A visualisation of polymorphic type checking

Yang Jun Greg Michaelson[†]

Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton, Edinburgh, EH14 4AS
phone: +44 131 451 3422
fax: +44 131 451 3431

yang/greg@cee.hw.ac.uk

[†]Primary contact.

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 visual programming environment in which function combinations are constrained by type matching.

Keywords: functional programming; polymorphic typechecking; visualisation; Standard ML

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 representations of types and functions are discussed and their incorporation in a visual programming system for an SML subset are considered. Finally, our approach is assessed and further work is suggested.

2 Novice misconceptions with types

In 6 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, 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[Mic96], 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 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 [1] 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 [Mil78] 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 [MTHM97], 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 [Soo90] proposed that type errors might be elucidated by traversing a tree of type checking decisions made by the W algorithm. Her incomplete system for an SML

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

fragment enabled such exploration by offering menus of type decisions at each level for further investigation. Duggan and Bent [DB96] 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[BS93] also discuss a system for SML. Parse trees are decorated during type checking and traversed to generate depth first explanations. Wand[Wan86] 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[RT97] provide support for type explanation in a CAML implementation using techniques derived from Wand and from Bent and Duggan. Rittri[Rit93] 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[JW86] 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[Tur90] 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[Car82]. 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[PVM94] 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[BC97] is also based on type directed interconnection but no type information is present in expression graphs. Kelso's general purpose system[Kel94] and Addis and Addis' functional schematic programming language Clarity [AA96] both label expression graphs with polymorphic types which are resolved incrementally during interconnection. Najork and Golin's Enhanced Show-and-Tell[NG90] is a visual data flow language which incorporates polymorphic type checking. Expression graphs are composed in part from typed icons for data sources.

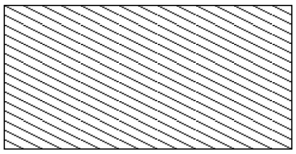
While type determined component interconnection ensures type consistency, it may be useful to explain why a proposed connection is inappropriate, to clarify misconceptions about component functionality. The above systems appear to offer no support for explaining typed interconnection constraints.

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 [2]. 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 remember and differentiate than iconic graphics.

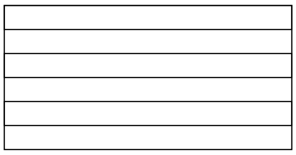
Here we discuss a pure functional SML subset providing integer, real, boolean, string, tuple, list and functions types.

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) `int`

b) `real`



c) `bool`

d) `string`

Figure 1: Base types

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`.

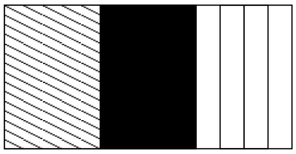


Figure 2: Tuple - `int * real * string`

A list type is represented with the element type above a reversed "L", for example a `(int * real * string) list` - Fig 3.

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 left hand corner - Fig 4.

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

a) generic list b)`(int * real * string) list`

Figure 3: List

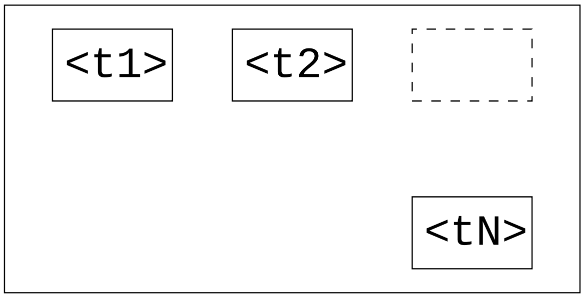


Figure 4: Function - <t1> -> <t2> ... -> <tN>

For example, Fig 5 shows a `string -> string list -> bool` function.

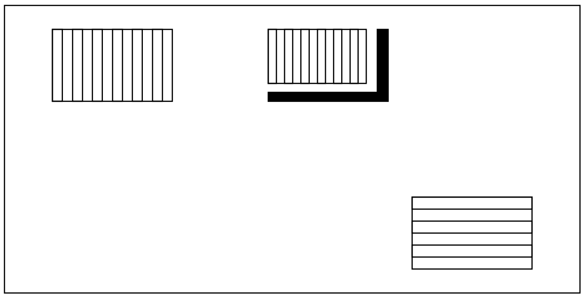


Figure 5: string -> string list -> bool

Finally, a polymorphic type variable is represented by a rectangle with the identifier in it - Fig 6.

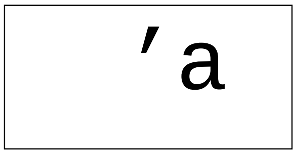


Figure 6: Polymorphic - 'a

For example, Fig 7 shows the `('a -> 'b) -> 'a list -> 'b list` type for the `map` function.

6 Function oriented icon generation

We think that visual functional programming offers a fertile framework within which to explore understanding of polymorphic type checking. However, this involves a number of fundamental representational considerations. Central to visual programming is the use of graphical visual-

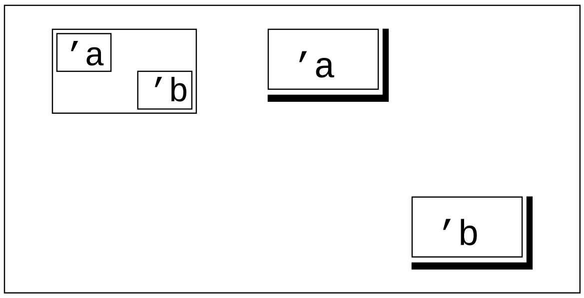


Figure 7: map - ('a -> 'b) -> 'a list -> 'b list

isations 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[Pag87], there are labeled rectangular representations of all constructs. In Reekie's proposed Visual Haskell[Ree94], 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, Green and Petre[GP92] found in an empirical study that people find graphical representations of programs harder to understand than 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.

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 8 shows the function icon. To the left of the type is a rectangle containing the move selector, the function name, the zoom selector and the collapse/expand selector. For example, the `map` function is shown in Fig 9.

We also generate icons from values of other types, here augmenting the type representation with a textual reminder of its type.

7 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 10 shows the function:

```
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:

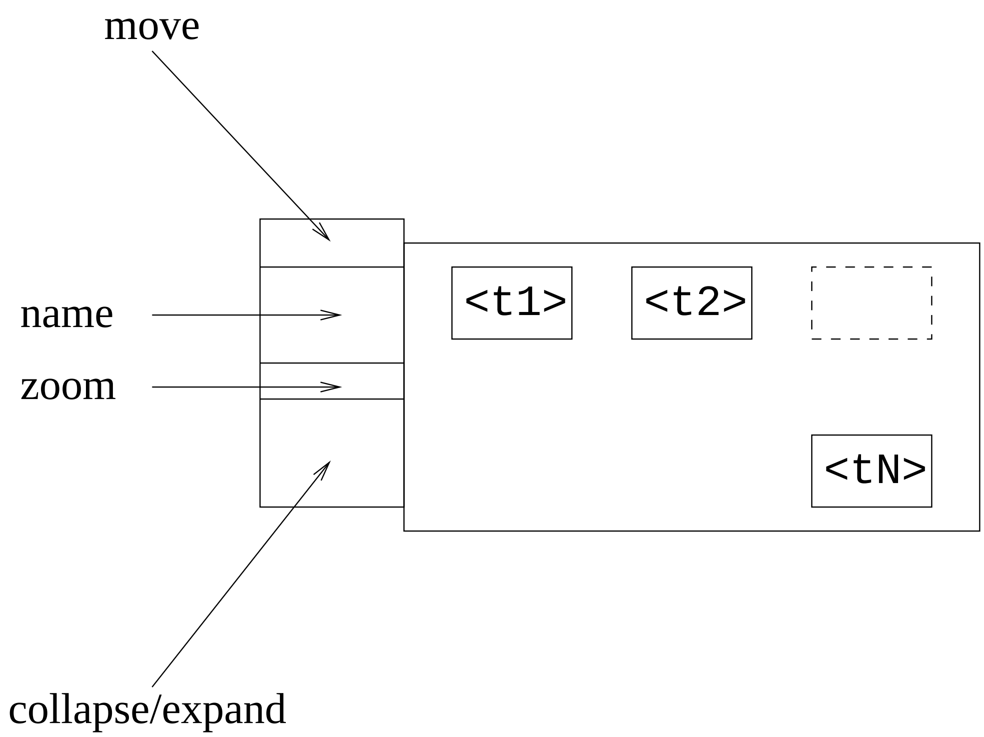


Figure 8: Function icon

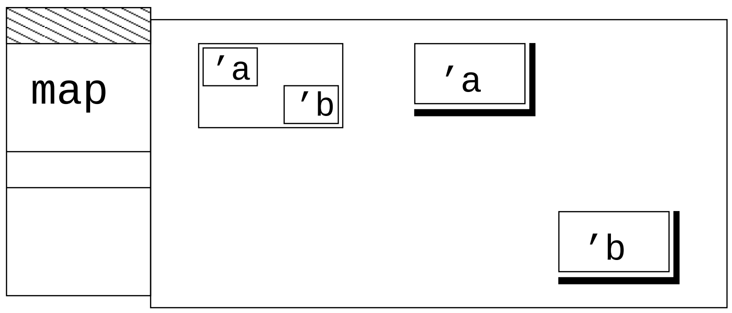


Figure 9: map icon

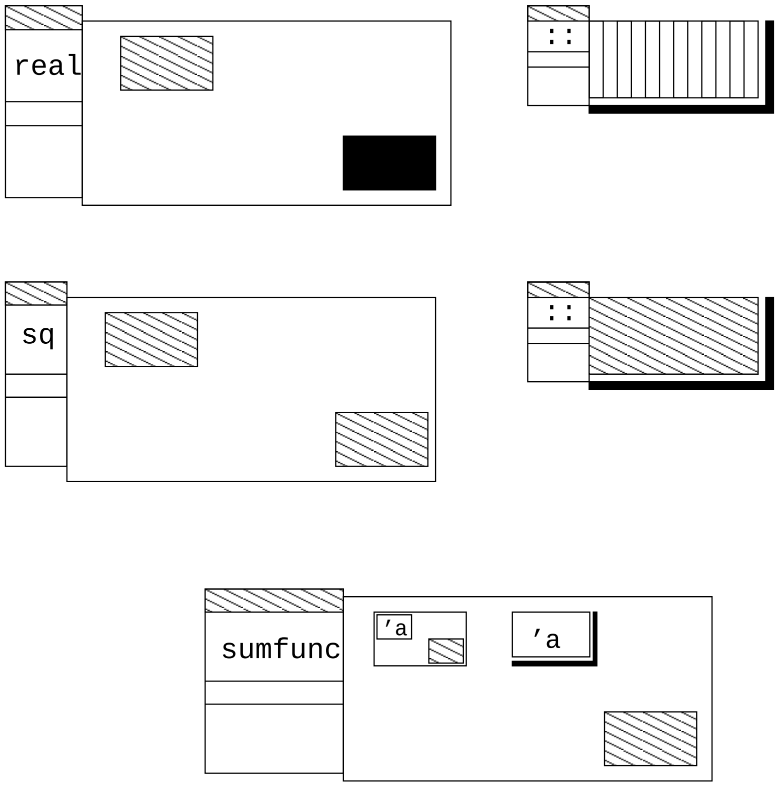


Figure 10: Example components

```
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 11 - highlights `sumfunc`'s first argument.

Selecting the highlighted argument - Fig 12 - 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`.

Note also that the collapse button has changed to `<`. Selecting this button - Fig 13 - collapses the join of `sq` with `sumfunc` to a new composite icon.

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 14 - has highlighted the single argument in the collapsed icon.

Once again, the icon may be connected to the argument - Fig 15.

Finally, the composite icon may be collapsed further to an icon for the final integer - Fig 16.

8 Arbitrary order argument provision

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

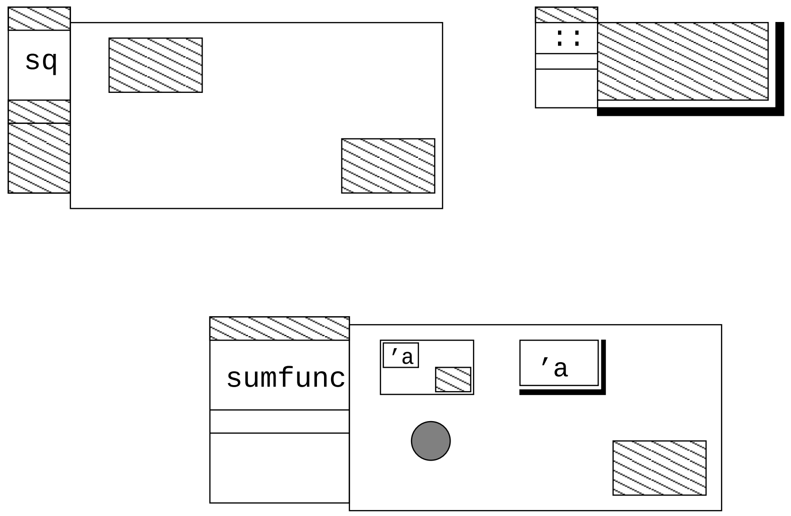


Figure 11: Selecting sq

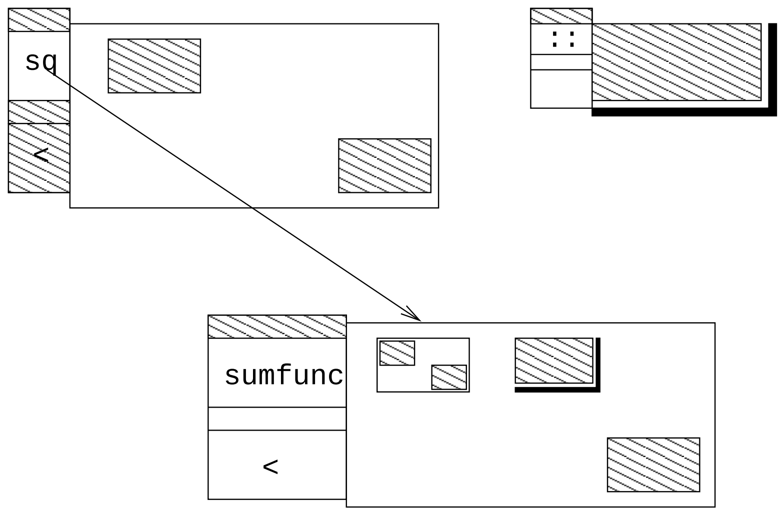


Figure 12: Connecting sq to sumfunc

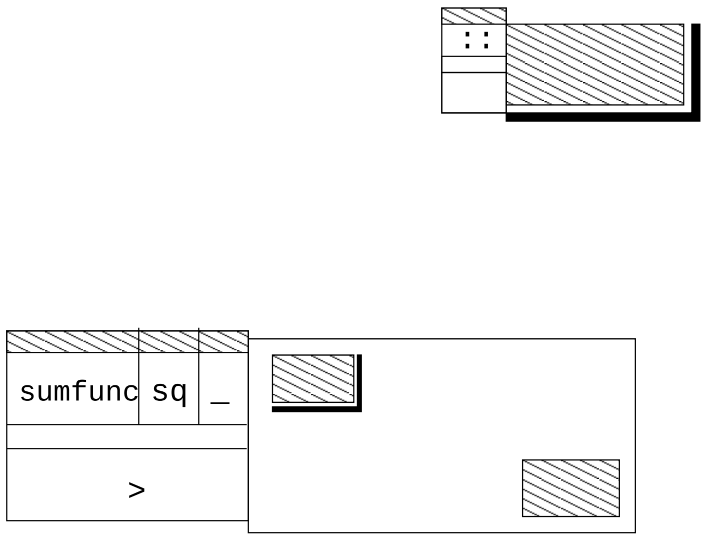


Figure 13: Collapsing sq joined to sumfunc

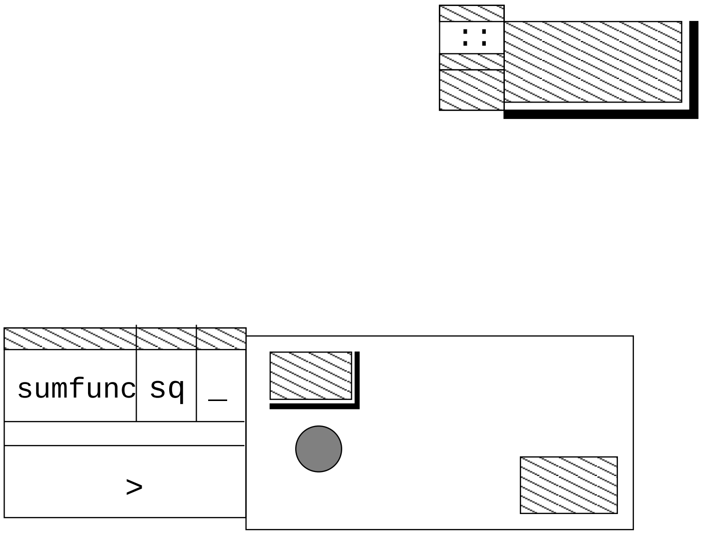


Figure 14: Selecting the int list

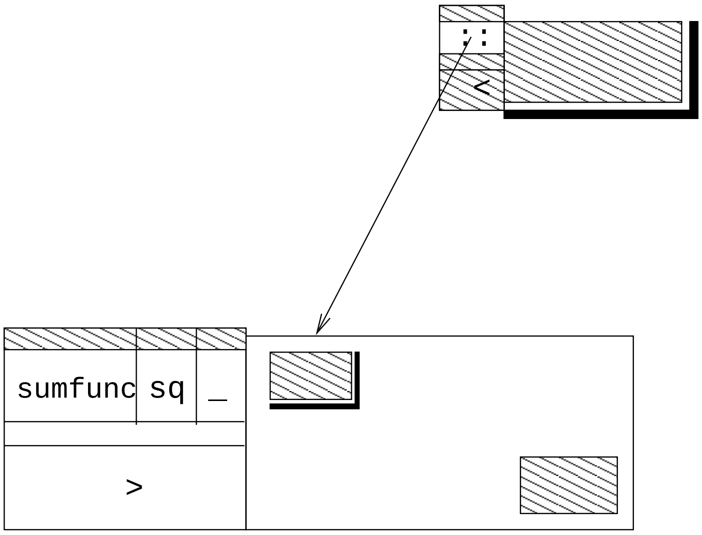


Figure 15: Connecting the int list to sumfunc/sq

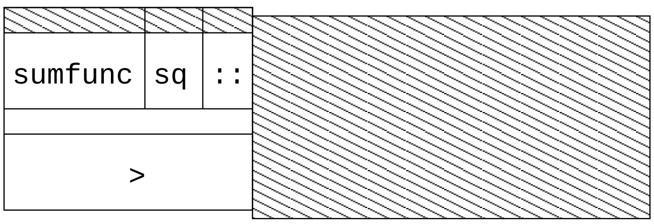


Figure 16: Collapsing int list joined to sumfunc/sq

argument provision. Consider once again Fig 10. If the `int list` is selected then `sumfunc`'s second argument is highlighted - Fig 17.

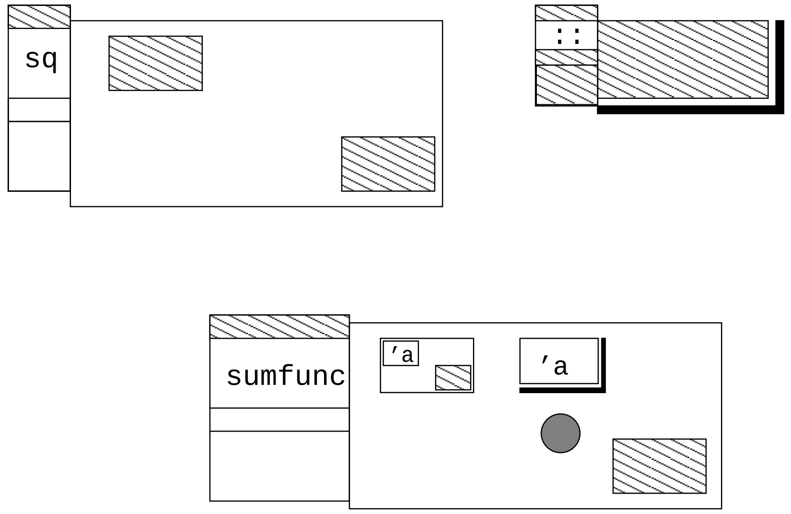


Figure 17: Selecting the int list

Subsequently, joining the `int list` icon to the second argument position unifies `'a` with `int` in `sumfunc`'s icon - Fig 18.

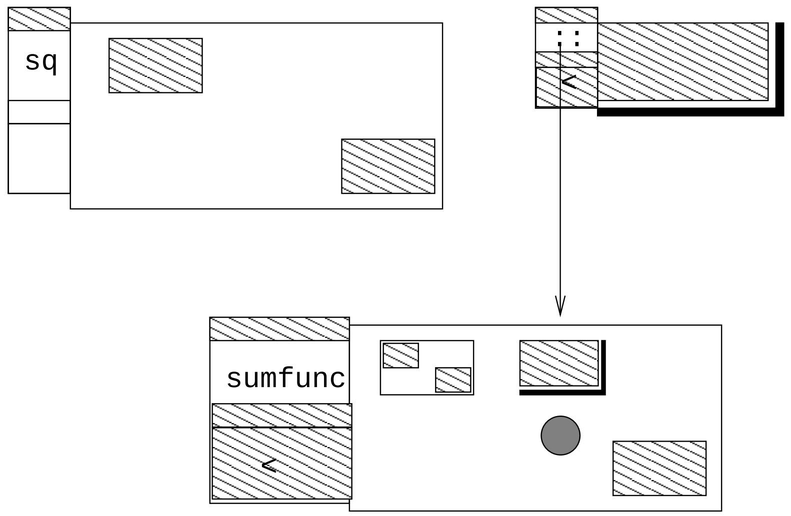


Figure 18: Connecting the int list to sumfunc

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 programming.

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
```

9 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 19 shows the interface layout.

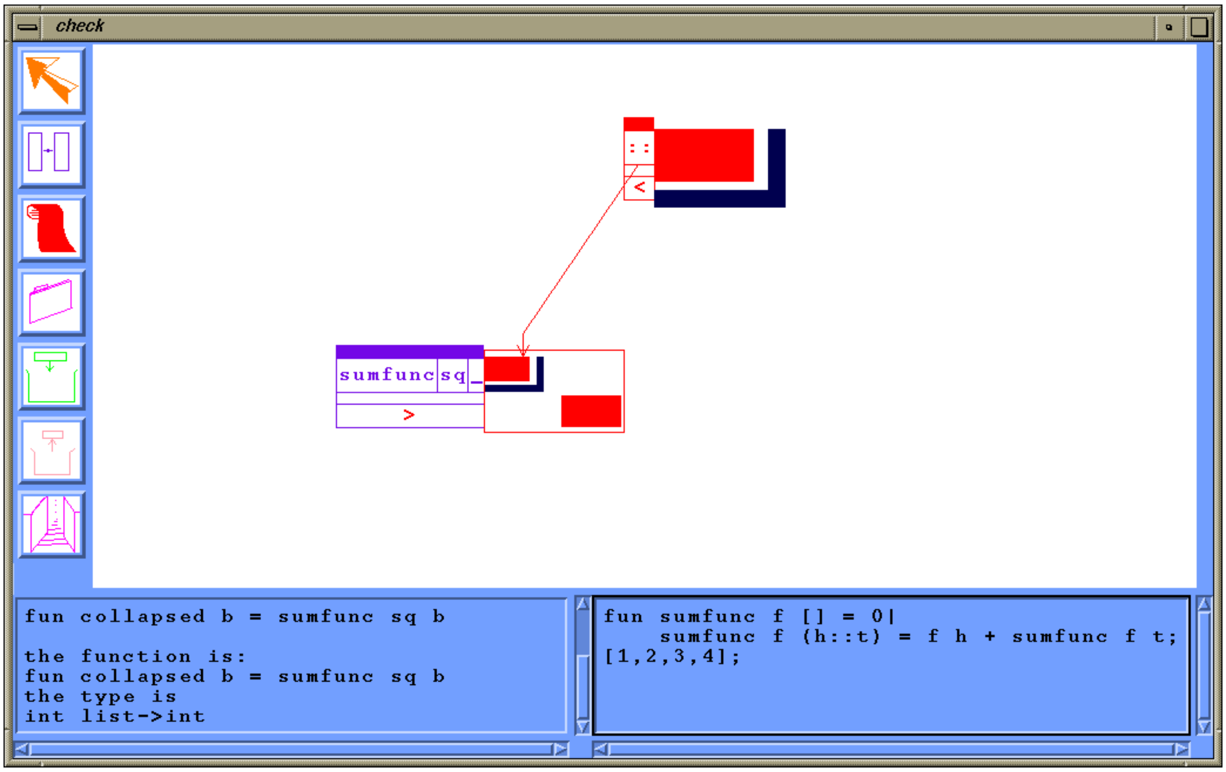


Figure 19: Interface

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. The type checker is based on Field and Harrison's account of the W algorithm[FH88]. SML values are represented within the system as abstract syntax trees and text is reconstructed from trees through pretty printing.

10 Cognitive dimension evaluation

Green and Petre[GP96] have enunciated a framework of cognitive dimensions for analysing HCI and the psychology of programming, which they apply to the usability evaluation of visual programming environments. We now consider briefly how our prototype system fits within their framework [GP96](p138-139).

Functional languages are what Green and Petre term 'abstraction tolerant' on their *abstraction gradient* from minimum to maximum levels of abstraction and encapsulation, enabling but not enforcing abstraction at all levels of programs. Our system reflects this through the hierarchy of text, base icon and collapsed icon corresponding to expressions, functions and applied functions.

Green and Petre comment that unrestrained *diffuseness*, with large numbers of symbolic and graphical entities required to express a meanings, makes representations bulky. In contrast, over terseness makes it hard to distinguish different but visibly similar components. We think that we have found a balance in the use of the function icon as the unit of visualisation with similar type representations indicating similar types but icon labeling and text expansion clarifying semantic differences.

A notational design which induces 'careless mistakes' is termed *error prone*. In textual input, our system is as error prone as any textual SML system. However, our type visualisation highlights appropriate and prevents inappropriate function applications. Our system is also free from *hidden dependencies* between components. At all levels, concise representations may be expanded to make the underlying program structures explicit.

Premature commitment, where a programmer must make decisions with having command of all relevant information, may arise where there are lots of internal dependencies in a notation or when a system unduly constrains the order of component interconnection. In our approach a programmer has more degrees of freedom than in linearly structured textual functional programming.

Our system brings *role expressiveness* in that the relation of components to each other is either explicit through interconnection or may be made so by expanding collapsed icons. It also lacks *viscosity* in that carrying a single change to icon interconnections usually involves at most two mouse button driven system operations: select and apply.

However, system *visibility* - the ability to view all parts of a program simultaneously or to juxtapose arbitrary components - is constrained by the nested hierarchy of icon applications. Furthermore, while our system provides *progressive evaluation* in terms of changing typed instantiations as components are connected, it does not offer semantic evaluation of programs. Finally, while our system makes strong use of colour and layout, this is related closely to what Green and Petre call the 'official' semantics of Standard ML so users have little control over such *secondary notations* [3].

11 Summary

We have shown how a simple visualisation of polymorphic types may be used within a visual programming environment to guide and elucidate type compatibility in function application. Our prototype system appears to rate well within Green and Petre's framework.

We now intend to use the environment with 1st year student who are new to functional programming, to evaluate whether or not it eases their understanding of polymorphic type checking when compared with text based presentations.

We also wish to extend the visualisation to more of Standard ML, in particular to user defined datatypes.

Acknowledgments

Yang Jun wishes to thank the China Scholarship Council for supporting this research during leave from Guizhou Normal University, Guiyang City, China.

References

[AA96] T. R. Addis and J. J. Townsend Addis. The Clarity Manual. Technical report, Department of Information Science, University of Portsmouth, May 1996.

[BC97] L. Braine and C. Clack. Object-Flow. In *Proceedings IEEE Symposium on Visual Languages (VL97)*, pages 418–419, 1997.

[BS93] M. Beaven and R. Stansifer. Explaining errors in polymorphic langauges. *ACM Letters on Programming Languages and Systems*, 2:17–30, 1993.

[Car82] L. Cardelli. Two-dimensional syntax for functional languages. Technical Report CSR-115-82, Department of Computer Science, University of Edinburgh, May, 1982.

[3] We have ignored the *closeness of mapping* and *consistency* criteria here as they relate to a language rather than an environment.

[DB96] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1), June 1996.

[FH88] A. Field and P. Harrison. *Functional programming.* Prentice-Hall, 1988.

[GP92] T. R. G. Green and M. Petre. When visual programs are harder to read than text. In G. C. van de Veer, M. J. Tauber, S. Bagnarola, and M. Antavoltis, editors, *Human-computer interaction: tasks and organisation, Proceedings of ECCE-6, CUD: Rome*, 1992.

[GP96] T. R. Green and M. Petre. Usability analysis of visual programming environments. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[JW86] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, USA*, pages 44–57. ACM Press, January 1986.

[Kel94] J. Kelso. A visual representation for functional programs. Technical report, Department of Computer Science, Murdoch University, Perth, Australia, December 1994.

[Mic96] G. Michaelson. Automatic assessment of functional program style. In *Proceedings of 1996 Australian Software Engineering Conference, Melbourne*, pages 38–46. IEEE Computer Society Press, July 1996.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17, 1978.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised).* MIT, 1997.

[NG90] M. A. Najork and E. Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *Proc IEEE Workshop on Visual Languages, Skokie, IL*, pages 215–220, October 1990.

[Pag87] F. G. Pagan. A graphical FP language. *SIGPLAN Notices*, 22(3):21–39, March 1987.

[PVM94] J. Poswig, G. Vrankar, and C. Morara. VisaVis: a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, 5:83–111, 1994.

[Ree94] H. J. Reekie. Visual Haskell: a first attempt. Technical Report Research Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, August 1994.

[Rit93] M. Rittri. Finding the source of type errors interactively. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden, 1993.

[RT97] L. Rideau and L. Thery. Interactive programming environment for ML. Technical Report 3139, INRIA, March 1997.

[Soo90] H. Soosaipillai. An explanantion based polymorphic type checker for Standard ML. Master's thesis, Dept of Computer Science, Heriot-Watt University, Edinburgh, Scotland, Sept 1990.

[Tur90] D. Turner. Enhanced error handling for ML. Technical report, CS4 Project, Department of Computer Science, University of Edinburgh, Scotland, May 1990.

[Wan86] M. Wand. Finding the source of type errors. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, USA*, pages 38–43. ACM Press, January 1986.