Improved Type Error Reporting

Jun Yang, Greg Michaelson, Phil Trinder and J. B. Wells
http://www.cee.hw.ac.uk/~{ceejy1,trinder,greg,jbw}

Department of Computing and Electrical Engineering
Heriot-Watt University, EDINBURGH, EH14 4AS, UK

Abstract. The error reports produced by compilers for languages with polymorphic type inference are often uninformative. Several attempts have been made to produce improved error reports. We define a manifesto for good error reports and use it to evaluate both the traditional algorithms, and several improved algorithms, including two of our own devising.

1 Introduction

Debugging type errors in programs written in polymorphic languages with type inference is hard [13, 7]. One reason is that much type information is implicit — inferred by the compiler. An error occurs in type inference when two *uses* of a program variable are found to be in conflict, where each use occurs in some program text that we call a *site*. These concepts are best illustrated with an example. In Figure 1, inference proceeds left-to-right, initially assigning a type variable `'a` to the function parameter `x`. From the first use of `x`, sited at `x 1`, the algorithm infers that `x` is a function from integer to some unknown type, i.e., `x` is of type `int` → `'b` where `'b` is a freshly chosen type variable standing for some as-yet-undetermined type. From the second use of `x`, sited at `x true`, the algorithm infers that `x` is a function from booleans to some unknown type, i.e., `x` is of type `bool` → `'c`. The conflict is detected when the unification of the two types for `x` (`int` → `'b` and `bool` → `'c`) fails.

```
fn x => (x 1, x true)
    stdIn:6.9-6.23 Error: operator and operand don't agree [literal]
      operator domain: int
      operand:         bool
      in expression:
        x true
```

Fig. 1. Example of inference failing

Unfortunately the errors reported by compilers are often uninformative. Version 110 of the SML/NJ compiler reports the error site `x true`, but does not mention `x 1`, which is one of the sources of the type conflict, and may be the real error. One reason for this is that compilers use type inference algorithms that are designed primarily to analyze programs efficiently, and have comprehensible error reporting as a subsidiary goal.

This paper focuses on improved error reporting for languages with polymorphism and type inference in the style of the Hindley/Milner type system. We define a manifesto for good type error reporting (section 2). We use the manifesto to identify problems with the error reporting of various inference algorithms (section 3). We briefly survey improved error reporting systems (section 4) and describe two new type inference algorithms with improved error reporting (section 5). We conclude with a comparative evaluation of type error reporting systems (section 6).

2 A Manifesto for Good Type Error Reporting

We think that good error reports should have the following properties:

1. Correct. This entails both *correct detection*: errors are reported exactly when the program is not legal, and *correct reporting*: all the reported sites contribute to the type conflict.
2. Precise. Each conflicting site should be located in the smallest useful amount of source text. Moreover, there should be a simple relationship between the conflicting type and the site from which it was inferred.
3. Succinct. Error report should maximise useful information and minimise non-useful information. Long and verbose explanations are tedious to read. Short and terse explanation are hard to understand.
4. Amechanical. An error report should not reproduce large amounts of counter-intuitive mechanical inference. This includes reporting with artificially-introduced type variables [15].
5. Source-based. The user need not know anything of the compiler internals to understand the error message. For example error reports should not use “core” syntax generated by the compiler from the original source syntax. The inference algorithms can infer and report on source syntax. However the implementation may be not source-based.
6. Unbiased. There should no inherent left-to-right (or similar) bias in the choice of where to report an error when multiple sites contribute to the error.
7. Comprehensive. The algorithm should be able to report all sites that contribute to the reported type conflict. The user can reason about the error from the reported sites and does not need to look at other parts of the program to locate the error.

Some error reporters aim to report multiple errors instead of stopping at the first error, while avoiding generating a cascade of bogus error messages. This is not included in the manifesto as it is not always desirable, for example many error reporters in the education sector stop at the first error.

3 Inference Algorithm Error Reporting

Type inference algorithms, like the $\mathcal{M}$ and $\mathcal{W}$ algorithms, are designed primarily to analyze programs efficiently, and have comprehensible error reporting as a subsidiary goal. Throughout the paper we use the $\mathcal{W}$ algorithm as implemented in Standard ML of New Jersey, Version 110.0.6, October 31, 1999 [9], and our own implementation of the $\mathcal{M}$ algorithm based on [6].

Compared to our manifesto, inference algorithm error reports are both correct (1) and succinct (3), but have the following problems.

- 2. Not precise. The $\mathcal{W}$ algorithm does not identify the precise sites in the program that conflict. In particular, it fails (i.e., notices an error) only at a function application, and an erroneous expression is often successfully type-checked long before its consequence collides at an application. For example, if typing succeeds for both function and argument expressions, but fails at the (outermost) function application, the entire expression is reported as the error site. Figure 2 illustrates this using New Jersey SML where $\mathcal{W}$ algorithm does not report the "obvious" (to a human) problem that the outermost function expects an integer, but its argument is boolean.
  The $\mathcal{M}$ algorithm is precise, in particular it always stops earlier than the $\mathcal{W}$ algorithm when there is a type error [6]. The $\mathcal{M}$ algorithm does not stop at application, it stops at constant, variable, λ abstraction instead. Figure 3 shows how it identifies the site `true` more precisely.

```
(fn x ⇒ x+1) ((fn y⇒ if y then true else false) false)

   stdIn:21.1-21.60 Error: operator and operand don't agree [literal]
     operator domain: int
     operand:         bool
     in expression:
       (fn x => x + 1)
           ((fn y =>
                  (case <exp>
                    of <rule>
                     | <rule>)) false)
```

Fig. 2. $\mathcal{W}$ Algorithm: Imprecise Error Location

- 4. Mechanical. The algorithms introduce a type variable for many constructs before instantiating the type and error reports use internal type variables that have no obvious relation to the program text. For example, Figure 4 shows the function and corresponding error report produced by Standard ML of New Jersey, Version 110.0.6 $\mathcal{W}$ algorithm. The type variable `'Z` appears nowhere in the program text, having been introduced during inference.

```
(fn x ⇒ x+1) ((fn y⇒ if y then true else false) false)

    -------------------- Error --------------------
    Error in expression:
        true

    Type inconsistent of requirement at the expression.
    required type: int
    actual type  : bool
```

Fig. 3. $\mathcal{M}$ Algorithm: Application Error Reporting

```
fun f2 x (h::t) = h :: f2 t h

    stdIn:35.1-35.27 Error: right-hand-side of clause doesn't agree with
    function result type [circularity]
      expression:  'Z list -> 'Z list
      result type:  'Z -> 'Z list
      in declaration:
        f = (fn arg => (fn <pat> => <exp>))
```

Fig. 4. Inference Algorithm: Counter-intuitive Reporting

- 5. Not Source-based. Errors are reported using text generated from the abstract syntax tree. For example in Figure 2, SML/NJ 110 reports the error using a pretty-printed version of its internal abstract syntax tree, which is different from the source syntax.
- 6. Biased. The algorithms have a left-to-right bias: type checking proceeds exhaustively from top to bottom and left to right within the program text. As shown in Figure 1, the $\mathcal{W}$ algorithm presumes that the first use of `x` is the correct one.
- 7. Not Comprehensive. The algorithms typically stop at the first site in the program where a conflict is detected when there may be other undetected conflicts in the program. For example, in Figure 5, Edinburgh Standard ML (core language) reports `f true` as the error, and does not find the second conflict with `f false` and the type error in the result tuple, i.e. if the type of the application of the function `f` is a specific type e.g `'Z`, then the type of the right-hand side should be the specific type `'Z`, but in the example, the type of right-hand side is a tuple type `'Z * 'Z * 'Z`.

```
fun f x = (f 3, f true, f false)
    Type clash  in:  (f true)

    Looking  for a:  int

    I have found a:  bool
```

Fig. 5. Inference Algorithm: Non-Comprehensive Reporting

4 Improved Error Reporting

The problems with inference algorithm error reporting have motivated work aimed at providing a better understanding of type error, often at the expense of additional computation. One approach is to provide better explanation of how the inference leads to a type error. The systems that use this approach are classified as *error explanation systems*, e.g., [1, 4, 11].

Another approach is to provide a better error report, without necessarily explaining the inference that lead to the error. Systems using this approach are classified as *error reporting systems*, e.g., [13, 5, 2, 8, 3]. A third approach is to provide a mechanism for the programmer to probe the type of subexpressions [2].

4.1 Error Explanation Systems

There have been several error explanation systems. For example, the system of Duggan and Bend uses a modified unification algorithm to record the reasons which led to a program variable having a particular type [4]. The system of Beaven and Stansifer [1] explains how the inference reaches the two conflicting types. Soosaipillai's system [11] explains the type inference by menu traversal.

Compared to our manifesto, type error explanation systems can be made comprehensive, precise, and correct. However they fail on several key points.

- 3. Not Succinct. Sometimes the textual explanation has so much information, it rapidly becomes tedious. Experts usually find this explanation too detailed to be of real help, though they find valuable information about the different positions in the programs that contribute to the type given by the tool [10].
- 4. Mechanical. The explanations make substantial use of internal type variables as bridges between instances, since they are the ones that get refined. In contrast a programmer is not concerned with these type variables. To understand the explanations, it is necessary to remember from which program variable the type variable is inferred and where it is refined. If there are more than a few type variables, it becomes impossible to remember what entity a type variable represents.
- 5. Not Source-based. Explanations use text generated from the abstract syntax tree, rather than the original program text.
- 6. Biased. They are based on the $\mathcal{W}$ algorithm, which has a left-to-right bias in inferring types and discovering errors.

4.2 Error Reporting Systems

Error reports can be improved by locating all error sites, by locating them more precisely within the program text, or by providing an explanation of the conflict. There have been several approaches, including the following.

- Wand's system [13] records the sites that contribute to each type deduction when type errors are detected, and uses this information to explain why the errors happen. The algorithm records substitutions together with function applications which are the causes of the substitution. A type error consists of two types that cannot be unified, and both are derived by some substitutions: the algorithm reports each application that caused one of these substitutions as a possible cause of the error. Where the inconsistency is found depends on the arbitrary order of traversal of the syntax tree during type analysis. Consequently, the number of candidate error sites is also decided by the programming style. The system lists the possible error sites; some of them are not the direct source of the type inconsistency. It is not clear if there is any relationship between the candidate error sites: the user needs to check the types of the proposed error sites against their own intentions.
- Johnson and Walz [5] give a maximum-flow approach to decide which usage is the most likely error source. The usage that is in the minority is a candidate for the mistake. However for many errors there is one correct usage and one incorrect usage and it is not clear how often the minority can be isolated, and sometimes the minority usage may be the correct type.
- Turner [12] considered improved error handling for unbound identifier error reports, source-based error reports, and mutually recursive declaration error reports. Similar to Johnson's method, his method chooses the minority use as the candidate for the error source.
- Bernstein and Stark give a method of debugging type errors in a so-called *open system* [2]. The user replaces a suspect expression within the program with a free variable. The system infers the type of the free variable, which the user can compare with that expected. The user can repeatedly probe the program in this way until the error is uncovered. In contrast to the other techniques, this requires user interaction to locate the error. In particular, the user must guess the probable location of the type error.
- In ongoing work, McAdam uses a graph of types to report errors [8]. The graph contains information about types and the sites that contribute to the types, and an explanation is generated by traversing the graph. However, to generate a concise explanation for some type errors, a suitable graph traversal must be selected. It is not clear what the best way to traverse the graph.
- The approach of Dinesh and Tip [3] requires no changes to the type inference algorithm or the type system. The basic idea is to apply dependence tracking to a rewriting-based implementation of an ML type inferencer. A program slice can be computed for each reported type inference error. A program slice consists of the parts of a program that(potentially) affect the values

computed at some point of interest. However it is unclear how accurate such slices will be in practice.

4.3 Error Reporting Systems vs. Manifesto

Wand's system is selected as a typical model based on the $\mathcal{W}$ algorithm, using a modified unification algorithm to generate the explanation. Later work such as Duggan and Bent [4] and McAdam [7] is similar in using modified unification algorithm to report or explain error. The open system is not included in our evaluation because it is user-driven type debugging, rather than error reporting. Also we do not include the work of Dinesh and Tip [3] because it is different from type inference.

```
fn n => n* (n=1)

------------------- Error sites ------------------
There are 5 possible sites,
the first one is where inconsistence happened
 BOOL ===> INT By  *(( n, =(( n, 1))))
 INT * INT ===> INT * INT By  =(( n, 1))
 INT * INT ===> INT * INT By  *(( n, =(( n, 1))))
 BOOL ===> BOOL By  =(( n, 1))
 INT ===> INT By  =(( n, 1))
```

Fig. 6. Example of Wand's Error Reporting

Wand's system We compare Wand's approach to our manifesto using Figure 6.

1. Correct. All the sites contribute to the type conflict between `int` and `bool`.
2. Precise. It reports only applications. It is not precise as the $\mathcal{M}$ algorithm, e.g. it can not report smallest sites such as the operator `*` and `=`.
3. Not concise. Its report can be a long list of error sites. In the simple example of Figure 6, it reports all the application subexpressions, and some of them are redundant.
4. Fairly mechanical. A site may contribute to substitutions to several types, there are redundant reports for a single site, for example the site `n=1` is reported three times in Figure 6. But it reports the reasons that contribute to the type error.
5. Not source-based. The implementation is not source-based.
6. Biased. Wand's approach is to use a modified unification algorithm to accumulate the reason of the type conflict. If the inference algorithm has left-to-right bias, the reports have left-to-right bias as well.

7. Comprehensive. It reports all the possible sites that contribute to the type error.

Johnson and Walz's method If a type inconsistency arises, a maximum flow technique is applied to the set of type equations to determine the most likely source of the error. The approach is:

1. Correct. It reports a subset of the possible error sources.
2. Precise. It can compare the uses of any operator, and report the operator such as `*` and `=` as the error site.
3. Succinct. It chooses the most likely error site.
4. Fairly mechanical. It selects the minority usage as the error site, however the majority uses may be the cause of the error. If there is just one correct usage and one incorrect usage it is not possible to decide which is in error. It is counter-intuitive in the sense that it does not report conflicting sites.
5. Can be source-based.
6. Unbiased. It does not assume the first use is the correct use.
7. Comprehensive. It compare all the uses and can find all the conflict sites.

The error explanations can not be illustrated as we have not implemented Johnson and Walz's method.

Turner's method If a type inconsistency arises, a count of the different uses is used to determine the most likely source of the error. The approach is:

1. Correct. It reports a subset of the probable error sources.
2. Precise. It compare the types in patterns and applications. But it is not precise as the $\mathcal{M}$ algorithm and Johnson and Walz's.
3. Succinct. It chooses the most likely error site.
4. Amechanical. It selects the minority usage as the error site.
5. Source-based. It records the expression the type checker is typing.
6. Unbiased. It does not assume the first use is the correct use.
7. Comprehensive. It can find all the conflict sites.

5 Improved Type Error Reporting

We now present two improved type error reporting algorithms. Both have been described in [14], and are only briefly outlined here.

5.1 Unification of Assumption Environments ($\mathcal{U}_{\text{AE}}$)

The first new algorithm is based on the unification of assumption environments. The key idea is to independently type each subexpression in an application and return an assumption environment for each subexpression which gives the local type constraints for every variable in the subexpression. The consistency of resulting assumption environments is checked at the root of subexpressions. Left-to-right bias is removed by unifying the assumption environments at the root of the subexpressions, where we compare the uses of each program variable in different subexpression to see if they are all consistent: in this way every subexpression is treated equally.

The $\mathcal{U}$AE algorithm takes a type environment, an assumption environment and an expression as its arguments. The type environment is the ordinary type environment as in the $\mathcal{W}$ algorithm. The assumption environment contains the type constraints for the program variables in previous typed subexpressions. At the start of type checking, the type environment and assumption environment are empty. The assumption environment is used to check the type consistency of those program variables in other subexpressions and to identify precisely the finer grain error sites.

```
fn x ⇒ (x 1, x true)

    -------------------- Error --------------------
    Type conflicts in expressions:
     x(1)
     x(true)

    the same program variables have type conflicts in different sites
        from the first expression
    x: int -> 'a

        but from the second expression
    x: bool -> 'b
```

Fig. 7. Example $\mathcal{U}$AE Error Report

The method is best illustrated by an example, and Figure 7 shows the output of our implementation for a simple function. On the same function, Figure 1 shows how the $\mathcal{W}$ algorithm proceeds from left to right, inferring a function type for `x` from its first use, and then reporting the conflict at the second use of `x`, i.e., it reports `x true` as the (only) error site in Figure 1. In contrast, the $\mathcal{U}$AE algorithm explains that `x`, which must have a monomorphic type because it is lambda-bound, is used inconsistently in different subexpressions.

Figure 8 shows how the $\mathcal{U}$AE algorithm identifies the conflict between the left-hand side of a function definition, and its uses within the body of the function. Figure 4 shows that the $\mathcal{W}$ algorithm only reports the entire function definition as an error.

$\mathcal{U}$AE vs. Manifesto

```
    fun f2 x (h::t) = h:: f2 t h
-------------------- Error --------------------
Type conflicts in expressions:
    f2(x)(h :: t)
    f2(t)(h)


 elements in a pattern have different types
   from the first expression        'c list
   from the second expression       'c
```

Fig. 8. Second Example $\mathcal{U}$AE Error Report

1. Correct. A reported error site or subexpression, has some relationship with other sites, i.e., its type conflicts with that of another site. The $\mathcal{U}$AE algorithm reports pairs of sites with conflicting types, e.g., it reports `x 1` and `x true` as a pair of conflicting sites.
2. Precise. It is more precise than the $\mathcal{W}$ algorithm. For example in Figure 4 the $\mathcal{W}$ algorithm reports the whole expression as the error site, in Figure 8 the $\mathcal{U}$AE algorithm reports the left-hand side and right-hand side type conflict of the recursive function `f2`. It locates the smallest pairs of type conflicting sites. And the reported conflicting types can be inferred from the reported sites directly.
3. Succinct. It reports by the type conflicting uses of program variables in the pairs, rather than how it concludes that there are type conflicts.
4. Amechanical. It reports the conflicting sites. Its type information is inferred from the reported sites. To understand why the reported sites are error sites, the user needs to understand the reported type information. The type information from the $\mathcal{U}$AE algorithm is from local typing of the reported sites, it is easier to understand than the type information from a long chain of inferences, such as that from the $\mathcal{W}$ algorithm.
5. Can be source-based. The implementation is partially source-based.
6. Unbiased. Every subexpression of an application is treated independently.
7. Comprehensive. The $\mathcal{U}$AE algorithm identifies the type conflicting uses of the same program variable in a pair of subexpressions.

5.2 Incremental Error Inference ($\mathcal{IEI}$)

The second new typing algorithm, incremental error inference, locates conflicts in application expressions when the $\mathcal{U}$AE algorithm cannot. This additional functionality is bought by making the algorithm more complex.

As we have seen in Section 3, and Figure 2 in particular, because the $\mathcal{W}$ algorithm fails (discovers a typing error) only at a function application, it often identifies large amounts of program text as the error site. The $\mathcal{M}$ algorithm always stops earlier than the $\mathcal{W}$ algorithm [6] and can be used to report a finer-grain error site as shown in Figure 3. The $\mathcal{M}$ algorithm brings a type constraint (or an expected type) that each subexpression must satisfy. For example, in the case of application, if the required type for application expression e_1e_2 is `real` from the context, then the required type for e_1 is $\beta \rightarrow real$, and the required type for e_2 is β.

Because it carries a context, the $\mathcal{M}$ algorithm reports a finer grained error site than the $\mathcal{W}$ algorithm, stopping at a constant, a variable, or a lambda expression. For example, Figure 3 shows the error message produced by the $\mathcal{M}$ algorithm for the same program as in Figure 2. The $\mathcal{M}$ algorithm stops at `true`, identifying a smaller error site, but it does not reveal the other sites which are in conflict.

The key idea behind $\mathcal{IEI}$ is to combine the $\mathcal{M}$ algorithm and the $\mathcal{U}$AE algorithm. When the $\mathcal{U}$AE algorithm fails (discovers a typing error) at an application, and $\mathcal{U}$AE finds no directly conflicting uses for each program variable in all the subexpression, $\mathcal{IEI}$ switches to the $\mathcal{M}$ algorithm, which always stops at a site which is smaller than the whole expression. In particular, If the $\mathcal{M}$ algorithm stops at the argument of a application, there is a conflict between the function and the argument of the application. For example when the $\mathcal{M}$ algorithm stops at `true` in Figure 3, it means that the function `(fn x ⇒ x + 1)` has a type conflict with its argument `((fn y⇒ if y then true else false) false)`. But the $\mathcal{M}$ algorithm stops too early, it does not report the `false` is another type error site.

When type check function application, the $\mathcal{IEI}$ algorithm locates the sources of the conflict by assuming that the argument subexpression where the $\mathcal{M}$ algorithm found a type inconsistency is type correct, and then type checking the function subexpression under that assumption. Hence we can find another conflicting site in the function application, and the reason for the conflict. This also removes the left-to-right bias of inference algorithms.

For example Figure 9 shows the error reported by $\mathcal{IEI}$ for the same program as in Figures 2 and 3.[1] The incremental error inference algorithm $\mathcal{IEI}$ gives error explanation messages by finding a pair of directly type conflicting sites, and showing the reasons for their conflicts. $\mathcal{IEI}$ reports the function `(fn x ⇒ x + 1)` and its argument `((fn y ⇒ if y then true else false ) false )` have type conflicts. Moreover, $\mathcal{IEI}$ identifies the reason for the conflicts: the required type of `x` is a `int` from the operator `+` and an operand `1` of `+`, but `x` as the argument in `(fn x ⇒ x + 1)` is supplied with a `bool` value. Also the subexpressions in `fn y ⇒ if y then true else false` at the sites of `true` and `false` is required as `int` type by function `(fn x ⇒ x + 1)` but they are `bool` type.

To avoid artificially-introduced type variables in error reporting, the $\mathcal{IEI}$ infers as much concrete type information such as base types as possible.

[1] As there are no common program variables that are in type conflicts in the subexpressions, the $\mathcal{U}$AE algorithm behaves in the same way as the $\mathcal{W}$ algorithm and reports the entire expression as the error site

```
(fn x ⇒ x+1) ((fn y⇒ if y then true else false ) false )

   =====================================================
   Possible errors in function:
   fn x=>x + 1
   =====================================================
   -------------------- Error --------------------
   Error in expression:
       +

   Error at use of the operator.
   required type: bool * int -> 'a
   operator type: int * int -> int

   -------------------- Error --------------------
   Error in expression:
      (x, 1)

   Type inconsistent of requirement at the expression.
   required type: int * int
   actual type  : bool * int

   =====================================================
   Possible errors in argument:
   (fn y=> if y then true else false)(false)
   =====================================================
   -------------------- Error --------------------
   Error in expression:
      true

   Type inconsistent of requirement at the expression.
   required type: int
   actual type  : bool

   -------------------- Error --------------------
   Error in expression:
      false

   Type inconsistent of requirement at the expression.
   required type: int
   actual type  : bool
```

Fig. 9. $\mathcal{IEI}$ Algorithm: Precise Error Location

Others have previously suggested using a combination of type-checking algorithms to obtain additional information about type errors. For example, Rideau and Thery combined a variant of the $\mathcal{M}$ algorithm with the $\mathcal{W}$ algorithm, also a combination of the $\mathcal{W}$ and $\mathcal{M}$ algorithms found in Standard ML of New Jersey, Version 110.0.6.

$\mathcal{IEI}$ vs. Manifesto We compare the $\mathcal{IEI}$ algorithm to the Manifesto using Figure 9.

1. Correct. Like the $\mathcal{U}$AE algorithm, the $\mathcal{IEI}$ algorithm reports pairs of type conflicting sites. The types of the error sites in a reported pair are in conflict. Both sites are contributors to the type conflict. For example, in Figure 9 $\mathcal{IEI}$ reports that argument `x` of `int` type is supplied with a bool value.
2. Precise. The $\mathcal{IEI}$ algorithm reports the smallest conflicting sites. For example, in Figure 2 the $\mathcal{W}$ algorithm reports the whole expression as the error site. In Figures 3, the $\mathcal{M}$ algorithm reports the `true` which is only a part of the bool application. This reporting is not sufficient to identify the type error. In Figure 9, $\mathcal{IEI}$ reports that the argument `x` of `int` is supplied with `bool` value `true` or `false`, which is type conflicting.
3. Succinct. Similarly to the $\mathcal{U}$AE algorithm, $\mathcal{IEI}$ reports the type conflict relationship of the sites and their types. For example, in Figure 9 it reports that the required type for the application `(fn y ⇒ if y then true else false) (false)` is `int` by the requirement of the `+` and an operand `1` of `+`, but its actual type is `bool`.
4. Fairly mechanical. $\mathcal{IEI}$ reports the pairs of sites with type conflicts. But the type information of the pairs is inferred by the $\mathcal{M}$ algorithm, which is passed from the top of the expression to its subexpression and may not be as easy to understand as that from the $\mathcal{U}$AE algorithm.
5. Can be source-based. The implementation is partially source-based.
6. Unbiased. $\mathcal{IEI}$ does not assume the first use is the correct use.
7. Comprehensive. $\mathcal{IEI}$ finds a pair of the conflicting sites.

6 Summary and Conclusion

Figure 10 summarises the algorithms' performance against our manifesto. Correctness(1) is omitted from the summary because all of the algorithms are correct. Source-based(5) is omitted, because we believe that an implementation of any algorithm could be adapted to be source-based. We make the following observations:

- The error reporting of the $\mathcal{W}$ and $\mathcal{M}$ algorithms is very similar, with the $\mathcal{M}$ algorithm having a slight advantage in recursive definitions and tuple expressions, because it is more precise. However, the $\mathcal{M}$ algorithm error reporting may be harder to understand than that of the $\mathcal{W}$ algorithm, because it is not comprehensive, and stops too early to supply enough helpful information. For example, it stops at `true` in Figure 3 without identifying `false` as another conflict site. The $\mathcal{W}$ stop too late and may reports a large chunk of code as the error site. For example, it reports the entire application expression as an error site in Figure 2.

Manifesto	$\mathcal{W}$	$\mathcal{M}$	Wand	Johnson and Walz	Turner	$\mathcal{U}$AE	$\mathcal{IEI}$
2. Precise	poor	very good	good	good	good	good	very good
3. Succinct	fair	fair	poor	good	good	good	good
4. Amechanical	poor	poor	fair	fair	fair	good	fair
6. Unbiased	No	No	No	Yes	Yes	Yes	Yes
7. Comprehensive	poor	poor	good	very good	good	good	good

Fig. 10. Error Reporting Systems vs. Manifesto

- Wand, Johnson, Turner, $\mathcal{U}$AE, and $\mathcal{IEI}$ produce better error reports than the $\mathcal{W}$ and $\mathcal{M}$ algorithms. They do not stop too early as the $\mathcal{M}$ algorithm or stop too late as the $\mathcal{W}$ algorithm.
- Wand's system is the weakest because it is biased and not succinct. In the worse situation, it complains all the subexpressions as the error sites.
- It is hard to distinguish between Johnson, Turner, $\mathcal{U}$AE, and $\mathcal{IEI}$ from our manifesto.

In addition to the manifesto, we can make the following observation based on general considerations.

- Turner's implementation is source-based. The implementations of $\mathcal{U}$AE and $\mathcal{IEI}$ are partially source-based. We consider that this could considerably improve error reporting in production compliers.
- Johnson's and Turner's method reports the error site by the comparison of the number of different uses. But many errors are of a form where there is one correct usage and one incorrect usage, and Turner's method reports the whole expression as a large error site. We consider that reporting the two conflicting subexpressions is better than reporting the whole expression.
- The type information from the pure $\mathcal{U}$AE algorithm is self-explained, i.e. the user can infer the type information from the reported sites, and the user does not need look at other parts of the original expression to reason about the type information. This makes the type information easier to understand.
- To identify error sites with greater precision, the $\mathcal{IEI}$ algorithm uses global inference, often reporting types from a long chain of inference. In effect, the $\mathcal{IEI}$ algorithm gains greater precision at the price of understandability. Turner's type information is similar to that from the $\mathcal{IEI}$.

From the observations, we conclude that the error reporting from the $\mathcal{U}$AE algorithm is the easiest to understand. However the $\mathcal{U}$AE algorithm is limited to reporting the conflicting uses of the same program variable. Other algorithms have shortcoming in type information of their error reporting, that is the type information comes from a long chain of inference.

7 Future Work

We consider that although the error site is important, the type information is critical for understanding the error report. The type information in the error report needs succinct explanations to improve its understandability. To give better error reporting, it is important to understand how humans give error explanation. To overcome the shortcomings of the type explanation systems, including the type error explanation systems, we have observed how human experts explain type inference and type errors and are developing a system that reports error in a similar way [15]. We now intend to complete our new explanation system, in particular investigating the generation of maximally succinct explanations from human-like techniques.

References

1. Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2:17–30, March 1993.
2. Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.
3. T. B. Dinesh and Frank Tip. A slicing-based approach for locating type errors. In *Proceedings of the USENIX Conference on Domain-Specific Languages (DSL'97)*. Santa Barbara, CA, Oct. 1997.
4. Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.
5. 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 Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 44–57. ACM Press, January 1986.
6. Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.
7. Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J. T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL'98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.
8. Bruce J. McAdam. Generalising Techniques for Type Debugging. In Phil Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 49–57. Intellect, March 2000.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. Laurence Rideau and Laurent Thery. Interactive programming environment for ML. Technical Report 3193, Institut National de Recherche en Informatique et en Automatique, March 1997.
11. Helen Soosaipillai. An explanation based polymorphic type-checker for Standard ML. Master's thesis, Department of Computer Science, Heriot-Watt University, 1990.
12. David Turner. Enhanced error handling for ML. CS4 Project, Department of Computer Science, University of Edinburgh, 1990.
13. 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. ACM Press, January 1986.

14. Jun Yang. Explaining type errors by finding the source of a type conflict. In Phil Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect, March 2000.
15. Jun Yang, Greg Michaelson, and Phil Trinder. How do people check polymorphic types? In Alan F. Blackwell and Eleonora Bilotta, editors, *Twelfth Annual Meeting of the Psychology of Programming Interest Group Proceedings*, pages 67–77. Memoria, April 2000.