A Friendly Introduction to RGP

Oliver Flasch

4 March 2013 (RGP release 0.3-1)

RGP is genetic programming system based on, as well as fully integrated into, the R environment. The system implements classical tree-based genetic programming as well as other variants including, for example, strongly typed genetic programming and Pareto genetic programming. It strives for high modularity through a consistent architecture that allows the customization and replacement of every algorithm component, while maintaining accessibility for new users by adhering to the "convention over configuration" principle. Performance critical sections have efficient implementations in C, making the system suitable for real-world application. Typical GP applications are supported by well-known R ideoms. For example, symbolic regression via GP is supported by the same "formula interface" as linear regression in R.

This text provides a friendly introduction to RGP, a flexible system for genetic programming (GP) in the R environment for statistical computing. After section 1 introduces GP in the abstract and section 2 sets the stage with typical applications of GP in general and RGP in particular, section 3 outlines the range and depth of RGP's features. RGP is a large package that can be daunting for the first-time user. To help getting started, section 4 provides a set of hands-on tutorials, beginning with simple tasks, including getting RGP up and running in an existing R installation, up to advanced topics like strongly typed genetic programming. The outlook in section 5 gives hints on where to go from here, including references to GP literature as well as RGP's comprehensive online documentation and web resources.

1 Genetic Programming

GP is a collection of techniques from evolutionary computing (EC) for the automatic generation of computer programs that perform a user-defined task [Poli et al., 2008, Banzhaf et al., 1998]. Starting with a high-level problem definition, GP creates a population of random programs that are progressively refined through variation and selection until a satisfactory solution is found.

An important advantage of GP is that no prior knowledge concerning the solution structure is needed. Another advantage is the representation of solutions as terms of a formal language (symbolic expressions), i.e. in a form accessible to human reasoning. The main drawback of GP is its high computational cost, due to the potentially infinitely large search space of symbolic expressions. On the other hand, the recent availability of fast multi-core systems has enabled the practical application of GP in many real-world application areas. This has lead to the development of a variety of software frameworks for GP, including DataModeler, Discipulus, ECJ, Eurequa, and GPTIPS.

All of these systems are complex aggregates of algorithms for solving not only GP specific tasks, such as solution creation, variation, and evaluation, but also more general EC tasks, like single- and multi-objective selection, and even largely general tasks like the design of experiments, data pre-processing, result analysis and visualization. Packages like Matlab, Mathematica, and R [R Development Core Team, 2009] already provide solutions for the more general tasks, greatly simplifying the development of GP systems based on these environments and also lowering the barrier of entry for users who already know the underlying package.

RGP[1] is based on the R environment for several reasons. Firstly, there seems to be a beneficial trend towards employing statistical methods in the analysis and design of evolutionary algorithms, including modern GP variants [Sun et al., 2009, Bartz-Beielstein et al., 2010]. Secondly, R's open development model has led to the free availability of R packages for most methods from statistics and many methods from EC. Also, the free availability of R itself makes RGP accessible to a wide audience. Thirdly, the R language supports "computing on the language", which greatly simplifies symbolic computation inherent in most GP operations. In addition, parallel execution of long-running GP runs is easily supported by the R package.

[1] The RGP package and documentation is available at `rsymbolic.org`.

2 Application Areas

GP in general, and RGP as a modular GP system in particular, has a wide array of possible application areas. Basically, GP is a evolutionary search heuristic for arbitrary symbolic expressions, i.e. mathematical or logical formulas.. A non-exhaustive list of RGP-applications include:

- *Symbolic Regression:* Given a set of measurement data divided into dependent and independent variables, symbolic regression can discover the functional relationship between dependent and independent variables. This relationship is represented as a symbolic expression, which can be used to gain insight into the data-generating process or system (system identification), and as a model to predict the values of dependent variables for unseen values of independent variables (intra- and extrapolation). Figure 1 provides a simple example.
- *Feature Selection:* Not all independent variables must have an influence on the values of the dependent variables. In many practical applications, only a small subset of independent variables affect the dependent variables. The task then is to identify this subset, which can be done by GP in a very robust fashion.
- *Automatic Programming:* As computer programs are symbolic expression, GP can be used for automatic programming, which explains the name of the method. This requires a set of program building blocks and a fitness function that assigns a numerical

quality measure to each candidate program. For small programs describing core algorithm components, this approach already works in practice.

- *General Expression Search:* The applicability of GP even goes beyond automatic programming. The method can be used to discover all structures that are representable by symbolic expressions of moderate complexity. Examples include electrical circuits, antenna designs, processing networks in manufacturing and logistics, and many others.

The RGP system is flexible enough to be applied in nearly all possible GP application areas. It already has been successfully applied in such diverse areas as support vector machine kernel generation for machine learning, surrogate model ensemble generation for engineering optimization, and time series prediction for water resource management applications.

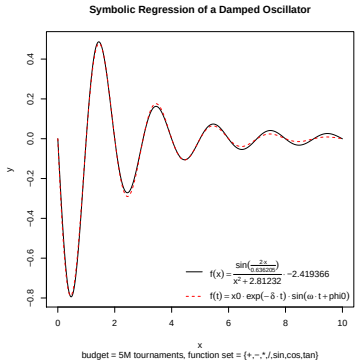

Figure 1: Symbolic regression of the governing law of a damped oscillator: RGP enables symbolic regression via genetic programming. This example shows how RGP is used to find the governing physical law of a damped oscillator. In contrast to other regression methods, the solution is expressed as a mathematical formula accessible to human interpretation and validation. In this figure, the true oscillator law and behaviour are shown in dashed red, the solution found by RGP is shown in solid black.

3 Features

To give an idea of the extend and limits of RGP's feature set, this section provides an non-exhaustive overview of the system. Detailed documentation of all functionality, including examples, can be found in the online help of the package.

3.1 Solution Representation

RGP represents candidate solutions, i.e. GP individuals, as R expressions that can be directly evaluated by the R interpreter. This allows the whole spectrum of functions available in R to be used as building blocks for GP. Because R expressions are internally represented as trees, RGP may be seen as a tree-based GP system. However, the individual representation can be easily replaced together with the associated variation and evaluation operators, if an alternative representation is found to be more effective for a given application [**?**].

Besides classical (untyped) GP, strongly typed GP is supported by a type system based on simply typed lambda calculus [Barendregt et al., 1992]. A distinctive feature of RGP's typed tree representation is the support for *function defining subtrees*, i.e. anonymous functions or lambda abstractions. In combination with a type system supporting function types, this allows the integration of common higher order functions like folds, mappings, and convolutions, into the set of GP building blocks, greatly increasing RGP's applicability in many "non-classical" GP application areas.

RGP also includes Rrules, a rule based translator for transforming R expressions. This mechanism can be used to simplify GP individuals as part of the evolution process as a means the reduce bloat, or just to simplify solution expressions for presentation and later use. The default rule base implements simplification of arithmetic expressions. Rrules can be easily extended to simplify expressions containing user-defined operators and functions.

3.2 GP Operators

RGP provides default implementations for several initialization, variation, and selection operators. The system offers clear interfaces for user-defined operators, as well as the possibility to replace the evolutionary algorithm used for GP search with user defined variants, without the need to rewrite other functionality.

Initialization Individual initialization can performed by the conventional grow and full strategies of tree building. When using strongly-typed GP, the provided individual initialization strategies respect type constraints and will create only well-typed expressions. Initialization strategies may be freely combined, e.g. to implement the well known ramped-half-and-half strategy.

Variation RGP includes classical and type-safe subtree crossover operators. Also, several classical and type-safe mutation operators are provided. The variation pipeline can be freely configured by combining several mutation and recombination operators to be applied in parallel or consecutively, with freely configurable probabilities.

Selection The system provides several single- and multi-objective selection operators. Other selection strategies can be easily added by the user. Multi-objective selection is supported via the EMOA package.[2] The multi-objective search strategy optimizes solution quality while, at the same time, controlling solution complexity and population diversity. For this purpose, RGP implements multiple complexity measures for GP individuals.

[2] The EMOA Evolutionary Multi-objective Optimization Algorithm toolbox for R is available at `http://git.datensplitter.net/cgit/emoa`.

3.3 Analysis and Visualization

The RGP system provides tools for the analysis and visualization of GP individuals and populations. GP individuals, i.e. symbolic regressions, can be visualized as trees (in multiple levels of detail), as formulas in mathematical notation, as points in a Pareto plot, or as plots of their input/output behaviour. GP populations can be visualized as forests of schematic trees, as Pareto plots, or as variable presence charts.

As RGP is based on R, a vast array of statistical tools for analyzing GP individuals, GP populations and GP system performance are readily available. For example, integration with the SPOT package for sequential parameter optimization allows the automatic tuning of critical GP algorithm parameters. The RGP online documentation provides examples for typical applications of each visualization and analysis technique.

Although RGP is basically a command-line driven system, like the underlying R package, graphical user interfaces are provided where they ease interaction and exploration. The graphical user interface for symbolic regression (see figure 2) allows direct manipulation of the most important GP parameters.

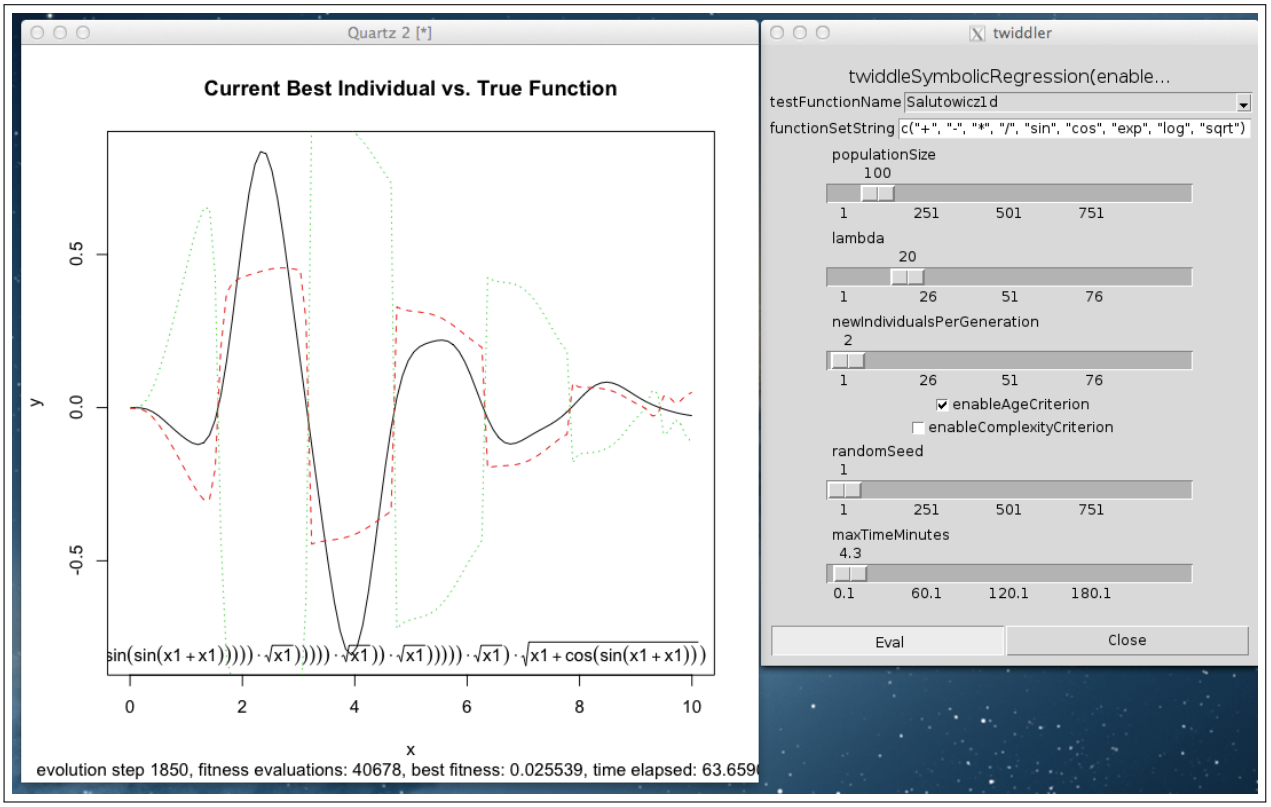

Figure 2: RGP's graphical user interface for symbolic regression: Although RGP is basically a command-line driven system, much like the underlying R environment, graphical user interfaces are provided where they ease interaction and exploration. The graphical user interface for symbolic regression allows the direct manipulation of the most important GP parameters.

4 Tutorials

To help getting started with RGP, this section provides a set of hands-on tutorials, beginning with simple tasks, including getting RGP up and running in an existing R installation, up to advanced topics like strongly typed genetic programming. All tutorials are meant to be followed stepwise in a running R session.

4.1 Installation

RGP is available as an R package on the comprehensive R archive network CRAN, making installation extremely simple. To install RGP and all it's dependencies, issue the following command in a running R session:

```
> install.packages("rgp")
```

A prompt will appear asking to select a CRAN mirror will appear if it is the first time an R package is installed in your R installation. Just select a mirror location near you. The installation of RGP may take some time, as dependencies are downloaded and compilation steps are performed.

4.2 Getting Started

This tutorial provides an interactive walkthrough of solving a simple symbolic modelling problem with GP. Only basic low-level RGP

functionality is used, high-level convenience functions are intentionally avoided to make each step in the modelling process clear and explicit.

In this first example, we configure RGP to create polynomial approximations of the sine function. To make RGP's functionality available in a running R session, the package has to be loaded via the `library` command:

```
> library("rgp")
```

Defining the GP *Search Space* In RGP, candidate solutions are represented as regular R functions. The bodies of these functions are build from a set input variables, a set of constants, and a set of function symbols. These members of these sets are often referred to as GP building blocks. In other words, these three sets define the symbolic expression search space.

As our example task is the approximation of the sine function with polynomials, we create a function symbol set containing only addition, multiplication, and subtraction.

```
> functionSet1 <- functionSet("+", "*", "-")
```

We then create a set of input variables containing just the symbol x. Thereby we restrict the search space to univariate functions, i.e. function of one variable:.

```
> inputVariableSet1 <- inputVariableSet("x")
```

Finally, we create a set of constants. Constants are not created directly, but via constant factory functions. Each time a constant has to be created during GP search, RGP calls a constant factory function. Here we use a single constant factory that returns constants from a normal distribution:

```
> constantFactorySet1 <- constantFactorySet(function() rnorm(1))
```

Defining the Fitness Function The fitness function, or objective function, associates a numerical fitness value to a candidate solution. RGP relies on the fitness function to direct its evolutionary search. The fitness function defines the problem to be solved by GP. As already mentioned, in this example, we will use RGP to find functions approximating the sine function in the interval `interval1` $[-\pi, \pi]$. We sample this interval in steps of size 0.1:

```
> interval1 <- seq(from = -pi, to = pi, by = 0.1)
> fitnessFunction1 <- function(f) rmse(f(interval1), sin(interval1))
```

By default, RGP minimizes fitness values, so lower values should be associated with better solutions. Here, we use the root mean error (RMSE) of a given sine approximation against the true sine function as a fitness function.[3]

[3] The problem defined here is a typical symbolic regression problem. RGP also features a simple interface for symbolic regression, which is introduced in the next tutorial on symbolic regression.

Performing the GP *Run* We are now ready to start the search for symbolic expressions of good fitness values, i.e. start the GP run:

```
> set.seed(1)
> gpResult1 <- geneticProgramming(functionSet = functionSet1,
+                                 inputVariables = inputVariableSet1,
+                                 constantSet = constantFactorySet1,
+                                 fitnessFunction = fitnessFunction1,
+                                 stopCondition = makeTimeStopCondition(5 * 60))
```

The first command will first set R's random number generator seed to a defined value (here 1) to create reproducible results. Then, we perform a GP run that stops after 5 minutes and store the results of this run in the R variable `gpResult1`. The GP runtime budget can be adjusted by changing the parameter to `makeTimeStopCondition`.

Analyzing the Result Population Finally, we select the best sine approximation found during the GP run:

```
> bestSolution1 <- gpResult1$population[[which.min(gpResult1$fitnessValues)]]
```

We then create a plot of the approximation created by *bestSolution*1 versus the true sine function (see figure 3):

```
> plot(y = bestSolution1(interval1), x = interval1, type = "l",
+   lty = 1, xlab = "x", ylab = "y")
> lines(y = sin(interval1), x = interval1, lty = 2)
```

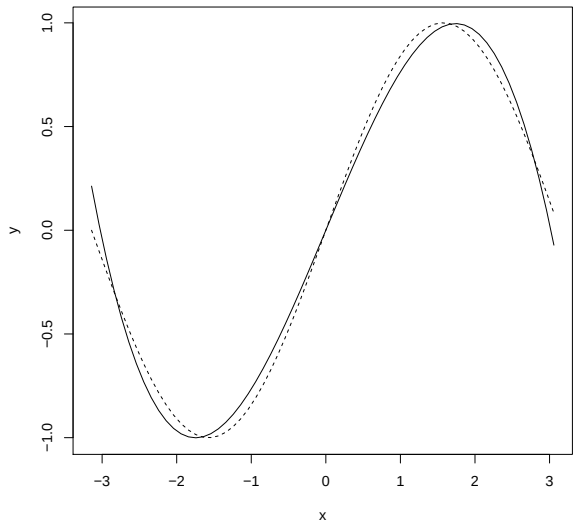

Figure 3: Best GP-generated polynomial approximation (solid line) versus true sine function (dashed line).

Next Steps This concludes this basic tutorial. Of course there is much room for experimentation. For example, we could change the

members of `functionSet1` by adding the cosine function `cos` and observe the effects on the GP result.

Please note that we omitted many RGP convenience functions that would have made this particular example much shorter. Also note that in this example, we only dealt with a single optimization criterion and functions defined on real numbers. RGP also supports multi-objective optimization of functions that work on arbitrary data types. The next tutorials give practical examples of some of these more advanced features.

4.3 Symbolic Regression

RGP offers convenience functions to simplify the solution of common GP tasks. This tutorial shows how to use the `symbolicRegression` function to solve symbolic modelling and regression tasks with minimal configuration work.

Theme of this tutorial is the discovery of a mathematical formula describing the behaviour of a physical system based on measurement data, i.e. symbolic regression. For sake of simplicity and clarity, we generate this data by applying a text-book formula describing a damped pendulum. The task of RGP then becomes the rediscovery of that formula and the numerical values of the formula's parameters.

Task Definition The formula below, given as an R function factory, represents a damped pendulum. The arguments are the starting amplitude A_0, gravity g, pendulum length l, phase ϕ (phi), damping factor γ (gamma), and radial frequency ω (omega).

```
> makeDampedPendulum <- function(A0 = 1, g = 9.81, l = 0.1, phi = pi, gamma = 0.5) {
+   omega <- sqrt(g/l)
+   function(t) A0 * exp(-gamma * t) * cos(omega * t + phi)
+ }
```

This function factory can now be used to generate functions describing the deflection of concrete pendulums of different specifications at a certain point in time t:

```
> pendulum1 <- makeDampedPendulum(l = 0.5)
> pendulum2 <- makeDampedPendulum(l = 1.2, A0 = 0.5)
```

The deflection of these pendulums can easily be plotted against time (see figure 4):

```
> interval1 <- seq(from = 0, to = 10, by = 0.05)
> plot(y = pendulum1(interval1), x = interval1, type = "l",
+   lty = 1, xlab = "t", ylab = "deflection")
> lines(y = pendulum2(interval1), x = interval1, lty = 2)
```

Creating Data We create a data frame of 512 samples of `pendulum1` in the time interval $[1, 10]$. To simulate real measurement data, we add normally distributed noise with mean 0 and standard deviation 0.01 to the simulated values.

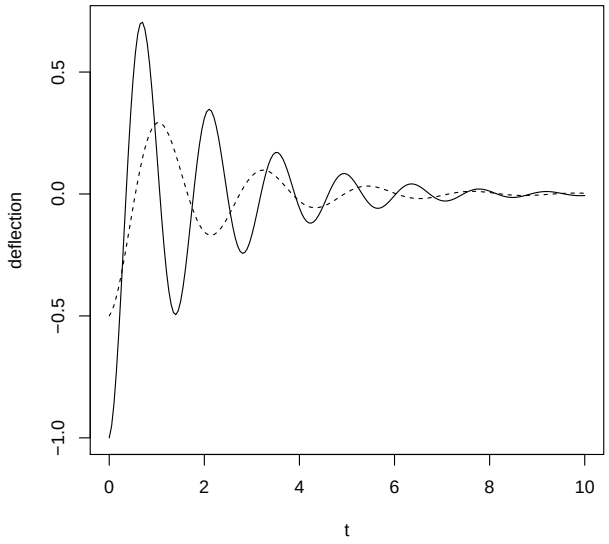

Figure 4: Deflection against time of two example pendulums: `pendulum1` is shown as a solid line, `pendulum2` is shown as a dashed line.

```
>   xs1 <- seq(from = 1, to = 10, length.out = 512)
>   pendulum1Data <- data.frame(time = xs1,
+     deflection = pendulum1(xs1) + rnorm(length(xs1), sd = 0.01))
```

Symbolic Regression Run We are now ready to start a symbolic regression run. Make sure to load the RGP package via `library("rgp")`. We choose a time budget of 2 minutes:

```
> modelSet1 <- symbolicRegression(deflection ~ time, data = pendulum1Data,
+                                 stopCondition = makeTimeStopCondition(2 * 60))
```

Result Analysis Selection and plotting of the model with best fitness can be performed as follows:

```
> bestModel1 <- modelSet1$population[[which.min(modelSet1$fitnessValues)]]
> plot(y = bestModel1(xs1), x = xs1, type = "l",
+   lty = 1, xlab = "x", ylab = "y")
> lines(y = pendulum1(xs1), x = xs1, lty = 2)
```

A slightly improved version of the output produced by these commands is shown in figure 1.

Next Steps The `symbolicRegression` commands offers many configuration options to explore. See the online help available by typing `?symbolicRegression` on an R command line for details or visit the RGP website at `rsymbolic.org`.

Perhaps most importantly, `symbolicRegression` supports multivariate regression simply via R's formula interface. To perform symbolic regression in two variables x_1 and x_2 with output variable y, the formula $y\ x1 + x2$ can be used as the first argument to `symbolicRegression`.

4.4 Strongly Typed Genetic Programming

Theme of this tutorial is the evolution of boolean functions via strongly typed genetic programming. Although slightly more complex than the previous tutorials, it will prepare you apply RGP to a much broader set of tasks.

Task Definition An example, we will use typed genetic programming to discover symbolic representations of the 3-parity function.[4] For reasons of flexibility, we start with an R-implementation of the general parity function:

[4] The 3-parity function is the parity function for 3 bits, i.e. a with 3 input parameters.

```
> parity <- function(x) {
+   numberOfOnes <- sum(sapply(x, function(bit) if (bit) 1 else 0))
+   numberOfOnes %% 2 != 0
+ }
```

For a boolean input vector x, the parity function returns true if the number of ones in x is odd. We specialize this general function to three parameters with the following wrapper function:

```
> parity3 <- function(x1, x2, x3) parity(c(x1, x2, x3))
```

Next, we use the RGP tool function `makeBooleanFitnessFunction` to convert `parity3` to a fitness function. The resulting `parityFitnessFunction` returns the number of different places in the value table of a boolean function presented as a parameter and the value table of the `parity3` function:

```
> parityFitnessFunction <- makeBooleanFitnessFunction(parity3)
```

This fitness function represents a distance metric: The Hamming distance of a 3-parameter boolean function given as the fitness function parameter from the `parity3` function, i.e. a norm. In other words, the `parityFitnessFunction` returns the number of input vectors for which a given boolean functions differs in output from the `parity3` function. As there are $2^3 = 8$ different possible boolean input vectors of length 3, the worst fitness is 8, and the best fitness is 0. Note the ambiguity of the term "worst fitness" in this case, as simply negating the output of a function of worst fitness yields a function with a perfect fitness of 0. GP search spaces of rich and interesting structure are very common, and it is often beneficial to customize GP search heuristic to exploit existing knowledge on search space structure to speed up search considerably.

As in the previous tutorials, we have to load the RGP package by issuing the `library("rgp")` command. Next, we define the set of symbolic expressions to be searched by RGP by providing GP building blocks for boolean functions. The constant factory set contains a single constant factory that creates boolean constants by fair coin-tosses:

```
> booleanConstantFactory <- function() runif(1) > .5
> booleanConstantSet <- constantFactorySet(
+   "booleanConstantFactory" %::% (list() %->% st("logical")))
```

The function set contains the boolean functions *and* (`&`), *or* (`|`) and *not* (`!`):

```
> booleanFunctionSet <- functionSet(
+   "&" %::% (list(st("logical"), st("logical")) %->% st("logical")),
+   "|" %::% (list(st("logical"), st("logical")) %->% st("logical")),
+   "!" %::% (list(st("logical")) %->% st("logical")))
```

The input variable set contains the three function parameters `x1`, `x2`, and `x3`:

```
> booleanInputVariableSet <- inputVariableSet(
+   "x1" %::% st("logical"),
+   "x2" %::% st("logical"),
+   "x3" %::% st("logical"))
```

The building block definitions above use a special RGP syntax for type annotations. The *`expression %::% type`* operator associates an R expression with an RGP type. An RGP type is either a base type of the form `st(`*`type name`*`)` or a function type of the form `list(`*`parameter type 1, parameter type 2, ...`*`) %->% `*`result type`*. This is a recursive definition, meaning that RGP types can express types for higher order functions, making them quite flexible. The theoretical basis of RGP's type system is the simply typed lambda calculus [Barendregt et al., 1992]. A noteworthy limitation of this system is the lack of the generic types available in programming languages such as C++ or Java.

With these definitions, we are able to explain the semantics of the types associated with the GP building blocks above:

- `list() %->% st("logical")` is the type of a function with no arguments that returns a boolean value.[5] This is the type of the single constant factory in the `booleanConstantSet` defined above.
- `list(st("logical"), st("logical")) %->% st("logical")` is the type of a function taking two boolean arguments and returning a boolean value. This is the type of each function in the `booleanFunctionSet` defined above.
- Trivially, `st("logical")` is the type of boolean values. This is the type of each input variable in the `booleanInputVariableSet` defined above.

[5] a `logical` in R's terminology

Strongly Typed Genetic Programming Run With the fitness function and search space defined, we are now ready to start a strongly typed GP run:

```
> typedGpResult1 <- typedGeneticProgramming(parityFitnessFunction, st("logical"),
+   functionSet = booleanFunctionSet,
+   inputVariables = booleanInputVariableSet,
+   constantSet = booleanConstantSet,
+   stopCondition = makeTimeStopCondition(30))
```

Note that `typedGeneticProgramming` expects the result type of the solution functions to generate as a second parameter, as this is not explicit from the building block definition. As we generate boolean functions, we state `st("logical")` here.

After running for 30 seconds, the result of the GP run is assigned to the variable `typedGpResult1`. As in the previous tutorials, the runtime (in seconds) can be adjusted by changing the parameter to `makeTimeStopCondition`.

Result Analysis Selection of the boolean function with best fitness is performed much like in the previous tutorials:

```
> bestFunction1 <- typedGpResult1$population[[which.min(typedGpResult1$fitnessValues)]]
```

See RGP's online documentation for details on visualizing and analyzing typed GP results. For example, the Rrules package includes as a RGP dependency can be employed for rule-based simplification of boolean functions.

4.5 Sequential Parameter Optimization for Genetic Programming

Finding good algorithm parameter settings for concrete Genetic Programming applications is a complex task. Sequential parameter optimization (SPO) provides a framework for applying modern statistical methods to solve this task. This tutorial shows how to apply the sequential parameter optimization toolbox (SPOT) to a very simple RGP setup.

SPOT Installation As RGP, SPOT is available as an R package on the comprehensive R archive network CRAN. To install SPOT and all it's dependencies, issue the following command in a running R session:

```
> install.packages("SPOT")
```

SPO Definition The SPOT package includes a very simple example for tuning RGP algorithm parameters. In this example, RGP is configured to use single-objective GP with tournament selection to solve the two-dimensional symbolic regression problem given in the following R script:

```
> x1 <- seq(0, 4 * pi, length.out = 201)
> x2 <- seq(0, 4 * pi, length.out = 201)
> y <- sin(x1) + cos(2 * x2)
> data1 <- data.frame(y = y, x1 = x1, x2 = x2)
> result1 <- symbolicRegression(y ~ x1 + x2,
+   data = data1,
+   populationSize = populationSize,
+   selectionFunction = makeTournamentSelection(tournamentSize = tournamentSize),
+   functionSet = arithmeticFunctionSet,
+   stopCondition = makeTimeStopCondition(time))
> bestFitness <- min(sapply(result1$population, result$fitnessFunction))
```

This R script is part of the SPOT package and does not need to entered by hand. It is reproduced here for illustrative purposes.

SPO is applied to find parameter settings for `populationSize` and `tournamentSize` that optimize (i.e. minimize) `bestFitness`. The region of interest (ROI) for this optimization given in the file `rgp0001.roi` and shown in Table 1. Parameters belonging to the problem design, in this case just the parameter `time`, giving the maximum runtime in seconds for a symbolic regression run, are given in the file `rgp0001.apd`. Parameters for SPOT are given in the file `demo17Rgp.conf`. All these files can be found in the directory indicated by the R expression `file.path(find.package("SPOT"), "demo17Rgp")`.

Parameter	Type	ROI Interval
`populationSize`	Integer	[20, 1000]
`tournamentSize`	Integer	[20, 1000]

Table 1: Region of interest used in the SPOT example for tuning RGP algorithm parameters.

SPO Run To start the SPO run, issue the following commands:

```
> library("SPOT")
> confPath <- find.package("SPOT")
> confPath <- file.path(confPath, "demo17Rgp")
> confFile <- file.path(confPath, "rgp0001.conf")
> spotConfig <- spot(confFile)
```

SPO Result Analysis After the SPO run, results are available in a list data structure stored in the `spotConfig` variable. See the SPOT online documentation for details.

5 Outlook

This concludes the short introduction to RGP. There are many more possibilities and use case scenarios not touched upon here. Also, RGP is a evolving system, so new functionality might be present to improve the performance or ease of use at tasks described in the tutorials. The Rsymbolic website[6] provides the most current information on the current state of RGP, as well as additional tutorials and documentation. This website also contains a roadmap of planned features, access to development versions, as well as instructions on how to contribute to the project.

[6] see `rsymbolic.org`

As already mentioned, RGP also offers detailed online documentation of all its functionality. Type `help(package = "rgp")` in a running R session to get an overview of this documentation.

References

Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* Morgan

Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-510-X.

Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

Thomas Bartz-Beielstein, Marco Chiarandini, Luis Paquete, and Mike Preuss, editors. *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, Berlin, Heidelberg, New York, 2010.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2009. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Yi Sun, Daan Wierstra, Tom Schaul, and Juergen Schmidhuber. Efficient natural evolution strategies. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 539–546, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9.

Imprint

Oliver Flasch
SPOTSeven group / Rsymbolic project
Cologne University of Applied Sciences
Steinmüllerallee 1
51643 Gummersbach
Germany
Web: `rsymbolic.org`
Email: `oliver.flasch@fh-koeln.de`