A Visual Interactive Framework for Formal Derivation

P. Agron[1], L. Bachmair[1], and F. Nielsen[2]

[1] Department of Computer Science, Stony Brook University, Stony Brook, New York
[2] Sony Computer Sciences Laboratory Inc, Tokyo, Japan

Abstract. We describe a visual interactive framework that supports the computation of syntactic unifiers of expressions with variables. Unification is specified via built-in transformation rules. A user derives a solution to a unification problem by stepwise application of these rules. The software tool provides both a debugger-style presentation of a derivation and its history, and a graphical view of the expressions generated during the unification process. A backtracking facility allows the user to revert to an earlier step in a derivation and proceed with a different strategy.

1 Introduction

Kimberly is a software tool intended to be used in college-level courses on computational logic. It combines results from diverse fields: mathematical logic, computational geometry, graph drawing, computer graphics, and window systems. A key aspect of the project has been its emphasis on visualizing formal derivation processes, and in fact it provides a blueprint for the design of educational software tools for similar applications.

Applications of computational logic employ various methods for manipulating syntactic expressions (i.e., terms and formulas). For instance, *unification* is the problem of determining whether two expressions E and F can be made identical by substituting suitable expressions for variables. In other words, the problem requires one to syntactically *solve* equations $E \approx F$. For example, the two expressions $f(x, c)$ and $f(h(y), y)$, where f and h denote functions, c denotes a constant, and x and y denote variables, are *unifiable*: substitute c for y and $h(c)$ for x. But the equation $f(x, x) \approx f(h(y), y)$ is not solvable.[1]

Logic programming and automated theorem proving require algorithms that produce, for solvable equations $E \approx F$, a *unifier*, that is, a substitution σ such that $E\sigma = F\sigma$. Often one is interested in computing *most general unifiers*, from which any other unifier can be obtained by further instantiation of variables. Such unification algorithms can be described by collections of *rules* that are

[1] Note that we consider *syntactic* unification and do not take into account any semantic properties of the functions denoted by f and h.

designed to transform a given equation $E \approx F$ into *solved form*, that is, a set of equations, $x_1 \approx E_1, \ldots, x_n \approx E_n$, with variables x_i on the left-hand sides that are all different and do not occur in any right-hand-side expression E_j. The individual equations $x_i \approx E_i$ are called *variable bindings*; collectively, they define a unifier: substitute for x_i the corresponding expression E_i.

For example, the equation $f(x, c) \approx f(h(y), y)$ can be *decomposed* into two equations, $x \approx h(y)$ and $c \approx y$. Decomposition preserves the solutions of equational systems in that a substitution that *simultaneously* solves the two simplified equations also solves the original equation, and vice versa. We can *reorient* the second equation, to a variable binding $y \approx c$, and view it as a partial specification of a substitution, which may be applied to the first equation to yield $x \approx h(c)$. The two final equations, $x \approx h(c)$ and $y \approx c$, describe a unifier of the initial equation. The example indicates that in general one needs to consider the problem of simultaneously solving several equations, $E_1 \approx F_1, \ldots, E_n \approx F_n$.

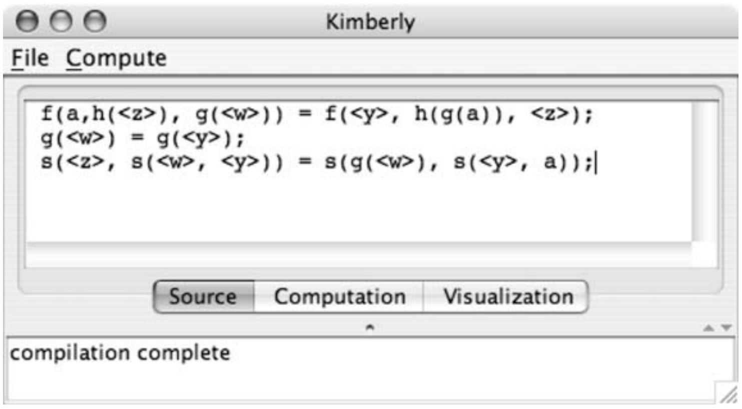
A *derivation* is a sequence of sets of equations as obtained by repeated applications of rules. If transformation rules are chosen judiciously, the construction of derivations is guaranteed to terminate with a final set of equations that is either in solved form (and describes a most general unifier) or else evidently unsolvable. Unsolvability may manifest itself in two ways: (i) as a *clash*, i.e. an equation of the form $f(E_1, \ldots, E_n) \approx g(F_1, \ldots, F_k)$, where f and g are different function symbols, or (ii) as an equation $x \approx E$, where E contains x (but is different from it), e.g., $x \approx h(x)$. The transformation rules we have implemented in *Kimberly* are similar to those described in [8].

In Section 2 we describe the visual framework for derivations. The current version of *Kimberly* provides graphical representation of terms and formulas as *trees*, but has been designed to be augmented with additional visualization functionality for representation of *directed graphs*, as described in Section 3. We discuss some implementation details in Section 4 and conclude with plans for future work.

2 Visual Framework for Derivations

Kimberly features a graphical user interface with an integrated text editor for specifying sets of equations. The input consists of a set of (one or more) equations, to be solved simultaneously. Equations can be entered via a "source" panel, see Figure 1. They can also be saved to and loaded from a text file. For example, the upper section of the window in Figure 1 contains the textual representation of three equations to be solved, $f(a, h(z), g(w)) \approx f(y, h(g(a)), z)$, $g(w) \approx g(y)$, and $s(z, s(w, y)) \approx s(g(w), s(y, a))$. Note that in *Kimberly* we distinguish between variables and function symbols by enclosing the former within angle brackets.

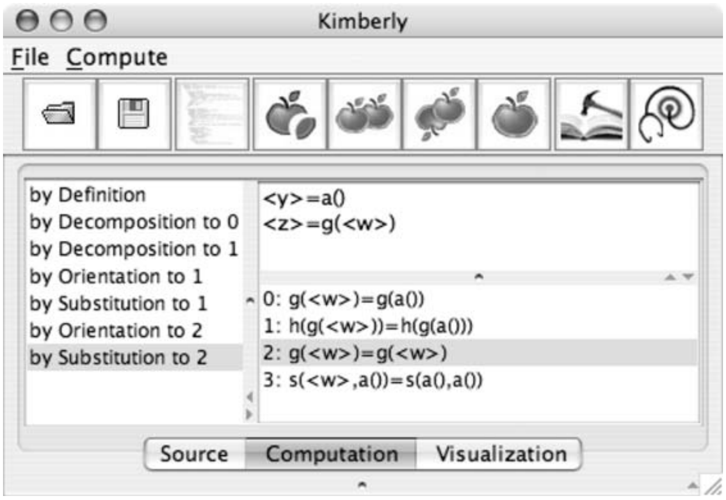
Once the initial equations have been parsed, the user may begin the process of transforming them by selecting an equation and a rule to be applied to it. The transformation rules include decomposition, orientation, substitution, elimination of trivial equations, occur-check, and detection of clash (cf. [8]), each of



```
f(a,h(<z>), g(<w>)) = f(<y>, h(g(a)), <z>);
g(<w>) = g(<y>);
s(<z>, s(<w>, <y>)) = s(g(<w>), s(<y>, a));
```

```
compilation complete
```

Fig. 1. The upper section of the window contains a tabbed view to manage panels named according to their functionality. The source panel, shown, provides text editing. The lower section is used to display messages generated by the application



```
by Definition
by Decomposition to 0
by Decomposition to 1
by Orientation to 1
by Substitution to 1
by Orientation to 2
by Substitution to 2
```

```
<y>=a()
<z>=g(<w>)
```

```
0: g(<w>)=g(a())
1: h(g(<w>))=h(g(a()))
2: g(<w>)=g(<w>)
3: s(<w>,a())=s(a(),a())
```

Fig. 2. The computation panel presents a debugger-style view of a derivation. The left section of the panel shows the history of rule applications. The right section lists the current equations, with variable bindings listed on top and equations yet to be solved at the bottom

which is represented by a button in the toolbar of the "computation" panel, see Figure 2. If an invoked rule is indeed applicable to the selected (highlighted) equation, the transformed set of equations will be displayed and the history of the derivation updated.

The derivation process is inherently nondeterministic, as at each step different rules may be applicable to the given equations. The software tool not only keeps

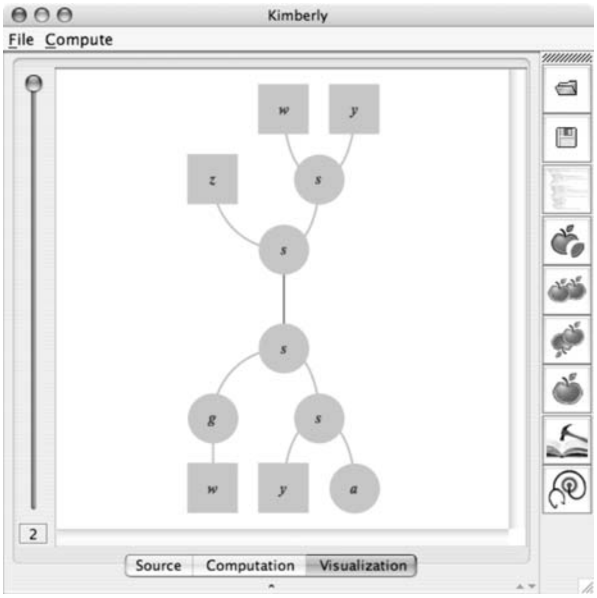


Fig. 3. The visualization module represents equations as pairs of vertically joined trees. Functions are depicted as circles and variables as rectangles. Tree edges are represented by elliptic arcs, while straight lines join the roots of two trees

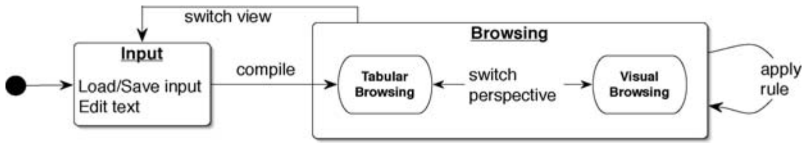


Fig. 4. Organization of user interaction

track of all rule applications, but also provides backtracking functionality to allow a user to revert to a previous point in a derivation and proceed with an alternative rule application.

Kimberly features a "visualization" module that supplies images of a trees representing terms and equations, and the user may switch between a textual and a graphical view of the current set of equations. The browser allows a user to view one equation at a time, see Figure 3.

The design of the user interface has been guided by the intended application of the tool within an educational setting. User friendliness, portability, and conformity were important considerations, and the design utilizes intuitive mnemonics and hints, regularity in the layout of interface elements, and redundant keyboard/mouse navigation. The transition diagram in Figure 4 presents a high-level view of user interaction.

Kimberly is a single document application, with all the user interface elements fixed inside the main window. Addition of a new browser to the application in-

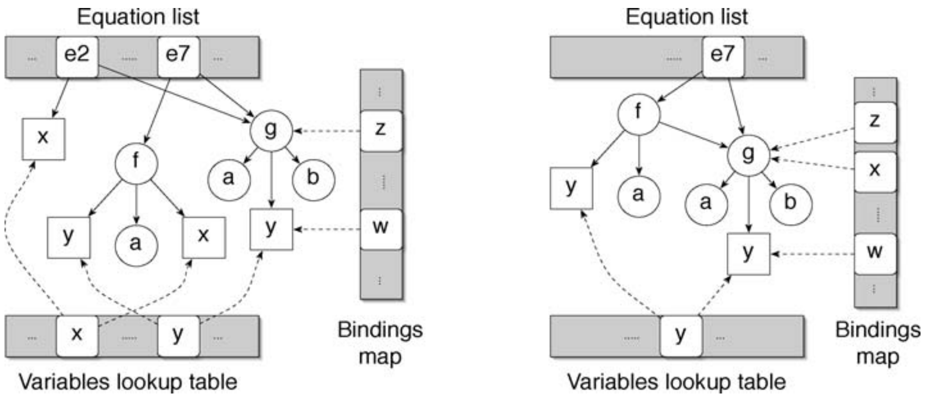


Fig. 5. Application of the substitution rule

volves only the introduction of a panel to host the new browser. The browsing architecture is designed to reflect interaction with the active browser, e.g., backtracking to an earlier state, to the remaining browsing modules.

For efficiency reasons we internally represent terms and equations by directed acyclic graphs, and utilize various lookup tables (dictionaries) that support an efficient implementation of the application of transformation rules. Specifically, we keep those equations yet to be solved in an "equation list," separate from variable bindings, which are represented by a "bindings map." An additional *variables lookup table* allows us to quickly locate all occurrences of a variable in a graph. The latter table is constructed at parse time, proportional in size to the input, and essential for the efficient implementation of applications of the substitution rule.

Figure 5 demonstrates the effect of an application of the substitution rule on the internal data structures. The rule is applied with an equation, $x \approx g(a, y, b)$, that is internally represented as *e2* on the equation list in the left diagram. The effect of the substitution is twofold: (i) the equation *e2* is removed from the equation list and added (in slightly different form) to the bindings map and (ii) all occurrences of x are replaced by $g(a, y, b)$, which internally causes several pointers to be redirected, as shown in the right diagram.

3 Visualization of Directed Graphs

Many combinatorial algorithms can be conveniently understood by observing the effect of their execution on the inherent data structures. Data structures can often be depicted as graphs, algorithmic transformations of which are naturally suited for interactive visualization. Results on graph drawing [1] can be combined with work on planar labeling for animating such dynamics. At the time this article was written no public domain software packages were available that were suitable for our purposes, therefore we decided to develop a novel method for browsing animated transformations on *labeled* graphs.

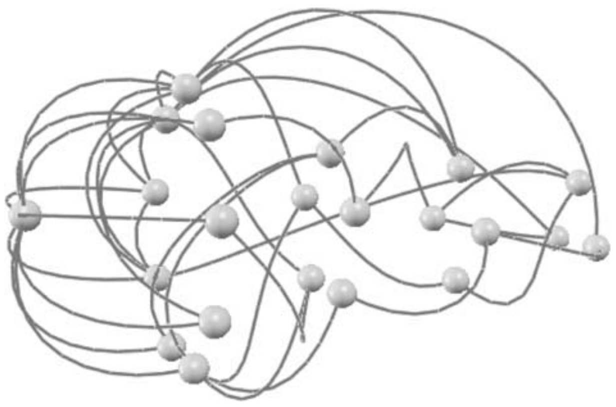


Fig. 6. A 3D layout of a randomly generated directed acyclic graph

We conducted a number of experiments in interactive graph drawing to equip *Kimberly* with an adequate visualization module for the representation of directed graphs. In particular, we studied issues of graph layout and edge routing (and plan to address labeling and animation aspects in future work).

Our graph drawing procedure is executed in two stages. The *layering step*, which maps nodes onto *vertices* (points in $\mathbb{R}^3$), is followed by the *routing step*, which maps edges onto *routes* (curves in $\mathbb{R}^3$).

In [2] it was shown that representing edges as curved lines can significantly improve readability of a graph. In contrast to the combinatorial edge-routing algorithm described in [2] (later utilized in [5] to address labeled graphs) our algorithm is iterative and consequently initialization sensitive. We treat vertices as sources of an outward irrotational force field, which enables us to treat routes as elastic strings in the solenoidal field (routes are curves that stay clear of the vertices). A route is computed by minimizing an energy term that preserves both length and curvature. Our approach is similar to the "active contours" methods employed for image segmentation [7, 9].

The layering step involves computing a clustering of a directed acyclic graph via topological sorting. We sorted with respect to both minimum and maximum distance from a source vertex and found that the latter approach typically produces significantly more clusters than the former. We map clusters onto a series of parallel circles centered on an axis, see Figure 6.

4 Implementation

The design of *Kimberly* is centered around providing multiple views of the derivation process. A flexible architecture allows for straightforward introduction of new visualization modules (*perspectives*). To achieve the flexibility we have followed the canonical MVC (Model-View-Controller) paradigm [4–pp 4-5]. The *model* object of the application keeps the history of rule applications and information on the currently selected step in the derivation and the selected equation. The *controller* object inflicts rule-firings on the model and notifies the perspectives of the changes. Perspectives are automatically kept in synch with the model; A perspective may, for example, change selection parameters of the model which would mechanically cause other perspectives to reflect the changes.

Images generated during the graph drawing stages are cached, as in-time image generation may take unreasonably long and adversely affect user experience. Memory requirements for storing the images may be very high, and hence images are stored compressed. An alternative approach would be to rely on a customized culling scheme under which only the elements in the visible region (intersecting the viewport) are drawn, using fast rectangle intersection algorithms and other computational geometry techniques.

For the 3D experiments we built an application with animation and simulation processes running in separate threads, enabling us to adjust simulation parameters such as step size while observing the effects in real time. Routes are implemented as polylines and computed by iteratively displacing the joints (mass points) under the sum of internal and external forces until the displacements become negligible. Motion of mass points is considered to be in a viscous medium and is therefore calculated under the assumption of $F = m\frac{dx}{dt}$. Segments of the polyline are treated as strings with *non-zero* rest length to encourage an even distribution of the mass points. At each step forces are scaled appropriately to ensure there are no exceedingly large displacements. Once the visualization scheme is fully understood we plan to base a *perspective* on it and make it an integral part of *Kimberly* .

5 Future Work

Further experimentation is needed to address the labeling and animation of graph transformations. We plan to use the same edge-routing algorithm to compute trajectories for animation of graph transformations. An advantage of drawing graphs in 3D is that edge intersections can always be avoided. But cognitive effects of shading and animation on graph readability need to be further investigated. Although the extra-dimensional illusion is convenient for setting up interaction, it is apparent that bare “realistic” 3D depictions are not sufficient.

Herman et al. [6] point out that while its desirable to visualize graphs in 3D because the space is analogous to the physical reality, it is disorienting to navigate the embeddings. Our research suggests that highly structured 3D layouts are more beneficial in this sense than straightforward extensions of the corresponding 2D methods because they are less disorienting. While our layout possesses axial symmetry, one can imagine a decomposition into spherical or toroidal clusters. The authors of [6, 3] corroborate our conviction that there is no good method for viewing large and sufficiently complex graphs all at once, and that interactive exploration of a large graph is indispensable. A key issue is how to take advantage of the graphics capabilities common in today’s computers to increase the readability of graphs.

Once the directed-graph browser has been integrated in the existing tool, the next step would be to consider applications to other derivation-based methods. Possible educational applications include grammars, as used for the specification of formal languages. The current system focuses on the visualization of derivation-based methods; future versions are expected to feature more extensive feedback to a user during the derivation process.

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing.* Prentice Hall, 1998.
2. David P. Dobkin, Emden R. Gansner, Eleftherios Koutsofios, and Stephen C. North. Implementing a general-purpose edge router. In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 262–271. Springer-Verlag, 1997.
3. Irene Finocchi. *Hierarchical Decompositions for Visualizing Large Graphs.* PhD thesis, Universita degli Studi di Roma "La Sapienza", 2002.
4. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
5. Emden R. Gansner and Stephen C. North. Improved force-directed layouts. In *Proceedings of the 6th International Symposium on Graph Drawing*, pages 364–373. Springer-Verlag, 1998.
6. I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.
7. M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331, 1987.
8. R. Socher-Ambrosius and P. Johann. *Deduction Systems.* Springer, 1996.
9. C. Xu and J. Prince. Snakes, shapes, and gradient vector flow. *IEEE Trans. Image Processing*, 7(3):359–369, 1998.