Explaining Type Errors by Finding the Sources of Type Conflicts

Yang Jun
ceejy1@cee.hw.ac.uk
Department of Computing and Electrical Engineering,
Heriot-Watt University

August 1999

Abstract

Typically a type error is reported when unification fails, even though the programmer's actual error may have occurred much earlier in the program. The W and M inference algorithms report the site where a type conflict is detected. The error message is isolated information: it is not clear from the error information the relationship between the site and the context in which the subexpression was typed. As a result, the error message may give little help to locate the source of the error.

This report investigates better methods of explaining type conflicts. We are trying to find a method that may be effective even when the user has little knowledge about type checking. We have implemented an improved type error explanation system and have proposed and implemented a hybrid algorithm.

1 Introduction

It is known that the place where a type error was detected may not be the place where the type error actually originated. The source of a type error may be far from where it is detected. This is because type-checking is processed from left to right and the context is refined by the previous type-checking processes . This is called left-to-right bias [1]. For example, after type checking *fn arg* $\Rightarrow$ *if arg then 3 else arg 3* NJ/SML-110 reports the whole if expression as the problem area - Figure 1. The error is the type conflicts between the monomorphic usages of *arg* in two sites, the first has type *bool* , the second has function type $int \rightarrow int$. Even worse, when the W algorithm succeeds at every subexpression and fails at the top application expression, it reports the whole expression as the error site, for example in Figure 2.

There are several type inference explanation system such as [4] , [6] and [7] . There are some inherent problems in textual explanations:
1. Type variables are the central point of the explanation since they are the ones that get refined. To understand the explanations, it is necessary to remember from which program variable the type variable

```
fn arg ⇒ if arg then 3 else arg 3

Error: case object and rules don't agree [tycon mismatch]
       rule domain: bool
       object: int →int
       in expression:
       (case arg
       of true ⇒ 3
       | false ⇒ arg 3)
```

Figure 1: Error message from NJ/SML-110 for conditional expression.

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

Figure 2: NJ/SML-110 gives the whole expression as the error site

fn arg ⇒ if arg 3 then 3 else arg 3.0
——————— Error ———————
Type conflicts in expressions:
arg(3)
arg(3.0)

the common program variables have type conflicts in different sites
from the first expression
arg: int → bool

but from the second expression
arg: real → int

Figure 3: Type conflicts in conditional expression

is inferred and where it is refined. For a certain number of type variables, it becomes impossible to remember what entity a type variable represents.
2. Sometimes the textual explanation has so much information, it become rapidly tedious.
3. The user needs sufficient knowledge about type checking to understand the explanations.
We are trying to find a method that may be effective even when the user has little knowledge about type checking.

2 Unification of assumption environments (UAE)

2.1 Introduction

Unification of assumption environments at the top level of AST, to treat each subexpression equally, can eliminate the left-to-right bias, and can point out all type conflicts automatically. This idea is due to Johan Agat and Jörgen Gustavsson, Chalmers Univ of Technology, June 1999 [11]. But ours is the first implementation of the idea. First, we show how the method works on the two simple examples.

Shown in Figure 3, after type checking *fn arg ⇒ if arg 3 then 3 else arg 3.0* , the system shows that the λ - bound argument *arg*, which is monomorphic, is used inconsistently in different subexpressions. At the first site *arg (3)*, *arg* is required to be *int → bool* type; at the second site *arg (3.0)*, *arg* is required to be *real → int* type.

Another example is shown in Figure 4. After type checking *fn x ⇒ (x 1, x true)*, the system shows that the λ - bound argument *x*, is used inconsistently in subexpressions of the tuple expression. At the first site *x (1)*, *x* is required to be *int → 'f* type; at the second site *x (true)*, *x* is required to be *bool → 'a* type.

fn x ⇒ (x 1, x true)

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

the common program variables have type conflicts in different sites
from the first expression
x: int → 'f

but from the second expression
x: bool → 'a

Figure 4: Type conflicts in tuple expression

2.2 The UAE Algorithm

The algorithm for unification of assumption environments is shown in Figure 5. We call this the UAE algorithm in later parts of this paper. In a unification-based type inference system, when unification fails, we have detected a type conflict. And it has been observed that where type conflicts are detected is often far from the site of the real error, because of the type-checking left-to-right bias.

Two principles are observed to implement the system:
1. Type error messages are reported the context of use. The philosophy is that a subexpression in an expression has connections with other subexpressions in some manner; to report the relationships of its use in context is better than to report just an isolated subexpression as an error site.
2. Error explanations should be complete and concise; the system should highlight the sources of type conflicts which directly contribute to the conflict but nothing more;

The algorithm removes the left-to-right bias by unifying the assumption environments at the top level of AST: in this way every subexpression is treated equally. First, the algorithm type-checks each subexpression individually as an open expression; this returns a substitution, a type for the subexpression and an assumption environment. This means that the subexpression has the type under the assumptions of the free variables in the assumption environment. Then the algorithm unifies the assumption environments from each subexpression at the top level of the AST to restore the type of each program variables. The free program variables in the top level application are constrained to be monomorphic, otherwise, a monomorphic type system should be used [4].

In the implementation, predefined functions such as *map* and operators such as + which are free variables in an expression are excluded from the assumption environments. It is more efficient when the predicates with known types are excluded from the assumption environments; the assumption environments are smaller. The unification procedure $GenU_{env}$ makes sure the definition of x in *let* is consistent with its usages in the subexpression e_2 of the *let* expression. There are two environments, the context *TE* which is passed downwards, the assumption environment *FTE* which is passed upwards during type checking.

The algorithm is syntax-directed and there is a type rule for each syntactic construct. Figure 5 shows the unification of assumption environments in application and the LET expression.

W: TypeEnv $\times$ Expression $\rightarrow$ Substitution $\times$ Type $\times$ FreeTypeEnv

Def W(TE, e) =

case e of

$x \Rightarrow$

if $x \notin Dom(TE)$ then

let γ be a new type variable

in (ID, a, FTE+$\{$x $\mapsto \gamma\}$)

else let

$\forall a_1 \cdots a_n.\tau = TE(x)$

$\beta_1, \cdots, \beta_n$, be new type variables.

in

$(ID, \{a_i \mapsto \beta_i\}\tau, emptyFENV)$

endif

$\lambda x.e_1 \Rightarrow$

let

β be a new type variable.

$(S_1, \tau_1, FENV) = W(TE + \{x \mapsto \beta\}, e_1)$

in

$(S_1, S_1\beta \rightarrow \tau_1, S(FENV \setminus \{x\})$

$e_1 e_2 \Rightarrow$

let

$(S_1, \tau_1, FENV_1) = W(TE, e_1)$

$(S_2, \tau_2, FENV_2) = W(TE, e_2)$

β be a new type variable

$S_{env} = U_{env}(S_1 FENV_1, S_2 FEVN_2)$

$S_3 = U(S_2\tau_1,\ \tau_2 \rightarrow \beta)$

in

$(S_{env}S_3S_2S_1, S_3\beta, S_{env}(FENV_1 \cup FENV_2))$

let $x = e_1$ in $e_2 \Rightarrow$

let

$(S_1, \tau_1, FENV_1) = W(TE, e_1)$

$(S_2, \tau_2, FENV_2) = W(TE, e_2)$

$TE = S_1(TE) + \{x \mapsto \sigma\}$

$(S_{env}, FENV) = GenU_{env}(S_1(FENV_1 \cup TE, S_2 FEVN_2)$

in

$(S_{env}S_2S_1, S_{env}\tau_2, FENV_1 \cup S_{env}(FENV \cup (FENV_2 \setminus \{x\}))$

endcase

Figure 5: The UAE Algorithm

3 Incremental error inference

3.1 Introduction

We propose an incremental error inference algorithm that finds the type conflicts in application expressions when the UAE algorithms cannot. The W algorithm fails only at a function application, and an erroneous expression is often successfully type-checked long before its consequence collides at an application. When the W algorithm succeeds at every subexpression and fails at the top application expression, it reports the whole expression as the error site, for example in Figure 2, implying some of its subexpressions are ill-typed. This error message is not very useful for locating the sources of type errors. In the case of application, when the W algorithm successfully type-checks each subexpression , fails at the top level expression, and each subexpression has no common free variables, the UAE algorithm behaves the same as the W algorithm, and cannot improve the error reporting.

The folklore algorithm - the M algorithm always stops earlier than the W algorithm [9] when there is a type error and it can be used to cure the problem by reporting a fine grain error site. The M algorithm brings the context of an expression down to its sub-or-sibling expressions. It keeps carrying a type constraint(or an expected type) that each expression must satisfy where the expression appears. For example, in the case of application, if the required type for application expression $e_1 e_2$ 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.

The M algorithm reports a finer grain site, at a constant, a variable, or a lambda expression. For example, in the example as - Figure 2, the M-algorithm stops at *true*. But we still do not know where the type conflicts are .

We combine the M algorithm and the W algorithm on-the-fly. When the W algorithm fails at the top expression, we switch to the M algorithm, which always stops at a site which is smaller than the whole expression. If the M algorithm stops at an argument site, it means the argument has a type conflict between the function and its arguments. For example, in the example as - Figure 2, when the M algorithm stops at *true*, it means that the function *(fn $x \Rightarrow x$ + 1)* has a type conflict with the type of *((fn y$\Rightarrow$ if y then true else false) false)*. We find the conflict by assuming that the site where the M algorithm found a type inconsistency is type correct, and then type checking the function node on the AST under the assumption. Hence we can find the conflicting site in the function applications, and the reason for the conflicts. In this way, we erase the type-checking left-to-right bias.

3.2 Examples

We give some examples of application expressions, where the W algorithm succeeds at every subexpression but fails at the top level, and reports the whole application expression as an error site: here the M algorithm reports an isolated error site. In these examples, there are no common program variables in the subexpressions, so the UAE algorithm behaves in the same way as the W algorithm. The incremental error inference algorithm INC_{app}, gives error explanation messages by finding a pair of directly type conflicting sites, and showing the reasons for their conflicts.

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

Error in expression:
    +
Error at use of the operator.
    required type: bool * int → 'k
    operator type: 'l * 'l → 'l
——————— Error ———————
Error in expression:
    (fn y⇒ if y then true else false)(false)
Type inconsistent with requirement at the expression.
    required type: int
    actual type : bool

Figure 6: Error message from incremental conflicts inference for anonymous function application.

3.2.1 Anonymous function application

Consider *(fn x ⇒ x+1) ((fn y⇒ if y then true else false) false)* , the example in Figure 2. While the W algorithm reports the whole expression as the error site, the M algorithm reports the *true* in the subsequent application expression as the error site. INC_{app} reports the + and the subsequent application expression as the directly conflicting usages - Figure 6. INC_{app} gives the reason of the conflicts as well. The required type for the operator + is *bool * int → 'k*, means that if the subsequent application expression is correct, then it is assumed that the argument of *(fn x ⇒ x +1)* is of *bool* type. So the the usage of + is not correct, which means that + should be replaced by another function or its two arguments *x* and *1* are not correct usages. Alternatively, if *(fn x ⇒ x +1)* is correct, then the required type for the subsequent application expression is of *int* type, but the application expression actually is of *bool* type.

3.2.2 Predicate function application

Consider the type check of *map(fn x⇒ x)(3)* . The M algorithm reports the *3* as the error site. INC_{app} traces back of the application, reports the predicate function *map* and the constant *3* as the directly conflicting usages - Figure 7. INC_{app} gives the required type for *map* in the context and its predefined type which can be considered as the interface for the argument function – the first argument should be a function, the second argument should be a list. From this information, the user can check the usage of its two arguments and find that the second argument is a integer rather than a list. Alternatively, if the second argument *3* is a correct usage, then the application of *map* to the first argument is not correct.

3.2.3 LET- bounded function application

Consider the LET expression *let fun inc x = x +1 in map inc [1.0, 2.0] end.* The W algorithm reports the whole application expression *map inc [1.0, 2.0]* as the error site. The M algorithm reports the *1.0* in

```
map(fn x⇒ x)( 3)
—————— Error ——————

Error in expression:
        map
Error at use of the operator.
        required type: 'f → int → 'e
        operator type: ('g → 'h) → 'g list → 'h list
—————— Error ——————
Error in expression:
        3
Type inconsistent with requirement at the expression.
        required type: 'b list
        actual type :   int
```

Figure 7: Error message from incremental conflicts inference for predefined function application.

```
let fun inc x = x +1 in map inc [1.0, 2.0] end
—————— Error ——————

Error in expression:
        inc
Type inconsistent with requirement at the expression.
        required type: real → 'i
        actual type : int → int
—————— Error ——————
Error in expression:
        [1.0, 2.0]
Type inconsistent with requirement at the expression.
        required type:int list
        actual type : real list
```

Figure 8: Error message from incremental conflicts inference for LET bounded function application.

map (fn x ⇒ x + 1.0) [3, 4, 5]
————— Error —————

Error in expression:
+
Error at use of the operator.
required type: int * real → 'R
operator type: 'S * 'S → 'S
————— Error —————
Error in expression:
[3, 4, 5]
Type inconsistent with requirement at the expression.
required type: real list
actual type : int list

Figure 9: Error message from incremental conflicts inference for structured data in function application.

the list as the error site. INC_{app} reports the LET-bounded function *inc* and the list as the conflicting sites -Figure 8. This contrast with the anonymous function application -Figure 6 and Figure 9, where INC_{app} reports the operator or argument function as the error site.

3.2.4 Structured data in application

Consider the type check of *map (fn x ⇒ x + 1.0) [3, 4, 5]* . The W algorithm reports the whole expression as the error site; the M algorithm reports the *3* in the list as error site. In the anonymous function application, INC_{app} reports the operator + in the function and the list as the type conflicting sites. The error explanation is straightforward. If the usage of list *[3, 4, 5]* is correct, then the + should be replaced by another function of type *int * real → 'R* or the usage of its two arguments x and *1.0* are not correct; if + is a correct usage, then the list *[3, 4, 5]* should be replaced by an real list; this is better than the message from the M algorithm which only reports the constant *3* as an error site. Because the algorithm should make sure that structured data is well typed first, and *3* is an element of the structured data, the M algorithm only reports an element as the error site, and it would be confused as to whether the structured data is well typed.

3.3 The INC_{app} algorithm

The UAE algorithm may be improved in the case of applications - Figure 10. When the algorithm succeeds at every subexpression and there are no common free variables in every subexpression, if the algorithm fails at the top level expression, then it is switched to the modified M algorithm M_{app} - Figure 11.

W: TypeEnv × Expression → Substitution × Type × FreeTypeEnv
Def W(TE, e) =
    case e of
    $e_1 e_2 \Rightarrow$
        let
            $(S_1, \tau_1, FENV_1) = W(TE, e_1)$
            β, β_1, β_2 are new type variables
            $S_1' = Unify(\tau_1, \beta_1 \rightarrow \beta_2)$
                handle Unify $\Rightarrow$
                    let $TE = TE \cup FENV_1$
                    in $M_{app}(TE, e, \beta)$
            $(S_2, \tau_2, FENV_2) = W(TE, e_2)$
            $S_{env} = Unify_{env}(S_1 FENV_1, S_2 FEVN_2)$
            $S_3 = Unify(S_2 \tau_1, \tau_2 \rightarrow \beta)$
                handle Unify $\Rightarrow$
                    let $TE = TE \cup FENV_1 \cup FENV_2$
                    in $M_{app}(TE, e, \beta)$
        in
            $(S_{env} S_3 S_2 S_1, S_3 \beta, S_{env}(FENV_1 \cup FENV_2))$
    endcase

Figure 10: Switch to M_{app} in application subexpression.

3.3.1 Error reporting

The M algorithm does not unify types at application expressions. It unifies at constant, variable and lambda expressions. The modified M algorithm M_{app} chooses predefined functions such as *map, o, hd, tl* and operators such as *+, -, div* etc in a subexpression as the only error reporting sites for the subexpression. The reasons are :

1. Those functions and operators always have correct types, so their types can be taken as the standards or interface for their usage.
2. The user is familiar with their types, and recognizes their usages easily; For other program variables, the user usually is not aware of their types, or does not care about their types during programming.
3. Their inferred type constraints are good clues for analyzing the usages of their arguments.

3.3.2 Incremental type error inference for application

The incremental type error inference algorithm INC_{app} for application subexpression is shown in Figure 12.

M_{app}: TypeEnv × Expression × Type → Substitution

```
Def M_app (TE, e, ρ) =
    case e of

    x ⇒
        if x ∉ Dom(TE) then fail
        else let
                ∀a_1 ··· a_n.τ = TE(x)
                β_1, ···, β_n, be new type variables.
        in
                Unify(ρ, {a_i ↦ β_i}τ)
    Predicate ⇒
        let
            ∀a_1 ··· a_n.τ = TE(Predicate)
            β_1, ···, β_n, be new type variables.
        in
            Unify(ρ, {a_i ↦ β_i}τ)
            handle Unify ⇒ report type error
    λx.e_1 ⇒
        let
            β_1, β_2 be new type variables.
            S_1 = Unify(ρ, β_1 → β_2)
            S_2 = M_app(S_1TE + x : S_1β_1, e, S_1β_2)
        in
            (S_1S_1)
    e_1e_2 ⇒
        let
            β be an new type variables.
            S_1 = M_app(TE, e_1, β → ρ)
            S_2 = M_app(S_1(TE), e_2, S_1β)
            handle Unify ⇒
                        let S_2" = M_app(TE, e_2, β)
                            β_3 be an new type variables
                        in M_app(S_2" TE, e1, S_2"β → β_3)
                            handle Unify ⇒ report error in e_1;
                            report error in e_2
        in
            (S_2S_1)
    let x = e_1 in e_2 ⇒
        let
            S_1 = M_app(TE, e_1, β), β be a new type variable
            S_2 = M_app(S_1(TE) + {x ↦ Close_TE(S_1β)}, e_2, S_1ρ)
        in
            (S_2S_1)
    fix f λ.e ⇒
        M_app(TE + {f ↦ ρ}, λ.e, ρ)

    endcase
```

Figure 11: Pseudo-code for the M_{app} algorithm.

Def INC_{app} (TE, e) =

case e of

$F_1 F_2 \Rightarrow$

If W succeeds at both F_1 and F_2, and fails at the top level of the application,

then switch to M_{app} -Figure 11

reports the type conflicts in F_1 by assuming the usage of F_2 is correct in INC_{app};

reports the type conflicts in F_2 by assuming the usage of F_1 is correct in INC_{app};

If W fails at F_1 or finds F_1 is not a function,

then switch to M_{app} to found more detailed type error reasons for F_1;

If a type error is detected at F_2 by W

then trace back F_1 with UAE of the FTEs from F_1 and F_2;

$F_1(F_2 F_3) \Rightarrow$

if a type error is detected at F_2 by W

then trace back F_1 with UAE of the FTEs from F_1 and F_2;

if a type error is detected at F_3 by W

then trace back F_2 first, if no type conflicts are found in F_2 then trace back F_1 ;

if W succeeds at F_1, F_2 and F_3 but fails at the top level of the applications

then switch to M_{app} , which checks each subexpression in INC_{app} .

$(F_1 F_2)F_3 \Rightarrow$

if a type error is detected at F_2 by W

then trace back F_1;

if a type error is detected at F_3 by W

then trace back F_1 and F_2 sequentially by UAE;

if W succeeds at F_1, F_2 and F_3 fails at the top level of the applications;

then switch to M_{app} , which checks each subexpression in INC_{app} .

endcase

Figure 12: The incremental type error inference INC_{app} for application.

4 Discussion

We now discuss different type error explanation methods.

Type error explanation systems: In Duggan's explanation system [6], internal type variables which act as bridge between instances, are actually the central point of the explanation, since they are the ones that get refined, but a programmer would not be concerned with those type variables. Beaven's Explaining Type Errors in Polymorphic Languages [4] gives a long list of explanation information about why an expression comes to its type. It would be a great burden to understand this information which explains the working process of the W algorithm. Experts usually find this explanation too detailed to be of real help, though they find very valuable information about the different positions in the programs that contribute to the type given by the tool. [10]

Listing the possible error sites: Mitchell Wand's approach [3] keeps a record of the pieces of code that contribute to each type deduction when type errors are detected, using the 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. About 30 percent of sites are real source of type errors. 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.

Maximum-flow approach: Greg F. Johnson and Janet Walz [2] 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. But 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.

Debugging type errors - see [8] and [5]. This provides a method for diagnosing type errors where the user can probe the internal type information which the ordinary inference algorithm hides. This is done manually. Sometimes it is difficult to make out the connections between each use of subexpressions.

Explaining type errors by type conflicts has some advantages, compared with the above methods, as it
1. Explains type errors by relation of usages in the context, not isolated individual usage.
2. Automatically finds the sources that directly contribute to the type conflicts.
3. Erases the left-to-right type checking bias.
4. Can be understood even when the user has little knowledge about type checking.

5 Conclusions and Future Work

Compared to type inference explanation systems , the UAE algorithm automatically locates the sources of type conflicts, and gives a pair of direct conflicting error sites rather than individually isolated error sites. This removes the left-to-right type-checking bias. When the W algorithm succeeds at every subexpression in an application and fails at the top application, the W algorithm reports the whole expression as an error site, and the M algorithm reports a constant, a variable or a anonymous function as an error site. The philosophy of our error message approach is to report type errors by the relation of type conflict sites rather than by an isolated error site. The incremental type error inference method INC_{app} cures this problem by tracing back the application nodes in the AST.

The UAE algorithm implemented in the W algorithm is straightforward to implement in M the algorithm. The only modification is to decide when to switch to incremental error inference to type-check an expression when all its subexpressions have no common program variables, or type-check those subexpressions that have no common program variables. Those algorithms can be implemented in an interactive programming environment.

We now propose to look at other sources of type explanation information. For example, current methods give sources of type errors at applications, variables or constants. But they cannot point out type conflicts between constants. Sometimes the result type of an expression depends on the types of constants. In the following example, the source of the type inconsistency comes from the use of the two underlined constants *1* and the constant *0.0*.

$$
\begin{array}{l}
\textit{fun } m\ f\ \underline{[1]} = \underline{[1]} \\
\quad\mid m\ f\ nil = nil \\
\quad\mid m\ f\ (h{::}t) = \textit{if } (h > \underline{0.0}) \\
\qquad\qquad\qquad \textit{then } (f\ h) :: (m\ f\ t) \\
\qquad\qquad\qquad \textit{else } m\ f\ t
\end{array}
$$

We also wish to develop a canonical set of test examples, that will be used for the comparative evaluation of type explanation systems.

6 Acknowledgements

I am grateful to Greg Michaelson and Phil Trinder for their help and encouragement. I am also grateful to Johan Agat and Jörgen Gustavsson for interesting discussion and useful ideas about the unification of assumption environments.

The work is supported by an ORS studentship and Heriot-Watt University.

References

[1] Bruce J. McAdam. *On the unification of substitutions in type-inference.* Technical Report ECS-LFCS-98-384, Department of Computer Science The University of Edinburgh, James Clerk Maxwell Building, The Kings Buildings. 1998.

[2] Greg F. Johnson and Janet Walz. *A maximum-flow approach to anomaly isolation in unification-based incremental type inference.* In Thirteenth Annual ACM Symposium on Principles of Programming Languages, pages 44-57. Association for Computing Machinery, ACM Press, January 1986.

[3] Mitchell Wand *Finding the source of type errors.* In Thirteenth Annual ACM Symposium on Principles of Programming Languages, pages 38-43. Association for Computing Machinery, ACM Press, January 1986.

[4] Beaven, M., and Stansifer, R. *Explaining type errors in polymorphic languages.* TOPLAS '93, vol 2, pp17-30, March 1993.

[5] Bernstein, K. L., and Stark, E. W. *Debugging type errors (Full version)*, Tech Report, State University of New York at Stony Brook , November 1995

[6] Dominic Duggan and Frederick Bent. *Explaining type inference.* Science of Computer Programming, (27):37-83, 1996.

[7] Helen Soosaipillai. *An explanation based polymorphic type-checker for Standard ML.* Master's thesis, Department of Computer Science, Heriot-Watt University, 1990.

[8] Zhong Shao and Andrew Appel. *Smartest recompilation.* In Twentieth Annual ACM Symposium on Principles of Programming Languages, pp 439 - 450, ACM Press, January 1993.

[9] Oukseh Lee and KwangKeun Yi. *Proofs about a folklore let-polymorphic type inference algorithm.* ACM Transactions on Programming Languages and Systems,Vol. 20, No. 4, pp. 707-723, July 1998 .

[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] Johan Agat and Jörgen Gustavsson. *Personal communication* Chalmers Univ of Technology, June 1999.