PP: Visual Programming System for Parallel Logic Programming Language GHC

Jiro Tanaka

Institute of Information Sciences and Electronics,
University of Tsukuba, Tsukuba, Ibaraki 305, Japan
email: jiro@is.tsukuba.ac.jp
fax: +81-298-53-5206

Abstract

Though parallel logic languages have lots of good features, the readability of those languages are not good enough. Therefore, we have developed visual programming system PP for parallel logic language GHC. Each definition clause of GHC program is expressed as a figure along with its textual representation. The system graphically assists the program input and its execution. The system has actually been implemented in SICStus Prolog and available on conventional Unix workstations. Examples of actual program input and its graphical execution are shown.

1 Introduction

Various kinds of parallel logic languages, i.e., PARLOG [Clark 86], Concurrent Prolog [Shapiro 83] and Guarded Horn clauses (GHC) [Ueda 85], have been proposed [Shapiro 89]. The basic computation mechanisms of these languages are quite similar. Horn clauses with guards are used for defining predicates, goals are executed in parallel, and they have some synchronization mechanisms between goals.

Though these languages have lots of good features, such as *expressiveness*, language *simplicity* and *efficiency*, it is often said that the readability of those language is low. For beginners, they often have *mental* difficulty to learn. For even the skilled programmers, parallel logic languages are rather difficult languages to read.

Therefore, we have developed visual programming system PP [1] for those languages. It enables us to input program by figures. It can also *graphically* execute the program. As a target language for our system, we chose a flat subset of GHC which allows only system predicates in guard [2].

[1] PP stands for "Program-less Programming" or "Pictorial Programming."

[2] Though we picked GHC as our target language, it does not mean our system is only applicable to GHC. Our system is also adaptable to other parallel logic languages or Prolog with slight modifications.

1.1 Low readability of GHC

Our observations on the low readability of GHC program and the merits of graphical representation can be summarized as follows.

- **Arrangement of goals** Goals in each definition clause are shown in an one-dimensional manner. It makes difficult to understand the relations between goals. Therefore, if we can position goals *graphically*, it leads to an easier understanding of a program.
- **Avoiding names of arguments** Each goal exchanges information through its arguments. Arguments of goals can be shared by *sharing* their names, which is often not easy to trace. It will be convenient, if this *sharing* relation is expressed *graphically*.
- **Argument mode** Similar to Prolog, GHC does not describe *explicitly* whether an argument is used as an input or an output. However, we usually know the *intended* usage of an argument. It will make GHC programs more readable, if we can express the input and output mode of variables explicitly.
- **Integrated environment** We do not have the good module structures in GHC. It makes the understanding of the program difficult. Therefore, we would like to provide an *integrated* visual environment, which dynamically manages the definition clauses of GHC programs.

1.2 Problems on existing visual programming system

Normally the effort for visual programming starts from visualizing existing text based programming languages. Futamura's PAD is one of such effort [Futamura 81]. However, the visualization of an existing conventional language is not very interesting. It is because the language level of the existing conventional language is too low. We cannot use the same framework for visualizing program execution. Side-effects of conventional languages are also harmful for visualization.

Since declarative languages are side-effect free, these languages are more suited for visualization. Various visualization proposals have already been done for functional languages [Dennis 74, Davis 81, Keller 81], an object-oriented language [TGS 89], logic programming languages [Kurita 84, Cox 85] and parallel logic programming languages [Ringwood 89, Kahn 92]. Other approach is the proposal of iconic-based visual programming language, such as HI-VISUAL [Hirakawa 87].

However, most of these efforts do not seem to be successful enough. Some of them were just proposals and were not actually implemented. Some experimental implementations were done for the rest, most of them were not widely used.

Usually visual programming system is *experimentally* developed. The limited numbers of primitives are actually implemented. It is only good for expressing *toy* programs. The number of users, workable machines and environment are also very limited. Therefore, visual programming system is never used for actual software development.

We analyzed the reason that main difficulty arises from proposing new visual programming languages, completely separated from the text-based language. Generally speaking,

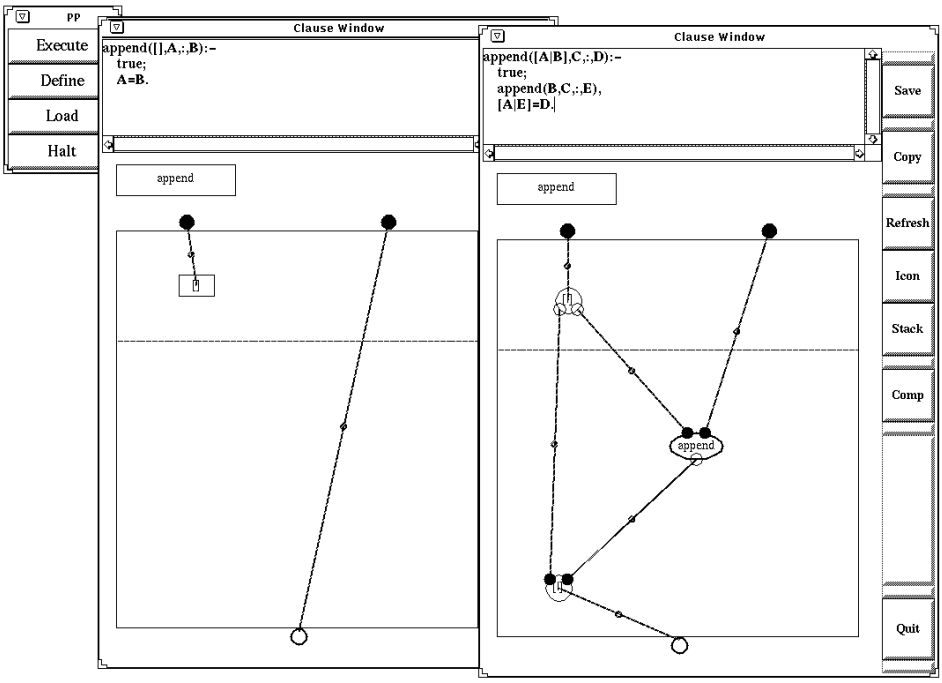

Figure 1: Program input example of *append*

program language should be *judged* by the total number of built-in functions, libraries, matching to Unix environment and available software tools.

We adopted a *hybrid* approach. In our system, graphical representation is always displayed accompanied with its textual representation, i.e., there always exists a textual representation which corresponds to the graphical representation. Since a textual representation is just a GHC program, it is always possible to utilize the latest implementation of GHC. It is also possible to use our system for part of the total system, mixed with the ordinary GHC system. We believe that our approach is more realistic and promising approach.

2 Visual programming system for GHC

In PP, as mentioned above, each definition clause of GHC program is expressed as a figure along with its textual representation. A program input example of *append* is shown in Figure 1.

When we start the system, PP menu pops up initially. PP menu consists of four buttons, i.e., "Execute," "Define," "Load" and "Halt." "Execute" is used to execute a program, after finishing the input of the program. "Define" is used to define a definition clause initially. "Load" is used to load definition clauses from the store. "Halt" is used to exit PP.

We can see two Clause Windows in Figure 1. These two windows correspond to the two definition clauses of *append*. Each Clause Window consists of two sub-windows, i.e., textual-view window and graphical-view window. The former displays the textual representation

of a program clause. The latter displays the graphical representation of the same program clause.

In PP, a new Clause Window is created, if we push "Define" button of PP menu. Two sub-windows of the Clause Window, i.e., textual-view window and graphical-view window are initially empty. We can input a clause definition from either sub-windows. If we input a figure from the graphical-view window, the corresponding textual code is automatically generated by the system. On the other hand, if we input the text from the textual-view window, the corresponding figure is also automatically created. The change of one sub-window immediately effects the display of the other sub-window, i.e., two sub-windows are *causally connected* to each other.

PP can be looked at from the viewpoint of MVC (Model-View-Controller), which is originally proposed in Smalltalk-80 [Cox 86]. Model corresponds to the store which contains definition clauses of the program. PP provides two Views, i.e., textual-view and graphical-view, for each definition clause. Controllers are attached to each View and accept messages from the user.

We also tried to minimize the *modes* of the system. For example, PP menu is always available, while PP is running. Since there is no specific execution mode, we can perform various kinds of works *in parallel*. For example, we can open arbitrary numbers of Clause Windows. We can perform *define*, *load* and *execute* in parallel. Since PP is completely fitted in X Window system and Unix, it can be utilized as one of software tools in the process of software development.

3 Input from graphical-view window

Figure 2 shows the program input process when the first definition clause of *append* is entered from the graphical-view window.

Graphical-view windows consists of three regions, i.e., predicate-name, guard and body regions. We can input a predicate name, which is initially set to *no_name*. This is done by clicking the predicate-name region and typing in the new predicate name. We can click the appropriate place of graphical-view window to create input and output arguments. Graphical-view window is clever enough to create appropriate figure elements, such as input arguments, output arguments, goals and connecting lines, depending upon the place where the click from the user is entered.

We can also *graphically* edit the definition clause. For example, we can *delete* or *move* goals *graphically*. When we move goals, arguments of goals and connecting arcs are also moved automatically.

The input from graphical-view window also effects the textual-view window. The corresponding textual codes have been created *incessantly*, as we input an element of a figure from the graphical-view window.

The textual notation is the one of ordinary GHC, except that ";" is used for expressing guard [3] and ":" is used to show the mode of arguments. We arrange input arguments before

[3] This comes from the implementation restriction of SICStus Prolog.

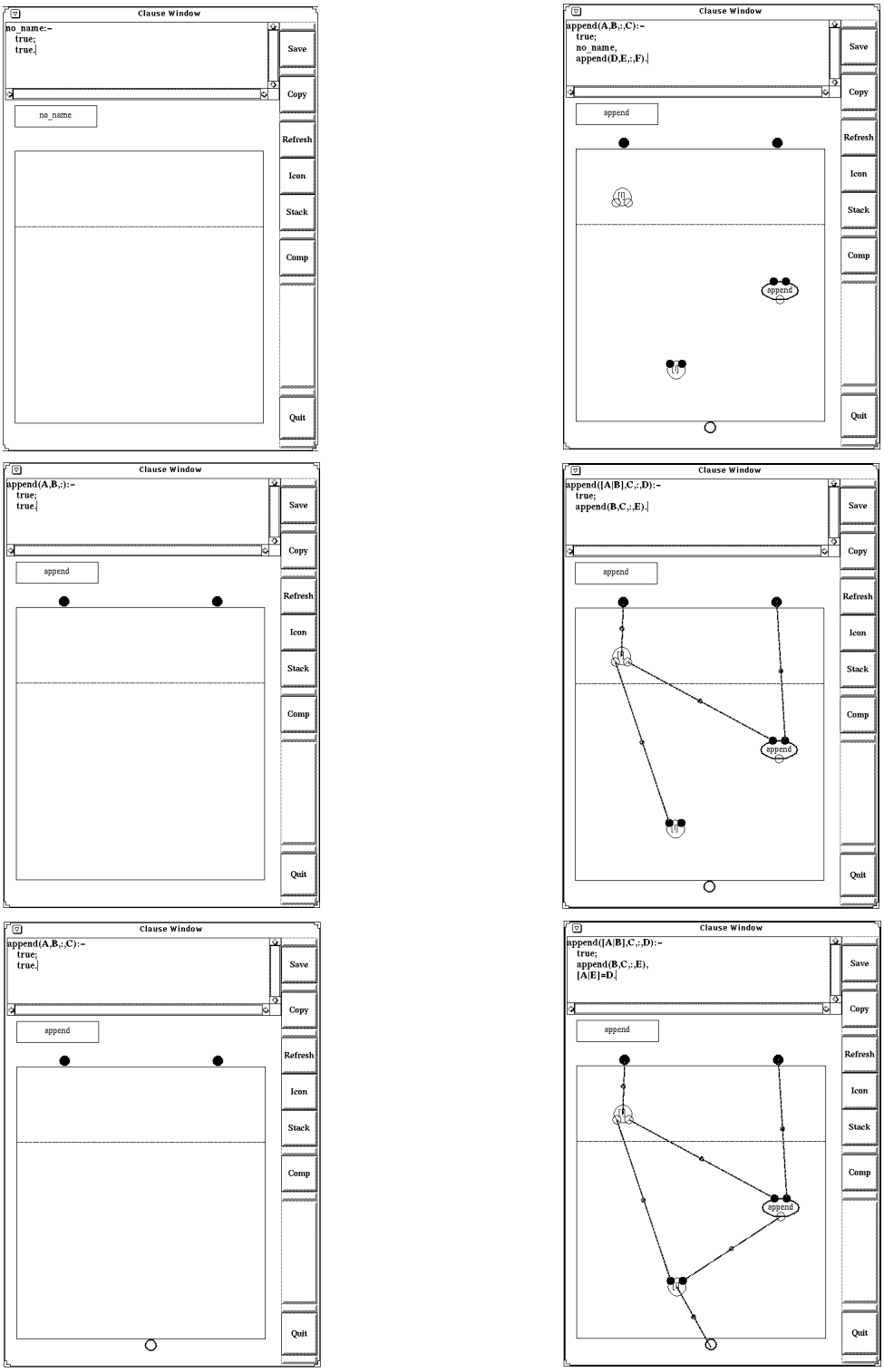

Figure 2: Input from graphical-view window

":" and output arguments after ":" [4]. As mentioned before, GHC has no *explicit* notion of input and output. Therefore, the distinction of input and output is just for convenience for the graphical display. We can simply neglect the mode, if needed, at run time.

We should also note that we did not use any name of variable in the graphical-view, window so far. Variables which consists of one capital letter, such as "A" and "B" are automatically generated.

One problem of generating textual code from the graphical notation is that sometimes textual notation cannot be determined uniquely. For example, we have generated the following text in the case of Figure 2.

```
append([A|B],C,:,D):-
  true;
  append(B,C,:,E),
  [A|E]=D.
```

However, some people may not be satisfied with the above code and prefer the following code.

```
append(A,B,:,C):-
  A=[D|E];
  append(E,B,:,F),
  [D|F]=C.
```

Of course, this is the matter of preferences in a certain extent. Our design choice is that we substitute guard unification to head, if possible, and guard or body is displayed as *true*, even if it is empty.

4 Input from textual-view window

Figure 3 shows the case, in which the same definition clause of *append* is entered from the textual-view window. The textual-view window is just an Emacs window and we can input text from this window.

The input from textual-view window effects the graphical-view window incessantly. Every time we input delimit symbols, such as "(", ",", and ":-", drawing routine is triggered and the corresponding graphical view is created and displayed.

As you see in Figure 3, PP has a very *sophisticated* parser which *completes* the insufficient information from textual-view and always generates a syntactically correct GHC definition clause.

[4] This notation is not *upward* compatible to existing GHC programs. To make it compatible, we also need to prepare *annotations* to specify the mode of arguments.

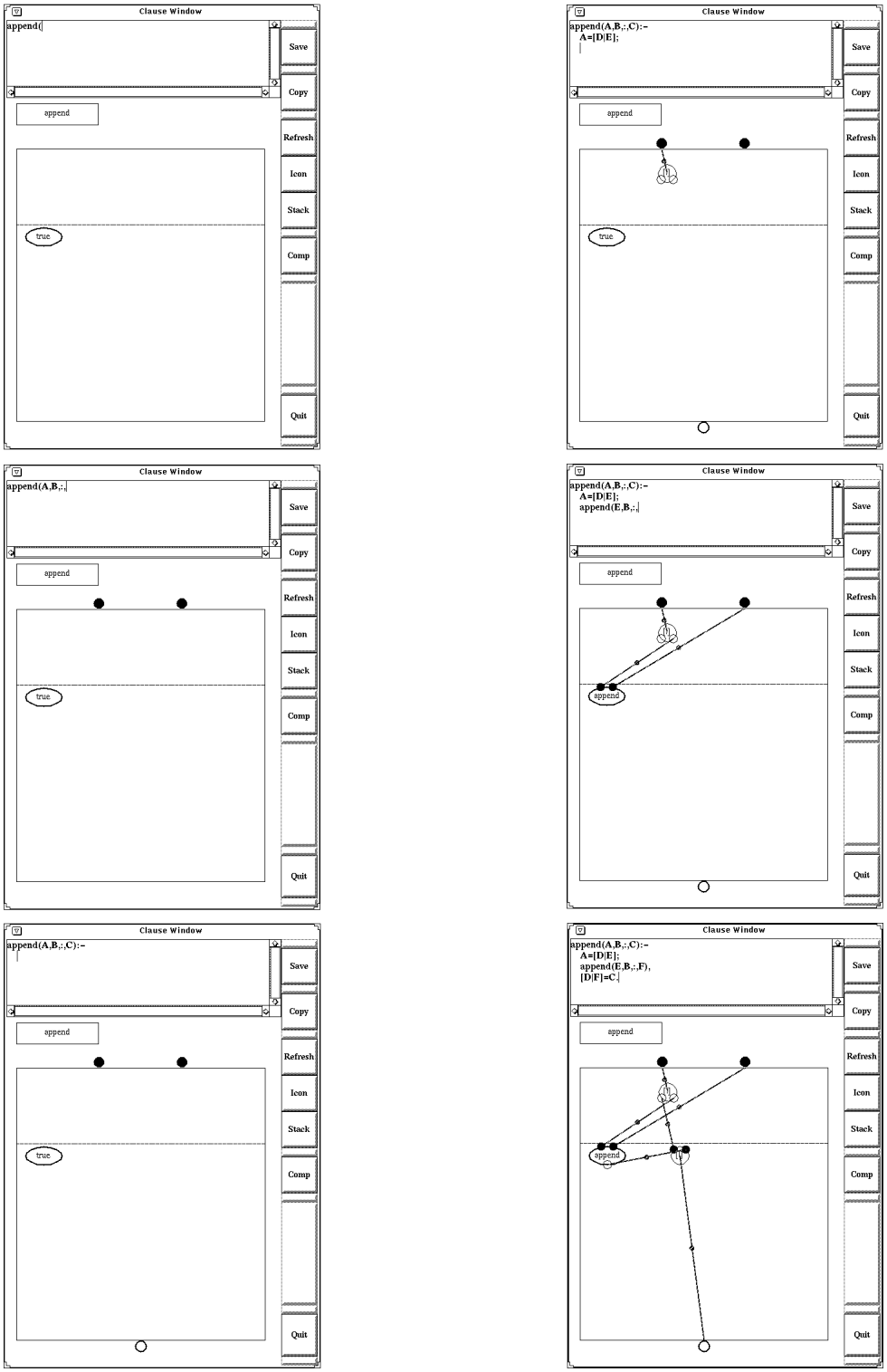

Figure 3: Input from textual-view window

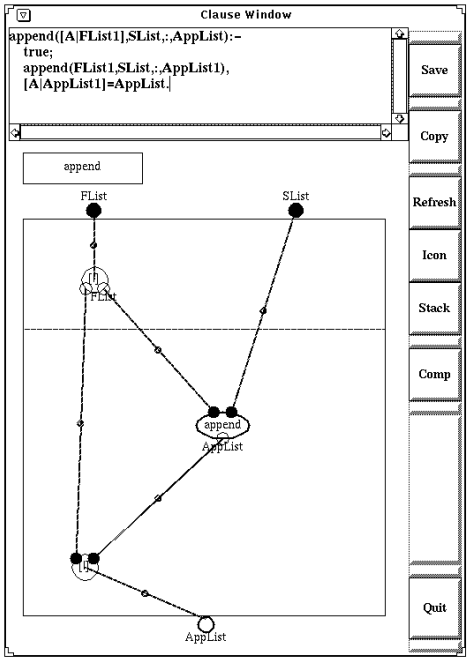

Figure 4: An example of labeled variables

5 Labeled variables

One of the advanced feature of PP is the treatment of variables. We did not use the name of variable in the graphical-view so far. In the textual-view, we simply used variables which consists of one capital letter.

However, we sometimes want to call a certain variable with a name. In general, the name of the variable may include the information about the *usage* of the variable. For example, a pair of variables which are used to express a difference list may be called "Head" and "Tail."

We prepared *labeled* variables for such purposes. PP has two kinds of variables, i.e., unnamed and labeled variables. An unnamed variable consists of upper-case letters and numbers only. They do not include lower-case letters in their name. They are not displayed in the graphical-view.

On the other hand, labeled variables include lower-case letters in their name, e.g., "Head" and "Tail". Labeled variables are displayed in the graphical-view.

Figure 4 shows the example in which labeled variables are included in the graphical-view. In PP, a duplicate use of the label is allowed in the graphical-view, which is distinguished by putting numbers at the postfix position in the textual-view.

Figure 4 may need some explanation. The predicate *append* has two input arguments and one output argument. We labeled those as $FList$, $SList$ and $Applist$ in the graphical-view. Since *append* is called recursively, it is informative to use the same labels also in the recursive call part. In PP, identical labels are distinguished by putting numbers at the postfix position in the textual-view. However, as mentioned before, guard unification is substituted to head, if possible, in the textual-view. Therefore, only $FList1$ shows up in the textual-view.

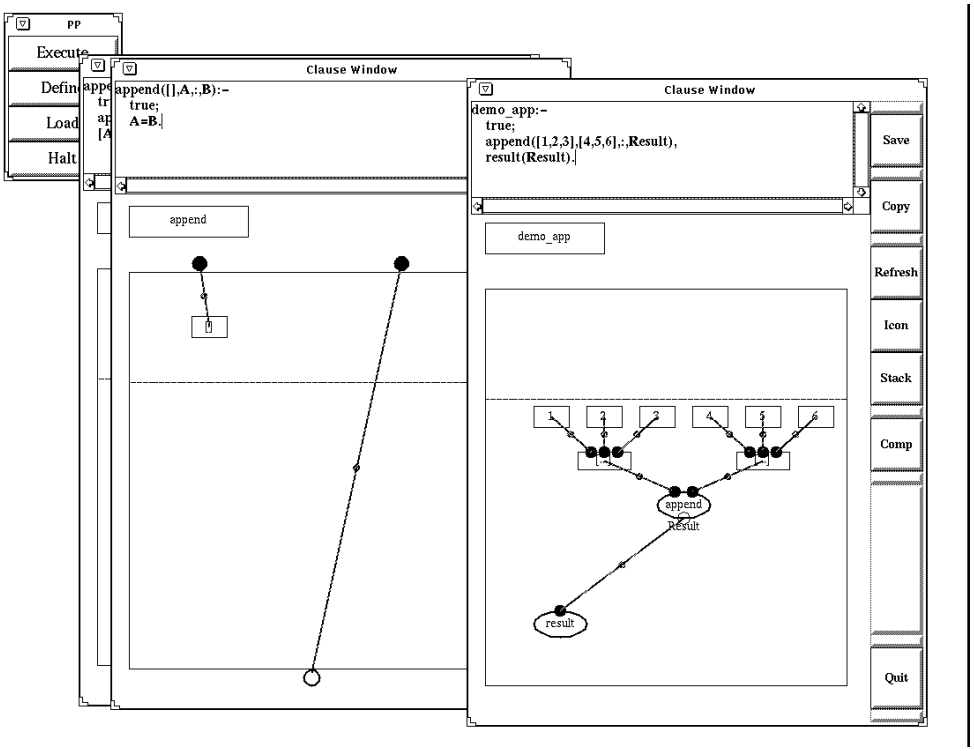

Figure 5: *demo_app* example

6 Program execution example

Figure 5 shows the *demo_app* example. In this case, the definition clause of *demo_app* was entered from the textual-view window. Note that *pretty drawing*, which is compatible to the textual notation of list, has been introduced to increase the readability of the graphical representation.

This *demo_app* program just tries to append two lists. We can execute this program by pushing "Execute" button of PP menu. Execution Window is created, if we push "Execute" button.

We can execute GHC goals in "Breadth-first," "Depth-first" or "Parallel" manner [5].

We enter a GHC goal, in this case, *demo_app* goal, to Execution Window and push "Run" or "Step" button of Execution Window [6]. The snapshots of program execution are shown in Figures 6 and 7.

In GHC, the end of computation means that all goals are disappeared from the goal queue. the goal which will be reduced to itself. Note that the "result" goal is used to keep the computation result on Execution window [7].

The execution by Execution Window is quite interactive. It is possible to edit goals in

[5] "Parallel" means to reduce all GHC goals in a goal queue at the same time.

[6] "Step" means stepwise execution.

[7] The "result" goal can be defined as a predicate which calls itself recursively.

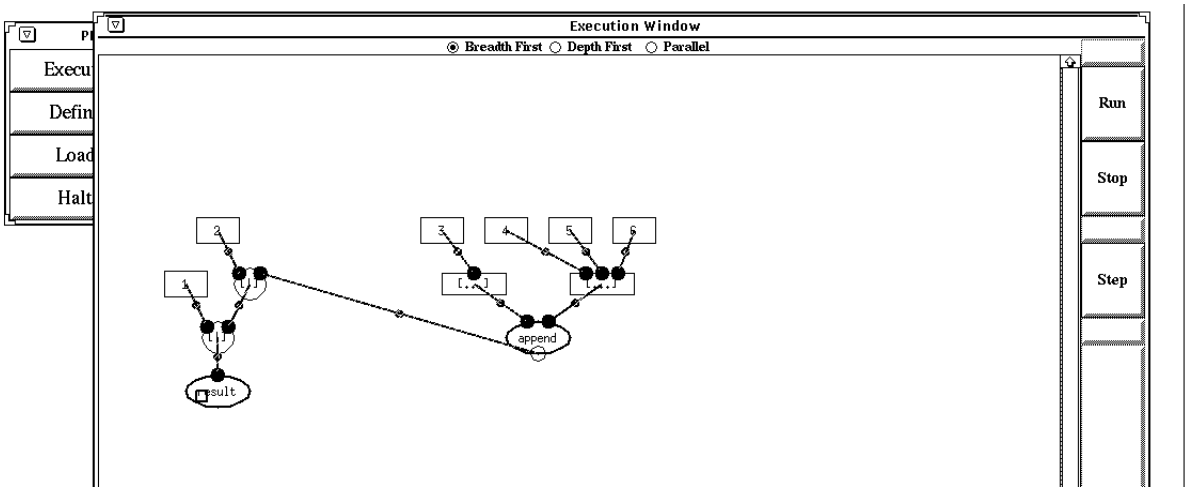

Figure 6: Program execution snapshot(1)

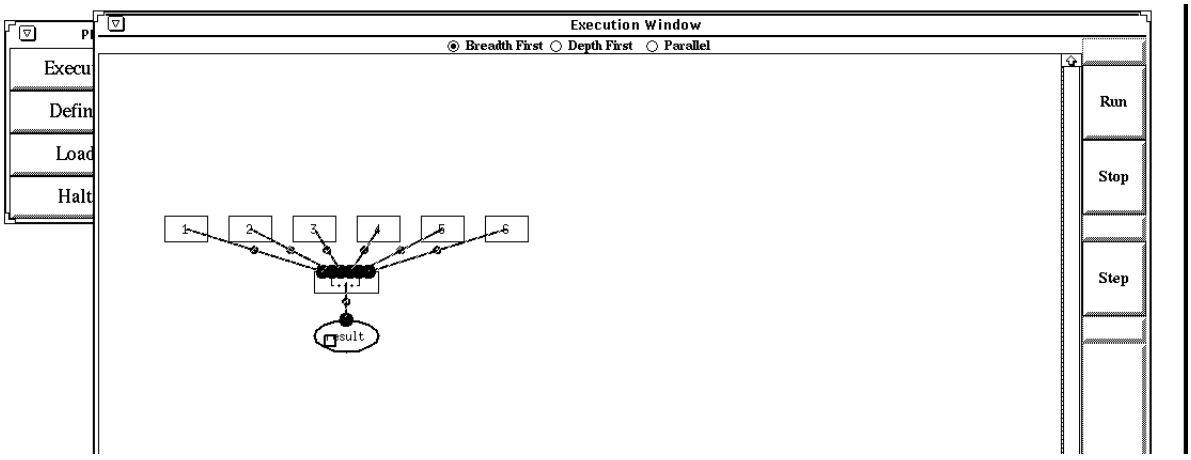

Figure 7: Program execution snapshot(2)

the middle of the execution. For example, we can delete goals, add new goals, or modify the connection of arguments in the middle of goal execution.

7 System implementation history

System implementation effort of PP has been started in 1989. The first version and the second version have been implemented 1990 and 1992, respectively. Both systems were developed on top of PSI-II. The first version is written in GHC, the second, in KL1. These was carried out as a part of the Fifth Generation Computer Project. The second version was demonstrated at the FGCS'92 in 1992. This version has been registered as ICOT Free Software.

The system we presented in this paper is the third version, which was started 1992, when the author was working at Fujitsu Laboratories. The third version was developed on top of ordinary Unix workstations. SICStus prolog was used as an implementation programming language. Graphic Manager (GM) package of SICStus Prolog was used for graphics and interaction in the X Window environment. (GM is written using the InterViews system.)

8 Concluding remarks

We have developed visual programming system PP, which graphically assists the input of a GHC program and its execution. In PP, definition clauses of a GHC program, which are often difficult to understand, can be expressed as a figure along with its textual representation. Main features of PP can be summarized as follows.

- **Graphical input** We can input definition clauses of GHC program by figures. Textual form of the definition clause is automatically generated by the system.
- **Textual input** Clause definitions are also entered by text. In such case, graphical representation is automatically created. Textual and graphical forms are *causally connected* each other.
- **Editing graphical input** We can *graphically* edit the definition clause at any time.
- **Save and load** Textual and graphical forms can be saved to the store and can be loaded, if necessary.
- **Integrated environment** The system provides an *integrated* visual environment, which dynamically manages the definition clauses of GHC programs.
- **Graphical execution** We can execute goals graphically and interactively.

The preliminary evaluation of PP shows that the programming work using PP is very comfortable, comparing to the ordinary programming work. The operational feeling of PP is very similar to that of Macintosh. The user can *enjoy* the programming work. PP is really interactive and we can start using without sufficient preliminary knowledge of GHC and PP.

In the case of the visualization of conventional languages, we often need to insert primitives for dynamic visualization to the source code. However, PP is a fully automatic visual system and we do not need to insert such primitives. it is because the gap between the graphical representation of the *static* program and that of program execution is small in GHC.

Since the idea of expressing a program graphically is not a completely new idea, there are lots of related works as discussed in Section 1.2. Comparing to these works, the critical difference exists in the adoption of the *hybrid* approach. In PP, graphical representation can naturally matches to the textual representation. Therefore, we can utilize the latest implementation of the textual world. It is also possible to use PP for part of the total system, mixed with the text based system.

The further refinement up of the system and adding animating capability such as seen in Pictorial Janus [Kahn 92] and ToonTalk [Kahn 96] are the topics for our future works.

9 Acknowledgments

The author would like to express thanks to Yukiko Ohta, Simon Martin and Katsujiro Nakano for their help and programming support.

References

[Clark 86] K. Clark and S. Gregory: PARLOG: Parallel Programming in Logic, ACM Transactions on Programming Languages and Systems, Vol.8, No.1, 1986, pp.1-49.

[Cox 85] P.T. Cox and T. Pietrzykowski: Lograph: A Graphical Logic Programming Language, in Proceedings of IEEE COMPINT 85, Montreal, 1985, pp.145-151.

[Cox 86] B.J. Cox: *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[Davis 81] A.L. Davis and S.A. Lowder: A Sample Management Application Program in a Graphical Data-Driven Programming Language, in Digest of Papers Compcon Spring 81, February 1981, pp. 162-167.

[Dennis 74] J.B. Dennis: First Version of a Data Flow Procedure Language, Lecture Notes in Computer Science, Vol.19, 1974, pp.362-376.

[Futamura 81] Y. Futamura, T. Kawai, H. Horikoshi and M. Tsutsumi: Development of Computer Programs by PAD (Problem Analysis Diagram), in Proceedings of the Fifth International Software Engineering Conference, IEEE, New York, 1981, pp.325-332.

[Hirakawa 87] M. Hirakawa, S. Iwata, I. Tanimoto, M. Tanaka and T. Ichikawa: HI-VISUAL Iconic Programming, in Proceedings of the 1987 Workshop on Visual Languages, August 1987, pp. 305-314.

[Kahn 92] K.M. Kahn: Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, in Proceedings of the International Conference on Fifth Generation Computer Systems 1992, ICOT, June 1992, pp.943-950.

[Kahn 96] K.M. Kahn: ToonTalk – An Animated Programming Environment for Children, *Journal of Visual Languages & Computing*, Vol.7, No.2, June 1996, pp.197-217.

[Keller 81] R.M. Keller: Applications of Feedback in Functional Programming, in Proceedings of 1981 Conference on Functional Programming and Computer Architecture, ACM, 1981, pp.123-130.

[Kurita 84] T. Kurita and K. Tamura: Dialog.I: An Iconic Programming System Based on Logic Programming, Bulletin of the Electrotechnical Laboratory, Vol. 48, No.12, 1984, pp.966-975.

[Ringwood 89] G. Ringwood: Predicates and Pixels, *New Generation Computing*, Vol.7, No.1, 1989, pp.59-80.

[Shapiro 83] E. Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report, TR-003, 1983. (Also in *Concurrent Prolog: Collected Papers*, Vol.1, The MIT Press, 1987, pp.27-83.)

[Shapiro 89] E. Shapiro: The family of Concurrent Logic Programming Languages, *ACM Computing Surveys*, Vol.21. No.3, September 1989, pp.413-510.

[TGS 89] The Gunakara Sun Systems: *Prograph Tutorial*, The Gunakara Sun Systems, 1989.

[Ueda 85] K. Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985. (Also in *Concurrent Prolog: Collected Papers*, Vol.1, The MIT Press, 1987, pp.140-156.)