A Pascal Based Approach towards Statistical Computing

Frank Katritzke, Wolfgang Merzenich,
Rolf-Dieter Reiss and Michael Thomas

Abstract

Most statistical programming languages (like S, R or XploRe) are dynamically typed and employ an interpreter for program execution. However, the interactivity supported by that approach is usually not required for libraries which are not modified by the user. We discuss the related performance problems and suggest the usage of a strongly typed language that includes the data manipulation facilities provided by statistical languages.

We illustrate our approach by comparing the coding techniques and runtime performance of our prototypical implementation of a Pascal based statistical language with a traditional one. An outlook on a client/server environment is given where components written in our language can be utilized in various frontends.

Key Words: Statistical language; Strong typing; Component software; Graphical scripting language

1 Introduction

Statistical analyses require the use of a programmable environment if tasks beyond the limited facilities of a menu system must be performed. Common general purpose languages like C or Fortran are inconvenient for implementing statistical algorithms which often require the manipulation of complete data sets. Statistical languages, like S (Becker and Chambers 1984), R (Gentleman and Ihaka 1997) or XploRe (Härdle, Klinke and Müller 2000), have been developed with the following main characteristics:

- The handling of data sets with arbitrary lengths is supported whereby memory management is automatically performed.
- Operators and functions can be applied to complete data sets without requiring explicit loops.
- Parts of data sets can be extracted and easily manipulated.
- These languages are generally applicable, i.e., they provide the usual control structures and subroutine mechanisms of general purpose languages.

Statistical languages are usually scripting languages offering the user an interactive dialog allowing commands that are interpreted and immediately executed. Subroutines (macros) are implemented by using the same statements that are employed in an interactive analysis. Chambers (1998) points out that the resulting smooth migration path from interactive usage to serious programming is important to reduce the start-up costs for new users.

Because it is not necessary to declare types of variables and formal arguments of subroutines, these languages are often called typeless. This approach can be better characterized as "dynamically typed", because each value in these languages is associated with a data type. However, any type checking is postponed until it is actually required and the type of a variable may change during the execution of a program.

Such a behavior is necessary in interactive languages, because it would be unacceptable for a user to declare all variables before starting a session. There are also some advantages when subroutines are implemented without declaring types of formal parameters and results. As a simple example, we consider a generic function for adding numbers in the S/R notation, namely,

```
add <- function (x, y) x + y
```

This function can be used to add values of any type, for which the +-operator is defined. However, more complex algorithms usually require an explicit check of the data types of the actual arguments. Thus, true generic functions are usually possible in special cases only. One should note that the templates of the strongly typed C++ language provide similar facilities.

2 Problems with Interpreted Languages

In the preceeding lines, we described the advantages of interpreted, dynamically typed languages, yet one should be aware of two serious drawbacks. Firstly, these languages usually perform semantic checks (e.g., for the existence of identifiers) when the code is actually executed which leaves many possibilities for typing errors to be undetected.

Secondly, type checks for each operation must be performed at run-time. This must be done not only once, but every time an instruction is executed. When performing simple operations like the addition of two real numbers, most of the time for the operation will be spent on verifying the types of the arguments, even when it is clear that — because of the design of the algorithm — no other types may occur. While the additional overhead is negligible for interactive execution of commands typed on a command line, it has a serious impact on routines that require intensive calculations.

Interpreted statistical systems usually offer the inclusion of external subroutines written in a compiled language like C or Fortran as a solution to that problem. Such an approach is inconvenient in various ways: a recompilation of a module is required for every target platform. Also, the developer of a module must be familiar with technical details like parameter passing conventions and

the usage of host-specific development systems. Error handling is problematic. In addition, general purpose languages are not well suited for performing statistical computing, which is the main reason for using a special statistical language in the first place.

Ousterhout (1998) distincts between high-level scripting languages and low-level system programming languages. He suggests to use scripting languages to combine components that are implemented in a system programming language. However, there is not yet a system programming language suited for statistical computing.

Therefore, we implement a strongly typed language that is based on a general purpose language (namely, Pascal) with the extensions required for statistical computing. Our language provides vector and matrix operations like other common statistical languages. Moreover, all variables, parameters and return types of subroutines must be declared. As a result, the language can be compiled and executed efficiently. Such a language cannot be applied as an interactive immediate language for command line execution, but it serves well for the implementation of statistical components.

In the next section, we outline some details of our language, called StatPascal, describe its inclusion in a menu system and measure its runtime performance. Then, we describe shortly the design of a prototype of a component integrating environment where StatPascal components are used within a graphical scripting environment. This allows their visual combination, as alternatives to a textual scripting language.

A good runtime performance is essential to achieve the desired interactivity of our graphical programs. We demonstrate that strong typing helps to achieve this goal, while the language maintains the power of vectorized arithmetic common in statistical languages, thus fostering the rapid development of statistical routines.

3 StatPascal

In this section, we describe some basic concepts of StatPascal and its implementation and compare it with other statistical languages.

3.1 The StatPascal Language Kernel

Huber (2000) points out that a statistical programming language usually provides a complete language kernel, similar to a general purpose language, with only a few extensions for statistical computing. He makes a distinction between the language itself and the statistical and numerical libraries it provides. The latter ones are addressed as literature written in the particular language. We therefore start with a description of the kernel of our language.

Because of the clearness of its syntax and the simplicity of its compiler, we chose Pascal (Jensen and Wirth 1974) as a starting point. StatPascal supports most of the standard Pascal constructs including a unit concept for creating

libraries. One should note that the selection of a particular syntax effects only the frontend of the compiler, which can be exchanged easily.

Of course, standard Pascal does not support the typical characteristics of statistical languages that are described above. However, by adding just two data structures called *vector* and *matrix,* together with extensions to the standard operators of the language, Pascal becomes attractive in a statistical context.

Our new data structures vector and matrix may be considered as one- and two-dimensional arrays. In contrast to the standard array structure there is no need to declare a maximum size. These structures can also be used in arithmetic expressions and as arguments and return types of functions. The following technical example shows the usage of these structures. We generate a data set and display the exceedances above a threshold, followed by the solution of a system of linear equations.

```
program demo;
var x, b: vector of real;
    A: matrix of real;
begin
  x := 2 + 3 * GaussianData (100);
  writeln (x [x > 5]);
  A := MakeMatrix (combine (1.0, 2.0, -2.0,
                            3.0, 1.0,  1.0,
                           -2.0, 4.0,  5.0), 3, 3);
  writeln (invert (A) * combine ( -3.0, 10.0, 0.0))
end.
```

The function call `GaussianData (100)` returns a vector with 100 standard normal random variates. In the next line, we extract the exceedances above 5 by indexing with a boolean vector obtained from the expression `x > 5`. `MakeMatrix` creates a matrix from a vector, which is then inverted and multiplied with a real-valued vector.

The StatPascal compiler generates a code for an abstract stack machine which is interpreted at runtime. Such an approach has been frequently used in Pascal implementations (Nori, Ammann, Jensen, Naegeli and Jacobi, 1981). It has the following advantages:

1. The compiler and the runtime environment can be easily ported to another platform.

2. The resulting binaries are not system dependent.

3. The language can be easily included into a host system.

Of course, it would be desirable to generate a machine code for the host processor or to output source code of a system programming language like C. Because our language is strongly typed and does not support reflection or the insertion of code at run-time, such a compilation is a straight-forward task.

However, we demonstrate in section 3.3 that because vector and matrix operations are performed by single instructions of the virtual machine, a reasonable runtime performance can be achieved with our approach.

The language kernel described so far does not contain any graphical or statistical operations. The latter ones could be implemented in StatPascal libraries; yet, graphical output cannot be achieved that way easily. We show two ways of combining StatPascal with other systems to provide the desired operations:

1. The language kernel can be included in a software system providing data management, statistical and graphical facilities. This approach is described in the next section.

2. A standalone version of StatPascal is available; yet, it is restricted to a non-graphical statistical library that currently provides some data generation routines and estimators for normal, Student and extreme value distributions. Data handling must be performed by reading and writing files manually. This version is useful for batch processing or server tasks when no controlling terminal is attached. With some slight modifications, it is utilized in the CORBA based client/server environment described in section 4.

A more detailed description of the language and its standard library is given in the StatPascal manual (Reiss and Thomas 2005).

3.2 Embedding StatPascal in a Menu System

In the section, we describe the inclusion of StatPascal in the Xtremes package (Reiss and Thomas, 2001). Xtremes provides facilities to load and manipulate data sets, a large number of interactive plot options and statistical methods especially for extreme value and generalized Pareto distributions.

Access to the facilities of a host system is accomplished by adding predefined functions and procedures to the StatPascal language, which will be compiled to new instructions of the underlying virtual machine. These instructions can be handled in an extension of the virtual machine and are used to exchange data with the host system. Such extensions do not modify the language kernel itself; they merely enlarge the literature written in (or, in this case, for) the language.

The StatPascal kernel is implemented in C++; extensions are implemented by enhancing the compiler and virtual machine classes using the inheritance features of C++ and by linking the resulting code with the target application. This approach limits the inclusion of the kernel to systems written in languages that can be linked with C++ code. It would therefore be desirable to encapsulate the language kernel using a component framework like CORBA (Object Management Group, 1995) and to install extensions to the virtual machine by registering callbacks.

As an example, we show how StatPascal displays a scatterplot of a bivariate data set loaded in Xtremes and adds a polynomial regression line (see Fig.

1). Although these options are available within the menu system, it can be convenient to script such a task. The required code is listed next.

```
program regression;

var x, y, c: vector of real;

begin
  x := columndata (1);
  y := columndata (2);
  ScatterPlot (x, y, 'Scatterplot');
  c := PolynomialRegression (x, y, 3);
  (* supporting points for regression polynomial *)
  x := realvector (min (x), max (x), 150);
  y := (((c [4] * x) + c [3]) * x + c [2]) * x + c [1];
  Plot (x, y, 'Scatterplot', 'Regression Polynomial')
end.
```

The functions `columndata` (to retrieve the columns of the active data set of the menu system), `plot` and `scatterplot` have been added to pass the required data between StatPascal and Xtremes and to perform the graphical output. It is up to the embedding system to further facilitate the usage of StatPascal programs, e.g. by making them available as menu options.

3.3 Runtime Performance

In this section, we compare the runtime performance of StatPascal and the R system. Since our main interest is in the language kernel, we have to avoid calls to sophisticated predefined functions. Instead, we implement a simple routine using only vectorized operations and basic functions. We choose the Hill estimator (an estimator for the reciprocal shape index of the tail of a distribution), which is given by

$$\hat{\alpha}_{n,k} = \frac{1}{k} \sum_{i=1}^{k} \log \frac{x_{n-i+1:n}}{x_{n-k:n}}, \quad k = 1, \dots, n-1.$$

$x_{n-k:n} \leq ... \leq x_{n:n}$ denote the $k+1$ largest values of the data $x_1, ..., x_n$.

The following implementations expect a univariate data vector $(x_1, \dots, x_n)$ and return a vector of the Hill estimates $(\hat{\alpha}_{n,1}, \dots, \hat{\alpha}_{n,n-1})$. We start with the R/S version.

```
hill <- function(x) {
  x <- log(rev(sort(x[x > 0])))
  n <- length(x)
  cumsum(x)[-n]/(1:(n - 1)) - x[-1]
}
```

The StatPascal equivalent is given next (*realvector* is a predefined data type equal to *vector of real*).

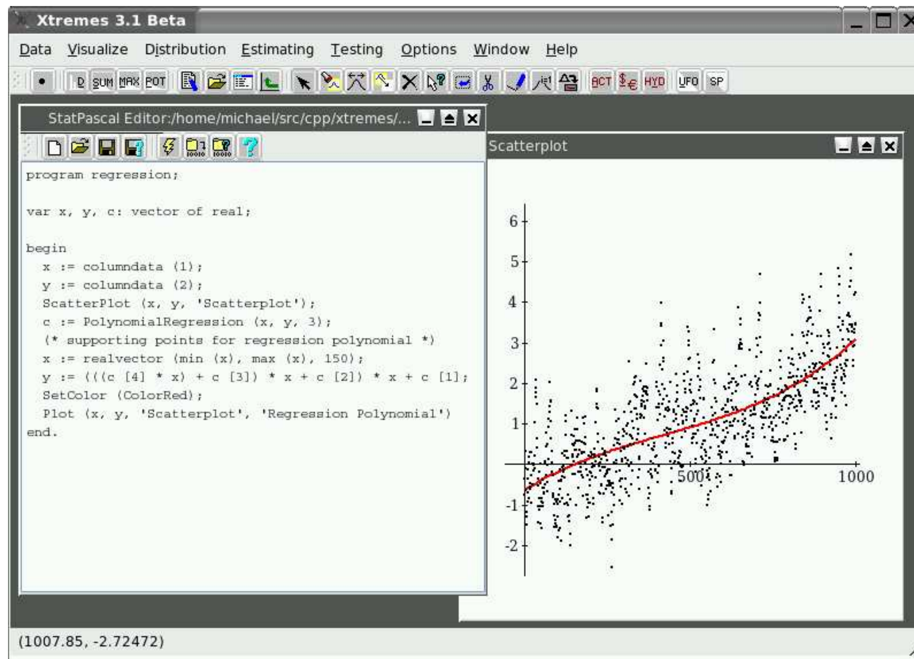

Figure 1: StatPascal within Menu-based Statistical System.

```
function hill (x: realvector): realvector;
var n: integer;
begin
  x := log (rev (sort (x [x > 0])));
  n := size (x);
  return (cumsum (x)) [1..(n-1)] / (1..(n-1)) - x [2..n]
end;
```

One can see that R and StatPascal provide similar vectorized operations. Especially, no explicit loop is required to evaluate the estimator simultaneously for all values of k. There is a significant difference in runtime performance. Table 1 shows the execution times for 10000 calls to the Hill estimator, applied to a newly simulated data set in each call (we used R 1.9, StatPascal and the GNU C compiler version 3.3.1 with full optimizations under Linux). The runtime for StatPascal also includes the compilation of the program, while R had already been started and parsed the routines. The C version was written carefully eliminating loops whenever possible. It consists of 78 lines, including an implementation of the quicksort algorithm.

StatPascal clearly outperforms the R system, at the cost of only a small declarative overhead in the implementation of the estimator. The increased performance is important for long running simulations and interactive visual-

n	R	SP	C	SP (loops)	Pascal
20	2.05	0.095	0.046	1.04	0.21
50	2.31	0.21	0.11	2.91	0.52
100	2.74	0.40	0.23	6.31	1.08
500	6.46	2.00	1.26	37.3	5.66
1000	11.2	4.08	2.63	81.0	11.5

Table 1: Time (in seconds) for 10000 evaluations of the Hill estimator on 2.4 GHz Intel Xeon processor.

izations.

One also recognizes that the performance of the virtual machine is unsatisfactory if the vector oriented extensions are not used; yet, it would be possible to generate code for a real processor, thus increasing runtime performance considerably. The last two columns of Table 1 show the execution time of an unoptimized standard Pascal implementation of the Hill estimator (not utilizing any vectorized expressions), under StatPascal and the fpc Pascal compiler.

4 A Component Integrating Statistical Environment

As a descendant of a classical system programming language, StatPascal is not suited for interactive use. We therefore need some other means to provide interactivity. In this section, we present a prototype of a client/server based component integrating statistical environment (called Risktec) where StatPascal programs are utilized.

We use CORBA as a middleware to manage the communication between clients and servers. The system is, therefore, open for the provision and usage of components in other softwares.

It seems important to hide the complexities of CORBA from users of the system. Within the Risktec environment, there is currently only a single interface definition for the components. They basically implement a function $f : S_1 \times \cdots \times S_n \rightarrow T_1 \times \cdots \times T_m$ and provide methods to inform a client about the number and data types of the arguments and results. Such a minimal interface serves well for numerical computations and visualizations.

Fig. 2 provides an overview of our environment. One can see that Xtremes provides a component server (consisting of a factory object (Gamma, Helm, Johnsson and Vlissides 1995)) that registers prototypes of components that follow our fixed interface specification. Clients use the component server by requesting named prototypes, which are then cloned and returned via a CORBA object reference. Moreover, a shared library (DLL) provides access to the components by means of a procedural interface, allowing the usage of StatPascal components from systems like MS Excel. More technical details including a

complete interface specification can be found in the RiskTec manual (Reiss and Thomas 2005).

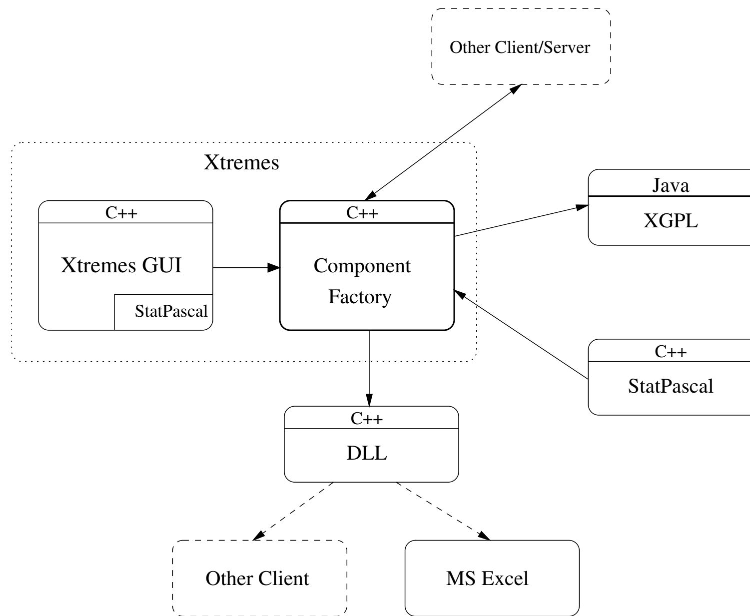

Figure 2: Client/Server environment RiskTec.

XGPL (Xtremes Graphical Programming Language, see Thomas and Reiss (2000) for the description of an early prototype) is a graphical scripting environment that allows the interactive combination of components by employing a graph editor. Nodes within the graph represent operations, while the edges determine the flow of data and thus the dependencies between nodes. XGPL provides interactive features like sliders which invalidate and recalculate the dependent nodes when an input value changes.

The implementation of a StatPascal server component is simple. A special program header defining the name of the component and its parameters is required:

```
component name;
inports varlist;
outports varlist;
```

Inports and outports define the input and output parameters of the component; they correspond to the sets S_i and T_i in the above function f. Before executing the component, the caller sets the input parameters, which become available as global variables within the program. The values assigned to the output variables at the end of the program are returned to the caller.

As a simple yet complete example, we retrieve a univariate data set containing daily exchange rates from a database, calculate the returns (i.e., the daily relative changes of the prices) and visualize them by means of a kernel density estimator. The bandwidth can be modified interactively by using a slider. Figure 3 show the graphical script and its output.

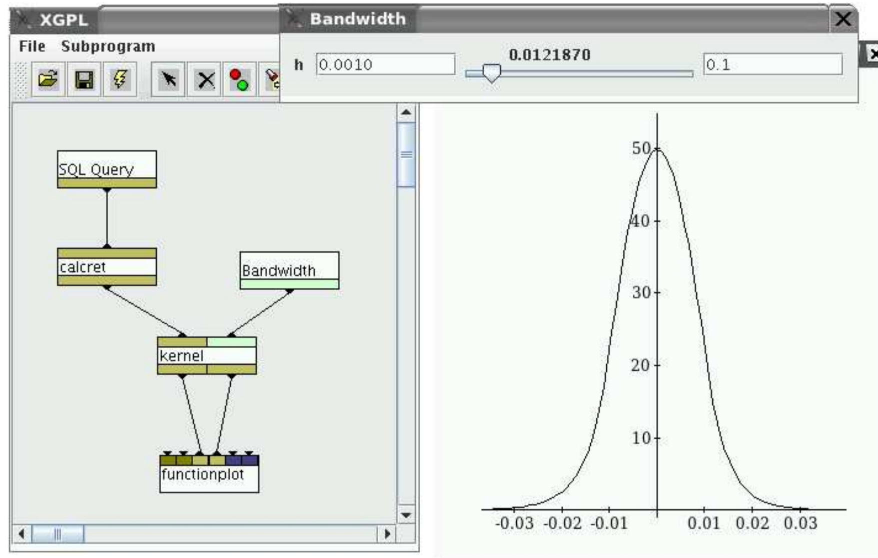

Figure 3: StatPascal Components in Graphical Script.

The SQL query node and the slider are implemented within the XGPL system. We use two StatPascal components (calcret and kernel) to calculate the returns and the kernel density. The kernel node returns the supporting points for a plot of the estimated kernel density, which is visualized using a plot component provided by Xtremes.

The complete code for the kernel density component is shown next.

```
component kernel;

inports z: realvector; h: real;
outports x, y: realvector;

const n = 100;

function kerneldensity (t: real): real;
    var zs: realvector;
    begin
        zs := (z − t) / h;
```

```
        zs := zs [(-1.0 < zs) and (zs < 1.0)];
        return sum (1.0-zs**2) * 3.0/(4.0*h*size(z))
    end;

begin
    x := realvect (min (z), max (z), n);
    y := kerneldensity (x)
end.
```

The XGPL system visualizes the data types of the input and output parameters with different colors. Note that the function `kerneldensity,` which defines a mapping between real values, is applied to a real vector, resulting in the automatic generation of a loop over its components.

5 Conclusion

By combining vectorized operations from statistical programming languages and strong typing from system programming languages, one can create a statistical language with a good runtime performance. Such a language is well suited to implement statistical components which may be combined using menu oriented systems, interactive textual scripting languages or graphical programming environments.

References

[1] Becker, R.A., Chambers, J.M. (1984), *S: An Interactive Environment for Data Analysis and Graphics,* Monterey, California: Wadsworth.

[2] Chambers, J.M. (1998), "Computing with Data: Concepts and Challenges," Technical Report, Bell Labs.

[3] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns,* Reading, Massachusetts: Addison-Wesley.

[4] Gentleman, R., and Ihaka, R. (1997), "The R Language," in *Proceedings of the 28th Symposium on the Interface,* eds. L. Billard and N. Fisher, The Interface Foundation of North America.

[5] Härdle, W., Klinke, S., and Müller, M. (2000), *Xplore Learning Guide,* Berlin: Springer.

[6] Huber, P.J. (2000), "Languages for Statistics and Data Analysis," *Journal of Computational and Graphical Statistics,* 9, 600-620.

[7] Jensen, K., and Wirth, N. (1974), *PASCAL - User Manual and Report,* Springer.

[8] Nori, K.V., Ammann, U., Jensen, K., Naegeli, H.H., and Jacobi, Ch. (1981), ”Pascal P implementation notes,” in *Pascal – The Language and its Implementation*, ed. D.W. Barron, Chichester: Wiley.

[9] Object Management Group (1995), *The Common Object Request Broker: Architecture and Specification.*

[10] Ousterhout, J.K. (1998), ”Scripting: Higher Level Programming for the 21st Century,” *IEEE Computer,* 1998, 23-30.

[11] Reiss, R.-D., and Thomas, M. (2005), *StatPascal: User Manual and Reference,* http://www.xtremes.de/xtremes/spman.pdf

[12] Reiss, R.-D., and Thomas, M. (2005), *RiskTec User Manual,* http://www.xtremes.de/xtremes/risktec.pdf

[13] Reiss, R.-D., and Thomas, M. (2001), *Statistical Analysis of Extreme Values,* Basel: Birkhäuser.

[14] Thomas, M., and Reiss, R.-D. (2000), ”Graphical Programming in Statistics: The XGPL prototype,” in *Classification and Information Processing at the Turn of the Millenium,* eds. R. Decker and W. Gaul, Berlin: Springer.