Improving Polymorphic Type Explanations

by

Jun Yang

Submitted for the Degree of
Doctor of Philosophy
at Heriot-Watt University
on Completion of Research in the
Department of Computing and Electrical Engineering
May 2001

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acknowledgments

I am most grateful to Greg Michaelson and Phil Trinder, my supervisors. They were always ready to listen to my work with patience. They pulled me through the different attempts of type explanations with very helpful suggestions and with a clear sense of direction. I can't thank them enough for those.

I heartily thank Joe Wells for his collaboration on the $\mathcal{U}$AE algorithm, and all our colleagues who took part in our expert type checking experiments.

I also thank ORS and Heriot-Watt University for continuing support for my study. I wish thank the Department of Computing and Electrical Engineering for supplying a partial scholarship to support my study.

Finally, I am grateful beyond words for the love and affection bestowed upon me from my family. I thank my family for their great efforts in giving financial support to enable me to complete my PhD study.

ABSTRACT

The benefits of static parametric polymorphic typechecking based on the Hindley-Milner scheme are well known: it minimizes semantic errors while improving component generality. However, systems based on the $\mathcal{W}$ algorithm and its variants tend to offer poor support for understanding type checking and type errors. While this efficient algorithm is widely used in functional language implementations, it does not correspond well to how people understand the types in programs. Understanding the types in reports produced by those algorithms tends to require the programmer to follow a chain of inference that may be long and complex. The important aspects of type error reports are accurately reporting the program text that contributes to the conflict and giving a good explanation of why the relevant program text causes a conflict. The type information in the error report needs succinct explanations to improve its understandability. To make these explanations better, it is important to understand how humans understand type errors.

We have investigated the use of interactive visualization to explain inferred type information. A novel visualization of functions and their types is presented based on the generation of 2D function-specific icons with graphical type representations which change dynamically as functions are applied. This visualization has been implemented for a Standard ML subset within a graphical environment in which expressions are constrained by type matching. It is shown that visualization has different strengths compared with a textual representation.

We have investigated better methods of explaining type conflicts. The philosophy of our approach is to find sources of type errors by reporting which parts of the program conflict, rather than identifying isolated error sites. We have implemented two new inference algorithms with this philosophy: the Unification of Assumption Environments ($\mathcal{U}\mathbf{AE}$) and the Incremented Error Inference ($\mathcal{IEI}$). Both algorithms are formally defined, and their error reporting is compared with conventional algorithms using numerous examples.

To understand how human experts explain type inference, experiments into human checking of both correct and incorrect polymorphic typed programs have been conducted. The results of the experiments and their implications for our proposed new checker are discussed.

A new algorithm has been devised to give human-like type explanations to help user to understand type inference in the style of Hindley-Milner type system. In particular, the system seeks to provide succinct explanations with minimal reference to artifacts of mechanized type inference. The system does not use type variables in the explanation if the type is not polymorphic. The system has been used primarily for explaining type correct programs, but the implications for type error explanation are also discussed.

CHAPTER 1
INTRODUCTION

Type systems are fundamental to programming languages in constraining combinations of constructs, both to ensure the well-formedness of programs and to reduce runtime errors. However, despite considerable research into type systems, almost all widely used languages provide relatively impoverished sets of types, by direct abstraction from those explicit at a low level in von Neumann computers. Thus, modern imperative languages typically offer: integers, floating point numbers, characters and booleans, corresponding to fixed size bit patterns; arrays and records, corresponding to fixed size bit pattern sequences; pointers or references, corresponding to machine addresses of bit patterns. More elaborate types, such as discriminated unions or objects, are usually so much syntactic sugar on top of this basic type repertoire.

A typical imperative language has a monomorphic type system, where a construct retains the same concrete type throughout its lifetime. In particular, once a variable is assigned a type it may only ever be associated with values of that type. Strong monomorphic type systems enable simple static checks on programs for type consistency but unduly restrict programming flexibility.

In contrast, in polymorphic type systems, a construct may take different types in different contexts. Strachey [56] distinguished ad-hoc from parameterized polymorphism. Weakly typed languages like LISP and Prolog, with ad-hoc polymorphism, allow variables to be associated with arbitrary typed values without any statically enforced constraints. While this enables considerable flexibility in programming, static checks on programs for type consistency may no longer be possible. Run-time type checking, to avoid invalid construct combinations, is time consuming and brings attendant space overheads for explicit type tagging on values.

In parameterized polymorphism, constructs are strongly typed but those types may include abstraction points. This enables the definition of generalized constructs with generic types, which are resolved when a concrete instance of the construct is created. The abstraction points are indicated by type variables, which may be uniformly replaced with other types: hence parameterized polymorphism.

For example, a generic list type, with an empty list `nil` and a constructor `CONS` may be defined in SML as:

```
datatype 'a list = nil | CONS of 'a * 'a list
```

This introduces a new type `'a list`. The type variable `'a` may be instantiated to any type, and the specific type is deduced from the context in which `CONS` is applied. Thus,

```
CONS(1,CONS(2,CONS(3,nil)))
```

is an `int list` because 1, 2 and 3 are `int`s. Similarly,

```
CONS("a",CONS("b",CONS("c",nil)))
```

is a `string list` because `"a"`, `"b"` and `"c"` are `string`s.

An important contributions to programming language design has been Milner's and Hindley's development of first order parameterized polymorphic type inference. This enables static type checking at compile time, minimizing the need for run-time type representations and checking. Hindley-Milner type systems [35] have most commonly been incorporated into functional languages like Standard ML(SML)[36], Miranda[58], Hope[11] and Haskell[24].

There is considerable anecdotal evidence that polymorphic type checking greatly reduces semantic errors in programs. Certainly, polymorphic types are central to the succinctness of contemporary functional languages, enabling a considerable degree of type-safe abstraction and hence component re-use. Nonetheless, parameterized polymorphism is provided in few languages in other paradigms, exceptions being the Napier imperative persistent database language [38], and λ-Prolog[34], a strongly typed Prolog variant used particularly for theorem proving.

It is possible that this apparent resistance to including polymorphic type systems in contemporary languages (e.g. Java) lies in a combination of unfamiliarity, and the aura of complexity and mathematical sophistication surrounding such systems. For example, in C++ and Java, there are virtual functions defined to enable runtime binding. Virtual functions require to set up extra code space and execution time which are not efficient. Sometimes, students are confused by when the virtual functions are needed. It would be useful to explain how a compiler make the run time binding and how a compiler makes the early binding. The later development of PC-lint [54] for finding error in C/C++ program extensively uses type information, but the error messages from the system sometimes are very confusing and it requires add casting information which make the program hard to read. In particular, it is observed that there is a fundamental mismatch between how polymorphic type inference algorithms deduce types and how people reason about types, which is particularly apparent in the reporting of polymorphic type errors [66].

This dissertation discusses several approaches to making polymorphic types more accessible. A new approach to explaining type information based on visualization has been explored. A study of human experts explaining types shows that they behave differently to those systems that explain how the $\mathcal{W}$ algorithm works. A system incorporating human techniques has been built incorporating a novel type inference algorithm.

1.1 Organization

Chapter 2 introduces Hindley-Milner's type inference for parameterized polymorphism and its limitations for providing type explanations is discussed. A survey of related work is presented [69].

Chapter 3 reports a visualization approach to presenting polymorphic types and function applications [63, 64]. A color representation of polymorphic types shows clearly the number of arguments and the type of each argument.

Chapter 4 defines a new type inference algorithm based on unification of assumption environments ($\mathcal{U}_{\mathbf{AE}}$) and a new error reporting algorithm ($\mathcal{IEI}$) [62, 70].

Chapter 5 introduces a study of how people explain types, and elucidates a simple set of rules which form the basis of our human-like type explanation system [66].

Chapter 6 describes an human-like type explanation system. The system uses human techniques to give type inference explanations which are very different to those from other systems [67, 68].

Chapter 7 draws conclusions from the research and considers future work.

1.2 Contributions

The following contributions are made in the fields of type explanation and programming environment.

1. *Visualization of polymorphic types* gives a novel approach to representing types compared with traditional textual representations.

2. $\mathcal{U}$AE and $\mathcal{IEI}$ are algorithms designed to improve error reporting. $\mathcal{U}$AE aims to improve the understanding of error reporting while $\mathcal{IEI}$ improve precision of error reporting at the cost of the understandability of type information.

3. *Human expert type explanation techniques* are identified. Experiments with expert human type checkers confirmed our intuitions that people tend to give explanations that avoid intermediate representations such as type variables and abstract syntax trees.

4. *A system that mimics human behavior* is completed. While there are some differences, comparison suggests that our system has similar behavior to human experts.

5. A survey of type explanation systems and a survey of type error reporting systems are provided.

1.3 Authorship

The visualization system presented at Chapter 2 uses the lexical and syntax analyzer, and type checker written by Greg Michaelson. The $\mathcal{U}$AE algorithm described at Chapter 4 is collaborative work with Joe Wells. Much of the thesis adapts text and data from the papers written with Greg Michaelson and Phil Trinder, as follows.

1. Jun Yang, Greg Michaelson and Phil Trinder: **Human and "human-like" type explanations**, *Proceedings of 13th annual workshop of PPIG,* G. Kadoda, editor, pages 163-172, University of Bournemouth, 17-20 April 2001.

2. Jun Yang, Greg Michaelson and Phil Trinder: **Explaining polymorphic types**, *Computer Journal,* Accept, Feb 2001.

3. Jun Yang, Greg Michaelson and Phil Trinder: **How do people check polymorphic types?** *Twelfth annual meeting of the psychology of programming interest group proceedings,* Alan F. Blackwell etc, editors, pages 67-77, Memoria, April 2000.

4. Jun Yang, Greg Michaelson and Phil Trinder: **Helping Students Understand Polymorphic Type Errors**, *Proceedings of 1st Annual Conference of the LTSN centre for Information and Computer Sciences,* pages 11-19, August 2000.

5. Jun Yang, Greg Michaelson: **A visualisation of polymorphic type checking**, *Journal of Functional Programming,* 10(1) , pages 57-75, January 2000.

6. Jun Yang, Greg Michaelson and Phil Trinder: **Explaining polymorphic types through visualization**, *Proceedings of 7th Annual Conference on the Teaching of Computing,* S. Alexander and U. O'Reilly, editors , pages 73-77, August 1999.

7. Jun Yang, Phil Trinder, Joe Wells and Greg Michaelson: **Examples to Compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}$AE, $\mathcal{IEI}$ algorithms**, *Technical report RM/00/12, Department of Computing and Electrical Engineering, Heriot-Watt University,* Oct 2000.

8. Jun Yang, Phil Trinder, Greg Michaelson and Joe Wells: **Improved type error reporting**, *Draft Proceeding of IFL2000 Implementation of Functional Languages,* 12th International Workshop, pages 71-86, September 2000.

9. Jun Yang, **Explaining type errors by finding the source of a type conflict**, *Trends in Functional Programming,* Phil Trinder, Greg Michaelson and Hans-Wolfgang Loidl,editors, pages 58-66, Intellect, March 2000.

CHAPTER 2
BACKGROUND

Summary

This chapter introduces Hindley-Milner type inference and indicates some difficulties people have with understanding it. To support the understanding of type inference, several type explanation systems have been developed based on tracing the behaviour of the $\mathcal{W}$ algorithm. Furthermore, there have been several approaches to improve error reporting over the $\mathcal{W}$ and $\mathcal{M}$ algorithms by making reporting comprehensive and more intuitive. The strength and difficulties of these systems are discussed.

2.1 Hindley-Milner Type Inference

One of the greatest contributions to functional languages is the Hindley-Milner type system. Hindley studied expressions of the lambda calculus with constants and the typing of expressions in combinatory logic [21]. He considers type expressions built from base types using the type constructor for function types. In Hindley's terminology, a `type scheme` is a type in which type variables can occur. Hindley defines the type inference rules in the form `A` $\vdash$ α `X` where `X` is an expression, α is a type scheme and `A` is a set of statements assigning a type scheme to every variable that occurs free in `X`. He proves the existence of principal type schemes: for all `X`, if for some α and `A` one has `A` $\vdash$ α `X` then there exists a type scheme, α_0, with the property that for all `A'`, α`'`, if `A'` $\vdash$ α`'` `X` then α`'` is a substitution instance of α_0, i.e. there exists a substitution S, so that $\alpha' = S\alpha_0$.

Milner [35] independently developed essentially the same system and extended the type inference rules so that a function once declared can be applied to expressions of different types. For example:

```
let fun I x = x in (I 3, I true) end
```
(2.1)

is typable in Milner's system.

Milner introduced a polymorphic type system. In Milner's type discipline there is a distinction between a `type scheme` and all the `instances` of the `type scheme`. A `type scheme` is a `generic type`. One of the typing rules is that if an expression has a `generic type`, then it also has all instances of the generic type. Milner's notion of a `type scheme` is an expression, ρ, of the form $\forall\bar{\alpha}.\beta$, where β is a type scheme in Hindley's system, and $\bar{\alpha}$ are `bound` type variables that are bound in ρ. E.g., in expression 2.1 `I` has type scheme $\forall\alpha.\alpha \to \alpha$ and all its instances:
`int -> int`, `bool -> bool`, `real -> real`, `int list -> int list`, and so on.

In expression 2.1, when `I` is declared, its type can be presented as type scheme $\forall\alpha.\alpha \to \alpha$ which in the application `I 3` can be instantiated to `int -> int`, and in the application `I true` can be instantiated to `bool -> bool`.

Milner's extension enables a large class of programs to be typed. During inference, the inferred types are checked for consistency. There is a type checking algorithm known

$x \in \text{ExpVar}$		expression variable
$P \in \text{Patt}$ ::=	x	pattern
|	(P, P')	
|	$[]$	
|	$P :: P'$	
$M \in \text{Match}$ ::=	$P \Rightarrow e$	match
$e \in \text{Exp}$ ::=	x	variable
|	$+$	arithmetic operator
|	$\texttt{fix}\ x'.\lambda M_1, \ldots, M_n$	lambda abstraction and recursive definition
|	$e_1 e_2$	application
|	$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	
|	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	
|	(e_1, e_2)	tuple
|	$(e_1 :: e_2)$	list
|	$[]$	empty list

Figure 2.1: Abstract syntax of `Exp`

.

as the $\mathcal{W}$ algorithm [35], which is used in the language Standard ML [36] and Haskell [27]
.

2.1.1 Hindley/Milner type discipline

We define a syntax and typing rules for a small language `Exp`, depicted in Figure 2.1. The language will be used to discuss the $\mathcal{W}$ algorithm, the $\mathcal{M}$ algorithm and two new inference algorithms: the $\mathcal{U}_{\text{AE}}$ algorithm (Chapter 4) and the $\mathcal{H}$ algorithm (Chapter 6)
.

`Exp` is based on the λ-calculus extended with pairs, lists, pattern matching parameters, booleans and conditionals, integer and floating point ("real") numbers, overloaded arithmetic operators (we use + as an example), and recursive functions. The language also has let-expressions enabling polymorphic types for the variables they define.

The set `Type` defined in Figure 2.2 is typable in the usual Hindley/Milner style and has numeric, function, list, and product types. Ordinary `type variables` are named using α and β.

`Type schemes` defined in Figure 2.2 (simple types modified by outermost quantification over some or all of the type variables) are named using σ.

The set `TypeSchemeEnv` defined in Figure 2.2 contains `type scheme environments` which are finite maps from program variables to type schemes. Type scheme environments are named using TE. Type scheme environments can be combined by the operator $\oplus$, which takes the type from its right argument when both arguments are defined for a program variable.

$\tau \in$ Type ::= $STRING \mid BOOL$	string and bool
$\mid \alpha$	type variable
$\mid \tau \rightarrow \tau$	function type
$\mid \tau$ *list*	list type

Let $\bar{\alpha}$ range over Fin(TyVar).

$\sigma \in$ TypeScheme ::= $\tau \mid \forall\bar{\alpha}.\tau$	type scheme
$\sigma_1 = \sigma_2$ if σ_1 and σ_2 are α-convertible	identifying type schemes
e.g. $(\forall\{\alpha\}.\alpha) = (\forall\{\beta\}.\beta)$	
$TE \in$ TypeSchemeEnv = ExpVar $\stackrel{fin}{\rightarrow}$ TypeScheme	type scheme environment

Figure 2.2: Type, type scheme and type environments.

2.1.1.1 Inference rules

Hindley/Milner type inference rules are shown in Figure 2.3 for a subset of the language.

For VAR, if the assumptions TE include the association of identifier *ident* with type τ, then we may infer that *ident* has type τ, i.e. we may "look up" a set of assumptions to find the type for a variable.

For ABS, if an identifier *ident* has type τ', and we can show that expression e has type τ, then the function *ident* $\Rightarrow e$, with formal parameter *ident* and body e, has type $\tau' \rightarrow \tau$, i.e. takes an argument of type σ and returns a result of type τ.

For REC, if an identifier *ident* and an expression e have the same type τ then the recursive function `fix x'` *ident* $\Rightarrow e$, with *ident* as the recursion point in the body e, has type τ.

For [APP], if exp_1 has type $\tau' \rightarrow \tau$, i.e. is a function from a τ' argument to a τ result, and e_2 has type τ, then applying e_1 to e_2 returns a value of type τ.

For [COND], if e_1 is boolean, and e_2 and e_3 have the same type τ, then the condition `if` e_1 `then` e_2 `else` e_3 returns a value of type τ.

[GEN] and [SPEC] are somewhat more involved. Polymorphic functions are characterised by type schemes of the form $(\forall\bar{\alpha}.Type)$ where α is a type variable which may be replaced by an arbitrary type. When a polymorphic function is to be used in a specific context, then it will ultimately have all polymorphic type variables fully instantiated with specific types. Note that different instances of the same polymorphic function will have different types. For example, the `SML` function to find the integer length of a list of arbitrary type:

```
fun length [] = 0 | length (h::t) = 1+length t;
```

[VAR] variables $TE + \{x : \tau\} \vdash x : \tau$

[ABS] abstraction $$\frac{TE + \{x : \sigma\} \vdash e : \tau}{TE \vdash x \Rightarrow e : \sigma \rightarrow \tau}$$

[REC] recursion $$\frac{TE + \{x : \tau\} \vdash e : \tau}{TE \vdash \texttt{fix x'}\ x \Rightarrow e : \tau}$$

[APP] application $$\frac{TE \vdash e_1 : \tau' \rightarrow \tau \quad TE \vdash e_2 : \tau'}{TE \vdash \texttt{(}\ e_1\ e_2\ \texttt{)} : \tau}$$

[COND] condition $$\frac{TE \vdash e_1 : \text{bool} \quad TE \vdash e_2 : \tau \quad TE \vdash e_3 : \tau}{TE \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

[GEN] generalisation $$\frac{TE \vdash e : \tau}{TE \vdash e : (\forall\ \{\bar{\alpha}\}.\tau)\ \text{-}\ \bar{\alpha} \text{ not free in } TE}$$

[SPEC] specialisation $$\frac{TE \vdash e : (\forall\ \alpha.\tau)}{TE \vdash e : [\sigma/\alpha]\tau}$$

Figure 2.3: The inference rules for `Exp`

`fn :` α `list -> int`

will have effective type `int list -> int` when applied to a list of integers and type `(string * real) list -> int` when applied to list of tuples of strings and real numbers.

The [SPEC] rule is used when a new instance of a polymorphic function is to be used. The quantifier $\forall$ is dropped and all instances of the type variable α are replaced with a new type β. The absence of quantification implies that the type after specialisation represents some specific type rather than an arbitrary type.

The [GEN] rule is used to convert a type which can not be instantiated any further into a polymorphic type, by quantifying the corresponding type variable(s). Subsequent instances of that type will be converted by [SPEC].

2.1.1.2 Example of using inference rules

We assume that our initial type environment TE contains the types for all the constant identifiers above, along with unquantified polymorphic types for all the function variables:

$TE = \{$`0:int,` $\cdots\ \}$

Consider the expression

$(x \Rightarrow y \Rightarrow yx)\ 0$

Assume TE_0 `=` $\{$ `0:int` $\}$, let

$TE_1 = TE_0$ `+` $\{$ `x : int` $\}$ and $TE_2 = TE_1$ `+` $\{$ `y: int ->int` $\}$.

We have the inference

$$\cfrac{\cfrac{\cfrac{TE_2 \vdash y\texttt{:int -> int [VAR]} \quad TE_2 \vdash x\texttt{:int [VAR]}}{TE_2 \vdash yx\ \texttt{:int [APP]}}}{TE_1 \vdash y \Rightarrow yx\ \texttt{:(int -> int) -> int [ABS]}}}{TE_0 \vdash x \Rightarrow yx \Rightarrow x : \texttt{int -> ((int -> int) -> int) [ABS]}}$$

$$\frac{TE_0 \vdash x \Rightarrow yx \Rightarrow x : \texttt{int -> ((int -> int) -> int)} \quad TE_0 \vdash 0\texttt{:int [VAR]}}{(x \Rightarrow y \Rightarrow yx)\ 0\texttt{:(int -> int) -> int [APP]}}$$

In type inference algorithms, these steps are carried out automatically by assigning new type variables to a bound variables, and unifying type expressions which should have the same types to find substitutions for the type variables. As we shall see, it is this process that renders automatic type inference unsuitable for type explanation.

2.1.2 The Hindley/Milner inference algorithms

There two principal kinds of Hindley/Milner inference algorithms, the bottom-up $\mathcal{W}$ algorithm, and the top-down $\mathcal{M}$ algorithm.

The $\mathcal{W}$ algorithm for a subset of the small language is shown in Figure 2.4. Here, the $\mathcal{W}$ algorithm unifies types only at an application subexpression and fails only at an application subexpression. It may report the whole application subexpression as an error site, but this large error site is not precise to locate the source of type error. The $\mathcal{M}$ algorithm for a subset of the small language is shown in Figure 2.5. For the subset of the language, the $\mathcal{M}$ algorithm unifies types at a program variable subexpression and a λ abstract subexpression, it does not unify types at application subexpressions. It is proven that the $\mathcal{M}$ algorithm stops earlier than the $\mathcal{W}$ algorithm when there is a type error [29]. It reports a finer grain error site than the $\mathcal{W}$ algorithm, but the type in its report may be harder to understand than that from the $\mathcal{W}$ algorithm.

The unification `UnifyTypes` and substitutions `S`, domain of type environment `Dom` are defined at Appendix A.7. After typing a subexpression, the $\mathcal{W}$ algorithm returns a substitution and a type. The substitution will be used to refine the type environment when the algorithm continues to type other subexpressions. The types in the type environment will become more and more concrete as the inference goes on. But if there is a type error, this brings the left-to-right bias, which assumes that the type information in the type environment is correct.

$\sigma = close(TE, \tau)$ denotes a type schemes by close a type τ under type environment TE, which is defined at Appendix A.8.

2.2 Problem with understanding Hindley-Milner typechecking

Polymorphic typechecking algorithms are well suited to the efficient checking of programs in polymorphic type languages but users find error reporting from such algorithms hard to comprehend.

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

Def $\mathcal{W}$(TE, e) =
    case e of
    $x \Rightarrow$
        if $x \notin Dom(TE)$ then fail
        else let
            $\forall\bar{a}.\tau = TE(x)$
            $\beta_1, \cdots, \beta_n$, be new type variables.
            in
            $(\{\}, \{a_i \mapsto \beta_i\}\tau)$
        endif
    $\lambda\ x\ .\ e_1 \Rightarrow$
        let
            β be a new type variable.
            $(S_1, \tau_1) = \mathcal{W}(TE + \{x \mapsto \beta\}, e_1)$
        in
            $(S_1, S_1\beta \rightarrow \tau_1)$
    $e_1 e_2 \Rightarrow$
        let
            $(S_1, \tau_1) = \mathcal{W}(TE, e_1)$
            $(S_2, \tau_2) = \mathcal{W}(S_1(TE), e_2)$
            β be a new type variable
            $S_3 = UnifyTypes(S_2\tau_1,\ \tau_2 \rightarrow \beta)$
        in
            $(S_3 S_2 S_1, S_3\beta)$
    `let` e_1 `in` $e_2 \Rightarrow$
        let
            $(S_1, \tau_1) = \mathcal{W}(TE, e_1)$
            $(S_2, \tau_2) = \mathcal{W}(S_1(TE) + \{x \mapsto close(S_1(TE), \tau_1)\}, e_2)$
        in
            $(S_2 S_1, \tau_2)$
    endcase

Figure 2.4: Pseudo-code for the $\mathcal{W}$ algorithm

$\mathcal{M}$: TypeEnv $\times$ Expression $\times$ Type $\rightarrow$ Substitution

Def $\mathcal{M}(TE$, e, $\rho) =$
    case e of

    $x \Rightarrow$
        if $x \notin Dom(TE\ )$ then fail
        else let
                $\forall \bar{a}.\tau = TE\ (x)$
                $\beta_1, \cdots, \beta_n$, be new type variables.
        in
                $UnifyTypes(\rho, \{a_i \mapsto \beta_i\}\tau)$
    $\lambda x.e_1 \Rightarrow$
        let
                β_1, β_2 be new type variables.
                $S_1 = UnifyTypes(\rho, \beta_1 \rightarrow \beta_2)$
                $S_2 = \mathcal{M}(S_1\ TE\ {+}x : S_1\beta_1, e, S_1\beta_2)$
        in
                $(S_1 S_1)$
    $e_1 e_2 \Rightarrow$
        let
                β be an new type variables.
                $S_1 = \mathcal{M}(TE\ , e_1, \beta \rightarrow \rho)$
                $S_2 = \mathcal{M}(S_1(TE\ ), e_2, S_1\beta)$
        in
                $(S_2 S_1)$
    let $x = e_1$ in $e_2 \Rightarrow$
        let
                $S_1 = \mathcal{M}(TE\ , e_1, \beta), \beta$ be a new type variable
                $S_2 = \mathcal{M}(S_1(\ TE\ ) + \{x \mapsto Close(TE, S_1\beta)\}, e_2, S_1\rho)$
        in
                $(S_2 S_1)$
    endcase

Figure 2.5: Pseudo-code for the $\mathcal{M}$ algorithm.

2.2.1 Understanding Hindley-Milner type inference

For example, consider checking function:

```
fix sign.  λ n => if (> n) 0 then ‘‘positive’’ else ‘‘non-positive’’
```

knowing that:

`0 : int` i.e. `0` is an integer
`‘‘positive’’ : string`
`‘‘non-positive’’ : string`
`> : int -> int -> bool` i.e. `>` compares two integers to return an integer

The rule for a function:

- assigns the type variable α to `n`
- checks the body i.e. checks the `if` expression:

  - checks the condition i.e. checks the application of `> n` to `0`:

    - checks the application of `>` to `n`:

      - deduces that `>` is `int -> int -> bool` from assumptions
      - assigns the type variable β to the result of the application
      - deduces that n is α from assumptions
      - unifies the anticipated type $\alpha - > \beta$ with the type of `>`, `int -> int -> bool`
      - deduces that α is `int` and β is `int -> bool`

    - assigns the type variable γ to the result of the application
    - deduces that `0` is `int` from assumptions
    - unifies the anticipated type `int ->` γ with the type of `int -> bool`
    - deduces that γ is `bool`

  - unifies the condition type `bool` with the required type for a condition `bool`
  - checks the `then` branch i.e. deduces that `‘‘positive’’` is `string` from assumptions
  - checks the `else` branch i.e. deduces that `‘‘negative’’` is `string` from assumptions
  - unifies the `then` and `else` branches, which must have the same type
  - deduces that the `if` has type `string`

- deduces that the function has type `int -> string`

To summarise, the algorithm:

- is exhaustive, considering all parts of a program;
- is top-down, left to right;
- introduces a type variable for many constructs before instantiating the construct's type.

In contrast, in our experience people appear to be far less rigid and rigorous in their checking of types, as we'll see in Chapter 5. They seem to:

- be partial, assuming a construct's type on the minimum necessary evidence;
- be opportunistic, scanning a program as 2D text for sources of evidence;
- seek concrete evidence (e.g. ground types) before introducing type variables.

We might caricature a typical human check of the above example as:

- `0` is `int` so `n` must be *int*, because arguments of `>` have the same type
- `''positive''` is `string` so the function result must be `string`
- so the function must be `int -> string`

We may note that the human:

- concludes that every case must be `int` because the first case is *int*;
- scans down the columns of patterns as well as across each case;
- focuses on the ground type objects 0 and `''string''`.

2.2.2 Understanding Hindley-Milner Error Reporting

We have observed that the Hindley-Milner algorithm is a poor source of guidance for humans in the presence of errors. For example, consider the following incorrect definition of the Standard ML function `map`. `map` function is supposed to apply a function argument `f` to every element of an argument list, for example, given:

```
fun double x = 2*x
```

then:

```
map double [1,2,3,4] ==> [2,4,6,8]
```

The incorrect definition is:

```
fun map f [] = 0 |
    map f (h::t) = f h::map f t;
```

where:

- `[]` $\longrightarrow$ empty list;
- `(h::t)` $\longrightarrow$ pattern where `h` matches the head of a list argument and `t` matches the tail of that list;

- `f h::map f t` ⟶ apply `f` to `h` and put result on front of list from applying `map` with `f` to `t`.

In SML, all function cases are supposed to return the same type. Here, however, the first case returns the `int 0` but the second case returns a polymorphic list.

Given this definition, the SML of New Jersey system (Version 0.93) reports:

```
example.sml:1.19-2.37 Error: rules don't agree (tycon mismatch)
  expected: ('Z -> 'Y) * 'Z list -> int
  found:    ('Z -> 'Y) * 'Z list -> 'Y list
  rule:
    (f,h :: t) => :: (f h,<exp> <exp> t)
example.sml:1.1-2.37 Error: pattern and expression in
val rec dec don't agree (tycon mismatch)
  pattern:    ('Z -> 'Y) -> 'Z list -> 'Y list
  expression: ('Z -> 'Y) -> 'Z list -> int
  in declaration:
    badmap = (fn arg => (fn <pat> => <exp>))
```

Note:

- the use of the system type variables `'X', 'Y` and `'Z`, which do not appear in the original function;
- the failure to identify explicitly the clash between the types of the first and second cases as the source of the error;
- the error messages refer to the abstract rather than the concrete syntax;
- the same error is reported in two different ways.

There is some work aimed at providing a better understanding of type error and type inference. One approach is to provide better explanation of how the inference leads to a type error. We classify systems using this approach as *explanation systems*, e.g., [4, 15, 55]. Another approach is to provide a better error report, without necessarily explaining the inference that lead to the error. We classify systems using this approach as *error reporting systems*, e.g., [5, 14, 26, 32, 59]. A third approach is to provide a mechanism for the programmer to probe the type of subexpressions [5]. To compare those approaches, a manifesto for good type error reporting is proposed. The manifesto is introduced in the next section.

2.3 A Manifesto for Good Type Error Reporting

In [69], we suggest that good type error reports should have the following properties:

1. Correct. Type errors must be reported exactly when it is not possible to find a legal typing of the program. That is, programs for which a legal typing can be found are never reported as erroneous, and programs for which a legal typing cannot be found are never reported as correct.

2. Accurate. The site(s) included in a type error report should match the portion of the program that contributes to the error. The reported site(s) should be:

   (a) Comprehensive. All program text that contributes to a type error should be reported. The user should be able to understand the error using only the reported site(s). This may require reporting multiple program sites.

   (b) Relevant. As little program text that is irrelevant to a type error should be reported as possible.

3. Intuitive. As far as possible type error explanations should use human-like reasoning. For example an error explanation should avoid using artificially-introduced type variables [66], and should not reproduce large amounts of counter-intuitive mechanical inference.

4. Succinct. Type error reports should maximize useful information and minimize non-useful information. Long and verbose explanations are tedious to read. Short and terse explanation are hard to understand.

5. Source-based. Type error reports should use program text that occurs in the actual source program and should avoid mentioning internal compiler entities such as generated type variables and elaborated core syntax.

2.4 Text based Hindley-Milner type explanation systems

Several type explanation systems have been based on tracing the behaviour of the $\mathcal{W}$ algorithm. Here, we consider the work of Soosaipillai[55], Duggan and Bent[15] and Beaven and Stansifer[4], and Ritri's proposal for an interactive type explanation system[49].

2.4.1 Soosaipillai's system

Soosaipillai's system [55] explains type inference by menu traversal. The system uses a global list to record the subexpressions and their inferred types as a global history list which contains the result types when inference of each sub-expression finishes. This information can be presented as a menu to enable traversal of the global list. An example of type explanation is shown in Figure 2.6.

The system's strengths are that the user can ask `why` to indicate which site needs explanation, an explanation does not instantiate unnecessary local type variables and an explanation is purely in terms of the types of corresponding sub-expressions. However, it is hard to explain types using the method. The explanation is bottom-up so a user needs to remember all the steps of the inference to understand the full explanation of the required type.

2.4.2 Duggan and Bent

Duggan and Bent's system system explains type inference by analysing the instantiation of type variables during unification. A modified unification algorithm is used to record all instantiations of type variables [15]. The approach is similar to that used by Wand[59] to identify the sources of type errors. Other systems use similar unification algorithms to give type inference explanation, for example Rideau and Thery's programming environment[47].

```
>>8+9+7;
8+9+7;
int
>>why;
1. 8+9+7 is int
>>why 1;
2. 8 is int
3. + is int*int->int
4. 9+7 is int
>>why 4;
5. 9 is int
6. + is int*int->int
7. 7 is int
>> why 2;
8. 8 is int(ground value)
>>
```

Figure 2.6: Soosaipillai's type inference explanation

The system uses the essentially unchanged $\mathcal{W}$ algorithm to walk over the abstract syntax tree for a program collecting equality constraints on the types of the program variables. The modified unification algorithm solves the equality constraints and collect the explanations. Like other explanation systems, the system records the program fragment which gave rise to that constraint, with the corresponding instantiation.

Initially a program variable `x` is given type `'A0` for some new non-generic type variable `'A0`. $\mathcal{W}$ collects constraints on `A0'` due to the usage of `x` in the program, and the unification solves these constraints, instantiating `A0'` by substitutions. For example, for the program fragment:

```
fn x => x + 1
```

`x` is initially assumed to have type `A0'`. Type checking the application `+(x, 1)` constrains `(A0' * int)` to be equal to `int * int)`(the domain type of `+`). Unification produces the substitution `[int/'A0]`. The modified unification stores the abstract syntax for `+(x, 1)` as an explanation for why the type of `x` is `int`.

The unification algorithm analyses the changes made to type graphs which represents substitutions during unification. The unification algorithm accumulates explanations in the process of recursively walking over the graph, and when it performs an instantiation at the end of a connection path, it records the accumulated explanations with the program fragment which led to unification being called. Explanations on edges are combined into location information when those edges lead from type variable nodes to type constructor nodes.

Sometimes, it is necessary to reverse the position of the link on the graphs. For example, consider:

```
fun f x y z = (x+y,y+z,x+1)
```

```
The explanation for the type of sum is as follows:
[0] sum : 'A0  by Assumption
[1] sum : int list -> 'A1  because...

'A0 (the type variable for sum) was instantiated to
int list -> 'A1
as a result of type-checking this declaration:
(* sum: 'A0 *)

fun sum (nil) = 0
        | sum (0 :: t: 'A2) = sum t
        | sum (h: 'A1 :: t: 'A3) = Int.+ (h,sum t)

[2] sum : int list -> int  because...

'A1 (the type variable for sum.h) was aliased to
int (the type for 0)
as a result of type-checking this declaration:
(* sum: int list -> 'A1 *)
fun sum (nil) = 0
        | sum (0 :: t: 'A2) = sum t
        | sum (h: 'A1 :: t: 'A3) = Int.+ (h,sum t)
```

Figure 2.7: Duggan and Bent's type inference explanation

The algorithm works from left to right and assumes that `x` is α, `y` is β and `z` id β. Now, from `x+y`, `x` and `y` have the same type, and, from `y+z`, `y` and `z` have the same type. From `x+1`, `x` and `1` have the same type and `1` is `int` so `x` is also `int`. Hence, explaining `z`'s type involves the reverse sequence: from `y+z`, `y` and `z` have the same type; from `x+y`, `x` and `y` have the same type; from `x+1`, `x` is `int`; so `z` is also `int`.

Figure 2.7 shows Duggan and Bent's system explaining the type of the program variable `sum` in our example. Like the $\mathcal{W}$ algorithm, the system first assigns a type variable to `sum`, and then instantiates the type variable to the final type:

The explanation is simpler than that from the direct trace of the the $\mathcal{W}$ algorithm, the technique used by Beaven's [4] discussed below. The explanation is closer to how people understand type inference but is still not very easy to understand. The aliasing generates local type variables which do not relate to the program itself. Furthermore, a type is represented as an annotated graph which quickly becomes lengthy, with similar problems to Soosaipillai's approach.

2.4.3 Beaven and Stansifer's system

Beaven and Stansifer's system [4] is intended for error explanation and illustrates how inference reaches two conflicting types. The system maintains the deductive steps of type inference to explain the reasoning of the type reconstruction process in relation to the program.

The system annotates types of subexpressions to the nodes of syntax tree. To explain the instantiation of a type step by step, a substitution is represented as a list $[(v_n, t_n), \cdots, (v_2, t_2), (v_1, t_1)]$. The system tracks the element (v_i, t_i) to explain the reason of a type evoluted from initial value to its final value.

It uses two function `Why` and `How` to explain how the type of an expression is determined by the type of its subexpressions, and how the types of those subexpressions are inferred. An explanation is generated by traversing an abstract syntax from the root to the leaves, and then back to the root. Thus explanations are essentially $\mathcal{W}$ algorithm traces.

Function `Why` uses the decorated syntax tree and the substitution to explain typing rules as specifically used in the expression of interest. The function `How` explains the circumstances which required the unification. Function `How` first explains the instantiation of types during the typing and map the application subexpressions to the substitutions. An example is shown in Figure 2.8.

The system's strengths are that there are no unnecessary local type variables in the explanations, unlike Duggan and Bent's system, explanations are given in English and the system can explain type errors. Although the explanations are bottom-up, and tend to be based on fine granularity information, a user still needs to remember all the steps of the inference. Furthermore, explanations tend to be lengthy and repetitive.

2.4.4 Rittri's Interactive Type Debugger

Rittri [49] has suggested an interactive type debugger which can be tied to any inference algorithm. This would allow programmers to explore the type of any subexpression and to ask about the origin of a specific constructors.

While the type deriver knows the type of every subexpression and how they were derived, it does not knows the programmer's intentions and mistaken beliefs. Thus, it cannot decide by itself how much to tell and if it tells all it knows at once, the programmer will be overwhelmed. The intention is that the programmer can identify those parts of a complex type that are surprising, and only these need to be explained. However, as with Soosaipillai's system, it is hard to identify a methodology for exploring type decisions. The style of a type explanation is very similar that in Duggan and Bent's system, which includes the instantiation of type variables. To understand such explanations, one needs to understand unification and the type system.

2.4.5 Summary of type inference explanation systems

While explanation systems can be made to be correct and accurate, they fail on several key points.

First of all, the explanations tend to be counterintuitive. They make substantial use of internal type variables as bridges between instances, since these are the sites of refinement through substitution. 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 very hard to remember what entity a type variable represents.

```
1 A type error was detected in the application >>(+(#,3))<<
2 The domain of the function >>+<< is not unifiable
3 with the type of the argument >>((# true), 3)<<.
4 Domain of function is >>(int * int)<<.
5 The argument has type >>(bool * int)<<.
6 **Why does the function >>+<< have type >>((int*int)->int<<?
7   The identifier >>+<< was assigned type >>(int*int)->int<<
8   as part of the initial environment.
9 **Why does the argument >>((# true), 3)<< is determined by
    type of each element.
10 Type of the pair >>((# true),3)<< is determined by type of
   each element.
11 **Why does the first element >>((fn b => if #)<< have type
     >>(bool->bool)<<?
12   The type of an application is the range of the function.
13   The function >>(fn b => if #)<< has type >>(bool->bool)<<
14   **Why does the function >>(fn b => if #)<< have type
       >>(bool->bool)<<?
15     The type of a function definition is determined by the
16     type inferred for the formal parameter >>b<< and by the
17     type of the function body >>if b then b else a<<
18     **Why does formal parameter >>b<< have type >>bool<<?
19       The type of a formal parameter is inferred from its use.
20       The variable >>b<< was initially assigned type variable
         >>'b<<.
21       **How did type variable >>'b<< come to be bound to type
         >>bool<<?
22         This binding arose during analysis of
           >>if b then b else a<<.
23         Since the expression >>b<< must have type >>bool<<
24         and the type variable >>'b<< is its type,
25         then the type variable >>'b<< must stand for >>bool<<.
26         **Why does the expression >>b<< have tyep >>bool<<?
27           [Elided.]
28    **Why does function's body >>if b then b else a<< have
           type >>bool<<?
29      The type of a conditional is determined by the
30      the types of its branches which must unify.
31      **Why does the "then" branch >>b<< have type >>bool<<?
32        [Elided.]
33      **Why does the else branch>>a<< have type >>bool<<?
34        [Elided.]
35  **Why does the second element >>3<< have type >>int<<?
36  All integer constants have type >>int<<.
```

Figure 2.8: Beaven and Stansifer's error explanation for `fn a => ((fn b => if b then b else a) true) + 3`

Furthermore, the explanations are not succinct. The textual explanations have so much information that they become wordy and diffuse. Experts usually find this explanation too detailed to be of real help, though they can find valuable information about the different positions in the programs that contribute to the type given by such systems [47].

Finally, the explanations are not source-based, using text generated from the abstract syntax tree, rather than the original program text. In many language implementations, a program is converted to a simpler "core" representation prior to type checking, which bears little relationship to the original source, as in our running example above.

We think that these difficulties arise directly from the realisation of the Hindley-Milner type inference scheme in algorithms like $\mathcal{W}$ for efficient inference. Attempts to generate human readable type explanations are necessarily constrained by sticking rigidly to traces from such algorithms.

We have been investigating how human experts explain types as a basis for a novel automatic type explanation system. Our experiments are described in Chapter 5, prior to discussion of our "human-like" explanation system.

2.5 Error Reporting Systems

Error reporting systems can improve over the $\mathcal{W}$ and $\mathcal{M}$ algorithms by making explanations comprehensive and more intuitive. There have been several approaches, including Wand's system [59], Johnson and Walz [26], Turner [58], Bernstein and Stark [5], McAdam [32], Dinesh and Tip [14], Choppella [13] and Gomard [17].

2.5.1 Error Reporting Systems

2.5.2 Wand's system

Wand's system [59] is the first system that 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. The idea has great influence to the later systems such as [4, 15, 47]. 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. The algorithm consists of a set of modifications to a structure-sharing unification algorithm [50]. The types are constructed by expanding the initial type variables, and taking notes of the reason(a subexpression) of a deduction. At any node of a type graph, there is a list of applications which together account for the presence of the node. If an inconsistency occurs, then this list of application subexpressions is reported for each of the two inconsistent nodes.

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. Figure 2.9 shows Wand's error reporting for expression `fn n => n * (n=1)`.

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

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))
```

Figure 2.9: Example of Wand's Error Reporting

2.5.3 Johnson and Walz

Johnson and Walz [26] 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.

The approach is used in the implementation of a ML-Oriented Editor. As the user edits and manipulates a program, unification is incrementally applied to determine its type correctness. If a type inconsistence arises, a maximum flow technique is applied to the set of type equations to determine the most likely source of error. In this way a determination can be made as to the relative strengths with which the set of type equation asserts multiple contradictory hypotheses. For example, the set of equations

$$\{\texttt{x = f(y), x = f(y), x = f(y), x = g(z)}\}$$

implies that `x` is predominantly the same type as `f(y)`, while the set of equations

$$\{\texttt{x = g(y), x = g(y), x = g(y), x = f(z)}\}$$

implies that `x` is predominantly the same type as `g(y)`. However, the standard unification algorithm cannot distinguish between these two cases as its rules combine to remove replicated information from its working set of equations. E.g. `if there is an equation t = x and x is a variable, t is a term, replace the equation by x = t`

Johnson and Walz [26] devise new unification rules E.g. `if there is an equation t = x and x is a variable, t is a term and x = t is not present, add the equation x = t`. Using the new rules, the system can decide the weighting of usage. E.g.

$$\texttt{x = f(y), x = f(w), w = g(z)}$$

is a consistent set of equations, any algorithm tolerating type conflicts should give the same final constraint for `x` that the standard algorithm does, namely `f(g(z))`. The devised algorithm gives constrain `f(g(z)[1])[2]`, which means `x` has the same type as `f(g(z))` once, has the same type as the form `f (*)` twice.

The system derives all the implied constraints and their weighting. When there are conflicting types, the system determine the error site according to the weighting. However, if 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.

2.5.4 Turner

Turner [58] considered improved error handling for unbound identifier error reports, source-based error reports, and mutually recursive declaration error reports. Similar to Johnson and Walz's method, his method chooses the minority use as the candidate for the error source. Unlike many implementations of compilers, error reportings are derived from `derived forms` which means error messages generated by the elaborator relate to the constructs of the translated form, not the original source text. This usually is very confusing for the programmer. Turner's implementation produces error messages relating to the original source text, without requiring the elaborator to have any knowledge of derived forms.

The implementation is based on the ML Kit [7]. The elaborator was modified so that it only returns a syntax tree as its result. The nodes of the syntax tree is annotated with error information if there is any. The error information including:

1. Information about the position of the phrase in the source text;
2. Information about whether the phrase is a derived form or not;
3. Error information;
4. Type information for use by the compiler.

The advantage of this scheme is that it is able to calculate the start and the end positions of a phrase, generate the appropriate source information , and pass it to the pretty printer for error reporting.

A special typescheme in the implementation is that all deferred substitution applying to a particular typescheme are collected together as a list of types associated with that typescheme. The resolving of a typescheme is to take its associated list of types and check them to make sure they are consistent. The algorithm is based on voting on the form of a type. E.g. if there are type conflicts, to match the types α, `int`, `int` and `string`, the type chosen should be `int`, it returns the substitution `[int/`α`]`, because the type `int` in a majority of the types in the list. There are some other complicated situations which have not been considered in the implementation.

2.5.5 Bernstein and Stark

Bernstein and Stark give a method of debugging type errors in a so-called *open system* [5]. A new type definition with an `assumption environment` is defined. The type definition admits principal typings and in addition is very closely related to the ML type system. An assumption environment is a finite mapping of variables to sets of simple types(not type schemes we have defined before). Even though the type definition has a polymorphic let construct, it does not use type schemes, only simple types. This is owing to the use of assumption environments. The type definition is essentially the same as that is presented by Zhong Shao and Andrew W. Appel [52]. Bernstein and Stark proved the relation of the type definition under type environment and type definition under assumption environment.

The type inference can give the possible types for an unbound program variable. 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. Furthermore, it is not clear how much of an issue large assumption environments really are in practice.

2.5.6 Bruce J. McAdam

In ongoing work, McAdam uses a graph of types to report errors [32]. 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 is the best way to traverse the graph. McAdam also proposes an interactive type debugging system incorporating some ideas of Turner's [58] and Mike Beaven and Ryan Stansifer [4].

2.5.7 Dinesh and Tip

The approach of Dinesh and Tip [14] 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.

2.5.8 V. Choppella

Venkatesh Choppella [13] reports a framework, based on syntactic and type constraints, for defining program slices that contribute to a given type error or similar syntactic property. The approach is to use a variant unification algorithm [44], `diagnostic unification`.

The algorithm provides information from which it is possible to directly obtain the set of all minimal proofs of the derivation of a given type constraint in a deduction system consisting of type rules and type constraints. A proof is minimal if the result of deleting any step or set of steps is ill-formed.

A type error is signaled by a type constraint that is trivially unsatisfiable. The algorithm can identify the subexpressions of interest. However it is hard to understand and explain the relations between those subexpressions.

2.5.9 Gomard

Gomard [17] identifies the untypable subexpressions using a special type "`untyped`". The system uses two-level syntax and type rules to isolates the untypable parts. However, if an expression has a type error, the entire expression is annotated with untypable, it is not possible to know which parts contribute to the type error.

2.6 Inference algorithm error reporting vs. Manifesto

We classify algorithms like $\mathcal{W}$ and $\mathcal{M}$ which have been designed primarily to analyze programs efficiently as *inference algorithms*. There are similar algorithms devised to improve error reporting [30, 31].

Used in examples, is the $\mathcal{W}$ algorithm implemented in Standard ML of New Jersey, Version 110.0.6, October 31, 1999 [37], and my own implementation of the $\mathcal{M}$ algorithm based on [29].

A suite of 16 type error examples are used as a basis for comparing error reports generated by different algorithms with our manifesto. Not all of these examples are reproduced here for each algorithm, but are available as a technical report [70]. These examples show clearly that inference algorithm error reports are correct. The reports are somewhat terse and abrupt because they do not generally explain how conflicting types were derived, so the reports do not quite qualify as being succinct. Other problems are:

- Inaccurate. $\mathcal{W}$ algorithm error reports are not comprehensive (2a) and can contain much that is irrelevant (2b). $\mathcal{W}$ algorithm reports are not comprehensive because they typically report the second conflicting site (or third if one also counts the variable binding), and do not report either the first or subsequent conflicting sites, as illustrated by the example in Figure 4.1. $\mathcal{W}$ algorithm reports can contain irrelevant text because all of the subexpressions of the failure point are reported, even though only small parts may be actually relevant. Figure 2.10 illustrates this. The right subexpression must be of type `bool` because of the `true` and `false` in the `then` and `else` branches of the `if` expression. The right `false` is not relevant to the error, because its value is thrown away by the function on the right. Nonetheless, the SML/NJ implementation of $\mathcal{W}$ reports the entire right subexpression including the right `false`. Even worse, the relevant occurrences of `true` and `false` are each obscured by uses of `<rule>`.

  $\mathcal{M}$ algorithm reports are typically less irrelevant than $\mathcal{W}$ reports (2a) but this is usually at the expense of being less comprehensive (2b). The error sites identified by the $\mathcal{M}$ algorithm include less irrelevant program text because the algorithm finds errors at smaller expressions, typically constants, variables, or λ-abstractions. $\mathcal{M}$ error reports can still include irrelevant text for the same reason as for $\mathcal{W}$, although generally there will be less. $\mathcal{M}$ error reports are even less comprehensive than $\mathcal{W}$ reports, because $\mathcal{M}$ always fails at an expression inside the one where $\mathcal{W}$ would fail [29]. For example, Figure 2.11 shows how the $\mathcal{M}$ algorithm identifies only the site containing the boolean value and not the lambda binding or the addition sites.

- Counterintuitive. Both the $\mathcal{W}$ and $\mathcal{M}$ 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 2.12 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.

- Not source-based. Implementations of $\mathcal{W}$ and $\mathcal{M}$ almost always report errors using text generated from an internal abstract syntax tree which does not correspond to the source. For example in Figure 2.10, SML/NJ 110 reports the error using a pretty-printed version of its internal abstract syntax tree, which is different from the source syntax. This is more a weakness of the implementations than of the general concepts behind $\mathcal{W}$ and $\mathcal{M}$ — it would not be hard to extend them to work directly on the source syntax.

```
(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)
```

Figure 2.10: $\mathcal{W}$ Algorithm: Inaccurate Error Location

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

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

Figure 2.11: $\mathcal{M}$ Algorithm: Non-comprehensive Error Reporting

2.7 Error Reporting Systems vs. Manifesto

2.7.1 Wand's

Comparing Wand's approach to the manifesto using, *inter alia* Figure 2.9, we find that it is correct(1) and comprehensive(2a), but fails on several points.

- 2(b). Error reports include irrelevant text. Details are reported for some number of function application sites and all subterms are reported, including irrelevant ones.
- 3. Fairly intuitive. A site may contribute to substitutions to several types, there

```
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>))
```

Figure 2.12: $\mathcal{W}$ Algorithm: Counterintuitive Error Reporting

are redundant reports for a single site, for example the site `n=1` is reported three times in Figure 2.9. But it reports the reasons that contribute to the type error.

- 4. Not succinct: reports are a long list of error sites. In the simple example of Figure 2.9, it reports all the application subexpressions, and some of them are redundant.
- 5. Not source-based.

2.7.2 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 cannot be illustrated as we have not implemented it. Compared with the manifesto, the method is correct (1) and succinct(4). It is also comprehensive(2a) in the sense that it locates all the conflicting sites, reporting the minority use as the error.

- 2b. Fairly relevant. It can compare the uses of any operator, and report the operator such as `*` and `=` as the error site.
- 3. Fairly intuitive. 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.
- 4. Succinct. It chooses the most likely error site.

2.7.3 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 cannot be illustrated as we have not implemented it. Compared with the manifesto, the method is correct (1), succinct(4), source-based(5) and comprehensive(2a) in the same sense as Johnson and Walz.

- 2b. Not relevant. It compares the types in patterns and applications, but is less precise than the $\mathcal{M}$ algorithm and Johnson and Walz's.
- 3. Fairly intuitive. It selects the minority usage as the error site.
- 4. Succinct. It chooses the most likely error site.

Table 2.1 summarises the algorithms' performance against our manifesto. It should be viewed together with the preceding text, and entries are not necessarily directly comparable.

2.7.4 Observations of inference algorithm type error reportings

From the study, following observations can be made:

- The error reporting of the $\mathcal{W}$ and $\mathcal{M}$ algorithms is similar, with the $\mathcal{M}$ algorithm having the advantage in recursive definitions and tuple expressions, because it is more relevant. Both suffer from reporting the point at which the conflict is first detected, making them neither comprehensive nor intuitive.

Manifesto	$\mathcal{W}$	$\mathcal{M}$	Wand	Johnson and Walz	Turner
1. Correct	Correct	Correct	Correct	Correct	Correct
2. Accurate					
2(a) Comprehensive	poor	bad	good	good	good
2(b) Relevant	fair	good	fair	fair	poor
3. Intuitive	poor	poor	fair	fair	fair
4. Succinct	terse	terse	verbose	good	good

Table 2.1: Error Reporting Systems vs. Manifesto

- Wand's system is weak because it is not succinct. In the worst case it reports all subexpressions as error sites.
- Turner's implementation is source-based. We consider that this could considerably improve error reporting in production compliers.
- Johnson and Walz'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.

2.8 Conclusion

The most widely used inference algorithm, the $\mathcal{W}$ algorithm, is devised to effectively type check, rather than support human understanding of types. Systems explaining how $\mathcal{W}$ works inevitably tend to be counterintuitive. Furthermore, the explanations are not succinct.

The improved error reporting systems do give better error reporting than that of the $\mathcal{W}$ and $\mathcal{M}$ algorithms. But they usually can not pinpoint the error sites. Sometimes it is hard to reason about the real error sources from the reported likely error sources and their types. The types in the error reporting need succinct explanation to support understanding.

To support better understanding of type inference explanations and type error explanations it is worth exploring new approaches to representing types and to study how people explain type inference and type errors. New inference algorithm based on those study of people's understanding of type inference need to be devised. In the following chapters , a visualization approach and a study of human experts explaining type inference are discussed.

CHAPTER 3
VISUALIZATION

Summary

This chapter explores the use of interactive visualization to improve the understandability of inferred polymorphic types.

Contemporary functional language implementations give little support for understanding polymorphic typechecking and type errors. The aim is to use visualization to support the understanding of type checking in function applications. A novel visualization of functions and their types is presented based on the generation of 2D function specific icons with graphical type representations which change dynamically as functions are applied. This visualization has been implemented for a Standard ML subset within a graphical environment in which expressions are constrained by type matching.

3.1 Problem with understanding Hindley-Milner types

The benefits of static parametric polymorphic typechecking based on the Hindley-Milner scheme [35], in minimizing semantic errors while improving component generality, are well known. Indeed, a central strength of modern functional languages lies in the ease with which they incorporate polymorphic typechecking. However, systems based on the $\mathcal{W}$ algorithm and its variants tend to offer poor support for understanding type checking and type errors. While this efficient algorithm is highly amenable to implementation, it is not immediately apparent that it corresponds well to how people understand the types of programs. Certainly, novices find difficulty with interpreting type error messages which relate to the places where type checking failed rather than the design mistakes and misconceptions that underly the failure. They also have difficulty with understanding the unification of polymorphic types, especially where both types contain type variables.

Our observation of student with previous imperative experience of Pascal or C++ in studying SML suggests that a number of difficulties in understanding polymorphic types may be encountered. For example:

- A more general type may be deduced than was anticipated or an arithmetic overload type error may occur where an imperative language would enforce a coercion. However, the main problem seems to lie in understanding type variables, especially as they behave differently to program variables. For example, consider:

  ```
  fn a => fn a => fn a => a+1
  ```

  Here, each variable a is declared in a new scope, so this curried function has type:

  ```
  fn :  'a -> 'b -> int -> int
  ```

  It may be applied to three different arguments, and effectively discard the first two and to add `1` to the last. In contrast, in the type:

```
fn :  'a -> 'a list -> 'a list
```

for function `last`,

```
fun last v [] = [v] | last v (h::t) = h::last v t
```

all occurrences of the type variable `'a` must be instantiated to the same type.

- Similar types are easily confused. For example, novice are confused by following textual representation:
  `int -> int -> int` and `(int -> int) -> int`;
  `int * int -> int` and `int * (int -> int)`.

- Students also seem to have conceptual problems with tuples. This may be exacerbated by their use as a mechanism for uncurried parameter passing: students may already be familiar with a similar bracketed notation for formal parameters in imperative languages. Nested types for nested tuples and lists of tuples are a further source of difficulty, as are those for nested lists.

- When applying a higher-order function that has more than one argument, novices have difficulties to understand the type consistency requirements between the arguments.

There have been a number of attempts to generate more useful type error information either directly from the $\mathcal{W}$ algorithm or through extensions and modifications to it but ultimately most of these suffer from the same explanatory limitations as the algorithm itself. The inference itself is count-intuitive [66]. In contrast, a number of visual programming systems have used polymorphic type checking to prevent badly typed construct combinations but these tend to lack clear explanations of their type checking regimes. To help novices to understand types a visualization of polymorphic types is proposed and implemented.

The following sections discuss visual and textual representations. Next the polymorphic type checking in visual programming systems is discuses. Finally, the implementation of a visualization of types is introduced. The visualization has been incorporated in a graphical system for an SML [37] subset: its design and use are presented.

3.2 Visual and textual

3.2.1 Graphical syntax

One approach to combine the graphical components in a visual language is to define a graphical syntax.

B. Backlund et al [3] describe several visual language-oriented design environments and conclude that the meta-techniques can be extended to languages having a graphical syntax. Luca Cardelli [12] discussed ideas of programming with boxes and other graphical data structures which are fit for functional languages. In the paper, a graphical syntax for a small functional language was described. These ideas may be the basis for a programming system using a structure editor to manipulate two-dimensional representations of programs. TableTalk [16] is a visual language intended as a database interface and employs visual metaphors in order to express the semantics of a database language. It uses a variety of visual representations to capture the semantics of a functional query language.

3.2.2 Graphic and icon presentations in visual functional programming

There are two approaches to representing the components in a program, one is to translate the a program into visual representation directly, where the combination of program components is directed by the textual language syntax, such as Visual Haskell [46] and U. Schreiweis' [51] Prolog Programming environment . Another approach is using a metaphor to guide the combination of program components, e.g. VisaVis [45], Pygmalion [57] and Clarity [1].

In the first category, some visual programming systems go little further than to translate a standard programming language into a visual representation. A good example is Visual Haskell [46]. Visual Haskell [46] is a attempt to develop a visual equivalent for the Haskell functional language. It is intended both as a program visualization tool and as a programming language in its own right. The approach used a translation from Haskell into an intermediate form VHIF with visual equivalents for each syntactic production of VHIF. The syntax of VHIF is close to Haskell's. It uses visual notations of the "box-and-arrow" style. Visual Haskell is closest to Show-and-Tell [39]. Show-and-Tell is also declarative in nature, and utilizes a polymorphic type system similar to Haskell's. But Visual Haskell is incomplete, lacking syntax for modules, type declarations, type classes and instances, arrays, and user-defined operators. U. Schreiweis [51] introduced an integrated Prolog Programming environment. The approach lays emphasis on the presentation of Prolog programs as graphical module interconnection graphs. The module consists of a set of predicates defining the functionality of the module and an interface. The interface of each module specifies the functionality it provides to other modules and the facilities it needs from other modules. The system provides animation of program execution. The implementation is based on hypertext system KnowS.

In the second category, VisaVis [45] is a functional visual language embedded in the Small-Talk system. The interaction strategy of substitution brings ease of construction to visual programs and integrates higher-order functions smoothly. The concept of supervised substitution prevents syntactical errors and realizes the key/keyhole metaphor. Pygmalion [57] is implemented in Small-Talk 72. Pygmalion could be said to be icon-oriented, and was programmed by direct manipulation actions on icons(programmed by example). David Gooding (since 1990), Jan Townsend-Addis and Tom Addis (since 1986) and Simon Gray (1994-96) at the University of Bath, have worked on Clarity [1], Clarity is a programming tool that enables users to generate programs though diagrams. Clarity has been developed to make computer modeling and simulation more accessible to non-programmers. Faith is the underlying interpreted language and is based upon a typed functional language. The form of Faith is similar to Miranda or ML. Skin [22], assumes a simple object-oriented interface to the underling application and components may flexibly adapt to changes in the application.

The systems in the first category preserve the user's conceptual model, but have poor modularity and rigid control structures. The programming is sometimes extremely complex for larger program. and there is no obvious evidences to support their benefits for programming in general. The systems in the second category supply flexibility in programming, user is guided by the a metaphor to achieve correct operations.

3.3 Explicit Types in Visual Languages

Our motivation for the work discussed here is to explore ways of enhancing novice understanding of parameterized polymorphic type checking.

3.3.1 Types in visual languages

There are a few visual languages and programming environments that have an explicit notion of types. They can be divided into two categories: those whose type system are modeled after traditional procedural languages e.g. Fabrik [25] and DataVis [20] , and those whose type systems are based on the Hindley-Milner approach, e.g. Enhanced Show-and-Tell [39], Cube [40, 41] VisaVis [45] and an extension of Forms/3 [10].

In the first category, the languages supply a set of predefined types such as arrays, records, enumerations, bitmaps, etc., but do not allow the user to define new types. For example in DataVis, types are associated with colors. Only fixed and very small set of types can be described by colors. And although DataVis uses the White color to denote unknown types, it does not deal well with polymorphic operations. Another drawbacks of the type inference system of Fabrik and DataVis is that it does not guarantee type safety: For example, they might try to reference a non-existing array element.

In the second category: Enhanced Show-and-Tell [39] was improved from Show and Tell [53] by adding a static polymorphic type system and higher-order functions using Hindley-Milner type inference. Cube [40, 41] is a successor of Show and Tell and Enhanced Show and Tell, which adds some key ideas from these two languages to visual logic programming. Cube's type system is based on Milner's system as well. VisaVis [45] is a visual language with a data flow based syntax and FFP based semantics, the higher-order version of FP. Its type inference system is based on Wand's type inference algorithm, which is a variation of the Hindley-Milner algorithm. Forms/3 [10] is a form-based visual language. Burnett's type inference system for Forms/3 is based on set constraints; the inference algorithm is also a variation of the Hindley-Milner algorithm. In Kelso's Visual Representation for Functional Programs [28] types are annotated in textual nodes. The languages in this category guarantee type safety, that is a well-typed program will not fail at run time due to a type error. To use those systems, users need to be trained. For example, in CUBE a type definition cube defines the list type. A type definition cube consists of a grey transparent cube with an icon on its top. The color grey distinguishes types from values, which are green. The icon names the type constructor that is to be defined. The cube has a port on its top side, which represents the one formal parameter of the type constructor. The port carries an icon on its outside, which is used to distinguish it from other ports. Inside the cube are two grey transparent boxes, which represent the two constructors of the list type. Each box carries an icon on its top, identifying the constructor.

Figure 3.1 summaries the representations of type systems in visual languages.

3.3.2 Discussion

In principle, a visual programming system may be syntax directed, where construct connection is only allowed if syntax correctness is preserved. This approach is taken in Cardelli's seminal proposal for visual functional programming[12]. However, syntax directed programming imposes an un-natural discipline on programmers. It is more common either to allow free interconnection, through what is essentially ad-hoc polymorphism with

Lang	**Type safety**	**Type Visual Representation**
Fabrik	not	not visible before wiring two object
DataVis	not	not visible before wiring two object
Kelso's	yes	visible as text labeled box
Clover	yes	not visible
Clarity	yes	visible
Enhanced Show-and-Tell	yes	visible type icons, non-unified types are grayed out
Cube	yes	visible type cube
VisaVis	yes	non-unified types raised off the programming surface
Form/3	yes	non-unified types are feedbacked

Table 3.1: The summary of the representation of types in visual languages

en-masse consistency checks on a final program, or to use a type system to dynamically constrain interconnection through incremental type checking.

A number of visual functional programming systems have been constructed, incorporating parameterized polymorphism to constrain interconnection. For example, Poswig et al's VisaVis[45] is a general purpose language where interconnection is based on what they term "less ad-hoc polymorphism". However, types do not appear in program graphs. Braine and Clack's environment for the object oriented functional language Clover[8] is also based on type directed interconnection but no type information is present in expression graphs, when there is no immediate feedback about the type error in a interconnection. Kelso's general purpose system[28] and Addis and Addis' functional schematic programming language Clarity [1] both label expression graphs with polymorphic types which are resolved incrementally during interconnection. Najork and Golin's Enhanced Show-and-Tell[39] is a visual data flow language which incorporates polymorphic type checking. Expression graphs are composed in part from typed icons for data sources.

While type determined component interconnection ensures type consistency, it may be useful to explain why a proposed connection is inappropriate, to clarify misconceptions about component functionality. The above systems appear to offer no support for explaining typed interconnection constraints.

Marian Petre [43] reported some research focusing on understanding how textual and visual representations for software differ in effectiveness. Novice and experts appear to see differently and they have different notational needs. Pure functional languages and their relatives, such as LISP and SML, often have simple grammars. However, the difficulties for novices are the programming concepts embodied in the languages. It is not difficult for a user to write a single function in text. But it is not easy, for example, to handle the applications of higher-order functions etc.

3.4 Visualization of Polymorphic Types

We wished to investigate whether the use of a 2 dimensional visualization might aid in the understanding of polymorphic types, in particular, to form a basis for a dynamic presentation of type unification during function application, to illuminate the simultaneous instantiation of all occurrences of a type variable with a common type.

The system can be divided into two levels, the first is textual function definition, to supply the functions, and check the syntax of input text. The second level is a pure visual programming environment, to interactively construct function applications. Thus, the user has the freedom to construct function and the system can direct the user to construct well-typed structured function composition.

3.4.1 Function oriented icon generation

Here we present a visualization of functions in which domain and range types are identified through graphical representations. Function combinations are constrained by polymorphic type checking and type match failures are identified by clashes between type representations. For well typed combinations, the type representations are changed, in particular to reflect bindings of polymorphic type variables.

Central to visual programming is the use of graphical visualization of program components to ease their identification and understanding of their interconnection. However, the manner and degree to which language constructs are represented graphically have significant impacts on the intelligibility and ease of use of visual programming systems.

At one extreme, all language constructs may be given graphical representations. Cardelli's proposal is based on icons with different shapes and enclosed symbols or labels for different constructs. In Pagan's FP system[42], there are labeled rectangular representations of all constructs. In Reekie's proposed Visual Haskell[46], labeled graphical constructs are are joined together within enclosing boxes by arcs. This approach leads to large numbers of similar representations which require text to distinguish between them and are substantially bulkier than the equivalent program code. Thus, less efficient use is made of display space on a screen, reducing the amount of program that may be viewed at one time.

More fundamentally, there is a danger of a mismatch between a task and a notation. For example, in an early empirical study, Green and Petre[19] found that for conditional logic design, people had more difficulty understanding graphical representations of programs than with the equivalent textual code. They suggest that the structure of the graphics makes it harder to scan a visual program, especially where graphical structures contain "knots" as a result of arc intersections. In a more recent survey of 19 evaluations of visual programming systems, Whitley[60] identified 9 where the visual notation was beneficial and 10 where it appeared to inhibit problem solving.

It was decided to adopt a mixed mode visualization based on the function as the fundamental unit of presentation. Functions are entered as text for display as icons based on an augmented form of the function type discussed above. Interactive graphical techniques are then used to connect actual to formal parameters, to collapse partially and fully applied functions to simpler iconic forms, and to expand collapsed function applications back into the original graphical form.

Fig 3.1 shows the function icon. To the left of the type is a rectangle containing four buttons. The move button is selected to move the icon. The function name button

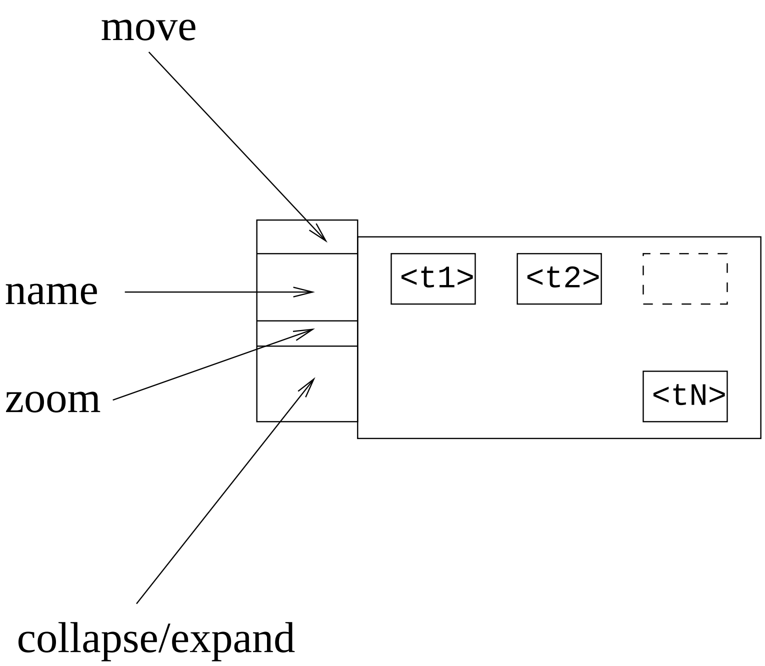

Figure 3.1: Function icon

is selected to identify other functions which may take this function as an argument by highlighting the appropriate argument positions. The zoom button is selected to increase (or decrease) the size of the icon. The collapse/expand button is selected to combine a function application made from linked icons into a single icon or to recreate the linked icons contributing to a collapsed icon. Icons are also generated from values of other types, here augmenting the type representation with a textual reminder of its type.

3.4.2 Visualization of types

In the 1st Year Functional Programming module at Heriot-Watt University, students with 1 term's experience of C++ are taught a pure functional subset of SML including integer, real, boolean, string, tuple and list types. Thus, our visualization was initially aimed at these. It was decided that a type would be represented by a rectangle with internal structure and color corresponding to its component types.

Base types are represented by solid color (Figure 3.2).

Tuple types are represented by juxtaposed rectangles of field types (Figure 3.3 shows a tuple type `int * real * string`).

List types as the component type within a distinctive brace (Figure 3.4 shows `'c list` and `'d list`).

Type variables are represented by an empty rectangle containing the type variable (Figure 3.4).

Function types by a sequence of curried domain types along the top from the left with the range type in the bottom right hand corner (Figure 3.4 shows `map` has a function as its first argument, a list as its second argument, result is a list and function `fn (x:real) => x` of the type `real -> real`).

The visualization clarifies the differences between apparently similar types. For ex-

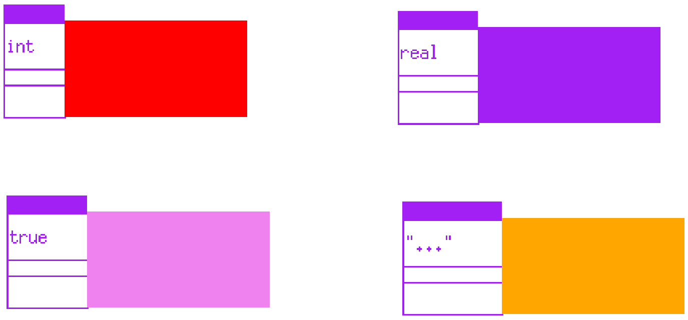

Figure 3.2: The visualization of base types

Figure 3.3: The visualization of tuple type `int * real * string`**.**

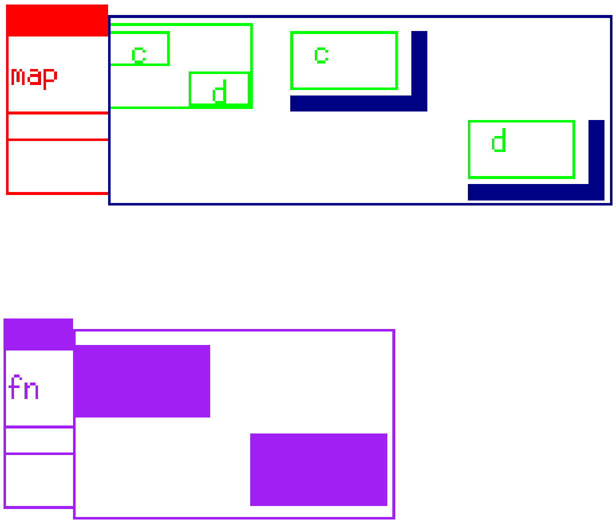

Figure 3.4: The function `map` **and function** `fn (x:real) => x`**.**

ample, Figure 3.5 shows how the visualization may be used to discriminate between the types:
`int -> int -> int` (top left)
`(int -> int) -> int` (top right)
`int * int -> int` (bottom left)
`int * (int -> int)` (bottom right)

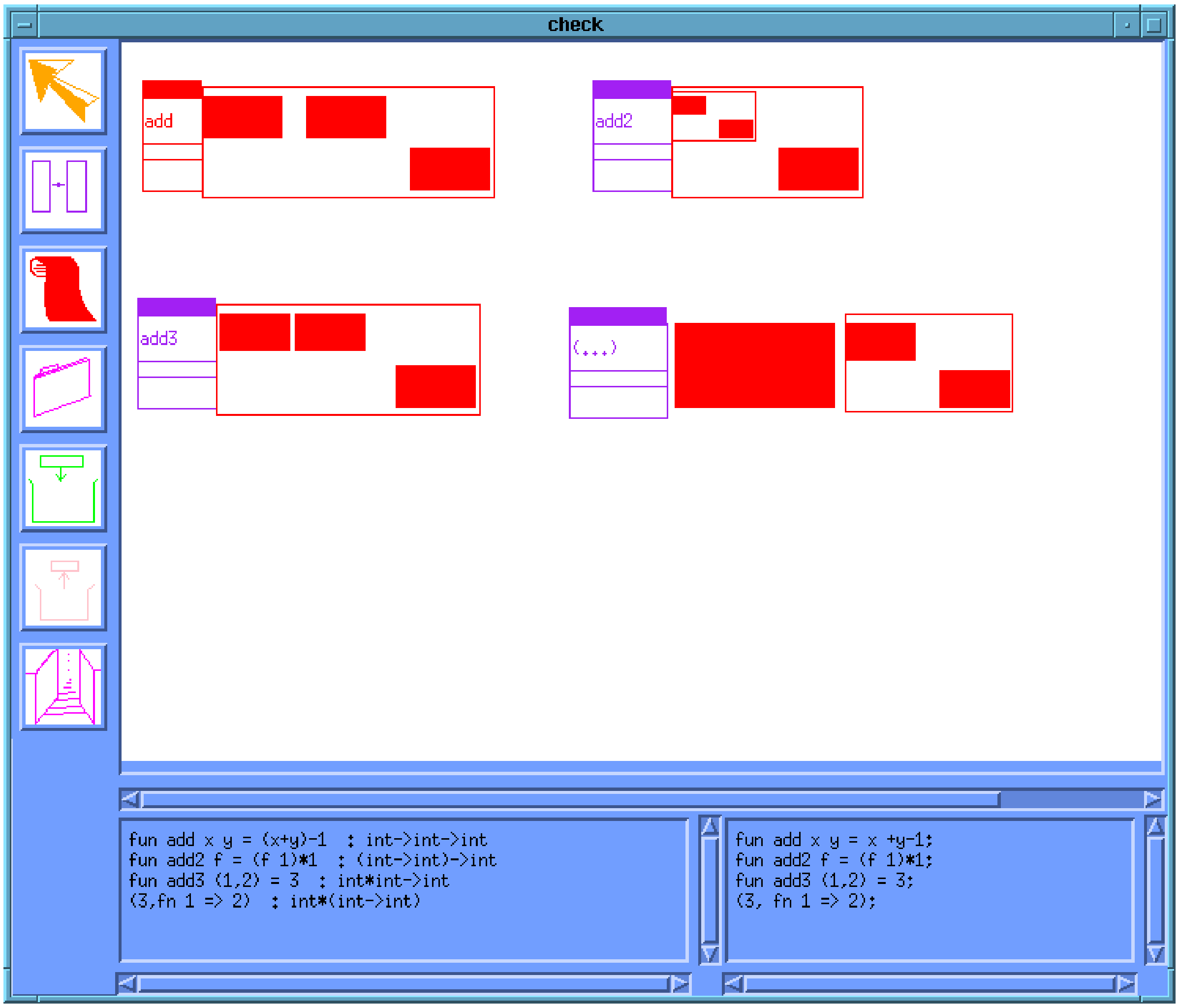

Figure 3.5: Differentiating similar types

3.4.3 Type visualization during function application

The type visualization provides information about which functions may be applied to which arguments. As in a text based system, the types must be compatible but here there is obvious visual evidence for compatibility. The types must either look the same or have the same structure with type variables in one corresponding to types in the other. For example, Figure 3.6 shows the function:

```
fun sumfunc _ [] = 0 |
    sumfunc f (h::t) = f h+sumfunc f t
fn : ('c -> int) -> 'c list -> int
```

which sums the application of function `f` to every element of an arbitrary list. On the left of Figure 3.6 are the function `real :int -> real` which converts an integer to a real and the function:

```
fun sq (x:int) = x*x
```

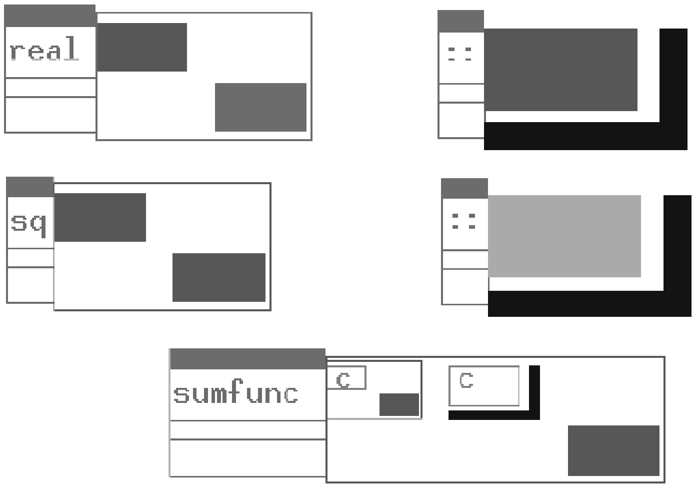

Figure 3.6: Example components

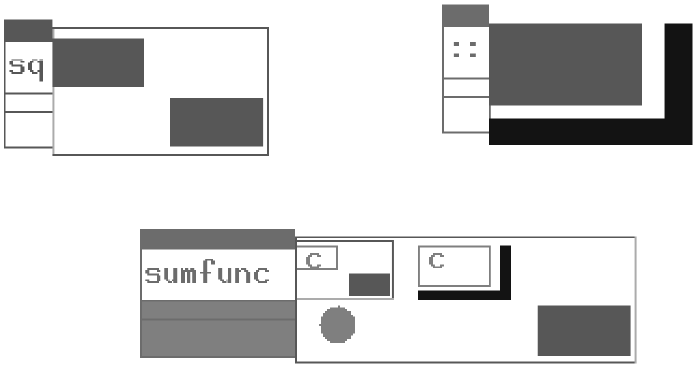

Figure 3.7: Selecting `sq` and highlighting the type consistent argument with circle

```
fn : int -> int
```

which squares its integer argument. On the right are icons for a list of strings and a list of integers.

By inspection, `sq` may be `sumfunc`'s first argument: `'a` will match `int` and `int` will match `int`. The function `real` may not be `sumfunc`'s first argument as the `real` type will not match `int`. Similarly, both lists may be `sumfunc`'s second argument.

The visualization also provides active information about which argument positions an icon may be attached to. For example, selecting `sq`'s name - Figure 3.7 - highlights `sumfunc`'s first argument.

Selecting the highlighted argument - Figure 3.8 - joins `sq`'s icon to that position and also changes the type of `'a` in the rest of `sumfunc`'s icon to reflect its unification with `int`.

Note also that the collapse button has changed to `<`. Selecting this button - Figure 3.9 - collapses the join of `sq` with `sumfunc` to a new composite icon.

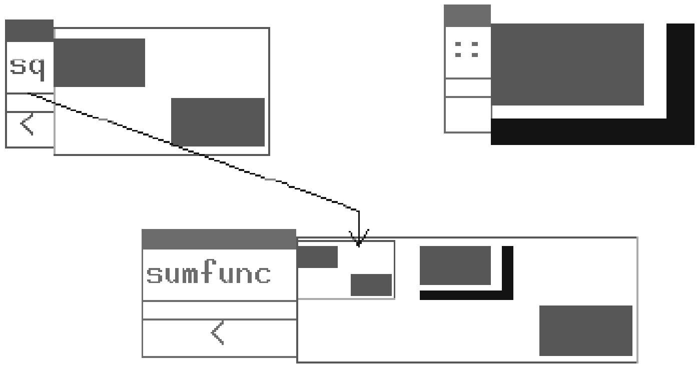

Figure 3.8: Connecting `sq` to `sumfunc`

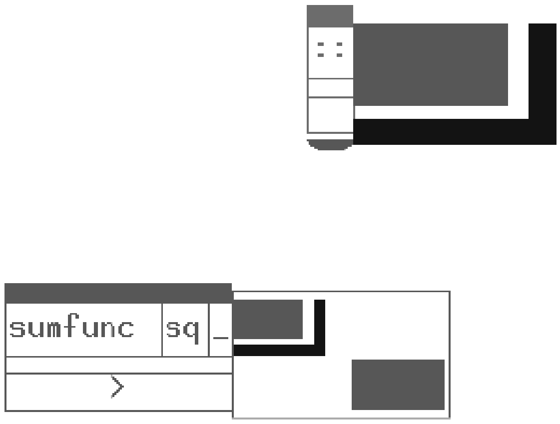

Figure 3.9: Collapsing `sq` joined to `sumfunc`

The icon for the connected `f` argument has been removed and the collapse button has changed to `>` showing that this icon can be expanded back up again.

Selecting the `int list` icon - Figure 3.10 - has highlighted the single argument in the collapsed icon for the application of `sumfunc` to `sq`.

Once again, the icon may be connected to the argument - Figure 3.11.

Finally, the composite icon may be collapsed further to an icon for the final integer - Figure 3.12.

3.4.4 Arbitrary order argument provision

Usually functions must be applied to arguments from left to right. However, in a visual environment it is possible to take advantage of the 2D space to enable arbitrary order argument provision. Consider once again Figure 3.6. If the `int list` is selected then `sumfunc`'s second argument is highlighted - Figure 3.13.

Subsequently, joining the `int list` icon to the second argument position unifies `'a` with `int` in `sumfunc`'s icon - Fig 3.14.

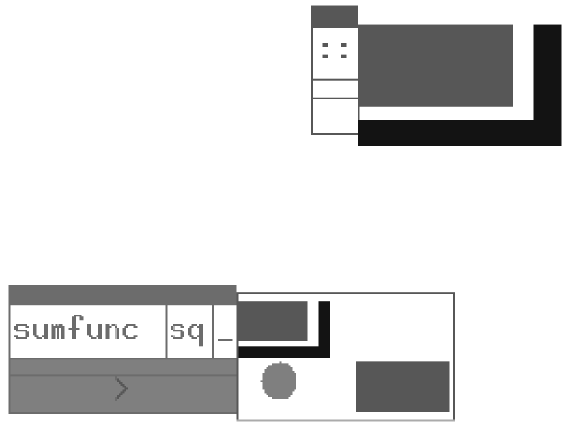

Figure 3.10: Selecting the `int list`

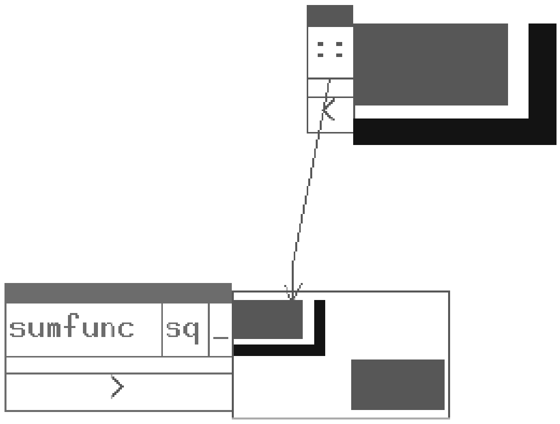

Figure 3.11: Connecting the `int list` to `sumfunc/sq`

Figure 3.12: Collapsing `int list` joined to `sumfunc/sq`

It was thought that this may be useful in explaining the link between different occurrences of the same type variable, as well as more generally for free form visual type exploration.

Text may be generated simply from an arbitrary order argument application. For a nested function with N layers, each with an associated pattern:

```
fun f p1 p2 ... pN = e
```

if the Ith pattern `pI` is applied to some argument `aI` then an equivalent text is:

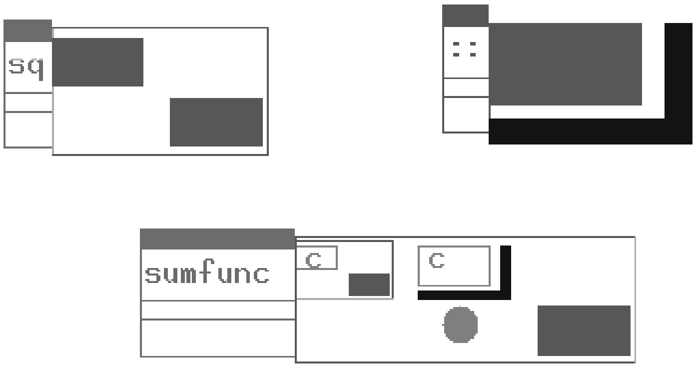

Figure 3.13: Selecting the `int list`

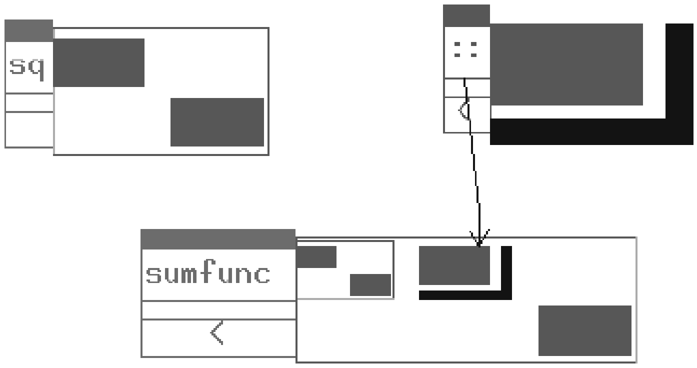

Figure 3.14: Connecting the `int list` to `sumfunc`

```
fun f' p1 p2 ... pI-1 = f p1 p2 ... pI-1 aI
```

3.4.5 Implementation

The visualization comprises a lex analyses, a syntax analyses and a type checker. It is implemented in C language using Motif X library.

3.5 Evaluation of the visualization

It was wished to evaluate the hypothesis that the above visualization would aid in the understanding of polymorphic types and their unification during function application.

3.5.1 Subjects

16 students took part in the evaluation, divided into 2 groups of 8. Our experiment was conducted with 2 groups of 8 subjects taking a 1st year course on Functional Programming in Standard ML at Heriot-Watt University. During the course, students were shown visualization diagrams to augment textual explanation of function types. All subjects would thus already be familiar with textual based type understanding. While this would weight the evaluation against the visualization, we felt it worth pursuing to check that the visualization was at least a plausible alternative to textual type presentation.

3.5.2 Question classifications

It was wished to compare the use of the visualization with the use of text on the same questions. It was also wished to eliminate any learning effect as a result of answering the visualization based questions. The questions are ranked by the number of type variables in each unification which is a good measure of their difficulty. The evaluation is based on a series of questions involving the types arising from function application. For each question, the subject is given the types of a function and its argument, and is asked to select one of four possible resultant types. In addition, they are given the options of the result being some other type or a type error. For an example, see Figure 3.15.

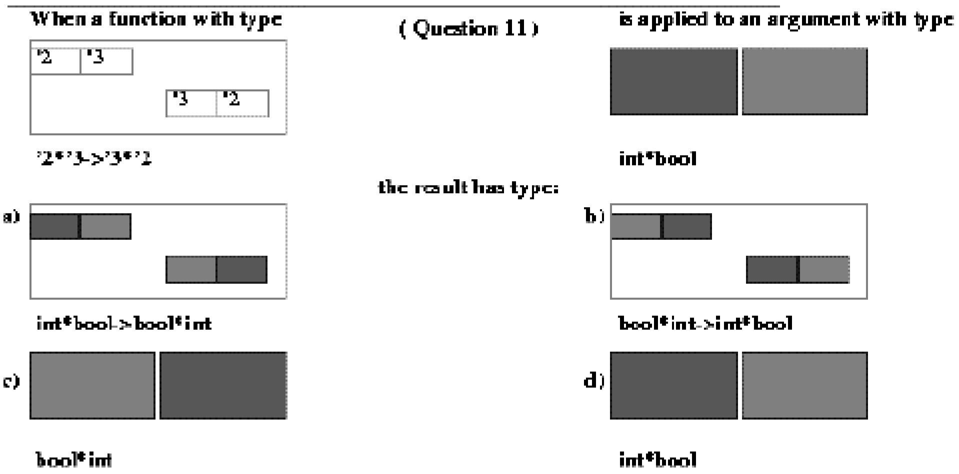

Figure 3.15: Question 11 - visualized

In each set there were 20 questions comprising 6 with no type variables, 3 with 1 type variable, 9 with 2 type variables and 2 with 3 type variables - Set 1. This set was then duplicated, changing the base types and the order of answer options - Set 2. Finally, textual versions of both sets were prepared. For each question there were six optional answers: four different type unifications, none of the above or type error. Note that the visualized questions also contained text.

3.5.3 Evaluation method

It was wished to compare the use of the visualization with that of the normal textual representation in solving type resolution problems. It was elected to make comparison by measuring the speed and accuracy with which students could solve a number of type unification questions.

At the start of the evaluation, all subjects were given a brief presentation on the visualization.

It was wished to see whether there were any transferable learning effects from visualization based problems to textual based problems or vice-versa. For the experiment, we divided the subjects up into two groups. The first group answered a visualized version of Set 1 while the second group answered a textual version of Set 1. Then, the first group answered a textual version of Set 2 while the second group answered a visualized version of Set 2. Finally, both groups answered a short questionnaire on their prior experience with polymorphic typed languages, and their subjective rating of the visualization. The experiment was conducted on paper and subjects were given 10 minutes in which to answer each test set.

3.5.4 Experimental results

For each subject, the number of wrong answers and the number of unanswered questions were recorded. Note that the sample size is really too small to draw other than indicative conclusions.

Figures 3.2 and 3.3 show summaries of raw results for numbers of wrong answers and of unanswered question. They also shows summaries of normalized results where the number of wrong answers is divided by the number of questions answered.

	visual			**text**		
	wrong	unanswered	normalized wrong	wrong	unanswered	normalized wrong
average	5.1	1.9	30.9%	6.4	0.4	32.6%
median	3.0	1.5	15.8%	2.5	0.0	13.2%

Table 3.2: Results - visualization followed by textual questions

	text			**visual**		
	wrong	unanswered	normalized wrong	wrong	unanswered	normalized wrong
average	7.0	0.9	35.9%	7.4	0.8	37.9%
median	4.5	0.0	22.5%	5.5	0.0	27.5%

Table 3.3: Results - textual followed by visualization questions

The group answering visualized followed by textual questions made fewer mistakes but failed to answer more questions on the visualized questions compared with the textual questions. On the contrary, the group answering textual followed by visualized questions made fewer mistakes but failed to answer more questions on the textual questions compared with the visualized questions. All differences were insignificant but might suggest a waning in interest from the first question set to the second set. T-tests showed that there were no significant differences between the groups so the results for all visualized and and all textual questions were combined - Figure 3.4. Overall, on the visualization questions, subjects made slightly fewer mistakes but failed to answer twice as many questions. However, T-tests again show that these differences are not significant.

After each experiment, each subject was asked to rate the visualization for ease of use, speed of use and preference - see Figure 3.5. On average, those who answered the visualized questions first found the visualization comparable in ease of use to and the same speed as the text. However, on average they also thought that the visualization would slightly hinder type error understanding. On average, those who answered textual questions first found the visualization slightly easier to use and slightly faster than the text, and thought that the visualization would make no difference to understanding of type errors.

	visual			**text**		
	wrong	unanswered	normalized wrong	wrong	unanswered	normalized wrong
average	6.3	1.3	34.4%	6.7	0.6	34.2%
median	4.0	0.5	20.4%	4.0	0.0	20.0%

Table 3.4: Combined visualization and textual question results

	visual then text			**text then visual**		
	difficulty	**speed**	**help**	**difficulty**	**speed**	**help**
average	4.1	4.1	4.3	3.4	4.9	3.9
median	4.0	4.0	4.0	3.5	5.3	4.3

Table 3.5: Questionnaire results by visual/textual order: difficulty: 1=easier, 4=same, 7=harder; speed: 1=slower, 4=same, 7=faster; help: 1=helps, 4=neutral, 7=hinders

All answers were then combined - Figure3.6. On average, those who answered the visualized questions first thought that the visualization was slightly easier to use and slightly faster than the text, but would make no difference to understanding type errors. Median scores for all answers suggest that this cohort found no difference between the visualized and textual approaches, and thought that the visualization would neither help nor hinder type understanding. Students also had an opportunity to provide other comments. Some said that they found the visualization confusing. Others indicated that they preferred the visualization. This would suggest that a future evaluation should also consider the subjects' dominant cognitive style on the Verbalizer/Imager dimension [48] as a factor in their relative performances on textual and visual questions. We were retrospectively pleased that a self-identified color blind student said that they found the visualization very helpful.

While one should be cautious in drawing conclusions from such small numbers of subjects, overall there seems to be no disadvantage to using the visualization as opposed to the text. Unfortunately, there is no apparent advantage. However, the subjects had three months experience of practical text based programming and had only seen the visualization in passing in lectures. Thus, we think that further investigation of the practical use of this visualization is worthwhile to see whether or not it can offer a viable alternative to text based type information.

	difficulty	**speed**	**help**
average	3.7	4.6	4.1
median	4.0	4.0	4.0

Table 3.6: Combined questionnaire results: difficulty: 1=easier, 4=same, 7=harder; speed: 1=slower, 4=same, 7=faster; help: 1=helps, 4=neutral, 7=hinders

In fact, our experiment showed the visualization to be no better or worse than the textual approach, despite the subjects' greater experience of textual type understanding. This suggests that further development and evaluation of the visualization might be worthwhile.

3.6 Conclusion

It has shown how a simple visualization of polymorphic types may be incorporated within a visual environment to guide and clarify type compatibility in function application. While the system is intended primarily for investigating type explanation, it would be straightforward to extend it into a full visual programming environment. It might be more fruitful to develop a new visual programming environment incorporating the visualization, using design criteria such as Yang et al's [71], based on Green and Petre's cognitive dimensions for visual programming [18]. It would be interesting to explore the application of such criteria specifically to visual functional programming.

> The empirical results do not show that the visualization is necessarily better than a traditional textual representation. Rather, they show that this way of communicating about types seems to have different strengths than those of the traditional textual representation. The empirical results also suggest that a combination of the visualization with the textual representation might achieve greater understandability than either can alone [1].

Later chapters present our work on textual representation of type inference explanation. The next chapter presents a new type inference algorithm $\mathcal{U}$AE algorithm. The $\mathcal{U}$AE algorithm is the base of our new textual type inference explanation system and type error explanations.

[1] Quotation of the capsule review of [63]

CHAPTER 4
$\mathcal{U}$AE and $\mathcal{IEI}$ ALGORITHMS

Summary

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 $\mathcal{W}$ and $\mathcal{M}$ inference algorithms report the site where a type conflict is detected, but the error message is isolated information: it is not clear what the relationship is between the site where error is reported and the context in which the subexpression was typed. Understanding the types in reports produced by those algorithms tends to require the programmer to understand a chain of inference that may be long and complex. As a result, the error message may give little help to locate the source of the error.

This chapter reports an investigation into better methods of explaining type conflicts. The philosophy of our approach is to find sources of type errors by reporting which parts of the program conflict, rather than isolated error sites. Two new inference algorithms are defined and implemented with this philosophy: the Unification of Assumption Environments ($\mathcal{U}$AE) and the Improved Error Inference ($\mathcal{IEI}$).

4.1 Type error reporting in $\mathcal{W}$ and $\mathcal{M}$ algorithm

Debugging type errors in programs written in polymorphic languages with type inference is hard [6, 31, 49, 59]. In particular, type error messages are often hard to understand. One reason is that much type information is implicit and is inferred by the compiler. Another reason is that the program point where the error is reported may not be the spot that is at fault. This happens because type inference algorithms are used which have been designed mainly to be efficient, and have good error reporting as a subsidiary goal. A type error exists when it is impossible to find a solution to the type constraints imposed by the program. The structure of the inference algorithm determines when it will discover that these constraints are unsolvable.

```
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
```

Figure 4.1: Example of error reporting from the $\mathcal{W}$ algorithm

The example in Figure 4.1 shows how the traditional $\mathcal{W}$ algorithm (as implemented in version 110 of the SML/NJ compiler) finds an error. Inference proceeds left-to-right.

At the site `fn x`, a fresh type variable `'a` which denotes some as-yet-undetermined type is assigned to to `x`. At the first use of `x`, sited at `x 1`, it is inferred that `x` is a function from integer to some unknown type, i.e., the earlier type `'a` for `x` is specialized into the type `int` $\rightarrow$ `'b`. In the second use of `x`, sited at `x true`, the algorithm reuses the type determined earlier for `x`, but then discovers that its argument is of type `bool`. Because `int` (the domain portion of `int` $\rightarrow$ `'b`) can not be unified with `bool`, a conflict is reported. (In contrast, in Figure 4.3, the $\mathcal{U}$AE algorithm does not reuse the type of `x` but instead infers that its second occurrence must be a function from booleans to some unknown type, i.e., `x` must of type `bool` $\rightarrow$ `'c`. The conflict is detected by $\mathcal{U}$AE when the unification of the two types for `x` (`int` $\rightarrow$ `'b` and `bool` $\rightarrow$ `'c`) fails.)

The error reporting from the $\mathcal{M}$ algorithm for the same expression is shown in Figure 4.2. The $\mathcal{M}$ algorithm type checks from left to right, and gives a finer grain error site : the argument of `x`, `true`. But similar to the $\mathcal{W}$ algorithm, it does not mention that another argument of `x` is used as `int` at site `x 1`.

```
fn x => (x 1, x true)

Error in expression:
    true

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

Figure 4.2: Example of error reporting from the $\mathcal{M}$ algorithm

This example shows that many algorithms report type errors at the program location they are processing when they discover the error. The $\mathcal{W}$ error message reports the site `x true`, but does not mention `x 1`, which is one of the sites contributing to the conflict, and may be the real error. Observe also that the conflicting types `int` and `bool` do not appear anywhere in the program text.

In a unification-based type inference system, when unification fails, we have detected a type conflict. Type conflicts are often detected far from the site of the real error, because of the type-checking left-to-right bias. The Unification of Assumption Environments($\mathcal{U}$AE) 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. This idea is due to a chat with ohan Agat and communication with Jörgen Gustavsson [2], but ours is the first implementation of the idea. First, we show how the method works on a simple example.

Figure 4.3 shows the output of our implementation for a simple error. The $\mathcal{U}$AE algorithm explains that `x`, which must have a monomorphic type is used inconsistently in different subexpressions.

Figure 4.4 shows how the $\mathcal{U}$AE algorithm identifies the conflict between a parameter pattern of a function definition and the body of the function.

Instead of using the types inferred for free variables in the leftmost subexpression in the subsequent subexpressions, the $\mathcal{U}$AE algorithm returns both a type and an as-

```
fn x => (x 1, x true)
```

——————— Error ———————

Type conflicts in subexpressions:

```
(x(1), x(true))

(x(1), x(true))
```

the same program variables have type conflicts in different sites
from the first sub-expression
x: int → 'a
but from the second expression
x: bool → 'b

Figure 4.3: Example of error reporting from the $\mathcal{U}$AE algorithm

```
fn (h :: t) => t 5
```

Type conflicts in expressions:

```
fn (h :: t) => t 5

fn (h :: t) => t 5
```

the same program variables have type conflicts in different sites from the first expression: t: 'a list
but from the second expression: t: int → 'c

Figure 4.4: Second Example $\mathcal{U}$AE Error Report

sumption environment for each subexpression. A returned assumption environment gives the local type constraints for all of the free variables that are used in the subexpression. The assumption environments are then unified, detecting type-conflicting uses of the free variables in the different subexpressions. This (mostly) removes the left-to-right bias of the $\mathcal{W}$ and $\mathcal{M}$ algorithms so that more error sites can be found and the error reports can be more comprehensive. [2] The left-to-right bias is not completely removed, because polymorphically typed variables are still treated like in the $\mathcal{W}$ and $\mathcal{M}$ algorithms.

Zhong Shao and Andrew W. Appel [52] presents a type system which does not support principal typings and is still very closely related to the `ML` type system. The type system is used to define the "minimum" interface so that compilation units never need to be recompiled unless their implementation changes. In the type system [52], a typing for an expression is a pair consisting of a type for the expression and a set of `monomorphic`

[2] The left-to-right bias of typing applications and conditional expressions are removed, but there is left-to-right bias in typing local definition `let` e_1 tt in e_2. It assumes that e_1 is type correct when type e_2.

types for each variable that occurs free in the expression. An assumption environment maps program variables to sets of monmorphic types. Later, Karen L. Bernstein and Eugene W. Stark [5] presents the application of the type system for a simple type error debugging and proof of the correctness of the algorithm. For example, consider typing following expression:

```
let fun k a b = a
    val g = k(h 1) (h true)
 in let val f = k(g 1) (g true)
     in (f 1, f true) end
end
```

The error message from the typing looks like:

```
Error: unbound variable: h: int -> int -> int -> 'a
Error: unbound variable: h: int -> int -> bool -> 'b
Error: unbound variable: h: int -> bool -> int -> 'a
Error: unbound variable: h: int -> bool -> bool -> 'b
Error: unbound variable: h: bool -> int -> int -> 'a
Error: unbound variable: h: bool -> int -> bool -> 'b
Error: unbound variable: h: bool -> bool -> int -> 'a
Error: unbound variable: h: bool -> bool -> bool -> 'b
```

This error reporting does not enable an easy understanding of the type error. The assumption environments tend to be too large to be practical for typing complex expressions.

An interactive type exploration environment based on a similar algorithm to $\mathcal{U}$AE has also been developed by Huch, Chitil and Simon[23].

4.2 Definition of $\mathcal{U}$AE algorithm

4.2.1 Expression

The $\mathcal{U}$AE algorithm is defined for the `Exp` language from Chapter 2, adapted to constant(`c`) and overloaded addition operator (`+`). The extended abstract syntax is given in Figure 4.5. `Exp` is based on the λ-calculus extended with pairs, lists, pattern matching parameters, booleans and conditionals, integer and floating point ("real") numbers, overloaded arithmetic operators (only $+$ is handled here), and recursive functions. The language also has let-expressions enabling polymorphic types for the variables they define.

4.2.2 Type, type scheme and type environment

The set Type is in the usual Hindley/Milner style and has numeric, function, list, and product types as depicted in Figure 4.6. Numeric types are named using ν, simple types (which include numeric types) are named using τ, while type schemes (simple types modified by outermost quantification over some or all of the type variables) are named using σ as defined in Figure 4.7. Ordinary type variables are named using α and β. Numeric type variables are named using δ, can only be replaced in substitution by numeric types, and can not be quantified.

The types are defined in Figure 4.6. The type scheme and type environment are described in Figure 4.7. The ConstType function is defined in Appendix A.3.

$x \in$ ExpVar		expression variable
$P \in$ Patt $::=$	x	pattern
$\mid$	(P, P')	list (empty)
$\mid$	$[]$	list
$\mid$	$P :: P'$	
$M \in$ Match $::=$	$P \Rightarrow e$	match
$c \in$ Constant	$=$ Int $\cup$ Real $\cup$ String $\cup$ Bool	int, real, string, bool constants
$e \in$ Exp $::=$	x	variable
$\mid$	$+$	primitive
$\mid$	c	constant
$\mid$	$\texttt{fix}\ x'.\lambda M_1, \ldots, M_n$	lambda abstraction and recursive definition
$\mid$	$e_1 e_2$	application
$\mid$	$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	condition
$\mid$	$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$	local definition
$\mid$	(e_1, e_2)	tuple
$\mid$	$(e_1 :: e_2)$	list
$\mid$	$[]$	empty list

Figure 4.5: `Exp`

The ancillary function SimplifyConstraint is defined in Appendix A.4. The unification functions are defined in Appendix A.5. Constraint sets are finite sets of pairs of types (both simple types and type schemes are allowed) and are written in the form $\{\sigma_1 \doteq \sigma'_1, \cdots, \sigma_n \doteq \sigma'_n\}$. Constraint sets are named using Δ. The definition of substitution is in Appendix A.7. A substitution S *solves* constraint set Δ iff $S(\sigma) = S(\sigma')$ for every constraint $\sigma \doteq \sigma' \in \Delta$. A substitution S is a *most general solution* of constraint set Δ iff iff S solves Δ and, for every other solution S' of Δ, there exists a substitution S'' such that $S' = S''S$ (where writing SS' denotes the usual notion of substitution composition). The function UnifyConstraintSet is defined so that UnifyConstraintSet$(\Delta) = S$ where S is a most general solution for Δ if Δ is solvable and so that UnifyConstraintSet(Δ) fails otherwise. For example, in evaluating

$$\text{UnifyConstraintSet}(\{\forall\{\alpha\}.\alpha \to \alpha \doteq \beta \to \beta\}),$$

the constraint set becomes

$$\{\gamma \to \gamma \doteq \beta \to \beta\}$$

and then becomes

$$\{\beta \doteq \gamma\}$$

which is not unifyable, because of the restrictions on members of QTyVar. More examples are below:

$\gamma \in$ QTyVar	quantified type variables
$\delta \in$ NumTyVar	type variables for number
$\alpha, \beta \in$ TyVar	ordinary type variables

QTyVar, NumTyVar and TyVar are disjoint sets.

$\rho ::= \alpha \mid \gamma \mid \delta$	type variables
$\nu \in$ NumType $::= INT \mid REAL \mid \delta$	number types

$\tau \in$ Type $::= STRING \mid BOOL$	string and bool
$\mid \nu$	number type
$\mid \alpha$	type variable
$\mid \tau \rightarrow \tau$	function type
$\mid \tau\ list$	list type
$\mid \tau * \tau'$	tuple type

$$
\begin{aligned}
\eta \in \text{QType} ::=\ & STRING \mid BOOL \\
& \mid \nu \\
& \mid \alpha \mid \gamma \\
& \mid \eta \rightarrow \eta \\
& \mid \eta\ list \\
& \mid \eta * \eta'
\end{aligned}
$$

Figure 4.6: Types.

Let $\vec{\gamma}$ range over Fin(TyVar).

$\sigma \in$ TypeScheme $::= \tau \mid \forall\vec{\gamma}.\tau$	type scheme
$\sigma_1 = \sigma_2$ if σ_1 and σ_2 are α-convertible	identifying type schemes
$TE \in$ TypeSchemeEnv $=$ ExpVar $\xrightarrow{fin}$ TypeScheme	
	type scheme environment

Figure 4.7: Type scheme, type environment.

$$\text{UnifyEnv}(\{x \mapsto \forall\{\alpha,\beta\}.\alpha \to \beta\}, \{x \mapsto \forall\{\alpha_1,\alpha_2\}.\alpha_1 \to \alpha_2\}) = \emptyset.$$

$$\text{UnifyEnv}(\{x \mapsto \forall\{\alpha,\beta\}.\alpha \to \beta\}, \{x \mapsto \forall\{\gamma\}.\gamma \to \gamma\}) \ \text{ fails.}$$

The Instance function is defined in Appendix A.6.

4.2.3 $\mathcal{U}$AE algorithm

The $\mathcal{U}$AE algorithm is defined in Figure 4.8 and Figure 4.9 and each rule is explained as follows.

1. Typing the empty list returns type $\alpha\ list$, where α is a new type variable. The $\mathcal{U}$AE also returns the type environment TE unchanged.

2. Typing a constant returns the type of the constant and the unchanged TE.

3. Typing a overloading operator returns a new number type variable δ and unchanged TE.

4. Typing a program variable, if the program variable exists in TE, then retrieve its type scheme in TE, instantiate the type scheme to type and return the unchanged TE.
   If the program variable does not exist in TE, its type is assumed as a fresh type variable β, and is stored in result TE.

5. Typing an application, the application and argument subexpressions are typed individually . Assuming that the function part has the type of the argument to a possible result, the type of the application is presented as a new type variable β. Next, the types of a same program variable used in e_1 and e_2 are found, i.e. EnvConstraints(TE_1, TE_2). Check if those types are consistent, i.e. UnifyConstraintSet$(\triangle)$. The result of typing the application is the refined type of β, and a environment which is refined disjoint of TE_1 and TE_2.

6. Typing the local definition, first type the local definition to find the type of x and TE_1. To type the body e_2, a new type environment is formed by disjoining the TE and `closed` type of x and TE_1. In this way, the type system is consistent with the NJSML type system and simplifies the algorithm of [5]. The result type environment is the disjoint of the result environments of typing the local definition e_1 and body e_2 with x removed. But this brings left-to-right bias for typing local definition.

7. Typing a tuple, first type each element subexpression of the tuple. Then check the type consistency of each program variable in each element subexpression.

8. Typing a list constructor, first type the head and tail individually. Next, check the type consistency of program variables in the head and tail.

9. Typing an abstraction, first type the pattern P under empty environment(program variables in the pattern subexpression are the local), then type the body of the abstraction e under the environment that the program variables in the pattern have been removed. Finally, check the type consistency of program variables in P and e by UnifyEnv(TE_1, TE_2).

$$\mathcal{U}_{\text{AE}} : \text{TypeSchemeEnv} \times \text{Exp} \rightarrow \text{Type} \times \text{TypeSchemeEnv}$$

$\mathcal{U}_{\text{AE}}\ (TE, e) =$
case e of
$[] \Rightarrow$ (1)
    $(\alpha\ list, TE)$ where α is a fresh type variable
$c \Rightarrow$ (2)
    (ConstType(c), TE)
$+ \Rightarrow$ (3)
    $(\delta \times \delta \rightarrow \delta, TE)$ where δ is fresh
$x \Rightarrow$ (4)
    if $x \in Dom(TE)$ then
        (Instance($TE(x)$), TE)
    else
        let β be a fresh type variable
        in $(\beta, \{x \mapsto \beta\} \cup TE)$
$e_1 e_2 \Rightarrow$ (5)
    let
        $(\tau_1, TE_1) = \mathcal{U}_{\text{AE}}(TE, e_1)$
        $(\tau_2, TE_2) = \mathcal{U}_{\text{AE}}(TE, e_2)$
        β be a fresh type variable
        $\triangle = \{\tau_1 \doteq \tau_2 \rightarrow \beta\} \cup \text{EnvConstraints}(TE_1, TE_2)$
        $S = \text{UnifyConstraintSet}(\triangle)$
    in $(S(\beta), S(TE_1 \oplus TE_2))$
`let` $x = e_1$ `in` $e_2 \Rightarrow$ (6)
    let
        $(\tau_1, TE_1) = \mathcal{U}_{\text{AE}}(TE, e_1)$
        $TE_{new} = TE \oplus \{x \mapsto Close(TE_1, \tau_1)\}$
        $(\tau_2, TE_2) = \mathcal{U}_{\text{AE}}(TE_{new}, e_2)$
        $S = \text{UnifyEnv}(TE_1, TE_2 \ominus x)$
    in $(S(\tau_2), S(TE_1 \oplus (TE_2 \ominus x)))$
$(e_1, e_2) \Rightarrow$ (7)
    let
        $(\tau_1, TE_1) = \mathcal{U}_{\text{AE}}(TE, e_1)$
        $(\tau_2, TE_2) = \mathcal{U}_{\text{AE}}(TE, e_2)$
        $S = \text{UnifyEnv}(TE_1, TE_2)$
    in $(S(\tau_1 * \tau_2), S(TE_1 \oplus TE_2))$

Figure 4.8: The $\mathcal{U}_{\text{AE}}$ Algorithm (continued in Figure 4.9)

$\mathcal{U}_{\mathbf{AE}}\ (TE, e) =$
case e of
$(e_1 :: e_2) \Rightarrow$ (8)
let
$(\tau_1, TE_1) = \mathcal{U}_{\mathbf{AE}}(TE, e_1)$
$(\tau_2, TE_2) = \mathcal{U}_{\mathbf{AE}}(TE, e_2)$
$\triangle = \{\tau_1\ list \doteq \tau_2\} \cup \text{EnvConstraints}(TE_1, TE_2)$
$S = \text{UnifyConstraintSet}(\triangle)$
in $(S(\tau_1\ list), S(TE_1 \oplus TE_2))$

$P \Rightarrow e \Rightarrow$ (9)
let
$TE' = TE \ominus P$
$(\tau_1, TE_1) = \mathcal{U}_{\mathbf{AE}}(\emptyset, P)$
$(\tau_2, TE_2) = \mathcal{U}_{\mathbf{AE}}(TE', e)$
$S = \text{UnifyEnv}(TE_1, TE_2)$
in $(S(\tau_1 \rightarrow \tau_2), S(TE_2 \ominus P))$

$\mathtt{fix}\ x.\lambda M \Rightarrow$ (10)
let
$TE' = TE \ominus x$
$(\tau, TE'') = \mathcal{U}_{\mathbf{AE}}(TE', M)$
$S = \text{if}\ \ x \notin Dom(TE'')\ \ \text{then}\ \ \emptyset\ \ \text{else}\ \ \text{UnifyTypes}(\tau, TE''(x))$
in $(S(\tau), S(TE \ominus x))$

$\mathtt{fix}\ x.\lambda M_1, \cdots, M_n \Rightarrow$ (11)
let
$(\tau_1, TE_1) = \mathcal{U}_{\mathbf{AE}}(TE, \mathtt{fix}\ x.\lambda M_1)$
$(\tau_n, TE_n) = \mathcal{U}_{\mathbf{AE}}(TE, \mathtt{fix}\ x.\lambda M_2, \cdots, M_n)$
$\triangle = \{\tau_1 \doteq \tau_n\} \cup \text{EnvConstraints}(TE_1, TE_n)$
$S = \text{UnifyConstraintSet}(\triangle)$
in $(S(\tau_n), S(TE_1 \oplus TE_n))$
$\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \Rightarrow$ (12)
let
$(\tau_1, TE_1) = \mathcal{U}_{\mathbf{AE}}(TE, e)$
$(\tau_2, TE_2) = \mathcal{U}_{\mathbf{AE}}(TE, e_1)$
$(\tau_3, TE_3) = \mathcal{U}_{\mathbf{AE}}(TE, e_2)$
$\triangle = \{\tau_1 \doteq BOOL, \tau_2 \doteq \tau_3\} \cup \text{EnvConstraints}(TE_1, TE_2)$
$\cup\ \text{EnvConstraints}(TE_2, TE_3) \cup \text{EnvConstraints}(TE_1, TE_3)$
$S = \text{UnifyConstraintSet}(\triangle)$
in $(S(\tau_2), S(TE_1 \oplus (TE_2 \oplus TE_3)))$

Figure 4.9: The $\mathcal{U}_{\mathbf{AE}}$ Algorithm (continued from Figure 4.8)

10. Typing a recursive function definition, first remove the recursive point x from TE, then type the definition body M under TE. If x appears in the result type environment TE'', check all the type of x are consistent by UnifyTypes$(\tau, TE''(x))$.

11. Typing a recursive function with different cases, type each case and check the types of the all case and the types of the recursive function x in all uses are consistent.

12. Typing a conditional expression, type each branch individually and check program variables are type consistent.

4.3 The error reporting of $\mathcal{U}$AE

Two principles are observed when reporting type errors:
1. Type error messages are reported in 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. An example in Figure 4.10 shows that the $\mathcal{U}$AE reports all the conflicting sites.

Another example in Figure 4.11 shows that the $\mathcal{U}$AE reports one site has type conflict with other sites, those sites are all hightlighted.

In the error reportings, error sites are hightlighted with color in a sub-expression in which the type conflicts are found. Instead of reporting the error sites in the whole expression, which may be very lengthy, it is focused on a sub-expression. More examples can be found in [70] and Appendix E.

4.4 Improved error inference $\mathcal{IEI}$

4.4.1 Introduction of $\mathcal{IEI}$

The second new typing algorithm, incremental error inference, is a modification of the $\mathcal{U}$AE algorithm which reports smaller sites for some errors. The $\mathcal{IEI}$ algorithm invokes the $\mathcal{M}$ algorithm when $\mathcal{U}$AE fails under some circumstances as explained below. We first explain a bit more about the $\mathcal{M}$ algorithm and how it differs from $\mathcal{W}$. 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 [29] and reports a smaller error site. The $\mathcal{M}$ algorithm uses an expected type given in advance that each subexpression must satisfy. For example, in the case of an 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 `'b` $\rightarrow$ `real` for some fresh type variable `'b`, and the required type for e_2 is whatever `'b` was inferred to stand for while analyzing e_1.

Because of the way the $\mathcal{M}$ algorithm carries along a required type, it always reports a smaller error site than the $\mathcal{W}$ algorithm, stopping at a constant, a variable, or a lambda expression. For example, Figure 4.13 shows the error message produced by the $\mathcal{M}$ algorithm for the same program as the $\mathcal{W}$ algorithm in Figure 2.10. The $\mathcal{M}$ algorithm stops at `true`, identifying a smaller error site, but it does not reveal the other sites which are in conflict.

```
fn x => if x then x+1 else x-2.2
```

Type conflicts in expressions:

```
if x then x + 1 else x - 2.2

if x then x + 1 else x - 2.2
```

expression after ‘then’ and expression after ‘else’ have different types
exp after ‘then’ : int
exp after ‘else’ : real
Type conflicts in expressions:

```
if x then x + 1 else x - 2.2

if x then x + 1 else x - 2.2
```

the same program variables have type conflicts in different sites
from the first sub-expression x: bool
but from the later sub-expression(es) x: int
Type conflicts in expressions:

```
if x then x + 1 else x - 2.2

if x then x + 1 else x - 2.2
```

the same program variables have type conflicts in different sites
from the first sub-expression x: bool
but from the later sub-expression(es) x: real
Type conflicts in expressions:

```
if x then x + 1 else x - 2.2

if x then x + 1 else x - 2.2
```

the same program variables have type conflicts in different sites
from the first sub-expression x: int
but from the later sub-expression(es) x: real

Figure 4.10: Explanation of all type conflicts

```
fn x => fn y => (x + y, x -1, y- 2.2)
```

Type conflicts in expressions:

```
(x + y, x - 1, y - 2.2)

(x + y, x - 1, y - 2.2)

(x + y, x - 1, y - 2.2)
```

the same program variables have type conflicts in different sites from the first sub-expression

x: NumStr'd

y: NumStr'd

but from the later sub-expression(es)

x: int

y: real

Figure 4.11: Explanation of type conflicts of one site with other sites

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 finds an error at a function application, without any directly conflicting uses for each program variable in all the subexpressions, $\mathcal{IEI}$ reruns type inference for the subexpressions using the $\mathcal{M}$ algorithm. It uses an expected type for each subexpression that is based on the types independently inferred for the other subexpressions. This finds small sites within subexpressions that conflicts with the type expected for the entire subexpression by the other subexpressions.

For example, Figure 4.12 shows the error reported by $\mathcal{IEI}$ for the same program as in Figures 2.10 and 4.13.[3] The $\mathcal{IEI}$ gives error explanation messages by finding a pair of directly type conflicting sites, and showing the reasons for their conflicts. In the example in Figure 4.12, $\mathcal{IEI}$ 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 the function `(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` are required to have `int` type, because the value of one of them is passed to the function `(fn x ⇒ x + 1)`, but `true` and `false` are of type `bool`. One of the benefits of this approach is that error reports tend to have concrete type information rather than artificially-introduced type variables.

Others have previously suggested using a combination of type-checking algorithms to obtain additional information about type errors. For example, Rideau and Thery [47]

[3]As there are no common program variables that are in type conflict 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 in expression: +
  Error at use of the operator.
  required type: bool * int -> 'a
  operator type: int * int -> int

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

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

Figure 4.12: $\mathcal{IEI}$ Algorithm: Comprehensive Error Location

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

Error in expression: true

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

Figure 4.13: $\mathcal{M}$ Algorithm: Non-comprehensive Error Reporting

combined a variant of the $\mathcal{M}$ algorithm with the $\mathcal{W}$ algorithm. A combination of the $\mathcal{W}$ and $\mathcal{M}$ algorithms is found in Standard ML of New Jersey, Version 110.0.6 [37].

4.4.2 The $\mathcal{IEI}$ algorithm

Def $\mathcal{IEI}$ (TE , e) =
  case e of
    $e_1 e_2 \Rightarrow$ handle fail ($\mathcal{U}$AE , e_1) or e_1 is not a function,
      $\Rightarrow \mathcal{M}_{\mathbf{app}}$ (TE , $e_1, t_{e_2} \to \beta$) to find
      more detailed type error reasons for e_1;
      handle fail ($\mathcal{U}$AE e) $\Rightarrow \mathcal{M}_{\mathbf{app}}$ (TE , e, β)
      handle fail ($\mathcal{M}_{\mathbf{app}}$, e_2) $\Rightarrow$
        reports the type conflicts in e_1 by assuming
        the usage of e_2 is correct ;
        reports the type conflicts in e_2 by assuming
        the usage of e_1 is correct ;
    $e_1(e_2 e_3) \Rightarrow$
      handle fail ($\mathcal{U}$AE e) $\Rightarrow \mathcal{M}_{\mathbf{app}}$ (TE , e)
      handle fail ($\mathcal{M}_{\mathbf{app}}$, e_2) $\Rightarrow \mathcal{M}_{\mathbf{app}}$ (TE , e_1, t_{e_2});
      handle fail ($\mathcal{M}_{\mathbf{app}}$, e_3) $\Rightarrow$ ($\mathcal{M}_{\mathbf{app}}$ (TE , e_2, t_{e_3});
        $\mathcal{M}_{\mathbf{app}}$ (TE , e_1, t'_{e_3}))
  endcase

Figure 4.14: The incremental type error inference $\mathcal{IEI}$ for application.

The $\mathcal{U}$AE algorithm may be improved in the case of applications - Figure 4.15. When the algorithm succeeds at every subexpression(e_1, e_2 and e_3) and there are no common free variables in every subexpression, if the algorithm fails at the top level expression, then $\mathcal{U}$AE is switched to the modified $\mathcal{M}$ algorithm.

The incremental type error inference algorithm $\mathcal{IEI}$ for application subexpression is shown in Figure 4.14. The handle is SML exception handler.

4.5 Comparison of the error reporting of algorithms

Now let us compare the error reporting against the manifesto [69] in Chapter 2.

4.5.1 $\mathcal{U}$AE vs. Manifesto

- 2a. Comprehensive. For many errors, $\mathcal{U}$AE reports all sites that contribute to a type error except for bindings whose only effect is to enforce that a variable has a simple type. It is not completely comprehensive, because like $\mathcal{W}$ and $\mathcal{M}$, if it can successfully type check a polymorphically typed definition, it later assumes the definition is blameless if an error is discovered in its uses.
- 2b. Poor relevance. On small examples $\mathcal{U}$AE's error reporting style seems easier to follow, but its reports for errors in large programs tend to include lots of irrelevant

$\mathcal{U}$AE: TypeEnv $\times$ TypeEnv $\times$ Expression $\rightarrow$ Substitution $\times$ Type $\times$ AssumEnv

$$
\begin{array}{l}
\text{Def } \mathcal{U}\textbf{AE}(TE\ , DT_{NAME_TE}, e) = \\
\quad \text{case e of} \\
\quad e_1 e_2 \Rightarrow \\
\qquad \text{let} \\
\qquad\quad (S_1, \tau_1, A_1) = \mathcal{U}\textbf{AE}(TE\ , DT_{NAME_TE}, e_1) \\
\qquad\quad \beta, \beta_1, \beta_2 \text{ are new type variables} \\
\qquad\quad S_1\text{'} = Unify(\tau_1, \beta_1 \rightarrow \beta_2) \\
\qquad\qquad\quad \text{handle Unify} \Rightarrow let\ TE\ \text{'} = TE \oplus A_1 \\
\qquad\qquad\qquad\qquad\qquad\quad \text{in } \mathcal{M}\textbf{app}\ (TE\ \text{'}, e, \beta) \\
\qquad\quad (S_2, \tau_2, A_2) = \mathcal{U}\textbf{AE}(TE\ , DT_{NAME_TE}, e_2) \\
\qquad\quad S_3 = Unify(S_2\tau_1, \tau_2 \rightarrow \beta) \\
\qquad\qquad\quad \text{handle Unify} \Rightarrow letTE\ \text{'} = TE \oplus A_1 \oplus A_2 \\
\qquad\qquad\qquad\qquad\qquad\quad \text{in } \mathcal{M}\textbf{app}\ (TE\ \text{'}, e, \beta) \\
\qquad\quad S'_{env} = Unify_{Env}(S_1(TE), S_3S_2(TE)) \\
\qquad\quad S_{env} = Unify_{Env}(S'_{env}S_3S_2(A_1), S'_{env}S_3(A_2)) \\
\qquad \text{in } \ (S_{env}S'_{env}S_3S_2S_1, S_{env}S'_{env}S_3\beta, S_{env}S'_{env}S_3(A_1 \oplus A_2)) \\
\quad \text{endcase}
\end{array}
$$

Figure 4.15: Switch to $\mathcal{M}$app in application subexpression.

program text. $\mathcal{U}$AE normally (except with polymorphically typed variables) finds an error when processing the expression which contains *all* of the non-binding sites that contribute to the error. It then reports a conflict between the immediate subexpressions of the expression where it detected the error. These subexpressions in real programs tend to contain lots and lots of irrelevant program text.

- 3. Intuitive. $\mathcal{U}$AE usually reports conflicts between different uses of variables and usually reports all of the conflicting sites. The programmer can then generally understand the type error just from the reported sites and does not need to look elsewhere. This is more intuitive, because the programmer is less likely to need to mentally construct a long chain of inferences to understand the type conflict.
- 4. Terse. Like $\mathcal{M}$ and $\mathcal{W}$, $\mathcal{U}$AE reports the conflicting types but does not explain how it inferred them.
- 5. Not source-based. Both the implementation and the specification are for the internal core syntax used by compilers.

4.5.2 $\mathcal{IEI}$ vs. Manifesto

We compare the $\mathcal{IEI}$ algorithm to the Manifesto. It is correct, but falls short in some respects.

- 2a. Fairly comprehensive. Because $\mathcal{IEI}$ is based on $\mathcal{U}$AE, it tends to be more comprehensive. It is not as comprehensive as $\mathcal{U}$AE because it tends to omit pieces of program text that connect the conflicting sites together. We see that the error reporting from the $\mathcal{W}$ algorithm is too long, from the $\mathcal{M}$ algorithm is too short.

- 2b. Relevant. The $\mathcal{IEI}$ algorithm is relevant in the same way as $\mathcal{W}$. When it is reporting conflicting sites which are not of function type, then every piece of program text reported will be relevant to the error. (At a conflict between function type and ground type, like $\mathcal{M}$ it may report a function in which case part or all of the function body may not be relevant.)

- 3. Somewhat nonintuitive. $\mathcal{IEI}$ takes the expression where $\mathcal{U}$AE found an error and analyzes each of its subexpressions under the assumption that all of the other subexpressions are blameless. This may cause the messages to be somewhat confusing.

- 4. Somewhat verbose and complicated. See Figure 4.12 for an example illustrating this.

- 5. Not source-based. The implementation is partially source-based.

Manifesto	$\mathcal{W}$	$\mathcal{M}$	**Wand**	**Johnson and Walz**	**Turner**	$\mathcal{U}$AE	$\mathcal{IEI}$
2. Accurate							
2(a) Comprehensive	poor	bad	good	good	good	good	good
2(b) Relevant	fair	good	fair	fair	poor	poor	good
3. Intuitive	poor	poor	fair	fair	fair	good	poor
4. Succinct	terse	terse	verbose	good	good	terse	verbose

Table 4.1: Error Reporting Systems vs. Manifesto

The result of comparison of the error reportings from the $\mathcal{W}$ and $\mathcal{M}$ algorithms with that from several improved algorithms (Wand's [59], Johnson and Walz [26], Turner [58], and two our devisings $\mathcal{U}$AE and $\mathcal{IEI}$) is shown in Table 4.1. Source-based is an implementation feature, which considerably improve understanding of the error reporting.

4.5.3 Observations

The following observations can be made from the study.

- 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 does not need to look at other parts of the original expression to reason about type information. This makes the type information easier to understand.

- To identify error sites with greater relevance, the $\mathcal{IEI}$ algorithm uses global inference, often reporting types that come from a long chain of inference without explanation information. In effect, the $\mathcal{IEI}$ algorithm gains greater relevance at the price of intuition.

From our observations with small examples, we have found the error reports from $\mathcal{U}$AE the easiest to understand. However, in large programs it suffers from including irrelevant program text in the error report. Understanding the types in reports produced by other algorithms tends to require the programmer to understand a chain of inference that may be long and complex.

4.6 Conclusions

We have implemented two new type error inference algorithms, both with the philosophy of reporting conflicting sites rather than by an isolated error site. Compared to other type inference explanation methods, our methods automatically locates the sources of type conflicts, and give a pair of direct conflicting error sites rather than individually isolated error sites. The reason for a type error can be found in the relationship of the conflicting uses. This removes most left-to-right type checking bias. The $\mathcal{U}$AE algorithm analyses the uses of common variables in each subexpression, to see if they are consistent in each subexpression. When there are no common variables between subexpressions, $\mathcal{IEI}$ finds the source of conflicts from the context. Both new algorithms can be implemented in interactive programming environments.

We can use the $\mathcal{U}$AE to type unclosed expressions, which have some unbound program variables. The $\mathcal{U}$AE gives the possible types for the unbound program variables by analysing their uses. A user supplied type assertions can be then compare against those infered types and to locate the error sources if there is a type conflict. For example, if we assume that user supplied type assertions are correct, then those sites that type conflicts are found are candidates of the type error sources.

The $\mathcal{U}$AE reports error sites with less precision than the $\mathcal{M}$ algorithm, but with more precise than the $\mathcal{W}$. The $\mathcal{U}$AE error reporting includes some unrelated subexpressions. For example, in Figure 4.16, the reporting includes `x` which does not contribute the type error.An improvement can be devised by a new set of typing rules. The error reporting of expression

```
fn x => if true then x+1 else x-2.2
```

would look like Figure 4.17: because the type error has nothing to do with the lambda-bound variable `x`, but is concerned with `+` and `-` and their other arguments `1` and `2.2`. The error reporting removes the unrelated subexpressions and reports directly conflicting subexpressions.

A direct comparison of type checking by $\mathcal{U}$AE and NJSML-110 is made by running those two and checking their results using a set of examples in Appendix E. Two differences are found:

1. The two systems differ for unclosed expressions, since the $\mathcal{U}$AE give a possible type for the expression, and types for the free variables. Those types are unquantified. NJSML simply reports unbound variables.

2. The two systems differ if there is unresolved overloading For example, consider this expression:

```
fn x => fn y => x + y;
```

```
fn x => if true then x+1 else x-2.2
```
Type conflicts in expressions:

```
if true then x + 1 else x - 2.2


if true then x + 1 else x - 2.2
```

the same program variables have type conflicts in different sites from the first sub-expression x: int
but from the second sub-expression x: real

Figure 4.16: Error reporting with smallest conflicting sites

```
fn x => if true then x+1 else x-2.2
```
Type conflicts in expressions:

```
if true then x + 1 else x - 2.2


if true then x + 1 else x - 2.2
```

Figure 4.17: Error reporting with smallest conflicting sites

SML/NJ decides that this is of type `int -> int -> int`. (It is suggested that it ought to delay the decision until later. Moscow ML delays the decision.) $\mathcal{U}$AE thinks the expression is of type `NumStr'a -> NumStr'a -> NumStr'a`.

Some examples are given in Appendix E to compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}$AE and $\mathcal{IEI}$ algorithms.

The important aspects of type error reports are accurately reporting the program text that contributes to the conflict and giving a good explanation of why the relevant program text causes a conflict. The type information in the error report needs succinct explanations to improve its understandability. To make these explanations better, it is important to understand how humans understand type errors. To overcome the shortcomings of type explanation systems, including 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 [66].

In the next chapter, we will introduce our study of human expert type inference explanation and type error explanation.

CHAPTER 5
HUMAN EXPERT TYPE EXPLANATIONS

Summary

Polymorphic typechecking algorithms are well suited to the efficient checking of programs in polymorphic type languages but users find error reporting from such algorithms hard to comprehend. Here we consider the results of experiments into human checking of both correct and incorrect polymorphic typed programs, and discuss their implications for our proposed new checker.

5.1 Experiment

We are interested in automating the provision of helpful explanations of polymorphic type checking, especially for naive users. To that end we have conducted a small study of human polymorphic type checking, to try to identify "best practice" as the basis for an automated system.

We hypothesized that human type checkers would:

- focus on ground or known types first;
- only partially check functions;
- use vertical relationships between patterns and cases in resolving and assigning types;
- only introduce type variables as a last resort.

We have carried out two experiments where experts were video recorded type checking sets of SML functions, following a "speak-aloud" protocol. The 34 functions in the first set are all type correct. The 32 functions in the second set contain type errors. Subjects were given up to 30 minutes to complete each set.

5.1.1 Subjects classification

9 subjects took part in the type correct experiment and 7 in the type error experiment. The subjects all have:

- at least post-graduate Computer Science experience;
- programmed extensively in polymorphic typed languages;
- implemented Hindley-Milner type checkers or worked extensively with them when implementing functional languages;
- tutored undergraduate students.

and may thus be considered experts rather than beginners.

We have carried out two experiments where experts were video taped type checking sets of SML functions, following a "speak-aloud" protocol.

Function types	**No. of functions**
ground types	1
ground and list types	3
ground, list and tuple types	4
ground types and type variables	1
ground and list types, and type variables	1
ground, list and tuple types, and type variables	2
function argument and ground types	2
function argument, ground and list types	4
function argument, list and tuple types, and type variables	2
function argument, ground, list and tuple types	1
function argument types and type variables	2
function argument and ground types, and type variables	1
function argument and list types, and type variables	4
function argument, ground and list types, and type variables	3
function argument, ground, list and tuple types, and type variables	1
list and tuple types, and type variables	1
type error...	1

Table 5.1: Function types in the experiments

5.1.2 Question classification

We have been unable to locate standard sets of type checking problems either for evaluating automated type checkers or for use with people. The questions in our experiments are drawn from the type checking problems that we set our 1st year Functional Programming students [33], augmented with additional questions. Each question consists of an untyped function definition and the subject is required to identify the type, or explain why there is a type error. Questions are all formed in a pure functional subset of SML, comprising base types, lists and tuples. The questions were chosen to reflect a range of function types. The questions are listed in Appendix G. For the 34 error free questions, the function types are shown in Table 5.1.

5.2 Experiment analysis and results

Initial inspection of the video taped sessions led to identification of 14 major inference techniques used during type checking are shown in Figure 5.1. Each subject's attempt at each question was then analyzed and sequences of the above techniques were recorded. The small number of subjects is not enough to form a basis for statistical analysis. Nonetheless, we are able to identify a number of clear trends by combining technique counts for all subjects. Table 5.2 summarizes the numbers of questions attempted and behavior instances identified.

- **Locate Concrete type evidence**
  GRND: locating ground type;
  OVER: locating overloaded operators;
  CNST: locating other "system" operators of known types e.g. constructors;

- **Using type skeleton**
  USKL: constructing type skeleton for whole function corresponding to the number of arguments;
  LSKL: constructing and refine type skeleton for local construct;
  SKEL: refining function type skeleton;

- **2D inspection of text**
  PATT: locating commonalities across patterns;
  TOPD: analyzing top-down, from nodes to leaves of AST;
  BOTU: analyzing bottom up, from leaves to nodes of AST;
  F: searching forwards and from a known type;
  B: searching backwards from a know type;

- **Using type variable**
  TVAR: using type variable;

- **Argument type and body type**
  BODY: identifying argument type from use in function body;
  ARG: identifying body construct type from known argument type.

Figure 5.1: Human expert techniques used in type checking

	no. subjects	**question attempts**	**attempts /subject**	**tech. count**	**count /attempt**
no error	8	147	18.38	1830	12.45
error	7	235	33.57	1663	7.08

Table 5.2: Total questions attempted and technique counts

Inference Technique	**tech. count**	**count total**
3. local known system operators	317	17.32
13. refine type skeleton	280	15.30
10. identify argument type	161	8.80
12. construct/refine local skeleton	156	8.52
4. construct function skeleton	151	8.25
6. top town	136	7.43
1. locate ground types	123	6.72
5. across patterns	112	6.12
7.bottom up	105	5.74
9. use type variable	100	5.46
8.forward search	82	4.48
2. locate overloaded operators	55	3.01
8. backwards search	36	1.97
11. body from argument use	16	0.87

Table 5.3: Questions with no type errors: technique counts for each technique and as a percentage of all technique uses.

Far more questions were attempted per subject and less techniques were used per question for those with type errors than for those lacking errors. Naively, this implies that the subjects found identifying type errors easier than inferring types, but we would need to assume that both tasks require equal effort. However, as we shall see, subjects are slightly more likely to take a more exhaustive approach to type inference for error free questions. Furthermore, it may be that locating dissonance is intrinsically easier than verifying consistency, but we cannot substantiate this speculation without further experimentation.

5.2.1 Error free question set

Table 5.3 shows the use of techniques in the error free questions, in descending order of technique counts:

Considerable use is made of type signature skeletons, which we did not anticipate. Typically, a subject starts with a minimal outline signature(8.25%), with a slot for each bound variable, and for the result. They then fill in the slots to finer degrees of detail as checking progresses(15.30%), identifying argument types(8.80%) and introducing sub-skeletons for structured bound variables(8.52%), totaling 40.87% of technique use. For example, consider inferring the type of:

```
fun count0 [] = 0 |
    count0 (0::t) = 1+count0 t |
    count0 (h::t) = count0 t
```

which counts how often `0` appears in a list of integers:

Inference Technique	tech. count	count total
3. local known system operators	372	22.37
13. refine type skeleton	206	12.03
1. locate ground types	200	11.73
5. across patterns	195	10.94
8.forward search	144	7.58
4. construct function skeleton	126	6.86
2. locate overloaded operators	114	6.01
12. construct/refine local skeleton	100	6.01
6. top town	87	5.05
8. backwards search	84	3.85
7.bottom up	64	2.77
10. identify argument type	46	2.59
11. body from argument use	43	1.86
9. use type variable	31	0.36

Table 5.4: Questions with type errors: technique counts for each technique and as a percentage of all technique uses.

1. initial skeleton: `->`
2. identify `[]` as empty `list:  list ->`
3. identify `0` as integer: `list -> int`
4. identify `(0::t)` as integer list: `int list -> int`

As hypothesized, much use is made of concrete type information. Type constructors(17.32%) often appear in patterns in our problem set and will be located early on in a left to right, or left top to left bottom, scan. Ground types(6.72%) are used less than constructors, which we did not anticipate. There is little use of overloaded operators(3.01%), reflecting their general absence from our question set.

Concrete type technique use totals 27.05%.

2D inspection of function text, comprising top down(7.43%), across pattern(6.12%), bottom up(5.74%), forward search(4.48%) and backwards search(1.97%), represents 25.74% of technique use confirming our hypothesis.

Low use is made of type variables(5.46%), as we hypothesized.

Contrary to our expectations, full checking was carried out for 114 questions(77.55%) and partial checking for 33(22.45%).

5.2.2 Error question set

The questions with type errors used in the experiment are listed in Appendix 4. Table 5.4 shows the use of techniques in the error questions, in descending order of technique counts:

Error label	**Error classification**
E-OVER	Unresolved overloading
E-GRTC	Ground type conflicts
E-CNST	Constructor type conflicts
E-LHRH	Inconsistencies between left and right hand sides of definitions
E-SWAP	Swapped bound variables
E-CASE	Inconsistencies between cases
E-UNIV	Non-universally quantified type variable

Table 5.5: Type error classification

Error class	**1**	**2**	**3**	**4**	**Total%**
E-OVER	CNST 19.1	OVER 14.0	SKEL 11.0	PATT 10.3	54.4
E-GRTC	CNST 27.8	F 9.7	USKL 9.4	SKEL 8.1	55.0
E-CASE	GRND 15.6	CNST 14.9	PATT 14.9	USKL 10.5	55.9
E-LHRH	CNST 20.8	USKL 14.6	GRND 12.3	PATT 12.2	59.9
E-SWAP	CNST 22.6	USKL 14.7	PATT 11.7	GRND 11.3	60.3
E-UNIV	GRND 23.7	PATT 15.8	F 15.8	CNST 7.9	63.2
E-CNST	CNST 38.1	F 12.4	PATT 10.5	USKL 9.5	70.5

Table 5.6: Top 4 most frequent used techniques for each error classification and percentage of all technique uses

For the type error questions, less use was made of type signature skeletons than for the error free problems, through the introduction of an initial outline(6.85%), subsequent refinement(12.02%), local skeletons(6.01%) and argument identification(2.59%), comprising 27.47% of technique use.

However, more use was made of concrete type information, through the identification of known operators(22.3%),ground types (11.73%) and overloaded operators(6.01%), comprising 39.34% of technique use.

2D inspection of text, involving across pattern(10.94%), forward search(7.58%), top down(5.05%), backwards search(3.85%) and bottom up(2.77%), comprised 30.19% of technique use. Note that cross pattern comparison was used far more than for the error free problems.

Type variable introduction(0.36%) was the least used technique, and used far less than for the error free questions.

Full checking was carried out for 145 questions(61.70%) and partial checking for 90(38.28%) questions. There was less use of full checking than for error free questions.

5.2.3 Relating error classifications and techniques

The type errors in the questions can be classified as Table 5.5.

Figure 5.6 shows the 4 most frequently used techniques, which account for between 54.4of all techniques for the indicated error class.

We find that different techniques, and sequences of techniques are used to explain different classes of type error. In particular, our experiment shows the following patterns, where the figures are the number of times a technique is used as a percentage of the total number of techniques used for questions in this error class.

Locating type of constructor(CNST) is most used in the explanation of E-CNST(38.1%). This reflects that structures connected by a constructor are treated as a unit.

Locating ground type(GRND) is most used in E-CASE(15.6%). This reflects people tend to compare the differences in the use of ground types in the cases of function definitions first.

The use of type variables(TVAR) is minimal in all classes(0.3% - 5.3%). It is least in E-GRTC(0.3%). This reflects people seeking concrete evidence before using type variables.

Techniques for locating concrete type evidence (GRND+OVER+CNST) are used considerably in all error classes(31.6%-46.7%). Using concrete type first can simplify the process of inference, since concrete types need no further refinement.

The total percentage of uses of type skeletons (USKL+ LSKL + SKEL) from (21.7% - 27.3%) is similar in all cases, that is most used in E-OVER(27.3%) and E-LHRH(27.1%), least in E-UNIV(21.7%).

Constructing a type skeleton(USKL) is most used in E-LHRH(14.6%), E-SWAP(14.7%). This reflects the fact that type skeletons can help to compare and record the current type and history type in those cases. It is used equally in E-GRTC(9.4%), E-CNST(9.5%), and E-OVER(9.9%), but is least used in E-UNIV(7.9%).

The greatest use of refining type skeleton(SKEL) is 40.3% in overloading errors(E-OVER). This reflects that people tend evaluate the type skeleton during resolving the overloading. Its use is similar in E-CASE(34%) and E-GRTC(33.5%), and in E-LHRH(24.7%), E-UNIV(24.4%) and E-SWAP(23%), and is least used in E-CNST(20%). This reflects that people depend less on elaboration of type skeletons in explanation of E-CNST than in explanation of other errors.

Locating commonalities across patterns(PATT) is most used in E-UNIV(15%), and E-CASE(14.9%), used equally in E-LHRH(12.2%), E-SWAP(11.7%) and E-CNST(10.5%); but is least used in E-GRTC(5.9%). This reflects that least effect is required to compare patterns in explanation of ground type errors.

Locating overloading operators(OVER) is most used in E-OVER(14.0%), and least used in E-CNST(0.0%), and E-UNIV(0.0%).

Analyzing top-down and bottom up(BOTU+BOTU) is most used in E-GRTC(13.7%) and least used E-UNIV(5.3%).

Searching forwards and backwards(F+B) is most used in E-UNIV(18.4%), and used equally in cases: E-CNST(13.4%), E-GRTC(12.5%), E-LHRH(11.1%), E-SWAP(10%), E-OVER(9.9%). This reflects that people tend to search forwards and backwards to find and compare the uses of non-universally quantified variable.

5.3 Discussion and conclusions

We have observed a number of important differences between human type checking, as characterised by our composite expert subject group, and the $\mathcal{W}$ algorithm. First of all, people find the elaboration of a skeletal type during checking extremely helpful in structuring the process. Furthermore, as we hypothesized, people rely on identification of constructs of known type to guide checking and make use of the 2D textual form to locate them. Finally, as we hypothesized, people use explicit type variables as a last resort. However, our prediction of people performing partial checking was confounded. We used two problem sets in the experiments. We tried to use type correct problem set to find the type inference techniques that people use, use type incorrect problem set to indify the techniques that people use to locate the source of type errors. We found that people use the same set of techniques in solving those two set problems. We have not found any comparable research specifically on human type checking. However, there are interesting correspondences with work on program comprehension. In particular, the observed use of type skeletons to guide type checking may be an instance of Brooks' top-down, hypothesis driven comprehension [9]. Similarly, the observed use of concrete type features is similar to Wiedenbeck and Scholtz's identification of the use of surface beacons to guide successful program comprehension [61]. Wiedenbeck and Scholz note that beacon use is particularly helpful for comprehension of programs with unknown purpose: our questions are uncommented and provide no semantic information through meaningful identifiers.

In the next chapter, we will present a type explanation system for our SML subset which will incorporate heuristics based on these conclusions. We hope that its explanations will be more useful, especially to naive users. The system will seek to provide succinct explanations by minimizing the explanation length, using explanation pruning heuristics.

CHAPTER 6
A HUMAN-LIKE TYPE EXPLANATION SYSTEM

Summary

A type explanation system motivated by explanations given by expert humans rather than by Hindley-Milner inference algorithms in introduced in this chapter. In particular, the system seeks to provide succinct explanations with minimal reference to artefacts of mechanised type inference. The system does not use type variables in the explanation if the type is not polymorphic.

6.1 Introduction

A prototype "human-like" type explanation system has been developed, which has been strongly informed by the experiments in the previous Chapter. Our objectives have been to:

- provide succinct explanations;
- base explanations on concrete type information as far as possible;
- use two dimensional (2D) examination of functions, down all patterns and all results for all cases, as well as across the pattern and result for an individaul case;
- minimise repetition in explanations.

The approach differs from the human inference techniques, discussed in the previous Chapter, in not giving exhaustive explanations based on a whole function and in not explicitly using a skeleton to present the explanation. The resulting explanations, as discussed below, are very different to those derived directly from the $\mathcal{W}$ algorithm and equivalents, but feel similar to those we observed in human type inference.

Our system produces type explanations in two stages. First of all, a modified type inference algorithm is applied to a function to construct an annotated type tree, which records a history of the inference of each node. The annotated type tree is then traversed to generate an explanation, using rules to ensure succinct, non-repetitive explanations. These stages are now discussed.

6.2 Type Checking for Human-Like Explanations

Our system provides explanations for a subset of SML including: `int`, `real`, `bool`, `string` and corresponding operators; tuples; lists; conditions; local definitions; recursive functions; patterns.

Given a set of functions whose types are to be explained, each is parsed and an abstract syntax tree (AST) is constructed. The AST is then type checked using the Unification of Assumptions Environment algorithm ($\mathcal{U}$AE) discussed in Chapter 4.

During the application of the $\mathcal{U}$AE to an AST, an annotated type is generated for each abstract syntax construct. The annotation indicates which rule was used to form it

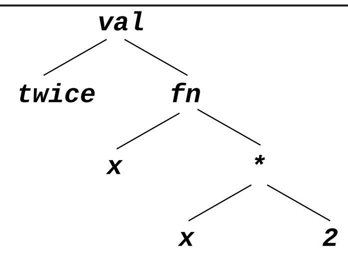

Figure 6.1: AST for `val twice = fn x => ((* x) 2)`

Rule annotation	Details
APP	function application
CONS	list concatenation
FN-L	function left hand side
FN-R	function right hand side
OP-*op*	expression with system operator *op*
OP-L	system operator left hand side
OP-R	system operator right hand side
VAL	`val` definition
VAL-L	`val` left hand side
VAL-R	`val` right hand side

Figure 6.2: Examples of rule annotations in type nodes.

and the piece of AST it corresponds to. The annotation also lists the assumption environments that contributed to the original type's inference. These assumption enviroments in turn contain annotated types with references to further assumption environments. Thus, the cyclic graph of assumption environments records a type inference history which is used subsequently to generate explanations. When a program variable is reached during AST traversal, it is augmented with an assumption environment associating it with a new type variable, annotated with that program variable. When a value of known type is reached, that type is noted, annotated with that value.

Once all branches of an abstract syntax construct have been visited, the corresponding node's type is found through application of the Hindley-Milner rules above to unify the child nodes' types. The node's final assumption environment is then formed by combining and unifying its and the child nodes' assumption environments. When an operator of known type is reached, an operator-specific rule is applied to unify the operand types and assumption environments.

Consider example `val twice = fn x => ((* x) 2)` with the AST shown in Figure 6.1. The AST is descended to build an equivalent type tree that lists for each AST node the rule that applies, an initial type variable or known type, and the corresponding sub-tree.

In an annotated type tree, each node is annotated as:

rule:*type* - *AST node*
assumption environment

Here, *rule* indicates which $\mathcal{U}$AE rule was invoked. Those required for our examples

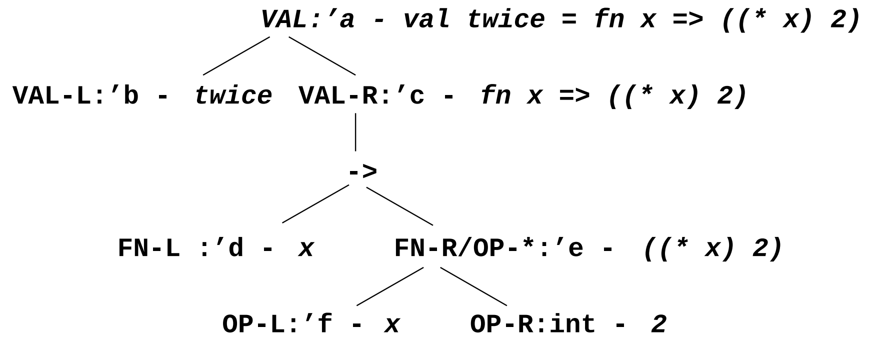

Figure 6.3: Initial annotated type tree for `val twice = fn x => ((* x) 2)`

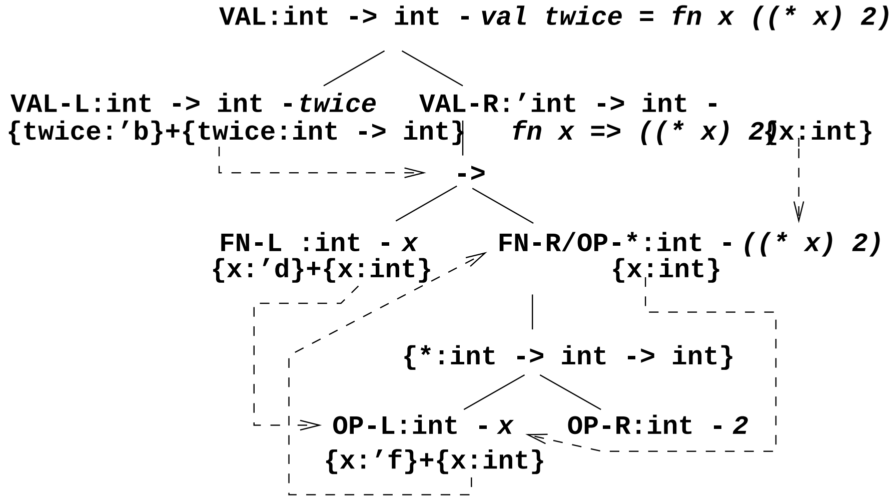

Figure 6.4: Complete annotated type tree for `val twice = fn x => ((* x) 2)`

are shown in Figure 6.2, the full rule annotations in type nodes are shown in Appendix B. *type* indicates the node's type. *AST node* is a textual representation of the node in the AST from which the type tree node was derived. *assumption environment* is the node's assumption environment if any. Figure 6.3 shows the type tree for the above example after AST descent.

The type tree is then ascended, constructing the full type for each node. The node's type variable is instantiated using an appropriate rule to combine the types from sub-nodes. The assumption environments for the contributing nodes are also recorded. Figure 6.4 shows the type tree for the example after type construction. Note that, for brevity, the initial assumption environment is not shown on each node.

We treat recursive functions defined using the SML `fun` form as a special case, to facilitate 2-dimensional analysis. For a recursive function, a matrix is constructed with a row for each function definition case, and a column for each curried argument and the result. During tree descent, each matrix entry is annotated as for type tree nodes. During

	left hand side pattern	right hand side expression
case 1	**CONS:'a list** ***[]***	**int** ***0***
case 2	**CONS: int list** ***(5::t)*** **{t:int list}**	**OP-+:int** ***1+count5 t*** **{t:'c, count5:'c -> int}**
case 3	**CONS: 'b list** ***(h::t)*** **{h:'b,t:'b list}**	**APP: 'e** ***count5 t*** **{t:'d, count5:'d -> 'e}**

Figure 6.5: Initial annotated type matrix for `count5`.

tree ascent, the types down each column are unified and the assumption environments across each row are unified and combined.

For the function to count how often an integer list contains a `5`:

```
fun count5 [] = 0 |
    count5 (5::t) = 1+count5 t |
    count5 (h::t) = count5 t
```

the initial matrix has the form shown in Figure 6.5. After type resolution the matrix has the form shown in Figure 6.6. Type annotation links in a matrix element are omitted when they originate from a subtree in that element.

6.3 Generating Explanations for Human-Like Type Checking

To generate an explanation from an annotated type tree, the tree is traversed and the node's annotated assumption environment, recording the history of the inference of that node's type, is inspected. However, as all contributing assumption environments are recorded for each node, there may be considerable repetition of information and circular chains of annotations. Thus, our system first attempts to remove redundant information, and to build a reduced tree where each node has one, non-circular explanation.

If a program variable appears in multiple assumption environments then all occurrences are checked for consistency and the shortest chain of annotations leading to a fully instantiated type is selected. Furthermore, if a chain of annotations for a program variable includes a cycle back to that variable, then the cycle is eliminated by removing the highest level link in the annotated type tree.

For example, in Figure 6.4, there is a cycle linking `x:int` in `((* x) 2)` through the node for `x` back to the node for `((* x) 2)`. We remove the link from `((* x) 2)` to `x`. The reduced annotation tree is shown in Figure 6.7.

The final explanations are structured by the top level function type and are generated systematically from left to right, from argument types to result types. Alternatives

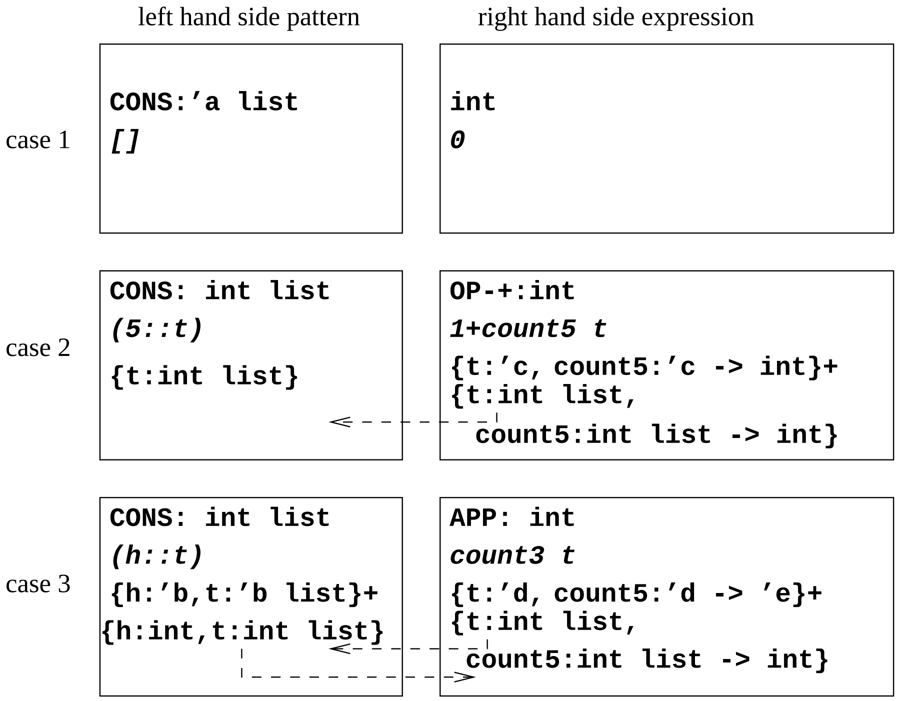

Figure 6.6: Complete annotated type matrix for `count5`.

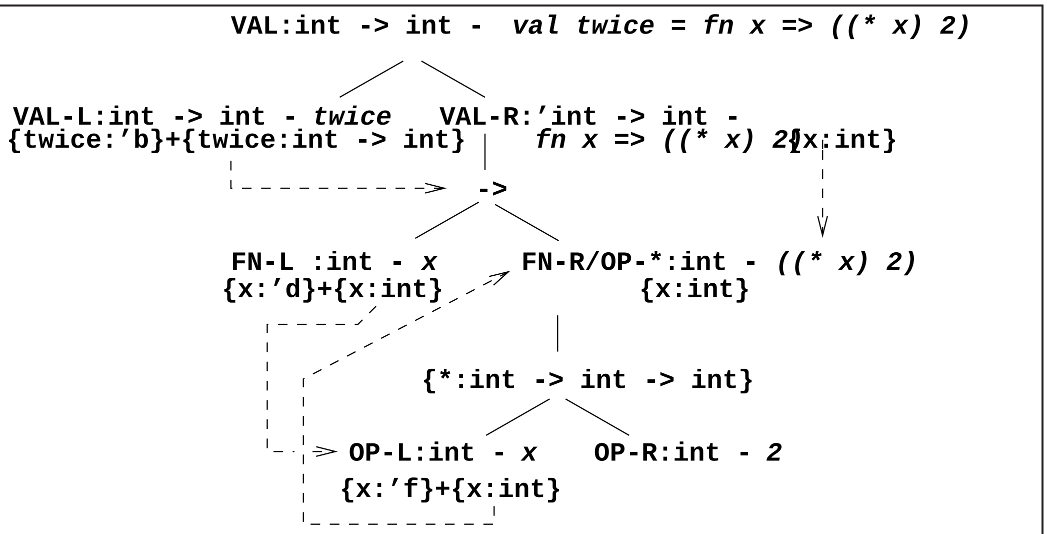

Figure 6.7: Reduced annotated type tree for `val twice = fn x => ((* x) 2)`

would be to start with a simplest type or to allow the user to select top-level types of interest. The gross structure of an explanation is determined by the rule annotation on a node, with the fine detail provided by the history chains of assumption environments. During explanation generation, if a sub-explanation has already been generated then a reference is made back to that sub-explanation.

For our SML subset, there are 40 rules for explanation corresponding to 40 distinctive sites of type inference which are classified in Appendix B. Here we discuss the final explanation for `val twice = fn x => x*2` which is:

```
"twice" and "fn x => x * 2" have the same type
 - LHS and RHS of "val twice = fn x => x * 2"
1: "fn x => x * 2", is a function type: int -> int
Argument 1: int
2:  the Bound variable "x" and "2" have the same type in "x * 2"
    - arguments/result of "*" have the same Number types
3: "2": int
Result: int
4:  Function body "x * 2": Int
    - result/arguments of "*" have the same Number types
    - see 2 above
```

Initially, it is observed that both sides of a `val` definition have the same type and the right hand side is chosen to give the explanation. At step 1, the final type for the right hand side function `fn x => x*2` is given. The argument is explained by exploring the use of the bound variable `x` in the body `x*2`. At step 2, it is observed that operands `x` and `2` of `*` have the same numeric type. Step 3 notes that `1` is an `int`. Hence, the argument `x` is `int`. Step 4 refers back to step 2 to justify the observation that the function body `x*2` has type `int`.

For example, the full explanation for the function that counts `5`s in a list is:

```
"fun count5 [] = 0
  |  count5 (5 :: t) = 1 + (count5 t)
  |  count5 (h :: t3) = count5 t3"
      is a function type: int list -> int
Argument 1: int list
1: "[]", "(5 :: t)" and "(h :: t3)" have the same type
   - same LHS pattern position
2: "[5]", "t" and "(5 :: t)" have same type
    -operands/result of "::"
     3: "5":  int
4:  "[5]": int list
Result: int
5: "0", "1 + (count5 t)" and "count5 t3" have the same type
   - same RHS result
6: "0": int
```

Note that the identifier `t` in the third definition case is numbered to distinguish it from that in the second case.

Step 1 notes that the left hand side patterns all have the same type. At step 2 it is found that the second case `(5::t)` has the shortest history and it is observed that a list containing `5`, the tail `t` and the whole list must have the same type. At step 3 it is found that `5` has the shortest history and that it is `int`. Thus, step 4 decides that the whole list must be `int list`, and hence the function argument must be `int list`. Step 5 notes that the right hand side result expressions must all have the same type. At step 6 it is found that `0` has the shortest history and that it is `int`. Thus it is concluded that the function result is `int`.

For more examples of explanations from the $\mathcal{H}$ algorithm, see Appendix F.

6.4 $\mathcal{H}$ algorithm

As we saw in Chapter 4, the $\mathcal{U}$**AE** algorithm gathers the types of the same program variables from different AST leaves and syntheses them at the root of those leaves where it checks the type consistency of each program variable. At this point, the $\mathcal{U}$**AE** can determine the types of a program variable in different subexpressions. If the type information contains inference explanation information, the $\mathcal{U}$**AE** can determine which explanation sequence is shortest and most concrete (i.e. less polymorphic, e.g. type $\alpha \rightarrow \beta$ is more concrete than type γ). In case that it is not possible to decide which is more concrete, $\mathcal{U}$**AE** picks the one with less number of explanations. The $\mathcal{U}$**AE** algorithm types patterns clause by clause. As we saw in Chapter 5 people can type patterns in 2D, i.e. people can type patterns clause by clause or column by column. To enable 2D explanation, the $\mathcal{U}$**AE** needed to be extended.

6.4.1 2D typing

To demonstrate 2D typing, the small language defined in Chapter 4 is extended with patterned cases in Figure 6.8.

Let $(\tau_{mn}, TE_{mn}) = \mathcal{U}\mathbf{AE}(TE, Patt_{mn})$,
$\mathcal{U}$**AE** types the $Patt$s to form a matrix of (τ, TE)s:

$$\left| \begin{array}{cccc} (\tau_{11}, TE_{11}) & \ldots & (\tau_{1n}, TE_{1n}) & (\tau_{1(n+1)}, TE_{1(n+1)}) \\ (\tau_{21}, TE_{21}) & \ldots & (\tau_{2n}, TE_{2n}) & (\tau_{2(n+1)}, TE_{2(n+1)}) \\ \ldots\ldots & & & \\ (\tau_{m1}, TE_{m1}) & \ldots & (\tau_{mn}, TE_{mn}) & (\tau_{m(n+1)}, TE_{m(n+1)}) \end{array} \right|$$

The last column of the previous matrix comes from typing the function body expressions

$$\mathtt{Exp}_1 \ldots \mathtt{Exp}_m.$$

We annotate this typing as:

$$\sum (\mathcal{U}\mathbf{AE}\ Pattern)$$

The function UnifyConstraintSet and EnvConstraints are defined in Appendix A.5. The types of a program variable in a row are unified; the result of the unification is S_{Env}.

All patterns and expressions in a column have the same type. In Figure 6.10, the types in one column are unified and the result of the unification is the substitution S_{column}. We annotate this unification process as:

$x \in$ ExpVar		expression variable
$P \in$ Patt $::=$	x	pattern
$\mid$	(P, P')	tuple
$\mid$	$[]$	list (empty)
$\mid$	$P :: P'$	list
$M \in$ Match $::=$	$P \Rightarrow e$	match
$c \in$ Constant	$=$ Int $\cup$ Real $\cup$ String $\cup$ Bool	int, real, string, bool constants
$e \in$ Exp $::=$	x	variable
$\mid$	$+$	primitive
$\mid$	c	constant
$\mid$	`fix` $x'.\lambda M_1, \ldots, M_n$	lambda abstraction and recursive definition
$\mid$	$e_1 e_2$	application
$\mid$	`if` e `then` e_1 `else` e_2	condition
$\mid$	`let` $x = e_1$ `in` e_2	local definition
$\mid$	(e_1, e_2)	tuple
$\mid$	$(e_1 :: e_2)$	list
$\mid$	$[]$	empty list
$\mid$	`fun` *Patterns*	recursive function

$$
\begin{aligned}
Patterns\ ::= &\ x\ \mathrm{Patt}_{11} \ldots \mathrm{Patt}_{1n} = \mathrm{Exp}_1 \\
\mid &\ x\ \mathrm{Patt}_{21} \ldots \mathrm{Patt}_{2n} = \mathrm{Exp}_2 \\
\mid &\ \ldots\ldots \\
\mid &\ x\ \mathrm{Patt}_{m1} \ldots \mathrm{Patt}_{mn} = \mathrm{Exp}_m
\end{aligned}
$$

Figure 6.8: Extend `Exp` with patterns.

$$
\begin{aligned}
&\mathrm{UnifyEnv}(\mathrm{TE}_1, \mathrm{TE}_2, \ldots, \mathrm{TE}_n) = \\
&\qquad \mathrm{UnifyConstraintSet}(\mathrm{EnvConstraints}(TE_1, TE_2) \ldots \cup \\
&\qquad \mathrm{EnvConstraints}(TE_1, TE_n) \ldots \mathrm{EnvConstraints}(TE_{n-1}, TE_n)) \\
&S_{Env1} = \mathrm{UnifyEnv}(TE_{11}, \ldots, TE_{1n}, TE_{1(n+1)} \\
&\ldots\ldots \\
&S_{Envm} = \mathrm{UnifyEnv}(TE_{m1}, \ldots, TE_{mn}, TE_{m(n+1)} \\
&S_{Env} = S_{Env1} \cup \ldots \cup S_{Envm}
\end{aligned}
$$

Figure 6.9: Unification of the assumption environments in one clause of definition.

$$\sum \text{(Unify } Env_{column})$$

$S_{t1} = \text{UnifyTypes}(\tau_{11}, \tau_{21}, \ldots, \tau_{m1})$
$S_{t2} = \text{UnifyTypes}(\tau_{12}, \tau_{22}, \ldots, \tau_{m2})$
......
$S_{tn} = \text{UnifyTypes}(\tau_{1n}, \tau_{2n}, \ldots, \tau_{mn})$
$S_{column} = S_{t1} \cup \ldots \cup S_{tm}$

Figure 6.10: Unification of the types in a column of definition.

A type skeleton $type_{sk}$ of a recursive definition is give in Figure 6.11.

```
type of the first argument pattern →
type of the second argument pattern →
...
type of the last argument →
type of the right-hand side pattern
```

Figure 6.11: A type skeleton

This type skeleton is refined by substitution:

$$S_{Env} \cup S_{column}.$$

The system types each subexpression individually. This gives the primary features for the typing rules :

1. The left-to-right bias is avoided [69]. If there is a type conflict between subexpressions, the typing treats the subexpressions equally.

2. It can find all the possible explanations for the type of a lambda bound program variable when the program variable is used in different subexpressions.

3. When a lambda bound program variable is used in different subexpressions, it can compare the simplicity of the explanations of its types in those subexpressions, and choose the most concise explanation. When typing a recursive definition, it compares the explanations of each type in one column, and selects the most concise explanation for the column.

4. It can give the type of any subexpression in an expression.

5. It types the patterns.

6.4.2 Annotating explanation in type tree

When people give a type inference explanation for a subexpression, they may say "this subexpression is used in the `if` branch, so its type is `bool`". Such people actually give some type rule information in their explanations.

The type explanations are given by the type relationships of subexpressions. A relationship constitutes of two fields: a subexpression and the type rule for typing the subexpression. To record the relationships, the type is extended with additional information.

In the type structure, we can see a type `Annotate (Type * ( e, RuleName))` which is the type annotated with a subexpression `e` which is the source of the type is issued, and the `ruleName` is the type annotation in the typing rules, which will be used to translate the explanation into English.

English explanations are given by the type information. The types are annotated with explanation information shown in Figure 6.12.

`RuleName` is the annotation information in types. The full classification is listed in Appendix B.
`InOperand` represents the last operand in a application;
`IfBranch` represents the `if` branch;
`ThenBranch` represents the `then` branch;
`ElseBranch` represents the `Else` branch;
`BaseType` represents the subexpression has a ground type;
`TypeOf` represents the subexpression has the type been explained;
`LHS` represents the type of left-hand side pattern;
`RHS` represents the type of right-hand side pattern;
`Element` represents an element of a tuple;
`FNLeft` represents the left-hand side pattern of an abstraction;
`FNRight` represents the right-hand side pattern of an abstraction;

For example, annotation:
`TypeOf` is translated as `the type of` ;
`BaseType` is translated as `has the same type as`;
`InOperand` is translated as `has the same type of the last operand in`.

The `ruleName` is added to the type of a subexpression during type inference. E.g. to type `fun fac n = if n = 0 then 1 else n * (fac(n-1))`: type from left to right; at the top of AST type `FUN (fac, n, ...`; follow the typing rule of `FUN`; the top-level type of the expression is annotated with the `FUN`; type the subexpression `fac`; assumes its type is α; annotate the type with `TypeOf`, i.e. `Annotate(`α`, (fac, TypeOf))`. This type can be translated as English text: `type of "fac" is` α.

Other explanation systems like Duggan's [15], do not have the `ruleName` in the type information. Since those systems only use the modified unification algorithm to record the subexpressions the inference algrithm went through, they do not record the relation of those subexpressions.

$\gamma \in$ QTyVar	quantified type variables
$\delta \in$ NumTyVar	type variables for number
$\alpha, \beta \in$ TyVar	ordinary type variables

QTyVar, NumTyVar and TyVar are disjoint sets.

$\rho ::= \alpha \mid \gamma \mid \delta$	type variables
$\nu \in$ NumType $::= INT \mid REAL \mid \delta$	number types
$\tau \in$ Type $::= STRING \mid BOOL$	string or bool
$\mid \nu$	number type
$\mid \alpha$	type variable
$\mid \tau \to \tau$	function type
$\mid \tau\ list$	list type
$\mid \tau * \tau'$	tuple type
| Annotate(Type * (Exp * RuleName))	type with type relationship

$\eta \in$ QType $::= STRING \mid BOOL$
$\mid \nu$
$\mid \alpha \mid \gamma$
$\mid \eta \to \eta$
$\mid \eta\ list$
$\mid \eta * \eta'$

RuleName ::= APP
| IfBranch
| ThenBranch
| ElseBrach
| BaseType
| TypeOf
| InOperand
| FNLeft
| FNRight
| Element
| LHS
| RHS

Figure 6.12: Extension of `Type` tree.

6.4.3 The $\mathcal{H}$ algorithm

The $\mathcal{H}$ algorithm is defined in Figure 6.14, Figure 6.15 and Figure 6.13. The unification algorithm UnifyEnv and substitution are the same as that of the $\mathcal{U}$AE algorithm. $\mathcal{H}$ is a modified $\mathcal{U}$AE algorithm. It types an expression and returns the type of the expression and a type environment. It uses a modified unification algorithm to accumulate the explanations of a type before return a substitution. For details of the unification, see [15].

1. Typing a empty list, the result type α *list* is annotated with $NILLIST$, to mean the type can be further instantiated to any list type.

2. The type of a constant is annotated with `BaseType` and the constant expression, to explain why there is a ground type.

3. Typing a number operator, the type of the operator is annotated with `NumOp` and the operator to explain the relation of the two arguments and result subexpressions.

4. Typing a program variable x, if x is in TE, its type has been annotated. If x is not in TE, assume its type as a new type variable, and annotate the type variable with `TypeOf`, to find an explanation for a program variable. In further typing, if a simpler type explanation for x is found replace all the explanations of x with the simpler explanation, so that the type explanation of x is always the simplest.

5. Typing a application, annotate the type of argument subexpression e_2 with `InOperand` and the result type with `SameTyApply` to explain that the arguments at the same position of the same function should have the same type. Find the types of a program variable in different subexpressions by EnvConstraints(TE_1, TE_2), check the type consistent by UnifyConstraintSet($\triangle$), and select the simplest explanation for a program variable.

6. Typing a local definition, annotate the body part e_2 with `LET` to explain that the type of the expression is the type of the body expression.

7. Typing a tuple, annotate each element subexpression with `Element` and the number of the element. For example, the first element is annotated as `Element 1`. Check the type consistency of each program variable in all subexpressions by UnifyEnv(TE_1, TE_2), and select the simplest type explanation for each program variable.

8. Typing a list constructor, annotate the head of the list with `HD`, and the tail with `TL`. Select the simpler explanation of the result type from the list of head type and tail type by SelectSimplest($\tau_{1reason}$ *list*, $\tau_{2reason}$).

   By EnvConstraints(TE_1, TE_2) and UnifyConstraintSet($\triangle$), select the simplest type explanation for each program variable.

9. Typing a abstraction, annotate the left-hand side as `FNLeft` and the right-hand side as `FNRight`. Select the simplest type explanation for each program variable from its uses in the left-hand side pattern P and the body subexpression e by UnifyEnv(TE_1, TE_2).

10. Typing a recursive function definition as is described in Section 6.4.1. The types of subexpressions in the same column are unifyable. The types of a program variable used in different subexpressions in the same row are unifyable. The types of the recursive function in different subexpressions in the body should be same. The argument patterns are annotated as `LHS`, and the result pattern is annotated as `RHS`.

11. Typing a conditional expression, annotate the `if branch` with `IfBranch` which has the known type `bool` type. Annotate the `then branch` with `ThenBranch` and the `else branch` with `ElseBranch`. The result type has the simplest explanation from the type of `then branch` and the `else branch`.

6.4.4 Selecting the shorter explanation

6.4.4.1 Multi-explanation for a program variable

A program variable can be used more than once in different sub-expressions. The $\mathcal{H}$ algorithm returns a TE for each subexpression, which is the type constrains for all its program variables. At the top level of the expression, $\mathcal{H}$ can analyse the uses of the same program variable in each subexpression. For example,

```
fun I x y z w = [x=y, y = z, z=w, y = "ha", z = "simple"]
```

Typing subexpression `x = y`, gives an TE that constrains the types of `x` and `y` to be the same; typing subexpression `y = "ha"`, returns an TE that constrains type of `y` to be the same type as string `"ha"`. After typing all the subexpressions $\mathcal{H}$ analyses all the TE s to decide which type explanation is the simplest for a program variable.

For example, the type of `x` can be explained by:

> `since` $x = y$`, type of x is the same as type of y, since` $y = z$`, the type of y is same as the type of z, since` z=*"simple"*`, z is string, hence x is string`.

A simpler explanation of the type of `x` is:

> `since` $x = y$`, type of x is the same as the type of y, since` $y =$ *"ha"*`, type of y is string, hence type of x is string`.

6.4.4.2 The selector

The selector judges the simplicity of the explanation of a type by the number of annotations of the type. If a type has different explanation choices, the selector selects the one with the minimum number of explanations.

For a **simple type** such as int, real, string, bool, type variable, the selector is the function `selectSimplest()` shown in Appendix C. In the case of its list constructor $e_1 :: e_2$, the type of this subexpression is either list type of the head e_1 or type of tail e_2, as both subexpressions may have different explanations. `SelectSimplest()` will decide which type is chosen for the type explanation of the list constructor. The same selection operation is used at the typing of `if` b `then` e_1 `else` e_2.

The simplest explanation is selected to give the result type of a top expression Consider

$$\mathcal{H} : \text{TypeSchemeEnv} \times \text{Exp} \rightarrow \text{Type} \times \text{TypeSchemeEnv}$$

$\mathcal{H}\ (TE, e) =$
case e of
$[] \Rightarrow$ (1)
  let $\tau_{reason} = Annotate(\alpha\ list, ([], NILLIST))$
  in (τ_{reason}, TE) where α is a fresh type variable
$c \Rightarrow$ (2)
  let $\tau_{reason} = (ConstType(c), (c,\text{BaseType}))$
  in (τ_{reason}, TE)
$+ \Rightarrow$ (3)
  let $\tau_{reason} = Annotate(\delta \times \delta \rightarrow \delta, (+,\text{NumOp}))$
  in (τ_{reason}, TE) where δ is fresh
$x \Rightarrow$ (4)
  if $x \in Dom(TE)$ then
  let $\tau_{reason} = \text{Instance}(TE(x))$
    in (τ_{reason}, TE)
  else
   let β be a fresh type variable
   $\tau_{reason} = Annotate(\beta, (x,\text{TypeOf}))$
   in $(\tau_{reason}, \{x \mapsto \beta\} \cup TE)$
$e_1 e_2 \Rightarrow$ (5)
  let
   $(\tau_1, TE_1) = \mathcal{H}(TE, e_1)$
   $(\tau_2, TE_2) = \mathcal{H}(TE, e_2)$
   $\tau_{reason} = Annotate(\tau_2, (e_2,\text{InOperand}))$
   β be a fresh type variable
   $\beta_{reason} = Annotate(\beta, (e,\text{SameTyApply}))$
   $\triangle = \{\tau_1 \doteq \tau_{reason} \rightarrow \beta\} \cup \text{EnvConstraints}(TE_1, TE_2)$
   $S = \text{UnifyConstraintSet}(\triangle)$
  in $(S(\beta_{reason}), S(TE_1 \oplus TE_2))$
`let` $x = e_1$ `in` $e_2 \Rightarrow$ (6)
  let
   $(\tau_1, TE_1) = \mathcal{H}(TE, e_1)$
   $TE_{new} = TE \oplus \{x \mapsto Close(TE_1, \tau_1)\}$
   $(\tau_2, TE_2) = \mathcal{H}(TE_{new}, e_2)$
   $\tau_{reason} = Annotate(\tau_2, (e_2,\text{LET}))$
   $S = \text{UnifyEnv}(TE_1, TE_2 \ominus x)$
  in $(S(\tau_{reason}), S(TE_1 \oplus (TE_2 \ominus x))$

Figure 6.13: The $\mathcal{H}$ Algorithm (continued in Figure 6.14)

$\mathcal{H}\ (TE, e) =$
case e of
$(e_1, e_2) \Rightarrow$ (7)
    let
        $(\tau_1, TE_1) = \ \mathcal{H}(TE, e_1)$
        $(\tau_2, TE_2) = \ \mathcal{H}(TE, e_2)$
        $\tau_{1reason} = Annotate(\tau_1, (e_1,\text{ELEMENT1}))$
        $\tau_{2reason} = Annotate(\tau_2, (e_2,\text{ELEMENT2}))$
        $\tau = (\tau_{1reason} * \tau_{2reason}, (e,\text{TUPLE}))$
        $S = \text{UnifyEnv}(TE_1, TE_2)$
    in $(S\tau, S(TE_1 \oplus TE_2))$

$(e_1 :: e_2) \Rightarrow$ (8)
    let
        $(\tau_1, TE_1) = \ \mathcal{H}(TE, e_1)$
        $\tau_{1reason} = Annotate(\tau_1, (e_1,\text{HD}))$
        $(\tau_2, TE_2) = \ \mathcal{H}(TE, e_2)$
        $\tau_{2reason} = Annotate(\tau_2, (e_2,\text{TL}))$
        $\tau = SelectSimplest(\tau_{1reason} list, \tau_{2reason})$
        $\triangle = \{\tau_1\ list \doteq \tau_2\} \cup \text{EnvConstraints}(TE_1, TE_2)$
        $S = \text{UnifyConstraintSet}(\triangle)$
    in $(S\tau, S(TE_1 \oplus TE_2))$

$P \Rightarrow e \Rightarrow$ (9)
    let
        $TE' = TE \ominus P$
        $(\tau_1, TE_1) = \ \mathcal{H}(\emptyset, P)$
        $\tau_{1reason} = Annotate(\tau_1, (e_1,\text{FNLeft}))$
        $(\tau_2, TE_2) = \ \mathcal{H}(TE', e)$
        $\tau_{2reason} = Annotate(\tau_2, (e_2,\text{FNRight}))$
        $\tau_{reason} = \tau_{1reason} \rightarrow \tau_{2reason}$
        $S = \text{UnifyEnv}(TE_1, TE_2)$
    in $(S(\tau_{reason}, (e, FN)), S(TE_2 \ominus P))$

Figure 6.14: The $\mathcal{H}$ Algorithm (continued from Figure 6.13)

$\mathcal{H}\ (TE, e) =$
case e of
 FUN (10)
 let
  $\sum\ (\mathcal{U}_{\mathbf{AE}}\ Pattern)$
  $\sum$ (Unify Env_{column})
  $S_{fix} = \text{UnifyTypes}(TE(x))$
 in $((S_{Env} \cup S_{column} \cup S_{fix})\tau_{tk}), \sum TE_{fix} \ominus x))$

`if` b `then` e_1 `else` $e_2 \Rightarrow$ (11)
  let
  $(\tau_1, TE_1) = \ \mathcal{H}(TE, b)$
  $\tau_{1reason} = (\tau_1, (b,\text{IfBranch}))$
  $(\tau_2, TE_2) = \ \mathcal{H}(TE, e_1)$
  $\tau_{2reason} = (\tau_2, (e_1,\text{ThenBranch}))$
  $(\tau_3, TE_3) = \ \mathcal{H}(TE, e_2)$
  $\tau_{3reason} = (\tau_3, (e_2,\text{ElseBranch}))$
  $\triangle = \{\tau_{1reason} \doteq BOOL, \tau_{2reason} \doteq \tau_{3reason}\} \cup \text{EnvConstraints}(TE_1, TE_2)$
   $\cup\ \text{EnvConstraints}(TE_2, TE_3) \cup \text{EnvConstraints}(TE_1, TE_3)$
  $S = \text{UnifyConstraintSet}(\triangle)$
  $\tau_{reason} = SelectSimplest(S(\tau_{2reason}), S(\tau_{3reason}))$
 in $(S(\tau_{reason}, (e, IF)), S(TE_1 \oplus (TE_2 \oplus TE_3)))$

Figure 6.15: The $\mathcal{H}$ Algorithm (continued from Figure 6.14)

```
fn f => fn a => fn y => fn x => fn b =>
                         [f a 1, f 3 b, f x y]
```

The type of `x` can be explained in different ways. One possibility is by the relations

`f x, f a` and `f a, f 3`.

The shortest explanation of the type of `x` is by the relation of:

`f x, f 3`

A simplest explanation is the one that is explained by the minimum number of relations. The $\mathcal{H}$ algorithm:

1. Searches a concrete type at the root, then passes the concrete type information to the leaves of the AST by the TE , and constructs the shortest explanation. If the shortest explanation is given at every subexpression, then it can be deduced that the whole expression has a shortest explanation. TE is set at the beginning of the type checking.

2. The concrete types for a program variable need to be selected at the beginning, with the simplest selected to store in the TE to be passed to the subexpressions. For example, at the the start of the type checking of the above example, if we have $TE = \{$ `f: 'a` $\rightarrow$ `int` , `f: int` $\rightarrow$ `int` $\}$ from different subexpressions, then $\mathcal{H}$ decides $TE = \{$ `f: int` $\rightarrow$ `int` $\}$.

6.5 Type inference explanation from the $\mathcal{H}$ algorithm

6.5.1 Partial type explanations

If there are concrete type subexpressions, the explanations are partial. An example is shown in Figure 6.16. The explanation does not mention the inference of the type of `h` and type of `t`. This make the explanation simpler than the explanation of the $\mathcal{W}$ algorithm inference from Duggan and Bent's [15] which is shown in Figure 6.17. Figure 6.17 shows that the $\mathcal{W}$ inference is given by explaining the type of program variable `h`. In Figure 6.16 the explanation given by the $\mathcal{H}$ algorithm does not issue any type variables, while in Figure 6.17, the explanation given by the $\mathcal{W}$ algorithm issues 4 type variables.

```
        fun f (0::t) = 1110
          | f (h::t) = h + (f t)
-------------------- elab ---------------------
"fun f (0 :: t) = 1110
  |  f (h :: t2) = h + (f t2)",
 is a function type: int list -> int
Argument 1: int list
1: "(0 :: t)" and "(h :: t2)" have the same type
       - same LHS pattern position
2: "[0]", "t" and "(0 :: t)" have same type
       -operands/result of "::"
     3: "0":  int
4:  "[0]": int list

Result: int
5: "1110" and "h + (f t2)" have the same type
   - same RHS result
6: "1110": int
```

Figure 6.16: Partial type inference explanation.

6.5.2 Finding ground type information first

Figure 6.16 shows that in the explanations given by the relationship of subexpressions and ground type, there is no type variable. The $\mathcal{H}$ algorithm searches a concrete type at the root, and then passes the concrete type information to the leaves of the same root by the TE , to make the explanation shortest.

```
          fun f (0::t) = 1110
            | f (h::t) = h + (f t)

    The explanation for the type of f is as follows:

    [0] f : 'A0  by Assumption

    [1] f : int list -> 'A1  because...

    'A0 (the type variable for f) was instantiated to
    int list -> 'A1
    as a result of type-checking this declaration:

    (* f: 'A0 *) fun f (0 :: t: 'A2) = 1110
                   | f (h: 'A1 :: t: 'A3) = Int.+ (h,f t)

    Continue explanation?
    [2] f : int list -> int  because...

    'A1 (the type variable for f.h) was aliased to int
    (the type for 0)
    as a result of type-checking this declaration:

    (* f: int list -> 'A1 *)
    fun f (0 :: t: 'A2) = 1110
    | f (h: 'A1 :: t: 'A3) = Int.+ (h,f t)
```

Figure 6.17: Duggan's type inference explanation.

6.5.3 2D pattern match

Figure 6.16 shows an explanation given by 2D pattern relation. The type of Argument 1 is explained by type of `(0::t)` and `(h::t2)`, which have the same type, because they are in the same LHS pattern position.

6.5.4 Selecting the simpler explanation

In Figure 6.16, the function result type is given by the type of `1110` and `h+ ( f t)`. They have the same type, because they are the same RHS result. $\mathcal{H}$ compares the type explanation information of those two subexpressions, and finds that the type explanation of `1110` is shorter. It then gives the type explanation of `1110`.

6.5.5 Removing unnecessary explanations

Redundant explanations come from the accumulation of the explanations during unification. For example, a type variable α which is the type of program variable `x` in subexpression `f x` is renamed to type β after typing subexpression `f y`, i.e. unifying α and β results in the renaming of type variable α to β, because `x` and `y` are the arguments of the same function `f`: the types of `f` in the two subexpressions should be unified. To make the explanation reasonable, before unifying α and β, it is necessary to reverse the explanations of all type variables. After unifying the types of `f` the explanation of the type of program variable `x` is a compound explanation which is accumulated from the explanation of `y` , i.e.

```
y is type 'b;
x is type 'b, x is type, which has the same type of
the Last Operand in "f(x)", has the same type of
the Last Operand in "f(y)", the type of "y", which is 'b.
```

The explanation of type of `x` is redundant with two `"TypeOf"` annotations, i.e. the "`x is type`".

There are two forms of redundant explanation:

1. Adjacent repeated explanation. E.g. an explanation:

   `type of x has the same type as x, has the same type as x.`

2. Explanations that give no further information. E.g. an explanation:

   `type of x is the same type as type of y , type of y is the same type as type of x.`

The pruner is the function `remRedundant()` shown in Appendix D.

6.5.6 No redundant explanations

If the type of a program variable has been explained, the system will not repeat the explanation. An example in Figure 6.18 shows that the type of `RHS result` has been shown during the explanation of the type of `LHS argument` at line `4`. The system does not repeat the explanation, instead, the system refers to the line number of the first explanation as `see 4 above`.

6.6 Comparison of the explanation of Duggan's and $\mathcal{H}$ algorithm

We compare Duggan's system and our system in some aspects. This comparison may be subjective. A better comparison should be an evaluation experiment. Duggan's system is the best one we have found published so far. It is based on NJSML110.

Both systems are correct, i.e. reports type error for uncorrect programs gives type for type correct program. Both systems are reasonable, i.e. the explanations from the systems are reasonable.

The reader may find that the explanations from Duggan's system need the user to deduce which type variable is instantiated to which type, requiring a long chain of inference. For example, in Figure 6.19:
`'A8` instantiated to `'A9 -> 'A10 -> 'A11 -> 'A10 * 'A11 * real`;

```
    fun f nil = 0.0
      | f (h::t) = h + (f t)
 -------------------- elab ---------------------
 "fun f [] = 0.0
   |  f (h :: t) = h + (f t)",
 is a function type: real list -> real
 Argument 1: real list
 1: "[]" and "(h :: t)" have the same type
     - same LHS pattern position
 2: "[h]", "t" and "(h :: t)" have same type
    - operands/result of "::"
      3: "h" has the same type as "h + (f t)"
         - argument/result of "+" have the same Number types
      4: "0.0" and "h + (f t)" have the same type
         - same RHS result
      5: "0.0": real
 6:  "[h]": real list
 Result: real
 7:  real - see 4 above
```

Figure 6.18: No redundant explanations.

`'A9` is instantiated to `'A10`;
`'A10` is instantiated to `real`.

In Figure 6.20, the $\mathcal{H}$ algorithm explains the types by the relations of subexpressions. There are no type variables in the explanation.

The explanation from $\mathcal{H}$ is shorter than that from Duggan's system.

Both systems are succinct, i.e. There is no unnecessary explanations such as `A = B` because `A = A, A = C, C = B`. Here `A = A` is unnecessary.

$\mathcal{H}$ selects the simpler explanation, but Duggan's system does not. $\mathcal{H}$ gives top-down relationship of expression, and eventually reaches the smallest subexpression. Duggan's does not necessarily reach the smallest subexpression.

$\mathcal{H}$ explains the type relationship between subexpression. Duggan's system explains the instantiation of type variables, but a user does not care about those new type variables during programming.

$\mathcal{H}$ gives all the types of a function argument. $\mathcal{H}$ does not mention the type of any program variable that is not needed in the explanation. The algorithm can be used to enquire about the type of any program variable. Duggan's system may display the types of all program variables.

Figure 6.21 shows the 1D type inference in Duggan's system, while Figure 6.22 shows the 2D type inference in $\mathcal{H}$.

```
fun f x y z = (x+y,y*z,z/2.0);

-----------------------------------
The explanation for the type of f is as follows:

[0] f : 'A8  by Assumption

[1] f : 'A9 -> 'A10 -> 'A11 -> 'A10 * 'A11 * real  because...

'A8 (the type variable for f) was instantiated to
'A9 -> 'A10 -> 'A11 -> 'A10 * 'A11 * real
as a result of type-checking this declaration:

(* f: 'A8 *)
fun f (x: 'A9) (y: 'A10) (z: 'A11) =
      (Real.+ (x,y),Real.* (y,z),Real./ (z,2.0))
[2] f : real -> real -> real -> real * real * real  because...

'A9 (the type variable for f.x)
was aliased to 'A10 (the type for f.y) as a
result of type-checking this expression:

Real.+ (x,y)

'A10 (the type variable for f.y)
was aliased to 'A11 (the type for f.z) as a
result of type-checking this expression:

Real.* (y,z)

'A11 (the type variable for f.z)
was instantiated to real as a result of
type-checking this expression:

Real./ (z,2.0)
```

Figure 6.19: Duggan's system explains types by instantiation of type variables

```
"fun f x y z = (x + y, y * z, z / 2.0)",
is a function type: real -> real -> real -> real * real * real
Argument 1: real
1: "x" has the same type as "y" in "x + y"
   - arguments of "+" have the same Number types
2: "y" has the same type as "z" in "y * z"
   - arguments of "*" have the same Number types
3: "z" and "2.0" have the same type in "z / 2.0"
   - arguments/result of "/" have the same Real types

Argument 2: real
4: "y":  real - see 2 above

Argument 3: real
5: "z":  real - see 3 above

Result: real * real * real
6:  "(x + y, y * z, z / 2.0)",
    is a Tuple type: real * real * real
Tuple Element 1: real
  7: "x + y" and "y" have the same type
     - result/arguments of "+" have the same Number types
  8: "y":  real - see 2 above
Tuple Element 2: real
  9: "y * z" and "z" have the same type
     - result/arguments of "*" have the same Number types
  10: "z":  real - see 3 above
Tuple Element 3: real
  11: "z / 2.0" has the same type as "z" and "2.0"
      - result/arguments of "/" have the same Real types
  12: "2.0": real
```

Figure 6.20: $\mathcal{H}$ explains types by instantiation of type variables

6.7 Error explanations

In related work Wand [59] and Duggan etc [15] use modified unification algorithms to accumulate "reason lists" which are the subexpressions that contribute the inference of the conflicting types. The "reason lists" are the potential error sources. But the method is not sufficient to fully explain type errors from the "reason lists". For example, given a expression with type error:

```
fn f => fn x => fn y=> f(y x) (y 3) (not x)
```

```
3. fun f 1 y = y | f x 2 = x;

[0] f : 'A6  by Assumption

[1] f : int -> 'A7 -> 'A8  because...

'A6 (the type variable for f) was instantiated to
int -> 'A7 -> 'A8 as a result
of type-checking this declaration:

(* f: 'A6 *) fun f (1) (y: 'A7) = y
               | f (x: 'A8) (2) = x
Continue explanation?
[2] f : int -> int -> int  because...

'A7 (the type variable for f.y) was aliased to
'A8 (the type for f.x) as a
result of type-checking this declaration:

(* f: int -> 'A7 -> 'A8 *)
fun f (1) (y: 'A7) = y
  | f (x: 'A8) (2) = x

'A8 (the type variable for f.x) was aliased to
int (the type for 1) as a result
of type-checking this declaration:

(* f: int -> 'A7 -> 'A8 *)
fun f (1) (y: 'A7) = y
  | f (x: 'A8) (2) = x
```

Figure 6.21: Duggan's system explains types by instantiation of type variables

Wand's algorithm gives the reason list of two subexpression: { `(y 3), (not x)` }. From this reason list, it is not possible to reason about the type error.

In contrast, in figure 6.23, the $\mathcal{H}$ algorithm gives the reason list of three subexpression: { `(y x), (y 3), (not x)` }, which is enough to reason about the type conflict of `x` or `y`. To avoid lengthy explanation, the $\mathcal{H}$ algorithm gives the explanation of type of `x` only:`x` can be `bool` type from the use of `not x`, or can be `int` type, from the uses of `(y x)` and `(y 3)`, since `x` and `3` are the arguments of the same lambda-bound variable `y`.

In Figure 6.24, the $\mathcal{H}$ algorithm distinguishes the same name in different scopes using an index. Consider:

```
"fun f 1 y = y
  |  f x 2 = x",
is a function type: int -> int -> int
Argument 1: int
1: "1" and "x" have the same type
    - same LHS pattern position
2: "1": int

Argument 2: int
3: "y" and "2" have the same type
    - same LHS pattern position
4: "2": int

Result: int
5: "y" and "x" have the same type
    - same RHS result
6:  int - see 3 above
```

Figure 6.22: $\mathcal{H}$ explains types by relationships between subexpressions

```
fn f => fn x => fn y=> f(y x) (y 3) (not x);

-------------------- Error --------------------
Type conflicts are found in expression:
   f(y x)(y 3)(not  x)


found expressions that should have the same type are
inferred different types:

    "x" the Operand of " not  x", is the same type as  bool

    "x" the Operand of "y x", is the same type as
        the Operand of "y 3", is the same type as "3": int
```

Figure 6.23: Example of type error explanation

`if x then (inc x ) x else 2 end`,

`x` is different from that in the definition:

`fun inc x = x + 1`.

$\mathcal{H}$ annotates the body part as

`if x1 then (inc x1 ) x1 else 2 end`

$\mathcal{H}$ explains that in the body part subexpression `inc`, `x1` has the same type as in `x + 1`, according to the local definition. Since `x + 1` is of `int` type, not a function type, `(inc x1)` is of `int` type, `(inc x1) x1` has type error.

```
fn x => let fun inc x = x + 1
        in if x then (inc x ) x else 2
        end;

-------------------- Error --------------------
fn x1 => let fun inc x = x + 1
          in if x1 then (inc x1 ) x1 else 2
         end;
Type conflicts are found in expression:
   (inc x1) x1

operator is not a function, but has type:
"inc x1" has the same type as "x + 1"
 has the same type as "x" and "1"
 - result/arguments of "+" have the same Number types
"1": int
```

Figure 6.24: Example of type error explanation

Another example is shown in Figure 6.25. The type error explanations are given by explaining how $\mathcal{H}$ infers two conflicting types. $\mathcal{H}$ reports that the type of `x` can be inferred as conflicting types `int` and `real` in different subexpressions `x >0` and `0.0 - x`.

The $\mathcal{H}$ error explanation uses 2D relations as well as type inference explanation. In Figure 6.26, $\mathcal{H}$ explains that `h1` can be inferred as `bool` type: from the RHS relation, `v` has the same type as `false`, from RHS relation, `v` has the same type as `h1`. However, from subexpression `h1>h2`, `h1` has `Number` type, which is inconsistent with `bool` type.

6.8 Conclusion

We have constructed a type explanation system for a simple, pure functional, list-based subset of SML, influenced strongly by our experimental investigation of expert human type explanation. A comparision of human expert explanations and the system explanations is presented in [68]. Our system will also explain type errors, by providing explanations of conflicting sites of unification found by the $\mathcal{U}$AE algorithm.

```
fn x => if x>0 then x  else 0.0 - x;

-------------------- Error --------------------
Type conflicts are found in expression:
    if x > 0 then x else 0.0 - x

found expressions that should have the same type are
inferred different types:

   in one expression has type:
     "x" and "0.0" have the same type in "0.0 - x"
     - arguments/result of "-" have the same Number types
1: "0.0": real

   in another expressions has type:
     "x" and "0" have the same type in "x > 0"
     - arguments of ">" have the same Number types
1: "0": int
```

Figure 6.25: Example of type error explanation

```
fun f [] = false
  | f [v] = v
  | f (h1::h2::t) = h1> h2;

-------------------- Error --------------------
Type conflicts are found in expression:
    f (h1 :: (h2 :: t)) = h1 > h2


found expressions that should have the same type are
inferred different types:

   "h1" has the same type as an Element in
    the tail of "(h1 :: (h2 :: t))",
     "[h1]", "(h2 :: t)" and "(h1 :: (h2 :: t))"
     have same type
     -operands/result of "::"
     "[]", "[v]" and "(h1 :: (h2 :: t))"
     have the same type
     - same LHS pattern position
     "[v]", whose Element type:
         "false", "v" and "h1 > h2" have the same type
          - same RHS result
     "false": bool

   "h1" and "h2" have the same type in "h1 > h2"
    - arguments of ">" have the same Number types Number'a
      (int or real, non-resolved.)
```

Figure 6.26: Example of type error explanation

CHAPTER 7
CONCLUSIONS

7.1 Summary

This thesis investigates techniques for making mechanical polymorphic type inference more understanding. Most type checking algorithms are based on Hindley-Milner type inference [35], which is devised to effectively type check, rather than support human understanding of types. In particular, type variables are used as place holders for the as yet unknown types of sub-expressions. Error reporting systems that build on the Hindley-Milner approach tend to use such intermediate representations in explaining how types are derived, although the programmer does not know about these type variables at all. Systems explaining how such algorithms work tend to be counterintuitive. Furthermore, the explanations are not succinct.

To support better understanding of type inference explanations and type error explanations it is worth exploring new approaches to representing types and to study how people explain type inference and type errors. A survey of type representations in visual programming languages has been presented in Chapter 2. A survey of error reporting from different algorithms has also been reported in Chapter 2.

In Chapter 3, it was shown how a simple visualization of polymorphic types [63] may be incorporated within a visual environment to guide and clarify type compatibility in function application. The empirical results do not show that the visualization is necessarily better than a traditional textual representation. Rather, they show that this way of communicating about types seems to have different strengths than those of the traditional textual representation. The empirical results also suggest that a combination of the visualization with the textual representation might achieve greater understandability than either can alone.

To support better understanding of type errors, the $\mathcal{U}$AE algorithm was devised. To report error sites with greater precision, the $\mathcal{IEI}$ algorithm was proposed [62]. Both are presented in Chapter 4, together with a comparison of the improved algorithms [69]. In large programs the $\mathcal{U}$AE suffers from including irrelevant program text in the error report. The $\mathcal{IEI}$ algorithm uses global inference, often reporting types from a long chain of inference. In effect, the $\mathcal{IEI}$ algorithm gains greater relevance at the price of succinctness. The type information from all the algorithm needs some explanation to support understanding.

We wished to build a type explanation system that is in some sense "human like"; that is it should incorporates rules intended to mimic the behavior of people explaining type derivations. We have observed in Chapter 5 a number of important differences between human type checking, as characterized by our composite expert subject group, and the $\mathcal{W}$ algorithm. People tend to give explanations that avoid intermediate representations, focusing on concrete aspects of the program being analyzed. In building our system, we attempted to incorporate rules that reflect such behaviors [67]. The system is introduced in Chapter 6.

The type explanation from the system are very unlike those from Hindley-Milner based approaches and appear more human-like to us. Comparison suggested that our system has similar behavior to human experts [68]. In particular, the system seeks to provide succinct explanations with minimal reference to artifacts of mechanized type inference. The system does not use type variables in the explanation if the type is not polymorphic. The implications for type error explanation are discussed.

7.2 Contributions

The primary research contributions in this thesis can be summarised as follows.

1. *Visualization of polymorphic types* gives a different approach for representing types from the tradition textual representation.

2. $\mathcal{U}$AE and $\mathcal{IEI}$ are novel algorithms designed to improve the error reporting. $\mathcal{U}$AE improves the understanding of error reporting, while $\mathcal{IEI}$ improve precision of error reporting in some way.

3. *Human expert type explanation techniques* are identified. The experiments with expert human type checkers confirmed our intuitions that people tend to give explanations that avoid intermediate representations such as type variables, and abstract syntax trees.

4. *A human-like type error explanation system* has been constructed and evaluated. We have analyzed the system's behavior using the same categories of explanation techniques on the same problem set. While there are some differences, comparison suggests that our system has similar behavior to human experts.

7.3 Limitations of current work

The current work has the following limitations, and these indicate further work.

1. $\mathcal{U}$AE and $\mathcal{IEI}$ work on a subset of SML. While it is straightforward to extend them to the all the syntax structures of core language, further investigation is needed to extend them to the full language, including user defined datatypes and modules.

2. The current $\mathcal{U}$AE and $\mathcal{IEI}$ implementations are free standing, while the $\mathcal{W}$ and $\mathcal{M}$ algorithms are implemented in compliers.

3. The error sites reported by the $\mathcal{U}$AE are not the most precise ones. The type information reported by the $\mathcal{IEI}$ is not easy to understand.

4. The $\mathcal{H}$ algorithm has not been formally evaluated with user group. One reason for lack of evaluation is that it is hard to measure when one type explanation is easier to understand than another explanation of the same expression.

5. The $\mathcal{H}$ algorithm gives long explanations for `Let` bound variables.

6. Lack of formal correctness proofs of the $\mathcal{U}$AE and the $\mathcal{H}$ algorithms.

Those are the motivations for future work.

7.4 Future work

There are the following five possible directions for future work.

Usability testing of the $\mathcal{H}$ algorithm explanations should be conducted with naive and expert users. The methods of testing how an explanation is more understandable than other kinds need to be developed.

The capability of catching the technique sequence of a user type explanation online could be developed where the system automatically adapts to the simplest or shortest human explanation techniques. A user could use a pointer to give type explanations for a sub-expression and the system could record the explanation process and analyses the technique sequences used by the user, to study the best way to give an explanation.

A new set of type rules is required to locate a most precise error site. The $\mathcal{U}$AE reporting may include some irrelevant parts which do not contribute directly to the type errors. By typing under a new set of type rules, the error reporting might be able to report the subexpressions that directly conflict. Armed with the type explanation information, error reporting would be easier to understand and more precise.

The explanations should be extended to a full language, including user defined discriminated union types and classes or modules. Currently the explanation methods work on a subset of the core language of SML. To develop explanations with user defined types, some experiments with how people understand the use of datatypes in programs should be conducted. In the same way, understanding of classes or modules and explanations of corresponding type errors need be studied.

Proofs of the possible correctness property of these algorithms are challenging work. It worth to proof the correctness of the $\mathcal{U}$AE and the $\mathcal{H}$ algorithms with respect to Hindley-Milner type inference.

The type inference explanation processes could be animated to avoid giving lengthy English text explanation for complex function definitions. The system might try to behave like a tutor by using pointers and colors, to indicate the type relations between sub-expressions in a program, and the process of type inference. An interactive system explaining the type of a user identified subexpression would be very useful, since only the programmer knows which part of program requires type explanation.

APPENDIX A
Ancillary definitions of inference algorithms

A.1 General mathematical definitions

$\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$.
$Fin(S) = \{S' \mid S' \ finite \wedge S' \subseteq S\}$.
$S \to S' = \{f \mid f \in S \times S' \wedge (\{(a,b),(a,c)\} \subseteq f \Rightarrow b = c)\}$
$Dom(f) = \{x \mid (x,y) \in f\}$ — the domain of function f
$S \overset{fin}{\to} S' = \{f \mid f \in S \to S' \wedge Dom(f) \text{ is finite }\}$
$\downarrow: (S \to S') \times \mathcal{P}(S) \to (S \to S')$
$f \downarrow S'' = \{a \mapsto b \mid a \mapsto b \in f \wedge a \notin S''\}$

A.2 Free type variables

$ftv(\forall \vec{a}.\tau) = ftv(\tau) \setminus \vec{a}$. — free variables in type scheme σ.

$ftv(\rho) = \{\rho\}$
$ftv(\eta) = \emptyset$ if $\eta \in \{STRING, BOOL, INT, REAL\}$
$ftv(\eta \to \eta') = ftv(\eta) \cup ftv(\eta')$
$ftv(\eta * \eta') = ftv(\eta) \cup ftv(\eta')$
$ftv(\eta \ list) = ftv(\eta)$

For a type environment TE,
$ftv(TE) = \bigcup_{x \in Dom(TE)} ftv(TE(x))$. — free variables in type environment TE.

A.3 ConstType function

ConstType: Constant $\to$ Type — type of constant used in Figure 4.8

ConstType(c) = if $c \in$ Int then INT
else if $c \in$ Real then REAL
else if $c \in$ Bool then BOOL
else STRING

A.4 The ancillary functions for unification

SimplifyConstraint($\tau \doteq \alpha$) = $\{\alpha \doteq \tau\}$ where $\tau \notin$ TyVar
SimplifyConstraint($\tau \doteq \delta$) = $\{\delta \doteq \tau\}$ where $\tau \notin$ NumTyVar $\cup$ TyVar
SimplifyConstraint($\eta \doteq \eta$) = $\emptyset$
SimplifyConstraint ($\eta_1 \to \eta_1' \doteq \eta_2 \to \eta_2'$) =
SimplifyConstraint($\eta_1 \doteq \eta_2$) $\cup$ SimplifyConstraint($\eta_1' \doteq \eta_2'$)

$$
\begin{array}{ll}
\text{SimplifyConstraint } (\eta_1 * \eta_1' \doteq \eta_2 * \eta_2') = & \\
\qquad\qquad \text{SimplifyConstraint}(\eta_1 \doteq \eta_2) \cup \text{SimplifyConstraint}(\eta_1' \doteq \eta_2') & \\
\text{SimplifyConstraint}(\eta\ list \doteq \eta'\ list) = & \\
\qquad\qquad \text{SimplifyConstraint}(\eta \doteq \eta') & \\
\text{SimplifyConstraint}(\forall\{\alpha_1, \cdots, \alpha_n\}.\tau \doteq \theta) = & \\
\qquad\qquad \text{SimplifyConstraint}(\{\gamma_1/\alpha_1, \cdots, \gamma_n/\alpha_n\}\tau \doteq \theta) & \\
\qquad\qquad \cup\{\gamma_i \neq \gamma_j \mid 1 \leq i < j \leq n\} & \\
\qquad\qquad \text{where } \alpha_1, \cdots, \alpha_n \text{ are distinct,} & \\
\qquad\qquad \text{and } \gamma_1, \cdots, \gamma_n \in \text{QTyVar are fresh.} & \\
\text{SimplifyConstraint}(\eta \doteq \forall\vec{\alpha}.\tau') = & \\
\qquad\qquad \text{SimplifyConstraint}(\forall\vec{\alpha}.\tau' \doteq \eta) & \\
\text{SimplifyConstraint}(\eta \doteq \eta') = \{\eta \doteq \eta'\} \text{ if no other case matches.} &
\end{array}
$$

$$
\begin{array}{l}
\text{SimplifyConstraintSet}(\emptyset) = \emptyset \\
\text{SimplifyConstraintSet}(\{\theta \doteq \theta'\} \cup \triangle) = \\
\qquad\qquad \text{SimplifyConstraint}(\theta \doteq \theta') \cup \text{SimplifyConstraintSet}(\triangle) \\
\text{SimplifyConstraintSet}(\{\gamma \neq \gamma'\} \cup \triangle) = \\
\qquad\qquad \{\gamma \neq \gamma'\} \cup \text{SimplifyConstraintSet}(\triangle)
\end{array}
$$

$$
S(\triangle) = \{(S\theta) \doteq (S\theta') \mid (\theta \doteq \theta') \in \triangle\} \cup \{S(\gamma) \neq S(\gamma') \mid (\gamma \neq \gamma') \in \triangle\}
$$

A.5 The UnifyConstraintSet, UnifyTypes and UnifyEnv functions

$$
\begin{array}{lll}
\text{UnifyLoop}(\triangle, S) = & \text{let } \ \triangle' = \text{SimplifyConstraintSet}(\triangle) & \\
& \text{in } \ \text{case } \triangle' \ \text{ of} & \\
& & \{\gamma_{i,1} \neq \gamma_{i,2} \mid i \in I\} \\
& & \Rightarrow S \downarrow \text{QTyVar} \\
& & \\
& & \{\gamma \doteq \gamma'\} \cup \triangle'' \text{ where } \{\gamma \neq \gamma', \gamma' \neq \gamma\} \cap \triangle'' = \emptyset \\
& & \Rightarrow \text{let } \ S' = \{\gamma/\gamma'\} \\
& & \quad\ \text{in } \ \text{UnifyLoop}(S'(\triangle''), S'S) \\
& & \\
& & \{\alpha \doteq \tau\} \cup \triangle'' \text{ where } \alpha \notin ftv(\tau) \\
& & \Rightarrow \text{let } \ S' = \{\tau/\alpha\} \\
& & \quad\ \text{in } \ \text{UnifyLoop}(S'(\triangle''), S'S) \\
& & \\
& & \{\delta \doteq \nu\} \cup \triangle'' \\
& & \Rightarrow \text{let } \ S' = \{\nu/\delta\} \\
& & \quad\ \text{in } \ \text{UnifyLoop}(S'(\triangle''), S'S) \\
& & \\
& & \triangle \\
& & \Rightarrow \text{fail}
\end{array}
$$

UnifyConstraintSet($\triangle$) = UnifyLoop($\triangle, \emptyset$)

UnifyTypes(τ, τ') = UnifyConstraintSet($\{\tau \doteq \tau'\}$)

EnvConstraints(TE_1, TE_2) = $\{TE_1(x) \doteq TE_2(x) \mid x \in (Dom(TE_1) \cap Dom(TE_2))\}$

UnifyEnv (TE_1, TE_2) = UnifyConstraintSet(EnvConstraints(TE_1, TE_2))

A.6 The Instance function

Instance ($\forall\{\alpha_1, \cdots, \alpha_n\}.\tau$) = $\{\beta_1/\alpha_1, \cdots, \beta_n/\alpha_n\}(\tau)$
$\beta_1, \cdots, \beta_n$ are fresh type variables, $\alpha_1, \cdots, \alpha_n$ are distinct.
Instance (τ) = τ

A.7 Substitution

$R, S \in$ Substitution $::= \{\tau_1/\alpha_1, \cdots, \tau_n/\alpha_n, \nu_1/\delta_1, \cdots, \nu_m/\delta_m, \gamma_1'/\gamma_1, \cdots, \gamma_p'/\gamma_p,\}$

For a substitution S and a type τ, $S\tau$ is the type resulting from applying every substitution component in S to τ.
$S(\eta) = \eta$ if $\eta \in \{INT, REAL, STRING, BOOL\}$ S apply to a type
$S(\rho) =$ if $\{\eta/\rho\} \in S$ then η else ρ
$S(\eta_1 \to \eta_2) = (S(\eta_1)) \to (S(\eta_2))$
$S(\eta_1 * \eta_2) = (S(\eta_1)) * (S(\eta_2))$
$S(\eta\ list) = (S(\eta))\ list$

For a substitution S and a type scheme $\sigma = \forall\{\alpha_1, \cdots, \alpha_n\}.\tau$,
$S\sigma = \forall\{\beta_1, \cdots, \beta_n\}.S\{\beta_1/\alpha_1, \cdots, \beta_n/\alpha_n\}\tau$, S apply to a type scheme σ
where $\alpha_1, \cdots, \alpha_n$ are distinct, $\beta_1, \cdots, \beta_n$ are distinct
and $\{\beta_1, \cdots, \beta_n\} \cap (itv(S) \cup ftv(\sigma)) = \emptyset$.

For a substitution S, the support
$supp(S) = \{\rho \mid S\rho \neq \rho\}$. support type variables of S
For a substitution S, the involved type variables
$itv(S) = \{\rho \mid \rho' \in supp(S)$
$\wedge \rho \in \{\rho'\} \cup ftv(S\rho')\}$. involved type variables of S

For substitution S and substitution R,
$RS = \{R(S\rho)/\rho \mid \rho \in supp(S) \cup supp(R)\}$. composition of substitution

For a substitution S and a type environment TE,
$S(TE) = \{x \mapsto S\sigma \mid x \mapsto \sigma \in TE\}$. S apply to a type environment TE

A.8 Operations on type environments

$FV(x) = \{x\}$ program variable in expression
$FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$
$FV(e_1, e_2) = FV(e_1) \cup FV(e_2)$
$FV(e_1 :: e_2) = FV(e_1) \cup FV(e_2)$
$FV([]) = \{\}$

$Close(TE, \tau)$ denotes a type scheme generated by closing a type τ under type environment TE, which is $\forall \alpha_1 \ldots \alpha_n . \tau$ where $\{\alpha_1 \ldots \alpha_n\} = ftv(\tau) \setminus ftv(TE)$.

$\ominus$: (TypeSchemeEnv $\times$ Patt) $\rightarrow$ TypeSchemeEnv

remove type variables in a pattern expression from domain of a type environment
$TE \ominus P = TE \downarrow FV(P)$

$TE \oplus TE' = (TE \downarrow Dom(TE')) \cup TE'$

APPENDIX B
Rule annotations in type nodes

Rule annotation	Details
APP	function application
InOperand	the argument of the same function at different applications
SameTyApply	the applications that have the same result types
IFBranch	ifthen branch
ThenBranch	then branch
ElseBrance	else branch
BaseType	ground type nodes
TypeOf	type of a sub-expression
HD	head of a list
TL	tail of a list
::	list constructor `::`
@	list connection operator `@`
ResultList	result type of list operator
InList	list type of type of a subexpression
ElementTy	elements of a list
TupleElement	elements of a tuple
Row	the number of clause in a recursive definition
LHS	left-hand side patterns of a recursive definition
RHS	right-hand side patterns of a recursive definition
LET	local definition

Rule annotation	Details (continued)
NumOp	number operator
RealOp	real number operator
IntOp	integer number operator
StringOp	string operator
CompOp	comparison operator
ResultNum	result type of number operator
ResultComp	result type of comparison operator
CONS	list concatenation
FN	function definition
FN-L	function left hand side
FN-R	function right hand side
FUN	recursive function definition
OP-*op*	expression with system operator *op*
OP-L	system operator left hand side
OP-R	system operator right hand side
VAL	`val` definition
VAL-L	`val` left hand side
VAL-R	`val` right hand side

APPENDIX C
Selecting the simplest explanation

```
 fun numOfReason  (RETYPE(ty, (e1, ty1, s)) ) = 1 + numOfReason ty
   | numOfReason  (ARROW(ty, ty')) = numOfReason ty + numOfReason ty'
   | numOfReason  (LIST ty) = numOfReason ty
   | numOfReason  (TIMES (h::t)) = numOfReason h + numOfReason (TIMES t)
   | numOfReason  other = 0

 fun simplest (nil, lastNum, lastResult) = lastResult
   | simplest ((h::t), lastNum, lastResult) =
         let val numberOfReason = numOfReason (h)
          in if numberOfReason<= lastNum
             then simplest (t , numberOfReason, h)
             else simplest (t,  lastNum, lastResult)
         end

 fun selectSimplest nil  = raise Type
   | selectSimplest [h] = h
   | selectSimplest (h::t) = simplest(t, numOfReason(h), h)
fun tyEqual (t1, t2) = equal (DeRETYPE t1, DeRETYPE t2)

(* sort the types (which will be unified ) into a list,
   the same types stick togather *)
fun quickSort [] = []
  | quickSort [m] = [m]
  | quickSort (a::bs) =
      let fun partition (l, r, nil) = (quickSort l) @ (a:: quickSort r)
            | partition (l, r, x::xs) =
                if tyEqual (x, a)
                then partition (x::l, r, xs)
                else partition (l, x::r, xs)
       in partition(nil, nil, bs)
      end

(* sp a sorted list of type to a list of list,
   in a list is type equal types,
   which are with different explanations *)
fun sp nil last result = result @ [last]
  | sp (h::t) last result = if tyEqual(h , (hd last ))
                          then sp t (h::last ) result
                          else sp t [h] (result @ [last] )

(* sp a sorted list of type to a list of list,
   in a list is type equal types, which are
```

```
   with different explanations *)
fun spToList nil = nil
  | spToList (h::t) = sp t [h] nil

(* sort the type to a list of list, then selects a type
   with simplest explanation in a list this forms a list
   of type with the simplest explanation  *)
fun selectSimplestTyList tyList =
     map selectSimplest (spToList(quickSort(tyList)))
```

APPENDIX D
Removing the redundant explanations

```
fun remRedundant(t_with_reason as RETYPE(ty, reason as (e, t, s))) =
     let
         fun eqReason ( (e1, t1, s1), ((e2, t2, s2), lineNo) ) =
                              Ex.eqExp(e1, e2) andalso (s1 = s2 )
         fun prReason (e1, _, s1) = "\n "^ s1 ^"  " ^ Ex.pr e1

         fun reasontoList  (RETYPE(ty, (e1, ty1, s)) ) l =
                              reasontoList ty (l @ [(e1, ty1, s)])
           | reasontoList other l = l

         fun sameLineReason (nil, lineNo) l = l
           | sameLineReason ((Ex.BOOLexpr(_),ty, s)::t, lineNo) l =
                                        sameLineReason (t, lineNo) l
           | sameLineReason ((Ex.NUMBERexpr(_),ty, s)::t, lineNo) l =
                                        sameLineReason (t, lineNo) l
           | sameLineReason ((Ex.RealNUMBERexpr(_),ty, s)::t, lineNo) l =
                                        sameLineReason (t, lineNo) l
           | sameLineReason ((Ex.STRINGexpr(_),ty, s)::t, lineNo) l =
                                        sameLineReason (t, lineNo) l
           | sameLineReason ((e,ty,"BaseTyOf")::t, lineNo) l =
                                        sameLineReason (t, lineNo) l
           | sameLineReason (h::t, lineNo) l =
                      sameLineReason (t, lineNo) ( l @ [(h, lineNo)])

         fun fixsize s = if size s <10
                         then s ^ space(10-size(s))
                         else s

         fun DeSingleTopReason
              (ty as RETYPE(ty1, (e1, ty1', "BaseTyOf"))) lineNo = ty
           | DeSingleTopReason
             (ty as RETYPE(ty1, (e1, ty1', s))) lineNo =
                  (
                  ExplainedReason_list :=
                       !ExplainedReason_list
                       @ (sameLineReason(
                        (reasontoList t_with_reason nil), lineNo) nil) ;
                        RETYPE(DeRETYPE ty1,
                         (e1, UNIT, "Explained" ^",\"^s^",\" ^ lineNo) )
                                 )
           | DeSingleTopReason other lineNo =
                (report("\n-------That is " ^ prType other); other)
```

```
fun isIn(x,[])= false
  | isIn(x,hd::tl) = eqReason(x , hd) orelse isIn(x,tl)
fun subset (nil, _) = true
  | subset ((h::t), l) = isIn(h, l) andalso subset(t, l)
fun eqsub (nil, nil) = true
  | eqsub (nil, _) = true
  | eqsub (_, nil) = false
  | eqsub ((h::t), (h1::t1)) =  eqReason(h , h1)
                                     andalso eqsub(t, t1)
fun rmhd (_, nil) = nil
  | rmhd (nil, _) = nil
  | rmhd ((h::t), (h1::t1))= if not (eqReason(h, h1))
                             then rmhd((h::t), t1)
                             else (h1::t1)
fun eqSub1 (l1, l2) = eqsub(l1, rmhd(l1, l2))
fun eqSub (l1, nil) = false |
    eqSub (l1, h::t) = eqSub1(l1, h::t) orelse eqSub(l1, t)

fun reasonlineNo (l1, l2) = #2 (hd (rmhd(l1, l2)) )

fun sameLRHReason (nil, ty, s) l = l
  | sameLRHReason ((h::t), ty, s) l =
      sameLRHReason (t, ty, s)
                   (l @ [((h, ty, s),
                          Int.toString(lineNumber()))])
fun  reasonLRHS  (RETYPE(ty, ( Ex.LHS(e_list), ty1, s)) ) =
                          sameLRHReason(e_list, ty1, s) nil
  |  reasonLRHS  (RETYPE(ty, ( Ex.RHS(e_list), ty1, s)) ) =
                          sameLRHReason(e_list, ty1, s) nil
  |  reasonLRHS _ = nil

fun sameLRHReason1 (nil, ty, s) l = l
  | sameLRHReason1 ((h::t), ty, s) l =
          sameLRHReason1 (t, ty, s) (l @ [(h, ty, s)])
fun  reasonLRHS1  (RETYPE(ty, ( Ex.LHS(e_list), ty1, s))) =
                         sameLRHReason1(e_list, ty1, s) nil
  |  reasonLRHS1  (RETYPE(ty, ( Ex.RHS(e_list), ty1, s))) =
                         sameLRHReason1(e_list, ty1, s) nil
  |  reasonLRHS1 _ = nil

fun whichLine reason (h::t) = if eqReason(reason, h)
                              then #2 (h)
                              else whichLine reason t
  | whichLine _ nil = Int.toString(~1)
```

```
        fun isLRHSExplained  (RETYPE(ty, (e1 as Ex.LHS(nil), ty1, s)
            )) l = false
          | isLRHSExplained  (RETYPE(ty,
                               (e1 as Ex.LHS(h::e_list), ty1, s)))l =
                isIn((h, ty1, s), l)
                orelse (isLRHSExplained  (RETYPE(ty,
                                    ( Ex.LHS(e_list), ty1, s)) ) l)
          | isLRHSExplained _ _ = false

        val result =  if eqSub(reasontoList t_with_reason nil,
                                          !ExplainedReason_list)
                      then  DeSingleTopReason t_with_reason
                             (reasonlineNo
                               (reasontoList t_with_reason nil,
                                   !ExplainedReason_list))
                      else (ExplainedReason_list :=
                            !ExplainedReason_list
                            @[(reason, Int.toString(lineNumber()))];
                            t_with_reason)
  in result end

| remRedundant ty = ty
```

APPENDIX E
Examples of error-reportings

We give some examples of SML to compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}$**AE**, $\mathcal{IEI}$ algorithms. The algorithms are discussed in [69].

E.1 Examples

The selected examples are SML expressions with type errors:

1. Type error with lambda-bound variables used as function and integer;

```
fn x => fn a =>  (x a, x 2, [x, 2]);
```

2. Ground type conflict with lambda-bound variables;

```
fun f (c::cs) (i::is) = if i> 0
                        then f cs is
                        else (c::[2.2])@ f is cs;
```

3. List type conflict with lambda-bound variables;

```
fun m f (h::t) = f h::f t ;
```

4. Type error variable in local definition;

```
let val f = fn x => let val y = x
                     in y 5 end
 in f 3 end;
```

5. Type error with local definition with same variables in body;

```
fn x => let val y = (fn z=> let val _ = x z
                             in (fn w=> w ) end)
         in (y 5, y true) end;
```

6. Type error with local definition with partial polymorphic types;

```
fn x=> let val f = fn y => y x
        in  (f (fn z => z))(f (fn u => fn v=> u)) end ;
```

7. Type error with local definition with same variables in local definitions;

```
fn x => let val f = fn y => y x val g = fn z => z x
         in (f (fn w => w + 1), g (fn u => not u)) end;
```

8. Type error with local definition with same variables in nest local definitions;

```
fn z=> let val x = z
        in (let val y = z 1 in x true end) end;
```

9. Variables with conflicting uses in application;

```
fun f y (h::t) = t y;
```

10. Variables with conflicting ground type at different sites;

```
fun f5 0 n = []
  | f5 m n = m ^ n ::(f5 (m-1));
```

11. Then branch and Else branches type conflicts;

```
fun f5 x  = if x>3 then x else 1.1 ;
```

12. Type conflicts in a pattern;

```
fun f4 [] = []
  | f4 (0::t) = f4 t
  | f4 (h::t) = h/2.0::f4 t;
```

13. Type conflicts between patterns;

```
fun fail3 p nil = p + p
  | fail3 p (h::t) = if p true then [h] else t ;
```

14. Type conflicts between patterns and overloading;

```
fun f 1 x = 0.0
  | f n x = (n*x) + (f (n-1) x);
```

15. Type errors in many sites;

```
fn x => if x then x+1 else x-2.2
```

16. Type errors in many sites and can stop typechecking earlier;

```
fn x => (x+1) (if x then x+1 else x-2.2)
```

E.2 Error reporting by algorithms

E.2.1 Example 1

```
fn x => fn a =>  (x a, x 2, [x, 3]);
```

$\mathcal{W}$ **algorithm reports:**

```
stdIn:18.29-18.35 Error: operator and operand don't agree [literal]
  operator domain: (int -> 'Z) * (int -> 'Z) list
  operand:         (int -> 'Z) * int list
  in expression:
    x :: 3 :: nil
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    3
Type inconsistent of requirement at the expression.
required type: int -> 'a
actual type  : int
```

$\mathcal{U}$AE **reports:**

```
Type conflicts in expressions:
   x(2)
([x, 3])

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

    but from the second expression
x: int
```

$\mathcal{IEI}$ **algorithm reports:**
Same as $\mathcal{U}$AE report.

E.2.2 Example 2

```
fun f (c::cs) (i::is) = if i> 0
                        then f cs is
                        else (c::[2.2])@ f is cs;
```

$\mathcal{W}$ **algorithm reports:**

```
stdIn:20.51-20.70 Error: operator and operand don't agree [literal]
  operator domain: real list
  operand:         int list
  in expression:
    f is
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    is

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

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   (c :: cs)
fn (i :: is)=> if i > 0
               then f(cs)(is)
               else c :: [2.2] @ (f(is)(cs))

the same program variables have type conflicts in different sites
    from the first expression
c: 'b
cs: 'b list

    but from the second expression
c: real
cs: int list
```

$\mathcal{IEI}$ algorithm reports:
Same as $\mathcal{U}$AE report.

E.2.3 Example 3

```
fun m f (h::t) = f h::f t ;
```

$\mathcal{W}$ algorithm reports:

```
stdIn:21.21-21.29 Error: operator and operand don't agree [circularity]
  operator domain: 'Z
  operand:         'Z list
  in expression:
    f t
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    t

Type inconsistent of requirement at the expression.
required type: 'i
actual type  : 'i list
```

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   f(h) :: (f(t))
```

```
operator domain type and operand type in conflict
operator domain type:'c list
operand         type:'c
```

$\mathcal{IEI}$ **algorithm reports:**
Same as $\mathcal{U}$AE report.

E.2.4 Example 4

```
let val f = fn x => let val y = x
                    in y 5 end in f 3 end;
```

$\mathcal{W}$ **algorithm reports:**

```
stdIn:22.2-22.57 Error: operator and operand don't agree [literal]
operator domain: int -> 'Z
operand:         int
in expression:
  f 3
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    3

Type inconsistent of requirement at the expression.
required type: int -> 'a
actual type  : int
```

$\mathcal{U}$AE **algorithm reports:**
Fails and switch to $\mathcal{IEI}$ alg.
$\mathcal{IEI}$ **algorithm reports:**

```
================================================
Possible errors in function:
f
================================================
-------------------- Error --------------------
Error in expression:
   f

Type inconsistent of requirement at the expression.
```

```
required type: int -> 'd
actual type  : (int -> 'e) -> 'e

================================================
Possible errors in argument:
3
================================================
-------------------- Error --------------------
Error in expression:
   3

Type inconsistent of requirement at the expression.
required type: int -> 'f
actual type  : int
```

E.2.5 Example 5

```
fn x => let val y = (fn z=> let val _ = x z
                             in (fn w=> w ) end)
         in (y 5, y true) end;
```

$\mathcal{W}$ algorithm reports:

```
  stdIn:23.7-23.20 Error: operator and operand don't agree [literal]
  operator domain: int
  operand:         bool
  in expression:
    y true
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    true

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

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   y(5)
(y(true ))

the same program variables have type conflicts in different sites
```

```
    from the first expression
y: int -> 'b -> 'b
x: int -> 'c

    but from the second expression
y: bool -> 'd -> 'd
x: bool -> 'c
```

$\mathcal{IEI}$ algorithm reports:
Same as $\mathcal{U}$AE report.

E.2.6 Example 6

```
fn x=> let val f = fn y => y x
        in  (f (fn z => z))(f (fn u => fn v=> u)) end ;
```

$\mathcal{W}$ algorithm reports:

```
stdIn:25.9-25.78 Error: operator is not a function [circularity]
  operator: 'Z
  in expression:
    (f (fn z => z)) (f (fn u => (fn <rule>)))
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    u

Type inconsistent of requirement at the expression.
required type: 'J
actual type  : ('I -> 'J) -> 'c
```

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   f(fn z=>z)
f(fn u=>fn v=>u)

the same program variables have type conflicts in different sites
    from the first expression
f: ((('c -> 'd) -> 'e) -> 'f) -> 'f
x: ('c -> 'd) -> 'e

    but from the second expression
f: ('d -> 'g) -> 'g
x: 'd
```

$\mathcal{IEI}$ algorithm reports:
Same as $\mathcal{U}$AE report.

E.2.7 Example 7

```
fn z=> let val x = z
        in (let val y = z 1 in x true end) end;
```

$\mathcal{W}$ algorithm reports:

```
stdIn:26.27-26.56 Error: operator and operand don't agree [literal]
  operator domain: int
  operand:         bool
  in expression:
    x true
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    true

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

$\mathcal{U}$AE algorithm reports:

Fails and switch to $\mathcal{IEI}$ alg.

$\mathcal{IEI}$ algorithm reports:

```
fn z=> let val x = z
        in (let val y = z 1 in x true end) end;

================================================
Possible errors in function:
x
================================================
-------------------- Error --------------------
Error in expression:
   x
Type inconsistent of requirement at the expression.
required type: bool -> 'c
actual type  : int -> 'd

================================================
Possible errors in argument:
true
================================================
-------------------- Error --------------------
Error in expression:
```

```
   true

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

E.2.8 Example 8

```
   fun f y (h::t) = t y;
```

$\mathcal{W}$ algorithm reports:

```
stdIn:28.21-28.24 Error: operator is not a function [tycon mismatch]
  operator: 'Z list
  in expression:
    t y
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    t

Type inconsistent of requirement at the expression.
required type: 'a -> 'g
actual type  : 'i list
```

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   (h :: t)
t(y)

the same program variables have type conflicts in different sites
    from the first expression
t: 'f list

    but from the second expression
t: 'b -> 'a
```

$\mathcal{IEI}$ algorithm reports:
Same as $\mathcal{U}$AE report.

E.2.9 Example 9

```
 fun f5 0 n = []
   | f5 m n = m ^ n ::(f5 (m-1));
```

$\mathcal{W}$ algorithm reports:

```
stdIn:29.41-29.46 Error: operator and operand don't agree [literal]
  operator domain: string * string
  operand:         string * int
  in expression:
    m - 1
stdIn:29.15-29.47 Error: types of rules don't agree [literal]
  earlier rule(s): int * 'Z -> 'Y list
  this rule: string * string -> string list
  in rule:
    (m,n) => m ^ n :: f5 (m - 1)
stdIn:29.2-29.47 Error: right-hand-side of clause doesn't agree
with function result type [tycon mismatch]
  expression:  'Z -> 'Y list
  result type:  string list
  in declaration:
    f5 = (fn arg => (fn <pat> => <exp>))
stdIn:29.43 Error: overloaded variable not defined at type
  symbol: -
  type: string
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    m
Type inconsistent of requirement at the expression.
required type: string
actual type  : int

Error in expression:
    f5

Type inconsistent of requirement at the expression.
required type: '176 -> 'Y list
actual type  : int -> 'W -> 'Y list
```

$\mathcal{U}$AE algorithm reports:

```
Type conflicts in expressions:
   m ^ n
(f5(m - 1))

the same program variables have type conflicts in different sites
    from the first expression
m: string

    but from the second expression
m: int
```

$\mathcal{IEI}$ **algorithm reports:**
Same as $\mathcal{U}$AE report.

E.2.10 Example 10

```
fun f5 x  = if x>3 then x else 1.1 ;
```

$\mathcal{W}$ **algorithm reports:**

```
stdIn:31.15-31.37 Error: types of rules don't agree [literal]
  earlier rule(s): bool -> int
  this rule: bool -> real
  in rule:
    false => 1.1
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    1.1

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

$\mathcal{U}$AE **algorithm reports:**

```
Type conflicts in expressions:
   x > 3
   x

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

    but from the second expression
x: real
```

$\mathcal{IEI}$ **algorithm reports:**
Same as $\mathcal{U}$AE report.

E.2.11 Example 11

```
fun f4 [] = []
  | f4 (0::t) = f4 t
  | f4 (h::t) = h/2.0::f4 t;
```

$\mathcal{W}$ **algorithm reports:**

```
stdIn:32.53-32.64 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    h / 2.0
stdIn:32.54 Error: overloaded variable not defined at type
  symbol: /
  type: int
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    h

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

$\mathcal{U}$AE **algorithm reports:**
Fails and switch to $\mathcal{IEI}$ alg.
$\mathcal{IEI}$ **algorithm reports:**

```
Error in expression:
   0

Type inconsistent of requirement at the expression.
required type: real
actual type  : int
Error in expression:
    /

Error at use of the operator.
 required type: int * real -> 'b
operator type: real * real -> real

Error in expression:
   (h, 2.0)

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

Type conflicts in expressions:
   fn []=>[]
  | (0 :: t)=>f4(t)
fn (h :: t)=>h / 2.0 :: (f4(t))
```

```
the same program variables have type conflicts in different sites
    from the first expression
f4: int list -> 'c list

    but from the second expression
f4: real list -> real list
```

E.2.12 Example 12

```
fun fail3 p nil = p + p
  | fail3 p (h::t) = if p true then [h] else t ;
```

$\mathcal{W}$ algorithm reports:

```
stdIn:33.19-33.69 Error: types of rules don't agree [tycon mismatch]
  earlier rule(s): (bool -> bool) * 'Z list -> bool -> bool
  this rule: (bool -> bool) * 'Z list -> 'Z list
  in rule:
    (p,h :: t) =>
      (case (p true)
        of true => h :: nil
         | false => t)
stdIn:33.21 Error: overloaded variable not defined at type
  symbol: +
  type: bool -> bool
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    p

Type inconsistent of requirement at the expression.
required type: 'h -> bool
actual type  : NumStr11

Error in expression:
    h

elements in a list have error types
   from this  element,  list type:   'j list
              required  list type:   NumStr11

Error in expression:
    t
```

```
Type inconsistent of requirement at the expression.
required type: NumStr11
actual type  : 'i list
```

$\mathcal{U}$AE algorithm reports:

Fails and switch to $\mathcal{IEI}$ alg.
$\mathcal{IEI}$ algorithm reports:

```
Error in expression:
    +
Error at use of the operator.
 required type: 'a -> 'b list
operator type: NumStr'c * NumStr'c -> NumStr'c

Error in expression:
   p

Type inconsistent of requirement at the expression.
required type: 'd -> bool
actual type  : NumStr'e
Error in expression:
   h

elements in a list have error types
   from the element(s), should be of type:   'f list
   but        required               type:   NumStr'e

Error in expression:
   t

Type inconsistent of requirement at the expression.
required type: NumStr'e
actual type  : 'g list
```

E.2.13 Example 13

```
 fun f 1 x = 0.0
   | f n x = (n*x) + (f (n-1) x);
```

$\mathcal{W}$ algorithm reports:

```
stdIn:34.14-34.47 Error: types of rules don't agree [literal]
```

```
  earlier rule(s): int * int -> real
  this rule: int * int -> int
  in rule:
    (n,x) => n * x + (f (<exp> - <exp>)) x
stdIn:34.2-34.47 Error: right-hand-side of clause
doesn't agree with function result type [literal]
  expression:  int -> real
  result type:  int -> int
  in declaration:
    f = (fn arg => (fn <pat> => <exp>))
```

$\mathcal{M}$ **algorithm reports:**

```
Error in expression:
    n

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

$\mathcal{U}$AE **algorithm reports:**
Fails and switch to $\mathcal{IEI}$ alg
$\mathcal{IEI}$ **algorithm reports:**

```
Error in expression:
   0.0

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

Error in expression:
    +

Error at use of the operator.
 required type: int * 'b -> real
operator type: NumStr'c * NumStr'c -> NumStr'c

Error in expression:
   (n * x, f(n - 1)(x))

Type inconsistent of requirement at the expression.
required type: real * real
actual type  : int * 'b
```

E.2.14 Example 14

```
fun fac n = if n = 0 then 1 else n* fac(n=1);
```

$\mathcal{W}$ algorithm reports:

```
stdIn:35.36-35.47 Error: operator and operand don't agree [literal]
  operator domain: int
  operand:         bool
  in expression:
    fac (n = 1)
```

$\mathcal{M}$ algorithm reports:

```
Error in expression:
    =
Error at use of the operator.
 required type: 'k -> int
operator type: ''a93 * ''a93 -> bool
```

$\mathcal{U}$AE algorithm reports:

Fails and switch to $\mathcal{IEI}$ alg.

$\mathcal{IEI}$ algorithm reports:

```
Error in expression:
    =

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

Type conflicts in expressions:
   fac
fn n=> if n = 0 then 1 else n * (fac(n = 1))

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

    but from the second expression
fac: bool -> int
```

E.2.15 Example 15

```
fn x => if x then x+1 else x-2.2
```

$\mathcal{W}$ reports

```
stdIn:18.28-18.33 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    x - 2.2
stdIn:18.9-18.33 Error: case object and rules don't agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case x
      of true => x + 1
       | false => x - 2.2)
```

$\mathcal{M}$ reports

```
Error in expression:
    x

Type inconsistent of requirement at the expression.
required type: NumStr5
actual type  : bool
```

$\mathcal{U}_{\mathbf{AE}}$ reports

```
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^


expression after 'then' and expression after 'else' have different types
 exp after 'then' :  int
 exp after 'else' :  real

-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if ^ then x + 1 else x - 2.2

 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2
the same program variables have type conflicts in different sites
    from the first sub-expression
x: bool
```

```
    but from the second sub-expression
x: int

-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if ^ then x + 1 else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^

the same program variables have type conflicts in different sites
    from the first sub-expression
x: bool

    but from the second sub-expression
x: real

-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^

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

    but from the second sub-expression
x: real
```

$\mathcal{IEI}$ **algorithm reports**

```
Error in expression:
    +

Error at use of the operator.
 required type: real * int -> real
operator type: real * real -> real

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

Type inconsistent of requirement at the expression.
```

```
required type: real * real
actual type  : real * int

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

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

-------------------- Error --------------------
Error in expression:
   (x, 2.2)

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

E.2.16 Example 16

```
fn x => (x+1) (if x then x+1 else x-2.2)
```

$\mathcal{W}$ **reports**

```
stdIn:2.30-2.35 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    x - 2.2
stdIn:2.10-2.36 Error: case object and rules don't agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case x
      of true => x + 1
       | false => x - 2.2)
stdIn:2.4-2.36 Error: operator is not a function [literal]
  operator: int
  in expression:
    (x + 1)
        (case x
          of true => x + 1
           | false => x - 2.2)
```

$\mathcal{M}$ reports

```
Error in expression:
    +
Error at use of the operator.
 required type: 'e -> 'd -> 'c
operator type: NumStr6 * NumStr6 -> NumStr6

-------------------- Error --------------------
Error in expression:
    x

Type inconsistent of requirement at the expression.
required type: NumStr10
actual type  : bool
```

$\mathcal{U}$AE reports

```
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^


expression after `then' and expression after `else' have different types
 exp after `then' :  int
 exp after `else' :  real

-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if ^ then x + 1 else x - 2.2

 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2


the same program variables have type conflicts in different sites
    from the first sub-expression
x: bool

    but from the second sub-expression
x: int
```

```
-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if ^ then x + 1 else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^


the same program variables have type conflicts in different sites
    from the first sub-expression
x: bool

    but from the second sub-expression
x: real

-------------------- Error --------------------
Type conflicts in expressions:
 if x then x + 1 else x - 2.2
 if x then ^^^^^ else x - 2.2

 if x then x + 1 else x - 2.2
 if x then x + 1 else ^^^^^^^

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

    but from the second sub-expression
x: real


-------------------- Error --------------------
Type conflicts in expressions:
(x + 1) if x then x + 1 else x - 2.2
(^^^^^) if x then x + 1 else x - 2.2

operator is not a function
which with type :  int

-------------------- Error --------------------
Type conflicts in expressions:
(x + 1) if x then x + 1 else x - 2.2
(^^^^^) if x then x + 1 else x - 2.2

(x + 1) if x then x + 1 else x - 2.2
```

```
(x + 1)^^^^^^^^^^^^^^^^^^^^^^^^^^^^^


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

    but from the second sub-expression
x: real
```

$\mathcal{IEI}$ **algorithm reports**

```
Error in expression:
    +
Error at use of the operator.
 required type: real * int -> real
operator type: real * real -> real

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

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

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

Error at use of the operator.
 required type: real * real -> int
operator type: NumStr'b * NumStr'b -> NumStr'b

-------------------- Error --------------------
Error in expression:
   (x, 2.2)

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

E.3 Observations

1. The $\mathcal{M}$ reports more precise error sites than other algorithms by reporting error site at constants, variables, and operators. The $\mathcal{M}$ may report large error sites at lambda abstract sub-expression, but this does not happen in our examples.

2. For example 1, 2, 5, 15 and 16, $\mathcal{IEI}$ gives better error message than other algorithms

3. For errors in LET bound variables, all algorithms report their application as error sites, but the type conflicts may in the definition of the variables.

4. $\mathcal{U}$AE and $\mathcal{IEI}$ report type errors in a conflicting pairs.

APPENDIX F
Examples of $\mathcal{H}$ algorithm type explanation

F.1 Chain of reasoning

```
fun f w x y z = [x = w, x=y, y = z, z=w, y = "ha"];


"fun f w x y z = [x = w, x = y, y = z, z = w, y = "ha"]",
has function type: string -> string -> string -> string -> bool list
Argument 1: string
1: "w" has the same type as "x" in "x = w"
   - arguments of "=" have the same Equality types
2: "x" has the same type as "y" in "x = y"
   - arguments of "=" have the same Equality types
3: "y" and ""ha"" have the same type in "y = "ha""
   - arguments of "=" have the same Equality types
4: ""ha"": string

Argument 2: string
5: "x":  string - see 2 above

Argument 3: string
6: "y":  string - see 3 above

Argument 4: string
7: "z" has the same type as "y" in "y = z"
    - arguments of "=" have the same Equality types
8: "y":  string - see 3 above

Result: bool list
9: "[x = w, x = y, y = z, z = w, y = "ha"]", is
     10: "y = "ha"" has the same type as  bool
11:  list of "y = "ha"": bool list
```

F.2 Shortest explanation

```
fun f w x y z = [x = w, x=y, z = w, y = z, y = "ha"];

"fun f w x y z = [x = w, x = y, z = w, y = z, y = "ha"]",
has function type:
string -> string -> string -> string -> bool list
Argument 1: string
1: "w" has the same type as "x" in "x = w"
   - arguments of "=" have the same Equality types
2: "x" has the same type as "y" in "x = y"
```

```
   - arguments of "=" have the same Equality types
3: "y" and ""ha"" have the same type in "y = "ha""
   - arguments of "=" have the same Equality types
4: ""ha"": string

Argument 2: string
5: "x":  string - see 2 above

Argument 3: string
6: "y":  string - see 3 above

Argument 4: string
7: "z" has the same type as "y" in "y = z"
    - arguments of "=" have the same Equality types
8: "y":  string - see 3 above

Result: bool list
9: "[x = w, x = y, z = w, y = z, y = "ha"]", is
     10: "y = "ha"" has the same type as  bool
11:  list of "y = "ha"": bool list
```

F.3 Secect shortest possible explanation

```
fun f w x y z = [ x=y, z = w, y = z, w = "yo",  y = "ha"];

"fun f w x y z = [x = y, z = w, y = z, w = "yo", y = "ha"]",
has function type: string -> string -> string -> string -> bool list
Argument 1: string
1: "w" and ""yo"" have the same type in "w = "yo""
   - arguments of "=" have the same Equality types
2: ""yo"": string

Argument 2: string
3: "x" has the same type as "y" in "x = y"
   - arguments of "=" have the same Equality types
4: "y" and ""ha"" have the same type in "y = "ha""
   - arguments of "=" have the same Equality types
5: ""ha"": string

Argument 3: string
6: "y":  string - see 4 above

Argument 4: string
7: "z" has the same type as "w" in "z = w"
   - arguments of "=" have the same Equality types
8: "w":  string - see 1 above
```

```
Result: bool list
9: "[x = y, z = w, y = z, w = "yo", y = "ha"]", is
     10: "y = "ha"" has the same type as  bool
11:  list of "y = "ha"": bool list
```

F.4 Explain by function applications

```
fun f F H G x a= [F [H] [3] (G x), F a a a];
```

```
"fun f F H G x a = [F [H] [3](G x), F a a a]",
has function type: (int list -> int list -> int list
-> 'a) -> int -> ('b -> int list) -> 'b -> int list -> 'a list
Argument 1: int list -> int list -> int list -> 'a
1: "F": has function type:
        int list -> int list -> int list -> 'a
     Argument 1: int list
     2:  the Operand of "F [H]""[H]", is
          3: "H" has the same type as "[H]", whose Element is
             the Operand of "F [H]", is the same type as
             the Operand of "F a", is the same type as
             "a" the Last Operand of "F a a",
             is the same type as
             the Last Operand of "F [H] [3]", is the same type as
             "[3]", whose Element is "3": int
     4:  list of "H": int list
      "F [H]" has the same type as
     Argument 2: int list
     5:  the Last Operand of "F [H] [3]"list of  int
     6: : int list - see 3 above
      "F [H] [3]" has the same type as
     Argument 3: int list
     7:  the Last Operand of "F [H] [3](G x)", is the same type as
         the Last Operand of "F a a a", is the same type as
         "a",  the Operand of "F a", is the same type as
         the Operand of "F [H]"
         list of  int
     8: : int list - see 2 above

     Result: 'a
     9:  "F [H] [3](G x)" has the same type as
         "F a a a" has the same type as 'a
Argument 2: int
10: "H""[H]", whose Element is
    the Operand of "F [H]", is the same type as
    the Operand of "F a", is the same type as
    "a" the Last Operand of "F a a", is the same type as
```

```
    the Last Operand of "F [H] [3]", is the same type as
    "[3]", whose Element is "3": int

Argument 3: 'b -> int list
11: "G": has function type: 'b -> int list
     Argument 1: 'b
     12:  the Operand of "G x", is the same type as
          "x": 'b - no operation on "x"

     Result: int list
     13:  "G x" has the same type as
          the Last Operand of "F [H] [3](G x)"
          list of  int
     14: : int list - see 7 above
Argument 4: 'b
15: "x": 'b - no operation on "x" - see 12 above

Argument 5: int list
16: "a": list of  int
17: : int list - see 3 above

Result: 'a list
18: "[F [H] [3](G x), F a a a]", is
     19: "F a a a": 'a - no operations on "F a a a"
20:  list of "F a a a": 'a list
```

F.5 Explain by overloaded operators

```
fun f x y z = ( x + y, y > z, floor( z));

"fun f x y z = (x + y, y > z,  floor  z)",
has function type: real -> real -> real -> real * bool * int
Argument 1: real
1: "x" has the same type as "y" in "x + y"
   - arguments of "+" have the same Number types
2: "y" has the same type as "z" in "y > z"
   - arguments of ">" have the same Number types
3: "z" the Operand of " floor  z", is the same type as  real

Argument 2: real
4: "y":  real - see 2 above

Argument 3: real
5: "z":  real - see 3 above

Result: real * bool * int
```

```
6:  "(x + y, y > z,  floor  z)",
    is a Tuple type: real * bool * int
Tuple Element 1: real
  7: "x + y" and "y" have the same type
     - result/arguments of "+" have the same Number types
  8: "y":  real - see 2 above
Tuple Element 2: bool
  9:  "y > z",
  (because the result of ">" has the  bool types)
  has the type  bool
Tuple Element 3: int
  10:  " floor  z" has the same type as  int
```

F.6 Explain by known type operator

```
fun f x y z = (x+y,y*z,z/2.0);
```

```
"fun f x y z = (x + y, y * z, z / 2.0)",
has function type: real -> real -> real -> real * real * real
Argument 1: real
1: "x" has the same type as "y" in "x + y"
   - arguments of "+" have the same Number types
2: "y" has the same type as "z" in "y * z"
   - arguments of "*" have the same Number types
3: "z" and "2.0" have the same type in "z / 2.0"
   - arguments/result of "/" have the same Real types

Argument 2: real
4: "y":  real - see 2 above

Argument 3: real
5: "z":  real - see 3 above

Result: real * real * real
6:  "(x + y, y * z, z / 2.0)",
is a Tuple type: real * real * real
Tuple Element 1: real
  7: "x + y" and "y" have the same type
     - result/arguments of "+" have the same Number types
  8: "y":  real - see 2 above
Tuple Element 2: real
  9: "y * z" and "z" have the same type
     - result/arguments of "*" have the same Number types
  10: "z":  real - see 3 above
Tuple Element 3: real
  11: "z / 2.0" has the same type as "z" and "2.0"
```

```
    - result/arguments of "/" have the same Real types
  12: "2.0": real
```

F.7 No repeated explanations

```
fun f x = x | f y = y+1;
```

```
"fun f x = x
  |  f y = y + 1", has function type: int -> int
Argument 1: int
1: "x" and "y" have the same type - same LHS pattern position
2: "x" and "y + 1" have the same type - same RHS result
3: "y + 1" has the same type as "y" and "1"
   - result/arguments of "+" have the same Number types
4: "1": int

Result: int
5:  int - see 2 above
```

F.8 Explain by 2D relations

```
fun f 1 y = y | f x 2 = x;
```

```
"fun f 1 y = y
  |  f x 2 = x", has function type: int -> int -> int
Argument 1: int
1: "1" and "x" have the same type - same LHS pattern position
2: "1": int

Argument 2: int
3: "y" and "2" have the same type - same LHS pattern position
4: "2": int

Result: int
5: "y" and "x" have the same type - same RHS result
6:  int - see 3 above
```

F.9 Index the program variables that have the same name

```
fun f 1 y z = z | f x 2 z = x | f x y z = 3;
```

```
"fun f 1 y z = z
```

```
  |  f x 2 z2 = x
  |  f x3 y3 z3 = 3", has function type: int -> int -> int -> int
Argument 1: int
1: "1", "x" and "x3" have the same type
    - same LHS pattern position
2: "1": int

Argument 2: int
3: "y", "2" and "y3" have the same type
    - same LHS pattern position
4: "2": int

Argument 3: int
5: "z", "z2" and "z3" have the same type
    - same LHS pattern position
6: "z", "x" and "3" have the same type
    - same RHS result
7: "3": int

Result: int
8:  int - see 6 above
```

F.10 List type explanation

```
fun sum [] = 0 | sum (h::t) = h+sum t;

"fun sum [] = 0
  |  sum (h :: t) = h + (sum t)",
has function type: int list -> int
Argument 1: int list
1: "[]" and "(h :: t)" have the same type
    - same LHS pattern position
2: list of "h", "t" and "(h :: t)" have same type
   -operands/result of "::"
     3: "h" has the same type as "h + (sum t)"
         - argument/result of "+" have the same Number types
     4: "0" and "h + (sum t)" have the same type
         - same RHS result
     5: "0": int
6:  list of "h": int list

Result: int
7:  int - see 4 above
```

F.11 Conditional expression

```
fun sum l = if l=[] then 0 else hd l+sum (tl l);

"fun sum l =  if l = [] then 0 else  hd  l + (sum( tl  l))",
has function type: int list -> int
Argument 1: int list
1: "l" the Operand of " hd  l"
     2: " hd  l" has the same type as " hd  l + (sum( tl  l))"
        - arguments/result of "+" have the same Number types
     3:  the ELSE branch " hd  l + (sum( tl  l))"
         has the same type as the THEN branch "0":  int
4: "l": int list

Result: int
5:  if l = [] then 0 else  hd  l + (sum( tl  l))
    has the same type as the THEN branch "0":  int
```

F.12 Minium type variable use

```
fun len [] = 0 | len (h::t) = 1+len t;


"fun len [] = 0
  |  len (h :: t) = 1 + (len t)",
has function type: 'a list -> int
Argument 1: 'a list
1: "[]" and "(h :: t)" have the same type
    - same LHS pattern position
2: list of "h", "t" and "(h :: t)" have same type
   - operands/result of "::"
     3: "h": 'a - no operations on "h"
4:  list of "h": 'a list

Result: int
5: "0" and "1 + (len t)" have the same type
    - same RHS result
6: "0": int
```

F.13 Locate the ground type subexpression

```
fun len l = if l=[1] then 0 else 1+len (tl l);


"fun len l =  if l = [1] then 0 else 1 + (len( tl  l))",
has function type: int list -> int
```

```
Argument 1: int list
1: "l" and "[1]" have the same type in "l = [1]"
   - arguments of "=" have the same Equality types
     2: Element of "[1]" has the same type as "1": int
3: "[1]": int list

Result: int
4:  IF  if l = [1] then 0 else 1 + (len( tl  l))
    has the same type as the THEN branch "0":  int
```

F.14 2D search ground type subexpression

```
"fun count (n, z, p) [] = (n, z, p)
  |  count (n2, z2, p2) (0 :: t) = count(n2, z2 + 1, p2) t
  |  count (n3, z3, p3) (h :: t3) =
             if h < 0
             then count(n3 + 1, z3, p3) t3
             else count(n3, z3, p3 + 1) t3",
has function type: int * int * int -> int list -> int * int * int
Argument 1: int * int * int
1: "(n, z, p)", "(n2, z2, p2)" and "(n3, z3, p3)" have the same type
   - same LHS pattern position
2:  is a Tuple type: int * int * int
Tuple Element 1: int
  3: for "n3" in "(n3, z3, p3)"
  4: "n3" and "1" have the same type in "n3 + 1"
     - arguments/result of "+" have the same Number types
  5: "1": int
Tuple Element 2: int
  6: for "z2" in "(n2, z2, p2)"
  7: "z2" and "1" have the same type in "z2 + 1"
     - arguments/result of "+" have the same Number types
  8: "1": int
Tuple Element 3: int
  9: for "p3" in "(n3, z3, p3)"
  10: "p3" and "1" have the same type in "p3 + 1"
      - arguments/result of "+" have the same Number types
  11: "1": int

Argument 2: int list
12: "[]", "(0 :: t)" and "(h :: t3)" have the same type
    - same LHS pattern position
13: list of "0", "t" and "(0 :: t)" have same type
    - operands/result of "::"
    14: "0":  int
15:  list of "0": int list
```

```
Result: int * int * int
16:  at Row 1 "(n, z, p)" is the same as Argument1 "(n, z, p)",
     has type: int * int * int - see 1 above
```

F.15 Select the simplest type explanation

```
fun fac 0 = 1 | fac n = n*fac (n-1);
```

```
"fun fac 0 = 1
  |  fac n = n * (fac(n - 1))",
has function type: int -> int
Argument 1: int
1: "0" and "n" have the same type
    - same LHS pattern position
2: "0": int

Result: int
3: "1" and "n * (fac(n - 1))" have the same type
    - same RHS result
4: "1": int
```

F.16 Select the simpler explanation in conditional expression

```
fun fac n = if n=0 then 1 else n*fac (n-1);
```

```
"fun fac n =  if n = 0 then 1 else n * (fac(n - 1))",
has function type: int -> int
Argument 1: int
1: "n" and "1" have the same type in "n - 1"
   - arguments/result of "-" have the same Number types
2: "1": int

Result: int
3:  if n = 0 then 1 else n * (fac(n - 1)) has the same type as
    the THEN branch "1":  int
```

F.17 Explain nested conditional expression

```
fun comp x y = if x<y then "less" else if x>y then "greater" else if x>0
then "equalPos" else "equalNeg" ;
```

```
"fun comp x y =  if x < y then "less" else  if x > y
                 then "greater"
                 else  if x > 0
                       then "equalPos"
                       else "equalNeg"",
has function type: int -> int -> string
Argument 1: int
1: "x" and "0" have the same type in "x > 0"
   - arguments of ">" have the same Number types
2: "0": int

Argument 2: int
3: "y" has the same type as "x" in "x > y"
   - arguments of ">" have the same Number types
4: "x":  int - see 1 above

Result: string
5:  if x < y
    then "less"
    else  if x > y
          then "greater"
          else  if x > 0
                then "equalPos"
                else "equalNeg"  has the same type as
    the THEN branch ""less"":  string
```

F.18 Explain by 2D and known type operators

```
fun merge [] (h2::t2) = h2 |
    merge (h1::t1) [] = h1 |
    merge (h1::t1) (h2::t2) = if h1<h2
                              then h1/merge t1 (h2::t2)
                              else h2/merge (h1::t1) t2;
```

```
"fun merge [] (h2 :: t2) = h2
  |  merge (h1 :: t1) [] = h1
  |  merge (h13 :: t13) (h23 :: t23) =
       if h13 < h23
       then h13 / (merge t13(h23 :: t23))
       else h23 / (merge(h13 :: t13) t23)",
has function type: real list -> real list -> real
Argument 1: real list
1: "[]", "(h1 :: t1)" and "(h13 :: t13)" have the same type
   - same LHS pattern position
```

```
2: list of "h1", "t1" and "(h1 :: t1)" have same type
   -operands/result of "::"
     3: "h2", "h1" and " if h13 < h23
                         then h13 / (merge t13(h23 :: t23))
                         else h23 / (merge(h13 :: t13) t23)"
    have the same type - same RHS result
     4:  if h13 < h23
         then h13 / (merge t13(h23 :: t23))
         else h23 / (merge(h13 :: t13) t23)  has the same type as
        the ELSE branch "h23 / (merge(h13 :: t13) t23)": Real
        - result/arguments of "/" have the same Real types
5:  list of "h1": real list

Argument 2: real list
6: "(h2 :: t2)", "[]" and "(h23 :: t23)" have the same type
   - same LHS pattern position
7: list of "h2", "t2" and "(h2 :: t2)" have same type
   - operands/result of "::"
     8:  real - see 3 above
9:  list of "h2": real list

Result: real
10:  real - see 3 above
```

F.19 Type explanation of nested local definition

```
let val y = fn x => x in let val f = fn x => let val a1 = [x,y] in let
val a2 = x 3 in let val a3 = y 3.3 in "banana" end end end in f end end;
```

```
 LET result "let val f = fn x =>
               let val a1 = [x, y]
                in let val a2 = x 3
                    in let val a3 = y 3.3
                        in "banana"
                       end
                   end
               end in f
            end",
has the same type as
1:  LET result "f"
2: "f" and "fn x => let val a1 = [x, y]
                    in let val a2 = x 3
                        in let val a3 = y 3.3
                            in "banana"
                           end
                       end
```

```
                    end"
have the same type
- LHS and RHS of "val f = fn x => let val a1 = [x, y]
                                   in let val a2 = x 3
                                       in let val a3 = y 3.3
                                           in "banana"
                                          end
                                      end
                                  end"
3: "fn x => let val a1 = [x, y]
             in let val a2 = x 3
                 in let val a3 = y 3.3
                     in "banana"
                    end
                end
            end",
has function type: (int -> int) -> string
Argument 1: int -> int
4:  the Bound variable "x" and "y" have the same type
   -  Elements of list "[x, y]" have the same type
     5: "y" and "fn x => x" have the same type
        - LHS and RHS of "val y = fn x => x"
     6: "fn x => x", has function type: int -> int
     Argument 1: int
     7:  the Bound variable "x" has the same type as
     8:  the Operand of "x 3", is the same type as "3": int

     Result: int
     9:  Function body
         the Bound variable "x":  int  - see 4 above
Result: string
10:  Function body "let val a1 = [x, y]
                     in let val a2 = x 3
                         in let val a3 = y 3.3
                             in "banana"
                            end
                        end
                    end"
has the same type as
11:  LET result "let val a2 = x 3
                  in let val a3 = y 3.3
                      in "banana"
                     end
                 end",
has the same type as
12:  LET result "let val a3 = y 3.3
```

```
                 in "banana"
                end",
has the same type as
13:  LET result ""banana"":  string
```

F.20 Simplest explanation in pattern matching

```
fun sum [] = 0 | sum (h::t) = h + sum t | sum (0::t) = sum t;


"fun sum [] = 0
  |  sum (h :: t) = h + (sum t)
  |  sum (0 :: t3) = sum t3",
has function type: int list -> int
Argument 1: int list
1: "[]", "(h :: t)" and "(0 :: t3)" have the same type
   - same LHS pattern position
2: list of "0", "t3" and "(0 :: t3)" have same type
   - operands/result of "::"
     3: "0":  int
4:  list of "0": int list

Result: int
5: "0", "h + (sum t)" and "sum t3" have the same type
   - same RHS result
6: "0": int
```

APPENDIX G
Human type explanation samples

```
Please explanation the type checking of each program variable:

fun sum [] = 0 |  sum (h :: t) = h  +  (sum t);
1.
fun G x y z u w = [x = y, y = z, y = w, y = u, z = true];

2.
fun F x y z = [ (3,x), (4, y), (6, z), (4, 2.2)] ;

3.
fun F x y z = [ (3, x), (3, (4, y)), (3, (4, (5, z)))];

4.
fun F x y z = [ (3, x), (3, (4, (5, z))), (3, (4, y))];

5.
fun F H G x = [ [H], [3], G(x) ];

6.
fun F H G x = [ G(x), [H], [3] ];

7.
fun f H G J  =  let val  x = (4, H) in [ (3, G), x, (5, J) ] end;

8.
fun f F G x y = (F x ;
                 G y ;
                 if true then F else G;
                 F 3)

9.
fun ff x = (x 1, x 1 true )

10.
fun count nil (n, z, p ) = (n, z, p)
  | count (0::t) (n, z, p ) = count t (n, z+1, p)
  | count (h::t) (n, z, p ) = if h<0
                              then count t (n+1, z, p)
                              else count t (n, z, p+1);



11.
```

```
fun f2 [] = 0.0 | f2 (h::t) = h + (f2 t);

12.
fun s f x = f 1 + ( f x + 0.1 );

13.
fun sumfunc _ 0 = 0
  | sumfunc f n = (f n) + (sumfunc f (n-1));

14.
fun fff f = let fun m nil = nil
                  | m (x::xs) = f(x) :: m(xs)
             in m
            end;


15.
fun f7 [] = [] | f7 (h::t) = (h,h)::(f7 t);

16.
fun a [] = [] | a ((x,y)::t) = x::a t;

17.
fun f3 [] = []
  | f3 (0::t) = 0::(f3 t)
  | f3 (_::t) = f3 t ;
18.
fun f8 [] = [] | f8 (h::t) = [h]::(f8 t);

19.
fun a _ [] = [] | a f (h::t) = (2*(f h))::(a f t);

20 .
fun b [] = [] | b ((x,y,z)::t) = (x+y/z)::b t;

21.
fun d [] = [] | d ((p,q)::t) = (p q)::(d t);

22.
fun c _ [] = []
  | c f (h::t) = if f h then (f h)::(c f t) else c f t;


23.
fun d _ _ [] = []
  | d p f (h::t) = if p h
```

```
                   then (f h)::(d p f t)
                   else d p f t;


24.
fun e _ _ [] = []
  | e f1 f2 (h::t) = (f1 (f2 h))::(e f1 f2 t);


25.
fun f _ _ [] = []
  | f p1 p2 (h::t) = if p1 (p2 h)
                     then h::(f p1 p2 t)
                     else f p1 p2 t;


26.
fun g _ [] [] = []
  | g f (h1::t1) (h2::t2) = if f h1
                            then h2::(g f t1 t2)
                            else g f t1 t2 ;

27.
fun h _ _ [] [] = []
  | h f1 f2 (h1::t1) (h2::t2) = if f1 h1
                                then (f2 h2)::(h f1 f2 t1 t2)
                                else h f1 f2 t1 t2 ;

28.
fun c _ [] = [] | c f ((x,y)::t) = (f x y)::(c f t);


29.
fun e _ [] = [] | e f ((x,y)::t) = (f x,f y)::(e f t);


30 .
fun g f ((x1,y1)::t1) ((x2,y2)::t2) =
       if f x1 x2
       then (y1,y2)::g f t1 t2
       else g f t1 t2
  | g _ _ _ = [];


31.
fun dd (p, nil) = nil
```

```
  | dd (p, h::t) = if p = hd h
                   then t
                   else h::dd ( p, t) ;
```

```
32.
fun s x y z = (x z) (y z);
```

```
33.
fun l f x = f ( f x);
```

APPENDIX H
Human type-error explanation experiment samples

[4]

```
1. fun fail3 p nil = p + p |
       fail3 p (h::t) = if p true then [h] else t

The system explains:

 fun fail3 p [] = p + p
  |  fail3 p2 (h :: t) =  if p2 true then [h] else t
found expressions that should have the same type are infered different types:

   there's one expression type:   "p + p" and " if p2 true then [h] else t"
   have the same type - same RHS result
1: "p + p" has the same type as "p" and "p"
    - result/arguments of "+" have the same Number types
2:  Num'a (int or real, non-resoved.)

   other   expressions    type:   "p + p" and " if p2 true then [h] else t"
   have the same type - same RHS result
1:  "if p2 true then [h] else t" and the THEN branch "[h]"
   have the same type, has the same type as
2: "[h]", is
     3: "h": 'b - no operations on "h"
4:  list of "h": 'b list



2.
fun f 1 x = 0.0 |
     f n x = (n*x) + (f (n-1) x)

The system explains:

-------------------- Error --------------------
 fun f 1 x = 0.0
  |  f n x2 = n * x2 + (f(n - 1) x2)
Type conflicts are found in expression:
   "0.0" and "n * x2 + (f(n - 1) x2)"

found expressions that should have the same type are infered different types:

   there's one expression type:   "0.0" and "n * x2 + (f(n - 1) x2)"
```

[4] Some simple samples are given with $\mathcal{H}$ implementation output.

```
   have the same type - same RHS result
1: "0.0": real

   other   expressions    type:   "0.0" and "n * x2 + (f(n - 1) x2)"
   have the same type - same RHS result
1: "n * x2" and "f(n - 1) x2" have the same type as "n * x2 + (f(n - 1) x2)"
   - arguments/result of "+" have the same Number types
2: "n * x2" and "n" have the same type
   - result/arguments of "*" have the same Number types
3: "n" and "1" have the same type in "n - 1"
   - arguments/result of "-" have the same Number types
4: "1": int


3.
fn s1 => fn s2 => size s1^s2

The system explains:

-------------------- Error --------------------
Type conflicts are found in expression:
   ( size  s1) ^ s2

as requirement of operator "^",
the Left operand " size  s1" should has the type:  string
"( size  s1) ^ s2": STRING
- result/arguments of "^" have the same String types
but actually has type:  int
 " size  s1" has the same type as  int


4.
fn x => size x andalso x


The system explains:
-------------------- Error --------------------
Type conflicts are found in expression:
    size  x

At the both sides of ‘andalso’ or ‘orelse’
should be expression of type bool
but I found on the left hand side have type:
" size  x" has the same type as  int
```

```
5.
fn p => fn q => (not p) q


Type conflicts are found in expression:
    not  p q

operator is not a function
which with type :   " not  p" has the same type as  bool


 6.
 fun f1 1 = true |
     f1 2 = true |
     f1 3.0 = 3.0


The system explains:
-------------------- Error --------------------
Type conflicts are found in expression:
   "1", "2" and "3.0"


found expressions that should have the same type are infered different types:

   there's one expression type:   "1", "2" and "3.0" should have the same type
   - same LHS pattern position
1: "1": int

   other   expressions    type:   "1", "2" and "3.0" should have the same type
   - same LHS pattern position
1: "3.0": real


 7.
 fun f5 x  = if x>3 then x else 1.1


The system explains:
-------------------- Error --------------------
Type conflicts are found in expression:
    if x > 3 then x else 1.1


found expressions that should have the same type are infered different types:
```

```
   there’s one expression type:
    "x" and "3" have the same type in "x > 3"
    - arguments of ">" have the same Number types
1: "3": int

   other   expressions    type:
   the THEN branch "x", has the same type as
1:  the ELSE branch "1.1":  real
```

```
 8.
 [fn x => x+1,fn x => x+1.0,fn x => x^x]
```

```
 9.
 [1,2,3]::[4,5]
```

```
 10.
 fun f6 nil = (0.0, 0.0) |
     f6 (h::t) = (h,h/2.0)::f6 t
```

```
 11.
 fun f2 v [] = v |
     f2 v (h::t) = v+(h::f2 t)
```

```
 12.
 fun f2 v [] = v |
     f2 v (h::t) = (v+h)::(f2 t)
```

```
The system explains:
-------------------- Error --------------------
 fun f2 v [] = v
  |  f2 v2 (h :: t) = v2 + h :: (f2 t)

Type conflicts are found in expression:
    f2 v2 (h :: t) = v2 + h :: (f2 t)


found expressions that should have the same type are infered different types:

   there’s one expression type:   "v" and "v2" have the same type
   - same LHS pattern position
1: "v" and "v2 + h :: (f2 t)" have the same type - same RHS result
```

```
2: list of "v2 + h", "f2 t" and "(v2 + h :: f2 t)" have same type
   -operands/result of "::"
     3: "v2 + h" has the same type as  Num'a (int or real, non-resoved.)
4:  list of "v2 + h": Num'a list

   other   expressions    type:   "v2" and "h" have the same type in "v2 + h"
   - arguments/result of "+" have the same Number types
1:  Num'a (int or real, non-resoved.)
```

```
 13.
 fun f3 [] = [] |
     f3 (h::t) = [h]::f3 [t]
```

```
 14.
 fun f4 [] = [] |
     f4 (0::t) = f4 t |
     f4 (h::t) = h/2.0::f4 t
```

```
 15.
 fun f5 0 n = [] |
     f5 m n = m ^ n ::(f5 (m-1))
```

```
 16.
 fun f p q r = (p/q, q div r)
```

```
The system explains:
-------------------- Error --------------------
Type conflicts are found in expression:
   (p / q, q div r)


found expressions that should have the same type are infered different types:

   there's one expression type:   "q" and "r" have the same type in "q div r"
   - arguments/result of "div" have the same Int types

   other   expressions    type:   "q" and "p" have the same type in "p / q"
```

```
   - arguments/result of "/" have the same Real types


 17.
 fun f10 f [] = true |
     f10 f (h::t) = (f h) +  (f10 f t)



 18.
 fun f [] = false |
     f [v] = v |
     f (h1::h2::t) = h1>h2

The system explains:

fun f [] = false
  |  f [v] = v
  |  f (h1 :: (h2 :: t)) = h1 > h2
-------------------- Error --------------------
Type conflicts are found in expression:
    f (h1 :: (h2 :: t)) = h1 > h2


found expressions that should have the same type are infered different types:

   there's one expression type:   "h1" has the same type as an Element
   in the tail of "(h1 :: (h2 :: t))",
   list of "h2", "t" and "(h2 :: t)" have same type
   -operands/result of "::"
   list of "h1", "(h2 :: t)" and "(h1 :: (h2 :: t))" have same type
   -operands/result of "::"
   "[]", "[v]" and "(h1 :: (h2 :: t))" have the same type
   - same LHS pattern position
1: "[]", "[v]" and "(h1 :: (h2 :: t))" have the same type
   - same LHS pattern position
2: "[v]", whose Element is "v"
   "false", "v" and "h1 > h2" have the same type
   - same RHS result
3: "false": bool

   other   expressions    type:   "h1" and "h2" have the same type
        in "h1 > h2"
        - arguments of ">" have the same Number types
          Number'b (int or real, non-resoved.)
```

```
19.
fun m f [1] = [1] |
    m f (h::t) = if h> 0.0
                 then (f h) :: (m f t)
                 else m f t
```

```
20.
fun f (h::0::t) = [ h + (f t) ]
```

```
21.
fun f x = if f(x) then 3 else x
```

```
22.
fun ff p h (h::t) = if p h
                    then ff h + p ff t
                    else p ff t
```

```
23.
fun fff f1 f2 [0] ["hello"] = [] |
    fff f1 f2 (h1::t1) (h2::t2) = if f1 h1
                                  then ((f2 h2)+1)::(fff f1 f2 t2 t1)
                                  else fff f1 f2 t1 t2
```

```
24.
fun e _ [] = [] |
    e f ((x,y)::t) = (f x,f y)::(e f t) |
    e f x y = (f 1, f 2) ::[]
```

```
25
fun f (c::cs) (i::is) = if i> 0
                        then f cs is
                        else (f is c)::[2.2]
```

```
The system explains:
-------------------- Error --------------------
Type conflicts are found in expression:
```

```
   the then/else branches "f cs is" and "(f is c) :: [2.2]"
   (which should have the same type.)


found expressions that should have the same type are infered different types:

   there's one expression type:    the THEN branch "f cs is",
   has the same type as
1:  "f cs is" has the same type as  "f is c"
    has the same type as the element of tail of "(f is c :: [2.2])",
   "[2.2]", whose Element is "2.2": real

   other   expressions    type:    the ELSE branch "(f is c) :: [2.2]"
   has the same type aslist of "f is c",
   "[2.2]" and "(f is c :: [2.2])" have same type
   -operands/result of "::"
   "[2.2]", is list of  real
```

26.

```
fun f (c::cs) (i::is) = if i> 0
                        then f cs is
                        else f is (c::[2.2])
```

27.

```
val reverse = fn xs =>
               let fun loop  ([],ys) = ys |
                       loop  (x::xs,ys) = loop (xs,x::ys)
                in loop (xs, 1)
               end
```

28.

```
fun split [] = ([],[]) |
    split ((x,y) :: rest) = let val (xs,ys) = split rest
                             in (x :: xs, ys::y)
                            end
```

29.

```
let val f = fn x => let val y = x
 in y 5 end in f 3
end
```

30.
```
fun m f [] = [] |
    m f (x :: xs) = (f xs) :: m f x
```

31.
```
fun fff f1 f2 [] [] = [] |
    fff f1 f2 (h1::first) (h2::second) = if f1 h1
                                then ((f2 h2)+1)::(fff f1 f2 second first)
                                else fff f2 f1 (0::first) second
```

32.
```
 fun f (c::cs) (i::is) = if i> 0
                         then f cs is
                         else (c::[2.2])@ f is cs
```

33.
```
 fun m f (h::t) = f h::f t
```

34.
```
 let fun k a b = a in fn h => k (h 1) (h true) end
```

35.
```
 fun f y (h::t) = t y
```

APPENDIX I
Abstracts of Publications

1. **Human and "human-like" type explanations** [68],

   Jun Yang, Greg Michaelson and Phil Trinder:

   Proceedings of 13th annual workshop of PPIG, pages 163-172, G. Kadoda, editor, University of Bournemouth, 17-20 Apr 2001.

   The behaviour of a "human-like" polymorphic type explanation system is analysed using the same categories as those used to characterise human expert type explanation behaviour. The results suggest that the computer system has a similar behaviour profile to that of a composite human expert.

2. **Explaining polymorphic types** [62],

   Jun Yang, Greg Michaelson and Phil Trinder:

   Computer Journal, Accept, Feb 2001.

   Hindley-Milner type inference forms a strong basis for polymorphic type checking but is less well suited to explaining polymorphic types, as it introduces intermediate constructs that relate poorly to a programmer's understanding of the original program. Here, we present a type explanation system which is motivated by explanations given by expert humans rather than by algorithms based on Hindley-Milner inference. In particular, the system seeks to provide succinct explanations with minimal reference to artefacts of mechanised type inference.

3. **How do people check polymorphic types?** [66]

   Jun Yang, Greg Michaelson and Phil Trinder:

   Twelfth annual meeting of the psychology of programming interest group proceedings , Alan F. Blackwell etc, editors, pages 67-77, Memoria, April 2000.

   Polymorphic typechecking algorithm efficiently locate type errors in programs, but users find error reporting from such algorithm hard to comprehend. We are investigating the development of a new polymorphic type checker that reports type errors in a more understandable form. Here we present the result of experiments into human checking of both correct and incorrect polymorphic typed programs, and discuss their implications for our proposed new checker.

4. **Helping Students Understand Polymorphic Type Errors** [65],

   Jun Yang, Greg Michaelson and Phil Trinder:

   Proceedings of 1st Annual Conference of the LTSN centre for Information and Computer Sciences , pages 11-19, August 2000.

   Polymorphic typechecking algorithm efficiently locate type errors in programs, but users find error reporting from such algorithm hard to comprehend. We are investigating the development of a new polymorphic type checker that reports type errors in a more understandable form. Here we present the result of experiments into human checking of incorrectly-typed polymorphic programs. We hope to identify the sequences of techniques people use to explain different categories of error. An outline of a type error explanation system incorporating our heuristics is given.

5. **A visualisation of polymorphic type checking** [63],

   Jun Yang, Greg Michaelson:

   Journal of Functional Programming , 10(1) , pages 57-75, January 2000.

   The understanding of polymorphic typechecking and type errors is poorly supported by contemporary functional language implementations. Here, a novel visualisation of functions and their types is presented based on the generation of function specific icons with graphical type representations which change dynamically as functions are applied. This visualisation has been implemented for a Standard ML subset within a graphical environment in which function combinations are constrained by type matching.

6. **Explaining polymorphic types through visualisation** [64],

   Jun Yang, Greg Michaelson and Phil Trinder:

   Proceedings of 7th Annual Conference on the Teaching of Computing , S. Alexander and U. O'Reilly, editors , pages 73-77, August 1999.

   A 2D visualisation of polymorphic types is presented. A pilot experiment to evaluate the use of the visualisation with novice functional programmers is reported and the visualization's incorporation in a visual programming environment is discussed.

7. **Examples to Compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}_{\mathbf{AE}}$, $\mathcal{IEI}$ algorithms** [70],

   Jun Yang, Phil Trinder, Joe Wells and Greg Michaelson:

   Technique report RM/00/12, Department of Computing and Electrical Engineering, Heriot-Watt University, Oct 2000.

   We give some examples of SML to compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}_{\mathbf{AE}}$, $\mathcal{IEI}$ algorithms. The algorithms are discussed in IFL2000 paper.

8. **Improved type error reporting** [69],

   Jun Yang, Phil Trinder, Greg Michaelson and Joe Wells:

   Draft Proceeding of IFL2000 Implementation of Functional Languages, 12th International Workshop, pages 71-86, September 2000.

   The error reports produced by compliers 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.

9. **Explaining type errors by finding the source of a type conflict** [62],

   Jun Yang,

Trends in Functional Programming, Phil Trinder, Greg Michaelson and Hans-Wolfgang Loidl,editors, pages 58-66, Intellect, March 2000.

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 $\mathcal{W}$ and $\mathcal{M}$ inference algorithms report the site where a type conflict is detected, but the error message is isolated information: it is not clear what the relationship is between the site where error is reported 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 aim to find a method that may be effective even when the user has little knowledge of type checking. The philosophy of our approach is to find sources of type errors by reporting which parts of the program conflict, rather than isolated error sites. We implement two new inference algorithms with this philosophy: the Unification of Assumption Environments ($\mathcal{U}$AE) and the Incremented Error Inference ($\mathcal{IEI}$).

BIBLIOGRAPHY

[1] T. R. Addis and J. J. Townsend Addis. The Clarity Manual. Technical report, Department of Information Science, University of Portsmouth, 1996.

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

[3] B. Backlund, O. Hagsand, and B. Pehrson. Visual language-oriented design environments. *Journal of Visual Languages and Computing*, 1990.

[4] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2:17–30, March 1993.

[5] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[6] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[7] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ml kit (version 1). Technical Report 93/14, Department of Computer Science,University of Copenhagen, 1993.

[8] L. Braine and C. Clack. Object-Flow. In *Proceedings IEEE Symposium on Visual Languages (VL97)*, pages 418–419, 1997.

[9] R. Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18:543–554, 1983.

[10] Magaret M. Burnett. Type and type inference in a visual programming language. In *IEEE Symposium on Visual Languages*, pages 238 – 243, 1993.

[11] R.M. Burstall, D.B.MacQueen, and D.T.Sanella. Hope: an experimental applicative language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980., 1980.

[12] L. Cardelli. Two-dimensional syntax for functional languages. Technical Report CSR-115-82, Department of Computer Science, University of Edinburgh, 1982.

[13] V. Choppella. Diagnosis of ill typed programs. Technical Report 426, Indiana University, 1995.

[14] 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.

[15] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[16] Richard G. Epsteim. The tabletalk query language. *Journal of Visual Languages and Computing*, 2:115–141, 1991.

[17] C. K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of the 17th ACM Symposium on Progrmming Languages*, 1990.

[18] T. R. Green and M. Petre. Usability analysis of visual programming environments. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[19] T. R. G. Green and M. Petre. When visual programs are harder to read than text. In G. C. van de Veer, M. J. Tauber, S. Bagnarola, and M. Antavoltis, editors, *Human-computer interaction: tasks and organisation, Proceedings of ECCE-6, CUD: Rome*, 1992.

[20] Daniel D. Hils. Datavis: A visual programming language for scientific visualization. In *Proc. 1991 ACM Computer Science Conference*, pages 439–448. San Antonio, TX, March 5-7 1991.

[21] R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, 1969.

[22] J. G. Hosking, W. B. Mugridge, S. Fenwick, and J. C. Grundy. Cover yourself with Skin. In *Proceedings of OZCHI'95, Wollongong, Australia*, 1995.

[23] F. Huch, O. Chitil, and A. Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 63–69, 2000.

[24] J. Hughes and S. Peyton Jones (eds). Haskell 98: a non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University., 1998.

[25] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: a visual programming environment. *ACM SIGPLAN Notices*, 23(11):176–190, November 1988.

[26] 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.

[27] S.L. Peyton Jones and P. Wadler. Report on the programming language haskell98: A non-strict, purely functional language. Technical report, February 1999.

[28] J. Kelso. A visual representation for functional programs. Technical report, Department of Computer Science, Murdoch University, Perth, Australia, 1994.

[29] 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.

[30] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical Report MEMO 2000-5, Korea Advanced Institute of Science and Technology, March 2000.

[31] 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.

[32] 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.

[33] Greg Michaelson. Elementary standard ML. UCL Press, 1995.

[34] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497 – 536, 1991.

[35] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17, 1978.

[36] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The definition of Standard ML (Revised).* MIT, 1997.

[37] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[38] R. Morrison, F. Brown, R. Connor, and A. Dearle. The napier88 reference manual. Technical Report Persistent Programming Research Report 77, University of St Andrews/University of Glasgow, July 1989.

[39] M. A. Najork and E. Golin. Enhancing Show-and-Tell with a polymorphic type system and higher-order functions. In *Proc IEEE Workshop on Visual Languages, Skokie, IL*, pages 215–220, 1990.

[40] M. A. Najork and S. M. Kaplan. A prototype implementation of the CUBE language. In *IEEE Workshop on Visual Languages, Seattle, USA*, pages 270–272, 1991.

[41] Marc A. Najork and Simon M. Kaplan. The cube language. In *IEEE Workshop on Visual Languages*, pages 218–224. Kobe, Japan, 1991.

[42] F. G. Pagan. A graphical FP language. *SIGPLAN Notices*, 22(3):21–39, 1987.

[43] Marian Petre. Why looking isn't always seeing readership skills and graphical programming. *Communication of the ACM*, 38(6), June 1995.

[44] G. S. Port. A simple approach to finding the cause of non-unifiability. In R.A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 651 – 665. MIT press, 1988.

[45] J. Poswig, G. Vrankar, and C. Morara. VisaVis: a higher-order functional visual programming language. *Journal of Visual Languages and Computing*, 5:83–111, 1994.

[46] H. J. Reekie. Visual Haskell: a first attempt. Technical Report Research Report 94.5, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, 1994.

[47] L. Rideau and L. Thery. Interactive programming environment for ML. Technical Report 3139, INRIA, 1997.

[48] R. Riding and I. Cheema. Cognitive styles - an overview and integration. *Educational Psychology*, 11(3+4):193–215, 1991.

[49] M. Rittri. Finding the source of type errors interactively. Technical report, Department of Computer Science, Chalmers University of Technology, Sweden, 1993.

[50] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–49, January 1965.

[51] U. Schreiweis, A.Keune, and H. Langendorfer. An integrated prolog programming environment. In *ACM SIGPLAN Notices*, April 1993.

[52] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 439–450. ACM Press, January 1993.

[53] Nan C. Shu. Visual programming, 1988.

[54] Gimpel Software. Pc-lint/flexelint, a diagnostic facility for c and c++. Software Version 7.5 and later, February 1998.

[55] H. Soosaipillai. An explanantion based polymorphic type checker for Standard ML. Master's thesis, Dept of Computer Science, Heriot-Watt University, Edinburgh, Scotland, 1990.

[56] C. Strachey. Fundamental concepts in programming languages. Technical report, Copenhagen, 1967.

[57] Harakd Thimbleby. View binding and user enhanceable systems. *The Visual Computer: An International Journal of Computer Graphics*, 1998.

[58] David Turner. Enhanced error handling for ML. CS4 Project, Department of Computer Science, University of Edinburgh, 1990.

[59] 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.

[60] K. Whitley. Visual programming languages and the evidence for and against. *Journal of Visual Languages and Computing*, 8:109–142, 1997.

[61] S. Wiedenbeck and J. Scholz. Beacons: a knowledge structure in progam comprehension. In G.Salvendy and M.J.Smith, editors, *Designing and using human-computer interfaces and knowledge based systems*, pages 82–87, 1989.

[62] 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.

[63] Jun Yang and Greg Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.

[64] Jun Yang, Greg Michaelson, and Phil Trinder. Explaining polymorphic types through visualisation. In S. Alexander and U. O'Reilly, editors, *Proceedings of 7th Annual Conference on the Teaching of Computing*, pages 73–77. University of Ulster, Jordanstown, August 1999.

[65] Jun Yang, Greg Michaelson, and Phil Trinder. Helping Students Understand Polymorphic Type Errors. In S. Alexander et al, editor, *1st Annual Conference of the LSTN Centre for Information and Computer Science*, pages 11–19, August 2000.

[66] 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, Cosenza, Italy, April 2000. Memoria.

[67] Jun Yang, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *Submission to Computer Journal*, Feb 2001.

[68] Jun Yang, Greg Michaelson, and Phil Trinder. Human and human-like type explanations. In G. Kadoda, editor, *13th Annual Meeting of the Psychology of Programming Interest Group Proceedings*, pages 163–172. University of Bournemouth, April 2001.

[69] Jun Yang, Greg Michaelson, Phil Trinder, and J.B. Wells. Improved type error reporting. In Markus Mohnen and Pieter Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 71–86, September 2000.

[70] Jun Yang, Phil Trinder, J.B.Wells, and Greg Michaelson. Examples to Compare the error reportings from the $\mathcal{W}$, $\mathcal{M}$, $\mathcal{U}$AE, $\mathcal{IEI}$algorithms. Technical Report RM/00/12, Department of Computing and Electrical Engineering, Heriot-Watt University, Oct 2000.

[71] S. Yang, M. M. Burnett, E. DeKoven, and M. Zloof. Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages and Computing*, 8:563–599, 1997.