Realistic Program Visualization in CafePie

Tohru Ogawa and Jiro Tanaka

Institute of Information Sciences and Electronics

University of Tsukuba

Tennodai 1-1-1, Tsukuba, Ibaraki 305-8573 Japan

tohru@softlab.is.tsukuba.ac.jp and jiro@.is.tsukuba.ac.jp

ABSTRACT
CafePie is a visual programming environment for CafeOBJ, an algebraic specification language based on term rewriting. CafePie shows term rewriting directly by using two types of visualizations: animated cartoon-like and Obi-shaped. A more abstract visualization schema is necessary instead of program understanding at the programming language level. Therefore we investigate the visualization schema, which uses more realistic expressions.
Here we visualize term rewriting with more realistic expressions by using figures, pictures and images. In CafeOBJ, rewriting rules are called "equations." An equation is composed of operators and variables. We map operators to realistic expressions so that equations are expressed as transformations of realistic expressions. We use visual transformation rules which give the program pictorial expressions.

Keywords
Visual Programming, Specification Languages,
Computer Human Interaction

1. INTRODUCTION

A visual programming system (VPS) visualizes the structures of programs in two or more dimensions using visual expressions such as graphics, pictures and so on. Much work has been carried out on visual programming (Myers, 1990). It is important to reflect the user's view about VPS. Most of users want to be able to frequently edit and execute the programs. A VPS, in which program can be edited and executed visually, is desired.

On the other hand, algebraic specification languages (ASL) are specification languages which express the models of the real world using elements such as sorts, operators and equations. We assume specifications written in ASL as programs. They are executable and the program executions are performed using term rewriting.

We have already developed a visual programming system CafePie(Ogawa and Tanaka, 1998a, 1998b; Ninomiya and Tanaka, 1996) (Pictorial Interactive Editor for CafeOBJ) for the algebraic specification language CafeOBJ (Nakagawa et al., 1997). CafeOBJ specification consists of module structures. Our system visualizes each module. We use the direct-manipulation techniques for program editing. Most of editing is performed with only the mouse. The same visualization schema is used for both program editing and execution. Since the program editing and execution are performed in one window, program modifications are reflected directly in program execution.

2. PROGRAM VISUALIZATION

The "visualization of program structure" means to express the structure of a program using some pictorial or graphical objects. Several program visualization systems, such as PP[4], have already been proposed.

We visualize the program structures of CafeOBJ by expressing the program elements with pictorial objects. We call each pictorial object an "Icon." We have chosen the following primitive elements for CafeOBJ: Sort, Operator, Variable and Equation. These elements are expressed as specific icons on the screen.

We assume terms as the basic data structures of ASL. By definition, a term is a variable or an operator, which can have other terms as its arguments. On the other hand, the relation between a sort and a term is like that of a container and its contents. The representation of a sort should be different from that of terms. We visualize a sort by a rectangle and an element of a term by an oval. Further, to distinguish operators and variables, we visualize an operator as a pale blue oval and a variable as an orange oval. It is important that these icons have colors. Users can distinguish colored icons easily. We express sorts and operators, which are the main elements of the module, by bright colors and the other elements by dark colors. The visualization rules, which are given by default in CafePie,

for these elements are shown below.

Sort Icon: We use a directed graph to describe the implication relations of sorts in our CafePie system. Sorts are represented using nodes and implication relations are represented using arcs. Many ordered sorts are used in CafeOBJ. Their relations mean implication. The implication relations are often described with Ben's diagram. This diagram is easy to understand as far as it is used to describe simple relations, but it becomes difficult when describing complicated relations (the left hand side of Fig. 1).

Term Icon: Before considering operator or variable representation, we consider the visualization of terms. We express terms with tree structures. The components of the term's icons, i.e. an operator or a variable, are represented using a node, and super-sub relations between the components are represented using an edge (the right-hand side of Fig. 1).

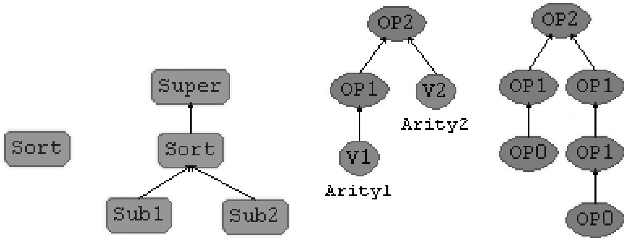

Fig. 1 Sort and Term Icons

Operator and Variable Icon: An operator, which has a coarity sort and arguments (arities), is represented with a pale blue oval and has a label for the operator name. The labels of arities are arranged at the bottom part of the operator. The label of coarity is arranged at the top part of the operator. Arrows are drawn from arities to operator and from operator to coarity. A variable is represented with the orange oval, and the sort of the variable is represented at its lower part (Fig. 2).

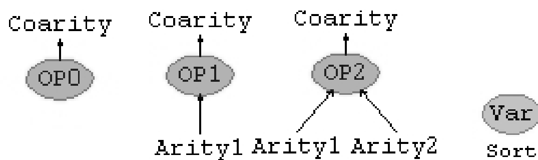

Fig. 2 Operator and Variable Icons

Equation Icon: An equation represents a term rewriting rule. A label is arranged in the center of the equation. The initial term is arranged on the bottom-left side of the label, and the rewritten term is on the bottom-right side. To represent a term rewriting rule, arrows from the left term to the right term are drawn via the label (Fig. 3).

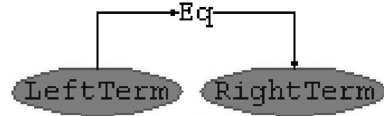

Fig. 3 Equation Icon

Module Field Icon: A program in CafeOBJ consists of modules. A Module contains other primitive elements: Sort, Operator, Variable, Equation. A Module is represented as a gray rectangle called "Field Icon." Users edit the module definition on this special Icon.

3. THE SYSTEM

In CafePie, all program-editing operations are handled in a uniform manner using direct-manipulation such as drag-and-drop and double-clicks. Program editing and execution are both performed in one window, which makes it possible to execute the program during the editing phase (Ogawa and Tanaka, 1998a).

The Process of programming in CafePie consists of the following:

1. Make a module,
2. Declare sorts and relations between sorts on the module,
3. Declare operators and variables on the module, and
4. Declare equations on the module.

A snapshot of CafePie is shown in Fig. 4. *Assistant Operation Part* consists of buttons, and is used for loading/saving a file, watching the help and so on. *Text Input Part* is used to input the names of icons. *Working Part* consists of *New-Field* and *Module-Field*. Users edit a program in *Working Part*. *New-Field* is used to make a new icon (such as sort, operator, variable and equation) on the module. *Module-Field* shows the current module to edit.

The user pushes the *File* button to load the file "*stack.mod*" (Program 1), which is a specification of *STACK* written in CafeOBJ language, and then a visualized program appears in Module Field of Working Part (Fig. 5). The user can modify the program using drag-and-drop operation.

Fig. 4 A snapshot of CafePie

```
Module TRIV { [ Elt ] }
Module STACK [X: TRIV] {
   [ NeStack < Stack ]
   signature {
      op empty :    -> Stack { Constr }
      op push : Nat Stack -> NeStack { Constr }
      op pop : NeStack -> Stack
      op top : NeStack -> Elt
   }
   axioms {
      var S : Stack
      var E : Elt
      eq pop(push(E, S)) = S.
      eq top(push(E, S)) = E.
   }
}
```

Program 1 STACK specification in CafeOBJ

Program Execution

In CafeOBJ, program execution is carried out using term rewriting. Our goal is visualizing the processes of term rewriting to show the execution process in a more intuitive way.

The execution process includes the following:

1. Make an initial term as input,
2. Send the initial term to the external CafeOBJ interpreter,
3. Receive the execution result and make the tracing process for each rewriting step from the received data, and
4. Show the tracing process.

The term is edited on the *Module-Field*. Moving the term onto the label of the module involves the program execution. The process of the program execution is carried out as follows. CafePie first sends the module definitions to the interpreter and then queries the input term to the interpreter. Finally CafePie receives the execution trace as a result. The result is processed by CafePie and is shown in the visualized form. Tracing result consists of terms, which emphasizes the process of reductions.

Fig. 5 A program visualized by CafePie

When CafePie gets the result, the input term is rewritten to the next term. CafePie shows the terms one after another like an animated cartoon. This is a dynamic representation and suits to check the rewriting flow at a time. Fig. 6 shows a process of term rewriting whose initial term is s(s(0)) + s(s(s(0))) of the module SIMPLE-NAT[1,3] and rewritten term is s(s(s(s(s(0))))).

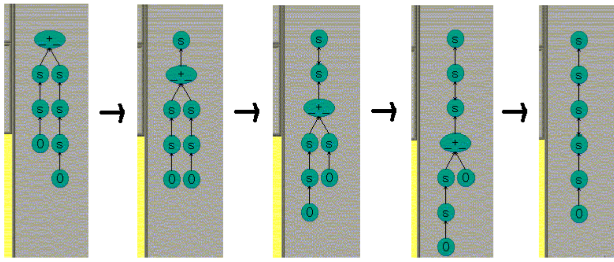

Fig. 6 Process of term rewriting visualized like an animated cartoon

After showing the last term, CafePie represents the tracing diagram in Obi-shape. This is a static display and suites to check one reduction process more closely. To let the user know which equation is used for the reduction, the equation label is represented between the subterms on which the reduction is carried out. It is an effective dynamic representation to appeal the term rewriting.

4. REALISTIC EXPRESSION

As explained above, CafePie can map directly the program codes to graphical objects as shown in Fig. 5. The process of term rewriting is visualized as tree structures consisting of icons. For example, CafePie visualizes the term

push(E3:Elt, push(E2:Elt,
push(E1:Elt, push(E0:Elt, empty)))),

as shown in Fig. 7. These visualizations are difficult to understand intuitively because CafePie uses only one-to-one associations between the program code and the icons. More realistic expressions of higher abstraction level are desired.

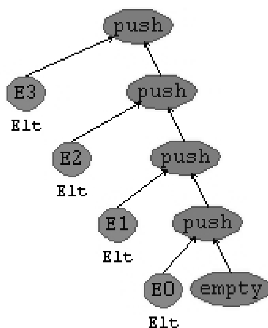

Fig. 7 STACK structures in CafePie

Visual Transformation Rule

Here we propose to visualize term rewriting with more realistic expressions, which use figures, pictures and images. We call these expressions visual objects. Conceptual images of the program need to be transmitted to users without presenting the program code. We consider the means of expressing actual objects. The actual object is characterized by its property such as shape or behavior. We pay attention to the shape. Therefore, we propose to use visual transformation rules so that the user can customize the shape of visual objects.

In CafePie, rewriting rules are called "equations." An equation is composed of operators and variables. There are two kinds of operators. One is the constructor of the coarity sort and the other is not. We call the former operator "C-op" and the latter "NC-op." In a term rewriting, the result term consists of C-op. If a term has NC-op, it is not a result term and it should be rewritten.

Users can map C-op to realistic expression for defining visual transformation rules. For example, the operators *empty* and *push* in the *STACK* specification are C-op. They can be defined using visual transformation rules.

Users often imagine *STACK* structures as data structures like piled-up packages. The operator *empty* is represented with a rectangle (the right hand side of Fig. 8) instead of the original visualization (the left hand side of Fig. 8).

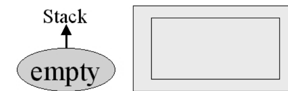

Fig. 8 Visualization of the operator *empty*

The operator *push* is visualized like the right hand side of Fig. 9. This figure means that the rectangle with *Elt* is arranged at the upper part of *Stack*.

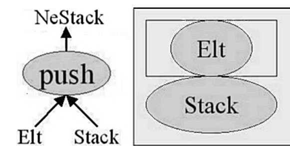

Fig. 9 Visualization of the operator *push*

NC-op has rewriting rules called "equations" because terms, which have the operator, should be rewritten. These equations can be expressed by using C-op's realistic expressions. We consider that NC-op is an action of term rewriting.

The operator *pop* is NC-op. It has the equation *pop(push(E, S)) = S*. The equation is visualized as shown in the right hand of Fig. 10. *Pop* is shown as the label of the arrow. It means an action. This figure means that the *STACK* without the top element is drawn after the *pop* action.

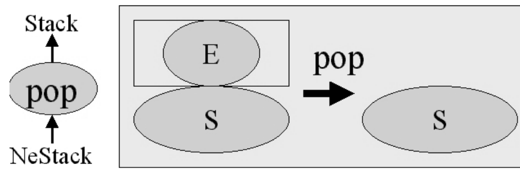

Fig. 10 Visualization of the operator *pop*

The operator *top* is also NC-op. It has the equation *top(push(E, S)) = E*. This is defined like the right hand of Fig. 11. This figure means that the top element of the *STACK* is drawn after the *top* action.

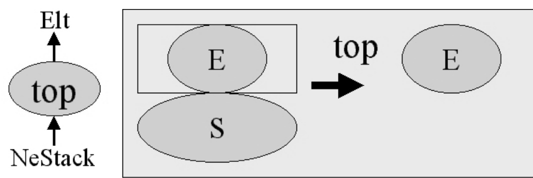

Fig. 11 Visualization of the operator *top*

Defining Visual Transformation Rule

We have developed an environment, which can edit these visual transformation rules on CafePie using the direct-manipulation. In our approach, these rules are defined by using no drawing action but a combination of visual objects which have been prepared. Therefore, the user can easily define the rules using the same paradigm as the program editing.

The editing of the rules needs the following two steps: preparing the visual objects for making the rules and defining the geometrical relations between the objects.

1. Preparing the visual objects: the system has some elementary figures such as rectangle and circle, and the user can make use of some images from files. If operator has arguments, the user can also use them as visual objects.
2. Defining the geometrical relations: to define the geometrical relations, the user creates a relation between two objects repeatedly. The user can also handle the related objects as one object.

The relation is given by the drag-and-drop operation. Suppose there are two objects: A and B. The user moves B toward A. When the user drops B onto A, dotted lines appear around A as shown in Fig. 12. These lines show the expected location of B. The location of B is selected among any of the nine parts of A by default: the upper left hand, the upper part, the upper-right hand, the left hand, the center, the right hand, the lower-left hand, the lower part or the lower-right hand (Fig. 12).

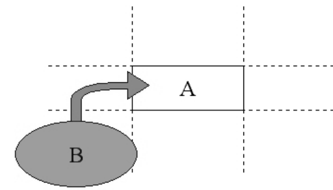

Fig. 12 Creating a relation between two objects

If B is dropped on the upper or lower part of A, B is arranged to stick to A. If B is dropped on the right or left side of A, they are also arranged to be close together. If B is dropped in the diagonal part of A, B is placed on the vertex of A.

The size of the dragged element is determined by the position of B. If the user drops B in the center of A, the size of B becomes smaller than A (the no.1 of Fig. 13). If B is dropped in the left or right part of A, the height of B is modified to have the same height as A (the no. 3 and 4 of Fig. 13). If B is dropped in the upper or lower part of A, the width of B is modified to have the same width as A (the no. 2 and 5 of Fig. 13). If B is dropped in the diagonal part of A, the size of B is changed to be the same size of A (the no. 6,7,8 and 9 of Fig. 13).

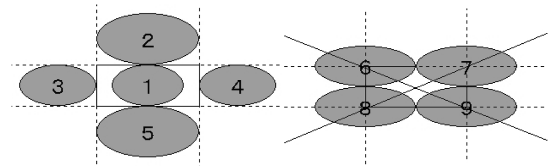

Fig. 13 Arrangement plan between two objects

If the user wants to change the location or the size of B, the user can modify these geometrical values by using drag-and-drop.

Fig. 14 shows the process of defining the visual transformation rules of the operator *push* shown in Fig. 9.

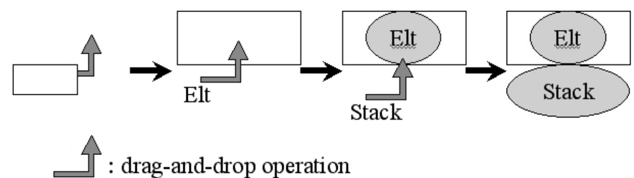

Fig. 14 Process of defining a visual transformation rule

First, the user makes a rectangle. Second, he moves the argument *Elt* of *push* in the center part of the rectangle. Third, he moves the argument *Stack* of *push* at the bottom part of the rectangle with the argument *Elt*. And finally, the visual transformation rule of *push* is defined. In this way, these rules are also edited using the same paradigm of

program editing and are performed using drag-and-drop.

The Term Rewriting using Visual Transformation Rules

Users can edit terms by using the visual transformation rules defined above. For example, the user substitutes the term

push(E0:Elt, empty)

for the variable *S1:Stack* of the term

push(E3:Elt, push(E2:Elt, push(E1:Elt, S1:Stack))) (Fig. 15).

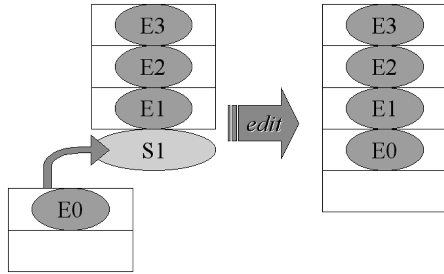

Fig. 15 Term editing using visual transformation rules(1)

After that, the term

push(E3:Elt, push(E2:Elt, push(E1:Elt, S1:Stack)))

is changed to the result term

*push(E3:Elt, push(E2:Elt,
push(E1:Elt, push(E0:Elt, empty)))).*

This visualization corresponds to Fig. 7.

Fig. 16 shows editing a term which has NC-op. This figure shows substituting the term

push(E1:Elt, push(E0:Elt, empty))

for the variable *NeStack* of the term

pop(NeStack).

After that, the term *pop(NeStack)* is also changed to the term

pop(push(E1:Elt, push(E0:Elt, empty))).

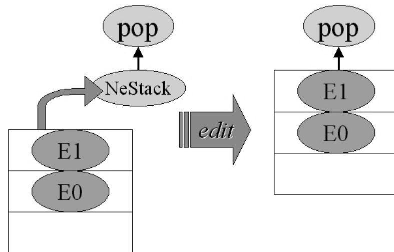

Fig. 16 Term editing using visual transformation rules(2)

In this way, terms can be edited using new visualization, introduced by visual transformation rules, while keeping up the operations paradigm of original editing.

Another Example

New visualization rules can be defined. For example, the elements of *Elt* are five expressions shown in Fig. 17.

Elt = { }

Fig. 17 Elements of *Elt* are five expressions

New visual transformation rules of these operators: *empty* and *push* can also be defined. The left hand side of Fig. 18 shows the new rule of the operator *empty*. This figure means "No Exit" because the Exit door has broken down. The right hand side of Fig. 18 shows the new rule of the operator *push*. This figure means a person, who has a face Elt, is rear of the Stack.

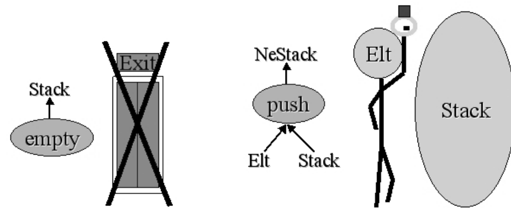

Fig. 18 Other visual transformation rules

Fig. 19 shows the term of the new visualization rules. Each person has a different expression. Each person cannot go forward because of the broken door. Only the person who is at the tail of the line can move. This mechanism also means the *STACK* structure. In this visualization, *STACK* means the line of people. In this way, programs can be expressed differently by defining different visual transformation rules.

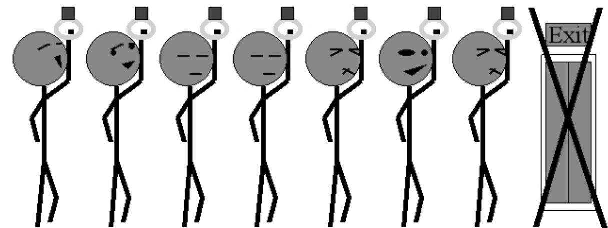

Fig. 19 Term expressed using the other visual transformation rules

5. RELATED WORKS

Various systems have been proposed, through which users can watch and analyze term rewriting system (TRS).

ReDux (Bundgen, 1993) is a workbench for TRS realized by a text-interface. ReDux has various interfaces with Knuth-Bendix completion algorithm and so on. They come up with various ideas in the text interface. However users cannot manipulate terms intuitively.

TERSE (Kawaguchi et al., 1994) is a visual support environment for TRS. The system can show the process of term rewriting visually. The system supports the environment for the program execution, but does not support program editing. CafePie visually supports not only program execution but also program editing. Users often understand the program through the execution and want to re-edit the program after that. Our main point is that CafePie can edit and execute the program visually. CafePie is the first system that shows the TRS execution dynamically. Moreover, users can easily modify the program visualization rules.

Visulan (Yamamoto, 1996) is a rule-based visual language. Bitmap-based visual transformation rules are used for expressing both programs and data. Visulan does not say anything about the operations of defining these rules.

VISPATCH (Harada et al., 1997) is a distributed figure-rewriting visual language. VISPATCH's rewriting timing and location are controlled by the user events. Rules are also normal figures and can be rewritten. These rules are defined by using drawing actions. Our system uses a combination of visual objects so that users can edit these rules as the same paradigm as the editing operations.

7 SUMMARY AND FURTHER RESEARCH

We have implemented CafePie, a visual programming environment for CafeOBJ. Term rewritings are visualized with more realistic expressions by using figures, pictures and images. We map operators to realistic expressions so that equations are expressed as transformations of realistic expressions. We use visual transformation rules which give the program pictorial expressions so that users can customize the term expression as they like. These rules are also edited using the same paradigm of program editing and are performed using drag-and-drop.

Our system CafePie is useful for the beginners of ASL. Our goal is to improve the system and to fascinate the advanced users.

REFERENCES

Ogawa, T., and Tanaka, J., 1998a, Double-Click and Drag-and-Drop in Visual Programming Environment for CafeOBJ," Proceedings of ISFST-98, pp.155-160, Hangzhou, China.

Myers, B. A., 1990, “Taxonomies of Visual Programming and Programming Visualization,” Journal of Visual Languages and Computing, Vol.1-1, pp.97-123,

Nakagawa, A. T., Sawada, T., and Futatsugi, K., 1997, “CafeOBJ User's Manual,” IPA, Japan.

Tanaka, J., 1997, “PP: Visual Programming System for Parallel Logic Programming Language GHC,” Parallel and Distributed Computing and Networks '97, pp.188-193, Singapore.

Kawaguchi, N., Sakabe, T., and Inagaki, Y., 1994, “TERSE: Term Rewriting Support Environment,” Workshop on ML and its Application, ACM SIGPLAN, pp.91-100, Florida.

Bundgen, R., 1993, “Reduce the Redex→ReDuX,” Rewriting Techniques and Application, LNCS 690, pp.446-450.

Yamamoto, K., 1996, “Visulan: A Visual Programming Language for Self-Changing Bitmap,” Proc. of International Conference on Visual Information Systems, pp.88-96, Melbourne, Australia.

Harada, Y., Miyamoto, K., and Onai, R., 1997, “VISPATCH: Graphical rule-based language controlled by user event,” In Proceedings of the 1997 IEEE Symposium on Visual Languages, pp.162-163, Capri, Italy.

Ninomiya, T., and Tanaka, J., 1996, “Visual Programming Environment for an Algebraic Specification Language,” JSSST’96, pp.213-216 Tsukuba, Japan, in Japanese

Ogawa, T., and Tanaka, J., 1998b, “Drag-and-Drop based Visual Programming Environment for CafeOBJ,” JSSST’98, pp.165-168, Tokyo, Japan, in Japanese.