Visualization of the Type System for Functional Programs

Abstract

We introduce an type-directed user interaction. Function is the component of interaction. Functions are represented with abstract graphical type annotations. The system explicitly annotates every function with its type. The types of every function can be awared from the shape, colour and structure of the explicit notions. The types of arguments in higher-order functions can be identified before the arguments are supplied. Type information of function application is feedbacked immediately. In this way, the system prevents type errors. We intend to use the system in teaching of type checking.

1 Introduction

The type checking mechanism of strongly typed languages offers safety guarantees. Strongly typed languages tend to be implicitly typed, the type inference system generates types for every expression in a program. The programmer does not need to give explicit types except in cases of ambiguity. The type annotations made by a user are checked automatically against these inferred types so that type errors can be located at compile time.

Implicitly typed programs relieve the user from having to provide a type declaration for every expression in a program. But in implicitly typed programs, one major difficulty for novices is that they can ignore types and maynot understand the souce of type errors. SML is equipped with a type inference engine which automatically derives the type of the top-level function. Users need not explicitly state the type of each variable, but novices can ignore the types and be unaware of type inconsistencies which may arise. The error messages from the complier may not be understood.

In explicitly typed programs, the programmer explicitly declares the type of of every variable. In this way, a programmer can aware of the types while programming. But it is space and time consuming to annotate types. Sometimes the programmer needs to provide type information which the system already knows.

We combine the merits of implicitly and explicitly typed programs. We consider style of interaction directed by the type checker. The user inputs implicitly typed function expressions and the system automatically annotates the type of every argument. The user can understand the type annotation by its shape, colour and structure. The explicit notions of types make users aware of the types of every variable and avoids type errors.

2 Background

We annotate functions in functional language SML with icons. Icons are the components of interaction. A survey of visual functional programming has been made.

2.1 Visual Functional Programming

2.1.1 Visual Functional Languages

Several other systems are similar in using icons as the components to construct program.

Skin [12], assumes a simple object-oriented interface to the underlyinging application and components may flexibly adapt to changes in the application. Skin is a visual functional language for defining flexible user interface components. It uses an icon and connector model for program construction. Skin specifies layout of interface components, including interactive components such as buttons and edit boxes. Functions have input parameters at the top(solid line with pin underneath) and result at the bottom(solid line with two pins above and one below).

Visual Haskell [13] is a attempt to develop a visual equivalent for the Haskell functional language. It is intended both as a program visualization tool and as a programming language in its own right. The approach used a translation from Haskell into an intermediate form VHIF. with visual equivalents for each syntactic production of VHIF. The syntax of VHIF is close to Haskell's. It uses visual notations of the "box-and-arrow" style. Visual Haskell is closest to Show-and-Tell [6]. Show-and-Tell is also declarative in nature, and utilizes a polymorphic type system similar to Haskell's. Visual Haskell is incomplete, lacking syntax for modules, type declarations, type classes and instances, arrays, and user-defined operators.

VisaVis [9] is a functional visual language embedded in the Smalltalk system. The interaction strategy of substitution brings ease of construction to visual programs and integrates higher-order functions smoothly. The concept of supervised substitution prevents syntactical errors and realizes the key/keyhole metaphor.

CLOVER [19], is a new visual language which is pure functional and pure object-oriented, and is also completely type safe.

2.1.2 Interface Toolkits for Functional Programming

Fudgets [14] is primarily a Graphical User Interface Toolkit for the functional programming language Haskell and the X Windows system. Fudgets allow users to program Graphical User

Interfaces at a high level in the functional language Haskell. Many library component are easy to learn and use in the common case but still flexible and customisable.

David Gooding (since 1990), Jan Townsend-Addis and Tom Addis (since 1986) and Simon Gray (1994-96) at University of Bath, have worked on Clarity, CLARITY is a programming tool that enables users to generate programs though diagrams. CLARITY has been developed to make computer modeling and simulation more accessible to non-programmers. Faith is the underlying interpreted language and is based upon a typed functional language. The form of Faith is similar to Miranda or ML.

Concruuent Clean's Event I/O [15] provides programmers with a very high-level declarative specification method for writing complex interactive applications in a pure high-order functional language. The Concurrent Clean system is a programming environment for the functional language Concurrent Clean, developed at the University of Nijmegen. The system is one of the fastest implementations of functional languages available at the moment. Its I/O libraries make it possible to do modern, yet purely functional I/O (including windows, menus, dialoges etc.) in Concurrent Clean.

ClockWorks [16] is a programming environment supporting the visual programming of object-oriented software architectures. In developing ClockWorks, user interface evaluation techniques were used, including heuristic evaluation, cognitive walkthrough and user evaluation. The development of ClockWorks was based on a task analysis of ClockWorks programmers.

Clock [17] is a language which incorporates elements of visual, functional and object-oriented programming. Clock is supported by the visual ClockWorks environment, allowing the development, browsing and editing of object-oriented software

2.1.3 Other Related Research on Programming Environment

Harakd Thimbleby [26] gave some examples of browse binding in several programming languages and interactive systems. One of the languages is Pygmalion which is implemented in Smalltalk 72. Pygmalion could be said to be icon-oriented, and was programmed by direct manipulation actions on icons(programmed by example).

Luca Cardelli [20] discussed ideas of programming with boxes and other graphical data structures which fit for functional languages. In this paper, a graphical syntax for a small functional language was described. These ideas may be the basis for a programming system using a structure editor to manipulate two-dimensional representations of programs.

Frank G. Pagan [30] describes an programming environment for an FP-style language with a graphical syntax. The graphcial syntax is chart-oriented.

B. Backlund et al [31] describe several visual language-oriented design environments and conclude that the meta-techniques can be extended to languages having a graphical syntax.

U. Schreiweis [21] introduced an integrated Prolog Programming environment. The approach lays emphasis on the presentation of Prolog programs as graphical module interconnection graphs. The module consists of a set of predecates defining the functionality of the module and an interface. The interface of each module specifies the functionality it provides to other modules and the facilities it needs from other modules. The system provide animation of program execution. The implementation is based on hypertext system KnowS.

Charles Rich [22] presented the initial design and partial implementation of an interactive programming environment to be used by expert programmers. The hierarchical representation of the internal structure of programs is of major theoretical interest. He also describes a deductive system which can verify the correctness of the representations involving side effects on complex data with structure sharing.

Tinker [23] is an experimental interactive programming environment which integrates program testing with program design. The approaches are programs are tested with examples and constructing programs by menu selection. New procedures are created by working out the steps of the procedure in concrete situation. Tinker constructs a procedure for the general case from sample calculations. The user communcates with Tinker by selecting operations from menus on an interactive graphic display rather than by typing commands.

Alan Bundy [25] outlined an editor for recursive and non-recursive logic programs that could be translated into Prolog. The editor ensure syntactically correct by only editing operations that preserve correct syntax and to ensure the termination and well-definedness of recursive programs.

Mike Brayshaw [27] presented a graphical Prolog tracer. This extended the original "Transparent Prolog Machine" to provide a faithful representation of the inner workings of the Prolog interpreter and allow a visual overview of execution for rapidly homing in on buggy code.

C. Crimi et al. [32] presented a system. The semantics of the icons underlying the visual languages are readily apparent. A semantic similarity measure is used to develop an algorithm to infer the grammar of a visual language. The algorithm forms the major part of the inference module of the described system.

J Reekie [33] presented a novel approach based on integrating a block-diagram development system with a very-high-level programming language. Program transformation is used to control the parallelism of a program.

TableTalk [24] is a visual language intended as a database interface and employs visual metaphors in order to express the semantics of a database language. It uses a variety of visual representations to capture the semantics of a functional query language.

Margaret M. Burnett discussed how the goals of a type system must differ for visual languages form those of a type system for textual languages. A type system [10] developed under these goals for the visual language Forms/3 is presented.

Marian Petre [29] reported some research focusing on understanding how textual and visual representations for software differ in effectiveness. Novice and experts appear to see differently and they have different notational needs.

2.2 Explicit Types in Visual Languages

There are a few visual languages and programming environments that have an explicit notion of types. They can be divided into two categories: those whose type system are modeled after traditional procedural languages e.g. Fabrik [1] and DataVis [3] , and those whose type systems are based on the Hindley-Milner approach, e.g. Enhanced Show-and-Tell [6], Cube [7, 8] VisaVis [9] and an extension of Forms/3 [10].

In the first category, the languages supply a set of predefined types such as arrays, records, enumerations, bitmaps, etc., but do not allow the user to define new type. For example in DataVis, types are associated with colors. Only fixed and very small set of types can be describled by colors. And although DataVis uses the White color to denote unknown types, it does not deal well with polymorphic operations. Another drawbacks of the type inference system of Fabrik and DataVis is that it does not guarantee type safety: for example, they might try to reference a non-existing array element.

In the second category: Enhanced Show-and-Tell [6] was improved from Show and Tell [5] by adding a static polymorphic type system and higher-order functions using Hindley-Milner type inference. Cube [7, 8] is a successor of Show and Tell and Enhanced Show and Tell, which adds some key ideas from these two languages to visual logic programming. Cube's type system is based on Milner's system as well. VisaVis [9] is a visual language with a data flow based syntax and FFP based semantics, the higher-order version of FP. Its type inference system is based on Wand's type inference algorithm, which is a variation of the Hindley-Milner algorithm. Forms/3 [10] is a form-based visual language. Burnett's type inference system for Forms/3 is based on set constraints; the inference algorithm is also a variation of the Hindley-Milner algorithm. In Kelso's Visual Representation for Functional Programs [11] types are annotated in textual nodes. The languages in this category guarantee type safety, that is a well-typed program will not fail at run time due to a type error. To use those systems, users need to be trained. For example, in CUBE a type definition cube defines the list type. Figure 1. shows a type definition cube which consists of a grey transparent cube with an icon on its top. The color grey distinguishes types from values, which are green. The icon names the type constructor that is to be defined. The cube has a port on its top side, which represents the one formal parameter of the type constructor. The port carries an icon on its outside, which is used to distinguish it from other ports. Inside the cube are two grey transparent boxes, which represent the two constructors of the list type. Each box carries an icon on its top, identifying the constructor.

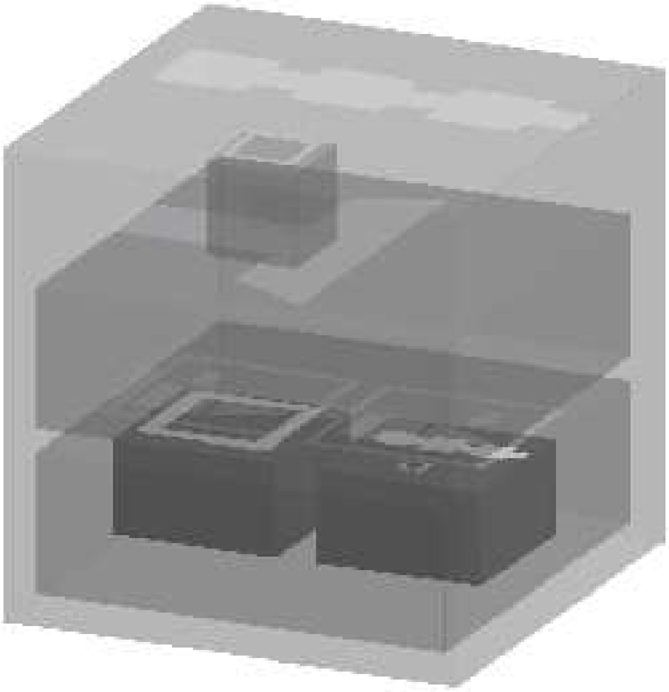

Figure 1: The definition of list type.

3 The Graphical Representation

We introduce a novel graphical type annotation also based on the Hindley-Milner type inference algorithm. In our system, a user can always recognize the types of an expression by the shapes of graphical type annotations or by inspection of the type context of the expression. The polymorphic type expression of higher order functions such as $('d \rightarrow' e) \rightarrow ('f \rightarrow' d) \rightarrow' f \rightarrow' e$ of function compose seems complicated to a novice and complicates the use of the function. We explain such complex type expressions in an easy understanding way.

In our system, when a user applies a function, they do not have to know the full syntax of the application, i.e. what they need to do is just to connect the icons. The connections are type-directed. This can prevent type errors.

3.1 Combine Textual and Visual Programming

Pure functional languages and their relatives, such as LISP and SML, often have simple grammars. However, the difficulties for novices are the programming concepts embodied. It is not difficult for a user to write single function in text. But it is not easy, for example, to handle the applications of higher-order functions etc.

We combine textual and visual programming in the system to avoid the complexity of visual programming with large numbers of different icons and the difficulties of constructing complex expression-graphs. In pure visual programming the user still needs to know a specified syntax and the visual system behaves like a drawing program. Green and Petrie [34] summarising other empirical studies, suggest that in general the structure of a program is harder to follow in a graphical form than in text. We suppose textual programming is more suitable for constructing each individual function. For instance, it is difficult to denote numerical parameters graphically. In this system, functions are input in text as in textual languages. Only applications of higher-order functions are visualized.

The system can be divided into two levels, the first is textual function definition, to supply the functions, and check the syntax of input text. The second level is a pure visual programming environment, to interactively construct function applications. Thus, the user has the freedom to construct function and the system can direct the user to construct well-typed structured function composition.

3.2 Symbolism

In our system, we also use colours to present basic data type but constant structure shadings to represent structual types.

Any integer type data is abstracted with a box labeled with "int" attached with a red filled box, Figure 2.

Figure 2: The integer type.

Any real type data is abstracted with a box labeled with "real" attached to yellow filled box, Figure 3.

Any string type data is abstracted with the box labeled with "..." attached to a pink filled box, Figure 4.

We represent bool with a magenta filled box, shown as Figure 5.

Figure 3: The real type.

Figure 4: The string type.

Figure 5: The bool type.

We present a list with "::" as its name, shown as Figure 7. Here we bracket the list element type as in Figure 6.

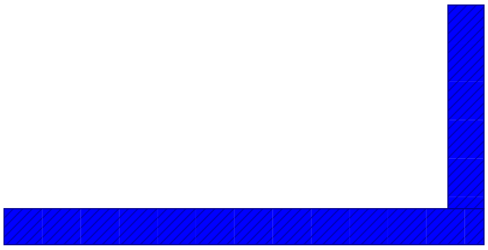

Figure 6: The symbol of list.

For example, an integer list is presented as Figure 7.

Figure 7: The integer list.

A tuple is shown with "(...)" as its name with the types of its elements attached together: Figure 8.

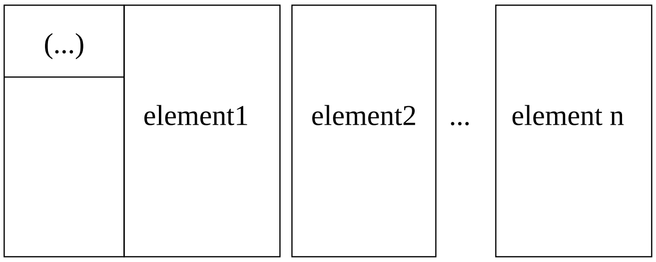

Figure 8: The tuple.

For example, a tuple of integer and real is annotated as Figure 9.

Figure 9: The tuple of integer and real.

Composite types are represented by listing the representation of their component types: For example, ([1,2,3],[1.1,2.2,3.3]) is annotated as Figure 10.

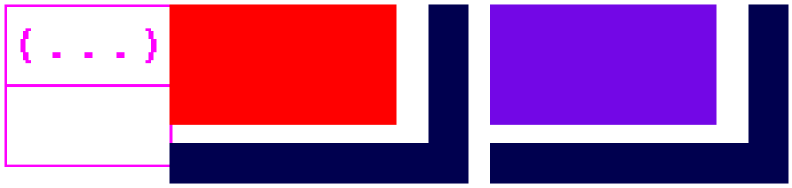

Figure 10: The tuple of list ([1,2,3],[1.1,2.2,3.3]) : int list*real list.

[[1,2],[3],[4,5]] is annotated as Figure 11.

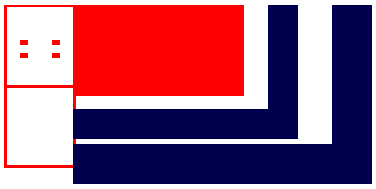

Figure 11: The list of integer list [[1,2],[3],[4,5]] :(int list)list.

[(1.1, [1,2,3])] is annotated as Figure 12.

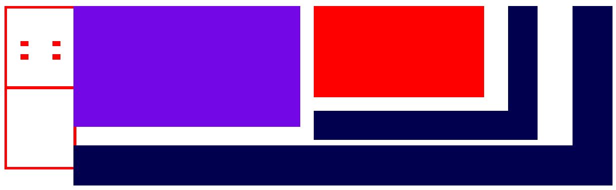

Figure 12: The list of tuple [(1.1, [1,2,3])] :(real*int list) list.

In this way, we can represent complex type simply, hierarchically. The outline of the graph describes the top-level of the type, the inner part of the graph describes the elements of the type structure.

3.3 Functions with Graphical Representation

The functions are represented as function icons: boxes labeled with the function's name attached by the box with the graphical annotations of the types of its arguments labeled on the upper left and the type of result labeled on the bottom right. E.g. function sq of type integer to integer is represented as:

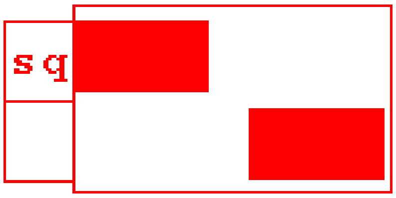

Figure 13: The function sq.

3.4 Polymorphic Types

We present polymorphical type with the textual name in a white box directly. For example, the expression of higher-order function compose: fun compose x y=x o y with type $('d \rightarrow' e) \rightarrow ('f \rightarrow' d) \rightarrow' f \rightarrow' e$ is presented as:

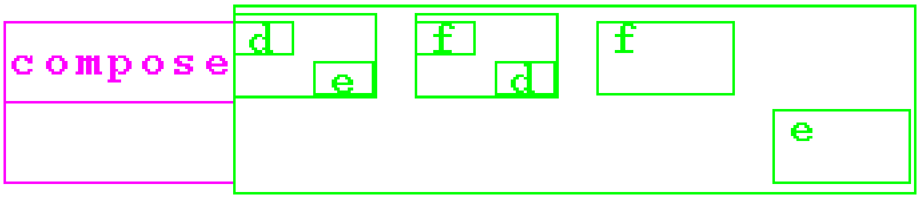

Figure 14: The function compose.

The number of arguments and the order of arguments are kept their original textual forms, thus it is nature for user. For example, the function map with type:$('c \rightarrow' d) \rightarrow' c$ *list* $\rightarrow' d$ *list*: fun map f (h::t)=f h::(map f t)|map _ []=[] is represented as: It means that

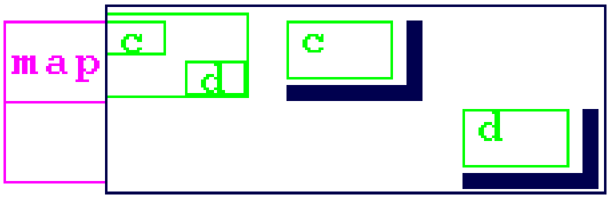

Figure 15: A polymorphic type of function map.

map takes a function and a list as its arguments.

4 Expression graph construction

To demonstrate of the effectness of the annotations, we present some with interactions of the system.

4.1 Type Directed Connection

One of the main principles of the system is ease of use. The manipulation of icons is simple, is to connect a selected icon with other icons. The connections are directed by the type checker. When we select an icon, the system highlights all the possible connections automatically.

There is a simple text editor for the user to define functions or input data in text. The system checks the syntax of input text. After a function definition, we can apply the function.

When an icon is selected, the icon is highlighted by a red colour box below its name. Shown as in Figure 16, we select the integer list.

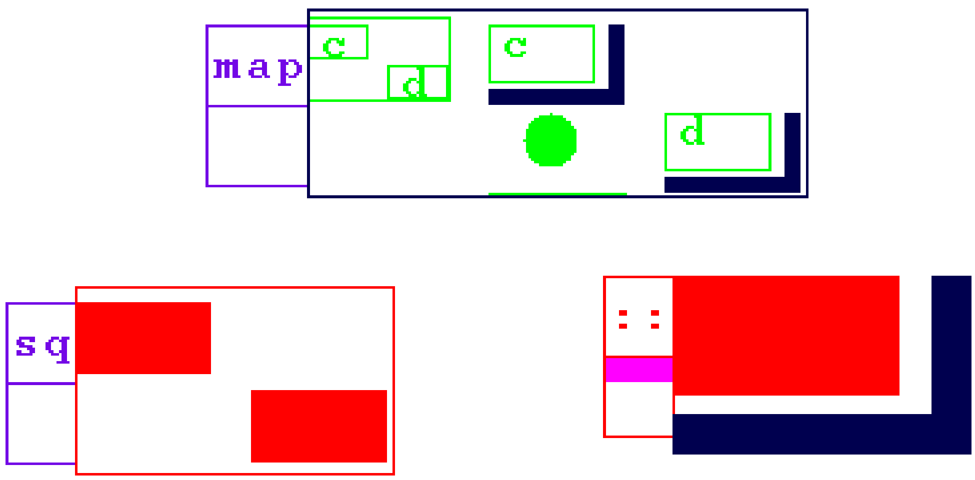

Figure 16: Select the int list, it can be the second argument of map.

To apply the selected icon as arguments, click on the "green light" of arguments on other icons. The system connects the icon to the argument by a line with an arrow, the type of the original icon flows to the argument, shown in Figure 17. If there is no "green light" displayed below a argument, it means the type is not appropriate, and the selected icon can not be used as the specified argument of the function. The system preserves the syntax of function application. For example, the user is not allowing to connect a cycle of the selected icon on itself.

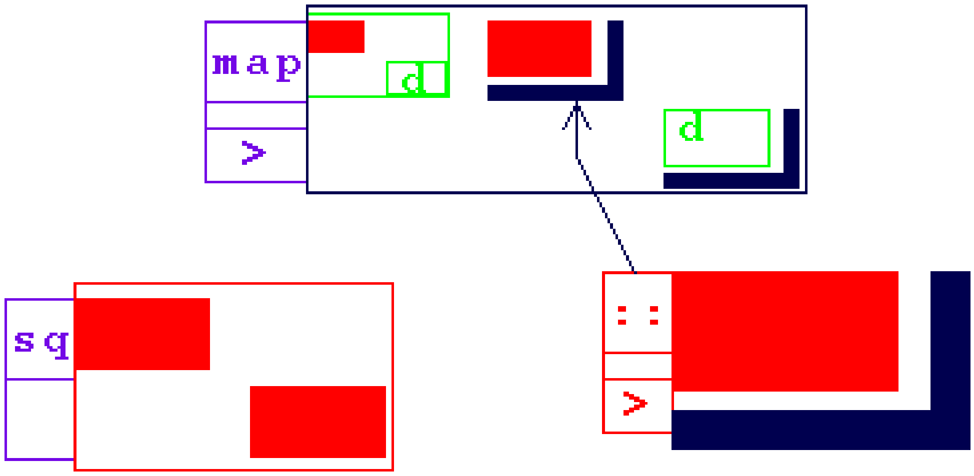

Figure 17: Function map take a real list as the second argument.

The system does type checking at the expression-construction time by not allowing a user to connect icons with arguments of other icons that are type inappropriate. If the user tries to connect an icon to an argument of another icon which is type-mismatched, there is no line and arrow to connect the two parts. The system refuses the connection.

Actually, user does not need repeatedly trying to link an icon to various icons to seek which icon's type matches the icon's type. Whenever an icon is selected, the system highlights which icons the user is allowed to interact with in order to achieve a type correct function applications. The type checker identifies all icons that can be used as the first argument of the selected function by a green arc below the name of the type appropriate icons, and identifies which arguments of all icons for which the selected icon can be used as the arguments by a "green light" under the annotation of the argument. In Figure 18, when we select icon of sq it indicates the icon of type int can be the first argument of function sq, and sq can be supplied to the three arguments of compose and the integer can be the first argument of sq.

When an icon takes another icon as its argument, the types of its other arguments and the type of its result are changed accordingly to reflect new type of the connected expression due to type unification. Also, the type of the icon taken as arguments also can be changed due to type unification. In Figure 19, for example, when we connect the third argument of function compose with an integer, the system displays the second argument should be a function of type: $int \rightarrow' a$. When we connect the first argument of function compose with function sq (Figure 20): The system displays that the second should be a function

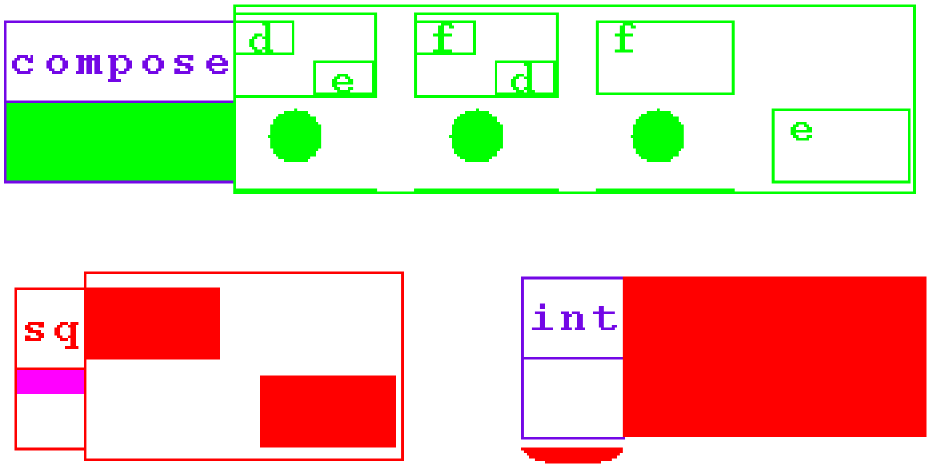

Figure 18: Selecting sq, integer can be the first argument of sq, sq can be used in any of compose's three arguments.

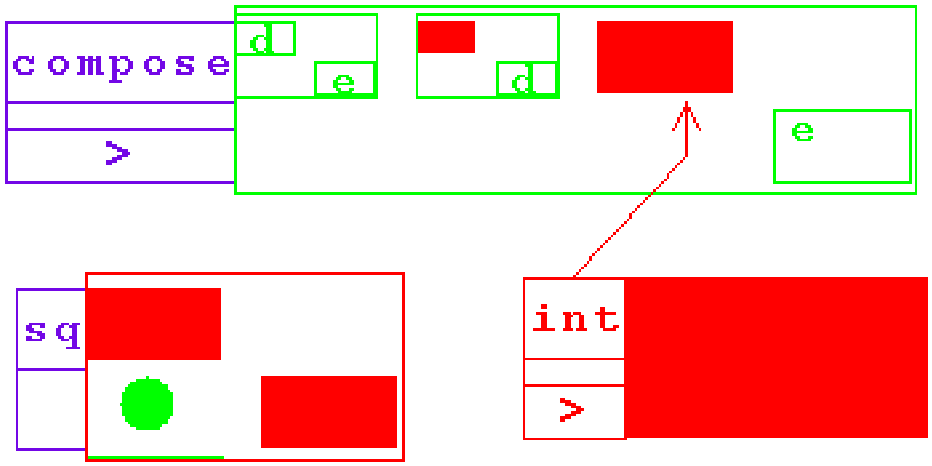

Figure 19: Select the integer and connect to the compose's third argument.

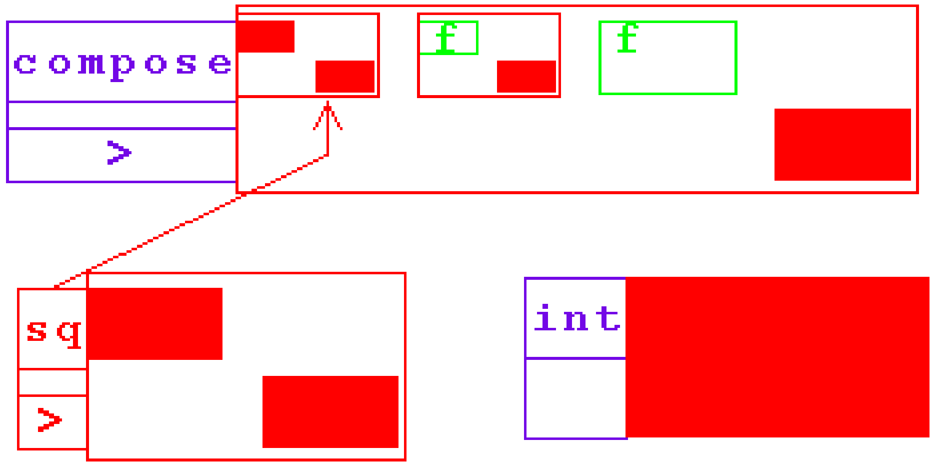

Figure 20: Function compose take a int list as its second argument.

of type: $'c \rightarrow int$.

The system conveys the type information of the arguments to user by its colour, shape, and structure. For example: In Figure 20 the first argument of function sq should be an integer. In Figure 16, the first argument of function map should be a function. The second argument of function map should be a list. In this way, the user can aware of the type of every argument before supplying the argument.

We support abstraction by collapsing the connection graph into a new icon with parameters types and result type. The new icon is labelled with "<" below its name. This icon can be used for further applications.

To collapse an connection graph, click on the ">" which is below the name of any icon in the connection by the third button of mouse.

Figure 22 shows the result of collapsing Figure 21. When an expression is collapsed into one icon, the names of the supplied arguments are displayed in the original parameters order with an arcs on their top, "_"s annotate the positions of arguments that have not yet been provided. For example, in Figure 22 the first argument and the third argument have not been supplied.

An icon with "_"s as its argument names can be provided with arguments. When it is provided with arguments, the "_"s will be replaced by the names of supplied arguments.

We can supply the arguments of higher-order function with more than one arguments in arbitrary order. E.g. fun add x y=x+y is shown as Figure 23.

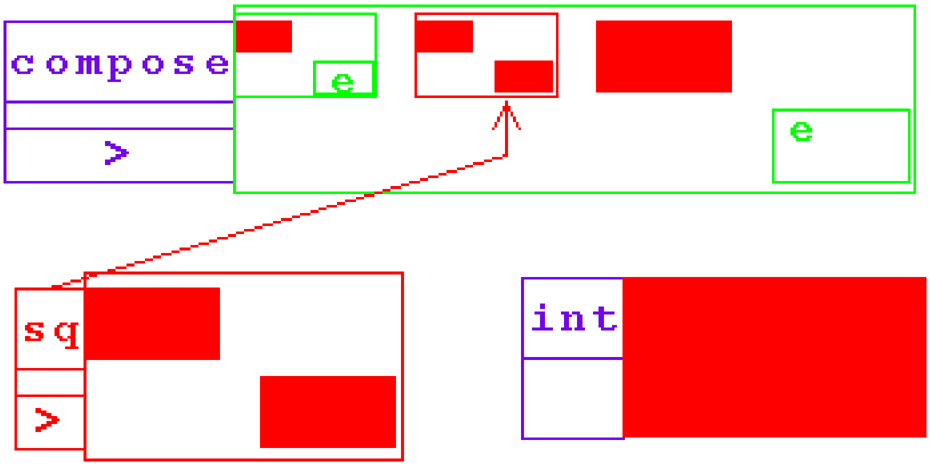

Figure 21: Function compose takes sq as its second argument.

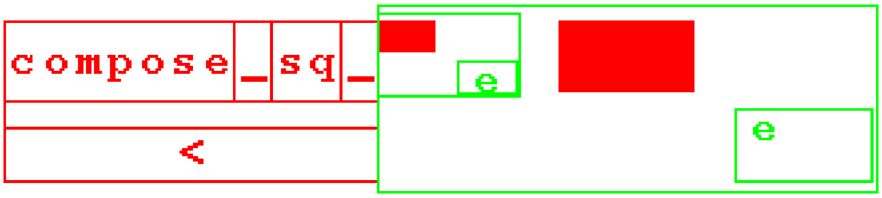

Figure 22: Function compose takes sq as its second argument, the first and the third arguments have not been supplied yet.

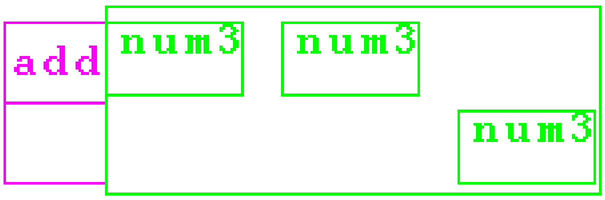

Figure 23: The function add.

There are two arguments in the function add. If we supply the first argument (x) with an integer 12, the type of first argument is instanced as integer and the type of second argument is instanced as integer accordingly: Figure 24.

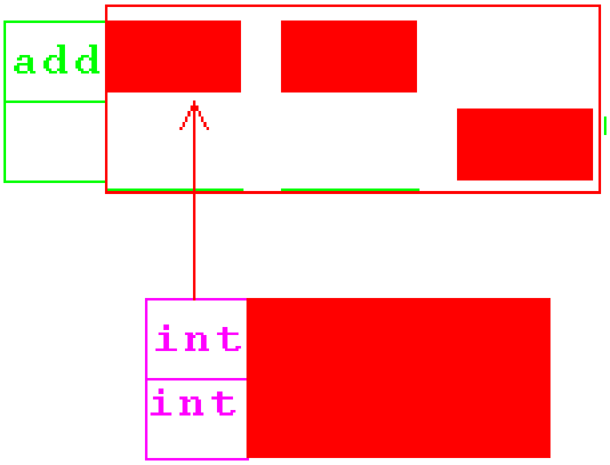

Figure 24: The function add's first argument is 12.

The result of this application is another function of type integer to integer, Figure 25:

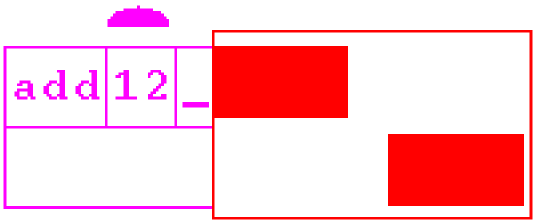

Figure 25: The function add's first argument is 12, return another function.

In textual function applications, we must give the first argument of the higher-order function first, then the applied function return another function. Visual programming inherits two dimensions, we can supply the arguments of higher-order function in arbitrary order. In this example, we can supply the second argument (y) first, as Figure 26 The result is a function of type integer to integer, Figure 27:

In the another case, higher-order function takes other functions as arguments. If we supply first argument of function map with function sq of type $int \rightarrow int$, Figure 33: The second argument should be an integer list, Figure 29:

If we supply the second argument with a real list, Figure 30:

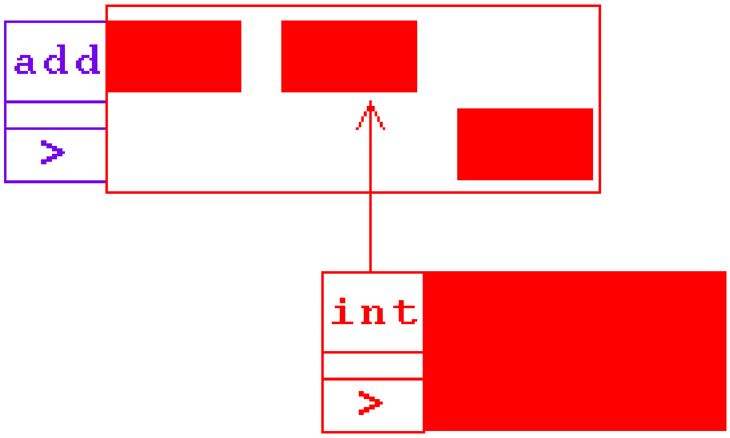

Figure 26: Input function add' second argument first.

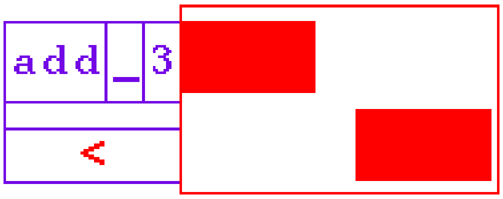

Figure 27: Input function add' second argument first, return another function.

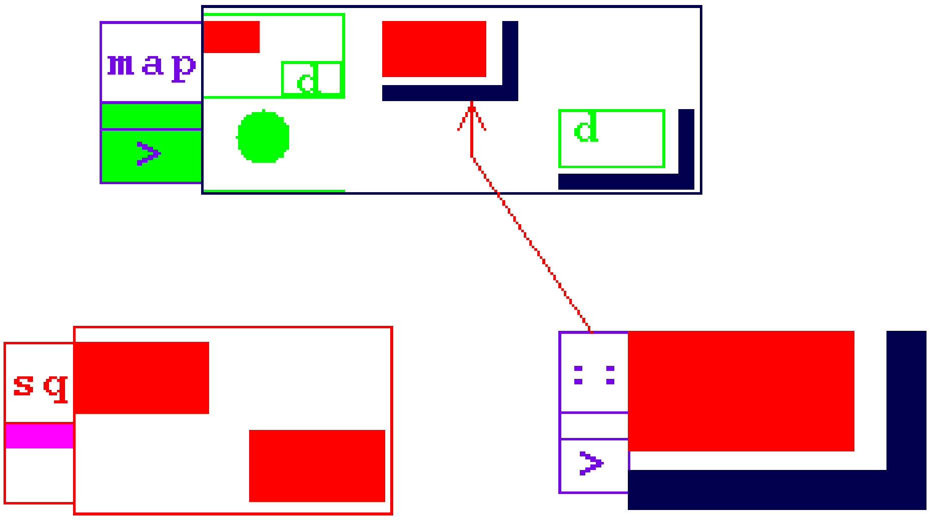

Figure 28: Function map takes function sq as first argument.

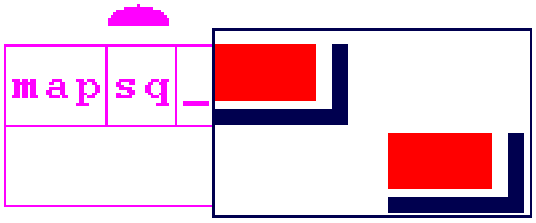

Figure 29: Function map takes function sq as first argument, return another function.

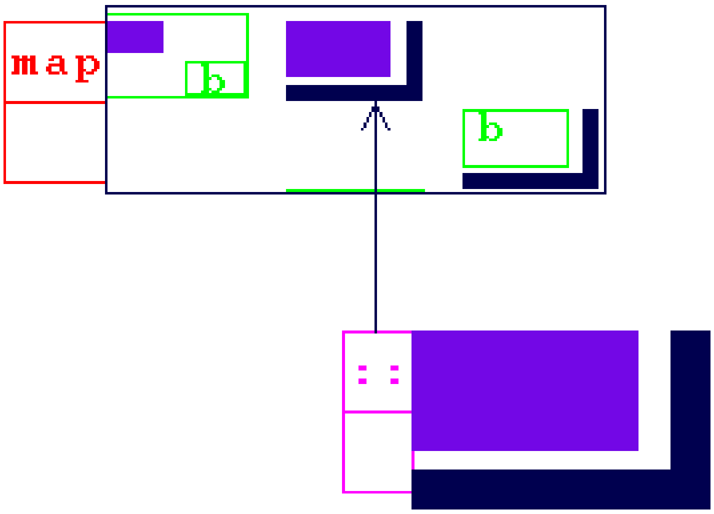

Figure 30: Function map takes a real list as the second argument.

The first argument should be a function of type $real \rightarrow' b$, the result be of type 'b, Figure 31:

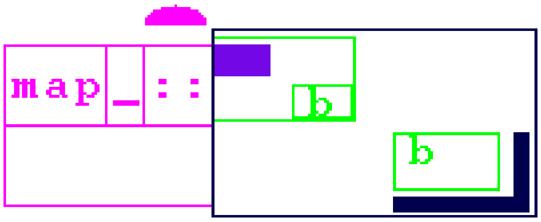

Figure 31: Function map takes a real list as the second argument, returns another function.

4.2 Edit

User can edit simple programs by defining new functions, selecting predefined functions, deleting functions, applying functions and de-applying functions.

When click on the name of an icon, the original connections of the icon will be disconnected, and its original type be restored automatically. The types of other icons that connected with the selected icon will be restored to their types which before the connection. Partial reconnection is possible at any time. For example, in Figure 32, when we select icon of function map, it disconnects with all its argument. In Figure 33, when we

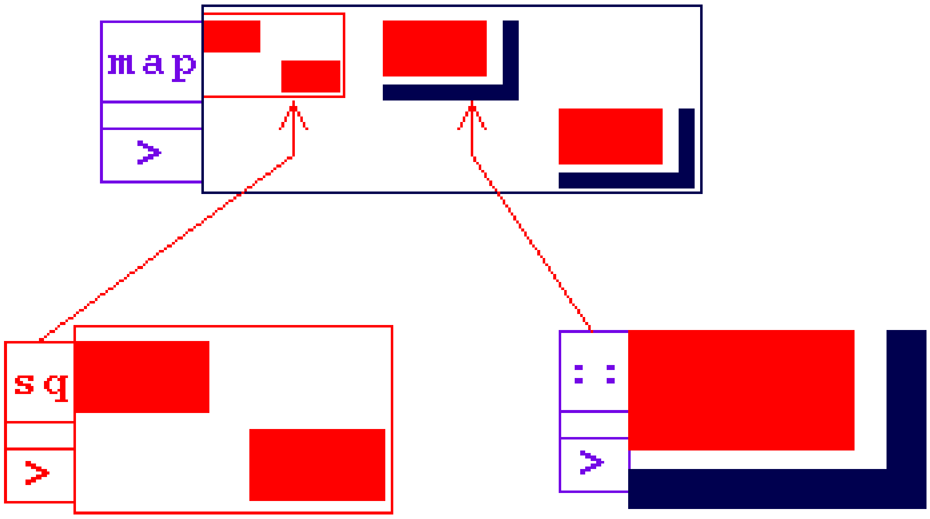

Figure 32: Connection of Function Application.

select icon of sq, it disconnects with icon of map.

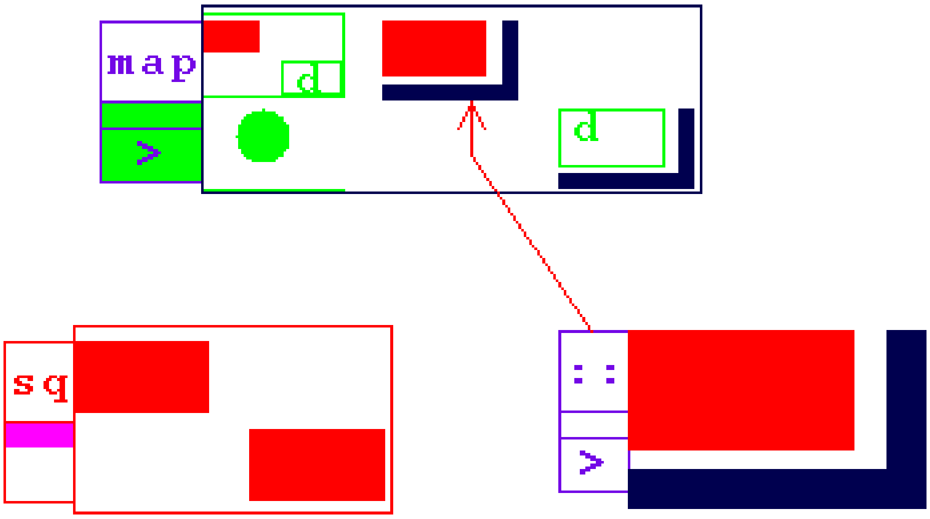

Figure 33: Select icon of sq, it disconnects with icon of map.

Users can modify the layout by dragging the icons. To drag an icon, click on any part of the icon by the first button of mouse, when the name of the icon becomes red drag the mouse with first button down. Shown in Figure 34.

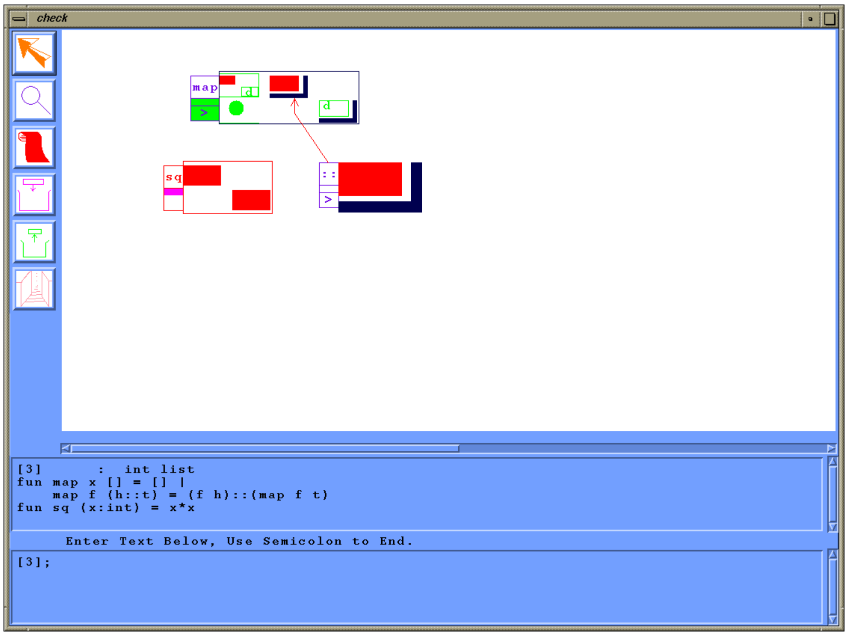

Figure 34: The snapshot of window .

To amplify the graph of type representation, chose the second the button on the left-hand side of window.

User can check the text of functions by selecting the icons and chose the third button on the left-hand side of window.

To delete an icon, select the icon and chose the fourth button on the left-hand side of window.

To recoverd deleted icons, chose the fourth button on the left-hand side of window.

To exit the system, chose the last button on the left-hand side of window.

5 Conclusion and Future Work

We have presented a novel graphical type visualization system based on Hindley-Milner type inference approach. User can determine the type of a function and the types of the arguments by the colour, shape, structure of the annotations.

We combine textual and visual programming in the system. Functions are defined in text. There are lexical analyser and parser to handle the text. Functions are the components of visual operations. The system incorporates a type checker. It annotates each function with its type. Users receive immediate feedback of type informations. And all the operations of function applications are type-directed. The system informs user the needed type of arguments before the arguments are supplied. This erases the type-related errors.

We intend to use the system in the teaching of type checking.

We have represented the types in detail. When the types are very complex, we represent the graph recursively and we provide a function to scale the graph. In further work, we can abstract the complex types with their hierarchical structures and their details are displayed only when needed.

For visualization of the SML language, we expect to implement the animation of the evaluation of programs in future work.

References

[1] Dan Ingalls, Scott Wallace, YU-Ying Chow, Frank Ludolph, and Ken Doyle. *Fabrik: A Visual Programming Environment.* OOPSLA'88, San Diego, September 1988. Apperared as ACM Sigplan Notices, 23(11):176 - 190, November 1988.

[2] Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. *The Fabrik Programming Environment.* IEEE Workshop on Visual Languages, pages 222 - 230, Pittsburgh, PA, 1988.

[3] Daniel D.Hils. *DataVis: A Visual Programming Language for Scientific Visualization.* In Proc. 1991 ACM Computer Scinece Conference, pages 439-448, San Antonio, TX, March 5-7 1991.

[4] Daniel D.Hils. *A Visual Programming Language for Visualization of scientific Data.* Ph.D. Thesis. Technical Report No. UIUCDCS-R-93-1809, Dept. of Computer Scinece, Univ. of Illionis at Urbana-Champaign, June 1993.

[5] Nan C. Shu, IBM Los Angeles Scientific Center. *Visual Programming.* Pages 222-237, VAN NOSTRAND REINHOLD COMPANY, New York, 1988.

[6] Marc A. Najork and Eric J. Colin. *Enhancing Show-and-Tell with a polymorphic type system and higher-order functions.* In IEEE Workshop on Visual Languages, pages 215-220, Skokie, IL, October 1990.

[7] Marc A. Najork and Simon M. Kaplan. *The Cube Language.* In IEEE Workshop on Visual Lnaguages, pages 218 - 224, Kobe, Japan, 1991.

[8] Marc A. Najork and simon M. Kaplan. *A Prototype Implementation of the Cube Language.* In IEEE Workshop on Visual Languages, pages 270-272, Seattle, WA, 1992.

[9] Jorg Poswig, Klaus Teves, Guido Vrankar, and Claudio Noraga. *VisaVis - A Higher-order Functional Visual Programming Language.* Journal of Visual Languages and Computing, pages 83-111, (1994).

[10] Magaret M. Burnett. *Type and Type Inference in a Visual Programming Language.* In IEEE Symposium on Visual Languages, pages 238 - 243, Bergen, Norway, 1993.

[11] Joel Kelso. *A Visual Representation for Functional Programs.* Technical Report, Austria, December 1994.

[12] John Hosking, Stephen Fenwick, Rick Mugridge and John Grundy *Cover Your Self With Skin.* Technical Report No. 94-9 March,1994

[13] H. John Reekie, School of Electrical Engineering, *Visual Haskell: A First Attempt* REsearch Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology Sydney, Australia

[14] Magnus Carlsson, Thomas Hallgren, Chalemrs University of Technology magnus, hallgrn@cs.chalmers.se *Fudgets- A Graphical User Interface in a Lazy Functional Language.* Functional Programming and Computer Architecture, March 93.

[15] Peter Achten, Tinus Plasmeijer, University of Nijmegen, The Netherlands. *The Beauty and the Beast.*

[16] T.C. Nicholas Graham, Catherine A. Morton and Tore Urnes. *ClockWorks: Visual Programming of Component-Based Software Architectures.* Journal of Visual Languages and Computing, Academic Press, pp. 175-196, July 1996.

[17] T.C. Nicholas Graham. *The Clock Language: Preliminary Reference Manual.* November 1996.

[18] Sigbjorn Finne and Simon Peyton Jones, Functional Programming Group. at the Department of Computing Science, Glasgow University. *Composing Haggis.* Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics, Maastricht, September, 1995, Springer Verlag.

[19] Braine, Lee, Chris Clack, Dept. of Computer Science, Univ. College London. *Object-Flow.* In 1997 IEEE Symposium on Visual Languages, Capri, Italy, Sept. 1997.

[20] Luca Cardelli, *Two-Dimensional Syntax for Functional Language.* Dept. of Computer Science, Univ. of Edinburgh. European Conference on Integrated Computer Systems, 1982.

[21] U. Schreiweis, A.Keune, H. Langendorfer, Technische Universitat Braunschweig, *An Integrated Prolog Programming Environment.* ACM SIGPLAN Notices, Volume 28, No. 4 April 1993.

[22] Charles Rich, Howard E. Shrobe, *Initial Report on a Lisp Programmer's Apprentice.* IEEE Transactions on Software Engineering, Vol. SE-4, NO.6, November 1978.

[23] Henry Lieberman and Carl Hewitt, Artificial Intelligence Laboratory, *A session With TINKER: Interleaving Program Design.* Conference Record of the 1980 LISP Conference.

[24] Richard G. Epsteim, Dept. of Mathematics and Computer Science, West Chester, Pennsylvania 19383, U.S.A. *The TableTalk Query Language.* Journal of Visual Languages and Computing (1991) 2, pp115-141.

[25] Alan Bundy, *Proposal for a Recursive Techniques Editor for Prolog.*

[26] Harakd Thimbleby, Stirling Univ. *Browse Binding and Viewscopes.*

[27] Mike Brayshaw, Marc Eisenstadt, Human Cognition Research Laboratory, The Open University, Milton Keynes, UK. *A practical graphical tracer for Prolog.* Int. J. Man-Machine Studies (1991)35, 597-631.

[28] Henrik Nilsson and Peter Fritzson, Programming Environment Laboratory, Dept. of Computer and Information Science, Linkoping Univ., Sweden, *Algorithmic Debugging for Lazy functional Language.* PLILP'92 Symposium in Leuven, Belgium.

[29] Marian Petre, *Why looking isn't Always Seeing Readership Skills and Graphical Programming.* Communication of the ACM, June 1995/Vol.38, No. 6

[30] Frank G. Pagan, Dept. of Computer Science, California State Univ. *A Graphical FP Language.* SIGPLAN NOTICES V22 3, March 1987.

[31] B. Backlund et al., *Visual Language-Oriented Design Environments.* Journal of Visual Languages and Computing(1990) 1.

[32] C. Crimi et al., *Grammatical Inference Algorithms for the Generation of Visual Languages.* Journal of Visual Languages and Computing (1990) 1, pp.355-368.

[33] J Reekie, School of Electical Engineering, Univ. of Technology, Syndney, Australia. *Integrating Block-Diagram and Textual Programming for Parallel DSP.* Gold Coast, Australia, August 16-21, 1992.

[34] T.R.G. Green and M. Petre, *When visual programs are harder to read than text.* in Human-computer interaction: tasks and organisation, Proceedings of ECCE-6, CUD:Rome, ed. G.C. van de Veer, M.J.Tauber, S. Bagnarola and M. Antavoltis, (1992).