ML-TID

A Type Inference Debugger for ML in Education

Ben Thorner
University of Cambridge
ben.thorner@cl.cam.ac.uk

Kathryn Gray
University of Cambridge
kathryn.gray@cl.cam.ac.uk

Abstract

Student programmers can struggle to understand how their code is wrong, while expert programmers become frustrated when the safety mechanisms of a language render programming more cumbersome. Across the spectrum of languages, those with type inference encourage the user to elide information that would otherwise catch errors at their source, making them more difficult to resolve.

In this paper, we present a novel debugging tool for ML that should lower the barrier to entry. ML-TID[1] exposes a standard debugging UI that allows the user to step-through the ML type inference algorithm in a controlled environment. We present details of the concept and implementation and results from a study of 20 undergraduates.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging, Programming Environments

General Terms Algorithms, Languages

Keywords Type-Inference, Type-Debugging

1. Introduction

Functional languages with implicit types are widely acknowledged to have usability issues for novice programmers. While the expert can avoid annotating every declaration with a type, a novice user is faced with understanding complex inference and unification algorithms. Part of the problem is that type error messages will often reference type variables and other syntax not present in the original program text, as in the following session with the CML interpreter.

```
- fun double f x = f f;
# Type clash  in: (f f)
# Looking  for a: 'a
# I have found a: 'a -> 'b
```

Implicit types are also found in isolated declarations of mainstream languages such as Bluespec and Haskell. Isolating the use of type inference limits the extent to which erroneous types may propagate, whilst retaining the advantage of reduced boilerplate. Conversely, inferred types of nested declarations are only found to be incorrect at the point of use. The following error message is taken from the SML/NJ interpreter and refers to the example program in Listing 1.

Listing 1. Erroneous Permutation Function

```
fun   perms [] = [[]]
    | perms (x::xs) = let fun
            map f [] = []
          | map f (y::ys) = (f y)::(map f ys)
        in let fun
            inject [] = [[x]]
          | inject (y::ys) = (x::y)::(inject ys)
        in let fun
            append [] zs = zs
          | append (y::ys) zs = y::(append ys zs)
        in let fun
            flat [] = []
          | flat (y::ys) = append y (flat ys)
        in
            flat (map inject (perms xs))
        end end end end;
```

```
# Error: right-hand-side of clause doesn't agree
# with function result type [circularity]
#
#  expression: 'Z list list
#  result type: 'Z list list list
#
# in declaration: perms = (fn nil => nil :: nil |
# :: (<pat>,<pat>) => let val <binding> in let
# <dec> in <exp> end end)
```

While the type clash occurs for the entire inductive clause of the `perms` declaration, a detailed analysis of the intermediary types shows the real error is in the definition of the local function `inject`.

Our tool allows the user to understand type errors in a debugging environment similar to that of Eclipse or Visual Studio. Each term in a declaration is associated with a type; a user can step-through the inference for their program, see how the types evolve, and compare the inferred types with their own expectations to find the source of an error. ML-TID has the dual benefit of helping novice programmers understand error reports, and at the same time form an intuition of the underlying inference and unification algorithms.

Using the above example as motivation, the next section illustrates how our tool can be used to carry out said analysis. Section 3 then gives an overview of a preliminary study of ML-TID involving 20 undergraduates, each having at least novice skill. In Section 4, we outline the novel aspects of our implementation, such as stepped interaction with the ability to go back and repeat inference steps. And in Section 6 we discuss our plans for future work on the tool. Our tool can be found at `bitbucket.org/benthorner/mltid/`.

[1] Meta Language - Type Inference Debugger

2. Concept and Design

ML-TID is the prototype implementation of a tool aiming to improve usability of ML by exposing intermediary state of the underlying type checker, both at points of error and at individual steps in the inference for a phrase. The following narrative is our expectation of a typical use case in the context of the example in Listing 1.

> A user enters the example into a standard interpreter, where it fails to compile due to a typing error; deciding the error to be non-trivial, the user copies the declaration into ML-TID.

Figure 1 shows a miniaturised screenshot of the user interface on startup. On the left are the control panel and custom editor; on the right is a pane used to display visualisations of the Abstract Syntax Tree (AST) for the declaration under examination, currently blank.

> Entering debug mode locks the editor and causes the right-hand pane to display the AST of the declaration in collapsed form; the **debug** button is used for toggling between modes.

Sub-phrases in the AST are annotated with any associated type. On entry to debug mode, we assign to each expression a fresh type variable. The types are then refined with each step of the inference process and eventually become representative of their sub-phrases.

> By repeated use of the **step** button, the user advances through the inference; each click of the button equates to moving a logical cursor between adjacent nodes of the AST.

After several steps through the program, we arrive at the situation in Figure 2. The boxed, darker highlight marks the cursor, whilst a lighter colour indicates simultaneous changes in other type annotations due to global constraints arising from the previous step. Certain labels in the tree are anomalous; these occur due to sugaring of the bare language to derive more convenient syntactic forms. As an example, the highlight in the editor pane covers the sub-phrase `[[]]` and yet the cursor presides over something entirely different.

> A tutorial session might be required to introduce the concept of a derived form; however, our experience suggests that users are seldom concerned with such deeply nested syntax.

At any point during the inference, one may revisit past steps in sequence using the **back** button, the effect of which is illustrated in Figure 3 by comparison with Figure 2. The ability to step in both directions facilitates analysis of the effects of unification; in this case, we assume the user can establish the following equivalences.

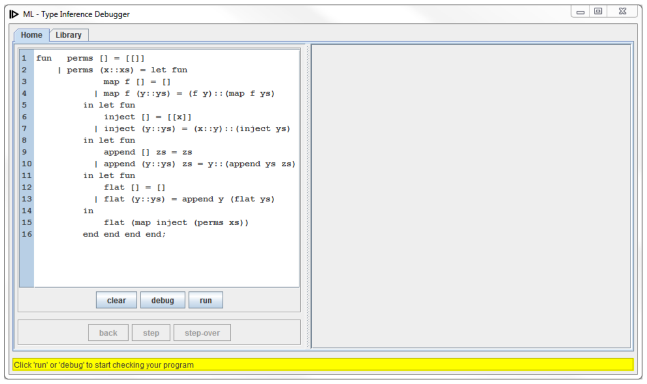
$$\gamma \text{ unifies with } \delta\,\texttt{list} \quad \text{and} \quad \epsilon \text{ unifies with } \delta\,\texttt{list}^2$$

A feature of many debuggers is the ability to affect multiple steps with a single command. The **step-over** button is used to complete inference for the sub-tree rooted at the cursor. Figure 4 illustrates the effect of using the step-over feature on entering debug mode, the cursor immediately advancing to just before the erroneous step.

> With a little guidance or experimentation, the user should see this step would normally unify the types assigned to the ground and recursive clauses of the inferred function `perms`.

Inspection shows a unification error has occurred due to a spurious type for the identifier `perms` in the inductive clause of the declaration. The only use of `perms` is to form a value in application with `inject` via the `map` function, which has the correct type; it is the injection function which is at fault and is also non-trivial to correct.

[2] Primes affixed to a letter symbolise iterations of the alphabet; these variables correspond to the same iteration and thus the primes are not shown.



```
fun   perms [] = [[]]
    | perms (x::xs) = let fun
            map f [] = []
          | map f (y::ys) = (f y)::(map f ys)
        in let fun
            inject [] = [[x]]
          | inject (y::ys) = (x::y)::(inject ys)
        in let fun
            append [] zs = zs
          | append (y::ys) zs = y::(append ys zs)
        in let fun
            flat [] = []
          | flat (y::ys) = append y (flat ys)
        in
            flat (map inject (perms xs))
        end end end end;
```

Click 'run' or 'debug' to start checking your program

Figure 1. Initial ML-TID Session

$$\texttt{inject: } (\xi\,\texttt{list}) \rightarrow ((\xi\,\texttt{list})\ \texttt{list})$$

Once a correction[3] is determined, one must toggle between modes in order to make the change. Clicking the **debug** button unlocks the editor and completes any remaining inference for the phrase, up to a point of error; a similar effect is had using the **run** button outside of debug mode. With inference successfully completed for the corrected `perms`, the context is modified to associate the name of the declared value with the inferred type, illustrated in Figure 5.

3. Evaluation and Feedback

Computer Science undergraduates from the University of Cambridge were recruited for a number of controlled experiments aiming to contrast the debugging performance of ML-TID and the CML interpreter. An experiment would begin with a brief tutorial of each application in the context of a simple example[4] demonstrating a unification error; the task was then to find and correct errors in a range of example programs divided between the two applications.

> Examples ranged from modifying `fun id x = x = x` so that the declared value had the type $\alpha \rightarrow \alpha$, to correcting a version of the permutation function familiar from Listing 1.

Each of the 20 participants had introductory experience with Core ML as part of the foundational material of the Cambridge syllabus; responses to a brief questionnaire given at the beginning of the experiments also indicate a variety of preferred interpreters[5] and some additional experience of related languages, such as Haskell and F#.

Interactive tools for programming are difficult to evaluate quantitatively, with the experience gained by each participant in solving a particular problem introducing bias when performing the same task under an alternative treatment. As a consequence, the experiments served mainly to inform subjects when completing a survey about their thoughts on using the tools, covering general preferences and comments in addition to a more detailed look at individual features.

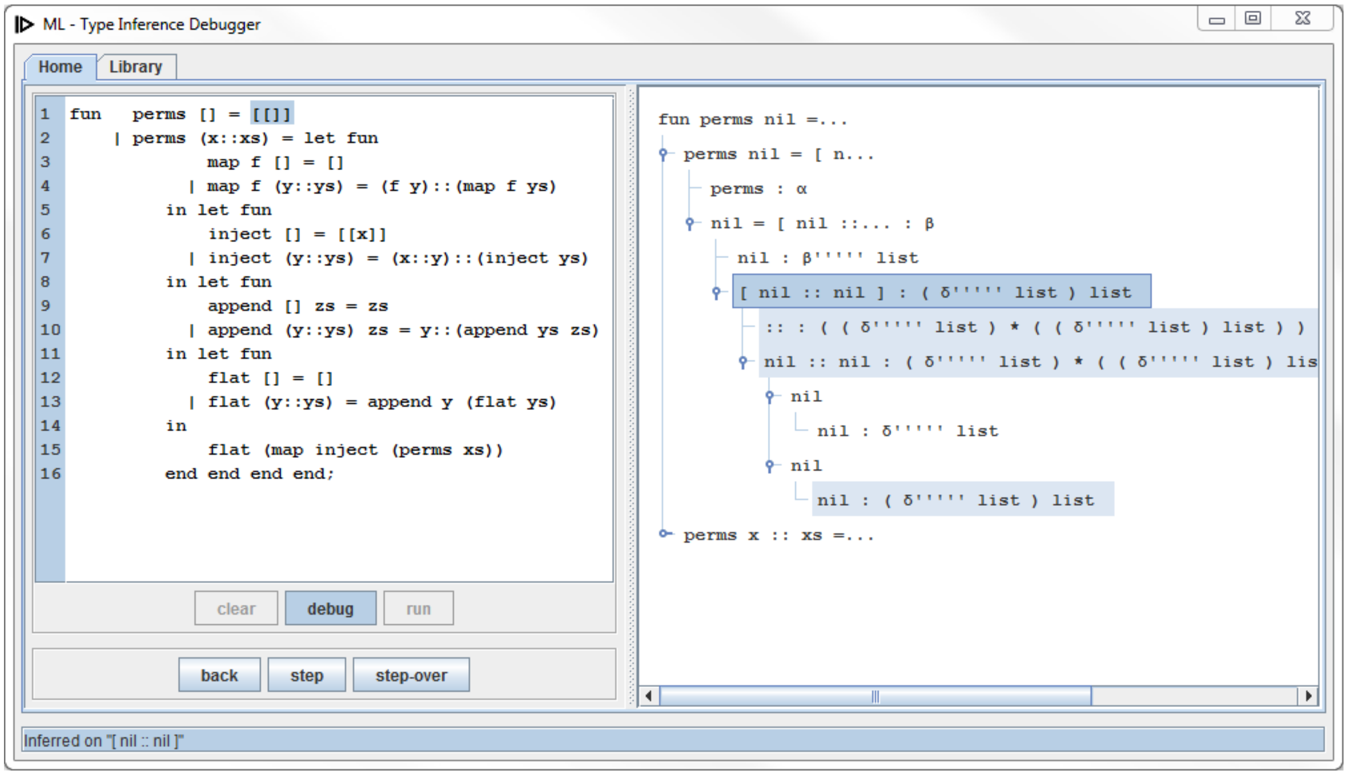
All participants thought ML-TID would be useful to some extent for debugging type errors, although three expressed no direct preference and three preferred CML. Below are some of the comments we received giving criticism and areas where ML-TID can improve.

> ...ML-TID would get very complicated with large functions...

[3] A possible correction is to replace `(x::y)` with `(x::y::ys)` in the definition of `inject`, while a non-trivial substitution giving the correct behaviour is `(x::y::ys)::(map (fn zs => y::zs) (inject ys))`.

[4] `fun bad f x = f f` was used in the tutorial for all experiments.

[5] Poly/ML (12)   CML (2)   SML/NJ (2)   Moscow ML (4)

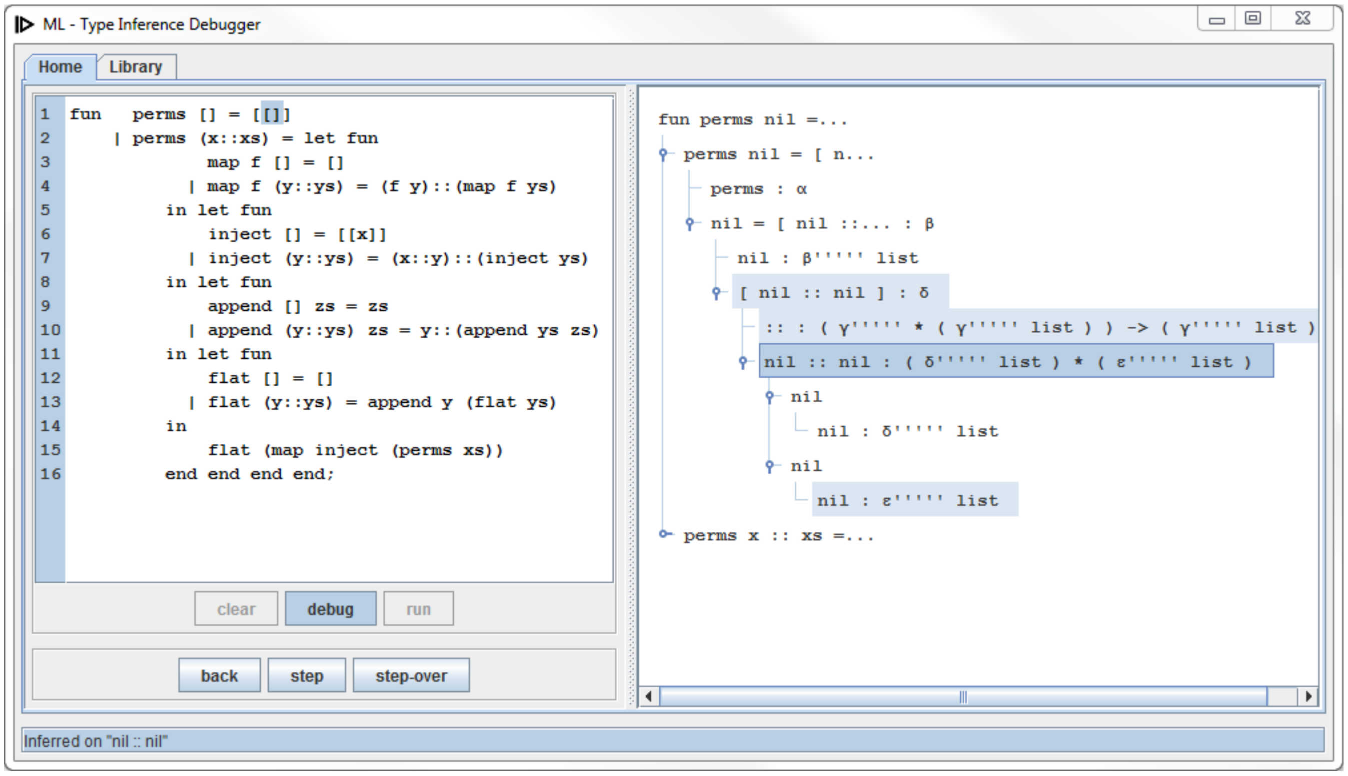


```
fun   perms [] = [[]]
    | perms (x::xs) = let fun
            map f [] = []
          | map f (y::ys) = (f y)::(map f ys)
        in let fun
            inject [] = [[x]]
          | inject (y::ys) = (x::y)::(inject ys)
        in let fun
            append [] zs = zs
          | append (y::ys) zs = y::(append ys zs)
        in let fun
            flat [] = []
          | flat (y::ys) = append y (flat ys)
        in
            flat (map inject (perms xs))
        end end end end;
```

```
fun perms nil =...
  perms nil = [ n...
    perms : α
    nil = [ nil ::... : β
      nil : β''''' list
      [ nil :: nil ] : ( δ''''' list ) list
        :: : ( ( δ''''' list ) * ( ( δ''''' list ) list ) )
        nil :: nil : ( δ''''' list ) * ( ( δ''''' list ) lis
          nil
            nil : δ''''' list
          nil
            nil : ( δ''''' list ) list
  perms x :: xs =...
```

Inferred on "[nil :: nil]"

Figure 2. Debugging with ML-TID

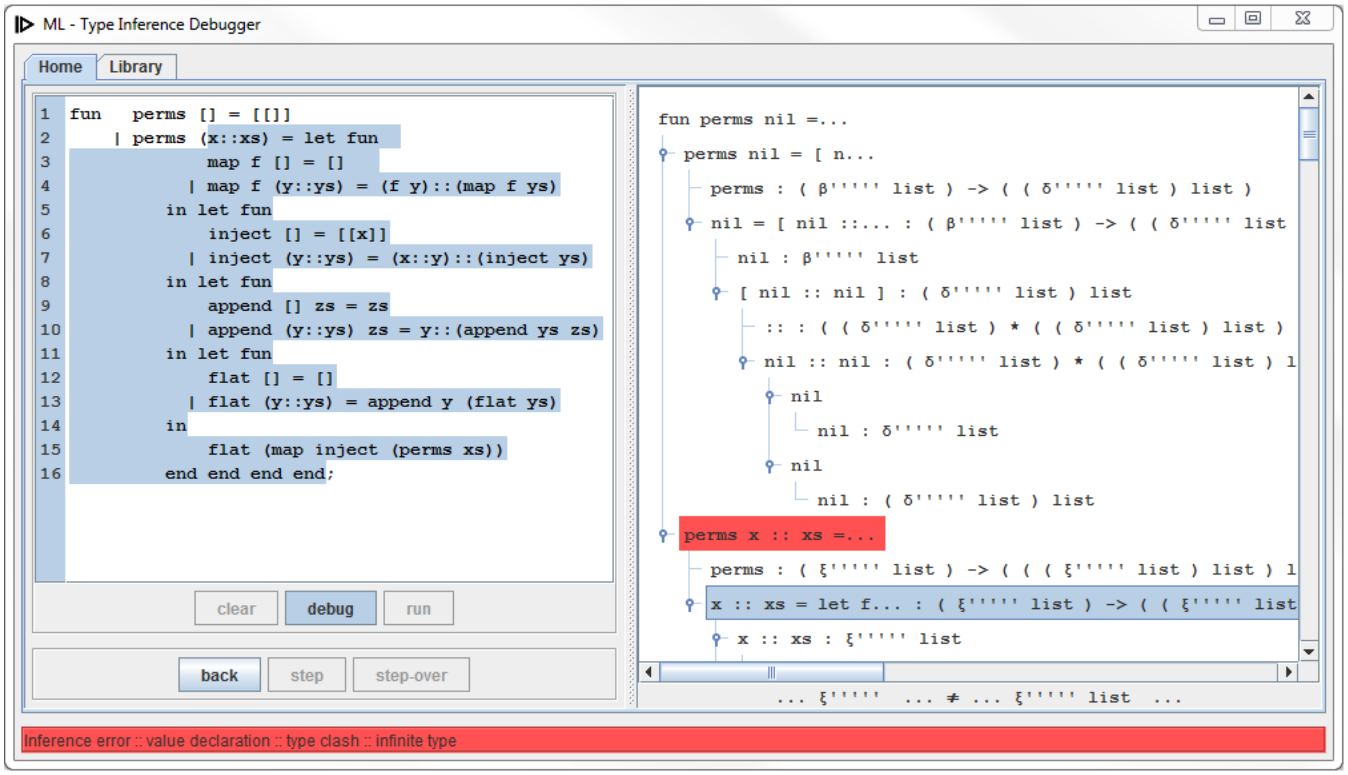


```
fun   perms [] = [[]]
    | perms (x::xs) = let fun
            map f [] = []
          | map f (y::ys) = (f y)::(map f ys)
        in let fun
            inject [] = [[x]]
          | inject (y::ys) = (x::y)::(inject ys)
        in let fun
            append [] zs = zs
          | append (y::ys) zs = y::(append ys zs)
        in let fun
            flat [] = []
          | flat (y::ys) = append y (flat ys)
        in
            flat (map inject (perms xs))
        end end end end;
```

```
fun perms nil =...
  perms nil = [ n...
    perms : α
    nil = [ nil ::... : β
      nil : β''''' list
      [ nil :: nil ] : δ
        :: : ( γ''''' * ( γ''''' list ) ) -> ( γ''''' list )
        nil :: nil : ( δ''''' list ) * ( ε''''' list )
          nil
            nil : δ''''' list
          nil
            nil : ε''''' list
  perms x :: xs =...
```

Inferred on "nil :: nil"

Figure 3. Backwards Step in ML-TID



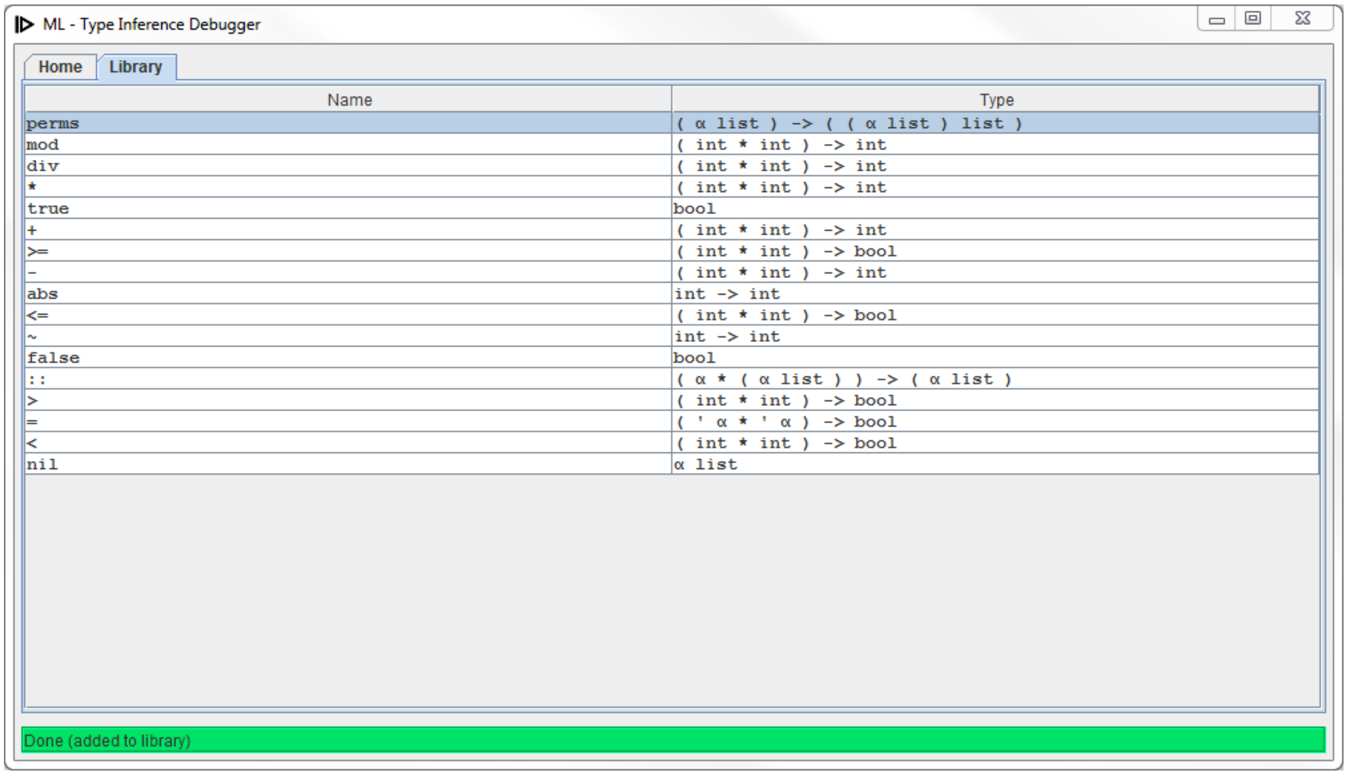
```
fun   perms [] = [[]]
    | perms (x::xs) = let fun
                map f [] = []
              | map f (y::ys) = (f y)::(map f ys)
          in let fun
                inject [] = [[x]]
              | inject (y::ys) = (x::y)::(inject ys)
          in let fun
                append [] zs = zs
              | append (y::ys) zs = y::(append ys zs)
          in let fun
                flat [] = []
              | flat (y::ys) = append y (flat ys)
          in
                flat (map inject (perms xs))
          end end end end;
```

```
fun perms nil =...
  perms nil = [ n...
    perms : ( β''''' list ) -> ( ( δ''''' list ) list )
    nil = [ nil ::... : ( β''''' list ) -> ( ( δ''''' list
      nil : β''''' list
      [ nil :: nil ] : ( δ''''' list ) list
        :: : ( ( δ''''' list ) * ( ( δ''''' list ) list )
        nil :: nil : ( δ''''' list ) * ( ( δ''''' list ) l
          nil
            nil : δ''''' list
          nil
            nil : ( δ''''' list ) list
  perms x :: xs =...
    perms : ( ξ''''' list ) -> ( ( ( ξ''''' list ) list ) l
    x :: xs = let f... : ( ξ''''' list ) -> ( ( ξ''''' list
      x :: xs : ξ''''' list
```

```
... ξ'''''  ... ≠ ... ξ''''' list  ...
```

Inference error :: value declaration :: type clash :: infinite type

Figure 4. ML-TID Step-Over Feature



Name	Type
perms	(α list) -> ((α list) list)
mod	(int * int) -> int
div	(int * int) -> int
*	(int * int) -> int
true	bool
+	(int * int) -> int
>=	(int * int) -> bool
-	(int * int) -> int
abs	int -> int
<=	(int * int) -> bool
~	int -> int
false	bool
::	(α * (α list)) -> (α list)
>	(int * int) -> bool
=	(' α * ' α) -> bool
<	(int * int) -> bool
nil	α list

Done (added to library)

Figure 5. Extended ML-TID Library

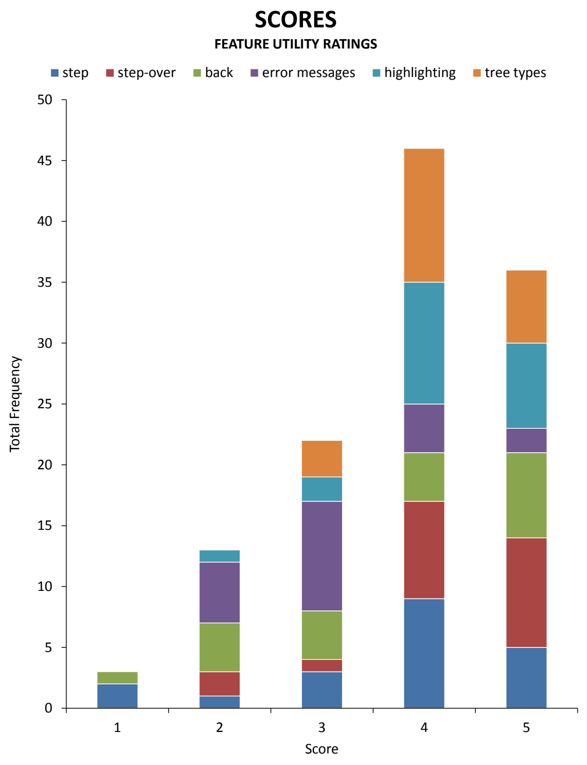


Figure 6. Survey Results for ML-TID Features

> ...[CML] was a lot simpler to look at, the tree structure [of ML-TID] was a sea of characters...

Despite these issues, a clear majority of the participants chose the current prototype as their preferred debugging tool; a sample of their feedback is shown below to lend further support to this claim.

> ...[ML-TID] showed exactly where the problem was — compared to CML's cryptic error messages...

> [ML-TID gives] additional insight into how types [are] inferred [and] helps you to think of possible solutions...

Figure 6 shows the qualitative scores assigned to a selection of ML-TID features, over all participants. Consistently popular were step-over, highlighting and type annotations in the tree; participants were observed using the former pair to immediately focus on errors, analysing the annotated types. Popularity of step and back is more variable; participants were observed employing these features together in order to contrast changes between states of the type checker. Even less popular were the custom error messages; although their concise form might have been difficult to interpret, the low score was not unexpected, given the motivation for the project.

4. Implementation

Whilst the application is self-contained, the implementation of ML-TID decouples user interaction and core functionality into separate components. The core defines an extensible architecture for representing inference statements of the form $C \vdash P \Rightarrow A$ asserting phrase P has property A under assumptions C on its free variables; properties are generally (polymorphic) types or possibly additional context. Graphical interaction with the core makes use of a simple engine interface, accepting textual strings and exposing both a mechanism for atomic inference steps and sufficient state to determine intermediary types of sub-phrases; features such as step-over and back are notably implemented as a sub-component of the GUI.

Inference statements are represented using members of the state, parsing and typing modules of the core, respectively corresponding to assumptions, phrases and (most) properties thereof; each member of the parsing module itself corresponds to a phrase in the conclusion of one or more typing rules, reflecting the delegation of inference to individual classes. The typing sub-component defines a base class extended by all logical types and supports unification of these; in addition, the module uses the wrapper design pattern to support polymorphic types and type functions, the latter instrumental in defining custom datatypes. State is essentially two mappings between strings and polymorphic or user-defined types; these are aggregated to form the context C and scoped using a mechanism similar to copy-constructors, replicating all mappings for immutable extension or modification during the inference of sub-phrases. Whilst the parser is non-trivial, a significant challenge has been defining the architecture for encapsulating parsed strings: the majority of nodes in the generated trees support atomic inference steps by implementation of a method with the following prototype.

```
public Phrase infer(Context c) throws Exception
```

Invocation of the method is recursive within a syntax tree but yields when a property changes, returning the object representing the last inferred sub-phrase; this not-so-incidentally allows the GUI to highlight the corresponding visual element, illustrated in Figure 2.

Java provides the Swing library for creation of graphical interfaces; the paradigm is to break apart the design into a number of panels, which are clearly discernible in the screenshots. The panels communicate using a synchronous event-based messaging system. Considering the various features of the user interface, one point of interest is the mechanism for exporting logical type hierarchies into strings: the objects are converted into a stream of parser tokens input to a pretty-printer, which replaces each token with a textual representation consistent with previous output, referring especially to the symbol used for each type variable object, across multiple streams. Another issue was supporting multiple and backward inference steps; whilst the former are immediate by multiple atomic steps, supporting backward steps required a logical timeline storing the necessary state to reproduce graphical elements of the interface as they appeared when the original step took place within the core.

5. Related Work

Approaches to improving usability issues of programming languages with implicit types largely correspond to one of three categories. **Inspection** systems allow the programmer to investigate the types of sub-phrases at a point of error. Bernstein and Stark [1] demonstrate a system permitting inference for open expressions; unbound identifiers are resolved as part of error messages, such that types of sub-phrases may be determined by application to a fresh (function) variable. Other examples [3] instead favour a graphical *point-and-click* method, with the added capability [2] of specifying assumed types for expressions. **Explanation** systems provide verbose output describing the inference steps leading to an error. Beaven and Stansifer [4] envisage a more interactive approach avoiding full explanation; their current system explains why a type is assigned to a phrase based on those of its sub-phrases, with additional description given to how types change due to unification. **Analytical** systems attempt to emulate manual resolution techniques. Lerner, Grossman and Chambers [5] recast the type checker as a decision procedure in a heuristic search and enumeration process for possible corrections to faulting programs; modifications not leading to an error are ranked for suggestion to the user.

Beyond the approaches given above, Jung and Michaelson [6] implement a visual programming language for a Standard ML subset, using colours and letters to represent base types and type variables, respectively; functions are entered as text for display as icons, with highlighting used to guide connection of value icons to positions representing compatible formal parameters. ML-TID falls between explanation and inspection systems, with inspection possible not only at the point of error, but also at preceding inference steps; we suggest that the ability to observe types evolve in the tree visualisation constitutes an implicit explanation of the inference algorithm.

6. Conclusions and Future Work

Our experience of evaluating ML-TID with students has shown that interactive inspection of the type checking process is preferable to the trial-and-error approach employed by many to identify the cause of a type error. Future work is now under consideration, including changes to the GUI and extended use or study of the tool.

> Based on criticisms of readability, we are exploring a different representation of type variables using a subscript notation; this avoids counting primes. Another simple extension is to add a button to complement **step-over**; this is a precursor to an alternative breakpoint scheme requested by users.

As to further use of the tool, we are looking to deploy ML-TID in a class of undergraduate students learning the Meta Language. Our hope is that we can gain a more complete understanding of the benefits of ML-TID and also investigate its usefulness as a teaching aid. The tool can be found at `bitbucket.org/benthorner/mltid/`.

Acknowledgments

ML-TID is the product of a research project for the University of Cambridge. The evaluation of the tool was much improved based on advice from Ian Davies and Richard Gibbens. We would also like to thank Gonville & Caius College for the funding that made the user study possible. Finally, our thanks go to all participants in the study and to everyone else who gave feedback on the prototype.

References

[1] K. L. Bernstein and E. W. Stark, *Debugging Type Errors*. State University of New York, 1995.

[2] A. Keane, *A tool for investigating type errors in ML programs*. University of Edinburgh, 1999.

[3] O. Chitil, F. Huch and A. Simon, *Typeview: A Tool for Understanding Type Errors*. Proc. 12th International Workshop on Implementation of Functional Languages, Pp. 63–69.

[4] M. Beaven and R. Stansifer, *Explaining Type Errors In Polymorphic Languages*. ACM Letters on Programming Languages and Systems, Vol. 2, Pp. 17–30.

[5] B. Lerner, D. Grossman and C. Chambers, *Searching for ML Type-Error Messages*. Proc. ACM Workshop on ML, 2006.

[6] Y. Jung and G. Michaelson, *A visualisation of polymorphic type checking*. ACM Journal of Functional Programming, Vol. 10, Pp. 57–75.