A Visualisation of Polymorphic Type Checking

YANG JUNG and GREG MICHAELSON

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, EH14 4AS, UK
(*e-mail:* {ceejy1,greg}@cee.hw.ac.uk)
(*URL:* http://www.cee.hw.ac.uk/Staff/greg.html)

Abstract

The understanding of polymorphic typechecking and type errors is poorly supported by contemporary functional language implementations. Here, a novel visualisation of functions and their types is presented based on the generation of function specific icons with graphical type representations which change dynamically as functions are applied. This visualisation has been implemented for a Standard ML subset within a graphical environment in which function combinations are constrained by type matching.

1 Introduction

The benefits of static parametric polymorphic typechecking based on the Hindley-Milner scheme, in minimising semantic errors while optimising component generality, are well known. Indeed, a central strength of modern functional languages lies in the ease with which they incorporate polymorphic typechecking.

However, systems based on the W algorithm and its variants tend to offer poor support for understanding type checking and type errors. While this efficient algorithm is highly amenable to implementation, it is not immediately apparent that it corresponds well to how people understand the types of programs. Certainly, novices express difficulty with interpreting type error messages which relate to the places where type checking failed rather than the design mistakes and misconceptions that underly the failure. They also have difficulty with understanding the unification of polymorphic types, especially where both types contain type variables.

There have been a number of attempts to generate more useful type error information either directly from the W algorithm or through extensions and modifications to it but ultimately these all suffer from the same explanatory limitations as the algorithm itself. In contrast, a number of visual programming systems have used polymorphic type checking to prevent badly typed construct combinations but these tend to lack clear explanations of their type checking regimens.

Here we present a visualisation of functions in which domain and range types are identified through graphical representations. Function combinations are constrained by polymorphic type checking and type match failures are identified by clashes between type representations. For well typed combinations, the type representations are changed, in particular to reflect bindings of polymorphic type variables.

The following sections discuss novice typechecking problems, textual support for polymorphic type checking in contemporary functional language implementations and the use of polymorphic type checking in visual programming systems. Next, our visualisation of types is discussed and its evaluation with novice functional programmers is considered. The visualisation has been incorporated in a graphical system for an SML subset: its design and use are presented. Finally, further work is suggested.

2 Novice misconceptions with types

In 7 years of teaching Standard ML to undergraduates we have observed a number of misconceptions about types. Some seem to be generic; others are specific to polymorphic type systems.

A first source of difficulty is with the distinction between different numeric types. In some languages used at school level, like BASIC and COMAL [1], or as first undergraduate languages like C, C++ and Java, the distinction between the integer and real types is elided. Mixed mode arithmetic and assignment is permitted with invisibly overloaded operators and automatic type coercions. This leads students to view integers and reals just as two different ways of writing numbers rather than as distinct types, and to a lack of understanding of numeric type inconsistencies. Ironically, this was not a problem in the bad old days when assembly and machine code were taught early in the syllabus as the concrete representations and operations made the differences painfully explicit.

There seem to be distinct problems with boolean types(Michaelson, 1996), in particular with the idea of `true` and `false` as values rather than as control indicators for conditional and repetitive constructs. Students often compare the result of a boolean expression with a boolean value, or return a boolean value from a conditional expression. This is made worse by the absence of a boolean type in some languages, for example LISP and C, where the presence or absence of particular values from some other type is used instead. Thus, we have noticed that students have difficulty in identifying that an argument or result has a boolean type.

Students also seem to have conceptual problems with tuples. This may be exacerbated by their use as a mechanism for uncurried parameter passing: students may already be familiar with a similar bracketed notation for formal parameters in imperative languages. Nested types for nested tuples and lists of tuples are a further source of difficulty, as are those for nested lists.

However, the main source of student difficulty lies in the understanding of type abstraction through type variables, which are not presented in early teaching of imperative languages. First of all, the very concept of a type as a value is foreign. Secondly, in block structured languages, each variable is distinct, even where they have the same identifier. In contrast, in a polymorphic type expression every occurrence of the same type variable must be instantiated to the same type; that is type

[1] COMAL is Danish imperative language with similar functionality to but more structured syntax than BASIC. It has been widely used in Scottish secondary schools.

variables are implicitly universally quantified. Furthermore, distinct type variables need not but may be instantiated to the same type. Finally, students have difficulty in accepting that a polymorphic type has been deduced when their intention in solving a particular problem has been to use specific types.

Our observations are supported by Whittle's study [2] of 1st year student errors in SML programs. He found that type errors were almost as common as syntax errors and that on average students had 2.3 type error messages for each command; a much higher rate than for syntax errors. Whittle notes that some type errors are caused by bad syntax but that students found type errors much harder to correct.

3 Textual support for polymorphic type checking

Milner's W algorithm (Milner, 1978) assigns a type, usually containing type variables, to each abstract syntax construct in a program component. It then seeks to derive an overall type for the component by unifying types for sub-constructs which should have the same type, to find consistent substitutions with which to instantiate the type variables.

While the W algorithm is excellent for type checking it seems to be a poor basis for identifying the causes, as opposed to the sites, of type errors. The W algorithm carries out a top-down, depth first check of a construct and finds errors when types which should be the same cannot be unified. Such failures reflect errors at points of construct use, for example function composition: however, such errors usually originate at points of construct definition, for example function declaration. Furthermore, errors are often identified in the sub-structure of a construct, which again makes it hard to locate their actual source, for example in the use of an overloaded operator with untyped bound variable operands.

Error intelligibility may be compromised by a lack of programmer understanding of the correspondences between an original program and the form in which it is checked. For example, in the SML Definition (Milner *et al.*, 1997), a distinction is made between the bare language and derived forms. Derived forms are used by programmers and are subsequently transformed to the bare language for implementation. For example, function definitions with multi-variable pattern matching are converted to nameless functions of single variables, whose bodies are tuple matching case expressions. Here, an error originating from the programmer's point of view in the head of a function definition appears to be identified in the body. For example, infix binary operators are often converted to prefix functions acting on two element tuples. Finally, errors may be reported in terms of the type variables introduced by the W algorithm, which need have no obvious relation to the original program.

Surprisingly little research has been conducted into providing better support for polymorphic type errors: most has been oriented towards refining the presentation of W algorithm checking sequences. Soosaipillai (1990) proposed that type errors might be elucidated by traversing a tree of type checking decisions made by the W

[2] Personal communication, Blue Note 1132, Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, Dec. 1996.

algorithm. Her incomplete system for an SML fragment enabled such exploration by offering menus of type decisions at each level for further investigation. Duggan and Bent (1996) modified the unification algorithm used in Hindley-Milner systems to try and isolate the sequence of decisions leading to particular types being ascribed to variables. They applied this approach to a mini-functional language. Beaven and Stansifer(1993) also discuss a system for SML. Parse trees are decorated during type checking and traversed to generate depth first explanations. Wand(1986) also records type checking decisions, focusing on function applications as the loci of type errors. He notes that type errors arise when constructs with inconsistent types are combined in an application, and seeks to find the source by further analysing the type derivations for those constructs. Rideau and Thery(1997) provide support for type explanation in a CAML implementation using techniques derived from Wand and from Bent and Duggan. Rittri(1993) suggests a modification to Wand's approach where the user interactively identifies those parts of a type which are deemed surprising and in need of further elucidation. Johnson and Walz(1986) discuss an approach to incremental type inference used in the MOE language based editor for an ML variant with an extended type system. Here, sites of type error detection are associated back to likely error sources. Where there are several candidate sources then the system tries to identify the most likely one. Finally, Turner(1990) describes a system which can identify type errors in SML programs in the derived form.

In all of these approaches, explanations are structured variants of the W algorithm type checking sequence. As with type checking itself, explanations grow with the size of the checked component and the complexity of the underlying type. There is a danger of getting lost in a morass of detail which is increasingly indirectly related to the original program component or the cause of the type error. Only Wand, and Johnson and Walz directly address the location of the source of errors.

4 Type checking in visual programming systems

Our motivation for the work discussed here was to explore ways of enhancing novice understanding of parameterised polymorphic type checking. Rather then revisiting text based explanation of polymorphic type checking, we wished to draw on graphical and visualisation techniques, in particular from visual programming systems. In such systems, all language constructs are represented as diagrams, typically as icons or labeled boxes, and programs are built by juxtaposing or interconnecting these representations in two dimensions.

In principle, a visual programming system may be syntax directed, where construct connection is only allowed if syntax correctness is preserved. This approach is taken in Cardelli's seminal proposal for visual functional programming(Cardelli, 1982). However, syntax directed programming imposes an un-natural discipline on programmers. It is more common either to allow free interconnection, through what is essentially ad-hoc polymorphism with en-masse consistency checks on a final program, or to use a type system to dynamically constrain interconnection through incremental type checking.

A number of visual functional programming systems have been constructed, incorporating parameterised polymorphism to constrain interconnection. For example, Poswig et al's VisaVis(1994) is a general purpose language where interconnection is based on what they term "less ad-hoc polymorphism". However, types do not appear in program graphs. Braine and Clack's environment for the object oriented functional language Clover(Braine & Clack, 1997) is also based on type directed interconnection but no type information is present in expression graphs. Kelso's general purpose system(Kelso, 1994) and Addis and Addis' functional schematic programming language Clarity (Addis & Addis, 1996) both label expression graphs with polymorphic types which are resolved incrementally during interconnection. Najork and Golin's Enhanced Show-and-Tell(Najork & Golin, 1990) is a visual data flow language which incorporates polymorphic type checking. Expression graphs are composed in part from typed icons for data sources.

While type determined component interconnection ensures type consistency, it may be useful to explain why a proposed connection is inappropriate, to clarify misconceptions about component functionality. The above systems appear to offer no support for explaining typed interconnection constraints.

5 A 2D colour type visualisation

We wish to take advantage of the high resolution graphics facilities found in contemporary personal computer technology in visualising type expressions. Our experience of student problems, discussed anecdotally above, suggests the use of graphic techniques to highlight differences between types and the successive instantiation of type variables. One possibility was to use special icons to identify types. However, we thought that this was likely to lead to visual clutter with complex, nested types as fine iconic detail is progressively lost. Instead, we decided to investigate the use of colour, where each base type is represented by a base colour and more complex types are represented by composing and nesting the base type colours [3]. We speculate that, while users may not be able to directly identify a type from a complex colouring, the colourings will still be sufficiently distinct to enable discriminations between correct and incorrect type combinations. We also speculate that colours may be easier to differentiate than iconic graphics.

Here we discuss a pure functional SML subset providing integer, real, boolean, string, tuple, list and functions types.

First of all, all types are visualised as rectangles. Base types are represented a rectangles of solid colour: `int` in red, `real` in purple, `bool` in pink and `string` in orange - Fig 1.

A tuple type is represented by a rectangle with vertical stripes for each of the field's types, for example, Fig 2 shows an `int * real * string`.

A list type is represented with the element type above a reversed "L". This is chosen to mirror graphically the SML postfix constructor notation. Fig 3 shows a

[3] Note that black and white presentation does not do justice to colour: we have provided schematic renderings of icon visualisations in the following account.

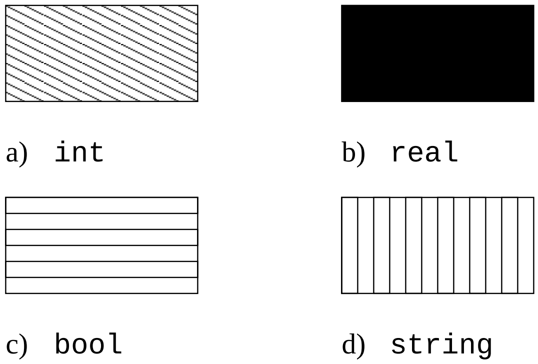

Fig. 1. Base types

Fig. 2. Tuple - `int * real * string`

`(int * real * string) list`. The list is the only structured datatype in our SML

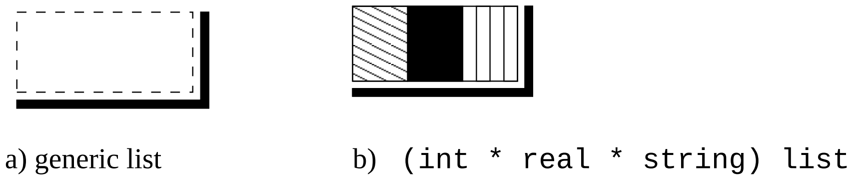

Fig. 3. List

subset so a unique mark is appropriate. For arbitrary datatypes, some general means of type identification would need to be devised: this is currently under consideration.

Formally, an SML function is a mapping from a single domain to a single range. In practise, it is usual to treat a curried function as if it were a function of several domains. Thus, a curried function type is visualised as a rectangle with the type for each level of nesting from left to right at the top and the result type in the bottom right hand corner - Fig 4. For example, Fig 5 shows a `string -> string list -> bool` function.

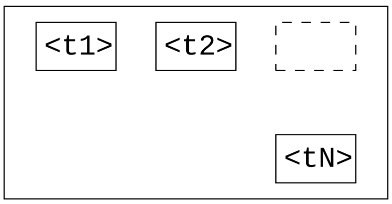

Fig. 4. Function - <t1> -> <t2> ... -> <tN>

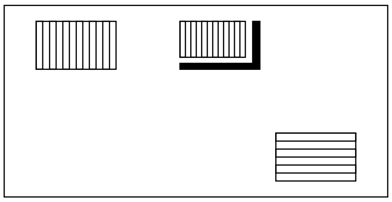

Fig. 5. string -> string list -> bool

Finally, a polymorphic type variable is represented by a rectangle with the identifier in it - Fig 6. For example, Fig 7 shows the `('a -> 'b) -> 'a list -> 'b list` type for the `map` function.

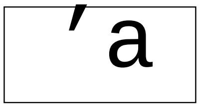

Fig. 6. Polymorphic - 'a

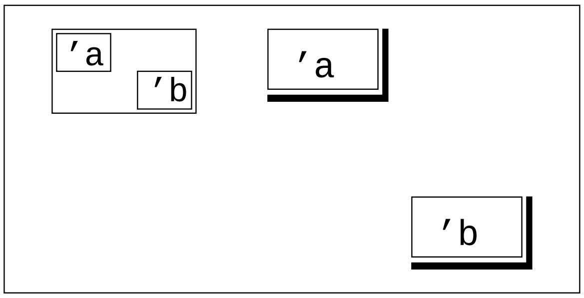

Fig. 7. map - ('a -> 'b) -> 'a list -> 'b list

6 Evaluation of the visualisation

We wished to evaluate our naive hypothesis that the above visualisation would aid in the understanding of polymorphic types and their unification during function application. All our subjects were to have at least introductory programming experience in Standard ML and would thus already be familiar with textual based type understanding. While this would weight the evaluation against the visualisation, we felt it worth pursuing to check that the visualisation was at least a plausible alternative to textual type presentation. In fact, our experiment showed the visualisation to be no better or worse than the textual approach, despite the subjects' greater experience of textual type understanding. This suggests that further development and evaluation of the visualisation might be worthwhile.

The evaluation is based on a series of questions involving the types arising from function application. For each question, the subject is given the types of a function and its argument, and is asked to select one of four possible resultant types. In addition, they are given the options of the result being some other type or a type error. For an example, see Figure 8.

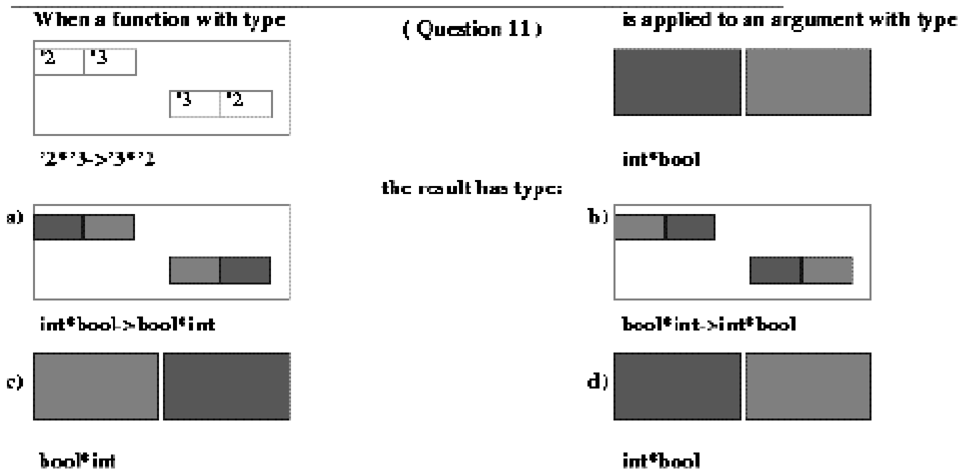

Fig. 8. Question 11 - visualised

We wished to compare the use of the visualisation with the use of text on the same questions. We also wished to check for any learning effect as a result of answering the visualisation based questions. We made up a set of 20 questions, with 6 involving no type variables, 4 involving one type variable, 8 involving two type variables and 2 involving three type variables - Set 1. We then duplicated this set, changing the base types and the order of answer options - Set 2. Finally, textual versions of both sets were prepared.

For the experiment, we divided the subjects up into two groups. The first group answered a visualised version of Set 1 while the second group answered a textual version of Set 1. Then, the first group answered a textual version of Set 2 while the second group answered a visualised version of Set 2. Finally, both groups answered a short questionnaire on their prior experience with polymorphic typed languages, and their subjective rating of the visualisation. The experiment was conducted on paper and subjects were given 10 minutes in which to answer each test set.

Our experiment was conducted with 2 groups of 8 subjects taking a 1st year course on Functional Programming in Standard ML at Heriot-Watt University. During the course, students were shown visualisation diagrams to augment textual explanation of function types. However, they still used a textual based prgramming environment for practical work. At the start of the experiment, the subjects were given a brief explanation of the visualisation.

Figures 9 and 10 show summaries of raw results for numbers of wrong answers and of unanswered question. They also shows summaries of normalised results where the number of wrong answers is divided by the number of questions answered.

The group answering visualised followed by textual questions made fewer mistakes but failed to answer more questions on the visualised questions compared

	visual			text		
	wrong	unanswered	normalised wrong	wrong	unanswered	normalised wrong
average	5.1	1.9	30.9%	6.4	0.4	32.6%
median	3.0	1.5	15.8%	2.5	0.0	13.2%

Fig. 9. Results - visualisation followed by textual questions

	text			visual		
	wrong	unanswered	normalised wrong	wrong	unanswered	normalised wrong
average	7.0	0.9	35.9%	7.4	0.8	37.9%
median	4.5	0.0	22.5%	5.5	0.0	27.5%

Fig. 10. Results - textual followed by visualisation questions

with the textual questions. Contrariwise, the group answering textual followed by visualised questions made fewer mistakes but failed to answer more questions on the textual questions compared with the visualised questions. All differences were insignificant but might suggest a waning in interest from the first question set to the second set.

T-tests showed that there were no significant differences between the groups so the results for all visualised and and all textual questions were combined - Figure 11. Overall, on the visualisation questions, subjects made slightly fewer mistakes but failed to answer twice as many questions. However, t-tests again show that these differences are not significant.

	visual			text		
	wrong	unanswered	normalised wrong	wrong	unanswered	normalised wrong
average	6.3	1.3	34.4%	6.7	0.6	34.2%
median	4.0	0.5	20.4%	4.0	0.0	20.0%

Fig. 11. Combined visualisation and textual question results

After each experiment, each subject was asked to rate the visualisation for ease of use, speed of use and preference - see Figure 12. On average, those who answered the visualised questions first found the visualisation comparable in ease of use to and the same speed as the text. However, on average they also thought that the visualisation would slightly hinder type error understanding. On average, those who answered textual questions first found the visualisation slightly easier to use and slightly faster than the text, and thought that the visualisation would make no difference to understanding of type errors.

All answers were then combined - Figure 13. On average, those who answered the visualised questions first thought that the visualisation was slightly easier to use and slightly faster than the text, and would make no difference to understanding type errors. Median scores for all answers suggest that this cohort found no difference between the visualised and textual approaches, and thought that the visualisation would neither help nor hinder type understanding.

	visual then text			text then visual		
	difficulty	speed	help	difficulty	speed	help
average	4.1	4.1	4.3	3.4	4.9	3.9
median	4.0	4.0	4.0	3.5	5.3	4.3

Fig. 12. Questionnaire results by visual/textual order: difficulty: 1=easier, 4=same, 7=harder; speed: 1=slower, 4=same, 7=faster; help: 1=helps, 4=neutral, 7=hinders

	difficulty	speed	help
average	3.7	4.6	4.1
median	4.0	4.0	4.0

Fig. 13. Combined questionnaire results: difficulty: 1=easier, 4=same, 7=harder; speed: 1=slower, 4=same, 7=faster; help: 1=helps, 4=neutral, 7=hinders

Students also had an opportunity to provide other comments. Some said that they found the visualisation confusing. Others indicated that they preferred the visualisation. This would suggest that a future evaluation should also consider the subjects' dominant cognitive style on the Verbaliser/Imager dimension (Riding & Cheema, 1991) as a factor in their relative performances on textual and visual questions. We were retrospectively pleased that a self-identified colour blind student said that they found the visualisation very helpful.

While one should be cautious in drawing conclusions from such small numbers of subjects, overall there seems to be no disadvantage to using the visualisation as opposed to the text. Of course, there is also no apparent advantage. However, the subjects had three months experience of practical text based programming and had only seen the visualisation in passing in lectures. Thus, we think that further investigation of the practical use of this visualisation is worthwhile to see whether or not it can offer a viable alternative to text based type information. To that end, we are developing a type exploration environment incorporating the visualisation. This is discussed in the following sections.

7 Function oriented icon generation

Central to visual programming is the use of graphical visualisations of program components to ease their identification and understanding of their interconnection. However, the manner in and degree to which language constructs are represented graphically have significant impacts on the intelligibility and ease of use of visual programming systems.

At one extreme, all language constructs may be given graphical representations. Cardelli's proposal is based on icons with different shapes and enclosed symbols or labels for different constructs. In Pagan's FP system(Pagan, 1987), there are labeled rectangular representations of all constructs. In Reekie's proposed Visual Haskell(Reekie, 1994), labeled graphical constructs are are joined together within enclosing boxes by arcs. This approach leads to large numbers of similar representations which require text to distinguish between them but are substantially bulkier than the equivalent program code. Thus, less efficient use is made of display space on a screen, reducing the amount of program that may be viewed at one time.

More fundamentally, there is a danger of a mismatch between a task and a notation. For example, in an early empirical study, Green and Petre(1992) found that for conditional logic design, people had more difficulty understanding graphical representations of programs than with the equivalent textual code. They suggest that the structure of the graphics makes it harder to scan a visual program. especially where graphical structures contain "knots" as a result of arc intersections. In a more recent survey of 19 evaluations of visual programming systems, Whitley(1997) identified 9 where the visual notation was beneficial and 10 where it appeared to inhibit problem solving.

We decided to adopt a mixed mode visualisation based on the function as the fundamental unit of presentation. Functions are entered as text for display as icons based on an augmented form of the function type discussed above. Interactive graphical techniques are then used to connect actual to formal parameters, to collapse partially and fully applied functions to simpler iconic forms, and to expand collapsed function applications back into the original graphical form.

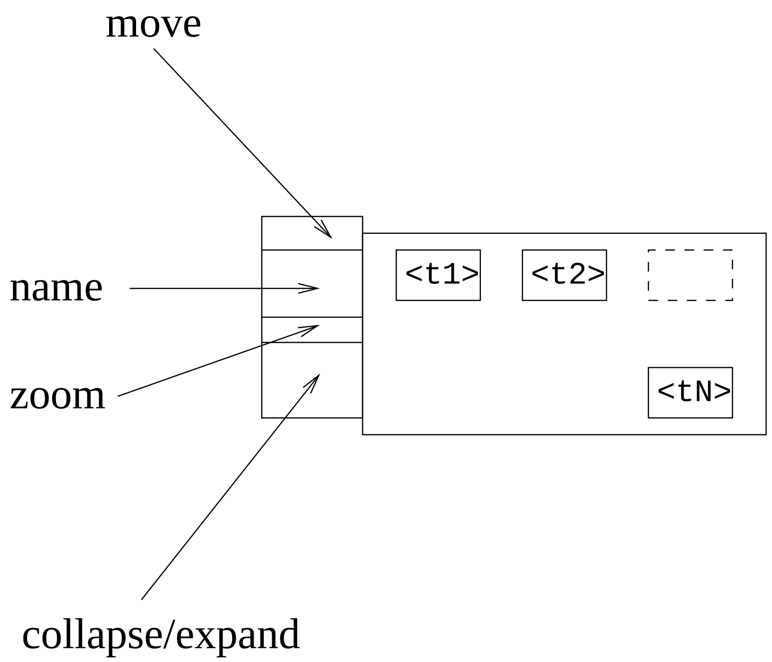

Fig. 14. Function icon

Fig 14 shows the function icon. To the left of the type is a rectangle containing four buttons. The move button is selected to move the icon. The function name button is selected to identify other functions which may take this function as an argument by highlighting the appropriate argument positions. The zoom button is selected to increase (or decrease) the size of the icon. The collapse/expand button is selected to combine a function application made from linked icons into a single icon or to recreate the linked icons contributing to a collapsed icon. For example, the `map` function is shown in Fig 15. We also generate icons from values of other

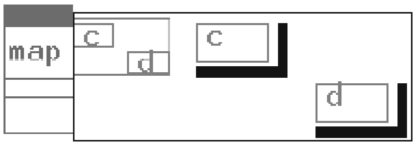

Fig. 15. map icon

types, here augmenting the type representation with a textual reminder of its type.

8 Implementation

The visualisation has been incorporated into a prototype visual programming environment for the above SML subset, in C++ using the Motif X Library. Fig 16 shows the interface layout.

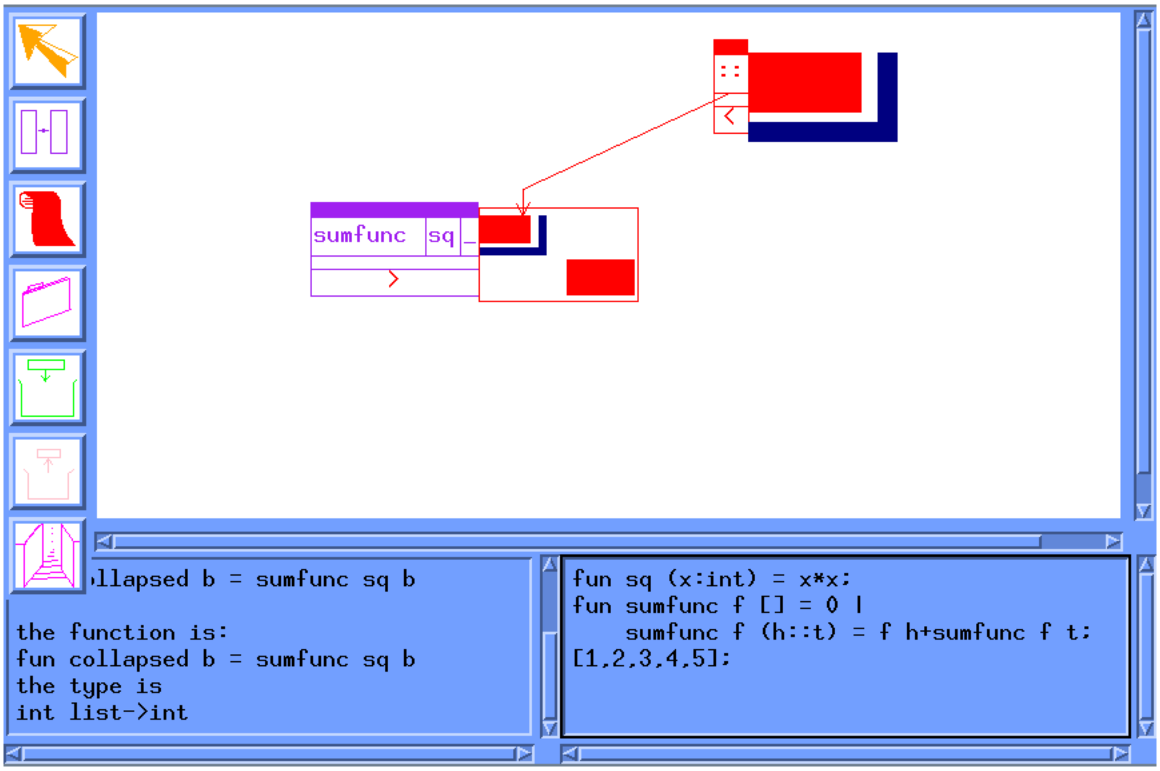

Fig. 16. Interface

On the left is a column of buttons for loading pre-defined functions, copying an icon, displaying text from an icon, saving text for all icons to file, deleting an icon, restoring a deleted icon, and exiting the system. Icons are displayed in the main central display. Icons have the functionality described above. The lower left hand display shows text from icons and information about system progress. The lower right hand display enables textual entry of SML for iconic visualisation.

Behind the visualisation are a lexical analyser, parser and type checker for the SML subset taught to first year students (Foubister *et al.*, 1997). The subset provides integer, real, boolean and string base types, which may be combined as tuples and lists. The type checker is based on Field and Harrison's account of the W algorithm(Field & Harrison, 1988). SML values are represented within the system as abstract syntax trees and text is reconstructed from trees through pretty printing.

9 Type visualisation during function application

The type visualisation provides broad information about which functions may be applied to which arguments. As in a text based system, the types must be compatible but here there is substantive visual evidence for compatibility. The types must either look the same or have the same structure with type variables in one in corresponding positions to arbitrary sub-structures in the other. For example, Fig 17 shows the function:

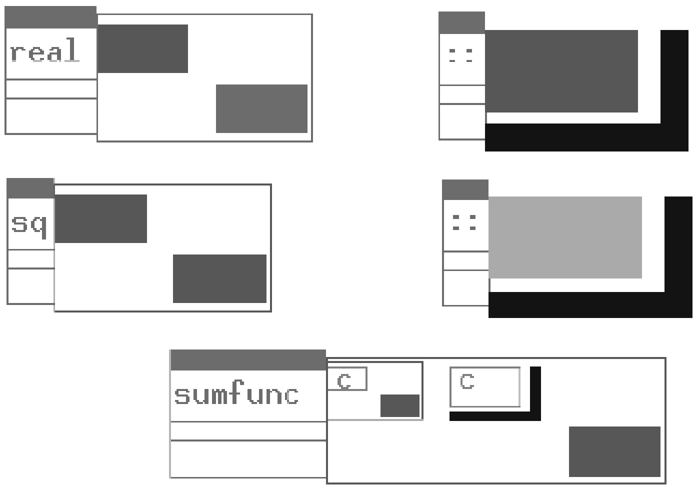

Fig. 17. Example components

```
fun sumfunc _ [] = 0 |
    sumfunc f (h::t) = f h+sumfunc f t
fn : ('a -> int) -> 'a list -> int list
```

which sums the application of function `f` to every element of an arbitrary list. On the left are the function `real : fn : int -> real` which converts an integer to a real and the function:

```
fun sq (x:int) = x*x
fn : int -> int
```

which squares its integer argument. On the right are icons for a list of strings and a list of integers.

By inspection, `sq` may be `sumfunc`'s first argument: `'a` will match `int` and `int` will match `int`. The function `real` may not be `sumfunc`'s first argument as the `real` type will not match `int`. Similarly, both lists may be `sumfunc`'s second argument.

The visualisation also provides active information about which argument positions an icon may be attached to. For example, selecting `sq`'s name - Fig 18 - highlights `sumfunc`'s first argument.

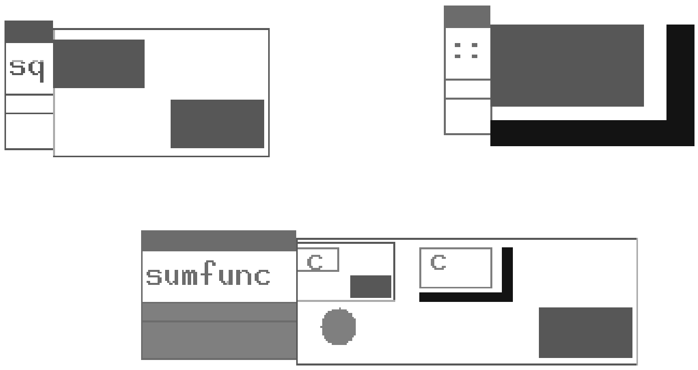

Fig. 18. Selecting sq

Selecting the highlighted argument - Fig 19 - joins `sq`'s icon to that position and also changes the type of `'a` in the rest of `sumfunc`'s icon to reflect its unification with `int`.

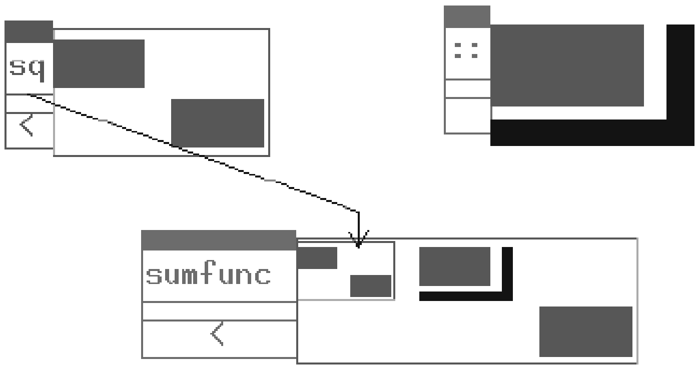

Fig. 19. Connecting sq to sumfunc

Note also that the collapse button has changed to `<`. Selecting this button - Fig 20 - collapses the join of `sq` with `sumfunc` to a new composite icon.

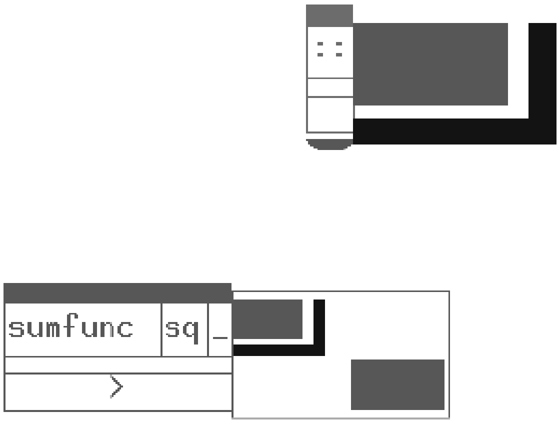

Fig. 20. Collapsing sq joined to sumfunc

The icon for the connected `f` argument has been removed and the collapse button has changed to `>` showing that this icon can be expanded back up again.

Selecting the `int list` icon - Fig 21 - has highlighted the single argument in the collapsed icon for the application of `sumfunc` to `sq`..

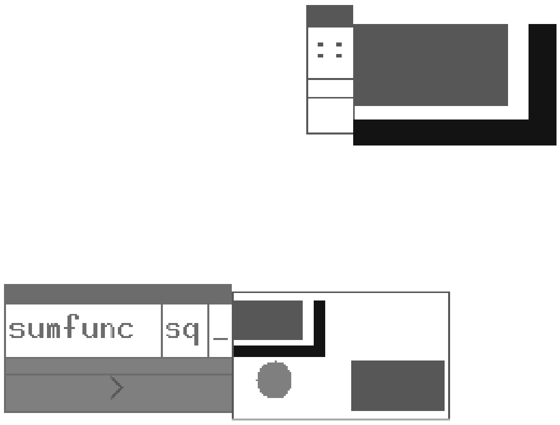

Fig. 21. Selecting the int list

Once again, the icon may be connected to the argument - Fig 22.

Finally, the composite icon may be collapsed further to an icon for the final integer - Fig 23.

10 Arbitrary order argument provision

Usually functions must be applied to arguments from left to right. However, in a visual environment it is possible to take advantage of the 2D space to enable

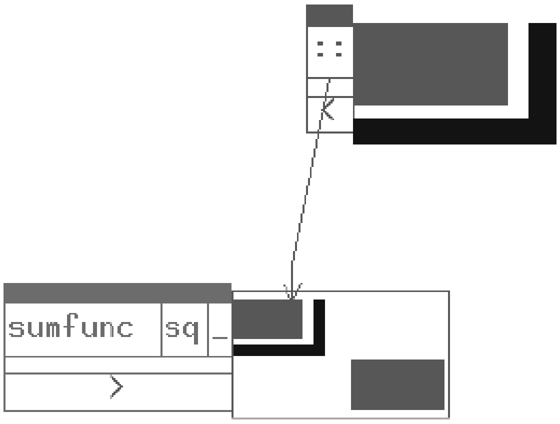

Fig. 22. Connecting the int list to sumfunc/sq

Fig. 23. Collapsing int list joined to sumfunc/sq

arbitrary order argument provision. Consider once again Fig 17. If the `int list` is selected then `sumfunc`'s second argument is highlighted - Fig 24.

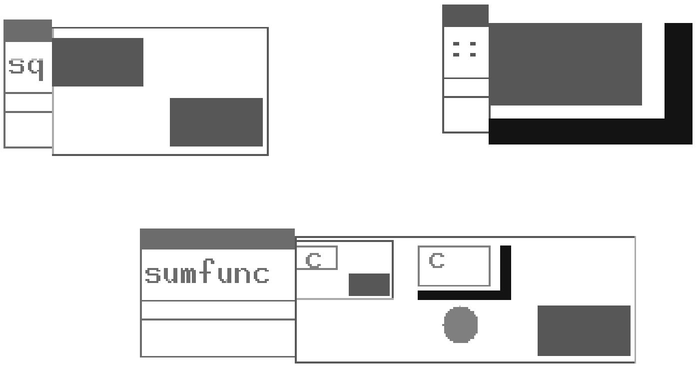

Fig. 24. Selecting the int list

Subsequently, joining the `int list` icon to the second argument position unifies `'a` with `int` in `sumfunc`'s icon - Fig 25.

We think that this may be useful in explaining the link between different occurrences of the same type variable, as well as more generally for free form visual type exploration.

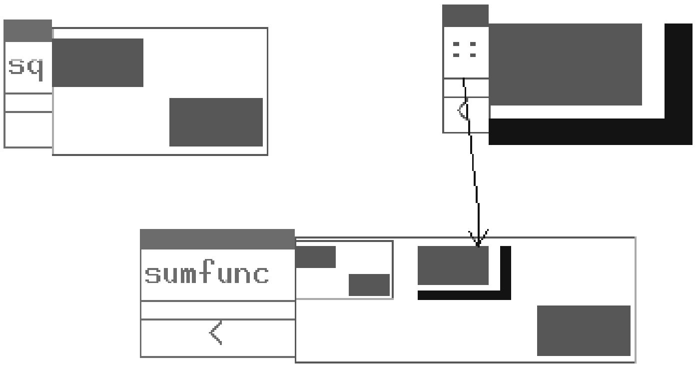

Fig. 25. Connecting the int list to sumfunc

Text may be generated simply from an arbitrary order argument application. For a nested function with N layers, each with an associated pattern:

```
fun f p1 p2 ... pN = e
```

if the Ith pattern `pI` is applied to some argument `aI` then an equivalent text is:

```
fun f' p1 p2 ... pI-1 = f p1 p2 ... pI-1 aI
```

11 Further Work

We have shown how a simple visualisation of polymorphic types may be incorporated within a visual environment to guide and clarify type compatibility in function application. We now intend to use it to investigate the elucidation of type errors where there are multiple potential error sources. The environment will be extended to highlight such sources and to structure their exploration through static inspection of and incremental changes to the program. The extended system will also be used for the comparative evaluation of different approaches to identifying error sources, discussed above, and of potential new error source identification and explanation schemes. To these ends, the core typechecking and visualisation system is being reconstructed in Standard ML with SML/Tk. This will support a larger pure functional subset of SML, in particular user defined datatypes.

Our questionnaire responses from textually experienced subjects suggested a range of reactions to the visualisation, from very positive to quite antagonistic. Thus, in the interests of supporting this variagated but predominantly text using audience, it might be worth augmenting the pure graphical presentation with text within the type rectangles, to offer greater support to those preferring text. Furthermore, an evaluation of the augmented representation, and comparison with the textual results discussed above, might enable clarification of the relative merits of text and graphics for presenting polymorphic type checking.

While the system is intended primarily for investigating type explanation, it would be straightforward to extend it into a full visual programming environment. This would involve generating text from visualised function applications, sending the text to a SML system and then visualising the results. However, we think that the current division between text for function bodies and graphics for function application may restrict ease of use for general programming. It might be more fruitful to develop a new visual programming environment incorporating the visualisation, using design criteria such as Yang et al's (1997), based on Green and Petre's cognitive dimensions for visual programming (Green & Petre, 1996). It would be interesting to explore the application of such criteria specifically to visual functional programming.

Acknowledgments

Yang Jun wishes to thank the China Scholarship Council for supporting this research during leave from Guizhou Normal University, Guiyang City, China, and ORS and Heriot-Watt University for continuing support.

We both wish to thank our colleague Phil Trinder and the anonymous referees for very helpful comments on earlier drafts of this paper.

References

Addis, T. R., & Addis, J. J. Townsend. (1996). *The Clarity Manual.* Tech. rept. Department of Information Science, University of Portsmouth.

Beaven, M., & Stansifer, R. (1993). Explaining errors in polymorphic langauges. *ACM Letters on Programming Languages and Systems*, **2**, 17–30.

Braine, L., & Clack, C. (1997). Object-Flow. *Pages 418–419 of: Proceedings IEEE Symposium on Visual Languages (VL97).*

Cardelli, L. (1982). *Two-dimensional syntax for functional languages.* Tech. rept. CSR-115-82. Department of Computer Science, University of Edinburgh.

Duggan, D., & Bent, F. (1996). Explaining type inference. *Science of Computer Programming*, **27**(1).

Field, A., & Harrison, P. (1988). *Functional programming.* Prentice-Hall.

Foubister, S., Michaelson, G., & Tomes, N. (1997). Automatic assessment of elementary Standard ML programs using Ceilidh. *Journal of Computer Aided Learning*, **13**, 99–108.

Green, T. R., & Petre, M. (1996). Usability analysis of visual programming environments. *Journal of Visual Languages and Computing*, **7**, 131–174.

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

Johnson, G. F., & Walz, J. A. (1986). A maximum-flow approach to anomaly isolation in unification-based incremental type inference. *Pages 44–57 of: Proceedings of 13th ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, USA.* ACM Press.

Kelso, J. (1994). *A visual representation for functional programs.* Tech. rept. Department of Computer Science, Murdoch University, Perth, Australia.

Michaelson, G. (1996). Automatic assessment of functional program style. *Pages 38–46 of: Proceedings of 1996 Australian Software Engineering Conference, Melbourne.* IEEE Computer Society Press.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**.

Milner, R., Tofte, M., Harper, R., & MacQueen, D. (1997). *The definition of standard ml (revised)*. MIT.

Najork, M. A., & Golin, E. (1990). Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. *Pages 215–220 of: Proc IEEE Workshop on Visual Languages, Skokie, IL.*

Pagan, F. G. (1987). A graphical FP language. *SIGPLAN Notices*, **22**(3), 21–39.

Poswig, J., Vrankar, G., & Morara, C. (1994). VisaVis: a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, **5**, 83–111.

Reekie, H. J. (1994). *Visual Haskell: a first attempt*. Tech. rept. Research Report 94.5. Key Centre for Advanced Computing Sciences, University of Technology, Sydney.

Rideau, L., & Thery, L. (1997). *Interactive programming environment for ML*. Tech. rept. 3139. INRIA.

Riding, R., & Cheema, I. (1991). Cognitive styles - an overview and integration. *Educational Psychology*, **11**(3+4), 193–215.

Rittri, M. (1993). *Finding the source of type errors interactively*. Tech. rept. Department of Computer Science, Chalmers University of Technology, Sweden.

Soosaipillai, H. (1990). *An explanantion based polymorphic type checker for Standard ML*. M.Phil. thesis, Dept of Computer Science, Heriot-Watt University, Edinburgh, Scotland.

Turner, D. (1990). *Enhanced error handling for ML*. Tech. rept. CS4 Project, Department of Computer Science, University of Edinburgh, Scotland.

Wand, M. (1986). Finding the source of type errors. *Pages 38–43 of: Proceedings of 13th ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, USA*. ACM Press.

Whitley, K. (1997). Visual programming languages and the evidence for and against. *Journal of Visual Languages and Computing*, **8**, 109–142.

Yang, S., Burnett, M. M., DeKoven, E., & Zloof, M. (1997). Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing*, **8**, 563–599.