Visualizing Logic Formula Evaluation in RISCAL*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

William Steingartner
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia
William.Steingartner@tuke.sk

July 16, 2018

Abstract

We report on initial results concerning the visualization of the evaluation of logic formulas that are formulated in the RISC Algorithm Language (RISCAL). Such formulas usually represent propositions that are supposed to be true; examples are mathematical theorems or verification conditions of algorithms that have been formally modeled and specified in RISCAL. The visualization is intended to aid the user to understand the truth value of a formula, in particular in those cases where a formula is unexpectedly not valid. To this aim, the visualization of a formula consists of a pruned evaluation tree that depicts exactly those evaluation branches that contribute to the overall truth value.

1 Introduction

The RISC Algorithm Language (RISCAL) [1, 3] is is a specification language and associated software system for modeling mathematical algorithms, formally specifying their behavior based on mathematical theories, and validating the correctness of algorithms, specifications, and theories by execution/evaluation. RISCAL pursues the strategy to "fully automatically" check programs and annotations. For this purpose, it restricts the domains of all types by formal parameters to finite sizes which makes it possible on the one hand to check all possible executions of a program and on the other hand to decide all formulas by evaluation [4, 5].

In previous work [6], we have elaborated the visualization of procedure executions as a hierarchical graph where each graph represents the state changes in the execution of a procedure; the user may switch by mouse-clicks from a graph on one level to a graph on the next level and back. This work was inspired by a toolset for the toy programming language Jane [7, 9, 10, 11] that illustrates the categorical semantics of Jane [8] as a directed graph whose nodes (the objects of the category) represent the states of a procedure execution and whose arrows (the morphisms of the category) represent the transitions among the procedure states. Starting

*Supported by the Austrian OEAD WTZ program and the Slovak SRDA agency under the contract SK 14/2018 "SemTech — Semantic Technologies for Computer Science Education" and by the Johannes Kepler University Linz, Linz Institute of Technology (LIT), Project LOGTECHEDU "Logic Technology for Computer Science Education".

with the node representing the initial procedure state and following the arrows representing the state transitions, the program execution can be understood.

While that previous work aimed to aid the understanding of the result of a procedure by a visual representation of its execution, the work presented in this paper aims to aid the understanding the truth value of a formula by a visual representation of its evaluation. This visualization consists of a hierarchical tree where each node represents the truth value of a (sub)formula. If a node represents a user-defined atomic formula, i.e., the application of a predicate defined by another formula, the user may switch by a mouse-click to the visualization of that formula and back by another mouse-click. Furthermore, the evaluation tree is pruned in such a way that only those branches are presented that contribute to the overall result. For instance, in the case of a universally quantified formula $(\forall x.\ F)$ whose truth value is "false", just one branch visualizing the evaluation of a single formula instance $F[x \mapsto a]$ is depicted where a is some value for variable x that makes F false; only if the truth value of $(\forall x.\ F)$ is "true", all branches visualizing the evaluation of all instances $F[x \mapsto a_1], \ldots, F[x \mapsto a_n]$ are depicted that demonstrate that all possible values $a_1, \ldots, a_n$ for x make the formula F true.

The rest of this paper is organized as follows: First, Section 2 discusses the motivation of our work by illustrating that the visualization of formulas by the visualization of procedures computing their truth values is not sufficient. Then Section 3 demonstrates the features of our new formula visualization and its practical use, while Section 4 discusses the implementation of the visualization and the underlying technology. Section 5 concludes and presents an outlook on future work.

2 Motivation

To begin with, one may question the motivation for a special tool for the visualization of formula evaluation: it is also possible to translate a formula into a program that computes the truth value of the formula and then to visualize the execution of that program. Indeed Figure 1 describes such a translation of a formula F (in RISCAL syntax) into a command $[\![\ F\ ]\!]$ (also in RISCAL syntax) that sets the variable `result` to the truth value of the formula.

For instance, given the formula

```
∀x:T. p(x) ⇒ ∃y:T. q(x,y)
```

the resulting command is:

```
result := true;
var xs:Set[T] = { x | x:T };
while result = true ∧ xs ≠ ∅[T] do
{
  choose x ∈ xs; xs := xs\{x};
  result := p(x);
  if result = false then
    result := true;
  else
  {
    result := false;
    var ys:Set[T] = { x | x:T };
    while result = false ∧ ys ≠ ∅[T] do
    {
      choose y ∈ ys; ys := ys\{y};
      result := q(x,y);
    }
  }
}
```

Then using the definitions

```
⟦ p(t1,...,tn) ⟧ := result := p(t1,...,tn);

⟦ ¬F ⟧ := ⟦ F ⟧ ; if result = true then result := false; else result := true;
⟦ F1 ∧ F2 ⟧ := ⟦ F1 ⟧ ; if result = true then ⟦ F2 ⟧
⟦ F1 ∨ F2 ⟧ := ⟦ F1 ⟧ ; if result = false then ⟦ F2 ⟧
⟦ F1 ⇒ F2 ⟧ := ⟦ F1 ⟧ ; if result = false then result := true; else ⟦ F2 ⟧
⟦ F1 ⇔ F2 ⟧ := ⟦ F1 ⟧ ; if result = true
                          then ⟦ F2 ⟧
                          else { ⟦ F2 ⟧
                                 if result = true then result := false;
                                                  else result := true; }

⟦ ∀x:T. F ⟧ :=
{
  result := true;
  var xs:Set[T] = { x | x:T };
  while result = true ∧ xs ≠ ∅[T] do
  {
    choose x ∈ xs; xs := xs\{x};
    ⟦ F ⟧
  }
}
⟦ ∃x:T. F ⟧ :=
{
  result := false;
  var xs:Set[T] = { x | x:T };
  while result = false ∧ xs ≠ ∅[T] do
  {
    choose x ∈ xs; xs := xs\{x};
    ⟦ F ⟧
  }
}
```

Figure 1: The Translation of Formulas to Commands

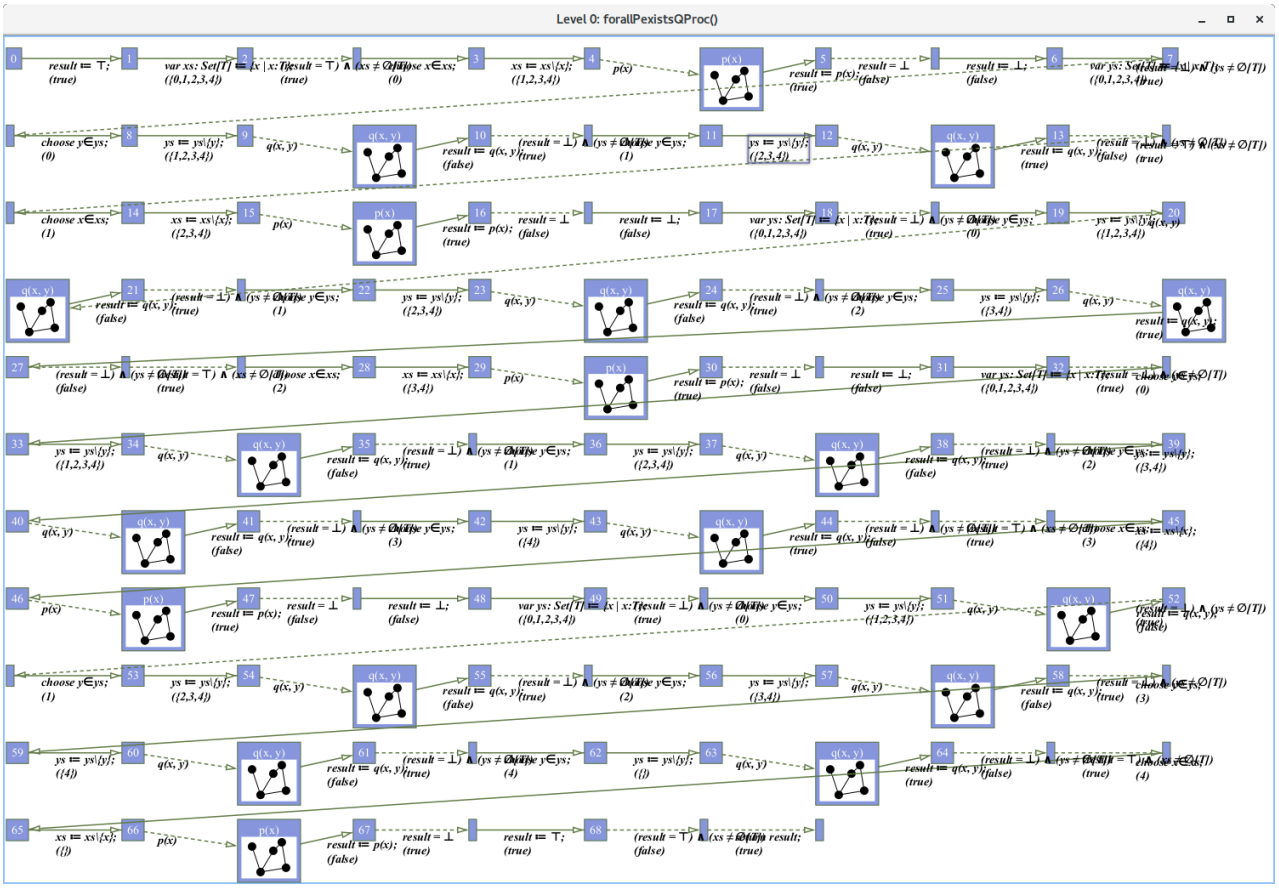

Figure 2: The Visualization of a Program Computing $(\forall x.\ p(x) \Rightarrow \exists y.\ q(x, y))$

```
val N = 4;
type T = ℕ[N];

pred p(x:T) ⇔ x < N;
pred q(x:T,y:T) ⇔ x+1 = y;

proc forallPexistsQProc():Bool
{
  var result:Bool;
  ... // insert command here
  return result;
}
```

we may indeed invoke the visualization mechanism of RISCAL to display the execution of procedure `forallPexistsQProc()`.

However, the result depicted in Figure 2 immediately illustrates the problem with this approach. This visualization, consisting of a sequence of tests and assignments, is much too low-level to convey any true understanding of the formula; it does not give any no high-level explanation *why* the evaluation of the formula results in the denoted truth value (in this example "true").

In the following, we will demonstrate another approach that is able to convey such an understanding.

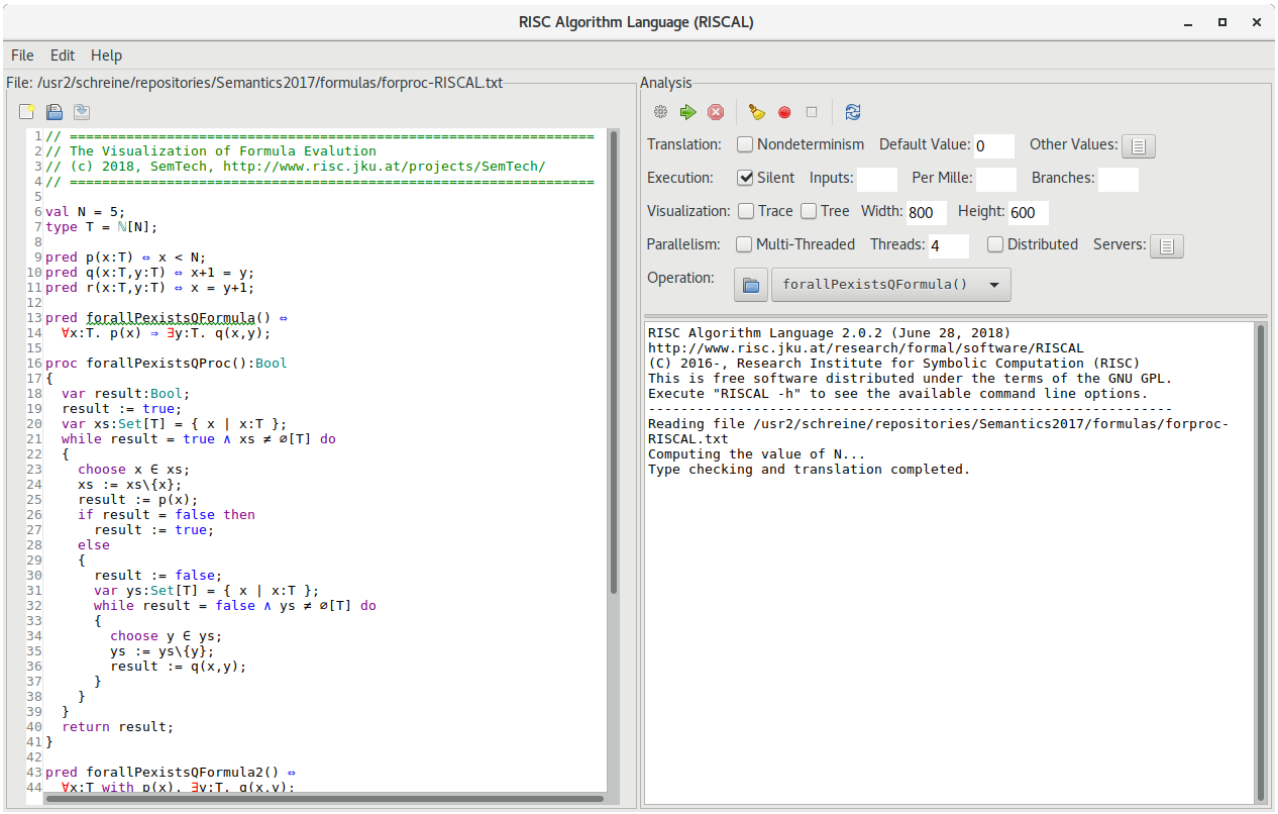

Figure 3: The RISCAL User Interface

3 Use of the Visualization

If RISCAL is started and visualization is enabled (see Section 4), the graphical user interface depicts a row "Visualization" (see Figure 3) with two (mutually exclusive) check box options "Trace" and "Tree".

If any of these options is selected, every run/check of a RISCAL operation (procedure, function, predicate, theorem) opens a window in which a visualization of this operation is displayed *provided that*

- the option "Nondeterminism" is *not* selected,
- the options "Multi-Threaded" and "Distributed" are *not* selected.

In other words, visualization is restricted to the deterministic execution of RISCAL operations in the sequential mode of the RISCAL software. The user may configure the size of the visualization area by the fields "Width" and "Height" (independent of this setting, the minimum size of a visualization area is 100 times 100 pixels).

The option "Trace" triggers the trace-based visualization of procedures discussed already in [6] (using an older version of the user interface). The remainder of this paper is dedicated to the option "Tree" that triggers a tree-based visualization of formula evaluation.

Visualization of Formula Evaluation In the following, we visualize the evaluation of the formula described in the previous section by the following specification:

```
val N = 4;
type T = ℕ[N];
pred p(x:T) ⇔ x < N;
pred q(x:T,y:T) ⇔ x+1 = y;
pred forallPexistsQFormula() ⇔ ∀x:T. p(x) ⇒ ∃y:T. q(x,y);
```

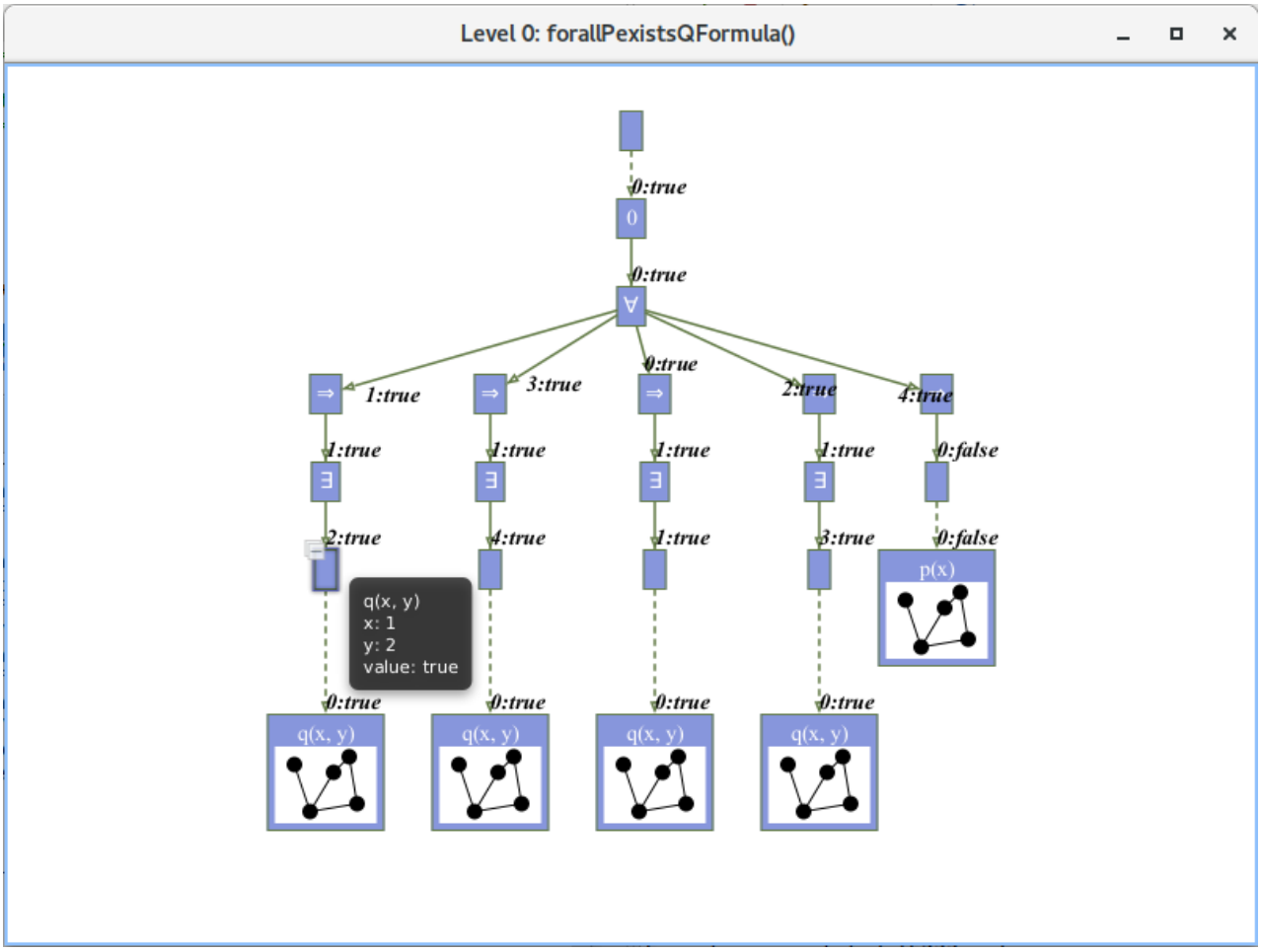

Figure 4: The Visualization of the Evaluation of $(\forall x.\ p(x) \Rightarrow \exists y.\ q(x, y))$

We select the operation `forallPexistsQFormula()`, the visualization option "Tree", and press the "Start Execution" button labeled with the green arrow. After the execution of the operation has finished, the window depicted in Figure 4 pops up.

This header of the window displays the predicate `forallPexistsQFormula()` whose evaluation is visualized; the tag "Level 0" indicates that this is the top level of the evaluation (every entry into the nested visualization of a predicate application increases this level by one). The main panel of the window depicts a tree whose nodes are laid out layer by layer from top to bottom with directed edges (arrows) connecting the nodes from one layer to the next one.

Nodes A tree may have five kinds of nodes:

- *The root node* represents the evaluation of the predicate for all possible arguments. By hovering with the mouse pointer over this node, a small window pops up that displays the definition of the predicate and whether its execution for all possible arguments resulted in an error or not. If there are n possible arguments and no error has occurred, the root has n children each of which represents an evaluation with one of the arguments. However, if an error occurs, the root one has one child representing the evaluation with the argument that triggered the error.

  In particular, if the operation is a "theorem" (keyword `theorem`) rather than a plain "predicate" (keyword `pred`), the evaluation triggers an error if there is any argument for which the formula defining the theorem

has truth value "false". In that case only the child representing this evaluation is displayed.

- *Numbered nodes* are children of the root node each of which represents the evaluation of the predicate for one particular argument. Nodes are numbered from 0 on, i.e., if there are n nodes, they have numbers $0, \ldots, n-1$ (the nodes are not necessarily laid out in the order of the numbers). By hovering with the mouse pointer over such a node, a small window pops up that displays the definition of the predicate, the value(s) of its argument(s), and the resulting truth value.

- *Labeled nodes* represent propositional formulas or quantified formulas where the label denotes the formula's outermost logical symbol (logical connective or quantifier). By hovering with the mouse pointer over such a node, a small window pops up that displays the formula, the values of its free variables (actually the values of all variables visible at the occurrence of the formula), and the truth value of the formula.

- *Empty nodes* represent formulas whose outermost symbol is not a logical connective or quantifier, in particular (but not exclusively) atomic formulas. By hovering with the mouse pointer over such a node, the same information as for labeled nodes is displayed.

- *Call nodes* display a small window with a graph symbol; such a node represents the application of a user-defined predicate. The header of this window displays the application itself. By hovering with the mouse pointer over such a node, the same information as for a labeled node is displayed. By double-clicking on such a node, the visualization switches from the current formula to the formula defining the predicate (see the paragraph "Nested Visualization" below).

  Call nodes are also generated for the applications of functions and procedures; however, the evaluation trees depicted for these entities are typically not very meaningful.

Nodes may be selected by a mouse click and moved to another location in the display.

Arrows Nodes are connected by two types of arrows:

- *Solid arrows* connect a formula to all those subformulas (respectively instances of subformulas) whose truth values are relevant for the truth value of the whole formula.

- *Dashed arrows* are used at the top of the tree to connect the root node to the numbered nodes (representing the individual invocations of the predicate) and at the bottom of the tree to connect empty nodes (representing atomic formulas) to call nodes (representing the invocations of the predicates and functions within the atomic formula).

The targets of all arrows are annotated with labels of form $n{:}v$ where n is the number of the subexpression and v is its value. Numbers start with 0 (the arrows are not necessarily laid out in the order of the numbers): a formula with a unary logical connective has one arrow with number 0, a formula with a binary connective has at most two arrows with numbers 0 (the first subformula) and 1 (the second subformula); a quantified formula whose variable can be assigned n values has at most n arrows with numbers $0, \ldots, n-1$ denoting the individual instances of the formula's body. Furthermore, if an atomic formula contains n procedure/function applications, the corresponding empty node has n arrows to the call nodes corresponding to these applications.

The labels associated to the arrows (actually to the target nodes of the arrows) may be selected by a mouse click and moved to another location in the display.

Nested Visualization By double-clicking on a call node which represents the application of an operation (in particular a predicate), the content of the window is modified to visualize the execution of that operation (the depicted evaluation tree is essentially only helpful for predicates since only these are defined by formulas). For instance, the left diagram in Figure 5 displays the evaluation of the predicate application $p(x)$ in branch 4

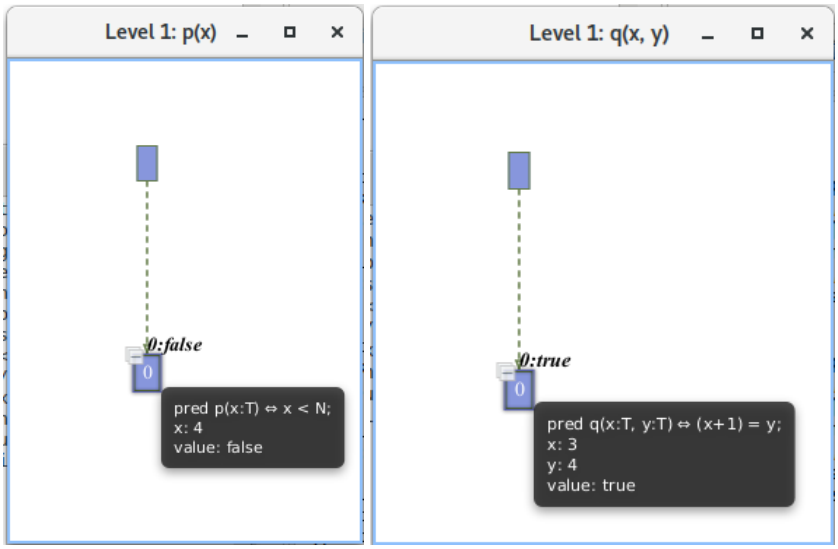

Figure 5: The Visualization of Predicates

of Figure 4, while the right diagram displays the evaluation of $q(x, y)$ in branch 3. The tag "Level 1" in the titles of the windows indicates that the visualization refers to the application of an operation that was invoked on "Level 0". If the "Level 1" operation would involve another operation application, it would contain another call node; by double-clicking on that node, we would move to "Level 2" and so on. A double click on any empty part of the window moves the display back to the previous level.

Tree Pruning Every evaluation tree is pruned such that it only displays the information necessary to understand how its truth value was derived:

- If the truth value of a conjunction $(F_1 \wedge F_2)$ is "false", only the first subformula is displayed whose truth value is "false".
- If the truth value of a disjunction $(F_1 \vee F_2)$ is "true", only the first subformula is displayed whose truth value is "true".
- If the truth value of an implication $(F_1 \Rightarrow F_2)$ is "true", only one subformula is displayed (either the "false" antecedent F_1, or, if this antecedent is "true", then the "true" consequent F_2).
- If the truth value of a universally quantified formula $(\forall x.\ F)$ is "false", only one instance $F[x \mapsto a]$ is displayed, where a is the first value encountered in the evaluation that makes F "false".
- If the truth value of an existentially quantified formula $(\exists x.\ F)$ is "true", only one instance $F[x \mapsto a]$ is displayed, where a is the first value encountered in the evaluation that makes F "true".

For example, in the tree depicted in Figure 4, all "true" implications $(p(x) \Rightarrow \exists y.\ q(x, y))$ are pruned: in branch 0, only the evaluation tree for the "false" atomic formula $p(x)$ is displayed; in all other branches, the evaluation tree for the "true" existential formula $(\exists y.\ q(x, y))$ is displayed. In the evaluation tree of every such existential formula only one branch is displayed corresponding to one "true" instance of atomic formula $q(x, y)$. By hovering the mouse pointer over the nodes, the corresponding variable values are displayed.

We further illustrate the role of tree pruning by the evaluation of the two "theorems" `forallPQR()` and `forallPQR2()` depicted below:

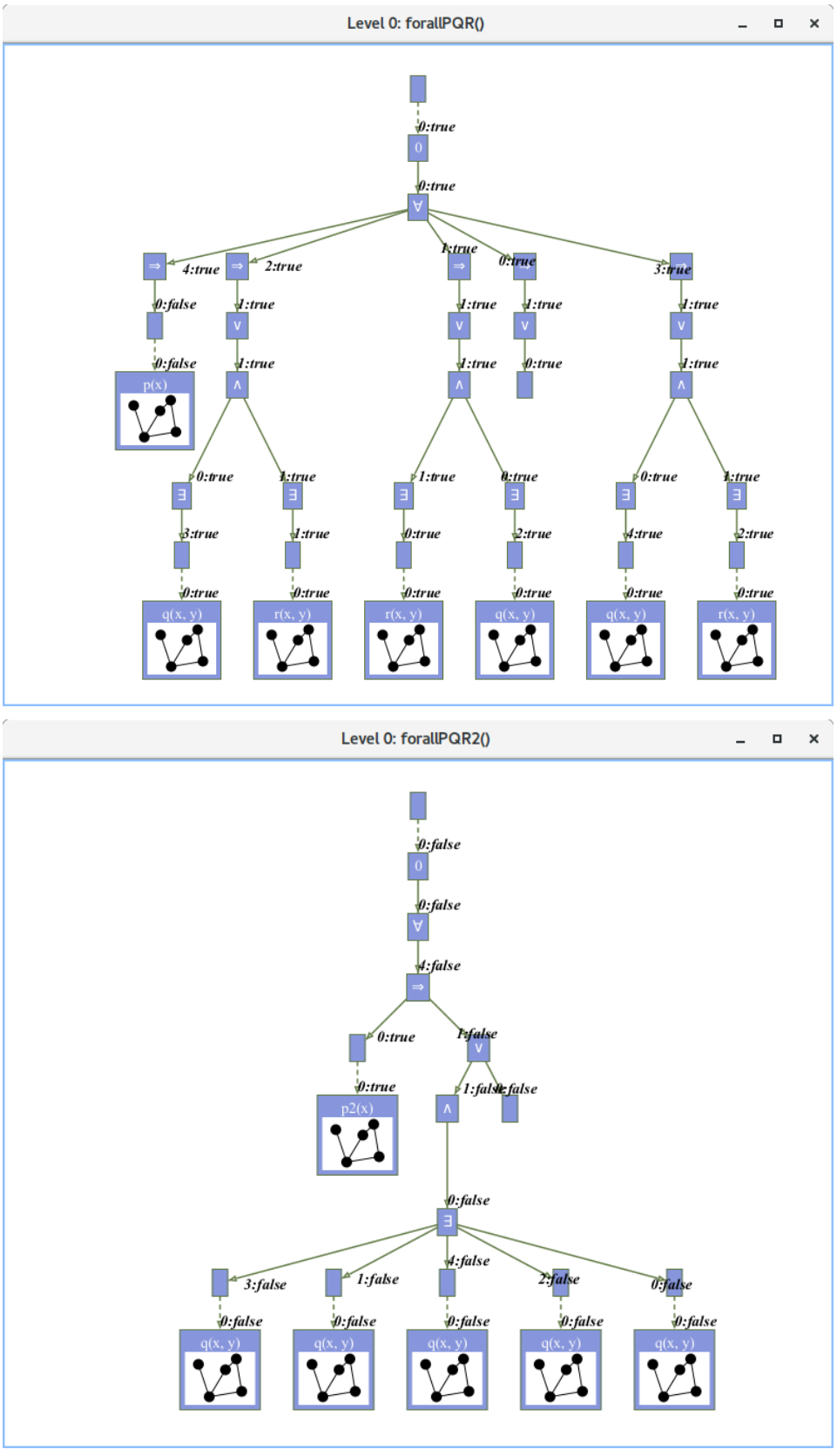

Figure 6: Pruned Evaluation Trees

```
val N = 4;
type T = ℕ[N];
pred p(x:T) ⇔ x < N;
pred p2(x:T) ⇔ x ≤ x;
pred q(x:T,y:T) ⇔ x+1 = y;
pred r(x:T,y:T) ⇔ x = y+1;
theorem forallPQR() ⇔ ∀x:T. p(x) ⇒ x = 0 ∨ (∃y:T. q(x,y)) ∧ (∃y:T. r(x,y));
theorem forallPQR2() ⇔ ∀x:T. p2(x) ⇒ x = 0 ∨ (∃y:T. q(x,y)) ∧ (∃y:T. r(x,y));
```

The first theorem is indeed valid as illustrated by the top diagram in Figure 6. Since the universally quantified formula is true, all branches of this formula are depicted. In each branch, the implication is true which leads to a continuation with a single branch. In this branch, the disjunction is true, which again leads to a continuation with a single branch. In this branch, the conjunction is true, which now leads to a split into two branches; each of these branches contains a true existential formula, for which it again suffices to depict a single branch.

On the contrary, the second theorem is invalid as illustrated by the bottom diagram in Figure 6. Since the universally quantified formula is false, only one branch of this formula is depicted. The implication in this branch is false which leads to a split into two branches, the first of which is immediately true. In the second branch, we have a false disjunction, which also leads to a split into two branches, of which the first one is immediately false. In the second branch, we have a false conjunction, of which only the first branch with a false existential formula needs to be shown. To demonstrate the invalidity of this existential formula, all its branches have to be depicted.

Terminating the Visualization By closing the window (via a click on the top right button of the window frame) the visualization is terminated.

4 Implementation of the Visualization

The visualization has been implemented with the help of the Eclipse GEF/Zest framework for graph visualization which in turn is based on JavaFX. Due to dependencies on these software libraries, the visualization features of RISCAL are only enabled, if the command line option `-trace` is provided (this is hidden from the user: the local installation of the `RISCAL` script does or does not set this option). Further details on the software dependencies are already given in [4] where the execution trace visualization was discussed.

However, the evaluation tree visualization is more demanding with respect to graph layouting. For this purpose, RISCAL deploys on Level 0 the implementation of the "space tree layout" algorithm provided by GEF/Zest, which gives a regular and compact layout. However, this implementation does not work for the trees on the higher levels (the reason is not clear; we suspect a bug or restriction of the implementation) ; for these the GEF/Zest implementation of the "Sugiyama layout" algorithm is employed, which however results in a less appealing layout.

Furthermore, the implementation of these layout algorithms provided by GEF/Zest has two drawbacks:

- The implementation becomes very slow for larger trees. This is deplorable all the more, as the layout is computed by that thread that also handles the graphical user interface; thus the RISCAL interface is blocked during this computation. In order to mitigate this problem, we attempt to limit the blocking time by visualizing only trees up to a certain maximum number of nodes (currently 500); for larger trees, RISCAL refuses any visualization attempt. However, even with this limit, the layout time may be still substantial; thus it is recommended to attempt the visualization first for very small model sizes before proceeding to larger ones.

- If the visualization area is too small, the layout algorithm may only compute a partial layout or no layout at all; this results in the placement of some/all nodes on the left upper corner of the visualization window. To overcome this problem, the user may choose the size of this visualization area by setting the values

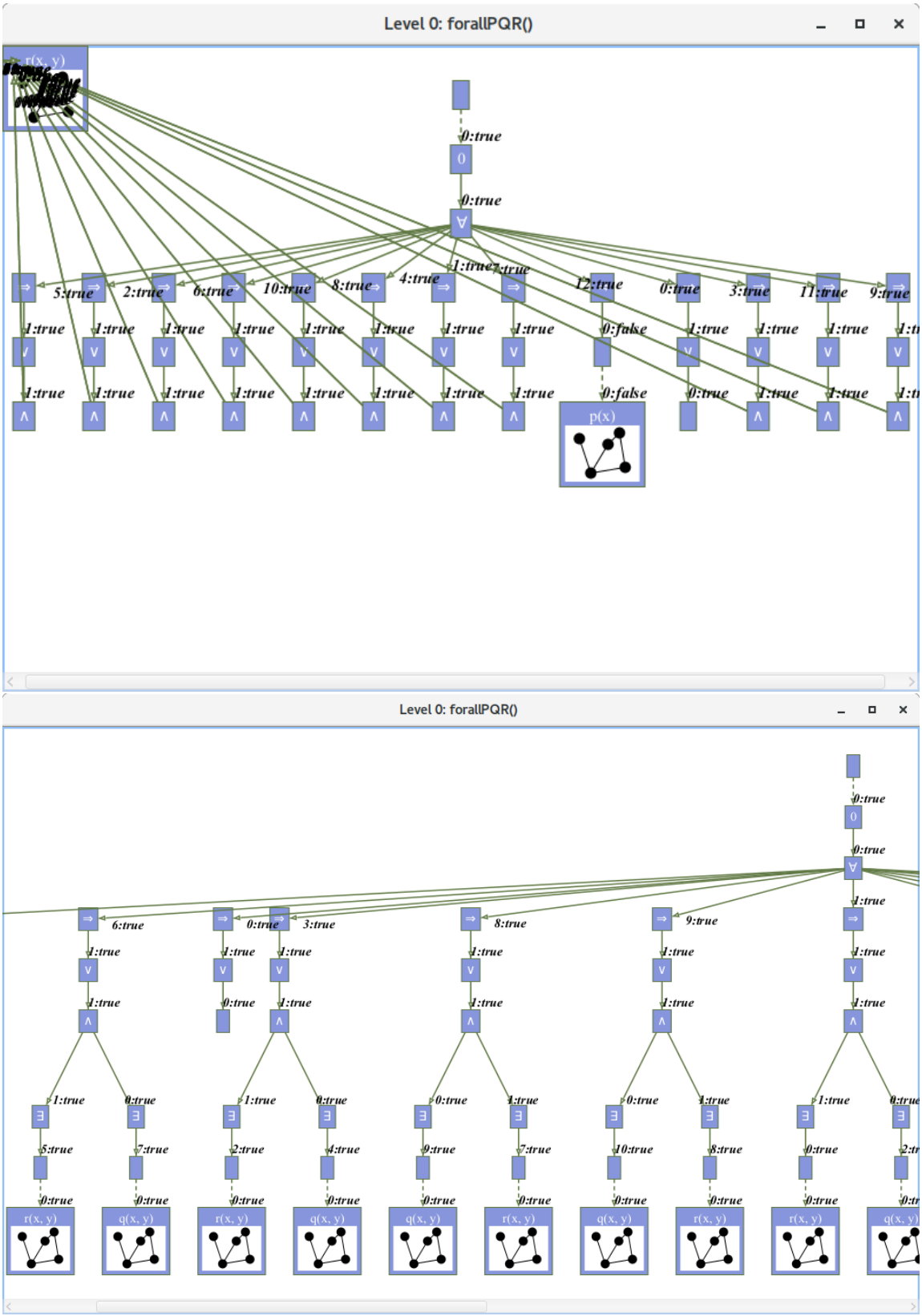

Figure 7: The Visualization of a Large Tree

"Width" and "Height" in the RISCAL user interface; if the resulting visualization is unsatisfactory, larger dimensions may be chosen. These dimensions may also wastly exceed the dimensions of the physical screen; e.g., we may choose a layout area of 16000 times 4000 pixel, even if our physical screen size is just 1920 times 1080. In this case, the window will be maximized to the physical screen size with scroll bars allowing to navigate within the layout area.

To illustrate the second problem and its solution, we show the visualization of formula `forallPQR()` introduced in the previous section for $N = 12$:

```
val N = 12;
...
theorem forallPQR() ⇔
  ∀x:T. p(x) ⇒ x = 0 ∨ (∃y:T. q(x,y)) ∧ (∃y:T. r(x,y));
```

The top diagram in Figure 7 displays the visualization for the default dimension 800 times 600 pixels; here the algorithm fails to layout many of the nodes, in particular most of the call nodes, which are placed all into the top left corner. Even the maximization of the window to the physical screen size 1920 times 1080 does not result in a fully satisfactory layout. However, if we set the layout area to 4000 times 1000 pixels we get the visualization depicted in the lower diagram of Figure 7. Clearly the tree exceeds horizontally the dimension of the window (in our illustration reduced from the original maximal screen size to about 800 times 600 pixels again), but we may use the horizontal scroll bar to investigate the currently not visible parts of the tree.

5 Conclusions

We believe that our visualization of the evaluation of logic formulas by pruned evaluation trees will considerably contribute to an understanding of complex formulas, in particular of formulas with nested quantifiers; this understanding may be thus gained in a much quicker way than by the usual inspection with "paper and pencil". In particular, this visualization will help users to understand why propositions that are supposed to be true are actually not; this is of particular importance in computer-based mathematics (mathematical theorems) and in the proof-based verification of computer programs (verification conditions) [4, 5].

In the future we plan to investigate the educational usefulness of such visualizations by the application of RISCAL in courses for students of computer science and mathematics [2]. We will also work (in analogy to the categorical semantics of execution traces presented in [8]) on a corresponding categorical semantics of formula evaluation, which may lead to further insights.

References

[1] RISCAL. *The RISC Algorithm Language (RISCAL)*. Mar. 2017. URL: https://www.risc.jku.at/research/formal/software/RISCAL.

[2] Wolfgang Schreiner. "Logic as a Path to Enlightenment (Work in Progress Report)". In: *CME-EI18: Computer Mathematics in Education — Enlightenment or Incantation?* Ed. by Walther Neuper. Workshop at CICM 2018, 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 17: CEUR Workshop Proceedings, 2018. URL: https://www.risc.jku.at/people/schreine/papers/CME-EI-2018.pdf.

[3] Wolfgang Schreiner. *The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0)*. Technical Report. Download from [1]. Johannes Kepler University, Linz, Austria: RISC, Mar. 2017.

[4] Wolfgang Schreiner. "Validating Mathematical Theories and Algorithms with RISCAL". In: *CICM 2018 — 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 13-17*. Ed. by F. Rabe, W. Farmer, G. Passmore, and A. Youssef. Vol. 11006. Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence. Springer, Berlin, 2018. URL: https://www.risc.jku.at/people/schreine/papers/CICM-2018.pdf.

[5] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. "Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models". In: *Post-Proceedings ThEdu'17, 6th International Workshop on Theorem proving components for Educational software*. Ed. by Pedro Quaresma and Walther Neuper. Vol. 267. Electronic Proceedings in Theoretical Computer Science (EPTCS). Gothenburg, Sweden, August 6, 2017: Open Publishing Association, 2018, pp. 120–139. URL: https://doi.org/10.4204/EPTCS.267.8.

[6] Wolfgang Schreiner and William Steingartner. *Visualizing Execution Traces in RISCAL*. Technical Report. Johannes Kepler University, Linz, Austria: Research Institute for Symbolic Computation (RISC), Mar. 2018. URL: https://www.risc.jku.at/publications/download/risc_5610/main.pdf.

[7] William Steingartner, Mohamed Ali M. Eldojali, Davorka Radaković, and Jiří Dostál. "Software support for course in Semantics of programming languages". In: *IEEE 14th International Scientific Conference on Informatics*. Poprad, Slovakia, November 14–16, 2017, pp. 359–364. URL: https://www.researchgate.net/publication/321341571_Software_support_for_course_in_Semantics_of_programming_languages.

[8] William Steingartner and Valerie Novitzká. "Categorical Semantics of Programming Langages". In: *Selected Topics in Contemporary Mathematical Modeling*. Ed. by Andrzej Z. Grzybowski. Vol. 331. Monographs. Czestochowa University of Technology. Chap. 11, pp. 167–192. URL: https://doi.org/10.17512/STiCMM2017.11.

[9] William Steingartner and Valerie Novitzká. "Learning tools in course on semantics of programming languages". In: *MMFT 2017 — Mathematical Modelling in Physics and Engineering*. Czestochowa, Poland, September 18–21, 2017, pp. 137–142. URL: http://im.pcz.pl/konferencja/get.php?doc=MMFT2017_streszczenia_wykladow.pdf.

[10] William Steingartner and Valerie Novitzká. "New Learning Tools for Teaching the Semantics of Programming Languages". In: *MMPE 2018 — Mathematical Modelling in Physics and Engineering*. Poraj Resort, Poland, June 18–21, 2018, pp. 63–68. URL: http://im.pcz.pl/konferencja/dokumenty/MMPE2018_abstract_book.pdf.

[11] William Steingartner and Iskender Yar-Muhamedov. "Learning Software for Handling the Mathematical Expressions". In: *Journal of Applied Mathematics and Computational Mechanics* 17.2 (2018), pp. 77–91. URL: https://doi.org/10.17512/jamcm.2018.2.07.