# Human and "human-like" type explanations

Yang Jun, Greg Michaelson[1] & Phil Trinder
*Dept. of Computing & Electrical Engineering*
*Heriot-Watt University, Riccarton, EH14 4AS*
*{ceejy1,greg,trinder}@cee.hw.ac.uk}*

Keywords: POP-II.B. *problem comprehension* POP-II.B. *types of programmer behaviour*

## Abstract

The behaviour of a "human-like" polymorphic type explanation system is analysed using the same categories as those used to characterise human expert type explanation behaviour. The results suggest that the computer system has a similar behaviour profile to that of a composite human expert.

## Introduction

We have been investigating techniques for making mechanical polymorphic type inference more understandable. Most type checking algorithms are based on Hindley-Milner type inference (Milner 1978), which introduces new intermediate constructs in the course of type determination. In particular, type variables are used as place holders for the as yet unknown types of sub-expressions. Error reporting systems that build on the Hindley-Milner approach tend to use such intermediate representations in explaining how types are derived, although the programmer did not use introduce them in the original program. Our experience suggests that users, especially naive users, find such explanations hard to understand.

A number of explanation systems have been constructed to aid users in elucidating types (Soosaipillai 1990) (Beaven 1993) (Duggan 1996). These tend to be based directly on Hindley-Milner algorithms and use intermediate type variables in their explanations.

In contrast, we wished to build a type explanation system that was in some sense "human like"; that is it should incorporates rules intended to mimic the behaviour of people explaining type derivations. Experiments with expert human type checkers confirmed our intuitions that people tend to give explanations that avoid intermediate representations, focusing on concrete aspects of the program being analysed. In building our system, we attempted to incorporate rules that reflect such behaviours.

We have recently completed our system (Yang 2001). Its explanations are very unlike those from Hindley-Milner based approaches and appear more human-like to us. We now return to our original analysis of human behaviour, and analyse the system's behaviour using the same categories on the same problem set. While there are some differences, comparison suggests that our system has similar behaviour to human experts.

## Original experiment

Eight expert subjects were video taped while solving a set of 34 error-free type explanation problems in a pure functional subset of Standard ML, corresponding to that taught to our 1st year undergraduate students. Initial analysis of the results led to the identification of 14 major techniques used to identify types.

These were:

1 1. Construct type skeleton for function. A schematic for a type signature is drawn, indicating function and tuple types but omitting fine detail.

[1] Contact author.

2 2. Construct and refine type skeleton for local type. A schematic for a local construct is used to guide type checking, for example for a tuple.

3 3. Refine function type skeleton. Details of the type skeleton are filled in.

4 4. Locate specific types. Constructs with ground types, such as integer, real or boolean values are identified.

5 5. Locate overloaded operators. Operators with ambiguous but restricted types are identified, such as arithmetic operations that apply to integers or reals.

6 6. Locate known system types. Constructs of known or constrained type provide the focus, for example list constructors or condition components.

7 7. Locate commonalities across patterns. The heads or the bodies of function cases are scanned for common features.

8 8. Analyse top down. Analysis proceeds systematically from a construct to its sub-constructs, for example checking the function and argument expressions in a function call.

9 9. Analyse bottom up. Analysis proceeds systematically from sub-constructs to a construct, for example checking the operands of an operator before deciding the overall type of an operation.

10 10. Identify argument type from use in body. The user focuses on the use of a formal parameter in a function body to identify its type.

11 11. Identify body construct type from known argument type. Type information from a function head is applied to type the body.

12 12. Search forwards and backwards from a known type. A known type's immediate context is explored.

13 13. Introduce type variable. A type variable is introduced as a place holder.

Further analysis provided frequency counts of technique use. As we only had a small number of subjects, we decided to amalgamate all results into a composite subject, to try and characterise "average" behaviour. Here we have grouped these techniques into four major classes to ease presentation:

A. Type skeleton - 1, 2 and 3;

B. Concrete types - 4, 5, and 6;

C. 2D Inspection - 7, 8, 9, 10, 11 and 12;

D.. Type variable - 13.

Fuller accounts of the experiment may be found in (Yang 2000a) and (Yang 2000b).

## Explanation system overview

Our system accepts a pure functional subset of Standard ML, including integer, real and boolean types, tuples, lists, curried functions and local definitions. It parses a function to build an abstract syntax tree.

The system then uses a modified version of our Unification of Assumption Environments(UAE) algorithm (Yang 2000c) to construct an annotated type tree reflecting the derivation of the type of each significant construct in the function. The UAE essentially unifies all assumption environments contributing to the inference of a construct's type and avoids the left to right bias in Milner's W algorithm. In particular, the UAE will unify types across cases, thus effectively enabling 2 dimensional type inspection.

Types may have multiple derivations, so the type tree is pruned to remove cycles and to leave shortest derivations. This typically removes chains of derivation that involve intermediate representations. The pruned type tree is finally traversed and an English-like explanation is generated. A full account may be found in (Yang 2001).

## Comparison

We ran the 34 problems through our system and analysed the output using the categories from the above experiment. That is, we tried to treat the system's explanations as though they were from a human subject.

The system used 407 techniques on 34 problems, an average of 11.97 techniques per problem. In comparison, the experts used 1830 techniques on 147 problems, that is an average of 12.45 techniques per problem. Thus, average technique use is very similar.

Comparative technique usage is summarised in Table 1.

| Technique class | Expert% | Expert Std.Dev. | System% |
|---|---|---|---|
| Type skeleton | 29.63 | 7.02 | 20.88 |
| Concrete types | 28.37 | 7.97 | 29.23 |
| 2D inspection | 36.87 | 6.75 | 42.50 |
| Type variable | 5.13 | 1.50 | 7.37 |

*Table 1: Expert and computer technique use on 34 problems.*

To derive the "Expert%" column, for each subject the total number of techniques used within a class is divided by the total number of questions attempted by that subject. These normalised figures form the basis of the overall average percentage technique class use shown here. The "Expert Standard Deviation" column is then formed by comparing each subject's normalised technique class use with this average.

It appears that experts make considerably more use of type skeletons than the system, beyond one standard deviation. However, the system always proceeds to explain a function's type from left to right through the type signature, so we count skeleton introduction and refinement as just two technique uses for all problems. For the experts, an occurrence of skeleton refinement is counted each time they return to fill in skeleton details.

Use of concrete, overloaded and known types is very similar between experts and our system. Based on the experiment with the experts, our system was biased to apply rules to identify such types.

The system makes greater use of 2D inspection than humans, but within one standard deviation. Again, following the experiment with the expert subjects, our system was biased to always attempt to apply rules across related patterns.

Our system also makes slightly greater use of type variables than the experts, beyond one standard deviation. We think that our system makes minimal use of type variables so we may be under-counting human type variable use.

Note that the experts only completed 18.38 problems on average. Thus, the above comparison may be biased by the mix of problems and the appropriateness of technique use across all 34 problems. In particular, if the last third of the questions require a different balance of techniques to the first two thirds, then the results from the system may be distorted. Considerable further analysis would be required to clarify this.

### Individual expert and system explanations

Overall, while the system diverges from the composite user, we think that it is still plausible as an individual user. Consider explaining the type of:

```
fun count [](n,z,p)  = (n,z,p) |
    count (0::t) (n,z,p) (0::t) =
     count t (n,z+1,p) |
    count (h::t) (n,z,p) =
     if h<0
     then count t (n+1,z,p)
     else count t (n,z,p+1)
```

which counts how many elements in a list of integers are negative, zero or positive. For example:

```
count [0,~1,0,2,0,~1,0,~2,3,4,~3,0] (0,0,0) ==> (4,5,3)
```

Our system's explanation is:

```
Function type: int list ->
               int * int * int ->
               int * int * int
Argument 1: int list
1: "[]", "(0::t)" and "(h::t3)" have the same
   type
   - same LHS pattern position
2: list of "0", "t" and "(0 :: t)" have same type
   -operands/result of "::"
     3: "0":  int
4:  list of "0": int list

Argument 2: int * int * int
5: "(n,z,p)", "(n2,z2,p2)" and "(n3,z3,p3)" have
   the same type
   - same LHS pattern position
6:  is a Tuple type: int * int * int
Tuple Element 1: int
  7: for "n3" in "(n3,z3,p3)"
  8: "n3" and "1" have the same type in "n3+1"
     - arguments/result of "+" have the same
       Number types
  9: "1": int
Tuple Element 2: int
  10: for "z2" in "(n2,z2,p2)"
  11: "z2" and "1" have the same type in "z2+1"
      - arguments/result of "+" have the same
        Number types
  12: "1": int
Tuple Element 3: int
  13: for "p3" in "(n3,z3,p3)"
  14: "p3" and "1" have the same type in "p3+1"
      - arguments/result of "+" have the same
        Number types
  15: "1": int
Result: int * int * int
16:  at Row 1 "(n,z,p)" is the same as Argument2
     "(n,z,p)", has type: int * int * int
     - see 5 above
```

At step 1, the system identifies that the left hand side case patterns for the first argument must have the same type. At step 2, it focuses on the pattern `(0::t)` because of the known constant `0`, identified as having type integer at step 3. Thus, the argument is identified as an integer list at step 4. At step 5, the system notes that the

left hand side case patterns for the second argument have the same type, which is identified as a tuple type at step 6. At step 7, the system focuses on `n` in the third case as it can use `n+1` at step 8 to infer that `n` and `1` have the same type, which is an integer from step 9. Similarly, at steps 10 to 12, the type of `z` is explained as integer, and, at steps 13 to 15, the type of `p` is explained as integer. Finally, at step 16, the type of the result in the first case `(n,z,p)` is noted as the same as the pattern in that case for `(n,z,p)`.

Compare this with the following expert explanation:

*"The 3rd argument is a triple and the result is likewise a triple. We start from the first line of the function. We know that the first argument is a list because it contains the nullary list constructor. We further know that the 1st argument is a list of integers because there's a 0 appended onto the list at the 2nd equation. So this is a list of integers. Looking, pattern matching, looking for occurrences of n, z and p. I know that the type of the 1st argument is the same as the type of the result argument because of the first equation (n,z,p). (n,z,p) triple occurs on both sides. I see that 1 is added to z. 1 is integer so I know that z must be an integer. So the middle element in those two triples is an integer. I see that 1 is added to p so teh last element if the triples is also an integer and finally I see that 1 is added on into the 1st element of the triple so that likewise is an integer."*

During this explanation, the expert filled in a type skeleton they had drawn at the start.

There are many points of similarity between the system's and the expert's explanations. Both focus on the pattern `(0::t)` in identifying the first argument as an integer list. The alternative would be to deduce that `h` is integer from the comparison `h<0`. For the system, this would have resulted in a longer explanation and is discarded. The expert explicitly states that they are starting at the first line and would find `(0::t)` before `h<0` in a case by case scan of the function.

Both focus on the first case as linking the result `(n,z,p)` with the argument pattern `(n,z,p)`, and both use the overloaded operator `+` with `1` to explain `n`, `z` and `p` as integers. However, there are no alternatives to such explanations, which would also be found by Hindley-Milner rules.

Consider a more verbose explanation, which makes more assumptions explicit:

*"Now we have the definition of a function `count` which is defined by pattern matching to apply to 3 different well actually it takes 2 arguments curried. The 1st argument must match one of 3 patterns. The 1st argument must be a triple so the 1st argument the 3 patterns it must match is `nil`, `0` cons `t` where `t` is a variable and anything cons anything. OK so the type of parameters must be monomorphic in ML and so that means the type of the 1st parameter must be int list. Now the type of the 2nd parameter must be just looking at the parameters all I know is that it must be a tuple. Now in the 1st clause we simply return the tuple so that does not constrain the 2nd parameter. In the 2nd clause we recursively invoke `count` with the tail of the first parameter which is fine. That'll match the type of the parameter of count and then another triple where the 1st and 3rd components are the same but the 2nd component has `1` added to it from the parameter so that means the 2nd component must be something that can have `1` added to it and I believe in ML that is only the type `int`. So we know that the 2nd parameter of `count` has the type of a triple the 2nd component of which is `int`. OK now let's look at the 3rd clause of the definition of count. Well, by similar reasoning to the 2nd clause, looking at the 3rd clause we see that all 3 components of the triple must be of type `int`. OK so we know and then the return value in the base case is a triple which is of type `int*int*int`, the 2nd case is simply recursive invocation and the 3rd case is one of 2 different recursive invocations so the type return value then is going to be `int*int*int` so the type of count is going to be `int list -> int*int*int -> int*int*int` where `*` binds more tightly than `->`.*

It is interesting that this subject also initially identifies three rather than two arguments and the corrects themselves. They make the 2D relationship between the three variants for the list argument explicit, as does the system, and they again use `(0::t)` to identify this as a list of integers. Again, the use of `+` with `1` identifies the tuple elements as integer, and the use of the tuple in the pattern and body of the first case identifies the broad type of the result. This explanation is more thorough those from the system or the first expert, as all case bodies are checked for consistency.

## Conclusions

We have developed a system whose explanations for polymorphic types are quite close to those of human experts. Treating the system as if it were an expert on a common set of questions suggests

similar technique class use. Comparing a system explanation with that of two experts on the same example suggests that similar sequences of techniques are used.

We now wish to:

- conduct usability testing of our system's explanations with naive and expert users;
- extend explanations to a full language, including user defined discriminated union types, and classes or modules;
- incorporate explanations in an environment to enable users to browse constructs with type errors.

## Acknowledgements

We wish to than the eight experts who took part in our experiment.

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

## References

Beaven, M. and Stansifer, R. (1993), Explaining type errors in polymorphic languages, *ACM Letters on Programming Langauges and Systems*, Vol 2, pp 17-30, March 1993.

Duggan, D. and Bent, F. (1996), Explaining type inference, *Science of Computer Programming*, Vol 27. pp37-83, 1996.

Milner, R. (1978), A theory of type polymorphism, *Journal of Computer and System Sciences*, No. 17.

Soosaipillai, H. (1990), An explanation based polymorphic type checker for Standard ML, MSc Thesis, Dept of Computer Science, Heriot-Watt University, Scotland, 1990.

Yang, J., Michaelson, G. and Trinder, P. (2000a), How do people check polymorphic types?, In A. F. Blackwell and E. Bilotta (editors), *Twelth Annual Meeting of Psychology of Programming Interest Group Proceedings*, Memoria, pp 67-77, Cosenza, Italy, April 2000.

Yang, J., Michaelson, G. and Trinder, P. (2000b), Helping students understand polymorphic type errors, *First Annual Conference of the LSTN Centre for Information and Computer Sciences*, S. Alexander et al (editors), LTSN-ICS, University of Ulster, pp 11-19, August 2000.

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

Yang, J., Michaelson, G. and Trinder, P. (2001), Explaining polymorphic types, submitted to *Computer Journal*, February 2001.