1992

Why that expression has this type, and how it got it

Mike Beaven

Report Number:
92-044

WHY THAT EXPRESSION HAS THIS TYPE, AND HOW IT GOT IT

Mike Beaven
Ryan Stansifer

CSD-TR-92-044
July 1992

Why that expression has this type, and how it got it

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: ryan@ponder.csci.unt.edu

July 29, 1992

Abstract

Strongly-typed languages present programmers with compile-time feedback about the type correctness of programs. Errors during polymorphic type checking most often take the form of a unification failure for two types. Finding the source of the type error in the code is possibly very difficult because the error may occur far from the spot where the inconsistency is detected. As functional languages utilize 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. We describe two functions that, given an expression and some type obtained by type reconstruction, answer the question why that expression has this type and how it got it.

1 Introduction

One annoying aspect of strongly-typed languages with parametric polymorphism is that it is often not clear why an expression cannot be well-typed. Many programmers have experienced the blank feeling that sometimes arises after an expression has been rejected by the compiler. Proponents of strong typing secretly smirk at this discomfort. They contend that it is good

for the programmer to think occasionally about the program under development. And, in any case, it is far better to investigate the problem when detected by the compiler than waiting for it to strike during some execution of the program. But 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 in the programming language ML.

The programming language ML [MTH90] is a strongly-typed functional language with polymorphism. Its interesting type system with the property that the types of all variables can be deduced by their use [Mil78] has made it popular and the model for the type systems of 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.

We are inspired by the paper [SSK+86] 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 are so complex it is useful to have a system that explains the steps taken to reach a conclusion. We doubt that this will replace lawyers soon, but 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. This same question is useful in the case of type errors. For an expression that is not typable the type analysis has arrived at two inconsistent types and the programmer may want to know why.

The place where the inconsistency was found may not play any role in where the mistake was made. It is the nature of Milner's type reconstruction algorithm that widely separated parts of the program affect the types that expressions have. Any part of the program might be the mistake in the view of programmer, but where the inconsistency is found depends on the arbitrary order of the traversal of the syntax tree.

Some interesting information may be obtained by examining all the contextual evidence. This is the approach of [JW86]. The usage that is in the minority is a likely candidate for the mistake. It is not clear how often the minority can be identified. Many errors are of the form where there is one correct usage and one incorrect usage. The approach we take here is entirely different, but the two approaches might be used together in a heuristic system that both explains deductions and suggests where the error may be found.

In his 1986 POPL paper Wand [Wan86] 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 process. His approach called for the collection of application subexpressions during the unification step of the type analysis. When a type error is detected a linearized list of application sites is issued.

We improve on Wand's work by doing more and deeper 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

⟨*expr*⟩ ::= ⟨*variable*⟩
| ⟨*integer*⟩
| ⟨*boolean value*⟩
| (⟨*expr*⟩ , ⟨*expr*⟩)
| ⟨*expr*⟩ ⟨*expr*⟩
| `fn` ⟨*var*⟩ `=>` ⟨*expr*⟩
| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
| `let` ⟨*var*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩ `end`

Figure 1: The syntax of a simple functional language

⟨*type*⟩ ::= ⟨*type variable*⟩
| `int`
| `bool`
| ⟨*type*⟩ * ⟨*type*⟩
| ⟨*type*⟩ `->` ⟨*type*⟩

Figure 2: The syntax of types

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 Background

In this section we explain the general concepts behind type reconstruction as can be found in [Rea89, Car87]. The next section describes the modifications to this process to allow for explanations.

2.1 Language

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. Consistent with SML of NJ we replace subexpressions of the language with the character # when we do not care 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.

2.2 Type reconstruction and polymorphism

Type reconstruction is the process of deducing the type of an expression from context. In a type checker for a language like Pascal, variables 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. Expression variables 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. If the domain of a function is polymorphic, then the function can be applied to arguments of more than one type. The simplest example of a polymorphic function is the identity function `(fn x=>x)`.

We can consider the collections of types that an expression may assume as being defined by a formal proof system. To understand the proof system we require two definitions: type assignment and typing judgment.

One, a type assignment which we shall denote by the letter A, maintains the already established or given list of bindings of variables and their types. Two, a typing judgment is a triple of the form

$$A \vdash e : \tau$$

where A is a type assignment, e is an expression, and τ is a type. In this case, we say "expression e has type τ under the assignment A." We can use these typing judgments to define when an expression has a type. In this way the set of types for an expression is defined. It is possible to formulate the typing rules so that each construct of the language has one rule. We give two of the well-known rules.

The type of an variable x is determined by the context. For this reason a type assignment is necessary. If x has been bound by some `fn` construct, then x has some value in the assignment A. The value in A is the type of x. This is the content of the first rule in the deductive system.

$$\frac{}{A \vdash x : A(x)} \qquad \text{if } x \in \mathrm{Dom}(A)$$

By this rule we mean: if x is one of the identifiers assigned a type by A, then the type of x is $A(x)$, the type associated to x by A; otherwise the rule does not apply.

The following type rule is for the application of the expression e_1 to e_2.

$$\frac{A \vdash e_1 : \tau_2 \texttt{ -> } \tau, \qquad A \vdash e_2 : \tau_2}{A \vdash e_1\, e_2 : \tau}$$

The type of the first expression e_1 must be a function. The type of the second expression must be equal to the domain of the type of e_1.

In [Mil78] 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 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.

A crucial insight in designing a theorem prover lies in recognizing that unification can be used to force types to be 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. If the domain is `'a list * bool` and the type of the argument is `int list * 'b`, then mapping `'a` to `int` and `'b` to `bool` makes the two identical. Such a mapping is called a substitution. The algorithm to find a substitution (if one exists) that makes two terms 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.

2.3 Unification

Unification is the process of finding the most general substitution which when applied to two types will result in a common instance. For example, if the type `'a -> int` is unified with `bool->'b` the most general substitution would map `'a` to `bool` and `'b` to `int`; the common instance would be `bool->int`. We call the function that evaluates a type in a substitution `Value`.

The key property of the unification algorithm is that upon a successful termination a substitution representing the *most* general unifier of the two types is returned. Furthermore, if this most general unifier exists the algorithm will successfully terminate. These properties are embodied in the statements [Rob65]

1. If S' =`Unify`(τ_1,τ_2,S) succeeds then `Value` S' τ_1 = `Value` S' τ_2.

2. If there is a substitution R such that `Value` R τ_1 = `Value` R τ_2, then S' =`Unify`(τ_1,τ_2,S) succeeds and there is a substitution T such that $R = TS'$.

There are many ways to represent a substitution in ML. 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 `Unify` function to keep extra information we must have a substitution represented as a list. Usually for efficiency of implementation, earlier bindings in the list are not modified when later bindings are added to the substitution. It is the responsibility of the function `Value` to remember to check earlier bindings when computing the value of a type variable in a substitution. `Value` does not just look up the variable in the list of pairs and return the associated type. This representation of substitutions plays an important role. Each atomic binding inserted in the list can be explained by one particular unification. If we had later bindings apply `Value` to their type, then possibly several unifications would be responsible for this value of the type variable.

```
datatype Expr =
    Var of  Variable * Type * ((TypeVariable * Type) list)
  | Num of  int
  | Bool of bool
  | Pair of Expr     * Expr * Type * Type
  | Fun of  Variable * Expr * Type * Type
  | App of  Expr     * Expr * Type * Type
  | Cond of Expr     * Expr * Expr * Type * Type * Type
  | Let of  Variable * Expr * Expr * Type * Type
  ;
```

Figure 3: A data structure for an augmented expression tree.

One would have to keep a list of reasons for each binding, like in Wand [Wan86], and it would be impossible to explain the instantiation of a type step by step.

3 Approach

In this section we describe what modifications we have made to the normal type analysis algorithm to keep extra information. We have written two new functions that explain the deductions made during the type analysis. We describe the nature of these functions and their output.

3.1 Structure

Extra information gathered during type analysis divides into two distinct kinds: one, information we need to add to the syntax tree of the expression being analyzed; two, information saved during unification.

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. In figure 1, a grammar for the syntax of expressions for the simple language which is the basis of our system is shown. An ML data structure representing this grammar is shown in figure 3. Each case of the data structure provides for its appropriate annotations. 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 function definition structure 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. `App` keeps 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.

A variable must also keep two pieces of information. For the same reason as the case of function definition, a variable keeps the type variable originally assigned to it in the type assignment. Because of the `let` statement, the variable construct must also keep a list of type variable renamings. These are the renamings of all the generic variables in the type of the expression variable construct. We explain more about the `let` statement and generic variables later.

The second part of the explanation structure is the concrete representation of the substitution mapping. The format of this representation 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 `Unify` it passes the current location in the syntax tree which `Unify` stores in any atomic binding resulting from this invocation.

3.2 Why and How

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 three (for the simple language we are considering) calls to `Unify`.

Our modified `TypeOf` decorates the parse tree during the recursive traversal of the syntax tree. All calls to `Unify` 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 program `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.

Consider the syntax tree in figure 4. 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 at the node `+(#,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 SML:

```
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)
```

Now it is clear that plus has the right domain, and that the argument 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 question that the function `Why` answers.

One after another `Why` explains the type rules for the constructs 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 is shown in figure 6.) Now `Why` turns to the explanation of the variables `b` and `a`. Structurally all variables 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 example in figure 4, `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`.

3.3 Explanation space

Explanations offered by the system fall into two categories. Structural explanations given by `Why` follow the structure of the parse tree or proof tree in a one-to-one fashion. When these explanations require `How` to explain the flow of information around the program, the 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 the language considered here. One node makes two calls to `Unify`. The function `How` explains the circumstances which required the unification. The further explanation of these circumstances is now a structural question.

Because it is possible to leap back into the tree at some previously explained point, the explanation space is infinite. Consider the previous example with the syntax tree shown in figure 4. In the course of the explanation we want to know why `b` has type `bool` in the

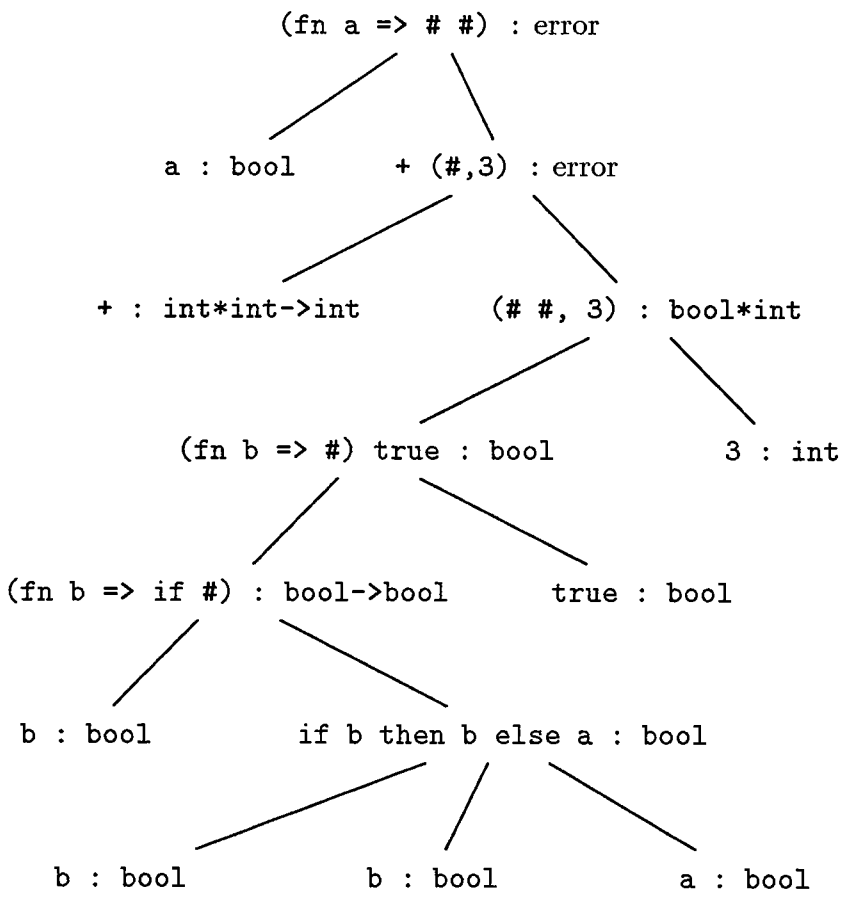

Figure 4: Syntax tree for `(fn a=>+((fn b=>if b then b else a) true,3))`

conditional part of the `if` statement. The information in the substitution leads up to the `if` statement and the typing rule which requires the conditional to have type `bool`. If we want to know why the conditional has its type, then we go back down to the variable `b` again.

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 `Unify` is called. The `TypeOf` function always annotates the node first and then calls `Unify`. The recursive annotation of the subtrees has already been completed.

3.4 Interface

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` for an expression with a type error is shown in figures 6 and 7. 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

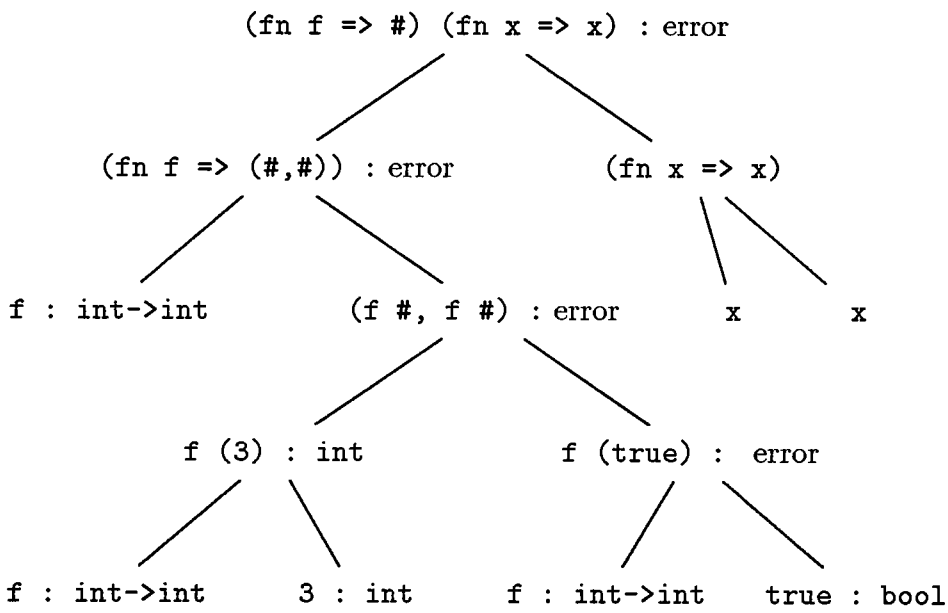

Figure 5: Syntax tree for `(fn f=> (f 3, f true))(fn x=>x)`

zero, one, or more other deductions. The programmer may find one of deductions dubious, as in the prior example with the function `plus`. The type of `+` is not suspicious. So, the programmer then chooses to investigate the type of the actual argument. With practice the programmer could quickly hone in on the part of the program that is contrary to the programmer's understanding of the program.

But it seems likely that a novice user could get lost following the cycles in the explanation space without knowing it. Therefore any good interface should keep a close watch on the path the programmer chooses and might suggest that the programmer had seen some one explanation enough times.

3.5 Examples

Figure 4 is the syntax tree for the following expression:

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

This expression cannot be typed because `b` is used as both a `bool` and an `int`. The output of `Why` and `How` is shown in the figure 6.

We illustrate the process of explanation with another example. Figure 5 is the syntax tree for the following expression:

```
(fn f=> (f 3, f true))(fn x=>x)
```

This expression cannot be typed. The explanation space for this figure is given in figure 7.

The explanation starts at the location of the error, the node `f (true)` in figure 5. This is an error because the domain of the function `int` does not unify with the type of the

```
The domain of the function +
is not unifiable with the type of the argument ((# true),3).
The function has type ((int*int)->int).
The argument has type (bool*int).
**Why does the function have this type?
  The type ((int*int)->int) of the variable + is determined by the environment.
**Why does the argument ((# true),3) have the type (bool*int)?
  The type of the pair ((# true),3)
  is derived from the type of its two elements.
  **Why does the first element ((fn b => #) true) have type bool?
    The type of an application is the range of a function
    The function (fn b => if b then b else a) has type (bool->bool).
    **Why does the function have that type?
      The type of a function definition is determined by its
      domain, the formal parameter, and its range, the body.
      **Why does the formal parameter b have type bool?
        The following binding further constrained the type 'b:
        Type variable 'b is bound to type bool
        **How did this binding arise?
          This binding is caused by the unification of
          'b with bool because the expression b is the test of a conditional.
          **Why does b have the type 'b
            The type bool of the variable b is determined by the environment.
            **How was this type for the variable obtained?
              The following binding further constrained the type 'b:
              [ and so on ... ]
      **Why does the body of the definition
        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?
          [ and so on ... ]
        **Why does the "else" branch a have type bool?
          The type bool of the variable a is determined by the environment.
          **How was this type for the variable obtained?
            The following binding further constrained the type 'a:
            Type variable 'a is bound to type bool
            **How did this binding arise?
              This binding is caused by the unification of 'b with 'a
              These types are unified because they are the branches of an if.
              [ and so on ... ]
 **Why does the second element 3 have type int?
   All numbers have type int.
```

Figure 6: Explanation space of `(fn a=> +((fn b=>if b then b else a) true, 3))`

```
The domain of the function f
is not unifiable with the type of the argument true.
The function has type (int->'b).
The argument has type bool.
**Why does the function have this type?
  The type (int->'b) of the variable f is determined by the environment.
  **How was this type for the variable obtained?
    The following binding further constrained the type 'a:
    Type variable 'a is bound to type (int->'b)
    **How did this binding arise?
      This binding is caused by the unification of
      'a with (int->'b).
      This unification was necessary to check the domain
      of a function against its argument.
      The function application is (f 3)
      **Why does f have the type 'a
        The type (int->'b) of the variable f is determined by the environment.
        **How was this type for the variable obtained?
          The following binding further constrained the type 'a:
          [ and so on ... ]
      **Why does 3 have the type int
        All numbers have type int.
**Why does the argument true have the type bool?
  All boolean values have type bool.
```

Figure 7: Explanation space for `(fn f=> (f 3, f true))(fn x=>x)`

argument `bool` as stated in the explanation in figure 7. The type of the function though was determined in the other part of the pair, because `f` is a formal parameter bound to the type variable `'a` originally. The function `How` is called to explain this. Again in figure 7 we find the explanation: the type variable `'a` was bound to type `int->'b` in a unification made necessary by the function application `f (3)`. Note this leads into a tight cycle in the explanation space, because the type of the application `f (3)` is determined by the type of `f`. This leads back again to the question why `f` has type `int->'b`.

3.6 Milner `let` construct

The `let` construct is of vital importance to the type system in the programming language ML. To illustrate this, consider the following function definition:

```
fn f => (f 3, f true);
```

Even if we give `f` the polymorphic type `'a->'b`, applying `f` to 3 and to `true` causes a type error. The first application identifies `'a` with the type `int`. The second unifies `'a`, which is now equivalent to `int`, with `bool`. The difficulty here is that the type system does not permit the type variables in the type of `f` to be instantiated separately with each invocation. The special treatment of `let` allows this instantiation.

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

The variable bindings here differ from those of function abstraction in that all instances are not required to have the same type. This is achieved by introducing generic type variables that may be instantiated over and over again.

The `let` construct complicates the explanation of how a variable's type gets instantiated. Without `let` all expression variables 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 in the substitution. In the presense of the `let` statement, expression variables are assigned types, not just type variables, which may contain generic variables. To explain the type of a `let`-bound variable 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 expression variable. `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
```

This expression is well-typed. We wish 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 well-typed expression. So we ask: why does it have type `int*bool`? The output of `Why` and `How` is given in the figure 8. 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`.

```
The type of a let statement is the type of the body.
The body is ((f 3),(f true)).
**Why does the body have type (int*bool)?
  The type of the pair ((f 3),(f true))
  is derived from the type of its two elements.
  **Why does the first element (f 3) have type int?
    The type of an application is the range of a function
    The function f has type (int->int).
    **Why does the function have that type?
      The type ('a->'a) of the variable f is determined by the environment.
      **How was this type for the variable obtained?
        This type ('a->'a) has generic type variables present.
        So, it was necessary to replace these with fresh type variables.
        The following list of pairs exhibits the correspondence.
        'a maps to 'b
        Thus, the type we are now contraining is ('b->'b).
        The following 2 bindings further constrained the type ('b->'b):
        Type variable 'b is bound to type 'c
        **How did this binding arise?
          This binding is caused by the unification of ('b->'b) with (int->'c).
          This unification was necessary to check the domain
          of a function against its argument.
          The function application is (f 3)
          **Why does f have the type ('b->'b)
            The type ('a->'a) of the variable f is determined by the environment.
            [ and so on ... ]
          **Why does 3 have the type int
            All numbers have type int.
        Type variable 'c is bound to type int
        **How did this binding arise?
          This binding is caused by the unification of ('b->'b) with (int->'c).
          [ and so on ... ]
 **Why does the second element (f true) have type bool?
    The type of an application is the range of a function
    The function f has type (bool->bool).
    **Why does the function have that type?
      The type ('a->'a) of the variable f is determined by the environment.
      **How was this type for the variable obtained?
        This type ('a->'a) has generic type variables present.
        So, it was necessary to replace these with fresh type variables.
        The following list of pairs exhibits the correspondence.
        'a maps to 'd
        [ similarly ... ]
```

Figure 8: Explanation space of `let f=(fn x=>x) in (f 3,f true) end;`

4 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. We do not claim to find the source of type errors. We are not even sure just exactly what a program error is. Certainly, there are programming errors in which the type system finds some unexpected way to deduce a type for an expression. 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.

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 [Wan86]. 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.

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. With more experience, navigation through the explanation space could be improved.

References

[Car87] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2):147–172, April 1987.

[JW86] Gregory F. Johnson and Janet A. Walz. 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*, pages 44–57, January 1986.

[Mil78] R. Milner. A theory of type polymorphism. *Journal of Computing Systems Science*, 17:348–375, 1978.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[SSK+86] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaaczek, P. Hammond, and H. T. Cory. The British Nationality Act. *Communications of the ACM*, 29(5):370–386, May 1986.

[Wan86] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, January 1986.