Explaining Polymorphic Types

Yang Jun, Greg Michaelson and Phil Trinder
Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, EH14 4AS, UK
{ceejy1,greg,trinder}@cee.hw.ac.uk

Abstract

Hindley-Milner type inference forms a strong basis for polymorphic type checking but is less well suited to explaining polymorphic types, as it introduces intermediate constructs that relate poorly to a programmer's understanding of the program. We present a type explanation system motivated by explanations given by expert humans rather than by Hindley-Milner inference algorithms. In particular, the system seeks to provide succinct explanations with minimal reference to artefacts of mechanised type inference.

1 Introduction

Type systems are fundamental to programming languages in constraining combinations of constructs, both to ensure the well-formedness of programs and to reduce runtime errors. However, despite considerable research into type systems, almost all widely used languages provide relatively impoverished sets of types, by direct abstraction from those explicit at a low level in von Neumann computers. Thus, modern imperative languages typically offer: integers, floating point numbers, characters and booleans, corresponding to fixed size bit patterns; arrays and records, corresponding to fixed size bit pattern sequences; pointers or references, corresponding to machine addresses of bit patterns. More elaborate types, such as discriminated unions or objects, are usually so much syntactic sugar on top of this basic type repertoire.

A typical imperative language has a monomorphic type system, where a construct retains the same concrete type throughout its lifetime. In particular, once a variable is assigned a type it may only ever be associated with values of that type. Strong monomorphic type systems enable simple static checks on programs for type consistency but unduly restrict programming flexibility.

In contrast, in polymorphic type systems, a construct may take different types in different contexts. Strachey [18] distinguished ad-hoc from parameterised polymorphism. Weakly typed languages like LISP and Prolog, with ad-hoc polymorphism, allow variables to be associated with arbitrary typed values without any statically enforced constraints. While this enables considerable flexibility in programming, static checks on programs for type consistency may no longer be possible. Run-time type checking, to avoid invalid construct combinations, is time consuming and brings attendant space overheads for explicit type tagging on values.

In parameterised polymorphism, in contrast, constructs are strongly typed but those types may include abstraction points. This enables the definition of generalised constructs with generic types, which are resolved when a concrete instance of the construct is created. The abstraction points are indicated by type variables, which may be uniformly replaced with other types: hence parameterised polymorphism.

For example, a generic list type, with an empty list `nil` and a constructor `CONS` may be defined in SML as:

```
datatype 'a list = nil | CONS of 'a * 'a list
```

This introduces a new type `'a list`. The type variable `'a` may be instantiated to any type, and the specific type is deduced from the context in which `CONS` is applied. Thus, `CONS(1,CONS(2,CONS(3,nil)))` is an `int list` because `1`, `2` and `3` are `int`s. Similarly, `CONS("a",CONS("b",CONS("c",nil)))` is a `string list` because `"a"`, `"b"` and `"c"` are `string`s.

An important contributions to programming language design has been Milner's and Hindley's development of first order parameterised polymorphic type inference. This enables static type checking at compile time, minimising the need for run-time type representations and checking. Hindley-Milner type systems [11] have most commonly been incorporated into functional languages like Standard ML(SML)[12], Miranda[19], Hope[4] and Haskell[7].

There is considerable anecdotal evidence that polymorphic type checking greatly reduces semantic errors in programs. Certainly, polymorphic types are central to the succinctness of contemporary functional languages, enabling a considerable degree of type-safe abstraction and hence component re-use. Nonetheless, parameterised polymorphism is provided in few languages in other paradigms, exceptions being the Napier imperative persistent database language [13], and λ-Prolog[10], a strongly typed Prolog variant used particularly for theorem proving.

We suspect that this apparent resistance to including polymorphic type systems in contemporary languages (e.g. Java) lies in a combination of unfamiliarity, and the aura of complexity and mathematical sophistication surrounding such systems. In particular, we claim that there is a fundamental mismatch between how polymorphic type inference algorithms deduce types and how people reason about types, which is particularly apparent in the reporting of polymorphic type errors.

We have investigated the human "inference" of polymorphic types, to aid the construction of a type explanation system which mimics human behaviour. In section 2, we discuss Hindley-Milner's type inference for parameterised polymorphism and its limitations for providing type explanations to people. In section 3, we consider how people explain types, and elucidate a simple set of rules which form the basis of our human-like type explanation system. In section 4, we present our human-like type explanation system. In section 5, we discuss related work, and we draw conclusions from our research in section 6.

2 Hindley-Milner Type Inference

2.1 Hindley-Milner Type Scheme

We first introduce a very simple functional language, to form the basis of the presentation of type inference:

exp	$\rightarrow$	$ident$	- identifier
		`fn` $ident$ `=>` exp	- function
		$(exp\ exp)$	- application

We will use the function `fn x => ((* x) 2)`, which doubles it's argument `x`, to illustrate type inference. Note that we introduce "constant" identifiers:

`2`	-	integer *two*
`*`	-	multiply two integers and return integer

Hindley-Milner polymorphic type inference may be expressed as a set of rewrite rules, which take the general form:

$$\frac{premise(s)}{conclusion}$$

That is, to infer *conclusion*, first establish $premise(s)$.

Inference is driven by *assumption environments*, each a set of identifier/type associations. A running assumption environment is denoted as A. Premises and conclusions take the general form:

$$A \vdash exp : type$$

That is, from prior assumptions A we may infer that expression exp has type $type$.

Types are represented by type expressions:

$type$	$\rightarrow$	$const$	- constructor
		var	- type variable
		$type_1 \rightarrow type_2$	- function
		$(\forall var.type)$	- type scheme

A *constructor* represents a base type, like `int`(integer) or *bool*(boolean). A type *variable*, usually a Greek letter, represents any type. A *function* represents a mapping from an argument type $type_1$ to a result type $type_2$. A *type scheme* represents a abstracted polymorphic type where the variable may be replaced by any type expression.

Thus, for our simple language, the "constant" identifiers have types:

`2`	:	`int`
`*`	:	`int` $\rightarrow$ `int` $\rightarrow$ `int`

VAR variables	$A.ident{:}\tau \vdash ident : \tau$	
ABS abstraction	$\dfrac{A.ident{:}\sigma \vdash exp : \tau}{A \vdash \texttt{fn}\ ident\ \texttt{=>}\ exp : \sigma \rightarrow \tau}$	
APP application	$\dfrac{A \vdash exp_1 : \sigma \rightarrow \tau \quad A \vdash exp_2 : \sigma}{A \vdash \texttt{(}\ exp_1\ exp_2\ \texttt{)} : \tau}$	
GEN generalisation	$\dfrac{A \vdash exp : \tau}{A \vdash exp : (\forall\ \alpha.\tau) \text{ - } \alpha \text{ not free in } A}$	
SPEC specialisation	$\dfrac{A \vdash exp : (\forall\ \alpha.\tau)}{A \vdash exp : [\sigma/\alpha]\tau}$	

Figure 1: Type rules for simple language

These are read as: `2` is an integer; `*` multiplies two integers to return an integer. Note that we treat binary operators as Curried functions to simplify presentation.

The polymorphic type rules in Figure 1 are after Field and Harrison[6].

For VAR, if the assumptions A include the association of identifier *ident* with type τ, then we may infer that *ident* has type τ, i.e. we may "look up" an identifier in a set of assumptions to find the associated type.

For ABS, with assumptions A augmented with the association of identifier *ident* and type σ, if we can show that expression *exp* has type τ, then the function `fn` *ident* => *exp* has type $\sigma \rightarrow \tau$, i.e. takes an argument of type σ and returns a result of type τ.

For [APP], if exp_1 has type $\sigma \rightarrow \tau$, i.e. is a function from a σ argument to a τ result, and exp_2 has type σ, then applying exp_1 to exp_2 returns a value of type τ.

The GEN and $SPEC$ rules are required to generalise and specialise polymorphic types.

Consider applying this scheme to `fn x => ((* x) 2)`. We assume that our initial assumption environment A contains the types for all the constant identifiers above, along with an appropriate type for the function variable:

$A = \{$`2:int, *:int -> int -> int, x:int` $\}$

We also assume that we have parsed the function and built an abstract syntax tree (AST). We work from the leaves of the AST back to the root, applying rules from the type scheme:

1.	$A \vdash$ `x:int`	VAR
2.	$A \vdash$ `*:int -> int -> int`	VAR
3.	$A \vdash$ `(* x):int -> int`	1, 2, APP
4.	$A \vdash$ `2:int`	VAR
5.	$A \vdash$ `((* x) 2):int`	3, 4, APP
6.	$A \vdash$ `fn x => ((* x) 2):int -> int`	1, 5, ABS

Given that `x` is `int` (1) and `*` is `int -> int -> int` (2), `* x` must be `int -> int` (3). Given that `2` is `int` (4), `((* x) 2)` must be `int` (5). Thus, `fn x => ((* x) 2)` must be `int -> int` (6).

The key, hidden step is the initial assumption that `x` is `int`. In type inference algorithms, such steps are performed automatically by assigning new type variables to bound variables, and unifying type expressions which should have the same types to find substitutions for the type variables. *As we shall see, it is this process that renders automatic type inference unsuitable for type explanation.*

2.2 W Algorithm

Milner's W algorithm[11] remains the classic basis for implementations of polymorphic type inference. Given an expression exp and an initial assumption set A, the algorithm:

$$W(A, exp) = (T, \tau)$$

attempts to find a type τ for exp, incidentally producing a set of substitutions T. Such substitutions take the form $\{new/old\}$ and are applied to type expressions to replace all occurrences of old with new. Substitution sets may be composed RST indicating that they are to be applied in order from right to left i.e. apply all from T, then apply from S, then apply from R.

We express the W algorithm as a set of rules over the abstract syntax of our simple language.

For an identifier:

$$\begin{aligned} W(A, ident) \quad \equiv \quad & ident{:}(\forall \alpha_1 \cdots \alpha_n.\sigma) \in A \\ & \beta_1 \cdots \beta_n \text{ are new type variables} \\ & (T, \tau) = (I, \sigma\{\beta_1/\alpha 1, \cdots \beta_n/\alpha_n\}) \end{aligned}$$

its type is found from the assumption environment. Any universally quantified type variables are replaced with new unquantified variables, indicating that generic types will be instantiated to specific but as yet unknown types. No substitutions are performed. Note that this stage introduces new type variables$\beta_1 \cdots \beta_n$.

For a function:

$$\begin{aligned} W(A, \texttt{fn}\ ident \Rightarrow exp) \quad \equiv \quad & (R, \rho) = W(A.ident : \beta, exp) \\ & (T, \tau) = (R, R\beta \rightarrow \rho) \end{aligned}$$

it is assumed that the bound variable has unknown type β and a type ρ is found for the function body. The overall function type will be $\beta \rightarrow \rho$ after applying appropriate substitutions from finding the body's type to the bound variable's type. Note that this stage introduces a new type variable β and may find instantiations for it and other type variables.

For an application:

$$\begin{aligned} W(A, (exp_1\ \ exp_2)) \quad \equiv \quad & (R, \rho) = W(A, exp_1) \\ & (S, \sigma) = W(RA, exp_2) \\ & \beta \text{ is a new type variable} \\ & U = unify(S\rho, \sigma \rightarrow \beta) \\ & (T, \tau) = (USR, U\beta) \end{aligned}$$

the function expression exp_1 is analysed to return a type ρ and associated substitutions. The argument expression exp_2 is then analysed, with those substitutions applied to the assumption environment, to return a type σ and further substitutions. If it is assumed that if the function application will return a value of type β, then the function type ρ and the function from applying the argument type σ to the result type β must be the same. They are unified to find consistent substitutions for the type variables that make both expressions equal. The final type is that of the result after these substitutions. Note that this stage also introduces a new type variable β and may find instantiations for it and other type variables.

We now illustrate the W algorithm inferring the type of `fn x => ((* x) 2)`. We use the same initial assumption environment as in the example above and only show the major steps in the inference:

1. $W(A,$`fn x => ((* x) 2)`$)$
2. $W(A.\mathrm{x}{:}\ \alpha,$`((* x) 2)`$)$
3. $W(A.\mathrm{x}{:}\ \alpha,$`(* x)`$)$
4. $W(A.\mathrm{x}{:}\ \alpha,$`*`$) \Rightarrow (I,$`int -> int -> int`$)$
5. $W(A.\mathrm{x}{:}\ \alpha,$`x`$) \Rightarrow (I, \alpha)$
6. $unify($`int -> (int -> int)`$, \alpha$ `->` $\beta) \Rightarrow \{\alpha/$`int`$, \beta/$`int -> int`$\}$
7. $\Leftarrow (\{\alpha/$`int`$, \beta/$`int->int`$\},$`int -> int`$)$
8. $W(A.\mathrm{x}{:}$`int`$, 2) \Rightarrow (I,$`int`$)$
9. $unify($`int -> int, int ->` $\gamma) \Rightarrow \{\gamma/$`int`$\}$
10. $\Leftarrow (\{\alpha/$`int`$, \beta/$`int->int`$, \gamma/$`int`$\},$`int`$)$
11. $\Leftarrow (\{\alpha/$`int`$, \beta/$`int -> int`$, \gamma/$`int`$\},$`int -> int`$)$

At step 2, it is assumed that the bound variable `x` is some unknown but definite type α. At step 6, `*`'s type `int -> int -> int` is unified with the type α `->` β, which represents the type of a function which can take `x`'s type as argument, to determine that α must be `int`[1]. By step 8, when the type of `2` is to be found prior to unification with `(* x)`'s type, the substitution of `int` for α has been applied to the assumption environment: in future `x` will have type `int`.

We allege that the introduction of type variables, and the deduced and applied chains of substitution, are the causes of difficulty in using algorithms like W to explain types.

3 Human Expert Type Explanations

3.1 Introduction

In the course of teaching functional programming at undergraduate and postgraduate level, we have observed that students have a number of conceptual difficulties with polymorphic typing. These seem rooted in prior familiarity with monomorphic type schemes in imperative languages. In particular, students appear not to understand the rules governing the introduction and use of type variables, and are often surprised when an inferred type is more general

[1] And that β must be `int -> int`

or more specific type than they intended. A further difficulty arises with the scope of type variables in type expressions. In a curried function definition, each nested instance of the same bound (expression) variable denotes a new variable with a new scope and extent. In contrast, in a type expression, all occurrences of a type variable denote the same variable. Finally, accounts and implementations of polymorphic type inference often conflate quantified and unquantified type variables, using the same symbols for both and omitting quantifiers. Thus, students find it hard to distinguish generic from specific but unknown type variables. All of these difficulties conspire to complicate the interpretation of type information from type inference, especially error messages.

We decided to investigate how experts explain types to try and identify guidelines for automatic explanations that are more readily understandable, especially by novices. To that end, we conducted the following experiment to identify human strategies in explaining types[22][21].

3.2 Characterising Human Type Explanations

In comparison with the W algorithm, in our experience people appear to be far less rigid and rigorous in their checking of types. They seem to be partial, assuming a construct's type on the minimum necessary evidence. They also seem to be opportunistic, scanning a program as 2D text for sources of evidence. Finally, they seem to seek concrete evidence (e.g. constant types) before introducing type variables.

Consider the SML function to count the number of `5`s in an integer list:

```
fun count5 [] = 0 |
    count5 (5::t) = 1+count5 t |
    count5 (h::t) = count5 t;
```

The W algorithm will explore each case from left to right, working through the cases from first to last, introducing type variables for `count5`, `[]`, `h` and `t`. In contrast, we might caricature a typical human check of our example as:

- `5` is `int`, so `(5::t)` is an `int list`, so the argument is an `int list`;
- `0` is `int`, so the result is `int`.

We may note that the caricature human check concludes that every case must be `int list -> int` because the first case is `... -> int` and the second case is `int list -> ...`. It also scans down the columns of patterns as well as across each case. Finally, it focuses on the constant types of values `0` and `5`.

Thus, we hypothesised that human type checkers would:

- focus on ground or known types first;
- use vertical relationships between patterns and cases in resolving and assigning types, which we term 2D text inspection;
- only introduce type variables as a last resort;

- only partially check functions.

As we shall see, our caricature appears not to be wholly accurate.

We have carried out two experiments where experts were video taped type checking sets of SML functions, following a "speak-aloud" protocol. We have been unable to locate standard sets of type checking problems either for evaluating automated type checkers or for use with people. The questions in our experiments are drawn from the type checking problems that we set our 1st year Functional Programming students[9], augmented with additional questions. Each question consists of an untyped function definition and the subject is required to identify the type, or explain why there is a type error. Questions are all formed in a pure functional subset of SML, comprising base types, lists and tuples. The 34 questions were chosen to reflect a range of function types and may contain multiple errors.

Eight subjects took part in the type correct experiment and seven in the type error experiment. Six people took part in both experiments. The subjects all have at least post-graduate Computer Science experience and have programmed extensively in polymorphic typed languages. Furthermore, they have all implemented Hindley-Milner type checkers or worked extensively with them when implementing functional languages. Finally, they have all tutored undergraduate or postgraduate students in functional programming. Thus our small cohort may be considered experts.

Subjects were given up to 30 minutes to complete each set of questions. Initial inspection of the video taped sessions led to identification of 13 major inference techniques used during type checking, shown in Figure 2. To simplify analysis, we have grouped the techniques into four major classes.

In elaborating a *type skeleton*, subjects typically drew the gross structure of a type signature, identifying tuple and function types but omitting the fine detail. They then systematically sought to fill in that fine detail using other techniques. In identifying *concrete types*, subjects looked for constants of known type, overloaded operators or known system operators as important sites of evidence. For *2D inspection*, subjects scanned the program text in various orders reflecting type relationships required for program constructs. Finally, subjects introduced *type variables* as place holders for further refinement or as tokens for further transmission.

Each subject's attempt at each question was then analysed and sequences of the above techniques were recorded. The small number of subjects is not enough to form a basis for statistical analysis. Nonetheless, we are able to identify a number of clear trends by combining technique counts for all subjects.

3.3 Results

Subjects attempted 147 error free questions, using 1830 technique instances, and 235 error questions, using 1663 technique instances.

Figure 3 shows technique percentage use for questions without and with errors, by major technique. For questions without errors, considerable use is

Technique Class	Techniques
Type skeleton	Construct type skeleton for function Construct and refine type skeleton for local construct Refine function type skeleton
Concrete types	Locate specific types Locate overloaded operators Locate known system operators
2D inspection	Locate commonalities across patterns Analyse top-down, from nodes to leaves of abstract syntax tree Analyse bottom up, from leaves to nodes of abstract syntax tree Identify argument type from use in function body Identify body construct type from known argument type Search forwards and backwards from a known type
Type variable	Introduce type variable

Figure 2: Major human type inference techniques.

Technique Class	Error Free Use%	Error Use%
Type skeleton	32.01	24.88
Concrete types	27.05	39.34
2D inspection	35.41	34.64
Type variable	5.46	0.36

Figure 3: Technique use for questions with and without errors.

made of type signature skeletons, which we did not anticipate. As hypothesised, much use is made of concrete type information. Similarly, much use was made of 2D inspection of function text, confirming our hypothesis. Finally low use is made of type variables, as hypothesised.

For questions with errors, less use was made of type signature skeletons than for the error free problems. However, more use was made of concrete type information. Use of 2D inspection of text is similar to that for the error free questions. Type variable introduction was again the least used technique, and used far less than for the error free questions.

Contrary to our expectations, for questions without errors, full checking was carried out for 114 questions(77.55%) and partial checking for 33(22.45%). For questions with errors, there was less use of full checking than for error free questions: full checking was carried out for 145 questions(61.70%) and partial checking for 90(38.28%) questions, More detailed discussion of the experiment and further analysis of results may be found in[21][22].

3.4 Discussion

We have observed a number of differences between human type checking, as characterised by our composite expert subject group, and the W algorithm. First of all, people find the elaboration of a skeletal type during checking extremely helpful in structuring the process. Furthermore, as we hypothesised, people rely on identification of constructs of known type to guide checking and make use of the 2D textual form to locate them. Finally, as we hypothesised, people use explicit type variables as a last resort. However, our prediction of people performing partial checking was confounded.

We have not found any comparable research specifically on human type checking. However, there are interesting correspondences with work on program comprehension. In particular, the observed use of type skeletons to guide type checking may be an instance of Brooks' top-down, hypothesis driven comprehension[3]. Similarly, the observed use of concrete type features is similar to Wiedenbeck and Scholtz's identification of the use of *surface beacons* to guide successful program comprehension[16]. Wiedenbeck and Scholz note that beacon use is particularly helpful for comprehension of programs with unknown purpose: our questions are uncommented and provide minimal semantic information through meaningful identifiers.

4 A Human-like Type Explanation System

4.1 Introduction

We are developing a prototype "human-like" type explanation system, which has been strongly informed by the experiments discussed above. Our objectives have been to:

- provide succinct explanations;

- base explanations on concrete type information as far as possible;
- use 2D examination of functions, down all patterns and all results for all cases, as well as across the pattern and result for an individaul case;
- minimise repetition in explanations.

We differ from the human inference techniques, discussed in the previous section, in not giving exhaustive explanantions based on a whole function and in not explicitly using a skeleton to present the explanation. The resulting explanations, as discussed below, are very different to those derived directly from the W algorithm and equivalents, but feel similar to those we observed in human type inference.

Our system produces type explanations in two stages. First of all, a modified type inference algorithm is applied to a function to construct an annotated type tree, which records a history of the inference of each node. The annotated type tree is then traversed to generate an explanation, using rules to ensure succinct, non-repetitive explanations. These stages are now discussed.

4.2 Type Checking for Human-Like Explanations

Our system provides explanations for a subset of SML including: `int`, `real`, `bool`, `string` and corresponding operators; tuples; lists; conditions; local definitions; recursive functions; patterns.

Given a set of functions whose types are to be explained, each is parsed and an abstract syntax tree (AST) is constructed. The AST is then type checked using the Unification of Assumptions Environment algorithm (UAE).

The UAE is closely related to the W algorithm and was motivated by an informal suggestion by Johan Agat[1]. Essentially, the UAE types each use of a program variable to produce an assumption environment. The assumption environments are unified to ensure consistency.

In the W algorithm, types are infered from left to right through expressions. When conflicts are found in later parts of expresssions they are not related back to the earlier parts. In the UAE, because every use of a program variable is typed and unified, the left to right bias of the W algorithm is avoided. Full details appear in [23, 24]. An interactive type exploration environment based on a similar algorithm to UAE has also been developed independently by Huch, Chitil and Simon[8].

During the application of the UAE to an AST, an annotated type is generated for each abstract syntax construct. The annotation indicates which rule was used to form it and the piece of AST it corresponds to. The annotation also lists the assumption environments that use contributed to the original type's inference. These assumption enviroments in turn contain annotated types with references to further assumption environments. Thus, the cyclic graph of assumption environments records a type inference history which is used subsequently to generate explanations. When a program variable is reached during AST traversal, it is augmented with assumption environment associating it with

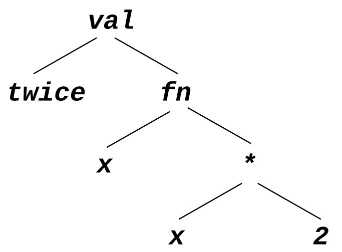

Figure 4: AST for `val twice = fn x => ((* x) 2)`

Rule annotation	Details
APP	function application
CONS	list concatenation
FN-L	function left hand side
FN-R	function right hand side
OP-*op*	expression with system operator *op*
OP-L	system operator left hand side
OP-R	system operator right hand side
VAL	`val` definition
VAL-L	`val` left hand side
VAL-R	`val` right hand side

Figure 5: Rule annotations in type nodes.

a new type variable, annotated with that program variable. When a value of known type is reached, that type is noted, annotated with that value.

Once all branches of an abstract syntax construct have been visited, the corresponding node's type is found through application of the Hindley-Milner rules above to unify the child nodes' types. The node's final assumption environment is then formed by combining and unifying its and the child nodes' assumption environments. When an operator of known type is reached, an operator-specific rule is applied to unify the operand types and assumption environments.

Consider once again our example `val twice = fn x => ((* x) 2)` with the AST shown in Figure 4. The AST is descended to build an equivalent type tree that lists for each AST node the rule that applies, an initial type variable or known type, and the corresponding sub-tree.

In an annotated type tree, each node is annotated as:

rule:*type* - *AST node*
assumption environment

Here, *rule* indicates which UAE rule was invoked. Those required for our examples are shown in Figure 5. *type* indicates the node's type. *AST node* is a textual representation of the node in the AST from which the type tree node was derived. *assumption environment* is the node's assumption environment if any. Figure 6 shows the type tree for the above example after AST descent.

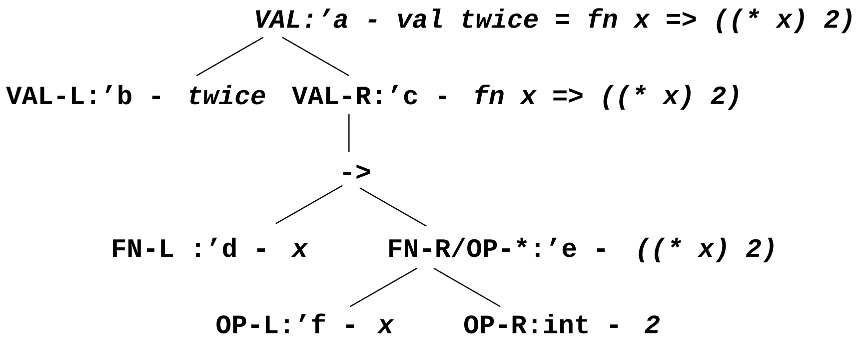

Figure 6: Initial annotated type tree for `val twice = fn x => ((* x) 2)`

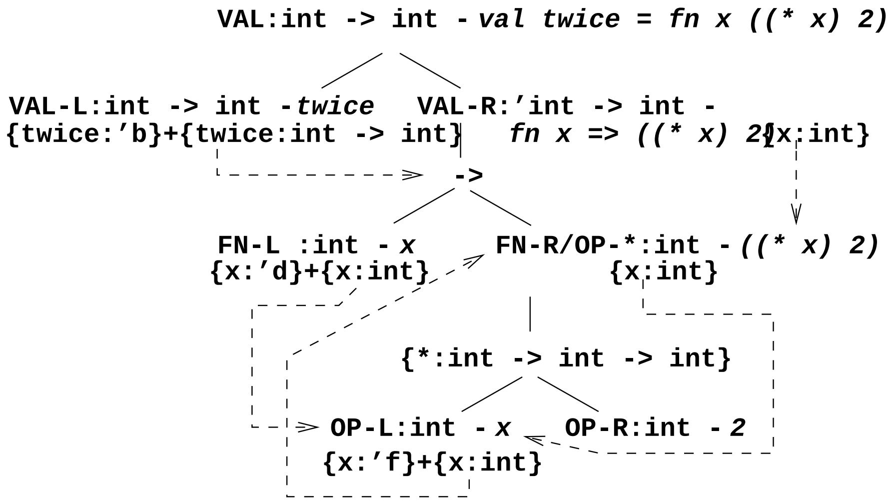

Figure 7: Complete annotated type tree for `val twice = fn x => ((* x) 2)`

The type tree is then ascended, constructing the full type for each node. The node's type variable is instantiated using an appropriate rule to combine the types from sub-nodes. The assumption environments for the contributing nodes are also recorded. Figure 7 shows the type tree for the example after type construction. Note that, for brevity, the initial assumption environment is not shown on each node.

We treat recursive functions defined using the SML `fun` form as a special case, to facilitate 2-dimensional analysis. For a recursive function, a matrix is constructed with a row for each function definition case, and a column for each curried argument and the result. During tree descent, each matrix entry is annotated as for type tree nodes. During tree ascent, the types down each column are unified and the assumption environments across each row are unified and combined.

	left hand side pattern	right hand side expression
case 1	**CONS:'a list** ***[]***	**int** ***0***
case 2	**CONS: int list** ***(5::t)*** **{t:int list}**	**OP-+:int** ***1+count5 t*** **{t:'c, count5:'c -> int}**
case 3	**CONS: 'b list** ***(h::t)*** **{h:'b,t:'b list}**	**APP: 'e** ***count5 t*** **{t:'d, count5:'d -> 'e}**

Figure 8: Initial annotated type matrix for `count5`.

For the function to count how often an integer list contains a `5`:

```
fun count5 [] = 0 |
    count5 (5::t) = 1+count5 t |
    count5 (h::t) = count5 t
```

the initial matrix has the form shown in Figure 8. After type resolution the matrix has the form shown in Figure 9. Type annotation links in a matrix element are omitted when they originate from a subtree in that element.

4.3 Generating Explanations for Human-Like Type Checking

To generate an explanation from an annotated type tree, the tree is traversed and the node's annotated assumption environment, recording the history of the inference of that node's type, is inspected. However, as all contributing assumption environments are recorded for each node, there may be considerable repetition of information and circular chains of annotations. Thus, our system first attempts to remove redundant information, and to build a reduced tree where each node has one, non-circular explanation.

If a program variable appears in multiple assumption environments then all ocurrences are checked for consistency and the shortest chain of annotations leading to a fully instantiated type is selected. Furthermore, if a chain of annotations for a program variable includes a cycle back to that variable, then the cycle is eliminated by removing the highest level link in the annotated type tree.

For example, in Figure 7, there is a cycle linking `x:int` in `((* x) 2)` through the node for `x` back to the node for `((* x) 2)`. We remove the link from `((* x) 2)` to `x`. The reduced annotation tree is shown in Figure 10.

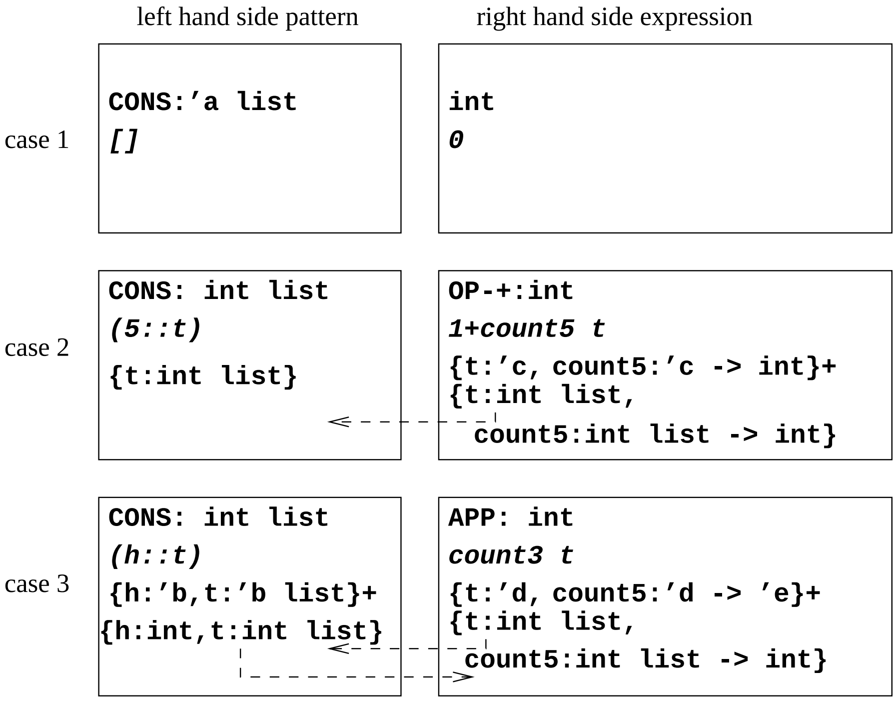

Figure 9: Complete annotated type matrix for `count5`.

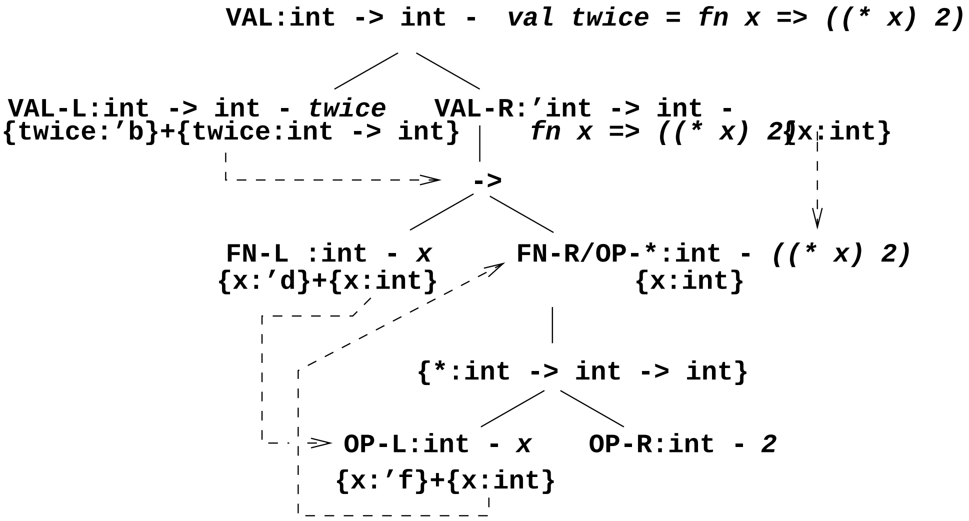

Figure 10: Reduced annotated type tree for `val twice = fn x => ((* x) 2)`

The final explanations are structured by the top level function type and are generated systematically from left to right, from argument types to result types. Alternatives would be to start with a simplest type or to allow the user to select top-level types of interest. The gross structure of an explanation is determined by the rule annotation on a node, with the fine detail provided by the history chains of assumption environments. During explanation generation, if a sub-explanation has already been generated then a reference is made back to that sub-explanation.

For our SML subset, there are 40 rules for explanation corresponding to 40 distinctive sites of type inference. Rather than list them all here, we discuss the final explanation for `val twice = fn x => x*2` which is:

```
"twice" and "fn x => x * 2" have the same type
 - LHS and RHS of "val twice = fn x => x * 2"
1: "fn x => x * 2", is a function type: int -> int
Argument 1: int
2:  the Bound variable "x" and "2" have the same type in "x * 2"
    - arguments/result of "*" have the same Number types
3: "2": int
Result: int
4:  Function body "x * 2": Int
    - result/arguments of "*" have the same Number types
    - see 2 above
```

Initially, it is observed that both sides of a `val` definition have the same type and the right hand side is chosen to give the explanation. At step 1, the final type for the right hand side function `fn x => x*2` is given. The argument is explained by exploring the use of the bound variable `x` in the body `x*2`. At step 2, it is observed that operands `x` and `2` of `*` have the same numeric type. Step 3 notes that `1` is an `int`. Hence, the argument `x` is `int`. Step 4 refers back to step 2 to justify the observation that the function body `x*2` has type `int`.

For example, the full explanation for the function that counts `5`s in a list is:

```
"fun count5 [] = 0
  |  count5 (5 :: t) = 1 + (count5 t)
  |  count5 (h :: t3) = count5 t3"
      is a function type: int list -> int
Argument 1: int list
1: "[]", "(5 :: t)" and "(h :: t3)" have the same type
   - same LHS pattern position
2: list of "5", "t" and "(5 :: t)" have same type
    -operands/result of "::"
     3: "5":  int
4:  list of "5": int list
Result: int
5: "0", "1 + (count5 t)" and "count5 t3" have the same type
   - same RHS result
```

```
6: "0": int
```

Note that the identifier `t` in the third definition case is numbered to distinguish it from that in the first case.

Step 1 notes that the left hand side patterns all have the same type. At step 2 it is found that the second case `(5::t)` has the shortest history and it is observed that a list containing `5`, the tail `t` and the whole list must have the same type. At step 3 it is found that `5` has the shortest history and that it is `int`. Thus, step 4 decides that the whole list must be `int list`, and hence the function argument must be `int list`. Step 5 notes that the right hand side result expressions must all have the same type. At step 6 it is found that `0` has the shortest history and that it is `int`. Thus it is concluded that the function result is `int`.

Further examples are given in the Appendix.

5 Related Work

5.1 Explanation Systems

Several type explanation systems have been based on tracing the behaviour of the W algorithm.

Soosaipillai's system [17] explains type inference by menu traversal. The system uses a global list to record the subexpressions and their inferred types as a global history list which contains the result types when inference of each sub-expression finishes. This information can be presented as a menu to enable traversal of the global list.

The system's strengths are that the user can ask `why` to indicate which site needs explanation, an explanation does not instantiate unnecessary local type variables and an explanation is purely in terms of the types of corresponding sub-expressions. However, it is hard to explain types using the method. The explanation is top-down so a user needs to remember all the steps of the inference to understand the full explanation of the required type.

Duggan and *Bent*'s system system explains type inference by analysing the instantiation of type variables during unification. A modified unification algorithm is used to record all instantiations of type variables [5]. The approach is similar to that used by Wand[20] to identify the sources of type errors. Other systems use similar unification algorithms to give type inference explanation, for example Rideau and Thery's programming environment[15].

The system uses the essentially unchanged W algorithm to walk over the abstract syntax tree for a program collecting equality constraints on the types of the program variables. The modified unification algorithm solves the equality constraints and collect the explanations. Like other explanation systems, the system records the program fragment which gave rise to that constraint, with the corresponding instantiation.

The resulting explanation is simpler than that from the direct trace of the the W algorithm, the technique used by Beaven's [2] discussed below. The explanation is closer to how people understand type inference but is still not very easy to understand. The aliasing generates local type variables which do not relate to the program itself. Furthermore, a type is represented as an annotated graph which quickly becomes lengthy, with similar problems to Soosaipillai's approach.

Beaven and *Stansifer*'s system [2] is intended for error explanation and illustrates how inference reaches two conflicting types. The system maintains the deductive steps of type inference to explain the reasoning of the type reconstruction process in relation to the program. It uses two function `Why` and `How` to explain how the type of an expression is determined by the type of its subexpressions, and how the types of those subexpressions are inferred. An explanation is generated by traversing an abstract syntax from the root to the leaves, and then back to the root. Thus explanations are essentially W algorithm traces.

The system's strengths are that there are no unnecessary local type variables in the explanations, unlike Duggan and Bent's system, explanations are given in English and the system can explain type errors. Although the explanations are bottom-up, and tend to be based on fine granularity information, a user still needs to remember all the steps of the inference. Furthermore, explanations tend to be lengthy and repetitive.

Rittri [14] has suggested an interactive type debugger which can be tied to any inference algorithm. This would allow programmers to explore the type of any subexpression and to ask about the origin of a specific constructors.

While the type debugger can determine the type of every subexpression and its derivation, it has no information about the programmer's intentions and mistaken beliefs. Thus, it cannot decide by itself how much to tell and if it tells all it knows at once, the programmer will be overwhelmed. The intention is that the programmer can identify those parts of a complex type that are surprising, and only these need to be explained. However, as with Soosaipillai's system, it is hard to identify a methodology for exploring type decisions. The style of a type explanation is very similar that in Duggan and Bent's system, which includes the instantiation of type variables. To understand such explanations, one needs to understand unification and the type system.

While explanation systems can be made to be correct and accurate, they fail on several key points. First of all, the explanations tend to be counterintuitive. They make substantial use of internal type variables as bridges between instances, since these are the sites of refinement through substitution. In contrast, a programmer is not concerned with these type variables. To understand the explanations, it is necessary to remember from which program variable the type variable is inferred and where it is refined. If there are more than a few type variables, it becomes very hard to remember what entity a type variable represents.

Furthermore, the explanations are not succinct. The textual explanations have so much information that they become wordy and diffuse. Experts usually find this explanation too detailed to be of real help, though they can find valuable information about the different positions in the programs that contribute to the type given by such systems [15].

Finally, the explanations are not source-based, using text generated from the abstract syntax tree, rather than the original program text. In many language implementations, a program is converted to a simpler "core" representation prior to type checking, which bears little relationship to the original source, as in our running example above.

We think that these difficulties arise directly from the realisation of the Hindley-Milner type inference scheme in algorithms like W for efficient computer use. Attempts to generate human readable type explanations are necessarily constrained by sticking rigidly to traces from such algorithms.

6 Conclusions

We have constructed a type explanation system for a simple, pure functional, list-based subset of SML, influenced strongly by our experimental investigation of expert human type explanation. Our system will also explain type errors, by providing explanations of conflicting sites of unification found by the UAE algorithm.

We next wish to extend our approach to a more substantial functional language fragment, including user defined types and abstract types, and ultimately classes or modules. However, as our experience with let-polymorphism suggests, there is a considerable increase in the volume of annotated type information as the complexity of constructs increases. Consequently, we anticipate the need to investigate more elaborate rules and techniques for explanation simplification and generation.

Given a "human-like" type explainer for a fuller language, it would be interesting to provide it as a front end to a common implementation, and to evaluate its usability, especially with naive users. It would also be interesting to explore its use in interactive type explanation, where a user interactively browses a program in a GUI environment to highlight program constructs for which explanation is required.

Finally, the human-like approach to explanation should be formalised and proved consistent, including soundness and completeness, with respect to Hindley-Milner type inference.

Acknowledgments

We wish to thank Joe Wells for his collaboration on the UAE algorithm, and all our colleagues who took part in our expert type checking experiments.

Yang Jun wishes to thank ORS and Heriot-Watt University for continuing support.

Appendix - Further Examples

Count how often f is true for list elements

```
"fun countf f [] = 0
  |  countf f2 (h :: t) =
      if f2 h then 1 + (countf f2 t) else countf f2 t", is
a function type: ('a -> bool) -> 'a list -> int
Argument 1: 'a -> bool
1: "f" and "f2" have the same type
   - same LHS pattern position
     2: "f2": is a function type: 'a -> bool
     Argument 1: 'a
     3:  the Operand of "f2 h",
         is the same type as "h": 'a - no operation on "h"
     Result: bool
     4:  "f2 h" has the same type as the condition "f2 h",
         has the same type as
     5:  bool
Argument 2: 'a list
6: "[]" and "(h :: t)" have the same type
   - same LHS pattern position
7: list of "h", "t" and "(h :: t)" have same type
   - operands/result of "::"
     8: "h": 'a - no operations on "h"
9:  list of "h": 'a list

Result: int
10: "0" and " if f2 h then 1 + (countf f2 t) else countf f2 t"
    have the same type
    - same RHS result
11: "0": int
```

Count negative, zero and positive list elements

```
"fun countsign (n, z, p) [] = (n, z, p)
  |  countsign (n2, z2, p2) (0 :: t) = countsign(n2, z2 + 1, p2) t
  |  countsign (n3, z3, p3) (h :: t3) =
      if h < 0
      then countsign(n3 + 1, z3, p3) t3
      else countsign(n3, z3, p3 + 1) t3",
 is a function type: int * int * int -> int list -> int * int * int
Argument 1: int * int * int
1: "(n, z, p)", "(n2, z2, p2)" and "(n3, z3, p3)" have the same type
   - same LHS pattern position
2:  is a Tuple type: int * int * int
```

```
Tuple Element 1: int
  3: for "n3" in "(n3, z3, p3)"
  4: "n3" and "1" have the same type in "n3 + 1"
     - arguments/result of "+" have the same Number types
  5: "1": int
Tuple Element 2: int
  6: for "z2" in "(n2, z2, p2)"
  7: "z2" and "1" have the same type in "z2 + 1"
     - arguments/result of "+" have the same Number types
  8: "1": int
Tuple Element 3: int
  9: for "p3" in "(n3, z3, p3)"
  10: "p3" and "1" have the same type in "p3 + 1"
     - arguments/result of "+" have the same Number types
  11: "1": int

Argument 2: int list
12: "[]", "(0 :: t)" and "(h :: t3)" have the same type
     - same LHS pattern position
13: list of "0", "t" and "(0 :: t)" have same type
     - operands/result of "::"
     14: "0":  int
15:  list of "0": int list

Result: int * int * int
16:  at Row 1 "(n, z, p)" is the same as Argument1 "(n, z, p)",
     has type: int * int * int - see 1 above
```

References

[1] Agat, J. and Gustavsson, J. *Personal Communication*. Chalmers University of Technology, Göteborg, June, 1999.

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

[3] Brooks, R. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18, 543–554, 1983.

[4] Burstall, R.M., MacQueen, D.B. and Sanella, D.T. HOPE: an experimental applicative language. CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[5] Duggan, D. and Bent, F. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[6] Field, A.J. and Harrison, P.G. *Functional Programming.* Addison-Wesley, 1988.

[7] Hughes, J. and Peyton Jones, S. (eds). Haskell 98: a non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, 1999.

[8] Huch, F., Chitil, O. and Simon, A. Typeview: a tool for understanding type errors. In *Proceedings of 12th International Workshop on Implementation of Functional Languages*, M. Mohnen and P. Koopman (eds), Aachner Informatik-Berichte, 63–69, 2000.

[9] Michaelson, G. *Elementary Standard ML.* UCL Press, 1995.

[10] Miller, D. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1:4, 1991, 497 – 536

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

[12] Milner, R., Tofte, M. and Harper, R. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[13] Morrison R., Brown F., Connor R and Dearle. A. *The Napier88 Reference Manual.* Persistent Programming Research Report 77, University of St Andrews/University of Glasgow, July 1989.

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

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

[16] Wiedenbeck, S. and Scholz, J. Beacons: a knowledge structure in program comprehension. In G.Salvendy and M.J.Smith, editors, *Designing and using human-computer interfaces and knowledge based systems*, pages 82–87, 1989.

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

[18] Strachey, C. Fundamental Concepts in Programming Languages. In *Notes for the International Summer School in Computer Programming*, Copenhagen, 1967.

[19] Turner, D. Miranda: a non-strict functional language with polymorphic types. In *Proceedings of Functional Programming Languages and Computer Architecture*, J.P.Jouannaud (Ed), Springer-Verlag, LNCS Vol. 201, 1985.

[20] Wand, M. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43. ACM Press, January 1986.

[21] Yang, J., Michaelson, G. and Trinder, P. Helping Students Understand Polymorphic Type Errors. *1*st Annual Conference of the LSTN Centre for Information and Computer Sciences, S. Alexander et al (Editors), LTSN-ICS, University of Ulster, pp11-19, August 2000

[22] Yang, J., Michaelson, G. and Trinder, P. How do people check polymorphic types? In A.F. Blackwell and E. Bilotta, editors, *Twelfth Annual Meeting of the Psychology of Programming Interest Group Proceedings*, pages 67–77, Cosenza, Italy, April 2000. Memoria.

[23] Yang, J. Explaining type errors by finding the source of a type conflict. In G. Michaelson, P. Trinder and H-W. Loidl, editors, *Trends in Functional Programming*, pages 58–66, Intellect, 2000.

[24] Yang, J., Wells, J., Trinder, P. and Michaelson, G. Improved type error reporting. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86, Aachner Informatik-Berichte, 2000