Rascal
the Advanced Scientific CALculator

Internal Moduleguide

Sebastian Ritterbusch

23rd May 2001

1 Introduction

Rascal was designed with modularity in mind. Existing C or C++ code should be easy to add to the system and then be accessible and usable through Rascal. This guide will explain how to add new modules to Rascal.

2 Concept

Rascal offers a parser together with basic datatypes. Functions are defined on these datatypes and provide the basic functionality. Modules can add new datatypes or new functions or just overload existing functions, even with different resulting type if necessary. Modules should provide documentation which will be included into the user manual. Modules should just depend on the generic datatypes so they can combined arbitrarily.

Modules are registered in the file *value.in*, which is the top-level *in*-file. In the directory *modules* there are the *in*-files for the different modules along with all other files necessary for the different modules. The *in*-files specify which other files are needed for this module, define new datatypes and functions for them or basic types. Based on these files *parsevalue* generates a C++ class *value* in *value.out* that incorporates all different datatypes, along with *value.cpp* that has the necessary static functions. Besides that the documentation is prepared in *value.tex* as well as the files *value.fn** which are used to export function definitions to the parser.

In general the class *value* should behave like all the datatypes included into the class, thus in general you won't have to change existing code. But there are some small things that are different: The comparism-operators do not return *integer* or *bool* but *value* which you currently have to cast to *integer* using the *asINTEGER()* member method.

The parser which is defined in *hoc.y* and *scan.l* is completely separated from the modules and along with the symbol-table which is defined in the *symtab*-files this is the core structure of Rascal.

3 Creating a new Module

The minimal requirement for a new module is just an *in*-file as shown in figure 1, but generally a module will consist of each a *cpp*, *hpp*, *in* and *tex*-file; as an example an *in*-file for a Taylor arithmetic is in figure 2. The *in*-files are split into sections, which don't have to be in a specific order, but it might be useful to have the newly defined types and corresponding files in the top of the file. Some Sections are lists others have just one parameter.

3.1 The Type Section

This list defines new datatypes; first the internal name which should be in capitals by convention, the second word specifies the representation in C++. Non of the two words may contain spaces and it is advisable to generally typecast templates to single words, like *valuetaylor* is nothing else as *taylor<value>*, this should be done in the corresponding *hpp*-file. If any other *#include*s are needed, these should also be *#include*d in the *hpp*-file. In case you just want to add a C++ function on existing datatypes, you might not need this section.

3.2 CPP-, HPP-, TEX-Section

Here you can specify the corresponding files. These are optional like all other sections and again there mustn't be any spaces in the paths. The *hpp*-file should contain necessary *#include*s and *typedef*s, the *cpp*-file additional necessary functions. The *tex*-file should contain user-documentation and will be linked to all other documentation. Therefore the *tex*-file should start with a new *\subsection{Modulename}* and explain how the module works along with examples. The command *make doc* will update the user guide.

3.3 Unary Functions

There are four kinds of unary functions:

- Constructors, to create expressions of the new type using existing types
- Unary operators (like "negative" *operator-* or "logical not" *operator!*); be aware that C++ uses the exclamation mark "!" where Rascal uses the tilde "~" as the exclamation mark is used for factorials (for which Rascal calls the unary function *fac*).
- Usual functions that compute something and will be available in Rascal
- The output function, which is needed to define how the new type can be printed

Each line consists of a function, the first word is the name of the function, the second the return type, the third the type of the operand and the fourth defines the code that does the work. Again no spaces are allowed within words, thus complicated functions should be in the corresponding *cpp*-file, which are called here. The result and operand-type have to be defined before and in the C++ part the value of the operand is in the variable *a*, which is of corresponding C++ type of the operand type. Instead of a type-name a wildcard "*" as an operand type. Then this overload will be used if there is no other overload fits better. Then the C++ code mustn't use the variable *a*, which type would be undefined, but the variable *va* of the C++ type *value*.

If a function is defined a second time with the same type in the argument, the last one will be used; thus as an example all generic methods can be overridden. An example for this is in the *ifraction*-module, where the division between two *integer*s was overridden to have a constructor for *ifraction*s.

3.3.1 Implemented operators and predence

The following table shows the predence of the operators and on which C++ functions they are mapped to.

Operator	C++-Function
=	
?:	
<,<=,>,>=	less, lesseq, greater, greatereq
==,!=	eq,neq
&	And
$\vert$	Or
+,-	add, sub
-	neg
*,/,%	mul, div, mod
ˆ,',!,˜	pow, transpose, fac, Not
·(·)	cell

To simplify the use of the *value*-class, the usual operators will be defined as front-ends to these functions where possible. Thus you may add two *value*-variables as $a + b$ and this will invoke the corresponding add(a , b). Note that some functions are upper case when the lower case version is a key-word in C++.

3.4 Binary Functions

Here binary functions and binary operators can be defined. To this section the same rules apply like for unary functions, only that there are now two operands that follow the result-type. A special function is the *cell*–function which is being invoked when a user applies the parentheses to an expression of this type. If the user invokes *cell* two-dimensionally, like $A(1, 2)$ then this is being mapped to $cell(cell(A, 1), 2)$.

3.5 Implicit Casts

It would be a bit tedious and sometimes even impossible to define all functions for all datatypes, but p.e. one expects that fractions can be used with intervals, even if both modules don't know of each other. This can be achieved if rascal gets informations on how to convert datatypes into each other. In above example the best would be to define how fractions can be converted to doubles. But it has to be said that casts take time and for the common cases specialized functions should be defined. Reasonable implicit casts for the *ifraction*-module are shown in figure 3. As a source object also a wildcard may be used, but this may lead to unexpected results. Generally casts that are defined later are being preferred, thus you might need to redefine casts you are not planning to override.

3.6 Procedures

Procedures are functions that return a value but do not take any arguments. This concept is helpful for help messages or internal status reports.

3.7 Special Functions

Besides the operators there are other special named functions that can be overidden:

3.7.1 `parseInteger VALUE STRING`

As soon the scanner identifies an Integer, this function is being called with the string of the number as argument. If the default conversion to integers does not suit your needs, override this function with a suitable method taking a string and returning a value.

3.7.2 `parseDouble VALUE STRING`

This function should convert a string that was identified as a floating point number into a value. If the standard conversion to double does not satisfy you, override this function.

4 Questions?

Please send questions, ideas, hints, critics and congratulations to *rascal@ritterbusch.de*.

```
TYPES
MINI    int
END

UNARYFUNCTIONS
mini            MINI    INTEGER a
output          STRING  MINI    a?string("1"):string("0")
END

BINARYFUNCTIONS
and     MINI    MINI    MINI    a&b
or      MINI    MINI    MINI    a|b
END

PROCEDURES
one     MINI    1
END
```

Figure 1: A minimalistic Module

```
CPP     modules/valuetaylor.cpp
HPP     modules/valuetaylor.hpp
TEX     modules/valuetaylor.tex

TYPES
TAYLOR  valuetaylor
END

UNARYFUNCTIONS
taylor     TAYLOR  MATRIX  (matrixtotaylor(a))
output     STRING  TAYLOR  (output(a))
END

BINARYFUNCTIONS
add     TAYLOR  TAYLOR  TAYLOR  (a+b)
add     TAYLOR  *       TAYLOR  (va+b)
sub     TAYLOR  TAYLOR  *       (a+vb)
sub     TAYLOR  TAYLOR  TAYLOR  (a-b)
sub     TAYLOR  *       TAYLOR  (va-b)
sub     TAYLOR  TAYLOR  *       (a-vb)
mul     TAYLOR  TAYLOR  TAYLOR  (a*b)
mul     TAYLOR  *       TAYLOR  (va*b)
mul     TAYLOR  TAYLOR  *       (a*vb)
div     TAYLOR  TAYLOR  TAYLOR  (a/b)
div     TAYLOR  *       TAYLOR  (va/b)
div     TAYLOR  TAYLOR  *       (a/vb)
cell       VALUE   TAYLOR  INTEGER (b<=a.dim()&&b>0)?a(b-1):value()
END
```

Figure 2: Example for an *in*-file of a Taylor Arithmetic

```
IMPLICITCAST
IFRAC   INTEGER (ifraction(a))
DOUBLE  IFRAC   (double(a.num)/a.den)
END
```

Figure 3: Implicit cast section of *ifraction.in*