4. The Scala method `searchInDatabase` is enclosed in a pair of braces with asterisks. That causes it to be executed in a background thread.

Moreover, it is easy to extend the functionality of this program. For instance, the search action may also be triggered by the user pressing the Enter key in the search text field (`searchTF`). For this purpose we can adapt the script `searchCommand`:

```
  searchCommand = searchButton + KeyEvent.VK_ENTER, searchTF
```

Here the plus symbol denotes choice, just like the semicolon denotes sequence. There are 7 other operators like these, most of which express a specific flavour of parallelism. `KeyEvent.VK_ENTER, searchTF`, represents the user pressing the Enter key when the Search text field has the focus.

Some other easy to do improvements to this program would be:

- allowing the user to cancel an ongoing search
- allowing the user to exit the program at any time, after a confirmation dialog
- disabling the search button when the text field is empty

More benefits

SubScript offers more benefits than just easy event handling and threading:

- Conciseness: the control flow specification takes usually much less space than in plain Scala. Even a single line can express significant portions of control flow
- Logic parallel behaviour: flavours of and-parallelism and or-parallelism turn out to allow elegant specification of practical behaviour patterns
- Mathematical format of script expressions: sequences, choices and parallelism are all written down as expressions in a mathematic format, with operators such as `+` and `;`. The basic syntax of such expressions is therefore already familiar. Moreover, the programmer may reason about the semantics of expressions by applying algebraic laws that hold for the operators.
- Syntactic sugar to allow specifications without the tons of semicolons, parentheses and braces that are normal in Java (but not in Scala)
- Good integration with Scala: scripts may be called from Scala, and Scala code may be executed by scripts. Scripts are class members, comparable to methods. They are inherited, and may be overridden in subclasses.
- Support for tweaking SubScript programs a notion of simulation time
- SubScript eases programming in the model-view-controller paradigm (MVC): the model is specified in plain Scala, the view is created using a GUI painter, and the controller is specified conveniently in SubScript. The resulting program structure resembles lasagne more than spaghetti.

Background

The scientific and technological advances leading to SubScript started about half a century ago.
In 1956 Stephen Cole Kleene published work on formal automata[4]. His work led to the concept of regular expressions.

Around 1960, John Backus and Peter Naur developed a notation to express context-free grammars, which was called Backus Naur Form or BNF[5]. Many variants would be developed later.

Also in 1960, the first compiler-compiler[6] was built, to be followed by a bunch of related products such as Yacc[7]. In 2008, Yacc creator Stephen C. Johnson said: "*The ideas and techniques underlying YACC are fundamental and have application in many areas of computer science and engineering. One application I think is promising is using compiler-design techniques to design GUIs - I think GUI designers are still writing GUIs in the equivalent of assembly language, and interfaces have become too complicated for that to work any more."*[8]

In 1962 the first version of the programming language Simula[9] was defined. It would become the first object oriented language. Simula also supported notions of parallelism and simulation time.

Hans Bekič tried to "develop an algebra of processes".10 His algebraic approach would later be followed up and improved by many researchers. More about this in the paper "A Brief History of Process Algebra"[11].

In 1973, Roy Campbell and Nico Haberman published the concept of path expressions: a mechanism for expressing permitted sequences of execution, inspired by regular expressions.[12]

Around 1980 some algebraic formalisms were launched:

- Communicating Sequential Processes (CSP)[13], by Tony Hoare
- Calculus of Communicating Systems (CCS)[14] by Robin Milner
- Algebra of Communicating Processes (ACP)[15] (also known as Process Algebra), by Jan Bergstra and Jan Willem Klop

Meanwhile, on the practical side, Jan van den Bos et al. developed the Input Tool Model (ITM)[16] and thereafter the Input Output Tool model (IOT)[17]. This applied the concept of path expressions to the specification of input patterns for interactive applications. IOT was implemented as a programming language extension to C, Pascal and Modula-2.

In 1987 André van Delft picked up the IOT implementation, and modified it so that not only input actions would be placed in the path expressions, but also internal actions and output actions. This way Scriptic-Pascal was formed, to be transformed into versions based on Modula-2, C, and C++. The language became little by little based on the Algebra of Communicating Processes, while offering additional constructs, e.g., for iterations and actions with a given duration.

Around 1990 Scriptic was meant to be a simulation language which also happened to be useful for the specification of GUI behaviour. It was only used in a few research projects on simulations, though. For several years Scriptic was left aside.

In 1996 a version of based on Java was made, now offering support for multithreading. Still, there was no big spaghetti coding problem to be solved for GUI programming, and Scriptic would then not have offered a good solution for it any way.

For 12 years the language was out of sight again. Meanwhile André van Delft noticed the emerging spaghetti problem in his daily work as a software developer. He adapted Scriptic so that it could better express event handling and threading issues. In 2009 this new version went open source.

In 2010 André van Delft decided to move to Scala as a base language, and to remake the language, making the syntax leaner and expressiveness bigger. Scala inspired some syntactic sugar rules that make SubScript program's easier to read and write.

Conclusion

SubScript has been based on developments in Computer Science that started over half a century ago. It seems very well applicable for developing many kinds programs that deal with input or parallelism.

References

[1] Edsger Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.
http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF

[2] http://en.wikipedia.org/wiki/Spagetti_code

[3] Jonathan Simon, "Rethinking Swing Threading", http://today.java.net/lpt/a/32

[4] http://en.wikipedia.org/wiki/Kleene

[5] http://en.wikipedia.org/wiki/Backus-Naur_form

[6] http://en.wikipedia.org/wiki/Compiler-compiler

[7] http://en.wikipedia.org/wiki/Yacc

[8] http://www.techworld.com.au/article/252319/-z_programming_languages_yacc?pp=1

[9] http://en.wikipedia.org/wiki/Simula

[10] H. Bekič: Towards a mathematical theory of processes. Technical Report TR 25.125, IBM Laboratory Vienna, 1971.

[11] J.C.M. Baeten: A Brief History of Process Algebra,
http://www.win.tue.nl/fm/0402history.pdf

[12] R.H. Campbell, N. Haberman: The Specification of Process Synchronization by Path Expressions. Springer-Verlag, Lecture Notes in Computer Science Vol. 16 (1974), 89-102
http://citeseer.comp.nus.edu.sg/context/14773/0

[13] http://en.wikipedia.org/wiki/Communicating_sequential_processes

[14] http://en.wikipedia.org/wiki/Calculus_of_Communicating_Systems

[15] http://en.wikipedia.org/wiki/Algebra_of_Communicating_Processes

[16] Jan van den Bos, Marinus J. Plasmeijer, Jan Stroet: Process Communication Based on Input Specifications. ACM Trans. Program. Lang. Syst. 3(3): 224-250 (1981)
http://portal.acm.org/citation.cfm?doid=357139.357141

[17] van den Bos, J., Plasmeijer, M. J. and Hartel, P. H. (1983) Input-output tools: A language facility for interactive and real-time systems. IEEE Transactions on software engineering, SE-9 (3). pp. 247-259. ISSN ISSN 0098-5589
http://doc.utwente.nl/55922/1/K141____.PDF