Explaining type errors in polymorphic languages*

Mike Beaven
Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907–7821
Internet: `mwb@cs.purdue.edu`

Ryan Stansifer
Department of Computer Sciences
University of North Texas
Denton, Texas 76203–3886
Internet: `stansifer@cs.unt.edu`

January 12, 1995

Abstract

Strongly-typed languages present programmers with compile-time feedback about the type correctness of programs. Errors during polymorphic type checking take the form of a unification failure for two types. Finding the source of the type error in the code is often difficult because the error may occur far from the spot where the inconsistency is detected. As functional languages use more and more complex type systems, the difficulty of interpreting and locating these errors will increase. To locate the source of type errors the programmer must unravel the long chain of deductions and type instantiations made during type reconstruction. This paper describes an approach that maintains the deductive steps of type inference and the reasons for type instantiations. The approach could be used in an interactive system to guide the programmer to the source of a type error or to explain why the compiler assigned a particular type to an expression.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*
General Terms: Languages
Additional Keywords and Phrases: Polymorphic type reconstruction, type errors

1 Introduction

One annoying aspect of strongly-typed languages with parametric polymorphism is that it is sometimes unclear why an expression cannot be typed. Many programmers have experienced the feeling of incredulity that sometimes arises after an expression has been rejected by the compiler. There is no reason why the compiler cannot be forthcoming about the type-checking analysis. In this paper we describe a way in which the type analysis can be explained so that the programmer may be led to the source of type errors. In particular, we consider the case of type reconstruction for all the essential constructs of the programming language ML.

The programming language ML [5] is a strongly-typed functional language with polymorphism. Its interesting type system, which has the property that the types of all identifiers can be deduced by their use [4], has made it popular and the model of type systems for other languages. Not only do we focus on this language as the object of our study, but we also have implemented our approach in that language using Standard ML of New Jersey [1].

We are inspired by the paper [7] concerning the British Nationality Act. In this paper the authors described capturing the legal requirements for establishing British citizenship in a PROLOG program. Because the requirements for British citizenship are so complex it is useful to have a system that explains

*Appeared in *ACM Letters on Programming Languages and Systems*, volume 2, numbers 1–4, March–December 1994, pages 17–30.

the steps taken to reach a conclusion. The same idea seems appropriate for the deductions made during type reconstruction.

We want to ask how the type system arrived at a particular type for an expression. Not only is the answer to this question useful when the type analysis has led to an unexpected type for an expression, but it is particularly useful in the case of type errors. In this case, type analysis has arrived at two inconsistent types and the programmer may find it difficult to understand why. After all, the programmer assumes, probably erroneously, that all the expressions are constructed correctly and have their intended type.

The place where the inconsistency was found may not be the part of the program where the mistake was made. It is the nature of the type system that widely separated parts of the program affect the types that expressions have. Any part of the program might be the mistake from the point of view of programmer, but where the inconsistency is found depends on the arbitrary order of the traversal of the syntax tree during type analysis.

Some interesting information may be obtained by examining all the contextual evidence. This is the approach of Johnson and Walz [3]. The usage that is in the minority is a candidate for the mistake. But it is not clear how often the minority can be isolated, since many errors are of the form where there is one correct usage and one incorrect usage. Also, there are ways of introducing errors where the minority usage may be the correct type despite the preponderance of the majority usages. One way is for an error in the program to be copied to many locations in the program using a text editor. Another, trickier, situation would be when an expression bound to a variable in a `let` statement is used so that it is incorrectly typed in many locations while the one location which causes it to not be well-typed is the correct usage. The approach we take here is entirely different. We do not try to find the "wrong" context, instead we want the ability to explain the context, right or wrong. The two approaches might be used together in some future, heuristic system that both explains deductions and suggests where the error may be found.

In his 1986 POPL paper Wand [9] suggested aiding programmers in their search for the source of type errors by presenting them with a list of the expression fragments which were the cause of binding type variables during the type reconstruction. His approach called for the collection of application subexpressions during the unification step of the type analysis. When a type error is detected a list of application sites is issued, one of which should be the cause of the error.

We improve on Wand's work by doing more and finer analysis. We maintain the deductions made by the reconstruction algorithm along with the location of the source fragments leading to the individual bindings during the unification process. This forms an "explanation space" which can be traversed to explain individual deduction steps and the flow of information from one part of the program to another. In addition, we also consider the effects of the Milner `let`-statement, which greatly complicates the deductive process.

2 Type reconstruction

We are interested in a simple functional language. For the sake of concreteness, we use ML-like syntax. The syntax for the language is shown in figure 1. Besides function definition and function application, the language has constructs for pairing, conditional expressions, and `let` expressions. Consistent with the practice in Standard ML of NJ we replace subexpressions of the language with the character `#` when we do not have space to show more than the top part of a particular expression.

The expressions of the language we consider to be meaningful are those that have a type from the collection described by the grammar in figure 2. Type variables will be written `'a`, `'b`, etc.

Type reconstruction is the process of deducing the type of an expression from context. In a type checker for a language like Pascal, identifiers must have an explicitly given type. These are then checked for consistent usage throughout the program. A type reconstruction algorithm does not merely check whether the declared types are combined consistently, but finds a consistent type if there is one. Identifiers are initially given a type variable as their type. A type variable stands for any type and, thus, represents the totally unconstrained type. If the type-checking requirements do not constrain the types of the expressions completely, some type variables will be left unspecified. This is how a polymorphic type is achieved.

In [4] it is proved that there is a theorem prover for the complete set of typing rules for the language. We call this algorithm `TypeOf`. A modified version of the original algorithm is the basis for our explanation system. The function `TypeOf` takes an expression, a type assignment, and a substitution as input and

$\langle expr \rangle$	::=	$\langle identifier \rangle$	
	|	$\langle integer \rangle$	integer constants
	|	$\langle boolean\ value \rangle$	boolean constants
	|	`(` $\langle expr \rangle$ `,` $\langle expr \rangle$ `)`	pair
	|	$\langle expr \rangle$ $\langle expr \rangle$	function application
	|	`fn` $\langle identifier \rangle$ `=>` $\langle expr \rangle$	function definition
	|	`if` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$	conditional expression
	|	`let` $\langle identifier \rangle$ `=` $\langle expr \rangle$ `in` $\langle expr \rangle$ `end`	`let` expression

Figure 1: The syntax of a simple functional language

$\langle type \rangle$	::=	$\langle type\ variable \rangle$	
	|	`int`	type of integers
	|	`bool`	type of boolean values
	|	$\langle type \rangle$ `*` $\langle type \rangle$	cartesian product
	|	$\langle type \rangle$ `->` $\langle type \rangle$	function type

Figure 2: The syntax of types

produces a type and a substitution. `TypeOf` does a postorder traversal of the syntax tree and enforces the typing rules for each construct, essentially building a proof tree of typing judgments. More details about this algorithm can be found elsewhere [2, 6].

A crucial insight in designing a theorem prover lies in recognizing that unification can be used to insure that two types are the same. For example, the typing rule for function application requires the domain of the function to be the same as the type of the actual argument. The algorithm to find a substitution (if one exists) that makes two types equal is called unification. Any substitution can be used to specialize a proof tree making the type of the conclusion an instance of the original type. Thus as long as two types are unifiable then appropriate proof trees can be found that result in the appropriate types being syntactically identical.

There are many ways to represent a substitution:

$$\theta = \{v_1 \mapsto t_1,\ v_2 \mapsto t_2,\ \ldots,\ v_n \mapsto t_n\}$$

In particular, one could represent a substitution as a function, or one could use a list of pairs. Normally this issue is of no great consequence. However, for purposes of modifying the unification function to keep extra information, we must have a substitution represented as a list. For reasons of efficiency it is common that the list:

$[\ (v_n,\ t_n),\ \ldots,\ (v_2,\ t_2),\ (v_1,\ t_1)\ ]$

represent the substitution:

$$\{v_1 \mapsto t_1,\ v_2 \mapsto \theta_1 t_2,\ \ldots,\ v_n \mapsto \theta_{n-1} t_n\}$$

where θ_i is the substitution:

$$\{v_1 \mapsto t_1,\ v_2 \mapsto \theta_1 t_2,\ \ldots,\ v_i \mapsto \theta_{i-1} t_i\}$$

With this list representation, one can specialize a substitution simply by adding a further atomic binding $(v_i,\ t_i)$. And there is no need to update the previous bindings. For our purposes the crucial value of this representation is that each atomic binding can be explained by one particular unification. Otherwise possibly several unifications would contribute to the value of a single type variable. In this case one would have to keep a list of reasons for each binding, like in Wand [9], and it would be impossible to explain the instantiation of a type step by step.

3 Approach

The approach we take has the following elements: annotations for the parse tree, augmented substitutions, augmented type assignments, and explanation functions which use this information.

The information we add to the syntax tree is most often the types of subexpressions captured when the node was analyzed. Each syntactic construct requires an individualized set of information. Basic constructs, integers and truth-values, have no need to store extra information because of their simple structures. Conditionals and pairs need only store the types of their subexpressions. The `let` statement `let` i `=` e `in` b `end` stores the type of its body b and the type of e.

The data structure representing function definition (`fn` i `=>` e) stores the type of its body and the original type variable assigned to the formal parameter. The original type variable is the starting point of all the atomic bindings that account for the instantiation of the domain.

Function application must keep two pieces of information: the two types which are unified during type analysis. With this information it is possible to report what unification took place because of this application construct.

An identifier must also keep two pieces of information. For the same reason as the case of function definition, an identifier keeps the type variable originally assigned to it in the type assignment. The identifier construct must also keep a list of type variable renamings. These are the renamings of all the generic type variables in the type of the identifier. Generic type variables are introduced by the `let` statement to implement the special effect the `let` statement has on polymorphic types.

Another part of the information kept is in substitution. The data structure used for substitution is a list of all atomic bindings of types to type variables made during the unification process. The analysis functions use these atomic bindings to trace the evolution of types from their original form to their final form. These bindings are augmented by a pointer into the annotated expression tree and a tag. The expression referenced by the pointer is the one responsible for the unification call which bound the type variable in that atomic binding. The tag adds further information so that the analysis can distinguish between the two unification calls in the typing of the conditional statement (one to unify the type of the test expression with `boolean` and the other to ensure that the branches of the conditional have the same type). When `TypeOf` calls unification it passes the current location in the syntax tree which is stored in any atomic binding resulting from this invocation.

Normally, running the function `TypeOf` produces a type for an expression. `TypeOf` does a recursive traversal of the syntax tree. If it encounters a type inconsistency, it stops and reports an error. This can occur only at one of the calls to unification.

Our modified `TypeOf` decorates the parse tree during the recursive traversal of the syntax tree. All calls to unification in our system keep pointers to the appropriate places in the tree. When an inconsistency is detected, the program reports on the problem. Since the problem is caused by the types of two expressions failing to unify, we ask the following sort of question (twice): why does the expression e have type τ?

We have written a function `Why` which answers this question given the decorated parse tree and the augmented substitution at the point the analysis stopped. This program's main purpose is to explain the typing rules as specifically used in the expression of interest.

For example, the expression

```
(fn a => +((fn b=>if b then b else a) true, 3))
```

does not have a type in ML. The error is detected in the body of the outermost function `+(#,3)`, the plus function applied to two arguments. At this point the type analysis detects that the domain of plus `int*int` does not match the type of the actual argument. Here is the error report from Standard ML of NJ:

```
Standard ML of New Jersey, Version 75, November 11, 1991
- (fn a => +((fn b=>if b then b else a) true,3));
std_in:1.10-1.45 Error: operator and operand don't agree (tycon mismatch)
  operator domain: int * int
  operand:         bool * int
  in expression:
    + (((fn <rule>)) true,3)
```

In such a case the programmer may find it clear that the operator plus has the right domain, and that the argument (operand) is wrong. But it may not be clear why the system thinks the type of the actual argument is `bool*int`. Hence we are interested in the question: Why does `(# #, 3)` have type `bool*int`? This is the kind of question that the function `Why` answers.

One after another `Why` explains the typing rules for the constructs. For this example these constructs are listed in the table below.

pair	`(# #, 3)`
function application	`(fn b => #) true`
function definition	`(fn b => if #)`
conditional	`if b then b else a`

(The actual output of the explanation system for this example is shown more fully in figure 3.) Finally `Why` turns to the explanation of the types of the identifiers `b` and `a` in the conditional. Structurally all identifiers are alike. They get their types from the environment. The variable `b` was originally assigned type `'b`. But that does not completely answer the question: How did `'b` come to be assigned the type `bool`? This is the question that the function `How` answers.

The function `How` has a different sort of task than `Why`. It explains the flow of information from one point of the program to another. In the case of the previous example, `How` must explain how `'b` came to be bound to `bool`. From information in the substitution we know `'b` was bound to `bool` to meet the requirement that the conditional test be of type `bool`. We jump then to that node in the syntax tree. At that point further structural information might be asked of `Why`.

Structural explanations given by `Why` follow the structure of the proof tree of typing judgements (built by the `TypeOf` function—see section 2) using the typing rules. Fortunately the structure of the proof tree corresponds closely to the parse tree of the expression. These explanations may require `How` to explain the flow of information around the expression. These `How` explanations break into two levels. The function `How` first explains the instantiation of types during the progress of the type reconstruction by following the trail of atomic bindings in the substitution. Secondly, each atomic binding maps to some node in the annotated syntax tree. The constructs in these nodes were ones in which the analysis required a call to unification. There are two such nodes in our language: the function application, which makes one call to unification, and the conditional expression, which makes two. The function `How` explains the circumstances which required the unification. The further explanation of these circumstances is now a structural question.

It is important to note that `How` can never leap to a part of the syntax tree that has not been annotated. All pointers to the syntax tree are to places where unification is called. The `TypeOf` function always annotates the node first and then calls the unification function. The recursive annotation of the subtrees has already been completed.

We have implemented `Why` and `How` to print a depth-first explanation of every piece of the deduction. (Loops are detected and broken in an ad hoc manner.) The output of `Why` and `How` is shown in figures 3, 5 and 6. Clearly such a full-length explanation is not suitable as an error message. We envision some sort of programmer interaction with the system to navigate through the explanation space. This interaction may go like this: The system explains one step of the deduction. This deduction may depend on zero, one, or more other deductions. The programmer may find one of deductions dubious, and can choose to investigate only that deduction. For instance, in the prior example with the function `plus`, the programmer will probably be satisfied with the type of `+`, and can concentrated on the type of its actual argument. With a well-designed interface the programmer could quickly focus on the part of the program that is contrary to the programmer's understanding of the program. In the actual output of the explanation system questions that can be further explored are marked with `**`, and the elaboration is indented.

Some type errors require the process of explanation to continue with structural analysis of the program after following a binding to another part of the program. The next example is designed to illustrate this. A consequence is that the explanation may jump back to a part of the expression which has already been explained, and so the explanation does not end.

The potentially infinite explanation space appears to be a major shortcoming, so it is necessary to justify this approach. It would seem that all explanations could be unraveled down to their most primitive pieces, because the type reconstruction process goes through certain, finite steps to reach the conclusion. This is

```
A type error was detected in the application >> (+ (#,3)) <<.
The domain of the function >> + << is not unifiable
with the type of the argument >> ((# true),3) <<.
Domain of function is >> (int*int) <<.
The argument has type >> (bool*int) <<.
**Why does the function >> + << have type >> ((int*int)->int) <<?
  The identifier >> + << was assigned type >> ((int*int)->int) <<
  as part of the initial environment.
**Why does the argument >> ((# true),3) << have type >> (bool*int) <<?
  Type of the pair >> ((# true),3) << is determined by type of each element.
  **Why does the first element >> ((fn b => #) true) << have type >> bool <<?
    The type of an application is the range of the function.
    The function >> (fn b => if #) << has type >> (bool->bool) <<.
    **Why does the function >> (fn b => if #) << have type >> (bool->bool) <<?
      The type of a function definition is determined by the type
      inferred for the formal parameter >> b << and by the
      type of the function body >> if b then b else a <<.
      **Why does formal parameter >> b << have type >> bool <<?
        The type of a formal parameter is inferred from its use.
        The variable >> b << was initially assigned type variable >> 'b <<.
        **How did type variable >> 'b << come to be bound to type >> bool <<?
          This binding arose during analysis of >> if b then b else a <<.
          Since the expression >> b << must have type >> bool << and
          the type variable >> 'b << is its type,
          then the type variable >> 'b << must stand for >> bool <<.
          **Why does the expression >> b << have type >> bool <<?
            [Elided.]
      **Why does function's body >> if b then b else a << have type >> bool <<?
        The type of a conditional is determined by the
        the types of its branches which must unify.
        **Why does the "then" branch >> b << have type >> bool <<?
          [Elided.]
        **Why does the "else" branch >> a << have type >> bool <<?
          [Elided.]
    **Why does the second element >> 3 << have type >> int <<?
    All integer constants have type >> int <<.
```

Figure 3: Explanation space for `(fn a=> +((fn b=>if b then b else a) true, 3))`

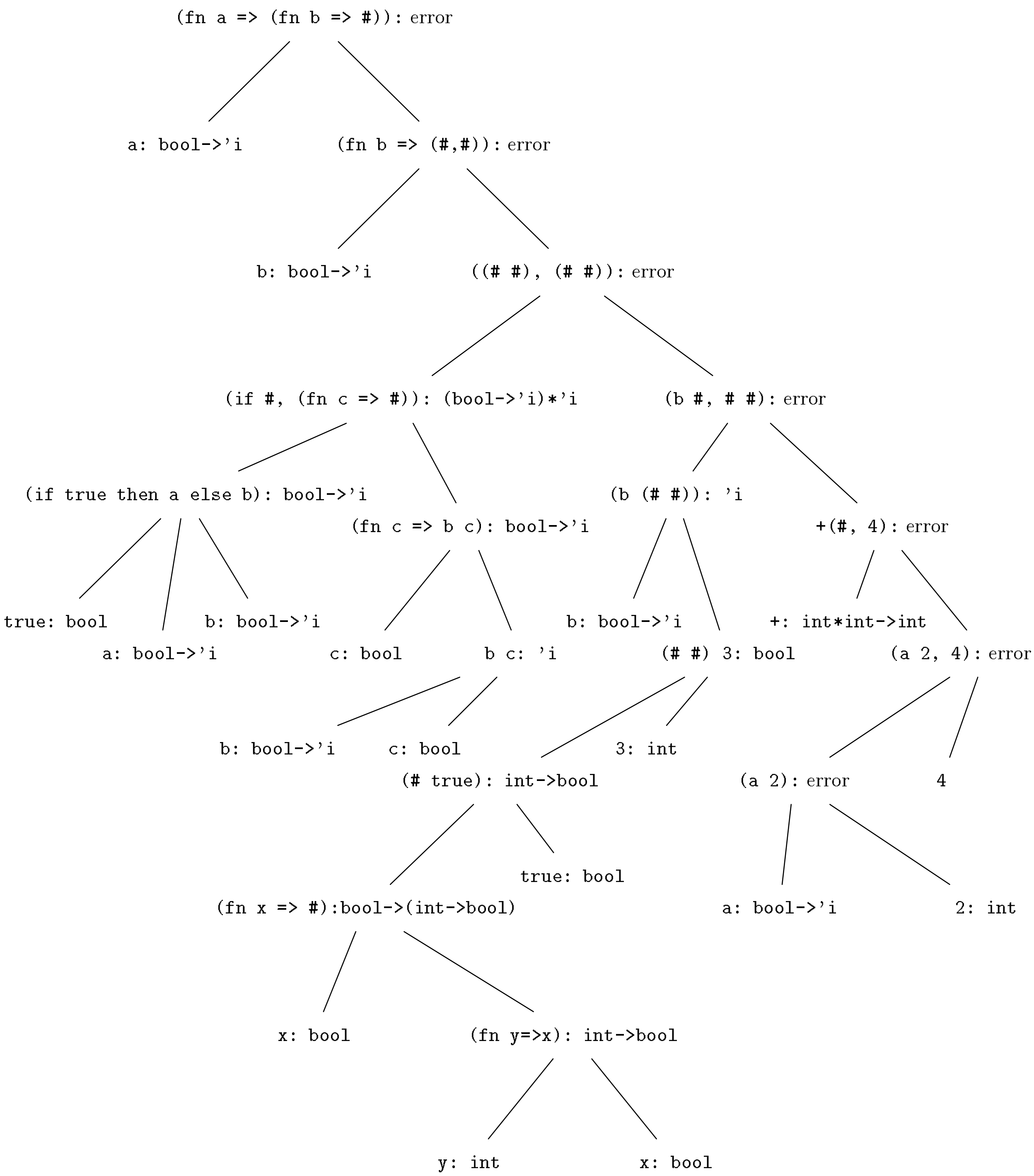

Figure 4: Syntax tree for (fn a => (fn b => ((if #,#),((b #),(+ #)))))

so, and by keeping the then-current substitution at each point in the program it is possible to trace back the steps one by one. Each piece of the explanation would begin: "at the time of the unification the types were ..." The then-current substitution would be used to explain the specifics. The explanation would depend on the order of the type analysis. But the programmer does not want to know about the implementation of the type analysis, nor about intermediate steps in the analysis. The explanation is more coherent from the perspective of the final substitution. And, in fact, the type error may only be comprehensible from the global point of view. After all, each local step is perfectly consistent. So we have chosen to explain each piece of the process from the perspective of the final substitution.

We consider the following expression:

```
(fn a => (fn b => (
     (if true then a else b, (fn c => b c)),
     (b ((fn x => fn y => y) 3 true), +(a 2, 4))
)))
```

This rather long expression containing a pair of pairs (we do not have 4-tuples in the simple language) is best comprehended by examining the syntax tree in figure 4. The expression cannot be typed. The explanation space for this figure is given in figure 5. Each of the elements of the pairs contributes a binding to the type of the formal parameter `'a`. The third binding (`'c` to `bool`) is the one we assume is suspicious. The appropriate question is asked in line 18 in figure 5. Why the expression

```
b ((fn x => fn y => y) 3)
```

introduces the type `bool` can only be understood by further analysis. The crucial question is asked in line 40: why does `x` have type bool? Actually the system does not answer the question at this point because it detects that it has answered the question before in the context of explaining the binding of the type variable `'a`. That part of the analysis has been omitted from figure 5 for lack of space. The interesting part goes like this:

```
**Why does formal parameter >> x << have type >> bool <<?
  The type of a formal parameter is inferred from its use.
  The variable >> x << was initially assigned type variable >> 'e <<.
  **How did type variable >> 'e << come to be bound to type >> bool <<?
    This binding arose during analysis of application >> ((fn x=>#) true) <<.
    The type >> ('e->('f->'e)) << was (at the time of analysis) the type
    of the function >> (fn x => (fn y => x)) <<.
    The type of the argument >> true <<  was determined to be >> bool <<.
```

Therefore, because the function `(fn x=>#)` was applied to the argument `true` its result is a function which returns a boolean value. Thus, as we have seen in from lines 19–26 of figure 5, `b` must be a function with domain `bool`. Since `a` has the same type as `b` (line 10), applying `a` to an integer is a type error.

4 Milner `let` construct

The `let` construct complicates the explanation of how an identifier's type gets instantiated. Without `let` all identifiers have an initial type variable assigned to them during the analysis of some function definition. `How` describes the instantiation of this type variable by gathering a list of all the atomic bindings that affect it. In the `let` expression `let` i `=` e `in` b, the identifier i is initially assigned not just a type variable but a type (the type of e), and this type may contain generic variables. The occurrences of i in the body b then get their initial types by renaming the generic variables into new unused type variables, which are distinct for distinct occurrences. To explain the type of a `let`-bound identifier it is necessary to know to which type variables the generic variables have been bound. So this information is kept in the syntax tree for every identifier. `How` checks this list to see if any generics were instantiated and apprises the user of all the name changes. Then it proceeds to explain the instantiation of the type as before.

Consider the following example:

```
let f = (fn x=>x) in (f 3, f true) end
```

```
A type error was detected in the application >> (a 2) <<.
The domain of function >> a << is not unifiable with type of argument >> 2 <<.
Domain of function is >> bool <<.
The argument has type >> int <<.
**Why does the function >> a << have type >> (bool->'i) <<?
  The type of a bound identifier is inferred from its use.
  The variable >> a << was introduced in function >> (fn a=>(fn b=>#)) <<.
  The variable >> a << was initially assigned type variable >> 'a <<.
  Type (bool->'i) is obtained for >> 'a << via the following 4 bindings:
  Type variable >> 'a << is bound to type >> 'b <<.
  Type variable >> 'b << is bound to type >> ('c->'d) <<.
  Type variable >> 'c << is bound to type >> bool <<.
  Type variable >> 'd << is bound to type >> 'i <<.
  **How did type variable >> 'a << come to be bound to type >> 'b <<?
    [Elided.]
  **How did type variable >> 'b << come to be bound to type >> ('c->'d) <<?
    [Elided.]
  **How did type variable >> 'c << come to be bound to type >> bool <<?
    This binding arose during analysis of application >> (b (# 3)) <<.
    The type >> ('c->'d) << was (at the time of analysis) the type
    of the function >> b <<.
    The type of the argument >> ((# true) 3) <<
    was determined to be >> bool <<.
    The domain of the function must be equal to the type of the argument.
    The binding arose during the unification of the type >> ('c->'d) <<
    with the type >> (bool->'i) <<.
    Type variable >> 'i << was introduced to stand for range of the function.
    **Why does the function >> b << have type >> (bool->'i) <<?
      [Elided.]
    **Why does the argument >> ((# true) 3) << have type >> bool <<?
      The type of an application is the range of the function.
      The function >> ((fn x=>#) true) << has type >> (int->bool) <<.
      **Why does >> ((fn x=>#) true) << have type >> (int->bool) <<?
        The type of an application is the range of the function.
        The function >> (fn x=>(fn y=>x)) << has type >> (bool->(int->bool)) <<.
        **Why does >> (fn x=>(fn y=>x)) << have type >> (bool->(int->bool)) <<?
          The type of a function definition is determined by the type
          inferred for the formal parameter >> x << and by the
          type of the function body >> (fn y=>x) <<.
          **Why does formal parameter >> x << have type >> bool << ?
            [Elided.]
          **Why does function body >> (fn y=>x) << have type >> (int->bool) <<?
            [Elided.]
  **How did type variable >> 'd << come to be bound to type >> 'i <<?
    [Elided.]
**Why does the argument >> 2 << have type >> int <<?
  All integer constants have type >> int <<.
```

Figure 5: Explanation space for `(fn a=>(fn b=>((if #,(fn c=>#)),((b #),(+ #)))))`

```
The type of a "let" expression is the same as the type of the body.
**Why does the body >> ((f 3),(f true)) << have type >> (int*bool) <<?
 Type of pair >> ((f 3),(f true)) << is determined by type of each element.
 **Why does the first element >> (f 3) << have type >> int <<?
   The type of an application is the range of the function.
   The function >> f << has type >> (int->int) <<.
   **Why does the function >> f << have type >> (int->int) <<?
     The identifier >> f << was introduced in the
     "let"-binding >> let f = (fn x => x) in (#,#) end <<.
     The analysis of the expression >> (fn x => x) << determined
     its type to be >> ('a->'a) <<.
     Since  >>f<< stands for this expression, it initially received this type.
     After renaming the generic type variable:  >> 'a << with a new variable,
     we get the type >> ('b->'b) <<.
     Type (int->int) is obtained for >> ('b->'b) << via the following binding:
     Type variable >> 'b << is bound to type >> int <<.
     **How did type variable >> 'b << come to be bound to type >> int <<?
       This binding arose during analysis of application >> (f 3) <<.
       The type >> ('b->'b) << was (at the time of analysis) the type
       of the function >> f <<.
       The type of the argument >> 3 << was determined to be >> int <<.
       The domain of the function must be equal to the type of the argument.
       The binding arose during the unification of the type >> ('b->'b) <<
       with the type >> (int->'c) <<.
       Type variable >> 'c << was introduced to stand for range of function.
       **Why does the function >> f << have type >> (int->int) <<?
         [Elided.]
       **Why does the argument >> 3 << have type >> int <<?
         All integer constants have type >> int <<.
     **Why does the expression >> (fn x => x) << have type >> ('a->'a) <<?
       The type of a function definition is determined by the type
       inferred for the formal parameter >> x << and by the
       type of the function body >> x <<.
       **Why does formal parameter >> x << have type >> 'a <<?
         The type of the formal parameter >> x << remained unconstrained from
         its initial assignment of type variable >> 'a <<.
       **Why does the function definition's body >> x << have type >> 'a <<?
         The type of the bound identifier >> x << has remained unchanged from
         its initial assignment of type variable >> 'a << in
         the function definition >> (fn x => x) <<.
 **Why does the second element >> (f true) << have type >> bool <<?
   The type of an application is the range of the function.
   The function >> f << has type >> (bool->bool) <<.
   **Why does the function >> f << have type >> (bool->bool) <<?
     [Similar.]
```

Figure 6: Explanation space for `let f=(fn x=>x) in (f 3,f true) end;`

This expression is well-typed. We use a well-typed example this time to emphasize that the explanations produced by `Why` are not only useful for explaining type errors, but are also useful for understanding the type of any expression. So we ask here: why does the expression have type `int*bool`? The output of `Why` and `How` is given in the figure 6. The explanation begins with a statement explaining that the type of a `let` construct is the type of its body, which in this case is the pair `(f 3,f true)`. The explanation then describes why the first element is of type int, and when it describes the type of `f` it must describe the generic nature of its type `'a->'a`. Note that `How` then states that the generic type variable `'a` is renamed to `'b`, and then proceeds to explain how the type `'b->'b` is unified with the type `int->'c`.

5 Conclusion

Our goal has been to develop an approach for maintaining the deductive steps of type inference in order to provide the programmer with meaningful explanations about the circumstances of type errors. The approach we have taken is to explain the reasoning of the type reconstruction process in relation to the program. In many cases this will lead the programmer to the insight necessary to repair the program. It is also useful for explaining the type the system finds for a well-typed expression. Not only is this useful for programmers unfamiliar with polymorphic type reconstruction, but there are even programming errors in which the type system finds some unexpected way to deduce a type for an expression.

We have described modifications to the traditional type analysis algorithm for Milner `let`-style polymorphism. These modifications store critical parts of the analysis process. We keep track of the atomic bindings comprising a substitution. This permits a more refined explanation of the deductions than in Wand [9]. We have described two functions `Why` and `How` which make use of this information to explain the steps in the inference of the type for expressions. This is useful for understanding type errors and understanding how a type is obtained for a particular expression.

We chose to present the explanation to the programmer from the perspective reached at the point the type inconsistency was detected. It seemed that this choice made the explanations more understandable than explanations using bindings current at different times during the type analysis. In fact, it is only by consistently using the final bindings that the inconsistency could be carried to the part of the expression which the programmer considered to be in error.

With more insight into the kinds of errors that are made, improved explanations could guide novice and expert programmers to the relevant parts of the program more easily. There are several opportunities to synthesize special cases into higher-level explanations that would be more meaningful to the programmer. With more experience, navigation through the explanation space could be improved.

Acknowledgements. The anonymous referees have aided considerably in improving this paper. One, in particular, devoted extraordinary effort in analyzing the paper and making suggestions for improvements. This referee has also brought to our attention a Master's thesis on this topic [8].

References

[1] Appel, A. W., and MacQueen, D. B. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programing* (Berlin, 1991), M. Wirsing, Ed., Springer-Verlag, pp. 1–13.

[2] Cardelli, L. Basic polymorphic type checking. *Science of Computer Programming 8*, 2 (April 1987), 147–172.

[3] Johnson, G. F., and Walz, J. A. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages* (January 1986), pp. 44–57.

[4] Milner, R. A theory of type polymorphism. *Journal of Computer and System Sciences 17* (1978), 348–375.

[5] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML*. MIT Press, 1990.

[6] Reade, C. *Elements of Functional Programming.* Addison Wesley, 1989.

[7] Sergot, M. J., Sadri, F., Kowalski, R. A., Kriwaaczek, F., Hammond, P., and Cory, H. T. The British Nationality Act. *Communications of the ACM 29*, 5 (May 1986), 370–386.

[8] Soosaipillai, H. An explanation based polymorphic type check for Standard ML. Master's thesis, Heriot-Watt Univeristy, Edinburgh, U.K., 1990.

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