How do people check 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

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

Abstract

Polymorphic typechecking algorithms efficiently locate type errors in programs, but users find error reporting from such algorithms hard to comprehend. We are investigating the development of a new polymorphic type checker that reports type errors in a more understandable form. Here we present the results of experiments into human checking of both correct and incorrect polymorphic typed programs, and briefly discuss their implications for our proposed new checker.

Introduction

Polymorphic typechecking

The Hindley-Milner typechecking algorithm is widely used in contemporary functional language implementations, for example Standard ML (Milner et al 1997) and Haskell (Peyton Jones et al 1998). The algorithm traverses a program's abstract syntax tree (AST)[1], assigning type variables at each node. It then applies rules corresponding to syntactic constructs to attempt to resolve the assigned type variables against specific type information deduced from program contents and context. For a full account, see Milner (1978).

For example, consider checking the following Standard ML-ish function, which returns a string depending on the sign of its integer argument:

```
fun sign 0 = "zero" |
    sign n = if (> n) 0 then "positive"
                        else "negative"
```

This function has type `int -> string`, i.e. it takes an integer argument and returns a string result.

Prior to type checking, it is assumed that:

```
0 : int                  i.e. 0  is an integer
[illegible]
[illegible]
"negative" : string
> : int -> int -> bool    i.e. > compares two integers to return a boolean
```

[1] An abstract syntax tree is a representation of the meaningful syntactic structure of a language utterance.

The rule for a function:

- checks the first case:
  - concludes `0` is `int` from assumptions
  - checks the body i.e. deduces `"zero"` is `string` from assumptions
  - concludes that the first case is `int -> string`
- checks the second case:
  - assumes parameter `n` has type α
  - checks the body i.e. checks the `if` expression:
    - checks the condition i.e. checks the application of `> n` to `0`:
      - checks the application of `>` to `n`:
        - concludes that `>` is `int -> int -> bool` from assumptions
        - assigns the type variable β to the result of the application
        - concludes that `n` is α from assumptions
        - unifies the anticipated type α -> β with the type of `>`, `int -> int -> bool`
        - concludes that α is `int` and β is `int -> bool`
      - assigns the type variable γ to the result of the application
      - concludes that `0` is `int` from assumptions
      - unifies the anticipated type `int -> γ` with the type of `int -> bool`
      - concludes that γ is `bool`
    - unifies the condition type `bool` with the required type for a condition, `bool`
    - checks the `then` branch i.e. concludes that `"positive"` is `string` from assumptions
    - checks the `else` branch i.e. concludes that `"negative"` is `string` from assumptions
    - unifies the `then and else` branches, which must have the same type
    - concludes that the `if` has type `string`
  - concludes that the second case has type `int -> string`
- unifies the cases

To summarise, the algorithm:

- is exhaustive, considering all parts of a program;
- is top-down, left to right;
- introduces a type variable for many constructs before instantiating a construct's type.

In contrast, 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;
- be opportunistic, scanning a program as 2D text for sources of evidence;
- seek concrete evidence (e.g. ground types) before introducing type variables.

We might caricature a typical human check of the above example as:

- `0` is `int` so `n` must be `int`
- `"zero"` is `string` so the function result must be `string`
- so the function must be `int -> string`

We may note that the caricature human check:

- concludes that every case must be `int -> string` because the first case is `int -> string`;
- scans down the columns of patterns as well as across each case;
- focuses on the ground type objects `0` and `"zero"`.

Note that we find that this caricature is not wholly accurate.

Hindley-Milner Error Reporting

We have observed that the Hindley-Milner algorithm is a poor source of guidance for humans in the presence of errors (Yang & Michaelson 2000). For example, consider the following incorrect definition of the Standard ML function `map`. The `map` function is supposed to apply a function argument `f` to every element of an argument list, for example, given:

```
fun double x = 2*x
```

then:

```
map double [1,2,3,4] ==> [2,4,6,8]
```

The incorrect definition is:

```
fun map f [] = 0 |
    map f (h::t) = f h::map f t;
```

where:

- `[]` ==> empty list;
- `(h::t)`==> pattern where `h` matches the head of a list argument and `t` matches the tail of that list;

- `f h::map f t` ==>apply `f` to `h` and put result on front of list from applying `map` with `f` to `t`.

In SML, all function cases are supposed to return the same type. Here, however, the first case returns the `int 0` but the second case returns a polymorphic list.

Given this definition, the SML of New Jersey system (Version 0.93) reports:

```
example.sml:1.19-2.37
 Error: rules don’t agree (tycon mismatch)
  expected: (’Z -> ’Y) * ’Z list -> int
  found:    (’Z -> ’Y) * ’Z list -> ’Y list
  rule:
    (f,h :: t) => :: (f h,<exp> <exp> t)
example.sml:1.1-2.37
 Error: pattern and expression in val rec dec
        don’t agree (tycon mismatch)
  pattern:    (’Z -> ’Y) -> ’Z list -> ’Y list
  expression: (’Z -> ’Y) -> ’Z list -> int
  in declaration:
    map = (fn arg => (fn <pat> => <exp>))
```

Note the:

- use of the system type variables[2] `’Y` and `’Z`, which do not appear in the original function;
- failure to identify explicitly the clash between the types of the first and second cases as the source of the error;
- error messages refer to the abstract rather than the concrete syntax;
- same error is reported in two different ways.

Edinburgh SML is somewhat more succinct, reporting:

```
Type clash  in:
 (f,(h :: t))=>((f h) :: ((% %) t))
Looking  for a:  int
I have found a:  ’a list
```

Both systems use the Hindley-Milner algorithm as the bases of their typecheckers.

[2] i.e. α and β for non-Greek keyboards...

Experiment

We are interested in automating the provision of helpful explanations of polymorphic type checking, especially for naive users. To that end we have conducted a small study of human polymorphic type checking, to try to identify “best practice” as the basis for an automated system.

Based on 12 years experience of teaching functional languages to undergraduate and postgraduate students, 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.

We have carried out two experiments where experts were video taped type checking sets of SML functions, following a ‘’speak-aloud’’ protocol.

Question classification

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 (Michaelson 1995), 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 questions were chosen to reflect a range of function types. For the 34 error free questions, the function types were:

- ground types (1)
- ground and list types (3)
- ground, list and tuple types (4)
- ground types and type variables (1)
- ground and list types, and type variables (1)
- ground, list and tuple types, and type variables (2)
- function argument and ground types (2)
- function argument, ground and list types (4)
- function argument, list and tuple types, and type variables (2)
- function argument, ground, list and tuple types (1)
- function argument types and type variables (2)
- function argument and ground types, and type variables (1)
- function argument and list types, and type variables (4)
- function argument, ground and list types, and type variables (3)
- function argument, ground, list and tuple types, and type variables (1)

- list and tuple types, and type variables (1)
- type error... (1)

The 34 error questions may contain multiple errors. Identifying the errors might involve locating:

- unresolved overloading (6)
- ground type conflicts (8)
- constructor type conflicts (3)
- inconsistencies between left and right hand sides of definitions (12)
- swapped bound variables (6)
- inconsistencies between cases (7)
- inconsistent numbers of arguments (3)
- non-universally quantified type variable (1)

Subject classification

8 subjects took part in the type correct experiment and 7 in the type error experiment. 6 people took part in both experiments. The subjects all have:

- at least post-graduate Computer Science experience;
- programmed extensively in polymorphic typed languages;
- implemented Hindley-Milner type checkers or worked extensively with them when implementing functional languages;
- tutored undergraduate students.

and may thus be considered experts rather than beginners. Subjects were given up to 30 minutes to complete each set.

Analysis and results

Initial inspection of the video taped sessions led to identification of 13 major inference techniques:

1. locate ground types
2. locate overloaded operators
3. locate other “system” operators of known types e.g. constructors
4. construct type skeleton for whole function corresponding to the number of arguments
5. locate commonalities across patterns
6. analyse top-down, from nodes to leaves of AST
7. analyse bottom up, from leaves to nodes of AST
8. search forwards and backwards from a known type
9. use type variable
10. identify argument type from use in function body

11. identify body construct type from known argument type
12. construct and refine type skeleton for local construct
13. refine function type skeleton

used during type checking. 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.

Overall comparison of attempts at questions with and without errors

Table 1 summarises the numbers of questions attempted and behaviour instances identified:

	subjects	question attempts	attempts/ subject	technique count	count/ attempt
no errors	8	147	18.38	1830	12.45
errors	7	235	33.57	1663	7.08

Table 1 - Total questions attempted and technique counts.

Far more questions were attempted per subject and less techniques were used per question for those with type errors than for those lacking errors. Naively, this implies that the subjects found identifying type errors easier than inferring types, but we would need to assume that both tasks require equal effort. However, as we shall see, subjects are slightly more likely to take a more exhaustive approach to type inference for error free questions. Furthermore, it may be that locating dissonance is intrinsically easier than verifying consistency, but we cannot substantiate this speculation without further experimentation.

Error free question set

Table 2 shows the use of techniques in the error free questions, in descending order of technique counts:

Inference Technique	tech. counts	count/ total%
3. locate known system operators	317	17.32
13. refine type skeleton	280	15.30
10. identify argument type	161	8.80
12. construct/refine local skeleton	156	8.52
4. construct function skeleton	151	8.25
6. top down	136	7.43
1. locate ground types	123	6.72
5. across patterns	112	6.12
7. bottom up	105	5.74
9. use type variable	100	5.46

8. forward search	82	4.48
2. locate overloaded operators	55	3.01
8. backwards search	36	1.97
11. body from argument use	16	0.87

Table 2 - Questions with no type errors: technique counts for each technique and as a percentage of all technique uses.

Considerable use is made of type signature skeletons, which we did not anticipate. Typically, a subject starts with a minimal outline signature (8.25%), with a slot for each bound variable, and for the result. They then fill in the slots to finer degrees of detail as checking progresses (15.30%), identifying argument types (8.80%), introducing sub-skeletons for structured bound variables (8.52%) and identifying the function body type from argument use (0.87%), totaling 40.95% of technique use.

For example, consider inferring the type of:

```
fun count0 [] = 0 |
    count0 (0::t) = 1+count0 t |
    count0 (h::t) = count0 t
```

which counts how often `0` appears in a list of integers:

- initial skeleton: `->`
- identify `[]` as empty list: `list ->`
- identify `0` as integer: `list -> int`
- identify `(0::t)` as integer list: `int list -> int`

As hypothesised, much use is made of concrete type information. Type constructors (17.32%) often appear in patterns in our problem set and will be located early on in a left to right, or left top to left bottom, scan. Ground types (6.72%) are used less than constructors, which we did not anticipate. There is little use of overloaded operators (3.01%), reflecting their general absence from our question set. Concrete type technique use totals 27.05%.

2D inspection of function text, comprising top down (7.43%), across pattern (6.12%), bottom up (5.74%), forward search (4.48%) and backwards search (1.97%), represents 25.74% of technique use confirming our hypothesis.

Low use is made of type variables (5.46%), as we hypothesised.

Contrary to our expectations, full checking was carried out for 114 questions (77.55%) and partial checking for 33 (22.45%).

Error question set

Table 3 shows the use of techniques in the error questions, in descending order of technique counts:

Inference Technique	tech. counts	count/ total%
3. locate known system operators	372	22.37
13. refine type skeleton	206	12.03
1. locate ground types	200	11.73
5. across patterns	195	10.94
8. forward search	144	7.58
4. construct function skeleton	126	6.86
2. locate overloaded operators	114	6.01
12. construct/refine local skeleton	100	6.01
6. top down	87	5.05
8. backwards search	84	3.85
7. bottom up	64	2.77
11. identify argument type	46	2.59
11. body from argument use	43	1.86
9. use type variable	31	0.36

Table3 - Questions with type errors: technique counts for each technique and as a percentage of all technique uses.

For the type error questions, less use was made of type signature skeletons than for the error free problems, through the introduction of an initial outline (6.85%), subsequent refinement (12.02%), local skeletons (6.01%), argument identification (2.59%) and body from argument use (1.86%), comprising 29.33% of technique use.

However, more use was made of concrete type information, through the identification of known operators(22.3%),ground types (11.73%) and overloaded operators (6.01%), comprising 39.34% of technique use.

2D inspection of text, involving across pattern (10.94%), forward search (7.58%), top down (5.05%), backwards search (3.85%) and bottom up (2.77%), comprised 30.19% of technique use. Note that cross pattern comparison was used far more than for the error free problems.

Type variable introduction (0.36%) was the least used technique, and used far less than for the error free questions.

Full checking was carried out for 145 questions (61.70%) and partial checking for 90 (38.28%) questions. There was less use of full checking than for error free questions.

Discussion and conclusions

We have observed a number of important 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 noticed that partial checking was most prevalent where the final type lacks type variables. This may suggest that expert type checkers are more punctilious than the novice students whom we observe from day to day.

We have not found any comparable research specifically on human type checking. However, there are interesting correspondences with work on program comprehension. In particular, our observed use of type skeletons to guide type checking may be an instance of Brooks' top-down, hypothesis driven comprehension (Brooks 1983). Similarly, our observed use of 2D inspection and concrete type features is similar to Wiedenbeck and Scholtz's identification of the use of surface beacons to guide successful program comprehension (Wiedenbeck and Scholz 1989). Wiedenbeck and Scholz note that beacon use is particularly helpful for comprehension of programs with unknown purposes: our questions are uncommented and provide no semantic information through meaningful identifiers.

Here, we have presented a brief overview of our experimental results. We intend to analyse them in more detail, to try and identify which technique sequences people find useful in general, and to relate such sequences to the different categories of error and error-free questions.

We are now constructing a type explanation system for our SML subset which will incorporate heuristics based on these conclusions. The system will be much less efficient than the linear W algorithm as heuristics may be applied repeatedly. However, we anticipate that system explanations will prove more useful than those from the W algorithm, especially to naive users. It is likely that there will be alternative explanations, corresponding to different traversals of the tree of W algorithm deductions, and that full explanations will include repetitive information. The system will seek to provide succinct explanations by minimising the explanation length, using explanation pruning heuristics whose choice will be guided by our observations of human behaviour.

Acknowledgements

We wish to thank Andrew Cook, Mohammad Hamdan, Kevin Hammond, Hans-Wolfgang Loidl, Robert Pointon and Joe Wells for taking part in our experiments.

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

References

Brooks,R. (1983) Towards a Theory of the Comprehension of Computer Programs, *International Journal of Man-Machine Studies*, 18:543-554

Michaelson,G. (1995), *Elementary Standard ML*, UCL Press

Milner,R. (1978) A Theory of Type Polymorphism in Programming, *Journal of Computer and Systems Sciences*, 17(3):348-375

Milner,R., Tofte,M., Harper,R., and McQueen,D. (1997) *The Definition of Standard ML: Revised 1997*, MIT Press

Peyton Jones,S.L. et al (1998) *Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language*, `http://www.haskell.org`

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

Yang,J. and Michaelson,G. (2000) A visualisation of polymorphic type checking, *Journal of Functional Programming*, Vol. 10, No. 1:57-75