Type Debugging in Functional Languages

5th November 1999

Bruce McKenzie[1] Brendon J Wyber

[1] Supervisor

Abstract

One of the features of functional languages, such as Haskell and SML, is that the data types of the variables and functions can be inferred from their usage. Unfortunately when an error occurs from this type inference, it may be difficult for a novice programmer to trace the error's source. We introduce two categories of type errors. *Explicit* type errors are caused by the user attempting to use incompatible types together. *Attributed* type errors are the result of editing or syntactical mistakes that are not detected until type checking occurs. The programmer requires differing forms of error messages to debug each category of type error more effectively. We present these display systems and determine that to implement them, the type of each sub-expression in the code must be inferred. This can be implemented using an algorithm given by Beaven and Stansifer.

Contents

Chapter 1

Introduction

One of the special features of certain functional languages, including Standard ML, Haskell, and HUGS, is that the user does not need to specify the data type of variables and functions; the compiler is able to infer the types of data and functions through their usage in the program. A second feature is that a function or data type may be *polymorphic,* having more than one valid type. A type checker is the part of the compiler that resolves the types, and it will fail when the user has inconsistently typed an expression. Unfortunately for users of the language, the source of the error can be difficult to locate and identify.

Consider the following item of code which should calculate a value of total which is $100 plus an additional tax of 15%:

```
total = inclGST 100 15
  where onlyGST base rate = (base * rate) div 100
        inclGST base rate = base + (onlyGST base rate)
```

When given to the HUGS interpreter the following error message is generated:

```
ERROR "error3.hs" (line 2):
(a -> a -> a) -> b -> c is not an instant of class "Num"
```

While the compiler has correctly identified the location of the error (line 2), it is not entirely obvious from the message that the error is because the user has incorrectly used the div operator as an infix operator, when it is a prefix operator, such that the operator must appear before its two arguments, not between them.

It is the aim of this project to find ways in which to present these error messages in a clearer manner, particularly for the novice user. Many first time users of a functional language will come from a procedural programming background, and will have difficulty with the different paradigms and will be unfamiliar the complexities of functional languages' advanced typing capabilities. The project will have an immediate practical benefit of allowing novices to more easily debug their programs.

In order to do this, we have split the type errors into two general categories introduced in this paper, *explicit* and *attributed.* Explicit type errors are the errors which occur when the user has attempted to use incompatibly-typed expressions together. Attributed type errors are caused, not by a misunderstanding of the types, but instead by editing mistakes and syntactical errors which are not detected until the type-checking phase of the compiler. Both of these kinds of errors, as far as the compiler is concerned, are type errors, but to the user they are distinctly different and require different information to debug them. For explicit type errors, a detailed explanation of how the types are inferred is required, while for attributed type errors a syntactical analysis would be more helpful. We discuss the design of how the two differing type explain systems could be able to integrated with an existing system, HUGS.

The emphasis in this work is how the error is presented to the user, as no algorithm to find the cause of an error seems available. Duggan and Bent [5] have said of such an algorithm "we do not believe there is a general solution to this [problem] which scales up".

Chapter 2 describes the fundamental algorithms of modern type-checkers. Chapter 3 discusses previously published work related to this problem. Chapter 4 examines the two different categories and how they emerge from the user's perspective and not the compiler's. Chapter 5 describes the design principles for the presentation of both kinds of errors. The implementation for the language HUGS is discussed in Chapter 6 and conclusions are presented in Chapter 7.

Chapter 2

The Milner Type Inference Algorithm

In order to demonstrate how a compiler determines the type of a polymorphic expression, we present an inference algorithm based on Milner's type checking system[14]. This algorithm is noted by Cardelli[2] as having the feature of being able to infer types without the necessity of having a type declaration. This is an important feature for an interpreter based program such as HUGS.

The algorithm, named $\mathcal{W}$, was the first of its kind to be implemented, being used by Milner for the language ML. Although it was developed independently, it shared a lot in common with work by Hindley, who developed a method for deriving the "principle type scheme" for terms in combinatory logic[7]. This is effectively the same kind of problem to type checking, and it was Hindley who first used the Unification Algorithm of Robinson [16] for this purpose.

In this chapter we discuss Robinson's Unification Algorithm and Milner's $\mathcal{W}$ algorithm as they provide a theoretical background for type checking. We also present a type inference system by Cardelli, which detailed the inference rules used.

2.1 Robinson's Unification Algorithm

The Unification Algorithm first presented by Robinson[16] is widely used in various forms in differing fields of Computer Science. The algorithm is discussed in many texts, both formally [9] and in terms of application [15]. A pseudo-code version is provided by Field and Harrison [6, p155], and a CAML version by Cousineau and Mauny [3, p147-8]. An extremely thorough discussion is given by Manna and Waldinger [10].

Unification attempts to find a set of substitutions of formulae to variables in order to make the two formulas identical. Consider the two expressions

$$6 \times u, \quad a \times (x + 4). \tag{2.1}$$

If we substitute $x + 4$ for u and 6 for a then the formulas become identical:

$$6 \times (x + 4).$$

Therefore we can say that the *substitution* $\{u \leftarrow x+4,\ a \leftarrow 6\}$ is a *unifier* for 2.1. The following definitions will be used throughout the rest of this paper.

We will use the $\blacktriangleleft$ symbol to represent the application of a substitution. For $e \blacktriangleleft \theta$, where $\theta = \{x_1 \leftarrow e_1, x_2 \leftarrow e_2, \ldots x_n \leftarrow e_n\}$, we simultaneously substitute every occurrence of x_i with e_i. This application occurs in one step and is not recursive; therefore

$$u + y \blacktriangleleft \{u \leftarrow y,\ y \leftarrow a\} = y + a$$

and

$$u + y \blacktriangleleft \{u \leftarrow y,\ y \leftarrow a\} \neq a + a.$$

The empty substitution $\{\}$ has no effect when applied to an expression.

The *composition* $\Box$ of two substitutions has the effect of applying the first substitution and then applying the second to the result; therefore

$$e \blacktriangleleft (\theta \Box \lambda) = (e \blacktriangleleft \theta) \blacktriangleleft \lambda.$$

If we have three substitutions θ, λ, and ϕ and

$$\theta \Box \lambda = \phi,$$

then it can be said that θ is *more general than* ϕ. Generally, the more general a substitution is, the fewer the changes that will be made to an expression. For example, $\{x \leftarrow y\}$ is more general than $\{x \leftarrow a,\ y \leftarrow a\}$ because

$$\{x \leftarrow y\} \Box \{y \leftarrow a\} = \{x \leftarrow a,\ y \leftarrow a\}.$$

By this definition, any substitution θ is more general than itself (2.2) and the empty substitution $\{\}$ is more general than any substitution(2.3).

$$\theta \Box \{\} = \theta \tag{2.2}$$

$$\{\} \Box \theta = \theta \tag{2.3}$$

Robinson's algorithm attempts to find the *most-general* unifier for two or more sets of substitutions . A unifier is said to be most-general if it is more general then any other modifier. Expression pairs can have multiple most-general unifiers. For example, for the expressions a and b, both $\{a \leftarrow b\}$ and $\{b \leftarrow a\}$ are most-general unifiers, but $\{a \leftarrow x,\ b \leftarrow x\}$ is not a most-general unifier (although it is a unifier). The most-general unifier will match two expressions doing the least amount of substitutions.

The Disagreement set is the set of the first terms to be encountered where a set of expressions begin to differ. Traverse each expression in the set from left to right and record the first terms which differ. Therefore the disagreement of $\{a + f(b),\ g + f(b)\}$ is $\{a, g\}$ as they differ in the first term of each formula. The disagreement set of $\{f(a) + b,\ d + c\}$ is $\{f(a), d\}$ and the disagreement set of $\{a + d(e),\ a + d(f)\}$ is $\{e, f\}$.

Robinson's Unification Algorithm is stated below. Note that A is a set of expressions for which we want to find the unifier.

Step 1 Set $k = 0$ and $\sigma_k = \{\}$.

Step 2 If $(A \blacktriangleleft \sigma_k)$ is a singleton, then terminate. σ_k is the most-general unifier.

Step 3 Find the Disagreement set B_k of $(A \blacktriangleleft \sigma_k)$. Let u and v be the first two terms in B_k, if they exist. If u is a variable and does not occur in v then $\sigma_{k+1} = \sigma_k + \{u \leftarrow v\}$, increment k and go to Step 2. If v is a variable and does not occur in u then $\sigma_{k+1} = \sigma_k + \{v \leftarrow u\}$, increment k and go to step 2. Otherwise terminate. A is not unifiable.

In summary, the Unification Algorithm traverses each expression in A, left to right, searching for where they differ. Then, when they find the difference, the algorithm adds a substitution if one of the differences is a variable. The program is guaranteed to terminate (an important property), as it will eventually run out of differing sub-terms or find a difference where neither of the terms is a variable (where it cannot unify), or the first term contains the second variable (which is an infinite unification error.

Example U1

Unify $A = \{a + b + c,\ a + f(c) + f(c)\}$:

1. $\sigma_0 = \{\}$
2. $B_0 = \{b, f(c)\}, \sigma_1 = \{b \leftarrow f(c)\}$ and $A \blacktriangleleft \sigma_1 = \{a + f(c) + c,\ a + f(c) + f(c)\}$
3. $B_1 = \{c, f(c)\}$. As $f(c)$ contains c, A is not unifiable.

Example U2

Unify $A = \{a + b + c,\ z + f(g) + c\}$:

1. $\sigma_0 = \{\}$
2. $B_0 = \{a, z\}, \sigma_1 = \{a \leftarrow z\}$ and $A \blacktriangleleft \sigma_1 = \{z + b + c,\ z + f(g) + c\}$
3. $B_1 = \{b, f(g)\}, \sigma_2 = \{a \leftarrow z, b \leftarrow f(g)\}$ and $A \blacktriangleleft \sigma_2 = \{z + f(g) + c\}$
4. A is unifiable and σ_2 is a most-general unifier.

Example U3

Type variables are represented by τ. Unify $A = \{(\tau \rightarrow \texttt{Char}),\ (\texttt{Bool} \rightarrow \texttt{Num})\}$

1. $\sigma_0 = \{\}$
2. $B_0 = \{\tau, \texttt{Bool}\}, \sigma_1 = \{\tau \leftarrow \texttt{Bool}\}$
   and $A \blacktriangleleft \sigma_1 = \{(\texttt{Bool} \rightarrow \texttt{Char}),\ (\texttt{Bool} \rightarrow \texttt{Num})\}$
3. $B_1 = \{\texttt{Bool}, \texttt{Num}\}$. Neither term is a variable so A is not unifiable.

2.2 Cardelli Type Inference System

Before we present Milner's algorithm we will discuss an inference system by Cardelli[2] which shows how type checking is done. The system is more powerful than Milner's algorithm, allowing for *non-shallow* types. A type is considered shallow if it is defined at the top level and there are no quantifiers in the type expression. The type expression $\alpha \to \beta$ is really $\forall\alpha.\forall\beta.\alpha \to \beta$, which is a shallow type.

Below is a list of the set of inference rules. [VAR] is the basic axiom, while the others are proper inferences. The horizontal bar

$$\frac{A}{B}$$

means that "from A we can infer B". We read $A \vdash e : \tau$ as "from the set of assumptions A we can deduce that the expression e has type τ". Also $A.x : \tau$ denotes the result from adding the assumption that x has type τ to set of assumptions A.

$$\begin{array}{lcl}
\textit{Variables} & A.x : \tau \vdash x : \tau & [\mathit{VAR}] \\[2ex]
\textit{Conditionals} & \dfrac{A\vdash e{:}\mathtt{Bool} \quad A\vdash e'{:}\tau \quad A\vdash e''{:}\tau}{A\vdash(\mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e''\ ){:}\tau} & [\mathit{COND}] \\[2ex]
\textit{Abstractions} & \dfrac{A.x{:}\sigma\vdash e{:}\tau}{A\vdash(\lambda x.e){:}\sigma\to\tau} & [\mathit{ABS}] \\[2ex]
\textit{Applications} & \dfrac{A\vdash e{:}\sigma\to\tau \quad A\vdash e'{:}\sigma}{A\vdash(e\ e'){:}\tau} & [\mathit{APP}] \\[2ex]
\textit{Let expressions} & \dfrac{A\vdash e'{:}\sigma \quad A.x{:}\sigma\vdash e{:}\tau}{A\vdash(\mathbf{let}\ x{=}e'\,\mathbf{in}\ e){:}\tau} & [\mathit{LET}] \\[2ex]
\textit{Fixed point} & \dfrac{A.x{:}\tau\vdash e{:}\tau}{A\vdash(\mathbf{fix}\ x.e){:}\tau} & [\mathit{FIX}] \\[2ex]
\textit{Generalizations} & \dfrac{A\vdash e{:}\tau}{a\vdash e{:}\forall\sigma.\tau} \quad (\sigma \text{ not free in } A) & [\mathit{GEN}] \\[2ex]
\textit{Specializations} & \dfrac{A\vdash e{:}\forall\alpha.\tau}{A\vdash e{:}\tau \blacktriangleleft \{\alpha\leftarrow\sigma\}} & [\mathit{SPEC}]
\end{array}$$

These inference rules can be used to infer the types. For example, we type the identity function $(\lambda x.x)$ to its most general type.

$$\frac{\dfrac{x : \alpha \vdash x : \alpha \qquad [\mathit{VAR}]}{\vdash (\lambda x.x) : \alpha \to \alpha \qquad [\mathit{ABS}]}}{\vdash (\lambda x.x) : \forall\alpha.\alpha \to \alpha \quad [\mathit{GEN}]}$$

This can be specialized for each specific case:

$$\frac{\vdash (\lambda x.x) : \forall\alpha.\alpha \to \alpha}{\vdash (\lambda x.x) : \mathrm{Num} \to \mathrm{Num} \quad [\mathit{SPEC}]}$$

Therefore we can infer that the type of $(\lambda x.x)3$ is Num:

$$\frac{\dfrac{3 : \mathrm{Num},\ x : \mathrm{Num} \vdash x : \mathrm{Num} \qquad [\mathit{VAR}]}{3 : \mathrm{Num} \vdash (\lambda x.x) : \mathrm{Num} \to \mathrm{Num} \quad [\mathit{ABS}]} \qquad 3 : \mathrm{Num} \vdash 3 : \mathrm{Num} \quad [\mathit{VAR}]}{3 : \mathrm{Num} \vdash ((\lambda x.x)3) : \mathrm{Num} \quad [\mathit{APP}]}$$

2.3 Milner's $\mathcal{W}$ Algorithm

Unlike Cardelli's inference rules, the $\mathcal{W}$ algorithm is designed to be automated. The W algorithm presented here is a slight variant introduced by Field and Harrison [1989] so that it matches the inference rules in Section 2.2.

When given a set of assumptions A and an expression e, if $\mathcal{W}$ succeeds then it will return (T, τ), where τ is the most general type of e and T is the substitution necessary to unify A and e. The algorithm is as follows:

$\mathcal{W}(A, e) = (T, \tau)$, where

(a) If e is the identifier x, then $T = \{\}$ and
if $x : \forall\alpha_1 \ldots \alpha_n.\sigma \in A$,
then $\tau = \sigma \blacktriangleleft \{\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n\}$,
where $\{\beta \mid 1 \leq i \leq n\}$ are new type variables.

(b) If $e = fg$, let

$$\begin{aligned}&(R, \rho) = \mathcal{W}(A, f)\\&(S, \sigma) = \mathcal{W}(A \blacktriangleleft R, g)\\&U = \mathcal{U}(\rho \blacktriangleleft S, \sigma \to \beta)\end{aligned}$$

where β is a new type variable. Then $T = U \Box S \Box R$ and $\tau = \beta \blacktriangleleft U$.

(c) If $e =$ **if** p **then** f **else** f', let

$$\begin{aligned}&(R, \rho) = \mathcal{W}(A, p)\\&U = \mathcal{U}(\rho, \texttt{Bool})\\&(S, \sigma) = \mathcal{W}(U \Box R \Box A, f)\\&(S', \sigma') = \mathcal{W}(U \Box R \Box A, f')\\&U' = \mathcal{U}(\sigma \blacktriangleleft S', \sigma')\end{aligned}$$

Then $T = U' \Box S' \Box S \Box U \Box R$ and $\tau = \sigma' \blacktriangleleft U'$.

(d) If $e = \lambda.xf$, let

$$(R, \rho) = \mathcal{W}(A.x : \beta, f)$$

where β is a new type variable. Then $T = R$ and $\tau = (\beta \blacktriangleleft R) \to \rho$.

(e) If $e =$ **fix** $x.f$, let

$$(R, \rho) = \mathcal{W}(A.x : \beta, f)$$

where β is a new type variable, $U = \mathcal{U}(\beta \blacktriangleleft R, \rho)$. Then $T = U \Box R$ and $\tau = \beta \blacktriangleleft (U \Box R)$.

(f) If $e =$ **let** $x = f$ **in** g, let

$$\begin{aligned}&(R, \rho) = \mathcal{W}(A, f)\\&(S, \sigma) = \mathcal{W}((A \blacktriangleleft R).x : \rho', g)\end{aligned}$$

where $\rho' = \forall\alpha_1 \ldots \alpha_n.\rho$ and $\alpha_1, \ldots, \alpha_n$ are the free variables in ρ which do not appear in $(A \blacktriangleleft R)$. Then $T = S \Box R$ and $\tau = \sigma$.

The algorithm is slightly complicated due to the concepts of *generic* and *non-generic* variables. Consider the following functions, g and g', defined as:

$$g = \lambda f.h(f\ 3)(f\ \texttt{True}) \tag{2.4}$$

$$g' = \textbf{let } f' = \lambda x.x \textbf{ in } h(f'3)(f'\,\texttt{True}) \tag{2.5}$$

In Equation 2.4, g would fail to be typed. This is because f is a non-generic variable; its instantiated type is shared for all occurrences in the body of g and the two occurrences of f differ ($\texttt{Int} \to \alpha$ and $\texttt{Bool} \to \alpha$ respectively). In Equation 2.5, which shows a similar equation, f' is a generic variable. Its substantiated type is separate for each occurrence. To quote Field and Harrison:

> To summarize, a type variable occurring in the type of an expression E is generic iff it does not occur in the type of the bound variable identifier of any λ-abstraction of which E is a sub-expression.

The $\mathcal{W}$ algorithm is inefficient to implement as it applies more substitutions than necessary. Milner[14] presented a second algorithm $\mathcal{J}$, which used a global environment E, which holds the assumptions of type substantiations, and a variant of the unification algorithm that modifies E as a side effect. It is this algorithm that is implemented in ML.

Example W1

Trace the operation of $\mathcal{W}$ on $\lambda x.x$. From (d) we get:

$$\mathcal{W}(A, \lambda x.x) = (R, (\beta \blacktriangleleft R) \to \rho),$$

where

$$(R, \rho) = \mathcal{W}(A.x : \beta, x).$$

From (a):

$$\mathcal{W}(A.x : \beta, x) = (\{\}, \beta).$$

Therefore substitution results in:

$$\begin{aligned} \mathcal{W}(A, \lambda x.x) &= (R, (\beta \blacktriangleleft R) \to \rho) \\ &= (\{\}, (\beta \blacktriangleleft \{\}) \to \beta) \\ &= (\{\}, \beta \to \beta). \end{aligned}$$

Example W2

Show the operation of $\mathcal{W}$ on $d = (e\ 3)$. The initial assumption set A would be:

$$A = \{3 : \texttt{Int},\ e : \epsilon \to \delta\}.$$

From (b) we get:

$$\mathcal{W}(A, d) = (U\square S\square R,\ \beta \blacktriangleleft B),$$

where

$$(R, \rho) = \mathcal{W}(A, e)$$

$$(S, \sigma) = \mathcal{W}(A \blacktriangleleft R, 3)$$

$$U = \mathcal{U}(\rho \blacktriangleleft S, \sigma \to \beta).$$

From (a) we get both:

$$\mathcal{W}(A, e) = (\{\}, \epsilon \to \delta)$$

$$\begin{aligned} \mathcal{W}(A \blacktriangleleft R, 3) &= \mathcal{W}(A \blacktriangleleft \{\}, 3) \\ &= \mathcal{W}(A, 3) \\ &= (\{\}, \texttt{Int}). \end{aligned}$$

The unification results in:

$$\begin{aligned} U &= \mathcal{U}(\rho \blacktriangleleft S, \sigma \to \beta) \\ &= \mathcal{U}((\epsilon \to \delta) \blacktriangleleft \{\}, \texttt{Int} \to \beta) \\ &= \mathcal{U}((\epsilon \to \delta), \texttt{Int} \to \beta) \\ &= \{\epsilon \leftarrow \texttt{Int},\ \delta \leftarrow \beta\}. \end{aligned}$$

Merging the results we get:

$$\begin{aligned} \mathcal{W}(A, d) &= (U \Box S \Box R, U\beta) \\ &= (\{\epsilon \leftarrow \texttt{Int},\ \delta \leftarrow \beta\} \Box \{\} \Box \{\}, \beta \blacktriangleleft \{\epsilon \leftarrow \texttt{Int},\ \delta \leftarrow \beta\}) \\ &= (\{\epsilon \leftarrow \texttt{Int},\ \delta \leftarrow \beta\}, \beta). \end{aligned}$$

Example W3

To present the operation of $\mathcal{W}$ when type checking

$$g = \textbf{let } f = \lambda x.x \textbf{ in } h(f\ 3)(f\ \texttt{True})$$

in the level of detail as in Examples W1 and W2 would be tedious; aside from the **let** component, the lambda abstraction and function applications have already been shown. We will concentrate on how generic type variables are handled by $\mathcal{W}$. The assumption set would initially be

$$A = \{3 : \texttt{Int},\ \texttt{True} : \texttt{Bool},\ h : \epsilon \to \delta \to \gamma\}.$$

By applying (f) we get:

$$\mathcal{W}(A, g) = (S_1 \Box R_1, \sigma_1),$$

where

$$(R_1, \rho_1) = \mathcal{W}(A, \lambda x.x).$$

From Example W1 we get:

$$\begin{aligned} (R_1, \rho_1) &= \mathcal{W}(A, \lambda x.x) \\ &= (\{\},\ \beta_1 \to \beta_1). \end{aligned}$$

Also from (f) we get:

$$(S_1, \sigma_1) = \mathcal{W}((A \blacktriangleleft R_1).f : \rho_1',\ h(f\ 3)(f\ \texttt{True}).$$

As $R_1 = \{\}$ and $\rho_1 = \beta_1 \to \beta_1$ then β_1 is the only free variable that is in ρ_1 and is not in $(R_1 \Box A)$. Hence, we can substitute α for β_1 to make a new assumption set A_α, where

$$A_\sigma = A.f : \forall\alpha.\alpha \to \alpha.$$

Therefore

$$(S_1, \sigma_1) = \mathcal{W}(A_\alpha,\ h(f\ 3)(f\ \mathtt{True}).$$

This means that when $(f\ 3)$ and $(f\ \mathtt{True})$ are type checked by rule (a) there will be a substitution with a separate β_n type variable for each instantiation, thus preserving the polymorphic property of generic type variables.

Chapter 3

Previous Work

This chapter will review previously published work. Bruce McAdam's [11] modifications to the $\mathcal{W}$ algorithm is an attempt to make type debugging easier by changing the type checking algorithms. The work of Beaven and Stansifer [1] and Duggan and Bent[5] developed type explainer systems which attempt to analyse the type error and present it to the user in a clearer way. All three systems will be discussed in the following chapter.

3.1 McAdam's Unification of Substitutions

McAdam identified a *left-to-right bias* in the $\mathcal{W}$ algorithm. If we reconsider Equation 2.4 from Chapter 2,

$$g = \lambda f.h(f\ 3, f\ \texttt{True}),$$

when the expression is type checked f is instantiated to $(\texttt{Int} \rightarrow \beta)$ and then fails to unify when later f is substantiated to $(\texttt{Bool} \rightarrow \beta)$. At this point it then reports the error, on the 'right-hand side' of the expression. This is misleading, as the error lies not with the instantiation of $(\texttt{Bool} \rightarrow \beta)$, which is where it is detected, but instead at the level above, in h, where f has failed to unify.

The reason for this bias lies with application rule (b) in the $\mathcal{W}$ algorithm (page 7). The second step is the following:

$$(S_1, \sigma) = (A \blacktriangleleft R, g).$$

In this step, the substitution R is applied to the assumption set A before the type of g is substantiated. This is the cause the left-to-right bias, as the result of evaluating $(R, \rho) = \mathcal{W}(A, R)$ is used before it is known to be consistent with g.

The solution to this is to evaluate $\mathcal{W}(A, f)$ and $\mathcal{W}(A, g)$ independently and then merge the results. This will effectively generate two pairs (S_1, ρ_1) and (S_2, ρ_2). It is then necessary to check that the two substitutions are consistent with each other and "create a substitution which contains the effect of both" [11, p.145]. To do this we must unify the substitutions.

McAdam's formal definition for a substitution unifier follows:

A substitution, S', unifies substitutions, S_0 and S_1, if $S_0 \blacktriangleleft S' = S_1 \blacktriangleleft S'$. In particular the most general unifier of a pair of substitutions is S' such that:

$$(S_0 \blacktriangleleft S' = S_1 \blacktriangleleft S') \wedge$$
$$((\forall S'' : S_0 \blacktriangleleft S'' = S_1 \blacktriangleleft S'') \Rightarrow (\exists R : S'' = S' \blacktriangleleft R))$$

We define the algorithm $\mathcal{U}_S(S_0, S_1)$ to return the substitution unifier. Examples of $\mathcal{U}_S$ include:

$$\begin{aligned} S_0 &= \{\alpha \leftarrow \texttt{Bool}\} \\ S_1 &= \{\beta \leftarrow \gamma\} \\ \mathcal{U}_S(S_0, S_1) &= \{\alpha \leftarrow \texttt{Bool},\ \beta \leftarrow \gamma\}, \end{aligned}$$

where S_0 and S_1 are completely independent. When they contain a common element we have to unify as in the example:

$$\begin{aligned} S_0 &= \{\alpha \leftarrow \beta \leftarrow \texttt{Bool}\} \\ S_1 &= \{\alpha \leftarrow \texttt{Int} \leftarrow \gamma\} \\ \mathcal{U}_S(S_0, S_1) &= \{\texttt{Bool} \leftarrow \gamma,\ \texttt{Int} \leftarrow \beta\}. \end{aligned}$$

The unification can still fail as seen in the following equation, which has an *occurs-within* error similar to Robinson's Unification Algorithm.

$$\begin{aligned} S_0 &= \{\alpha \leftarrow \texttt{Int} \leftarrow \beta\} \\ S_1 &= \{\beta \leftarrow \texttt{Int} \leftarrow \alpha\}. \end{aligned}$$

We will omit McAdam's definition of $\mathcal{U}_S$ as it is available with proofs in his extended paper[12]. The new version of $\mathcal{W}$, $\mathcal{W}'$, differs only in the case of applications:

(b) If $e = fg$, let

$$\begin{aligned} &(R_1, \rho_1) = \mathcal{W}(A, f) \\ &(R_2, \rho_2) = \mathcal{W}(A, g) \\ &W = \mathcal{U}_S(R_1, R_2) \\ &\sigma_1 = \rho_1 \blacktriangleleft W \\ &\sigma_2 = \rho_2 \blacktriangleleft W \\ &V = \mathcal{U}(\sigma_0,\ \sigma_1 \rightarrow \beta) \end{aligned}$$

where β is a new type variable. Then $T = V \square W \square R_1$ and $\tau = \beta \blacktriangleleft V$.

The advantage of $\mathcal{W}'$ is that if the types of f and $g \rightarrow \beta$ fail to unify it is caught after the evaluations. The left-to-right bias is no longer present, therefore the error message will be generated in the position where the error belongs.

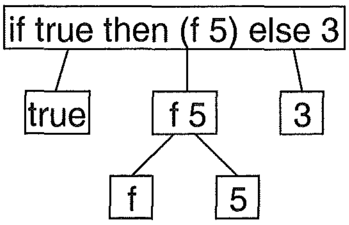


Figure 3.1: A Syntax Tree for **if** true **then** (f 5) **else** 3

<expr>	::=	*<identifier>*	
	|	*<integer>*	integer constants
	|	*<boolean value>*	boolean constants
	|	(*<expr>*, *<expr>*)	a pair
	|	*<expr>* *<expr>*	functional application
	|	`fn` *<identifier>* `:=` *<expr>*	function definition
	|	`if` *<expr>* `then`*<expr>* `else` *<expr>*	conditional expression
	|	`let` *<identifier>* `=`*<expr>* `in` *<expr>* `end`	**let** expression

Figure 3.2: Syntax of a Simple Functional Language

3.2 Beaven and Stansifer's Type Explainer

Beaven and Stansifer[1], presented a method where the unification algorithm remembers and stores the substitutions performed during the type checking stage. It uses this stored information when tracing through a *syntax tree* of an expression to display a step by step explanation of the type inferences. A syntax tree is a directed graph where each vertex represents an expression and with edges pointing to each of its sub-expressions, as shown in Figure 3.1.

In their implementation, Beaven and Stansifer represent substitutions as a list of pairs. For reasons of efficiency they store the substitution terms in a continuously specialising series, such that:

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

which represents the substitution:

$$\{v_1 \leftarrow t_1, v_2 \leftarrow (t_2 \blacktriangleleft \theta_1), \ldots, v_n \leftarrow (t_n \blacktriangleleft \theta_{n-1})\}$$

where θ_i is the substitution:

$$\{v_1 \leftarrow t_1, v_2 \leftarrow (t_2 \blacktriangleleft \theta_1), \ldots, v_n \leftarrow (t_i \blacktriangleleft \theta_{i-1})\}.$$

This system has the advantage that by simply adding and subtracting further atomic bindings (v_i, t_i) the entire substitution is specialised and generalised, similar to pushing to and popping from a stack.

They implemented the type explanation system for the simple functional language as described in Figure 3.2. The system consisted of the following elements:

- a representation of the generated parse tree,
- details of the substitutions at each node,

- details of type assignments,
- and functions, why and how, that traverse the above information and present the information to the user.

As their system performs the type checking, it traverses the syntax tree and stores, at each node, the unification results and the type values assigned. The exact information stored is dependent on the expression type.

Constants do not store any additional information.

Pairs store the types of its sub-expressions.

Function definitions store the type of the body and the initial type variable that is created for the identifier. This identifier is used as the base for which bindings will apply to form the domain type.

Function applications store the two types which are formed by unification of the application.

Identifiers must store the original type variable created for them. They also record a list of all the type bindings that occur. This is to allow the **let** statements to have polymorphic types, as discussed in Chapter 2.

Conditionals stores the type of their sub-expressions.

The details of the explanation algorithms, why and how, are not described in the paper, but example outputs are shown [1, p.23, 27, 29]. The explanations are extremely verbose; the type trace for

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

is over 50 lines long. In practise this would require a user interface designed to allow the user to browse the given information. Requirements for such user interfaces are presented in Chapter 5.

3.3 Duggan and Bent's Type Explainer

A different approach was developed by Duggan and Bent[5]. For each program a type variable is found. Each substitution that occurs for the variable is then stored with the snippet of program code responsible for the substitution. If the substitutions are unified successfully, then the code snippets are used to provide the explanation. For example, in the Standard ML program fragment:

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

x is initially assigned a type of τ_1, which is a type variable. The function body contains the application (converted from infix to prefix notation) $+(x, 1)$ which has the type (τ_1 * int) and the + function has the type (int * int). Unification produces the substitution $\{\tau_1 \leftarrow int\}$ and the program fragment $+(x, 1)$ is given as the explanation.

This approach is not sufficient to explain more complex expressions. For example, given the following section of code:

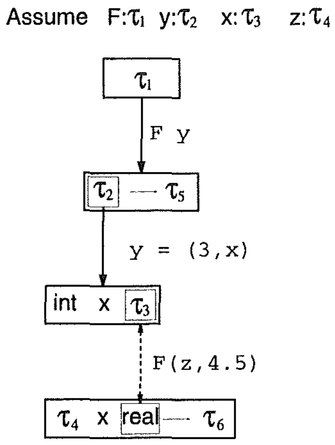


Figure 3.3: Type Tree for F function

```
F y
y = (3,x)
F (z, 4.5)
```

if the simple system were used, it would give the explanation for the type of x being `real` as `(F( z, 4,5))`. This is insufficient information to clearly explain the type. Consequently the typing information is built into a directed graph. Each vertex stores the assigned types and the directed edges are stored with the code fragments that created them. A modified unification algorithm traverses this graph collecting the code fragments and the list is presented as an explanation. The type graph of the previous section of code is shown in Figure 3.3. Sub-expressions are graphically represented by nesting the vertices. Using this method the explanation for the type of x becomes:

x : real	
`F y`	gives F: $\tau_1 = \tau_2 \rightarrow \tau_5$
`y = (3,x)`	gives y: $\tau_2 = \texttt{int} \times \tau_3$
`F (z, 4.5)`	gives x: $\tau_3 = \texttt{real}$

The system developed by Duggan and Bent, called SML/E, explains type assumptions for the Standard ML of New Jersey (SML/NJ). It consists of a command line shell which loads source files and provides the **describe** and **explain** commands each of which take a single variable identifier as an argument. Variables in **let** expressions are referred to using a *<function identifier>* . *<variable identifier>* format. The **describe** command describes when the given variable is defined. The **explain** command provides an explanation of why a variable has a certain type. An example of the running program is given in Figure 3.4. The system uses `'A0`, `'A1`,... to refer to the type variables. The input from the user is bold-faced and <exp> is used to display shortened sub-expressions.

One drawback of the system is that it only explains the types of identifiers and not sub-expressions. A prototype of the system is publicly available[4].

```
esh<4> describe f
f was introduced in the following declaration:
(* f:  'A0 *)
fun f ((x:'A1,y:'A2,z:'A3))=(if x then (Int.+(<exp>,<exp>)) else z)

esh<5> describe f.x
f was introduced in the following declaration:
(* f:  'A0 *)
fun f ((x:'A1,y:'A2,z:'A3))=(if x then (Int.+(<exp>,<exp>)) else z)

esh<6> explain f.x
The explanation for the type of f.x is as follows:
[0] f.x :  'A1 by Assumption
[1] f.x :  bool because...
'A1 (the type variable for f.x) was instantiated to bool as a
result of requiring the conditional test to have type bool in
this expression:
(if x then (Int.+ (y,1)) else z)
```

Figure 3.4: Transcript of SML/E

Chapter 4

The User's Perspective

4.1 Attributed and Explicit Type Errors

The term 'type error' can often be misleading to the novice user; they could have made another form of error which is only detected by the type checker. For example, consider the following incorrect statement in HUGS:

```
onlyGST :: Int -> Int -> Int
onlyGST base rate = base * (rate div 100)
```

The error occurs because div, the integer division operator, is a prefix operator and the user has used it as an infix operator. The correction would be to use div correctly (div rate 100), or to promote it to infix using the quotes (rate 'div' 100). From the user's perspective this is a syntactical error, but it parses correctly because the 'div' operator is allowed to be used as an argument to a function. When the interpreter compiles the statement it generates the following error:

```
ERROR "/.../example1.hs"
(line 2): Type error in function binding
*** term           : onlyGST
*** type           : Int -> ((a->a->a)->b->Int) -> Int
*** does not match : Int -> Int -> Int
```

This error message highlights several differences between the user's view of the error and the compiler's view of the error:

- As already discussed, it is caught as a type error by the compiler when the user would consider it to be a syntax error.
- The error is caught at a place that differs from where the user would expect the error to be. This is because the user defined the type of onlyGST. If they had omitted the declaration of onlyGST then the error would not have been caught until onlyGST was used.
- The compiler indicates that the error lies at the second argument of onlyGST, or the rate argument. This gives the user a general location of the error, but does not specifically pinpoint the error. Examples can be contrived where the error is more obscured.

- The typing is obscuring the problem. The div operator is typed as (a->a->a) and the 100 is typed as b, but a novice would have difficulty in interpreting it. Once the user is aware of what the problem is, it is easier to understand what the error message means and how it came about, but working backwards in an attempt to derive the error from the message is much less intuitive.

The error message should have been more explicitly localised. Due to the declaration of onlyGST, the compiler knew its type and what the type and name of its second argument should have been. The error message:

```
ERROR "/.../example1.hs"
(line 2): Type error in argument for function onlyGST
*** term           : rate
*** type           : (a -> a -> a) -> b -> Int
*** does not match : Int
```

would have been more helpful.

The fundamental problem is that there is no intuitive way for the compiler to determine that the error is not related to rate but to div instead.

Consequently from the user's perspective we introduce two categories of type errors. The *attributed* type errors are similar to those discussed above; errors that are not caught until checked by the type checker, whose mismatches are symptoms of another fault. The *explicit* type errors, are errors where the user has genuinely mismatched types. The compiler cannot make any distinction between the two categories.

4.2 Explaining Each Category

Our distinction between attributed and explicit type errors in not made for arbitrary reasons, as the user requires different information in order to resolve the error.

Attributed type errors are caused by the following:

- incorrect usage of infix and prefix operators,
- similarly incorrect usage of the unary minus or subtraction symbol
- accidently inserted characters that happen to match an identifier
- transposed characters,
- incorrect use of parentheses,
- duplicated sections of code caused from a cut and paste editing mistake[1].

The location of the error can be more helpful to the user than the reason why the error occurred or what the specifics of the error is. For trivial errors, the mere knowledge of the existence of the error and general location is enough for an experienced user to find it. For novice users, attributed type errors will

[1] As more and more powerful drag-and-drop editors become more available, these kinds of errors become more frequent.

be very common and may only require simple changes to fix, however finding them could be far from trivial.

For explicit type errors, where the error is the result of an incorrect type matching by the user, different information is required to locate the source. It is this type of error that has been the focus of Duggan et al. as discussed in Chapter 3. Here the user may need to trace through the derivation of each type inference to see how the error occurred.

Because of this distinction of errors, and the differing information needed to correct the errors, a single method to present the error to the user can not easily deal with both categories of type errors. A full explanation system would produce unnecessary and possibly misleading information for attributed type errors and a more higher-level location-based type-error explainer would be no help for a user attempting to trace an explicit type error.

Since a compiler cannot determine which category of type error an error is, both mechanisms should be available at all times and should not interfere with each other. The next chapter discusses the design aspects and requirements for each of these type error presenters.

Chapter 5

Displaying the Error

While identifying the source of a type error and locating the section of the code is important, it is wasted if the result is not displayed to the user in an understandable fashion. In this chapter we discuss the design of an interface used to display type errors.

5.1 Attributed Type Errors

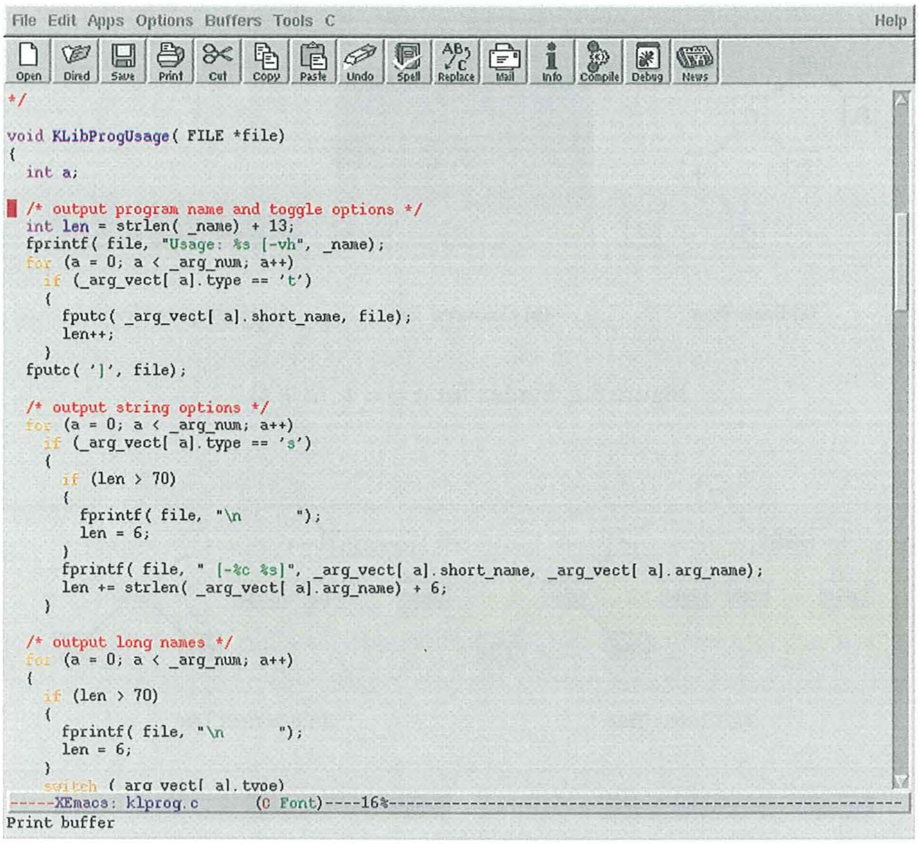
For displaying attributed type errors, the location of the error is the most important property to display to the user. While this is typically detailed by a line number, sometimes with reference to a specific location within a line, this information can be incorrect. The error may have occurred elsewhere in the source file and the indicated location is just where the type checker first detects the symptoms. Therefore, a global view of the type information is desirable for the complete file.

This task is made easier if the user's development environment included a file viewer. A modern Graphical User Interface (GUI) is ideal for this, having many advantages over the standard text-style terminal, including easy availability and use of colour, and a built-in type of pointing device, typically a mouse.

5.1.1 Source Highlighting

The source highlighted editor is a common development tool that is used for many languages. This tool is a standard GUI-based editor that colours the text being edited according to rules based on the grammar of the target language. This is typically used with a type of automatic indenting (where new lines are automatically indented based on code layout) as well as parentheses matching, (as one parenthesis is typed in its opposing member of the pair is highlighted in some way).

While source highlighting rules can be easily defined for several classes of imperative languages, it is a much more difficult for a functional language such as HUGS. For example, in C, the highlighting is typically used to display syntactical structures, such as `while` and `for` loops, and code blocks. Function



```
*/

void KLibProgUsage( FILE *file)
{
  int a;

  /* output program name and toggle options */
  int len = strlen( _name) + 13;
  fprintf( file, "Usage: %s [-vh", _name);
  for (a = 0; a < _arg_num; a++)
    if (_arg_vect[ a].type == 't')
    {
      fputc( _arg_vect[ a].short_name, file);
      len++;
    }
  fputc( ']', file);

  /* output string options */
  for (a = 0; a < _arg_num; a++)
    if (_arg_vect[ a].type == 's')
    {
      if (len > 70)
      {
        fprintf( file, "\n      ");
        len = 6;
      }
      fprintf( file, " [-%c %s]", _arg_vect[ a].short_name, _arg_vect[ a].arg_name);
      len += strlen( _arg_vect[ a].arg_name) + 6;
    }

  /* output long names */
  for (a = 0; a < _arg_num; a++)
  {
    if (len > 70)
    {
      fprintf( file, "\n      ");
      len = 6;
    }
    switch ( arg vect[ a].type)
-----XEmacs: klprog.c      (C Font)----16%--------------------------------------------------
Print buffer
```

Figure 5.1: Example of syntax highlighting for C in xemacs

calls are clearly distinguishable, and the expressions in statements are typically shown as the same highlighted level, an example shown in Figure 5.1.

For source highlighting in functional languages it is not immediately clear what would be beneficial to show to the user. It is also unclear how to compose rules for an editor without the editor essentially parsing the entire file to build the syntax tree. This is simpler for interpreted systems, such as HUGS as the parser is always available. For compiled systems, such as Standard ML of New Jersey (SML/NJ), this would involve re-running the compiler.

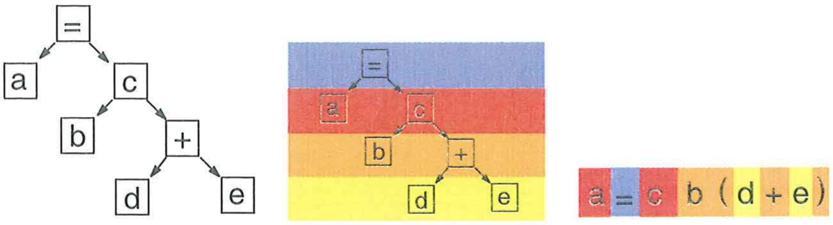
As the type checking algorithm is closely related to the syntax tree it would make sense to highlight according to this. Consider the following equation:

$$a = c\ \ b\ \ (d + e)$$

which has the syntax tree shown in Figure 5.2(a). An expression consisting of composite functions, such as $(x + y)$ which is really $((+x)\ y)$, these are expressed with the deepest function first $(+)$ and the children of the node are the arguments (x and y). If each level was assigned a colour (as in Figure 5.2(b)), the resulting syntax highlighting would be as shown in Figure 5.2(c).

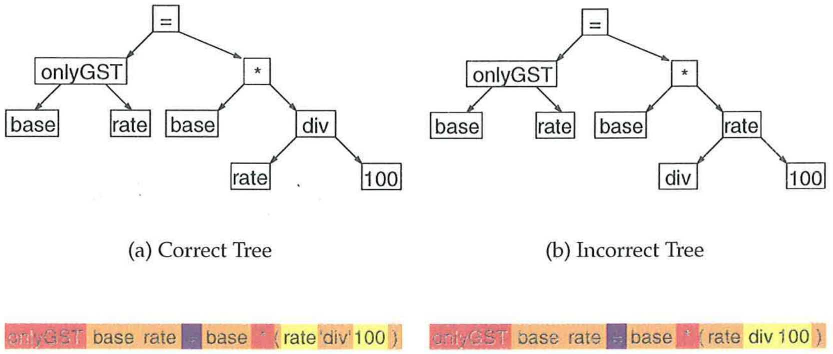
Consider the two following HUGS statements:

```
onlyGST:: Int -> Int -> Int
```



(a) Basic Tree (b) Coloured Tree (c) Highlighted Syntax

Figure 5.2: Syntax for $a = c \ \ b \ \ (d + e)$


=
onlyGST
*
base
rate
base
div
rate
100
=
onlyGST
*
base
rate
base
rate
div
100
onlyGST base rate = base * (rate 'div' 100)
onlyGST base rate = base * (rate div 100)

(a) Correct Tree (b) Incorrect Tree

(c) Correct Highlighting (d) Incorrect Highlighting

Figure 5.3: Highlighting by Syntax

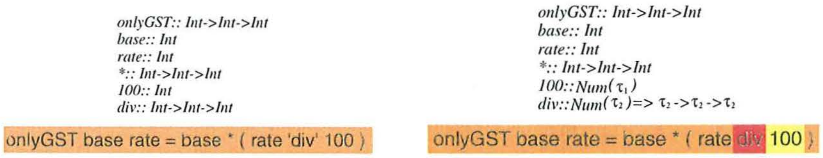
```
onlyGST base rate = bae * (rate div 100)

onlyGST:: Int -> Int -> Int
onlyGST base rate = bae * (rate 'div' 100)
```

both define `onlyGST`, but the second definition contains a type error. When we use the highlighting-by-syntax method on the source we get the syntax trees in Figures 5.3(a) and 5.3(b) with the corresponding highlighted syntax in Figures 5.3(c) and 5.3(d). From the coloured text it is easier to spot the attributed type error discussed in Chapter 4.

A criticism of the colouring-by-syntax method is that it will lead to a very colourful display in the editing environment. The constant change of colour may lead to an error being skipped, unless the highlighted area is restricted. Also, if the location of the error is incorrect it may not help tracing the error.

Another method of highlighting is to choose a colour based on type. Each basic type in an expression could be assigned a colour. More complex data types would have to be 'simplified'. Lists could be assigned the colour of their



onlyGST:: Int->Int->Int
base:: Int
rate:: Int
*:: Int->Int->Int
100:: Int
div:: Int->Int->Int

```
onlyGST base rate = base * ( rate 'div' 100 )
```

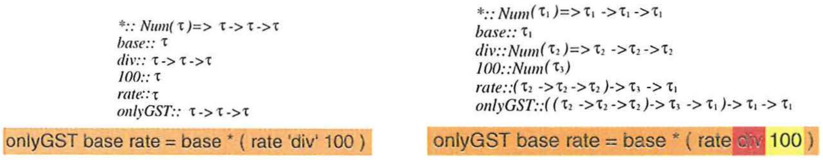
(a) Correct Highlighted Type

onlyGST:: Int->Int->Int
base:: Int
rate:: Int
*:: Int->Int->Int
100::Num(τ_1)
div::Num(τ_2)=> τ_2->τ_2->τ_2

```
onlyGST base rate = base * ( rate div 100 )
```

(b) incorrect Highlighted Type

Figure 5.4: Highlighting by Type, With Declaration



*:: Num(τ)=> τ->τ->τ
base:: τ
div:: τ->τ->τ
100:: τ
rate::τ
onlyGST:: τ->τ->τ

```
onlyGST base rate = base * ( rate 'div' 100 )
```

(a) Correct Highlighted Type

*:: Num(τ_1)=>τ_1 ->τ_1->τ_1
base:: τ_1
div::Num(τ_2)=>τ_2 ->τ_2->τ_2
100::Num(τ_3)
rate::(τ_2 ->τ_2->τ_2)->τ_3 -> τ_1
onlyGST::((τ_2 ->τ_2->τ_2)->τ_3 -> τ_1)->τ_1->τ_1

```
onlyGST base rate = base * ( rate div 100 )
```

(b) incorrect Highlighted Type

Figure 5.5: Highlighting by Type, Omitting Declaration

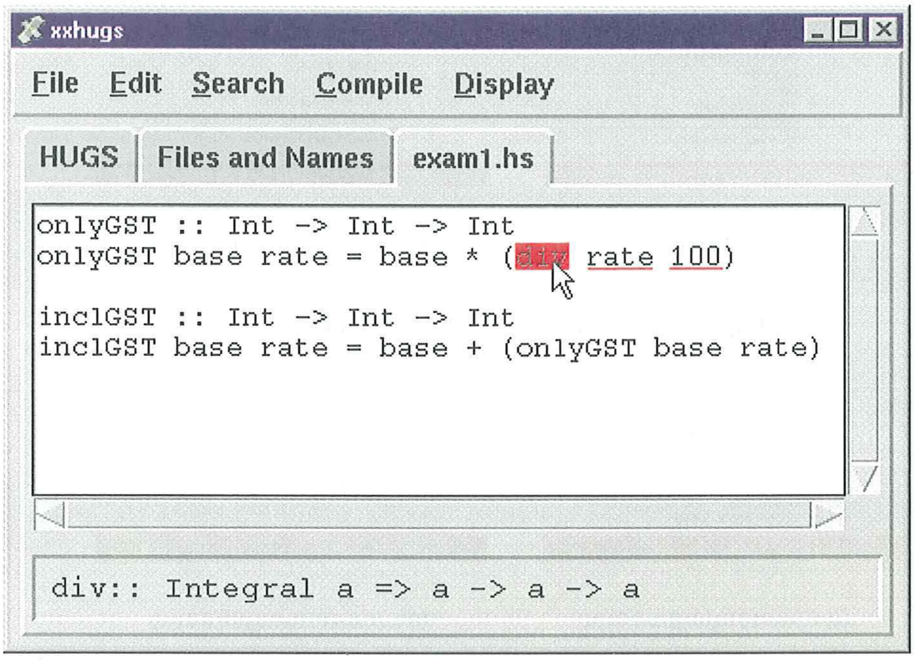
member types; tuples, the type of the first element; functions, the type of their range; and polymorphic types, their instantiated type. The result of this colouring system can be seen in Figures 5.4(a) and 5.4(b), as well as the instantiated types for each lexical element. For the given example the attributed type error stands out very clearly.

The highlight-by-type method requires more work than the highlight-by-syntax method. The type for each lexical token must be instantiated. In the above example, the types of OnlyGST, base, and rate are known from the declaration. This information is used to unify the * operator to Int->Int->Int. However, because of the error, the type of the div operator differs between the correct and incorrect version. Even if the type declaration of onlyGST was omitted, the highlighting would be the same, even if the calculated types were different (as shown in Figures 5.5(a) and 5.5(b)).

While the highlight by type method can show more errors, for expressions with many differing types in them it can still generate overly colourful source displays.

5.1.2 A Context-Sensitive Pointer

A pointing device can be used to retrieve information from an editing environment; when the user desires to view type information about a particular lexical token the user could just point at it and the information could then be calculated and displayed. Figure 5.6 shows how it could be typically used in a functional language environment. The pointer is over the div token, which becomes fully highlighted. The type of div is displayed in a status area under



```
onlyGST :: Int -> Int -> Int
onlyGST base rate = base * (div rate 100)

inclGST :: Int -> Int -> Int
inclGST base rate = base + (onlyGST base rate)
```

```
div:: Integral a => a -> a -> a
```

Figure 5.6: Context Sensitive Pointing Device

the main file display and the arguments to div (because its type is a function) are underlined.

The advantage of a pointer-style display is that only the lexical elements under the pointer have to be checked, not the entire file, removing the necessity of having to remember the type information for every part of the current source. Another advantage of this method is that when a file is being changed, the type information does not have to be continuously recalculated for the currently edited section of code. A disadvantage is that the information received is very localized, and information about the overall structure of the code is not displayed.

5.2 Explicit Type Errors

The display requirement for explicit type errors is a lot more detailed. A display should have the following requirements:

- the exact type of each expression and sub-expression should be shown,
- superfluous details should be able to be hidden,
- the system should be able to scale up, so that large and complex expressions can be handled,
- the results of type unification should be available,
- and it should be easy to navigate through the display to get to the relevant expression as necessary.

A commonly-used structure which has many of the required qualities is the tree. A type explainer based around a tree has several advantages:

- the type explainer can closely follow the syntax level of the expressions, which is important as the type algorithm also follows it,
- entire branches can be collapsed/ignored if they are correct or displayed to the required level of detail,
- trees can scale well by breaking down large collections of information into a more manageable hierarchy,
- and due to the common usage of trees, there are several available widget sets that can handle them, hence they are easier to prototype.

For a more detailed type checking, every type application would have to be traced through. No 'simplification', as for highlighting, would be done. The tracing of expressions like $s = (a + b + c)/2$ would have to be presented at its more functional level, $s = (/\ (+\ (+\ a\ b)\ c)\ 2)$. The resulting binary tree for the following HUGS statement:

```
areaOfTrangle a b c = sqrt(s*(s-a)*(s-b)*(s-c))
          where s = (a + b + c)/2
```

is shown in Figure 5.7. Each expansion of s has been shaded for clarity. This syntax tree can be displayed as a browsable tree similar to the file and directory browsers available in modern graphical shells.

Since it is not feasible to have the tree displayed in the source, the display would occur in a separate window. This also allows room for additional information to be displayed. An example of the type of display is shown in Figure 5.8, which has a window divided into three areas; a button bar, the browsable tree, and an area for explanation. Figure 5.9 shows how someone would step through the tree.

5.3 Required Type Information

To implement the type explainers discussed in this chapter, the type of each sub-expression in an expression syntax tree needs to be inferred. Therefore the type explainer of Beaven and Stansifer is more suited to this task than the type explainer of Duggan and Bent.

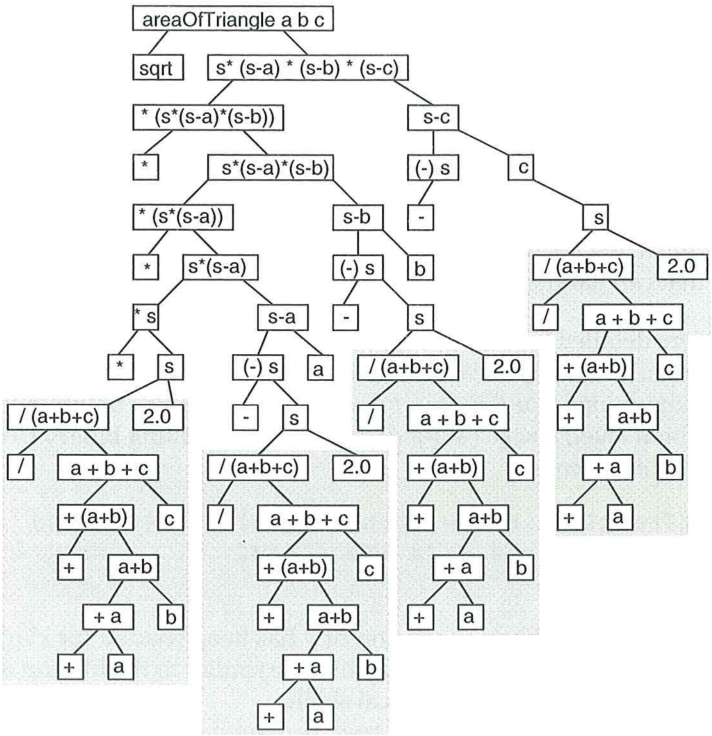


Figure 5.7: Type Tree for `areaOfTriangle`

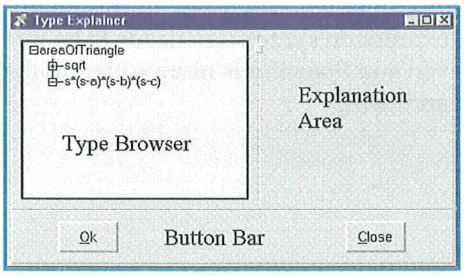


Figure 5.8: Layout for Type Explainer

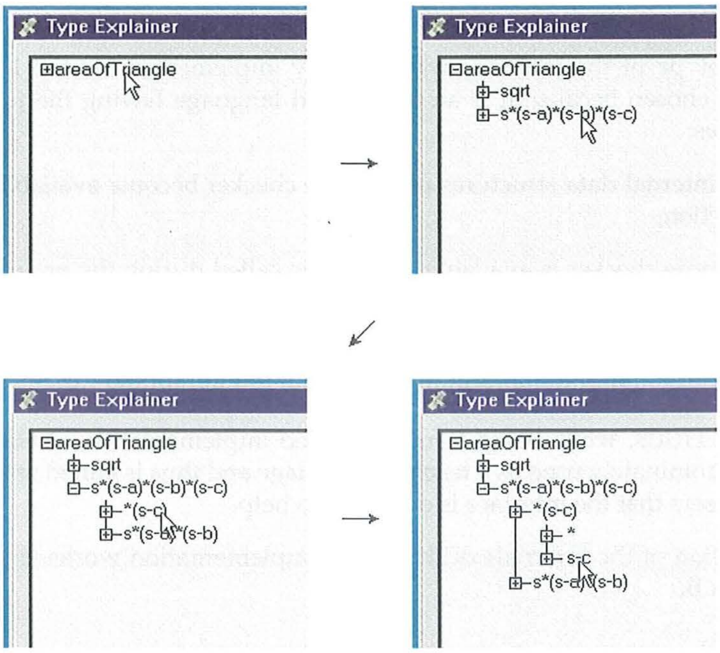


Figure 5.9: Browsing through the Type Explainer

Chapter 6

Implementation

The prototype of the interface was partially implemented for the language HUGS[8], chosen because it is an interpreted language having the following advantages:

- the internal data structures of the type checker become available for inspection,
- the type checker is available and can be called during the program execution,
- the interpreter has some built-in type diagnostics (such as the `:type` command) that are available during program execution,
- and HUGS, while being a fully featured implementation of Haskell, is predominately used as a teaching language and thus is aimed at the level of users that the interface is designed to help.

A discussion of the internals of the HUGS implementation works is given in Appendix B.

6.1 Overview of the System

The main component of the system was a graphical shell, `xxhugs.tcl`, written in **tcl/tk**, using the **tix** package for additional widgets. The second component was a modified version of the HUGS interpreter.

To start the system, the user would run the command:

```
xxhugs.tcl filename
```

where *filename* is the name of the HUGS source file. This will start the `xxhugs` script. After displaying its own windows, the `xxhugs` script will start HUGS with the supplied source file, with its input and output piped to and from `xxhugs`. Output from HUGS is captured by `xxhugs` until the prompt is received, then HUGS is blocked, waiting for input. As the user interacts with `xxhugs`, commands will be sent to HUGS, and the command results are displayed by `xxhugs`. The execution flow is summarised in Figure 6.1.

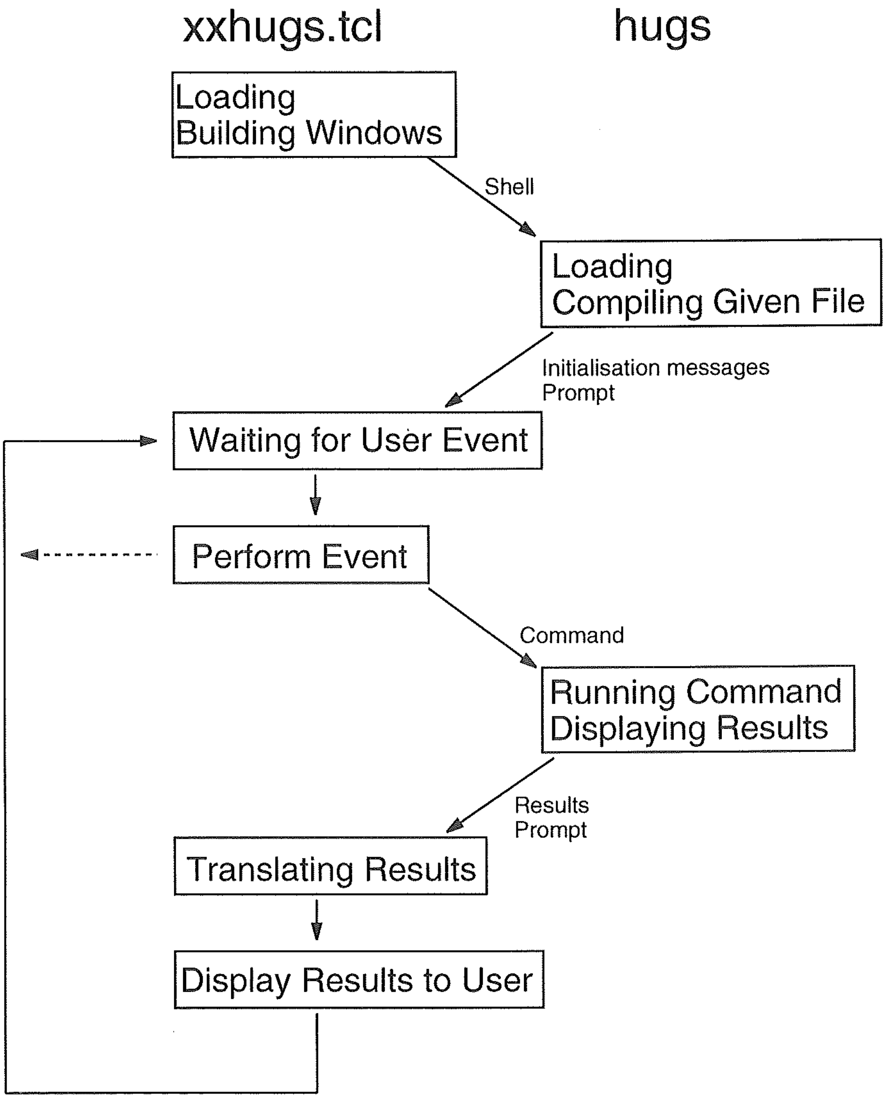
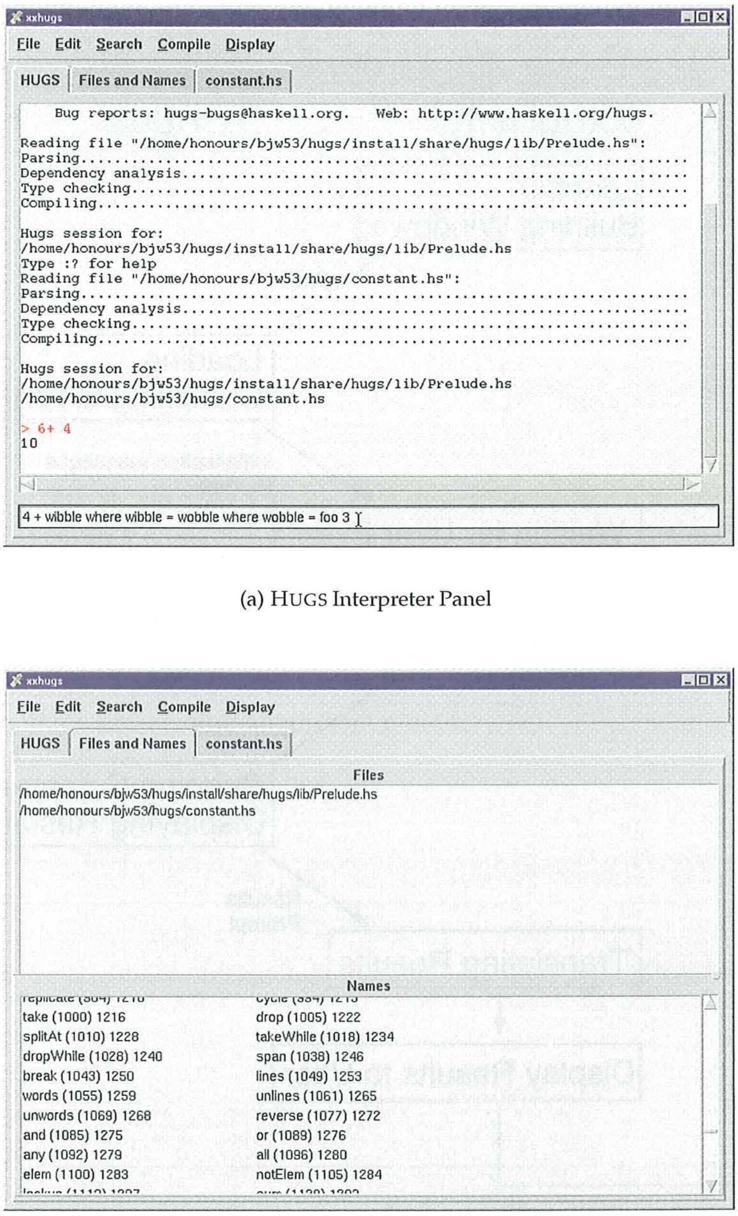


Figure 6.1: xxhugs Execution Flow



```
    Bug reports: hugs-bugs@haskell.org.   Web: http://www.haskell.org/hugs.

Reading file "/home/honours/bjw53/hugs/install/share/hugs/lib/Prelude.hs":
Parsing..................................................................
Dependency analysis......................................................
Type checking............................................................
Compiling................................................................

Hugs session for:
/home/honours/bjw53/hugs/install/share/hugs/lib/Prelude.hs
Type :? for help
Reading file "/home/honours/bjw53/hugs/constant.hs":
Parsing..................................................................
Dependency analysis......................................................
Type checking............................................................
Compiling................................................................

Hugs session for:
/home/honours/bjw53/hugs/install/share/hugs/lib/Prelude.hs
/home/honours/bjw53/hugs/constant.hs

> 6+ 4
10
```

```
4 + wibble where wibble = wobble where wobble = foo 3
```

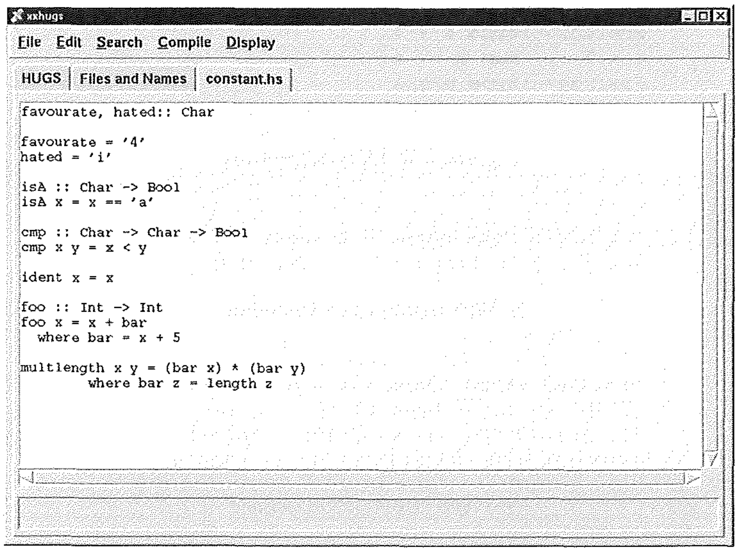
(a) HUGS Interpreter Panel

```
/home/honours/bjw53/hugs/install/share/hugs/lib/Prelude.hs
/home/honours/bjw53/hugs/constant.hs
```

```
take (1000) 1216            drop (1005) 1222
splitAt (1010) 1228         takeWhile (1018) 1234
dropWhile (1028) 1240       span (1038) 1246
break (1043) 1250           lines (1049) 1253
words (1055) 1259           unlines (1061) 1265
unwords (1069) 1268         reverse (1077) 1272
and (1085) 1275             or (1089) 1276
any (1092) 1279             all (1096) 1280
elem (1100) 1283            notElem (1105) 1284
```

(b) Files and Names Panel

Figure 6.2: The xxhugs Display



```
favourate, hated:: Char

favourate = '4'
hated = 'i'

isA :: Char -> Bool
isA x = x == 'a'

cmp :: Char -> Char -> Bool
cmp x y = x < y

ident x = x

foo :: Int -> Int
foo x = x + bar
  where bar = x + 5

multlength x y = (bar x) * (bar y)
        where bar z = length z
```

(c) Source Panel

Figure 6.2: (cont) The `xxhugs` Display

Figure 6.2 shows the display of `xxhugs`. It consists of a single window which contains various panels selected by clicking on the tab-like buttons along the top, below the menus.

The main display for the HUGS interpreter is shown in a panel (Figure 6.2(a)), with a text-input box below it. The user can enter commands to the HUGS interpreter in the bottom text entry box. This will send it to the interpreter which processes the command. The command is printed, with the results, in the scrollable text-output box in the centre of the panel.

Other panels would be used to display the source files (Figure 6.2(c)). The files are opened by `xxhugs` and displayed to the user in a large scrollable text area. These files can be edited and various menu commands would be able to rerun the HUGS interpreter with the new versions of the files. Each file would be displayed in its own panel, as the prototype can make and remove panels dynamically.

6.2 Extracting Information for HUGS

The HUGS interpreter has several built-in commands related to its internal state, the most important of which is the `:type` command. An example of its use is shown in Figure 6.3 (with the user's input in bold text). Note that HUGS has a concept of *type classes*, which is a method of grouping similar data types that share a desired property. `Num` is a type class, whereas `Int` and `Real` are types which are members of the `Num` class. HUGS uses the lower case letters for type variables. The type of `Num a => a` means that `a` is any type that is a member of the `Num` type class.

```
Prelude> :type (+)
(+) ::  Num a => a -> a -> a
Prelude> :type 4 + 7
4 + 7 ::  Num a => a
Prelude> :type until
until ::  (a -> Bool) -> (a -> a) -> a -> a
```

Figure 6.3: HUGS Session

```
Prelude> :type three + 4 where three = 3
let {...} in three + 3 ::  Num a => a
```

(a) With DEBUG_CODE Undefined

```
Prelude> :type three + 4 where three = 3
\d1455 -> let {three($0 (0 (fromInt d1455
3)) $0) $0 $0} in (#0 d1455 + three)
(fromInt (#0 (#0 d1455)) 4) ::  Num a => a
```

(b) With DEBUG_CODE Defined

Figure 6.4: Types of Let Expressions

Other commands were added to HUGS to extract information for the prototype. For example, the commands `:bjw_files` and `:bjw_names` displayed the internals of the `scriptName` and `tabName` structures. This produced the required information that was shown in the "Files and Names Panel" shown in Figure 6.2(b).

In order to implement the error explainers, the system needs to extract the information required from each sub-expression. The built-in `:type` command can extract this information. Furthermore, if HUGS is compiled with the DEBUG_CODE hash-defined to 1, then additional information will be defined. The resulting difference between the output of `:type` is shown in Figure 6.4.

The main problem with using the just `:type` command is that it only works when a source file has been compiled successfully. If the file fails to compile then all the declarations in the source file are forgotten, even the correctly typed ones. Consequently, the system ,if left unchanged, could only explain type definitions that are correct, and not why type inference failed, thus failing to fulfil one of the main purposes of the project.

The internal type information is stored in HUGS in large arrays (as discussed in Appendix B) as it is being compiled. When an error is detected, the links into these arrays are not updated, hence causing the data to be 'forgotten', however the data in the arrays are not overwritten. Extracting this 'unsaved' information for the type explainer was the first step. The `:type` command checked that the information stored was valid (which it was not, as it was 'forgotten'), but it also parses its argument, a process that could overwrite the required, but forgotten, table entries. This was further complicated by the garbage collector of HUGS that could completely rearrange the contents of the array storing the pairs.

This problem is solved by forcing HUGS not to forget the file information, by making it pretend that the source file has compiled correctly. However, do-

```
foo ::  Int -> Int
foo x = x + bar
  where bar = x + 5
```

(a) Source of file

```
Tracing foo
3170 is a name 945
  tabName[ 945].type = -79694
  tabName[ 945].defn = 0
  tabName[ 945].code = 21118
-79694 is a pair (-79693,1438)
-79693 is a pair (1426,1438)
1426 is typecon 1
  typecon[ 1].kind = -13
  typecon[ 1].what = 130
  typecon[ 1].defn = 0
  typecon[ 1].text = 228 "(->)"
1438 is typecon 13
  typecon[ 13].kind = 102
  typecon[ 13].what = 130
  typecon[ 13].defn = 0
  typecon[ 13].text = 1900 "Int"
1438 is typecon 13
  typecon[ 13].kind = 102
  typecon[ 13].what = 130
  typecon[ 13].defn = 0
  typecon[ 13].text = 1900 "Int"
```

(b) Output from trace

Figure 6.5: Tracing `foo`

ing this introduced inconsistencies into HUGS causing it to become unreliable.

The compiler also immediately discards the temporary information generated when it is compiling sub-expressions. This information would normally no longer be required once compilation was successful, but it is important for a type explainer. This information is primarily stored in the arrays for pairs, which are only de-allocated by the garbage collector. Disabling the garbage collector will, therefore, stop this information from being overwritten. This is not an ideal solution as large files will cause the entire HUGS interpreter to run out of memory. However, increasing the number of pairs initially allocated is a simple if inefficient solution. Since the internal 32-bit `cell` value indexed pairs by a negative integer, this causes no problem with pair addressing.

While some routines were able to extract information about global variables (Figure 6.5 shows the result of the `:bjw_trace` command tracing the type of the `foo` identifier), a system-wide type examiner was not completed. Types of sub-expressions within **let** style expressions, as expected, are the most difficult to extract. Due to this difficulty, the prototype type explainers were never fully implemented. We believe that this still can be done, with some rewriting of the HUGS internals and further work in the implementation.

Chapter 7

Conclusions

As discussed in Chapter 4, what a compiler considers to be a type error can differ from a user's perspective, particularly if the user is a novice. General editing mistakes can produce errors that will not be detected until the type checking stage. We divide type errors into two categories. *Attributed* type errors are errors detected by the type checker but are due to editing and other kinds of mistakes. *Explicit* type errors are errors where the user has genuinely made a mistake in the type inference of their expressions. The error reporting mechanism of the compiler should display only the relevant information to the user. However the compiler is not capable of differentiating between the two different categories of type error. Therefore different non-conflicting methods for displaying each category should be available to the user in the editing environment.

Often for attributed type errors the most important aspect of the error is its location. Unfortunately, due to the nature of the type inference and unification algorithms, the source of the error can be different from where its symptoms are first detected. Consequently a method for displaying information about the entire file may be necessary for the error to be traced. Chapter 5 illustrated how this can be done through source highlighting, either by syntax or type.

For explicit type errors, a more detailed break-down of how the types have been inferred is necessary. We show how a tree-browsing interface is more suited for exploring the types of expressions. This would have to be done in a separate display window to the source code, and would need to display to the unification results.

Both of the proposed displays, for attributes and explicit error, require the types of all sub-expressions to be inferred. Consequently, we conclude that a type-explainer which follows the syntax tree is needed to provided all the required information. We present such a system by Beaven and Stansifer.

A partial implementation of a type debugger for the language HUGS was presented and some of the problems with extracting the required information was discussed. Information about global identifiers is particular easy to extract, while information about internal statements, such as the inner **let** statements, are difficult to obtain as the information generated is thrown away immediately afterwards. We believe that such debuggers are still possible, but require significant changes to existing parser and type checking sections of code.

Appendix A

Lambda Calculus

Lambda calculus forms the theoretical basis of functional programming.

A.1 Lambda Expressions

Lambda calculus is a system designed to manipulate lambda expressions[13]. While lambda expressions are simple in content, they are extremely powerful in application and form the building blocks of functional languages. Two concepts which make this true are *abstraction* and *beta reduction*.

Abstraction allows common functions to be generalized into more powerful forms. For example, when we want to square the number 4 we would use the equation 4 × 4. To square the number 6 we use the equation 6 × 6. As the format to square a number is a general form we abstract the number by giving it a name. This name can then be applied instead, as shown in the HUGS function below.

```
square x = x * x
```

In a pseudo-lambda calculus form this is as follows:

$$\lambda x.(\text{square } x)$$

In the function format the formal name is between the λ and the period, while to the right of the period is the actual expression. Note that in the expression *square* is an abbreviation for the squaring function, the definition of which lies beyond the scope of this document.

A formal definition of a lambda expression follows:

```
<expression> ::= <name> | <function> | <application>
<function> ::= λ <name> . <body>
<body> ::= <expression>
<application> ::= (<function expression> <argument expression>)
<function expression> ::= <expression>
<argument expression> ::= <expression>
```

Note that a name can be any sequence of non-blank characters.

A.2 Beta Reduction

While abstraction is a technique used by all styles of programming, beta reduction provides the foundations for functional languages. Beta reduction is the formal name for the replacement of an application with an expression. This can be done using the notation:

(<function> <argument>) => <expression>

In lambda expressions the application is used to determine how the reduction is performed. In the lambda expression:

$$(\lambda g.g)\ x$$

the function expression is $\lambda g.g$ (which is also known as the identity function) while the argument expression is x. To reduce this, the bound variable in the function expression (which is g) is replaced with x. The steps are listed below:

$$\begin{aligned}&(\lambda g.g\ x)\ =>\\&x\end{aligned}$$

Consider, however, another lambda expression:

$$\lambda f.\lambda a.f\ a$$

This is also known as the apply function, which will apply an argument to a function resulting in a single expression. Below is an example of it in action, where it is used to resolve the identity function with the argument g.

$$\begin{aligned}&(\lambda f.\lambda a.(f\ a)\ g)\ \lambda x.x\ =>\\&(\lambda a.(\lambda x.x)\ a)\ g\ =>\\&(\lambda x.x)\ g\ =>\\&g\end{aligned}$$

While these examples are very simple they do demonstrate the basic definition of lambda expressions.

Appendix B

The Internals of the HUGS Interpreter

B.1 Overview

This appendix describes the internals of the June 1998 version of HUGS, an interpreted version of Haskell with an emphases on type checking. HUGS is written in C but, unlike most C programs, some of the `.c` files include other `.c` files. Figure B.1 is a diagram showing which .c files include others.

Only two of the files are relevant to this discussion: `storage.c`, which is where the data is actually stored, and `type.c`, which has the type checker built into it.

B.2 Basic Data Storage

Internally, all data is represented by an atomic unit, called the *Cell*, which is represented by a signed 32 bit integer. Table B.1 shows the correlation between the Cell's integer value and the data type it represents. For example the character `'a'` is 12,102 (CHARMIN+97), while the integer value zero has a Cell value of 1,073,751,453.

Negative Cell values refer to pairs, which the interpreter uses as the basic building block for storing its internal trees and lists. The values are stored in two large arrays of cells, typically 100,000 entries long. The values are referenced from the top of each array, the first from heapTopFst and the second from heapTopSnd, through the fst() and snd() macros respectively.

An example of how values are stored is shown below. The following HUGS code is compiled from a file called `constant.hs`.

```
favourite, hated :: Char
favourite = '4'  hated = 'i'
```

The *favourite* constant becomes internally represented by the Cell value of 3165 and the *hated* constant by 3166. These Cell values are used as offsets (once NAMEMIN is subtracted) into an array which contains all the name information. The tabName[] structure gives the following information:

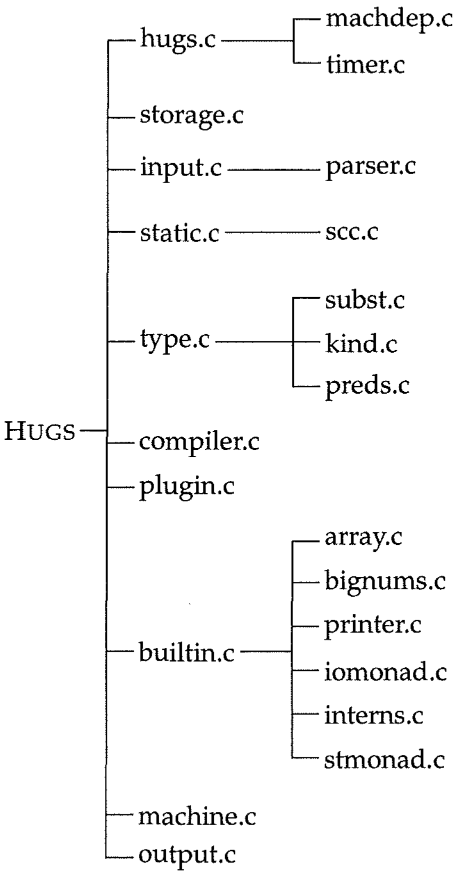


Figure B.1: HUGS Source File Tree

Starting Value	**Range Size**	**General**
Negatives	< 0	Pointers to pairs
1	200	Special values
TUPMIN 200	NUM_TUPLES 100	Tuples
OFFMIN 300	NUM_OFF 1024	Generic types/stack offsets
MODMIN 1325	NUM_MODULE 100	Modules
TYCMIN 1425	NUM_TYPCON 800	Type constructor names
NAMEMIN 2225	NUM_NAME 16000	
SELMIN 18225	NUM_SELECTS 100	Dictionary selectors
INSTMIN 18325	NUM_INSTS 600	Instances
CLASSMIN 18925	NUM_CLASS 80	Class
CHARMIN 19005	NUM_CHARS 256	The characters
INTMIN 19261	>= INT_MIN	The integers
(INTMIN + MAXPOSINT)/2		INTZERO

Table B.1: The Basic Cell Ranges

```
tabName[3165-NAMEMIN].text         = 5620
tabName[3165-NAMEMIN].line         = 3
tabName[3165-NAMEMIN].mod          = 1326
tabName[3165-NAMEMIN].parent       = 0
tabName[3165-NAMEMIN].arity        = 0
tabName[3165-NAMEMIN].number       = 0
tabName[3165-NAMEMIN].type         = 1433
tabName[3165-NAMEMIN].defn         = 0        --> 19057
tabName[3165-NAMEMIN].code         = 21084
tabName[3165-NAMEMIN].primDef      = 0
tabName[3165-NAMEMIN].nextNameHash = 0
```

The .text field is the offset into the text array where all textual items are stored. The .line and .mod fields are references to the module and the line number where the name is defined. The .type field is set to the Cell value which represents the Char type. The code field points to the section of code which will evaluate the expression. The first time it is evaluated, the .defn field is changed to 19,057, which is the Cell representation of the character '4'.

B.3 Type Representation

Types are represented internally by the command Cell within the range from TYCMIN (1425) to TYCMIN + NUM_TYPCON - 1 (2224). The Cell value (minus TYCMIN) is used as an offset into an array which contains the type information. The entry for characters looks like this:

```
tabTycon[1433-TYCMIN].text          = 1888
tabTycon[1433-TYCMIN].line          = 456
tabTycon[1433-TYCMIN].mod           = 1325
tabTycon[1433-TYCMIN].arity         = 0
tabTycon[1433-TYCMIN].kind          = 102
tabTycon[1433-TYCMIN].what          = 130
tabTycon[1433-TYCMIN].defn          = 0
tabTycon[1433-TYCMIN].nextTyconHash = 0
```

The .kind and .what combination specifies that it is a basic type construction that was declared in .mod 1325 which is the 'prelude.hs' library file defined at line 456. Note that kind is usually set to the value 102, which represents the basic types.

Classes are stored by a similar method. The Cell value (minus CLASSMIN) is used as an offset to the tabClass array which contains the class details. For example, the 'Num' class is defined as follows:

```
tabClass[18928-CLASSMIN].text       = 154
tabClass[18928-CLASSMIN].line       = 152
tabClass[18928-CLASSMIN].mod        = 1325
tabClass[18928-CLASSMIN].level      = 2
tabClass[18928-CLASSMIN].sig        = 102
tabClass[18928-CLASSMIN].supers     = -1356
tabClass[18928-CLASSMIN].numSupers  = 3
```

```
tabClass[18928-CLASSMIN].members    = -17706
tabClass[18928-CLASSMIN].numMembers = 8
tabClass[18928-CLASSMIN].defaults   = -17722
tabClass[18928-CLASSMIN].instances  = -18437
```

The Cell pairs from the `.super` field form a linked list showing the classes ('Eval', 'Show', and 'Eq'). The linked list from the `.members` field shows a linked list of the names ('+', '-', '*', 'negate', 'abs', 'signum', 'fromInteger', and 'fromInt').

```
&text[154] = "Num"

heapTopFst[-1356] = 18943    heapTopSnd[-1356] = -1352
heapTopFst[-1352] = 18938    heapTopSnd[-1352] = -1351
heapTopFst[-1351] = 18925    heapTopSnd[-1351] = 0

heapTopFst[-17706] = 2418    heapTopSnd[-17706] = -17708
heapTopFst[-17708] = 2419    heapTopSnd[-17708] = -17710
heapTopFst[-17710] = 2420    heapTopSnd[-17710] = -17711
heapTopFst[-17711] = 2255    heapTopSnd[-17711] = -17713
heapTopFst[-17713] = 2421    heapTopSnd[-17713] = -17715
heapTopFst[-17715] = 2422    heapTopSnd[-17715] = -17717
heapTopFst[-17717] = 2423    heapTopSnd[-17717] = -17719
heapTopFst[-17719] = 2424    heapTopSnd[-17719] = 0
```

B.4 Resolving Types of Expression

When HUGS is given an expression it is broken into a binary tree and stored in the Cell pairs. Before it is evaluated, its type is determined.

If the following example is entered:

```
if True then 'r' else 't'
```

the entire expression is represented by a single Cell -78654. The Cell pairs look like this:

```
heapTopFst[-78654] = 21      heapTopSnd[-78654] = -78653
heapTopFst[-78653] = 2317    heapTopSnd[-78653] = -78652
heapTopFst[-78652] = 19119   heapTopSnd[-78652] = 19121
```

The value 21 is the reserved value code for conditionals. 2317 is the name 'True' which has a type of Bool. 19119 and 19121 are the respective Char values 'r' and 't'. In the above expression all types can be identified simply by checking the range of the Cell values.

When this code is type checked, the following code is run:

```
#define check(l,e,in,where,t,o)  \
     e=typeExpr(l,e);shouldBe(l,e,in,where,to,o)

  Int beta = newTyvars(1);
```

```
check( 1, fst3( snd( e)), e, cond, typeBool, 0);
check( 1, snd3( snd( e)), e, cond, var, beta);
check( 1, thd3( snd( e)), e, cond, var, beta);
tyvarType(beta);
```

The first line of the code, after 'define', creates a new type variable and assigns its offset to `beta`. The next line then gets the conditional text expression (in this case 'True'), determines its type and then attempts to unify the type to Bool using the shouldBe() function. The third line gets the then-expression and attempts to unify its type (unifying it to the new initialised type variable will always work). The fourth line then gets the else-expression and attempts to unify it to the type given by the then-expression. The last line then sets the resulting type into various global variables.

The heart of the unification is in the unify() function from `subst.c`, which the ShouldBe() macro calls. The unify sorts through various Tyvar structures that it allocates as required, checking each one for the least binding value.

B.5 Type Checking A Simple Application

Another example of the type unification at work would be the following code, which is an example of an application.

```
isA :: Char -> Bool
isA x = x == 'a'
```

When this is compiled the name 'isA' becomes represented by the Cell value of 3167. In the name structure the `.arity` field is set to 1, as it has one argument. Its `.type` field is set to -70533. Examination of these Cell pairs shows the following:

```
heapTopFst[-79533] = -79532  heapTopSnd[-79533] = 1432
heapTopFst[-79532] = 1426    heapTopSnd[-79532] = 1433
```

The Cell values 1426, 1432, and 1433 are all offsets to type constructs which refer to `'(->)'`, Bool, and Char. Thus the type checker knows that `isA` has the type 'Bool -> Char'.

When asked to evaluate `isA 'r'` the debugging output looks like this.

```
17) to check: isA 'r'
new type variable: _0 ::: *
new type variable: _1 ::: *
new type variable: _2 ::: *
binding type variable _0 to _1 -> _2
18) to check: isA
18) result: Char -> Bool
tt unifying types: Char->Bool with _1->_2
vt binding type variable: _2 to Bool
vt binding type variable: _1 to Char
19) to check: 'r'
19) result: Char
```

```
tt unifying types: Char with Char
17) result: Bool
tt unifying types: IO () with Bool
False
```

Unify()'s first step allocates three blank structures to store type constructor information that will be generated by the expression. One constructor is used to store the resulting type of the expression, while the other two are used to form the (type)->(type) that is directly implied from the two inputs. It then binds the _0 result value to the other two.

The type check then determines what the `isA` type is, which it then binds to the constructors. Note that when _2 is bound with Bool so is _0, due to the first binding. It then determines the type of 'r' which is bound trivially, then the expression is evaluated, and the result is bound to IO () which prints it.

In the binding notes above, the t's refer to constant types while the v's refer to variable types, so `vt binding type variable:  _1 to Char` means that the type variable _1 is being bound to the constant type Char.

B.6 Another Example

Consider the identity function `ident x = x`. When examining the internals of the name structure we get a more complicated pair list.

```
ident = 3169

tabName[ 3169-TYCONMIN].type = -79624

heapTopFst[-79624] = 50      heapTopSnd[-79624] = -79623
heapTopFst[-79623] = -79622  heapTopSnd[-79623] = -79618
heapTopFst[-79622] = 102     heapTopSnd[-79622] = 102

heapTopFst[-79618] = -79617  heapTopSnd[-79618] = 301
heapTopFst[-79617] = 1426    heapTopSnd[-79617] = 301
```

The value 50 is a special value reserved for POLYTYPE. The second value of the first pair refers to a pair which contains its kind and type. The kind of both is set to 102, which is a special value meaning that the kind is a normal type. The type values map to 301, 1426, and 301. Note that 1426 is the `'(->)'` type constructor. 301 are cells that are offsets. An offsets is a special kind of type constructor variable which is a pointer to the stack of the machine when it is running. It allows HUGS to refer to types in its stack while reducing the need to keep allocating type constructors for some of its immediate values. Obviously the identity function returns the same type as its argument.

When this command is evaluated with the command `ident 't'` the following output is produced:

```
18) to check: ident 't'
new type variable: _0 ::: *
new type variable: _1 ::: *
new type variable: _2 ::: *
```

```
binding type variable _0 to _1 -> _2
19) to check: ident
new type variable: _3 ::: *
tt unifying types: _3-> _3 with _1 -> _2
vt binding type variable: _3 to _2
vt binding type variable: _2 to _1
20) to check: 't'
20) result: Char
vt binding type variable: _1 to Char
18) result: Char
tt unifying types: IO () with Char
't'
```

The function starts off as before, allocating three type constructors and binding them. However, as the type of `ident` is polymorphic, it creates an extra type constructor which is bound to the other variables. When the type of 't' is determined it is bound directly to _1. When `ident` is run on other constants of differing types the debugging output is the same.

Bibliography

[1] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1):17–30, March 1993.

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

[3] Guy Cousineau and Michel Mauny. *A Functional Approach to Programming*. Cambridge University Press, 1998.

[4] Dominic Duggan. SML/E: A type explanation facility for standard ML. `http://guinness.cs.stevens-tech.edu/~dduggan/smle/`, 26th January 1999.

[5] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[6] Anthony J Field and Peter G Harrison. *Functional Programming*. International Computer Science Series. Addison-Wesley, 1989.

[7] R Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the American Mathematics Society*, 146:29–60, 1969.

[8] M Jones. *An Introduction to Hugs v10.01*. University of Nottingham, 1994.

[9] J. W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation. Artificial Intelligence. Springer-Verlag, second edition, 1987.

[10] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming*, volume 2. Deductive Systems. Addison Wesley, 1990.

[11] Bruce J McAdam. On the unification of substitutions in type inference. *Lecture Notes in Computer Science 1595, Implementation of Functional Languages*, IFL'98:137–152, 1998.

[12] Bruce J McAdam. On the unification of substitutions in type inference. Technical Report ECS-LFCS-98-384, Laboratory for Foundations of Computer Science, The University of Edinburgh, UK, March 1998.

[13] Greg Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. International Computer Science Series. Addison-Wesley, 1989.

[14] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[15] Chris Reade. *Elements of Functional Languages*. Addison Wesley, 1989.

[16] J. A. Robinson. A machine-orientated logic based on the resolution principle. *Journal or the*, 12:23–49, 1965.