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

Polymorphic types in programming languages facilitate code reuse, increase reliability and reduce semantic errors in programs. Hindley-Milner type inference forms a strong basis for checking polymorphic types but is less well suited to explaining them, as it introduces intermediate constructs that relate poorly to a programmer's understanding of the program. We report an experiment into expert human type explanation, and uncover a simple set of rules for human-like explanations. We present a type explanation system based on these rules rather than Hindley-Milner inference. The system uses a new $\mathcal{H}$ inference algorithm to annotate types with explanations and is designed to produce succinct, non-repetitive 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 with single inheritance, 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 [1] 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 contribution 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 [2] have most commonly been incorporated into functional languages, for example Standard ML(SML)[3], Miranda[4], Hope[5] and Haskell[6].

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. As ever there are exceptions to this rule including the addition of generics to object oriented languages like Java [7, 8, 9]; the Napier imperative persistent database language [10]; and λ-Prolog[11], a strongly typed Prolog variant used for theorem proving.

We suspect that this apparent resistance to including polymorphic type systems in contemporary languages 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.

$x \in \text{ExpVar}$		expression variable
$e \in \text{Exp}$	$::= x$	identifier
	$\mid \lambda x \; . \; e$	lambda abstraction
	$\mid e_1 e_2$	application
	$\mid \texttt{let } x = e_1 \texttt{ in } e_2$	let binding

Figure 1: Exp_1 Abstract syntax.

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 type inference for parameterised polymorphism and its limitations for providing type explanations to people. In section 3, we report an experiment into how human experts explain types, and distill a simple set of rules for human-like type explanation. The key rules focus on concrete types, and two dimensional analysis of function definition patterns. In section 4, we present our human-like type explanation system based on these rules. The system uses a new type inference algorithm, termed the $\mathcal{H}$ algorithm, to annotate types with information about how they were inferred. The type inference history is recorded in a graph based on the expression's abstract syntax tree, and the graph is traversed to generate succinct, non-repetitive explanations. 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 introduce a very simple functional language, Exp_1, in Figure 1 to form the basis for a presentation of type inference. To illustrate inference we will use a simple Exp_1 function λ`x.((* x) 2)`, that doubles it's argument `x`. Note that we introduce "constant" identifiers:

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

A *type* is represented by a type expression, and Figure 2 shows that in Exp_1 a type, τ, is either a base type, a type variable, or a function type. Base types are integer or boolean. A type variable, α, represents any type. The type of a function is a mapping from an argument type to a result type. A *type scheme* represents an abstracted polymorphic type where the type 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`

$\tau \in \text{Type} ::= \texttt{int} \mid \texttt{bool}$	integer and bool
$\mid \alpha$	type variable
$\mid \tau \rightarrow \tau$	function type
$\sigma \in \text{TypeScheme} ::= \tau \mid \forall \alpha_1 \cdots \alpha_n . \tau$	type scheme
$TE \in \text{TypeSchemeEnv} = \text{ExpVar} \xrightarrow{fin} \text{TypeScheme}$	type environment

Figure 2: Types, Type Schemes and Type Environments.

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

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 or *type environments*, TE, each a set of identifier/type associations, as shown in Figure 2. Premises and conclusions take the general form:

$$TE \vdash e : \tau$$

That is, from prior assumptions TE we may infer that expression e has type τ.

The rules for inferring polymorphic types for expressions in $\text{Exp}_1$are given in Figure 3, after Cardelli[12].

For VAR, if the assumptions TE include the association of identifier x with type(scheme) σ, then we may infer that x has type σ, i.e. we may "look up" an identifier in a set of assumptions to find the associated type.

For ABS, if with assumptions TE, augmented with the association of identifier x and type τ_1, we can show that expression e has type τ_2, then the function $\lambda x \,.\, e$ has type $\tau_1 \rightarrow \tau_2$, i.e. takes an argument of type τ_1 and returns a result of type τ_2.

For APP, if e_1 has type $\tau_1 \rightarrow \tau_2$, and e_2 has type τ_1, then applying e_1 to e_2 returns a value of type τ_2.

For LET, if e_1 has polymorphic type(scheme) σ, and with TE augmented with the association of identifier x and σ, we can show that expression e_2 has type τ, then $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ has type τ.

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

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

TE = `{2:int, *:int -> int -> int, x:int }`

[VAR] variables $TE \oplus \{x : \sigma\} \vdash x : \sigma$

[ABS] abstraction $\dfrac{TE \oplus \{x : \tau_1\} \vdash e : \tau_2}{TE \vdash \lambda x\ .\ e\ :\ \tau_1 \rightarrow \tau_2}$

[APP] application $\dfrac{TE \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad TE \vdash e_2 : \tau_1}{TE \vdash (\ e_1\ \ e_2\ )\ :\ \tau_2}$

[LET] let binding $\dfrac{TE \vdash e_1 : \sigma \quad TE \oplus \{x : \sigma\} \vdash e_2 : \tau}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$

[GEN] generalisation $\dfrac{TE \vdash e : \sigma \text{ - } \alpha \text{ not free in } TE}{TE \vdash e : (\forall \alpha . \sigma)}$

[SPEC] specialisation $\dfrac{TE \vdash e : (\forall \alpha . \sigma)}{TE \vdash e : [\tau / \alpha]\sigma}$

Figure 3: Hindley Milner inference rules for Exp_1

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.	$TE \vdash$ `x:int`	VAR
2.	$TE \vdash$ `*:int -> int -> int`	VAR
3.	$TE \vdash$ `(* x):int -> int`	1, 2, APP
4.	$TE \vdash$ `2:int`	VAR
5.	$TE \vdash$ `((* x) 2):int`	3, 4, APP
6.	$TE \vdash$ `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[2] remains the classic basis for implementations of polymorphic type inference. Given an expression e and an initial assumption set TE, the algorithm:

$\mathcal{W}$: TypeSchemeEnv $\times$ Exp $\rightarrow$ Substitution $\times$ Type

$$
\begin{array}{ll}
\mathcal{W}(TE, x) \equiv & \forall \alpha_1 \cdots \alpha_n . \tau = TE(x) \\
 & (I, \{\beta_i/\alpha_i\}\tau) \text{ - } \beta_i \text{ new} \\
\mathcal{W}(TE, \lambda x\ .\ e_1) \equiv & (S_1, \tau_1) = \mathcal{W}(TE \oplus \{x : \beta\}, e_1) \\
 & (S_1, S_1\beta \rightarrow \tau_1) \\
\mathcal{W}(TE, e_1 e_2) \equiv & (S_1, \tau_1) = \mathcal{W}(TE, e_1) \\
 & (S_2, \tau_2) = \mathcal{W}(S_1(TE), e_2) \\
 & S_3 = UnifyTypes(S_2\tau_1,\ \tau_2 \rightarrow \beta) \text{ - } \beta \text{ new} \\
 & (S_3 S_2 S_1, S_3\beta) \\
\mathcal{W}(TE, \text{let } x = e_1 \text{ in } e_2) \equiv & (S_1, \tau_1) = \mathcal{W}(TE, e_1) \\
 & (S_2, \tau_2) = \mathcal{W}(S_1(TE) \oplus \{x : \forall \alpha_1 \cdots \alpha_n . \tau_1\}, e_2) \\
 & \text{- } \alpha_i \text{ free in } \tau_1,\ \alpha_i \text{ not in } S_1(TE) \\
 & (S_2 S_1, \tau_2)
\end{array}
$$

Figure 4: $\mathcal{W}$ Type Inference Algorithm for Exp_1

$$W(TE, e) = (T, \tau)$$

attempts to find a type τ for e, incidentally producing a set of substitutions T. A *substitution* has the form $\{new/old\}$ and is applied to type expressions to replace all occurrences of *old* with *new*. New bindings can be added with $\oplus$ and removed with $\ominus$. Substitutions R, S and T 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 all from S, then apply all from R.

Figure 4, after Field and Harrison [13], defines the W algorithm on the abstract syntax of Exp_1, omitting cases where inference may fail, and writing β_i for new type variables. We write I for the identity substitution which has no effect on type expressions.

An identifier's type is inferred 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$\beta_1 \cdots \beta_n$.

A function's type is inferred in two steps. Substitutions and a type, (S_1, τ_1) are found for the function body e, using an environment augmented with a binding of the function parameter, x, to a new type variable, β. The function type is $\beta \rightarrow \tau_1$ after applying the substitutions from finding the body's type to the parameter type. Note that this stage introduces a new type variable β and may find instantiations for it and other type variables.

The type of a function application is inferred in four steps. The function expression e_1 is analysed to return a type τ_1 and associated substitutions. The argument expression e_2 is then analysed, with those substitutions applied to the assumption environment, to return a type τ_2 and further substitutions. If the function's result has type β, then the function type τ_1 must have the same type as a function from τ_2 to β, and they are unified to find consistent substitutions. The result is the composition of all substitutions and the application of the third substitution to β.

The type of a let definition is inferred in three steps. The defining expression e_1 is analysed to return a type τ_1 and associated substitutions. The modified assumption environment is updated with an association between the defined variable x and a universally quantified form of that type. Analysis of the result expression e_2 produces a further substitution and a type τ_2. Finally, the composition of the substitutions, and the result expression's type are returned.

We now illustrate the W algorithm inferring the type of λ`x.((* x) 2)` using an assumption environment with bindings for the two constants:

$$TE' = \{\texttt{2:int, *:int -> int -> int}\}$$

Showing only the major steps in the inference, we have:

1. $W(TE', \lambda\texttt{x.((* x) 2)})$
2. $W(TE' \oplus \{\text{x: } \alpha\}, \texttt{((* x) 2)})$
3. $W(TE' \oplus \{\text{x: } \alpha\}, \texttt{(* x)})$
4. $W(TE' \oplus \{\text{x: } \alpha\}, \texttt{*}) \Rightarrow (\{\}, \texttt{int -> int -> int})$
5. $W(TE' \oplus \{\text{x: } \alpha\}, \texttt{x}) \Rightarrow (\{\}, \alpha)$
6. $unify(\texttt{int -> (int -> int)}, \alpha \texttt{ -> } \beta) \Rightarrow \{\texttt{int}/\alpha, \texttt{int -> int}/\beta\}$
7. $\Leftarrow (\{\texttt{int}/\alpha, \texttt{int->int}/\beta\}, \texttt{int -> int})$
8. $W(TE' \oplus \{\text{x:}\texttt{int}\}, 2) \Rightarrow (\{\}, \texttt{int})$
9. $unify(\texttt{int -> int, int -> } \gamma) \Rightarrow \{\texttt{int}/\gamma\}$
10. $\Leftarrow (\{\texttt{int}/\alpha, \texttt{int->int}/\beta, \texttt{int}/\gamma\}, \texttt{int})$
11. $\Leftarrow (\{\texttt{int}/\alpha, \texttt{int -> int}/\beta, \texttt{int}/\gamma\}, \texttt{int -> int})$

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 strict left to right and bottom-up resolution of type variables as place holders for unknowns, 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

[1] And that β must be `int -> int`

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 or more specific 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 an identifier denote the same type variable. For example, consider:

```
fun inclast a a a = a+1
val inclast = fn : 'a -> 'b -> int -> int
```

This function looks as if it has three formal parameters all called `a`. In fact, this is a shorthand for:

```
val inclast = fn a => fn a => fn a => a+1
```

that is, a curried function of one unknown `a`, whose body is a function of another unknown `a`, whose body is a function of an integer `a` to which 1 is added. When `inclast` is applied to three arguments, the first two are discarded and the incremented value of the third is returned.

Now consider:

```
fun join3 a b c = [a,b,c];
val join3 = fn : 'a -> 'a -> 'a -> 'a list
```

Here, the three curried parameters `a`, `b`, and `c` are joined together to form a list. Thus, they must all have the same unknown type `'a`.

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[14][15].

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 extended to deal with patterns 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 use the following strategies.

- 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. Examples of partial checking include locating just one error in an expression with multiple type errors, and deducing type information from a single value, e.g. that the result of `count5` is `int` from the `0` in the first pattern.

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[16], 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 two sets of 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

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 5: Major human type inference techniques.

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 5. 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.

Technique Class	Error-free Use%	Error Use%
Type skeleton	29.6	18.8
Concrete types	28.4	44.2
2D inspection	36.9	34.9
Type variable	5.1	2.1

Figure 6: Technique use for questions with and without errors.

Figure 6 shows technique percentage use for questions without and with errors, by major technique. For questions without errors, considerable use is 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(77.6%) and partial checking for 33(22.5%). For questions with errors, there was less use of full checking than for error free questions: full checking was carried out for 145 questions(61.7%) and partial checking for 90(38.3%) questions, More detailed discussion of the experiment and further analysis of results may be found in[14][15].

3.4 Discussion

We have observed a number of differences between human type checking, as characterised by our 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[17]. Similarly, the observed use of concrete type features is similar to Wiedenbeck and Scholz's identification of the use of *surface beacons* to guide successful program comprehension[18]. 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 individual case;
- minimise repetition in explanations.

We differ from the human inference techniques, discussed in the previous section, in not giving exhaustive explanations 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. The algorithm is clearly more computationally expensive than the W algorithm, but we argue that the improved quality of the explanations justifies the additional computation.

We present our type explanation system in three stages. Firstly we introduce our new $\mathcal{H}$ type inference algorithm that annotates types with information about how they were inferred. Secondly, we illustrate how the $\mathcal{H}$ algorithm is used to construct an annotated type graph based on the abstract syntax tree, recording a history of the inference of each node. Thirdly, we discuss how the annotated abstract syntax tree is traversed to generate an explanation, using rules to ensure succinct, non-repetitive explanations.

4.2 The $\mathcal{H}$ Algorithm

Our human subjects used techniques to explain types that cannot be illustrated using a language as simple as Exp_1 from section 2.1. For example the experts analysed the 2 dimensional structure of patterns, and properties of overloaded operators. Figure 7 gives the abstract syntax of a more sophisticated language, Exp_2, used by our explanation system. The new language is a subset of SML including patterns, tuples, lists, conditionals, local definitions, recursive functions, and base types: `int`, `real`, `bool`, `string` with corresponding operators. Operators may be overloaded, e.g. `+` may be applied at `int` or `real`. The syntax is unusual in making pattern matching function definitions explicit.

$\text{Exp}_2 ::= \text{val } x = e$		
$x \in \text{ExpVar}$		identifier
$c \in \text{Constant}$	$= \text{Int} \cup \text{Real} \cup \text{String} \cup \text{Bool}$	int, real, string, boolean constants
$e \in \text{Exp}$	$::= c$	constant
	$\mid +$	primitive operation
	$\mid x$	identifier
	$\mid e_1 e_2$	application
	$\mid$ `if` e `then` e_1 `else` e_2	condition
	$\mid$ `let` $x = e_1$ `in` e_2	local definition
	$\mid$ `val` $x = e$	definition
	$\mid (e_1, e_2)$	tuple
	$\mid (e_1 :: e_2)$	list
	$\mid []$	empty list
	$\mid$ `fn` $\{m\}$	lambda abstraction
	$\mid$ `fun` d_x	pattern function
$p \in \text{Patt}$	$::= x$	identifier
	$\mid c$	constant
	$\mid (p, p')$	tuple
	$\mid []$	empty list
	$\mid p :: p'$	list
$m \in \text{Match}$	$::= p \Rightarrow e$	match
$d_x \in \text{PattDefn}$	$::= \{x \ \{p\} = e\}$	pattern function definition

Figure 7: Exp_2 Abstract Syntax

The richer set of types for Exp_2 are given in Figure 8. To support overloaded numeric types we distinguish numeric from ordinary type variables; likewise to support quantified types we distinguish quantified from ordinary type variables. Type schemes, type environments and substitutions are unchanged from Exp_1The annotated types are central to our explanation system, and form a graph as the type component τ may contain other annotated types. There are a total of 40 annotation rules corresponding to 40 distinctive sites of type inference. A complete list of rules can be found in [19]. Rather than reproduce them here, we include only the annotations used in the following examples.

An Exp_2 expression whose types are to be explained, is parsed and an abstract syntax tree (AST) is constructed. The AST is then typed using the Human-like or $\mathcal{H}$ algorithm, which is based on the Unification of Assumption Environment algorithm ($\mathcal{U}_{\mathbf{AE}}$). The $\mathcal{U}_{\mathbf{AE}}$ is closely related to the W algorithm and was motivated by an informal suggestion by Johan Agat[20]. Essentially, the $\mathcal{U}_{\mathbf{AE}}$ types each use of a program variable to produce an assumption, or type, environment. The type environments are unified to ensure consistency. In the W algorithm, types are inferred from left to right through expressions. When conflicts are found in later parts of expressions they are not related back to the earlier parts. In the $\mathcal{U}_{\mathbf{AE}}$, because every use of a program variable is

$\gamma \in$ QTyVar	quantified type variables
$\delta \in$ NumTyVar	type variables for number
$\alpha, \beta \in$ TyVar	ordinary type variables
$\nu \in$ NumType ::= `int` $\mid$ `real` $\mid \delta$	numeric types
$\tau \in$ Type ::= `string` $\mid$ `bool`	string or boolean
$\mid \nu \mid \alpha$	num/ord type var.
$\mid \tau \rightarrow \tau$	function type
$\mid \tau$ `list`	list type
$\mid \tau * \tau'$	tuple type
$\mid Annotate(\tau, e, r)$	annotated type
$\eta \in$ QType ::= `string` $\mid$ `bool`	
$\mid \nu \mid \alpha \mid \gamma$	
$\mid \eta \rightarrow \eta$	
$\mid \eta$ `list`	
$\mid \eta * \eta'$	
r $\in$ RuleName ::= APP	function application
$\mid$ CONS	list concatenation
$\mid$ FNL	function left hand side
$\mid$ FNR	function right hand side
$\mid$ OP-*op*	primitive operator *op*
$\mid$ OP-L	prim. op. left hand side
$\mid$ OP-R	prim. op. right hand side
$\mid$ VAL	`val` definition
$\mid$ VAL-L	`val` left hand side
$\mid$ VAL-R	`val` right hand side

Figure 8: Exp_2Types.

typed and unified, the left to right bias of the W algorithm is avoided. Full details appear in [21, 19]. An interactive type exploration environment based on an algorithm similar to the $\mathcal{U}_{\text{AE}}$ has been developed independently by Huch, Chitil and Simon[22].

The $\mathcal{H}$ algorithm generates types containing annotations justifying how the type was inferred. Figures 9 and 10 define the $\mathcal{H}$ algorithm for Exp_2, omitting cases where inference may fail, and writing β_i for fresh ordinary type variables and δ for fresh numeric type variables. Unification occurs on sets of type constraints, denoted $\triangle$. A constraint set is a set of type equalities, each denoted $\sigma \doteq \sigma'$. The ancillary functions used in the algorithm are given in Appendix B. For brevity the definitions for val, tuples and conditionals are omitted here, but can be found in [19], along with a more comprehensive description of the algorithm. The definition for val is similar to let, for tuples is similar to list construction, and for conditionals is similar to pattern definition.

The explanation of definitions (1) to (8) of the algorithm are as follows. The explanation of the definition (9) for pattern matching function definitions is deferred to be discussed with an example in section 4.3.1.

(1) The type of an empty list, β `list`, is annotated with NILLIST.

(2) The type of a constant is determined using the ConstType function and annotated with BaseType.

(3) The type of a numeric operator, $\delta \times \delta \rightarrow \delta$ is annotated with NumOp to explain the relationship between the two arguments and result.

(4) If the type of an identifier x is in the type environment TE, retrieve the annotated type, instantiate it, otherwise assume it's type to be β in the type environment and annotate the type with TypeOf to record the assumption.

(5) The type of an application is inferred by typing the function and argument, annotating the argument with InOperand to indicate that it is used as argument to a function. A new type variable β is introduced for the result of the function and is annotated with SameTyApply. The types inferred from function and argument are unified together with the constraint that the function type τ_1 must unify with the argument to assumed result type $(\tau_{reason} \rightarrow \beta_{reason})$.

(6) The type of a local definition is inferred by typing both subexpressions, and unifying the type environments. The body, e_2 is annotated with LET to indicate it is the body of a local definition.

(7) The type of a list constructor is inferred by typing the two expressions, annotating the first with HD, the second with TL, selecting the simplest explanation for the result type, and finally unifying the constraints from both expressions.

(8) The type of an abstraction is inferred by by typing the pattern and the body, annotating the pattern with FNLeft and the body with FNRight, and unifying the environments.

4.3 Annotating the Abstract Syntax Tree

The $\mathcal{H}$ algorithm is used to generate an annotated type for each node in the abstract syntax tree. The annotation indicates which rule was used to form it (r), the piece of AST it corresponds to (e), and the type environments (TE) that contributed to the original type's inference. These type environments in turn contain annotated types with references to further type environments. Thus, the cyclic graph of type 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 the type environment associating it with a new type variable and 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 type environment is then formed by combining and unifying it's and the child nodes' type environments.

$$\mathcal{H} : \text{TypeSchemeEnv} \times \text{Exp}_2 \rightarrow \text{Type} \times \text{TypeSchemeEnv}$$

$$\mathcal{H}(TE, []) \equiv (Annotate(\beta\ \texttt{list},\ [],\ \text{NILLIST}),\ TE) \tag{1}$$

$$\mathcal{H}(TE, c) \equiv (Annotate(\text{ConstType(c), c, BaseType}),\ TE) \tag{2}$$

$$\mathcal{H}(TE, +) \equiv (Annotate(\delta \times \delta \rightarrow \delta, +, \text{NumOp}),\ TE) \tag{3}$$

$$\mathcal{H}(TE, x) \equiv \tag{4}$$

if $x \in Dom(TE)$ then
let $\forall \alpha_1 \cdots \alpha_n . \tau = TE(x)$
in $(\{\beta_i / \alpha_i\}\tau,\ TE)$
else
$(Annotate(\beta, x, \text{TypeOf}), \{x : \beta\} \oplus TE)$

$$\mathcal{H}(TE, e_1 e_2) \equiv \tag{5}$$

let
$(\tau_1, TE_1) = \mathcal{H}(TE, e_1)$
$(\tau_2, TE_2) = \mathcal{H}(TE, e_2)$
$\tau_{reason} = Annotate(\tau_2, e_2, \text{InOperand})$
$\beta_{reason} = Annotate(\beta, e_1, \text{SameTyApply})$
$\triangle = \{\tau_1 \doteq \tau_{reason} \rightarrow \beta_{reason}\} \oplus \text{EnvConstraints}(TE_1, TE_2)$
$S = \text{UnifyConstraintSet}(\triangle)$
in $(S(\beta_{reason}), S(TE_1 \oplus TE_2))$

$$\mathcal{H}(TE, \texttt{let}\ x = e_1\ \texttt{in}\ e_2) \equiv \tag{6}$$

let
$(\tau_1, TE_1) = \mathcal{H}(TE, e_1)$
$TE_{new} = TE \oplus \{x : \text{Close}(TE_1, \tau_1)\}$
$(\tau_2, TE_2) = \mathcal{H}(TE_{new}, e_2)$
$\tau_{reason} = Annotate(\tau_2, e_2, \text{LET})$
$S = \text{UnifyEnv}(TE_1, TE_2 \ominus x)$
in $(S(\tau_{reason}), S(TE_1 \oplus (TE_2 \ominus x))$

$$\mathcal{H}(TE, e_1 :: e_2) \equiv \tag{7}$$

let
$(\tau_1, TE_1) = \mathcal{H}(TE, e_1)$
$\tau_{1reason} = Annotate(\tau_1, e_1, \text{HD})$
$(\tau_2, TE_2) = \mathcal{H}(TE, e_2)$
$\tau_{2reason} = Annotate(\tau_2, e_2, \text{TL})$
$\tau = \text{SelectSimplest}(\tau_{1reason}\ \texttt{list},\ \tau_{2reason})$
$\triangle = \{\tau_1\ \texttt{list} \doteq \tau_2\} \oplus \text{EnvConstraints}(TE_1, TE_2)$
$S = \text{UnifyConstraintSet}(\triangle)$
in $(S\tau, S(TE_1 \oplus TE_2))$

Figure 9: The $\mathcal{H}$ Algorithm (see also Figure 10, and Appendix B)

$$\mathcal{H}(TE, \mathtt{fn}\ p \Rightarrow e) \equiv \qquad (8)$$
$$\begin{array}{l}
\quad \text{let} \\
\qquad TE' = TE \ominus \text{ftv}(p) \\
\qquad (\tau_1, TE_1) = \mathcal{H}(\emptyset, p) \\
\qquad \tau_{1reason} = Annotate(\tau_1, p, \text{FNLeft}) \\
\qquad (\tau_2, TE_2) = \mathcal{H}(TE', e) \\
\qquad \tau_{2reason} = Annotate(\tau_2, e, \text{FNRight}) \\
\qquad S = \text{UnifyEnv}(TE_1, TE_2) \\
\quad \text{in } (S(\tau_{1reason} \to \tau_{2reason}, (e, FN)), S(TE_2 \ominus \text{ftv}(p)))
\end{array}$$
$$\mathcal{H}(TE, \mathtt{fun}d_x) \equiv \qquad (9)\text{See Section 4.3.1}$$
$$\begin{array}{l}
\quad \text{let} \\
\qquad TEs = \sum_{\mathcal{U}_{\mathbf{AE}}}(d\ ) \\
\qquad S_{column} = \sum_{column}(TEs) \\
\qquad S_{env} = \sum_{env}(TEs) \\
\qquad S_{fun} = \text{UnifyTypes}(\{TE_{ij}(x) | i, j \in Z\}) \\
\quad \text{in } ((S_{env} \cup S_{column} \cup S_{fun})\tau_x), \sum TE_{Exp} \ominus x)
\end{array}$$

Figure 10: The $\mathcal{H}$ Algorithm (see also Figure 9 and Appendix B)

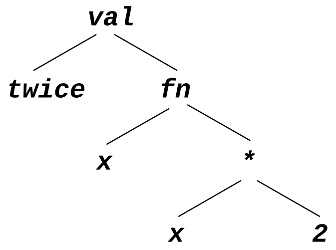

Figure 11: AST for `val twice = fn x => ((* x) 2)`

When an operator of known type is reached, an operator-specific rule is applied to unify the operand types and type environments.

Consider once again our example `val twice = fn x => ((* x) 2)` with the AST shown in Figure 11. 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.

Figure 12 shows the form of annotated nodes. *rule* indicates which $\mathcal{U}_{\mathbf{AE}}$ rule was invoked. *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. *type environment* is the node's type environment if any. Figure 13 shows the type tree for the above example after AST descent. Note the combined nodes `VAL-R/FN` and `FN-R/OP-*`, which, to simplify presentation, conflate a function as the right hand side of a value definition, and an operation as the right hand

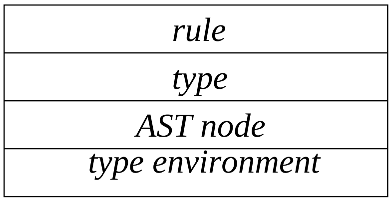

Figure 12: Annotated type tree node

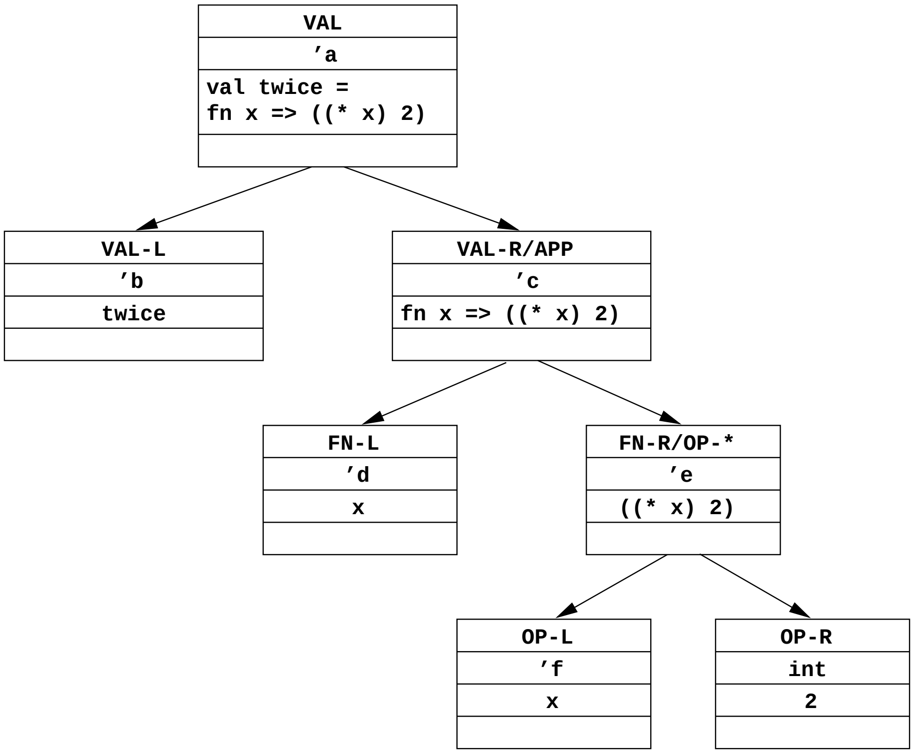

Figure 13: Initial annotated type tree for `val twice = fn x => ((* x) 2)`

side of a function, respectively.

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 type environments for the contributing nodes are also recorded. Figure 14 shows the type tree for the example after type construction. Note that, for brevity, the initial type environment is not shown on each node.

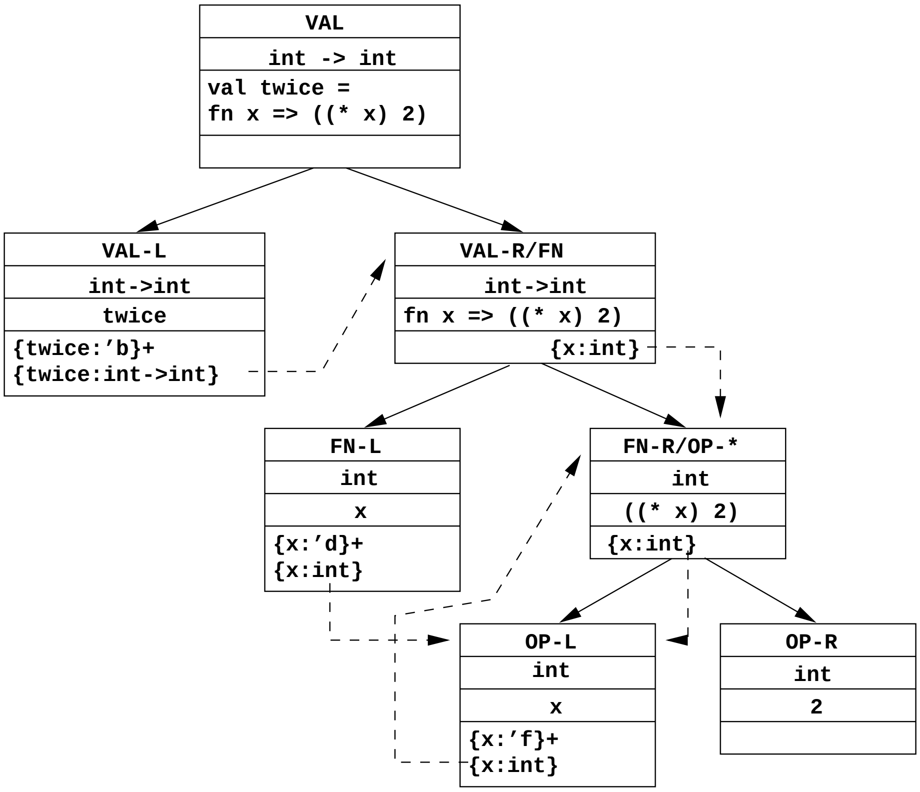

Figure 14: Complete annotated type tree for `val twice = fn x => ((* x) 2)`

4.3.1 Pattern Matching Function Definitions

In the W Algorithm, multiple-case function definitions are treated as a sequence of separate single case functions which should all have the same type. Each case is checked to produce a substitution set and a type. Each case's type is then unified with its predecessor's type to produce another substitution set; each case's type having been modified by the substitution set from the preceding unification. The type for the whole function is then the type for the last case to be so treated. Each case is checked left to right and the whole definition is checked top to bottom. For further details see Field and Harrison [13].

In the following account, pattern matching function definitions defined using the SML `fun` form are treated as a special case to facilitate 2-dimensional analysis. When typing such a function,

$$\begin{array}{l}
\texttt{fun}\ x\ p_{11}\ldots p_{1n} = e_1 \\
\quad x\ p_{21}\ldots p_{2n} = e_2 \\
\quad \ldots \\
\quad x\ p_{m1}\ldots p_{mn} = e_m
\end{array}$$

the $\mathcal{U}_{\mathbf{AE}}$ algorithm generates a matrix of pairs of type and type environment with a row for each definition case, and a column for each pattern and for the result. We denote the type matrix, $TEs = \sum_{\mathcal{U}\mathbf{AE}}(d) =$

$$\left|\begin{array}{cccc}
(\tau_{11}, TE_{11}) & \ldots & (\tau_{1n}, TE_{1n}) & (\tau_{1(n+1)}, TE_{1(n+1)}) \\
(\tau_{21}, TE_{21}) & \ldots & (\tau_{2n}, TE_{2n}) & (\tau_{2(n+1)}, TE_{2(n+1)}) \\
\ldots\ldots & & & \\
(\tau_{m1}, TE_{m1}) & \ldots & (\tau_{mn}, TE_{mn}) & (\tau_{m(n+1)}, TE_{m(n+1)})
\end{array}\right|$$

The example function below, that counts how often an integer list contains a `5`, has the initial matrix shown in Figure 15.

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

The inference uses the following observations. The types in each column must be unifiable, and are combined into a single substitution S_{column} as detailed in Figure 16. We write $\sum_{column}$ as shorthand for these equations, and τ_{i1} for the unified type in the first column, and τ_{i2} for the types in the second etc. Similarly, in every row, each identifier must have a unifiable type at all occurrences, and these are combined into a single substitution S_{Env} as detailed in Figure 17. We write $\sum_{env}$ as shorthand for these equations. The function bodies must have unifiable types: $\sum TE_{Exp} = TE_{1(n+1)} \cup TE_{2(n+1)} \cup \ldots\ldots TE_{m(n+1)}$. Finally, we write τ_x for the function type constructed from the unified column types, i.e. $\tau_{i1} \rightarrow \tau_{i2} \rightarrow \ldots \rightarrow \tau_{in+1}$.

After unification the `count5` matrix has the form shown in Figure 18. Type annotation links in a matrix element are omitted when they originate from a subtree in that element.

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

Figure 15: Initial annotated type matrix for `count5`.

$$S_{t1} = \text{UnifyTypes}(\tau_{11}, \tau_{21}, \ldots, \tau_{m1})$$
$$S_{t2} = \text{UnifyTypes}(\tau_{12}, \tau_{22}, \ldots, \tau_{m2})$$
$$\ldots\ldots$$
$$S_{tn} = \text{UnifyTypes}(\tau_{1n}, \tau_{2n}, \ldots, \tau_{mn})$$
$$S_{column} = S_{t1} \cup \ldots \cup S_{tn}$$

Figure 16: $\sum_{column}$Unification of pattern columns

$$S_{Env1} = \text{UnifyEnv}(TE_{11}, \ldots, TE_{1n}, TE_{1(n+1)})$$
$$\ldots\ldots$$
$$S_{Envm} = \text{UnifyEnv}(TE_{m1}, \ldots, TE_{mn}, TE_{m(n+1)})$$
$$S_{env} = S_{Env1} \cup \ldots \cup S_{Envm}$$

Figure 17: $\sum_{env}$Unification of Identifier Types.

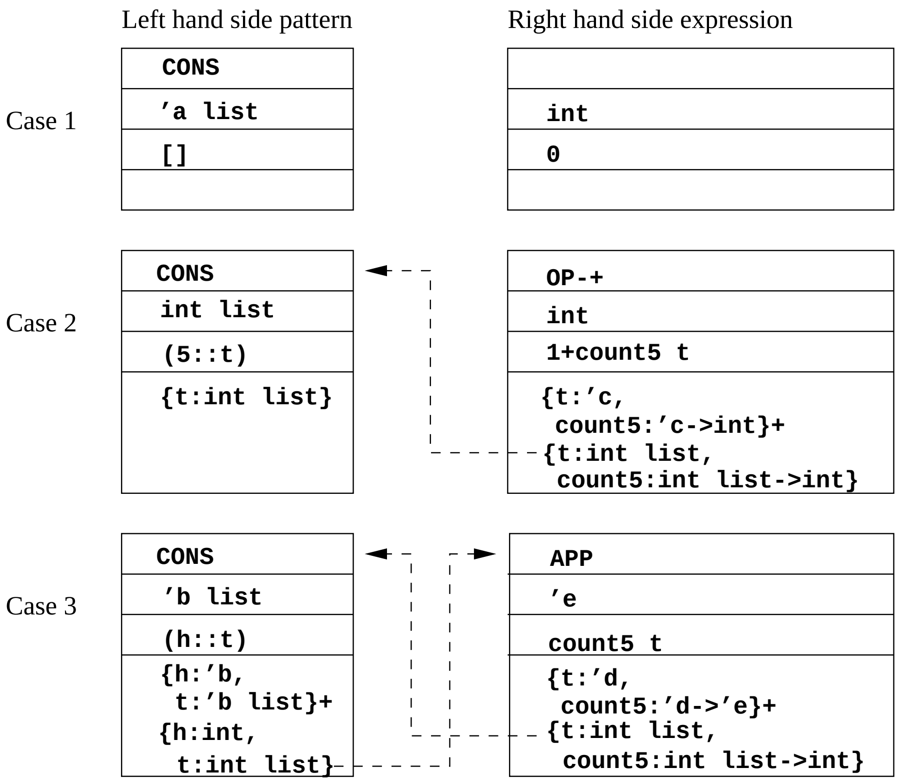

Figure 18: Complete annotated type matrix for `count5`.

4.4 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 type environment, recording the history of the inference of that node's type, is inspected. However, as all contributing type 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. A full description of such a graph traversal is tedious: here we give a textual description supported by example. A more formal treatment can be found in [19].

If a program variable appears in multiple type environments then all occurrences 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 14, 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 19.

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 type environments. During explanation generation, if a sub-explanation has already been generated then a reference is made back to that sub-explanation, e.g. the last line in the following example.

For our SML subset, there are 40 rules for explanation corresponding to the 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

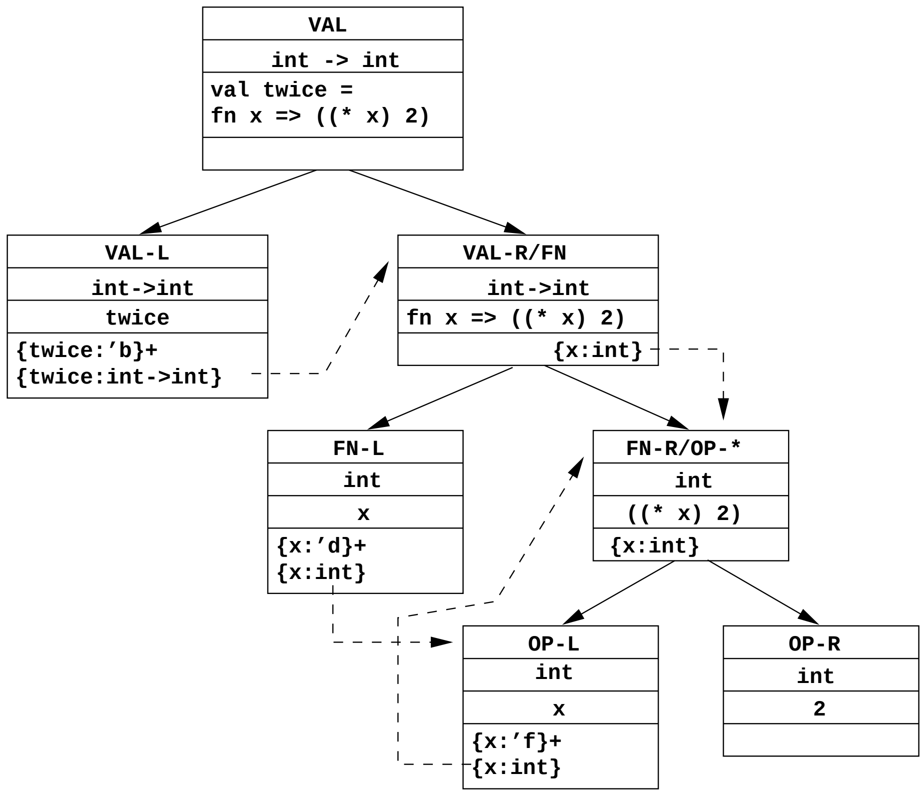

Figure 19: Reduced annotated type tree for `val twice = fn x => ((* x) 2)`

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 Appendix A.

5 Related Work

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

Soosaipillai's system [23] 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 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 [24]. The approach is similar to that used by Wand[25] 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[26].

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 and Stansifer [27] discussed next. 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 [27] 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 functions `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 tree 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 [28] 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 to 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 inference system.

Bernstein and *Stark* give a method of debugging type errors in a so-called *open system* [29]. A new type definition with an assumption environment is defined. The type definition admits principal typings and in addition is very closely related to the ML type system. An assumption environment in their system is a finite mapping of variables to sets of simple types, i.e. not including type schemes as defined above. Even though the type definition has a polymorphic let construct, it does not use type schemes, only simple types. This is owing to the use of assumption environments. The type definition is essentially the same as that presented by Shao and Appel [30]. Bernstein and Stark proved the relation of the type definition under type environment and type definition under assumption environment.

Their type inference can give the possible types for an unbound program variable. The user replaces a suspect expression within the program with a free variable. The system infers the type of the free variable, which the user can compare with that expected. The user can repeatedly probe the program in this way until the error is uncovered. In contrast to the other techniques, this requires user interaction to locate the error. In particular, the user must guess the probable location of the type error. Furthermore, it is not clear how much of an issue large assumption environments really are in practice.

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 [26].

Finally, the explanations are not source-based, tending to use 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, purely 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 $\mathcal{U}_{\mathrm{AE}}$ 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. We expect that generating concise yet understandable explanations for the types of mutually recursive top level constructs will provide significant challenges. Furthermore, 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 more complete 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 $\mathcal{U}_{\mathrm{AE}}$ algorithm, and all our colleagues who took part in our expert type checking experiments. We also wish to thank the anonymous referees for helpful comments.

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

Appendix A 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
```

Appendix B Ancillary Functions

The ConstType function simply returns the type of constants.

ConstType (c)
$c \in$ Int = `int`
$c \in$ Real = `real`
$c \in$ Bool = `bool`
$c \in$ String = `string`

The following functions are all defined fully in [19].

The ftv function returns the free type variables in a type, type scheme or type environment.

Close (TE, τ) denotes a type scheme generated by closing a type τ under type environment TE, that is $\forall\alpha_1 \dots \alpha_n.\tau$ where $\{\alpha_1 \dots \alpha_n\} = ftv(\tau) \backslash ftv(TE)$.

SelectSimplest(τ, τ') returns the annotated type τ if it has a simpler explanation than τ', and otherwise returns τ'.

Constraint sets, denoted $\triangle$, are finite sets of pairs of type schemes, written $\{\sigma_1 \doteq \sigma'_1, \cdots, \sigma_n \doteq \sigma'_n\}$.

SimplifyConstraint normalises a constraint, e.g. always writing type variables on the left, and is defined over the structure of types [19]. Similarly, SimplifyConstraintSet normalises a set of constraints.

EnvConstraints constructs a constraint set for the common identifiers of two type environments:
EnvConstraints$(TE_1, TE_2) = \{TE_1(x) \doteq TE_2(x) \mid x \in (Dom(TE_1) \cap Dom(TE_2))\}$.

Applying RemoveQTyVar to a constraint set eliminates all quantified type variables, if none are constrained to be equal, otherwise returns the constraint set unchanged.

Unification Functions

UnifyEnv (TE_1, TE_2) = UnifyConstraintSet(EnvConstraints(TE_1, TE_2))

UnifyTypes(τ_1, τ_2) = UnifyConstraintSet($\{\tau_1 \doteq \tau_2\}$)
UnifyTypes(τ_1, τ_2, τ_3) = UnifyConstraintSet($\{\tau_1 \doteq \tau_2, \tau_1 \doteq \tau_3, \tau_2 \doteq \tau_3\}$)
etc.

UnifyConstraintSet$(\triangle)$ = UnifyLoop$(\triangle, \emptyset)$

$$
\begin{array}{ll}
\text{UnifyLoop}(\triangle, S) = & \text{case RemoveQTyVar(SimplifyConstraintSet}(\triangle)) \text{ of} \\
& \emptyset \Rightarrow \emptyset \\
& \{\gamma \doteq \gamma'\} \cup \triangle'' \text{ where } \{\gamma \neq \gamma', \gamma' \neq \gamma\} \cap \triangle'' = \emptyset \\
& \qquad \Rightarrow \text{let } \; S' = \{\gamma/\gamma'\} \\
& \qquad \quad\;\; \text{in } \; \text{UnifyLoop}(S'(\triangle''), S'S) \\
& \{\alpha \doteq \tau\} \cup \triangle'' \text{ where } \alpha \notin ftv(\tau) \\
& \qquad \Rightarrow \text{let } \; S' = \{\tau/\alpha\} \\
& \qquad \quad\;\; \text{in } \; \text{UnifyLoop}(S'(\triangle''), S'S) \\
& \{\delta \doteq \nu\} \cup \triangle'' \\
& \qquad \Rightarrow \text{let } \; S' = \{\nu/\delta\} \\
& \qquad \quad\;\; \text{in } \; \text{UnifyLoop}(S'(\triangle''), S'S) \\
& \triangle \\
& \qquad \Rightarrow \text{fail}
\end{array}
$$

References

[1] Strachey, C. (2000) Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, Kluwer, 13:1&2, 11–49.

[2] Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, Academic Press, 17, 348-75.

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

[4] Turner, D. (1985) 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.

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

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

[7] Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P. (1998) Making the future safe for the past: Adding Genericity to the Java Pro gramming Language *OOPSLA 98*, Vancouver, October, ACM Press, New York.

[8] Cartwright and Steele G.L. Jr. (1998) Compatible Genericity with Run-time Types for the Java^{TM} Programm ing Language *OOPSLA 98*, Vancouver, October, ACM Press, New York.

[9] Myers, A.C., Bank, J.A. and Liskov, B. (1997) Parameterized Types for Java *Proc. ACM Principles of Programming Languages*, Paris, France, January.

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

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

[12] Cardelli, L. (1984) Basic polymorphic type checking. *Science of Computer Programming*, 8:2, 147–72.

[13] Field, A. J. and Harrison, P.G.,. (1988) *Functional Programming.* Addison-Wesley, Wokingham.

[14] Yang, J., Michaelson, G. and Trinder, P. (2001) Human and 'human-like' type explanations in G. Kadoda(Ed), Proceedings of 13th Annual Workshop of the Psychology of Programming Interest Group, Bournmouth, April, Bournmouth University, 163-172.

[15] Yang, J., Michaelson, G. and Trinder, P. (2001) How human-like are "human-like" polymorphic type explanations? Proceedings of 2nd Annual Conference of the LSTN Centre for Information and Computer Sciences, London, September, University of Ulster, 97-101.

[16] Michaelson, G. (1995) *Elementary Standard ML.* UCL Press, London.

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

[18] Wiedenbeck, S. and Scholz, J. (1989) 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*, Elsevier, Amsterdam, 82–87.

[19] Yang, J. (2001) Improving Polymorphic Type Explanations PhD Thesis, Dept of Computer Science, Heriot-Watt University, Edinburgh, Scotland.

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

[21] Yang, J. (2000) 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*, Intellect, Bristol, 58-66.

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

[23] Soosaipillai, H. (2000) An explanation based polymorphic type checker for Standard ML. Master's thesis, Dept of Computer Science, Heriot-Watt University, Edinburgh, Scotland.

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

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

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

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

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

[29] Bernstein, K.L. and Stark E.W. (1995) Debugging type errors (full version). Technical Report, State University of New York at Stony Brook.

[30] Shao Z. and Appel A.W. (1993) Smartest Recompilation *T*wentieth Annual ACM Symposium on Principles of Programming Languages, January, 439–450.