University of Edinburgh
Division of Informatics

# A Tool for Investigating Type Errors in SML Programs

Undergraduate Dissertation
Computer Science

Iain Denniston

June 1, 2001

**Abstract:** This project aims to create a tool to aid the location and correction of type errors in ill-typed SML programs. Specifically, it will provide the facility to determine the type of an expression or part thereof, and also to assume a different type for such an expression. The implementation should be integrated into a popular existing SML compiler and should provide a graphical user interface.

# Preface

## Assumptions and Notes

It is assumed for the purposes of this paper that the reader has a working knowledge of SML; i.e. has written (a few) SML programs, and is aware of the basic syntax of the language. Readers unfamiliar with SML should locate one of the many texts on the language (for example [Paulson1996]) and familiarise themselves with its workings.

It should be noted that on occasion some words in this document are used to convey different meanings in similar contexts. This is unavoidable and is due to the overuse of particular words in computer science generally. Wherever possible an alternative word has been used but certain words could not be replaced with synonyms. The words that are likely to cause the most difficulty are “type” and “expression”; however, the reader should be able to determine the correct meaning from the context.

In addition, although the premise of this project extended to any computing platform and operating system, the implementation discussed was created on an Intel x86 architecture running Mandrake Linux 7.1These factors, however should not have any bearing on the outcome.

## Acknowledgements

I would like to thank, my supervisor, Don Sannella and, my director of studies, Murray Cole, for their guidance and support through out this project.

I also would like to thank both my families, Dennistons and Wrights for their support and understanding throughout the year.

# Contents

# 1. Introduction

## 1.1 The Problem

Programming is not an exact science and there are an almost infinite number of ways to write a program to do any one task. Since this is the case, there are an almost a limitless number of errors that can accidentally be written into a given program. Locating and eradicating those errors is a tedious, time-consuming and expensive task, therefore it would be of great benefit to software engineers if any errors could be found and corrected more easily.

In view of this, there have been many attempts to ensure that programs containing certain kinds of bugs cannot be compiled; Standard Meta Language (SML or simply ML) contains one of the better examples of this. The approach present in ML is to use a strongly typed language, that is, all elements of ML have a type and can only be used to interface with other elements of a compatible type.

While this approach is largely successful, and it is much more likely that an ML program is correct (i.e. devoid of programming errors) than a similar program written in a language such as C, the approach introduces its own unique difficulties. The most severe of these, from the perspective of the ML programmer, is the type error.

For an ML programmer, type errors represent the most frustrating of all errors to locate, understand and correct. If there were a way to easily repair all type errors then programming in ML would be both simpler and more popular.

This project then, is to attempt to address the issue of type errors and how to quickly locate, understand and correct any type conflict.

## 1.2 Previous Approaches

Given that type errors are a major difficulty faced in writing an ML program, and the larger a program becomes the harder it often becomes to discover the real source of the error, it is hardl y surprising that there has been a significant amount of previous work in this area.

There are several approaches to the problem, the most obvious of which are: giving the programmer more information about the error in question, or giving the programmer a better idea of where the error actually occurs, as it can be located in an entirely separate place in the code from where the error was actually reported. A summary of various different methods for debugging type errors is available in the paper "Generalising Techniques for Type Debugging" by Bruce M$^c$Adam [M$^c$Adam]

The approach used by this project follows on from work by Alison Keane [Keane1999] (see [Section 1.4]) and attempts to provide the programmer with a way to investigate type errors for herself in an interactive fashion, instead of simply providing more information for her.

## 1.3 My Approach

As noted, and is obvious from the title of the project, the approach to understanding type errors used by this project is one of investigation. That is, the programmer should be allowed to and encouraged to investigate type errors for herself, attempting to understand what went wrong and why. It is hoped that this approach will bring a new level of understanding to programmers, especially for beginners, and by locating and solving type errors more readily, it should be easier to avoid making the error in the future.

The aim of this project is to provide a tool that can be run when an ill-typed ML program is applied to an existing ML compiler. The tool will allow the programmer to check the type of expressions within the program, to verify that they are as expected, and then to alter the type of any expression in the program that does not have the type expected, without having to rewrite the code. It should be noted that such an altered program can never be compiled rather the programmer is assisted in locating the error in question, and discover exactly what caused it, allowing her to change the source code as required, thereby fixing the error. The altered source code would then be copied by the user to the compiler.

This approach to the problem is a variation of the trial and error approach that most programmers will use to fix such programs anyway. The project simply aims to provide a tool that makes such error location easier and more methodical. In addition, it allows the compiler to take over some of the work the user may have to do normally, such as finding the type of any given expression within the program – which is much less error-prone.

## 1.4 Previous Work with the Same Approach

Alison Keane's project of a similar title [Keane1999] used the same approach, however it had several problems, which resulted in it being restricted in use to nothing more than a research tool.

Its major failings were: firstly, it did not cover the whole of the ML language, but only a small sub set of the language. Secondly, the project was not integrated into an existing ML compiler. Its third and final failing was that it had a poor user interface. It did however establish the viability of the general approach.

Hence, the aim of this project is to correct all the problems noted in the previous work in this problem area by:

- developing a tool that is integrated with the popular Moscow ML system, which provides a full implementation of SML
- provide a more carefully-designed user interface.

# 2. Type Checking

## 2.1 What Is Type Checking And Why Have It?

Types are inherent to all languages, be they computer languages or natural languages, and all elements of any language have some type. For instance, in natural (spoken) languages, the elements are words, and they have types such as nouns and verbs. Types are used to define how elements of a given language can interact; this interaction is defined in a grammar for the language.

The grammar of a given language describes how a phrase can be built up from the elements of that language. Grammars are based on rules, and to create a correct, or valid, phrase for a language, there must be a precise application of those rules that exactly describe the phrase. If such an application of the rules does not exist then the given phrase is not part of the language described by the grammar.

To avoid having to list all possible combinations of elements that belong to a language, the rules of a grammar are made by stating which types of elements can go where in a valid expression. Each element is then classified as a type, and as such, it is possible to define precisely where each element can and cannot go in a phrase belonging to that language.

In natural, or spoken, languages, it is acceptable to bend or even break the rules in a given grammar, so long as the meaning of the phrase is clear to all parties concerned. In computer programming, things are very different. Computers are deterministic and as such need a specific input to generate a specific output. Hence, grammars for most, if not all, programming languages are very strict and must be rigorously followed if the programmer wishes to create a correct program. Some computer programming language grammars are stricter than others are, but all imply a level of strictness much greater than is applied to natural languages.

Computer languages generally include several different types, such as: int, real, bool, etc. Furthermore, expressions have types: 1+2: int, true or false: bool. In many languages, it is easy to convert from one type to another. For instance, in C, some types are considered interchangeable; e.g. (short) integers and characters are essentially the same on many architectures. As such the compiler (more precisely the type checking part of it) will allow them to be interchanged throughout a program. This however, can be the cause of many errors in programs written in such languages.

There is another group of languages that does not allow any interchanging between expression types, that is, these languages only allow the expressions of the exact types required to be used in a given circumstance, and if more than one expression type would suffice, then the type of the expression used must be compatible with the rest of the program. ML is a one of the most popular of this group of languages, which are known as strongly typed languages.

Strongly typed languages are often preferable, since they do not allow many, if any, “unsafe” programmes to be compiled [Sethi1989]. In fact, it is far more likely that a strongly typed program is correct (i.e. will work as designed for the purpose given) than an equivalent non-strongly typed program.

It should be obvious; therefore, that type checking is not only a desirable, but also a necessary aspect, of these strongly typed languages.

## 2.2 How Type Checking Is Accomplished

There are several built in types and operators on those types, in every programming language. The least a programmer might expect would be the integer type, and the basic mathematical operations (addition, subtraction, multiplication and division). In such an over simplistic language, each of the operations would be expected to be defined as a binary operator that is, each takes two arguments, combines them in some known way and returns a result. The two arguments would be expected to be of the type integer.

If the language was extended to include strings, then it is necessary to be able to check that the mathematical operators were not applied to the strings, since this is unlikely to make sense. To do this would be quite simple. All that would be required would be that when an expression was used with an operator, the type checker would check the type of that expression. If the expression's result was of type integer, then the whole expression, with operator, would be a valid expression and the type checker could proceed to the next expression. If, however, the expressions result were of type string, then the type checker would note that this was a type error and inform the programmer.

Complications arrive due to the nesting of expressions. Take, for example, simple addition. Addition, can be nested as follows:

$$(1 + 2) + (3 + 7)$$

This causes problems for the type checker as the arguments to the second '+' are themselves expressions. The type checker must then ensure that the result of these two expressions will be of type integer in order to satisfy the criteria that the '+' operator takes two arguments both of which are integers. In this case, it is simply a case of noting that the result of the addition of two integers is known to be itself an integer, hence if the two sub-expressions are correct then the overall expression will be correct. Therefore, the type checker must evaluate all sub-expressions before evaluating the main expression.

Although the above case is somewhat trivial, it displays the possibility that type checking has to be quite complex. For instance, there is an almost infinite level of nesting that could occur in any expression. If then the above, trivial, language is extended to include many more types, branching and looping constructs, and user defined functions, it is clear that the problem quickly becomes complicated.

Even though the task may become more difficult as the size and complexity of a given program increases, basic type checking follows the same simple rules. These rules are to; firstly, do a depth first traversal of the tree that the program describes that is, type check all sub expressions before type checking expressions. Then, when type checking a leaf of the tree, that is type checking an atomic expression (one with no sub-expressions) look up (in a simple table created for the program) all variables and operations used in the expression, ensuring that they are of compatible types. If they are compatible types, then the resultant type of the expression will be known from looking up the table mentioned; this type is then used as the overall type for this expression in any expression of which this one is a sub-expression. The look up operations would be made to a table that is edited as the program is type checked. Such a table would hold the types of all the inbuilt operations and variables, and would then be appended to with the types of any user defined variables and operations (and their scope in the program).

## 2.3 Complications Type Checking Encounters, And Their Solutions

As shown, basic type checking can be extremely simple, and even at its most complex it is a case of following several simple rules that will lead to the required result. Complications arise however, due to two other features present in some programming languages, including ML; polymorphism and user defined datatypes.

### 2.3.1 Polymorphism and User Defined Datatypes

Without polymorphism, and user defined datatypes, writing programs would be much more tedious than it currently is. These concepts are present in most programming languages to some extent, and provide a shortcut to the programmer, allowing for the faster and more efficient development of programs.

Polymorphism when referring to types is the ability of an operation to be applied to more than one type of argument; it can also be applied to variables where it refers to the ability of that variable to take more than one type of value. An example of a common polymorphic operation is the addition operator. In most programming languages, the infix operator "+" is used to represent addition, much as it is in normal mathematics. However, in computers the addition of integers and floating-point numbers is accomplished via very different algorithms and often by completely different hardware. Despite this the same operator, "+", is used to represent both types of addition, hence the following are both legal:

```
1 + 2
1.5 + 2.3
```

The fact that the same operator is used in both cases, yet the algorithms invoked at each point are different, is polymorphism. This type of polymorphism is often termed overloading polymorphism. Overloading polymorphism (normally termed simply overloading) describes what happens in the situation that occurs when different algorithms are used for the same operator, depending on the type of the arguments given. A second, perhaps more interesting, kind of polymorphism is where one algorithm is used regardless of the types of the arguments used.

Such polymorphism can be seen in use in a function to calculate the length of a list in ML. Here a list is simple a collection of like typed elements, denoted by square brackets surrounding the collection each element of which is separated by a comma. For example:

```
        [1,2,3]
        [‘a’,’b’,’c’]
        []
```

Are all lists; the first is a list of integers, the second is a list of characters, and the third is the empty list.

As should be obvious, the type of elements in a list does not affect the length of a list. Indeed, the first two lists above have different types of elements yet have the same length. A function to calculate the length a list in ML could be written as follows:

```
        fun Length  []          = 0
        |     Length (hd::tl) = 1 + Length tl;
```

If polymorphism were not available then such a function would not be possible, instead, there would have to be one function for each of the different type of list possible. For example, one function for list of integers, one function for lists of characters, etc. This type of polymorphism is often termed parametric polymorphism. [Sethi1989]

User defined datatypes similarly make programming a much easier task. User defined datatypes are classes of data structures constructed by the user (programmer) in such a way as to aid both understanding and easy of use of the program. User defined datatypes can be used to make very complex structures, for instance trees and linked lists. They can also be recursive and can often be nested; that is, it is pos sible to use a previously user defined datatype in another user defined datatype.

Given that user defined datatypes in themselves, can be extremely complex, it is not surprising that they also complicate type checking, although not to the same extent as polymorphism.

## 2.3.2 Type Checking with Polymorphism and User Defined Data Types

Polymorphism is the harder of the two noted complexities to deal with. User defined datatypes can be deceptively easy to type check depending on the constraints imposed by the language. As far as user-defined datatypes are concerned, there are three types of language; those that do not allow user defined datatypes, those that allow user defined datatypes but have a strict equality condition, and those that allow user defined datatypes and have a weak equality condition. Clearly, the first type of language is of no concern in this section, whereas the second and third are.

Languages that allow user-defined data types with a strict equality will only recognise two types as identical if they are both declared as the same type. This means that two different variables of different types but with identical structure and values will not be seen as the same. For example, if a user defined data type was created that simply provided a different name for the integers, a direct comparison between the integers and this new type would not be possible since the type checker would perceive them as entirely different. For this type of language, type checking is a case of doing as for the basic case, but is slightly more complicated because the lookup table must be extended to hold information of any new type defined.

On the other hand, languages that allow user defined data types with weak equality are much more difficult to type check. In this kind of language, two variables can be considered equal if they have the same structure and value, regardless of their actual type. Using the example of a new type, which simply renames the integers from above, this kind of language would allow a direct comparison between variables of these two types since they will always have the same structure, and will be equal if they have the same value.

This third kind of languages, is much more difficult to type check since the type checker must be able to verify that the types of expressions used as arguments to operations are structurally equal to the types required. Given that the language of type definitions is very powerful, with different constructors carrying values of different types, together with recursion, this then becomes a complex and difficult task. The simplest way to deal with this kind of type checking problem is for the type checker to compare types as they are defined. If a new type were defined that is the same as a previous type, then the type checker would simply note that these two types were equal in its lookup table, thereby making it easy to compare them. However, if two types are actually the same, but with alternative definitions, then difficulties can arise, for example:

```
        datatype IorF = I of int | F of real
        datatype ForI = F of real | I of int,
```

both of which are identical in structure. However, since the ordering differs, the type checker must be able to spot that these are the same regardless of the order in which they were defined. The easiest way to accomplish this is to include some kind of absolute ordering for datatypes, that is, when the type checker checks a new data type, it will order the clauses of the new data type in an absolute order (for example, alphabetical order by constructor name). All data types will be ordered in an identical order so that equal data types can be spotted easily.

Polymorphism poses a much greater problem to type checking than user-defined data types. The problem is that the type checker must be able to work out what type any given expression in a given program has at a given point, however, with polymorphism and the standard types an expression may have many different types. Take for instance the list length function mentioned earlier:

```
        fun Length  []        = 0
        |    Length (hd::tl) = 1 + Length tl;
```

With only the basic types available, the overall type of this function cannot be identified, since it can be applied to many different types of list. Given this problem, to be able to adequately type c heck such a function, the set of types must be expanded to include the set of polymorphic types. Each of these polymorphic types is compatible (in isolation) with any other type that either does not contain the same polymorphic type or the types are equal. The polymorphic types are represented by the first letters of the Greek alphabet.

It is now possible to see that the type of the list length function as written above is:

$\alpha$ list $\rightarrow$ int

That is, the function can take any kind of list as an argument and will return an integer result. The type variable $\alpha$ can be instantiated with any type to give, for example, int list $\rightarrow$ int, (bool * int tree) list $\rightarrow$ int or $\beta$ list list $\rightarrow$ int. The problem now is how the type checker ensures that expressions with polymorphic types are correctly typed.

Initially when type checking a function such as the list length function, the type checker will assume that each of the variables, arguments and expressions of that function have different polymorphic types. The type checker will then attempt to create a type binding which is as general as possible, while still being compatible with the usage of functions, variables etc having known types.

To begin with this involves looking up the table that contains a list of all the variables, constants and functions declared so far, and matching any variables, constants and functions in the function definition with ones in the table. Once any of these have been matched and the types of any literals in the code have been worked out, then th e types for parts of the function may be known. The type checker will then go through a process known as unification.

Unification attempts to do two things: firstly it attempts to ensure that any polymorphic types applied to each other are compatible, then it attempts to create the single most general type it can from the unification of these two polymorphic types, that new type is then used to work out the type of the application.

Length is being defined as a function so its type is taken to be $\alpha \rightarrow \beta$ initially. In the second line, its result is added to 1 so $\beta$ is unified with int to give Length the type of $\alpha \rightarrow$ int. In the first line, Length is applied to "[]" having type $\gamma$ list so $\gamma$ list is unified with $\alpha$ to give Length the type $\gamma$ list $\rightarrow$ int. This type is compatible with all the other type information in the function definition so it is the final type of Length, apart from the cleaning up of type variable names to give $\alpha$ list $\rightarrow$ int.

In the case where the types being unified are the same, then unificati on is trivial. In the case where one of the types being unified contains the other type being unified then unification does not take place and a type error is reported. This is to avoid the possibility of infinite types, which would be the result if unification were allowed to take place. This is known as circularity. An example of types that would cause this would be $\alpha$ and $\alpha$ list. Attempting to unify these types would cause circularity because initially, the $\alpha$ would be instantiated to $\alpha$ list, however, the original $\alpha$ list type would have to be instantiated to be $\alpha$ list list. Now, the types would be $\alpha$ list and $\alpha$ list list, however, the unification could not stop here since the types are not yet equal, and hence unification would continue indefinitely.

## 2.4 Problems Due to Type Errors, and Their Solutions

The greatest difficulty encountered in type checking, from the programmers perspective, is the type error. Although type errors can find many errors that otherwise have been overlooked, they are often difficult to understand for the following reasons. Type errors occur when the type checker discovers types in the program that are not compatible. For instance the following example would cause a type error in most programming languages:

```
“s” + 1
```

The main difficulty raised by such errors is that it is not always obvious where the problem originates. Because of the flexibility introduced by polymorphic types, the position where the type checker reports the error is often not the actual source of the problem. Rather the actual source of the problem occurs in a completely different place from the reported position of the error. For example, consider the ML code:

```
let
	val x = “s”;
	val y = 2;
in
	x + y;
end;
```

Here the type checker would find the error on the fifth line where x is added to y. However, the real source of the problem is actually on line two, where the x is declared as a string. Although the above example is somewhat trivial, it adequately displays the problem and it is clear that it could in fact become more difficult to solve. For example, if x had been declared in another module, and in another file, in that case, the error may be even more difficult to locate.

Type errors are the major problem for any ML programmer, particularly beginners and polymorphism exasperates the problem. Polymorphism can cause type errors to become more confusing and less specific than otherwise would be possible. For a detail example of this, see [Section 5.1] in chapter 5.

Currently, the only solution to the type error problem implemented by the compilers is to give the programmer an error message detailing the position of the error and what its cause at that point. However, as noted earlier the position of the error is often not where the source of the problem lies, and the error messages provided can often be very confusing.

# 3. User Interface

## 3.1 User Interface Options

Clearly any interactive computer program written must have some kind of user interface. Currently there are two main options for general software user interfaces; command line (i.e. text based) and graphical user interface (GUI), each having its merits and demerits.

## 3.2 Why Choose A GUI For The Tool?

At present, for most new programs a GUI seems to have become almost mandatory. This is understandable given that a GUI is normally much easier to learn and use than a command line based tool. In most circumstances, a command line tool requires a greater degree of understanding and depth of knowledge on the part of the user than a GUI based tool of similar nature. This is because with a GUI everything necessary is (visibly) at users fingertips.

In any program where the selection of text is required (as is true in this tool), then a command line tool runs into immediate difficulties. Such a tool would require the user to either count characters, words, lines, phrases or any combination thereof to specifically identify the portion of text required to be selected. Not only is this tiresome for the user, it is prone to error, increasingly so as the text entered grows in length. If an error was made in the counting, it may not be immediately obvious to the user that it has occurred. This may then lead to an inaccurate result. As such, a command line based version of this tool would be more likely to be counter productive. This leaves a GUI as the only real option.

This is not a problem in a GUI. The problem of selecting a portion of text is simplified to clicking and dragging the mouse. In the case that the selection was not what was wished, it should be immediately obvious as the area highlighted by the selection (as happens in most, if not all, GUIs) would not be what the user wanted. In addition, a GUI is much easier to learn and use, and is almost certainly more accessible to novices.

## 3.3 Design Goals for the GUI

Unfortunately, GUIs currently created for tools range from the very good to the very poor. The vast difference in the qualities of GUIs is probably due to two major factors; one is that not enough time is spent on designing the GUI, the other being that the GUI is normally designed by the creator of the software and not by the user. The first factor can be caused by the pressure to create a good working tool, which then leaves little time for the interface to be designed. The second factor is one that is very common, where the creator of the software decides what the user will and will not want to do which is often very different to what the user actually requires. Therefore, it is clear that defining the goals for the GUI is not only important, but has to be done well if a good GUI is to be created.

Obviously, the main goal for the design of a GUI, or any user interface for that matter, should be that it is easy to use. The problem is to define what that means in the context of this tool. There are a few obvious factors, which must contribute to this "easy to use" description of the interface. Firstly, the selection of a specific portion of text must be easy to do precisely. This requires that the user interface be responsive to the users actions – interfaces that suffer from long delays between user actions and program responses will often cause the user to make errors. In addition, the interface must be clear; that is, the text must be easily readable and selected text must be significantly different to non-selected text.

Secondly, any actions possible must be clear to the user. This means that any buttons or menu items must be clearly named and in an order that is both consistent with the rest of the users chosen interface (i.e. window manager) and groups operations in a logical and sensible order.

Thirdly, the GUI must provide all the possible functionality that is available. There is little point in having a GUI that does not provide the functionality that a command line interface for the same tool would provide. However, not all functionality needs to be immediately obvious to the user, as long as it is available via some means.

The final goal of a GUI is that it should be aesthetically pleasing, though this is not as important as any of the other noted goals. Being a subjective goal it is more difficult to accomplish, however it should be possible to design an interface, which is generally pleasing.

## 3.4 General GUI Design Problems

The whole area of human computer interaction (HCI) is a rapidly growing area of study, and there have been many texts published on such interactions, for example [Dix1998]. The main goal of such study is to understand both the technology and the users sufficiently to ensure that it is possible to build intuitive and good user interfaces for the technology that meet as many of the users' needs as possible. Alas, most of the GUIs currently available to users are based more on tradition and the programmer's views about what is good, than are based on scientific premises.

The main problems generated in the designing of GUIs are the same regardless of what purpose the GUI serves. These problems stem from understanding what the user requires and the best way to present that information. The real difficulty is trying to predict the audience that will eventually use the tool and how they would be likely to work with it.

It is impossible to create a GUI that will be ideal for everyone, so a compromise is necessary. However, despite having to compromise, several criteria should always be observed when constructing a user interface. For instance, it is known that people have a limited short term, working memory capacity; this should never be exceeded. (The actual capacity is 7±2 chunks, unfortunately, there is no real agreement on what a chunk is, however, a good rule to follow would be to never have the user have to remember more than 7 things at once.) Other criteria that must be observed involve the use of conventions. No conventions should be overruled without good reason; for example, the colour red should not be used to signify “go” since convention says that red signifies “stop” (or “danger”). [Dix1998]

## 3.5 Designing the GUI for this Tool

Any user interface to the tool should, firstly provide access to as much functionality as is available, and secondly provide easy to use access to such functionality. Therefore, in this case, the GUI has to provide a method of selecting a portion of text from the text that has been entered into the compiler. The GUI then has to allow the user to query the type of the text highlighted, and the GUI would have to display the type of the text selected, or some helpful text in case there is no type associated with the text highlighted. It should also provide a method for selecting a new type to impose on a highlighted expression and to re-typecheck the program with those changes in place.

Hence, the GUI requires at least two buttons, one for querying the types of highlighted expressions, and one for applying new types to those expressions. Also required are a text box in which the program code can be displayed to be highlighted, another text box to allow entry of a new type, and an area, perhaps yet another text box, where the type of the currently queried expression is displayed, along with the expression.

Additional functionality useful would be a quit button, and perhaps a separate button to allow the code to be re-typechecked, this would allow several expressions to have their types changed without necessarily having to re-typecheck for each change individually.

The biggest problem is to decide on a layout that best suits the purpose of the tool. There is, unfortunately, no precise way of measuring what object should go where in the window, and so it is simply a case of educated trial and error for the most part. There are a few conventions that should be followed, and a few requirements that should be met, though. Conventions like the fact that the Quit/Cancel button (if one is present) normally resides at the bottom right of any window, and that buttons should have clear concise text (three words at most, fewer words are better – but not at the expense of clarity). In addition, the text boxes should be large enough to show a reasonable amount of text, where a reasonable amount of text is judged by the purpose of the text box. So for instance, the text box for entering new types could be quite small, perhaps only 40 characters wide, and 1 character tall, but it should allow for a limitless amount of text to be entered into it. The other text box will need to be large enough to accommodate more text as it is required to hold the entire program code. In addition, since it is unreasonable to expect arbitrarily large pieces of text to fit in the text box, it should be scrollable. The final condition that must be met is that the text area displaying the type of a given expression needs to be large enough to display arbitrarily large amounts of text, since there is no limit on the size of a data type, however, most types will probably be quite small, so the area need not be too big.

# 4. Implementation

## 4.1 Platform

In order to accomplish this project an existing compiler and graphical tool kit needed to be selected. Though the choice was not large, there was some choice available in both respects, and the final choices were the Moscow ML compiler and the mGtk graphics tool kit.

### 4.1.1 The Compiler: Moscow ML

For a project such as this, it was necessary to locate a compiler that could be used as the basis for the tool to be constructed. There are several criteria that such a compiler should have, the most important of which is; the compiler must be open source, or at least the source code for the compiler must be fully available, and modifiable under an agreeable user licence.

Another criterion with which the compiler had to comply was that it had to be reasonably easy to understand and to modify. This required that the compiler was modular enough for it to be easy to follow the logic of the program through to completion. This initially signals the ML Kit compiler as a possible candidate, and indeed, it was designed to be a kit for building ML compilers (hence the name). However, it was discovered that the compiler was actually overly modular, and so difficult to extend [Keane1999], hence the Kit Compiler was not suitable.

If the project was to be of use to the ML community then it also had to cover most of the ML language – if not all of it. This coupled with the preferable criterion that the compiler should be a currently popular compiler led to the choice of Moscow ML. The Moscow ML compiler fits all of the required criteria, the preferable criteria and has the benefit that it is still under active development so there are those whose knowledge of the compiler is extensive.

Regardless of which compiler is used, the only real part of the compiler that is of interest for this project are the type checking routines. In Moscow ML, there are two slightly different approaches to type checking; one is for the interactive compiler, the other is for the batch compiler. Both use the same functions, however there are small differences in the way the functions are applied. In view of this, it was decided to concentrate on one of these two methods of compilation for this tool, although it would be possible to extend the functionality to cover both, there was insufficient time to do that. The decision was made that the interactive compiler should be the one to use this tool, as it is probably the most often used, and is almost certainly the way new users become accustomed to the use of ML. Hence, it seemed more useful to let this tool be part of the interactive compiler only.

Moscow ML follows the standard compiler model, at least as far as the type-checking phase of compilation. Any program submitted to the compiler is first passed through the lexer, which divides the program into tokens, each of which represents an element of the language. The stream of tokens produced is then passed to the parser, one token at a time. The parser then takes these tokens and constructs an Abstract Syntax Tree (AST). This AST is then passed on to the type checker, which will traverse the tree in its entirety, inferring and checking types as required.

In fact, as with most compilers, the parser calls the lexer, and the lexer will lex one token which will be passed back to the parser, which will deal with it as necessary before calling the lexer again. However, for the purposes of this project, that distinction is insignificant. For more information on general compiling techniques see [Appel1998].

One of the most useful functions provided by the lexer and parser in Moscow ML is the location information. This information is attached to key nodes in the AST during the parsing phase, with information provided from the lexer. The information, describes where, in the input stream, the current token started (i.e. at which character it started) and where the following token starts. Hence, this uniquely defines the position in the input stream where any given expression occurs. This information proved to be invaluable in the creation of the tool required.

The Moscow ML AST format is worth mentioning at this point. As is generally the case for compilers, the AST of Moscow ML bears a close resemblance to the grammar of the language it describes. However, there are several differences. See [Appendix A] and [Appendix B] for the Moscow ML grammar and AST definition respectively.

Two main points about the grammar and the AST of Moscow ML are of note. Firstly, the grammar that Moscow ML describes is not the ML definition of the language, but rather a superset of it. That is, Moscow ML extends the ML language. The extensions, and details of their benefits, are detailed in [mosman] and [mosref]. Secondly, as noted, the AST is slightly different to the grammar. The difference is due to the following factors; one is that several of the Moscow ML grammar rules have been combined (where appropriate) into a single AST data type. Another is that there are elements in the AST that are not required in the grammar, but are in the AST to make things simpler, an example of which is the hash tables that are part of the AST, but not the grammar; these help to keep a track of the current environment in which the tree exists.

Although the AST points noted are not problems as such, understanding that these issues exist aids both in the understanding of the type checking routines, and in where the additional type information is and is not necessary.

Once the AST is fully created (i.e. the entire program has been lexed and parsed), Moscow ML will type check it. This is accomplished by functions available in the “Elab” module of the compiler. Most of the work for this project concentrated on these functions, therefore, it is worth briefly studying them - recalling that basic type checking is quite a simple task, entailing only that the type checker checks that the types required are the same as those given for an expression.

In Moscow ML, type checking is accomplished by passing a type argument as the last argument to most of the lower level "elab" functions. Before the functions are called, this final argument is set to the type that the type checker expects the expression to be. In the case where this type is not unifiable with the actual computed type of the expression, then a type error has occurred, and an error is printed to the consol window. This is where the tool of this project comes into play, the concept being to insert the tool between the type error occurring and the compiler returning to the ready for input state. At this point, the user will be allowed to experiment with the types in the program and will be able to discover the source of the problem.

Although Moscow ML matches all the criteria given, it is not devoid of problems. Amongst the more pervasive was the often poor naming of modules, functions and variables and the lack of comments throughout the code. The poor naming of modules was one of the most difficult problems to overcome, and the use of the Unix "grep" command was invaluable. The lack of comments was the other big difficulty. Throughout almost the entire code, the only comments are in the form of notes made by the original writers to themselves regarding what to revise in future versions of the compiler. Despite these niggling problems, the compiler in general seemed to be fairly well thought out and reasonably well written, with obvious paths of logic through out the code once the programmers style had been discovered and understood.

Despite the noted shortcomings of the compiler, it was deci de that the best way of extending the compiler would be to stick to a similar style of programming. The reason for this decision was that it was thought that program code that is consistent throughout is far easier to understand than code that constantly switches styles. In addition, it made the task simpler, since it allowed for the copying of much of the existing code, with only minor modifications required.

### 4.1.2 The Graphics Tool Kit: mGtk

Since ML does not contain any visual components as standard, in order to create a graphical user interface, a graphics tool kit (GTK) had to be found. Currently for ML there are a limited number of possible options, and of those available, fewer were of possible use.

In the search for a GTK to be used, three options presented themselves. These were SML/tk, mGtk and mosmlgl. As with the compiler choice, it was necessary for the GTK to fulfil several criteria. Most importantly, the GTK should offer as much graphical functionality as possible. This instantly rules out mosmlgl, since it did not offer enough graphical functionality (it covers only a small subset of the open GL library at present). Secondly and equally important, was that the GTK should be able to be compiled by Moscow ML, since the Moscow ML compiler is b uilt and compiled with a previous version of the Moscow ML compiler. This then also ruled out SML/tk, since even though it claims to be able to be compiled with any compiler complying to the SML standard, it failed to compile with Moscow ML. As it is designed primarily for SML/NJ this to be expected. This then left the only viable option as mGtk.

mGtk is an interface between Moscow ML and the GIMP tool kit (Gtk), and has been designed and developed specifically for Moscow ML, though it is still in the very early beta stages of development (version beta 0.3 had just been released at the time of writing). Gtk is the tool kit that was created to ease the creation of the GNU Image Manipulation Package (GIMP) series of programmes, and as such is mostly complete as a GTK. mGtk provides functions that are direct relations to the functions in the Gtk, but are in ML instead of C++. mGtk then coverts these function calls to the C++ function calls, and calls the glib library routines, to perform the actual functionality. Therefore, any functionality available in the Gtk, should also be available in mGtk.

Unfortunately, mGtk lacks any documentation, the only documentation available for any of the functionality being the Gtk documentation, which itself is far from complete. The authors of mGtk recognise the need for documentation of the package, but have not yet produced any. This is understandable given that the software is still in an early beta version. As for the Gtk documentation, it is incomplete and difficult to follow. Much of the Gtk functionality is not in the modules expected, and the help pages often result in simply confusing the user. In addition, although the help pages do list all the functions available, there is no indication of how to use many of them, or even if they are accessible by the user at all. These problems extend directly to mGtk since the Gtk documentation is all that is available for it, therefore to use mGtk effectively was problematic but at the time of writing was the only choice available.

## 4.2 My Implementation

The implementation of this project split into two distinct parts; the core tool, and the graphical user interface. The core is everything in the tool that was not part of the graphical user interface, that is, the functionality of the system to which the user interface gives access.

### 4.2.1 The Core Tool

The core tool itself splits into two distinct parts. These are; the expression type query tool, and expression type changing tool. Each of these parts of the tool could be considered a different tool in its own right and this is how the project was approached.

The type query tool was the first part to be considered, and was also, it turned out, the easiest part to implement though it was not devoid of problems. The goal for this tool was that the user should be able to select any phrase in a program and ask what type it is. The tool should then respond with the appropriate type for that phrase, or some sort of useful message when the phrase does not have a type. The simplest way to accomplish this task was to attach extra information to the abstract syntax tree in the form of a Type Option Reference variable at each node of the tree. This variable is initially set to “Ref NONE” but is then instantiated by the type checker to be the type of the expression as represented by the node. This new information is of the form “Ref SOME X” where X is the type in question. The tool can then look up the AST to find the expression for which the user was looking, and return the value of X to the user, or, in the case where there was no type, a useful message.

In order to avoid having to change the original compiler as much as possible, to avoid introducing errors and further complications, it was decided that the easiest way to add the type information to the AST would be to take a copy of the tree, modify the copy, then use the copy as required throughout the tools lifetime. Since compilation does not continue (after the tool has been used), it is not necessary to provide a way to convert back to the original AST from the modified version. The new modified AST would then be applied to specially modified versions of the compiler functions to accomplish the type checking.

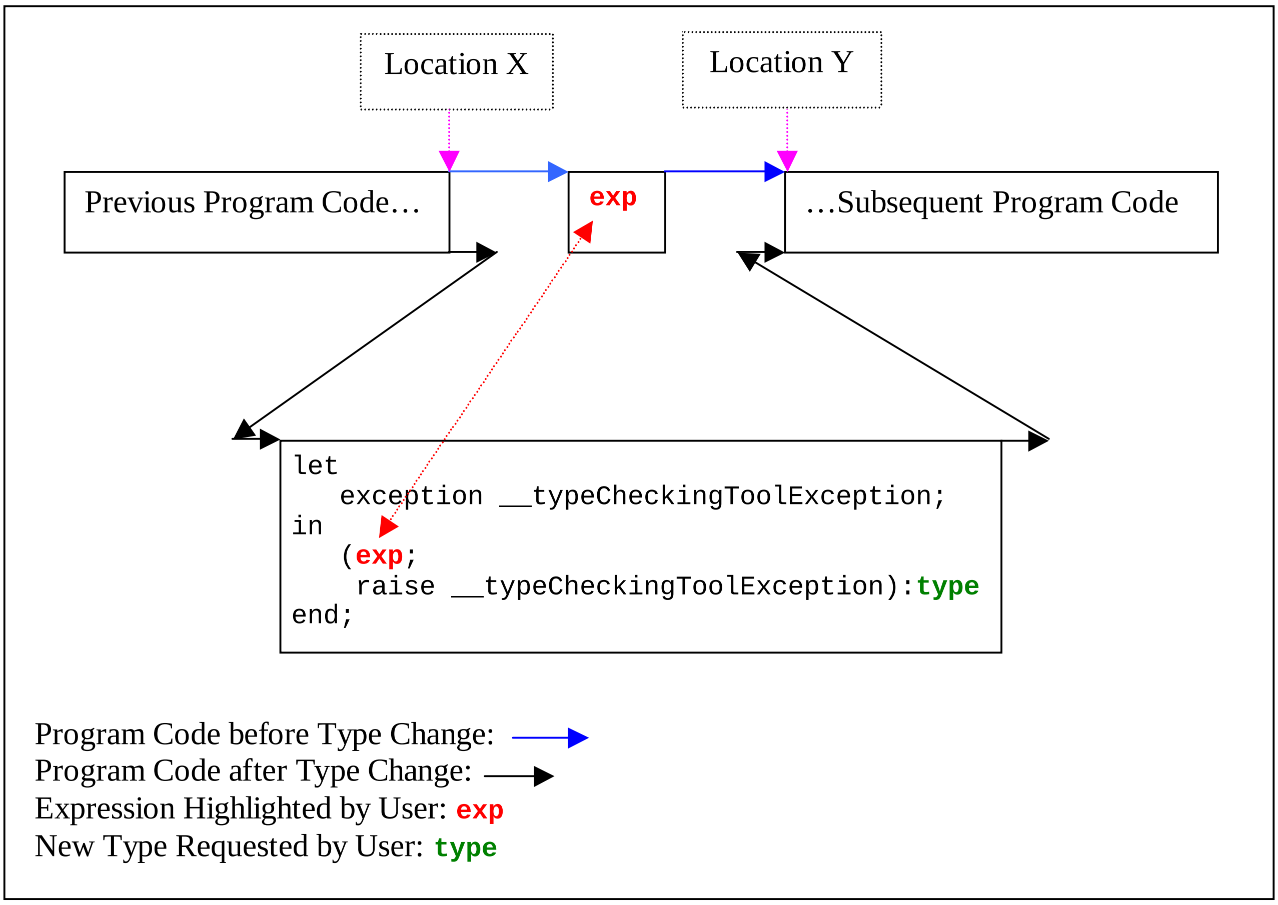
The expression type changing tool was considered next. This posed slightly more difficult and complicated questions. Firstly, there was the question as to how to effect the change in the program. There are several possible answers to this question; for example, it could be possible to modify the type checker to use types the user requested via the tool whenever they were available. Another possibility would be to create a new type checking routine accomplish that same functionality. However, for the sake of simplicity and time to implement, a third option was decided on. As shall be seen however, this third option was far from ideal, and is less useful than either of the other options might have been, had time permitted their use.

It was decided that the tool would be built by relying on features of the ML language. In ML it is possible to give an expression a desired type by attaching the ":" symbol to the expression, followed by the type required. Initially this may seem like the perfect solution; if a user wanted to change the type of an expression then the tool, would modify the AST a form of which would be as if it had ":X" attached to the node of which the user wishes to change the type. Then the AST could be re-type checked and the type checker would use the type X for the expression as requested. However, unfortunately for the purposes of this tool, if the type required is not unifiable with the type of the expression then a type error is generated. This is not what is required; if the user requests that the type of an expression is changed to X then it should be changed to X regardless of whether it unifies with the expression or not. Therefore, another solution was necessary.

In ML, it is also possible to raise exceptions at any point within an expression. Exceptions have the characteristic that they are polymorphic, and will take on any type as required, that is, they can pretend to be of any type as is required. Therefore, it should be clear that if we were to raise an exception which had attached to it the code ":X" then the result of the whole expression would be of type X, which is as we require. The only problem remaining is then how to declare and raise an exception within an arbitrary piece of code. This at first seems simple and the initial solution relies on yet another ML feature, the let…in…end expression.

The "let X in Y end" expression would allow for an exception to be declared in the X section and then would allow for that exception to be raised in the Y section, the exception would not be available outside the let…end expression, and so would be unlikely to corrupt any of the other code in the program. The result is displayed in figure 4.1.

In figure 4.1, the blue arrows represent the code as the user sees it, the previous program code, followed by the expression they highlight, followed by any subsequent


Location X
Location Y
Previous Program Code…
exp
…Subsequent Program Code
Program Code before Type Change:
Program Code after Type Change:
Expression Highlighted by User: exp
New Type Requested by User: type

```
let
   exception __typeCheckingToolException;
in
   (exp;
    raise __typeCheckingToolException):type
end;
```

Figure 4.1

code. The black arrows represent what occurs after the type change is effected, but this new code will be invisible to the user and, as far as she is aware, the code has remained unchanged. The location information, signified by the purple arrows, remains the same, so the new code completely invisibly replaces the old code. This code change effectively removes the expression selected from type checking, allowing the user to ask the type checker if she were, in some way, to make the type of this expression the type she has specified, would this allow the program to type check correctly. If this were the case, then the user could then attempt to change this expression, in the source code, to be of that type in whatever way she wished.

What occurs when the user requests the change of the type of an expression is that the tool will initially search for location X and Y (as in the figure). These two numbers should uniquely define the expression required. Once discovered, the tool will insert a new node in the AST in place of the node representing the expression. This new node will be the node representing the let expression as seen in the figure. In this let expression, the first part of the in…end section is the expression that is being replaced hence nothing is lost. Also, note that the name of the expression has been chosen in order to avoid conflicts with user-defined exceptions, it is in fact highly unlikely that a user would use the same name for an exception as is chosen here. However, even if the user were to use such as exception, no problems should be encountered.

The problems caused by this approach to changing the type were many and several remain in the final implementation. Firstly, the let…in…end expression cannot be placed in any arbitrary piece of code. It may only go in certain places as defined by the Moscow ML grammar [Appendix A]. This is not necessarily a problem however, since the expression can go in most places that would be required – it is notably not allowed on the left-hand side (i.e. before the "=" sign) of function and variable declarations. This need not be a problem since a simple resolution exists, namely applying the change to the arguments supplied to a function when it is being used – or to apply the changes to the code after the "=" sign. Neither of these solutions is ideal, but should nevertheless accomplish the user's task.

Secondly, hiding the let…in…end expression completely posed another problem. As soon as this is done, changing the type of any sub expression of the main expression replaced has no effect on the main expression or anything outside that expression although it remains possible to do. In fact, the sub expression need not even be unifiable with the type of the main expression. These two problems are present in the final version of the tool as created for this project.

## 4.2.2 The GUI

As for the core tool, the general goal for the GUI was to meet all the criteria previously mentioned [Chapter 3] as best as possible. However, before this goal could be reached, it would be necessary to learn the mGtk language. Although this seems a trivial task initially, it quickly became clear that this part would take up most of the time set aside for the construction of the GUI.

The initial goal for this part of the tool was to create a basic interface that would allow the user to highlight an expression in an ill typed ML program, in a GUI window, and have the GUI report the type of that expression back to the user. This required four things; a window, a text frame in the window in which the user could see their program, and in which the text could be highlighted, a button to command the tool to display the type of the currently highlighted phrase, and a label or text box to report the type as discovered by the tool.

The secondary goal for the GUI was to take the tool as created for the previous goal, and extend it to add functionality to allow the user to change the type of the phrase highlighted. This required the addition of two new features to the GUI; a new text box to allow the entry of the new type required, and a button to command the tool to perform the type change. Tertiary goals were to add a quit button, and to make the user interface as user friendly as possible.

As previously noted, before any of the goals could be realised the implementation language had to be learned from scratch. This initially seemed like a reasonable and attainable goal in the time limits set for the project, but latterly this proved to be a somewhat unrealistic view.

The eventual GUI is actually simply a hacked version of one of the examples as provided by mGtk. Although it would be possible to create a new and better GUI for the tool, it would require much time and effort, most of which would be used in attempting to learn the mGtk ML extensions. Unfortunately, this project ran out of time before such an attempt could be made. Hence, the user interface created for the project is far from ideal. It barely covers the main two aims of the GUI; providing a two-button interface, with two text boxes and a label, each of which does as described earlier [Section 3.5].

An interesting note about the GUI is that the Moscow ML interactive compiler is currently only a command line (i.e. text) based compiler, whereas the GUI is obviously a graphical tool. This means that the user has to adjust between using the Moscow ML compiler via a textual interface, to using the tool via a graphical interface. This initially may seem to be a problem, nevertheless, it appears to work well, and make sense in the context of the tools being used. It is likely that the user will write their code in a text editor of some description anyway, copying and pasting into the interactive compiler as required, therefore changing between the interface types should not pose a problem.

The idea of the GUI is that it is launched as a new window when an attempt is made to evaluate an incorrectly typed Moscow ML program in the interactive compiler. The new window appears on top of the user's current screen with the program, as applied to the compiler, visible in the largest of the text boxes. When the tool is no longer required, the user simply closes it (using the window manager close buttons, since no quit button exists) and the interactive compiler could then be used as normal again. In the case when the program applied to the compiler is correctly typed, the compiler performs as normal, and the fact that the tool exists is not apparent to the user.

# 5. Example

## 5.1 Example Used in Detail

To display the tool working, and illustrate the need for such a tool, below is an example (figure 5.1a and 5.1b), in which an error is located using both some compilers and the tool created.

```
let
    fun addend (x, [])         = x
    |   addend (x, (hd::tl))  = (hd::(addend(x, tl)))
in
    fn (x, elm, lst) => case x of
                          1 => addend(elm, lst)
                        | 2 => (elm::lst)
                        | 3 => (elm::(addend(elm, lst))
end;
```

Figure 5.1a – Example Code used (Error highlighted in red)

This example (due to Alison Keane [Keane1999]) shows a function designed to provide the following functionality, when passed a number, an element (of type α) and a list (of type α list), in the case that the number is "1" the element is added to the end of the list, in the case where the number is "2" the element is added to the beginning of the list, and in the case when the number is "3", the element is added to both the beginning and the end of the list. Note that for any other number a "Match" exception will be raised, indicating that an argument could not be matched with the clauses of the function.

Although this example seems correct at first glance, it contains an error commonly made even by experienced ML programmers. The error is on the first line of the addend function and is that the text after the "=" sign should read "[x]", "x::[]" or "x::nil" instead of simply x (the error is highlighted in the figure). That is, currently the function's type will be:

$\alpha$ list $\rightarrow$ $\alpha$ list $\rightarrow$ $\alpha$ list

rather than the intended:

$\alpha \rightarrow \alpha$ list $\rightarrow$ $\alpha$ list.

This error, while easy to make, can be difficult to spot and correct. A correct version of the code can be seen in figure 5.1b.

```
let
    fun addend (x, [])         = [x]
    |   addend (x, (hd::tl))  = (hd::(addend(x, tl)))
in
    fn (x, elm, lst) => case x of
                          1 => addend(elm, lst)
                        | 2 => (elm::lst)
                        | 3 => (elm::(addend(elm, lst))
end;
```

Figure 5.1b – Example Code with error corrected (Corrected error highlighted in red)

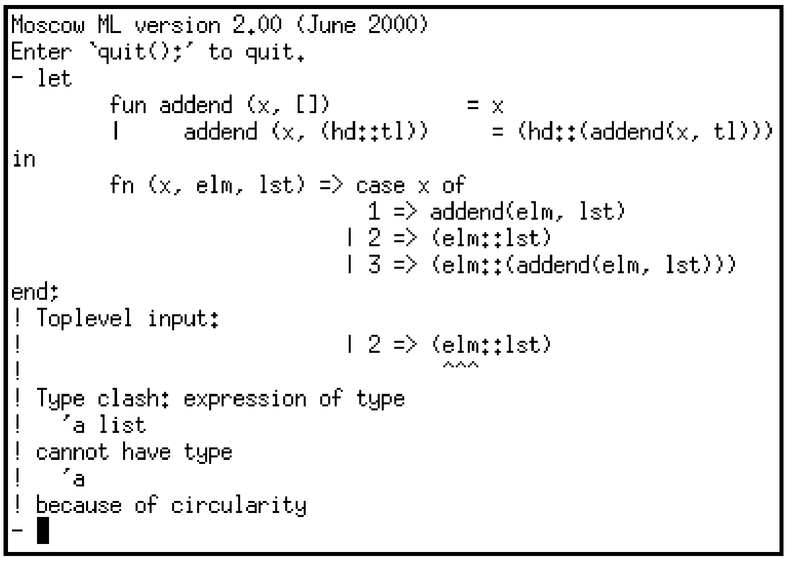
## 5.2 Some ML Compilers' Responses to the Erroneous Code

The responses given by compilers to the erroneous code vary in both clarity of understanding and usefulness. The Moscow ML v2.0 compiler responds as can be seen in figure 5.2a and the SML/NJ v110.0.7 compiler responds as can be seen in figure 5.2b (over page).

The response, as can be seen, could be difficult to understand to those not overly familiar with the workings of type checking. Of the two responses presented, the SML/NJ one is potentially the most confusing of the two, giving both a poor indication of where the error lies (if the statement "elm :: lst" occurred several times in the code, then tracking down the problem becomes difficult) and also of the nature of the actual error. The error given seems to suggest that there is an element of type 'Z list list. Since the programmer did not knowingly enter a term of this type, this can be the source of much confusion.

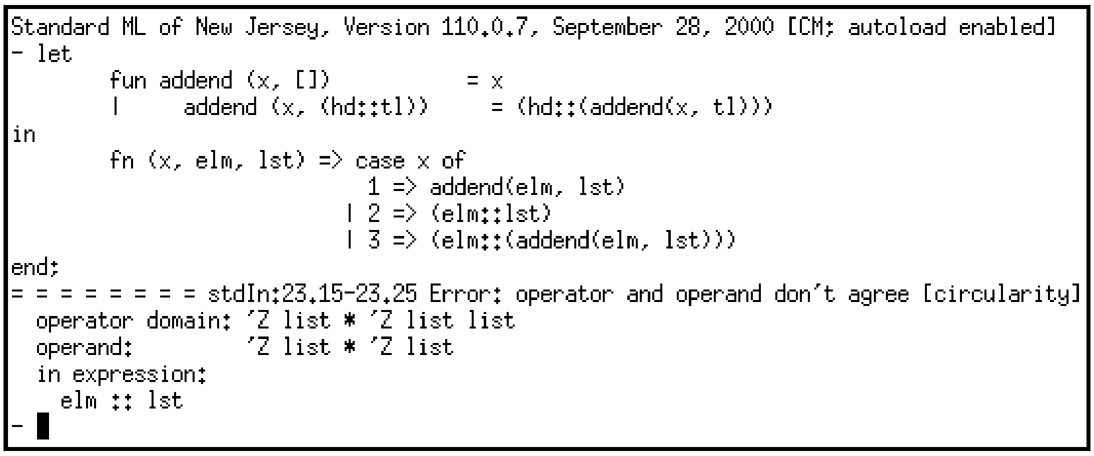
The response of Moscow ML to the error is much clearer on both counts, providing an accurate position of where the error was encountered, and a better description of the problem. Nevertheless, the description is still confusing since it may not be clear why "elm" was attempting to be unified with α list.

Both compilers, however, report that the error is due to circularity, in other words the attempt to unify α with α list, which would result in an infinite type of the form α list list list…etc. Although this is true from the type checker's perspective, circularity does not cause the actual problem, indeed many novice ML programmers may not fully understand that to which circularity refers, leading to possible further confusion in the error reported. Many novice ML programmers when presented with this situation would read the code line by line attempting to analyse where the problem lay, to a large extent ignoring the compiler's response as it only serves to confuse them. It should be clear, therefore, that an alternative approach to the standard error messages provided by most ML compilers is required.



```
Moscow ML version 2.00 (June 2000)
Enter `quit();' to quit.
- let
        fun addend (x, [])           = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                             1 => addend(elm, lst)
                           | 2 => (elm::lst)
                           | 3 => (elm::(addend(elm, lst)))
end;
! Toplevel input:
!                          | 2 => (elm::lst)
!                                  ^^^
! Type clash: expression of type
!   'a list
! cannot have type
!   'a
! because of circularity
- █
```

Figure 5.2a – Moscow ML



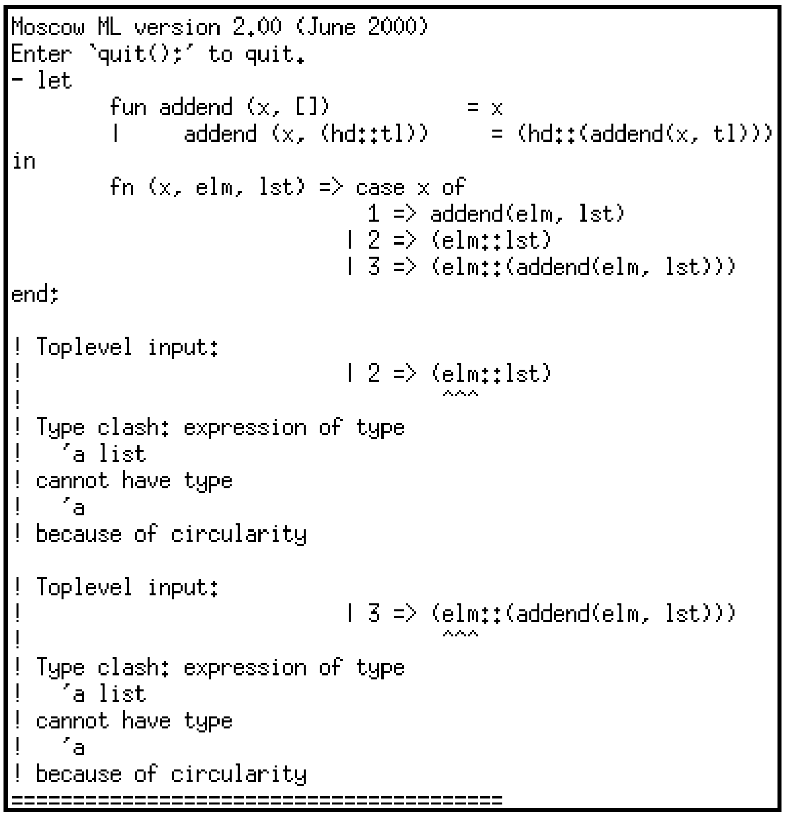
```
Standard ML of New Jersey, Version 110.0.7, September 28, 2000 [CM; autoload enabled]
- let
        fun addend (x, [])            = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                              1 => addend(elm, lst)
                            | 2 => (elm::lst)
                            | 3 => (elm::(addend(elm, lst)))
end;
= = = = = = = = stdIn:23.15-23.25 Error: operator and operand don't agree [circularity]
  operator domain: 'Z list * 'Z list list
  operand:         'Z list * 'Z list
  in expression:
    elm :: lst
- 
```

Figure 5.2b – SML/NJ

## 5.3 Walk Through of Code when Used With This Projects Program

The aim of this project was to provide a more efficient way of locating type errors in ML programs. This example shows an attempt to provide some proof as to the success or otherwise of the project.

To begin this example, the Moscow ML compiler is run as would normally be expected. The erroneous code is then cut and pasted into the compiler, and the return key is pressed, at which point the compiler responds as in figure 5.3.



```
Moscow ML version 2.00 (June 2000)
Enter `quit();' to quit.
- let
        fun addend (x, [])            = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                              1 => addend(elm, lst)
                            | 2 => (elm::lst)
                            | 3 => (elm::(addend(elm, lst)))
end;

! Toplevel input:
!                           | 2 => (elm::lst)
!                                   ^^^
! Type clash: expression of type
!   'a list
! cannot have type
!   'a
! because of circularity

! Toplevel input:
!                           | 3 => (elm::(addend(elm, lst)))
!                                   ^^^
! Type clash: expression of type
!   'a list
! cannot have type
!   'a
! because of circularity
========================================
```

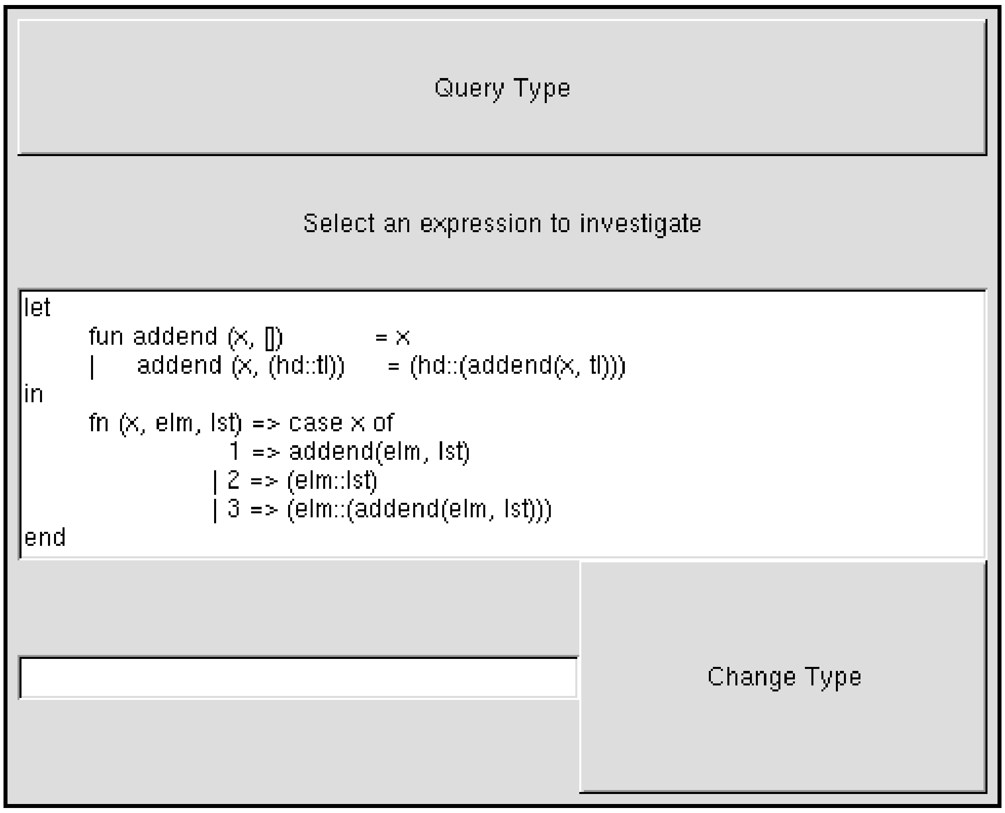
Figure 5.3 – The tools response to the erroneous code

We can make several observations regarding the response of the compiler to this error. Firstly, note that the compiler returns two type errors, separated by a blank line. The first of these two errors is displayed as would be expected by the original compiler. The second error indicates that there is a further type error in the code, and that these are the only type errors present in the code.

As the type checker is not stopped when a type error is encountered (normally an exception is raised and the type checking process halts) any further type errors in the code will be found and displayed, hence the second error is displayed here. While this can aid the location of any error in the code, the user could quickly become overwhelmed by the number of errors if there are many type errors in the code. In addition, any type errors other than the first may be misleading to the user since they may be a side effect of the first error. It was decided to report them, however, allowing the user to respond accordingly, ignoring them if required.

Another point to note is the line of "=" signs at the bottom of the output which is used to distinguish between error printouts. As a new batch of type errors may be reported after every change made to the code, the line divides errors that relate to the latest action from errors relating to previous actions.

At this point, the GUI is displayed (this happens almost simultaneously with the errors being displayed, and occurs only when a type error is detected). The initial window presented to the user can be seen in figure 5.4. Note that due to the way the screen shots were captured, the X-window interface is not visible, although it does exist for the GUI.

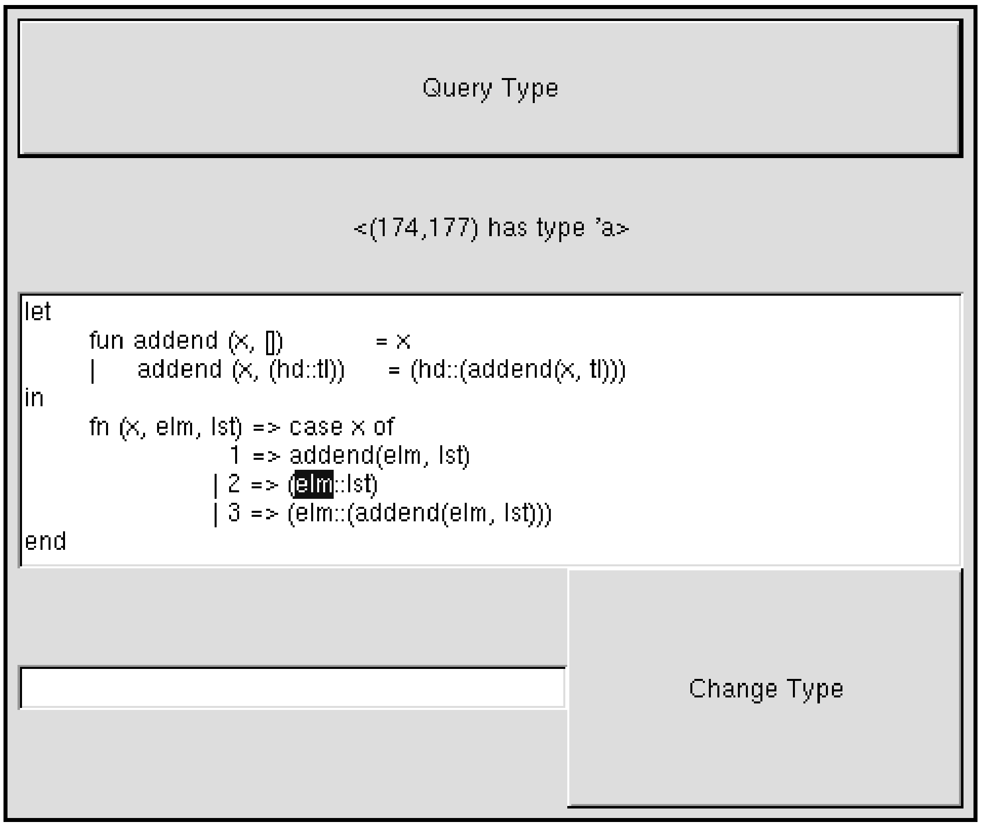


```
let
        fun addend (x, [])          = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                  1 => addend(elm, lst)
                | 2 => (elm::lst)
                | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.4 – The GUI in its initial state

In the figure, the code as entered into the compiler is displayed in the main text window. The button for querying the type of a highlighted expression is at the top of the window and the label directly below it, currently displaying “Select an expression to investigate” will display the type of any expression highlighted after the button is pressed. The area for entering new types to apply to the code is at the very bottom of the window, adjacent to the button that when pressed wil l convert the type of the given expression to the type requested.

Given the initial error that was reported (figure 5.3), the most obvious starting point would seem to be to check the types of those elements noted in the type error. Having decided this, we then can highlight the desired expression in the code and query its type. In this example, we start with the “elm” expression as indicated by the compiler as the source of the error. The result of highlighting the “elm” expression, then pressing the Query Type button is shown in figure 5.5.



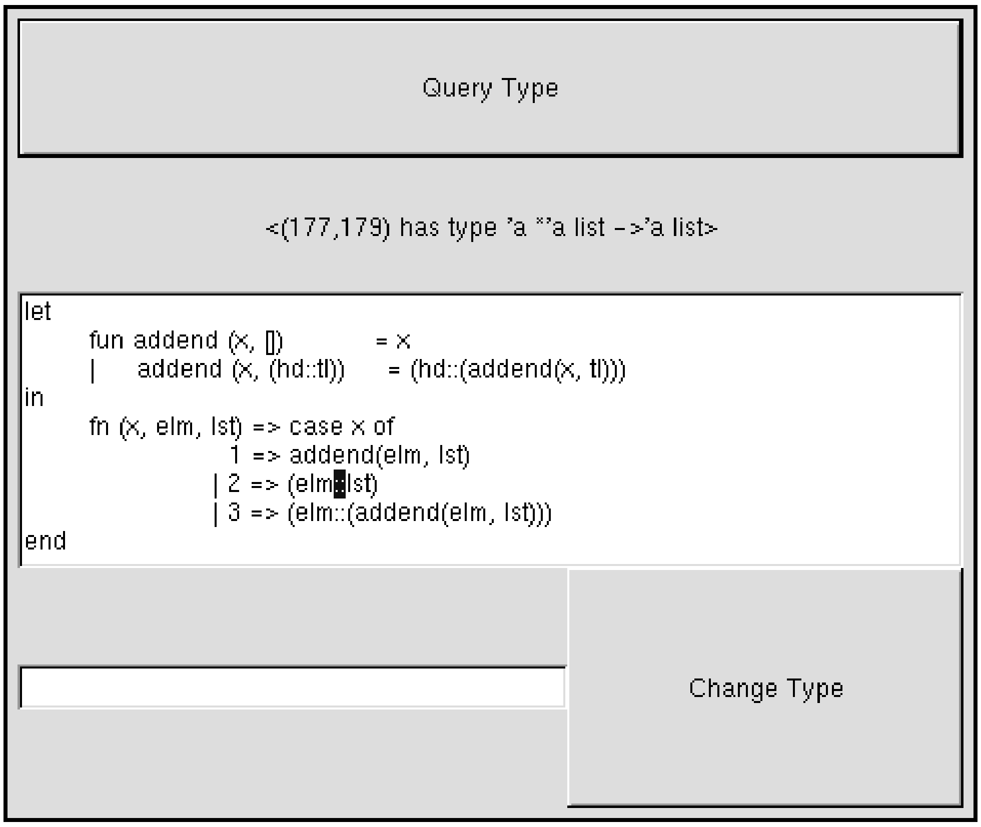
<(174,177) has type 'a>

```
let
        fun addend (x, [])            = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                        1 => addend(elm, lst)
                      | 2 => (elm::lst)
                      | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.5 – Result of querying the type of the highlighted expression.

Note that now the label text has changed to reflect both the position of the highlighted text and the type of the expression highlighted. The fact that it is the position of the text that is displayed rather than the text itself is one of the failings of the GUI. This issue could be addressed by the addition of another text box, similar to the one that holds the program code, but for the expression highlighted and the type of that expression.

The user at this point should note that the type of “elm” is as expected at this point, and so the question then moves to the “::” operator. The user then highlights and queries this expression, with the result as shown in figure 5.6.

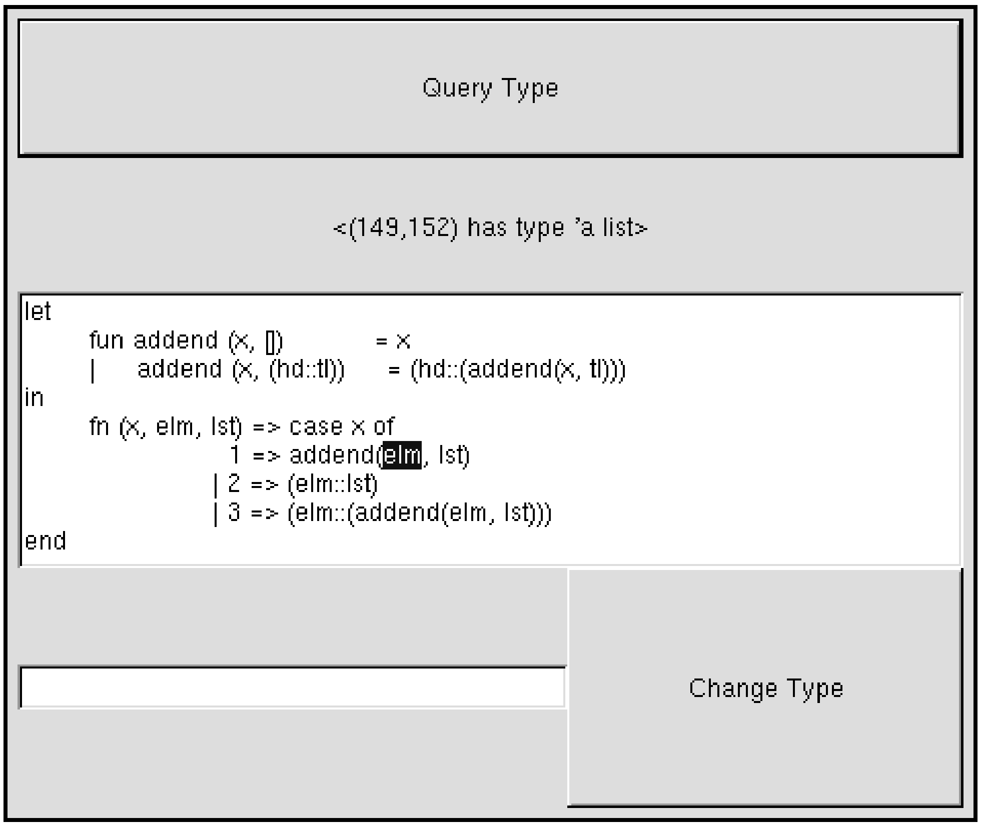


<(177,179) has type 'a *'a list ->'a list>

```
let
      fun addend (x, [])          = x
      |    addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
      fn (x, elm, lst) => case x of
                   1 => addend(elm, lst)
                 | 2 => (elm::lst)
                 | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.6 – Result of querying the type of the highlighted expression.

We now note that the result of this type query is as expected. The user now addresses the problem of attempting to discover why the compiler reported an error at this point. It should be realised that if the type of “elm” here is correct then perhaps it is incorrect elsewhere and if this were the case then the error reported by the compiler may make sense, since it is impossible to give the “elm” expression two different types simultaneously. With this thought in mind, the next expression to test would be the “elm” immediately preceding the one, which the error notes as the problem (see figure 5.7, over page).



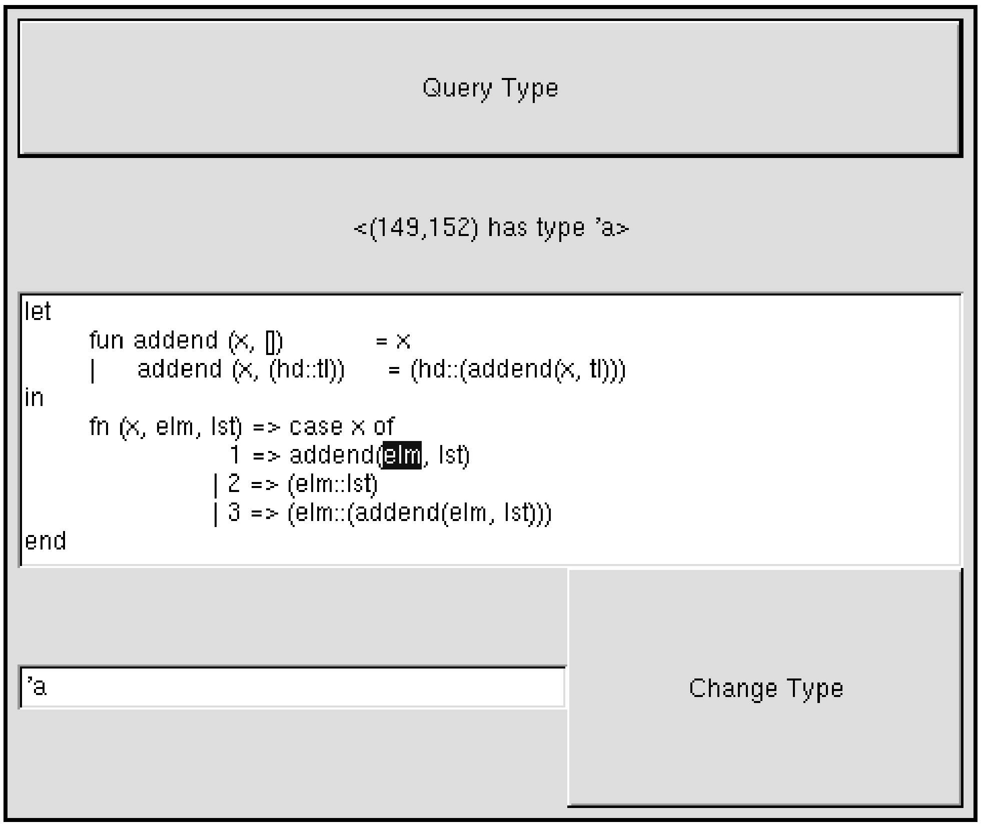
<(149,152) has type 'a list>

```
let
        fun addend (x, [])           = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                         1 => addend(elm, lst)
                       | 2 => (elm::lst)
                       | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.7 – Result of querying the type of the highlighted expression.

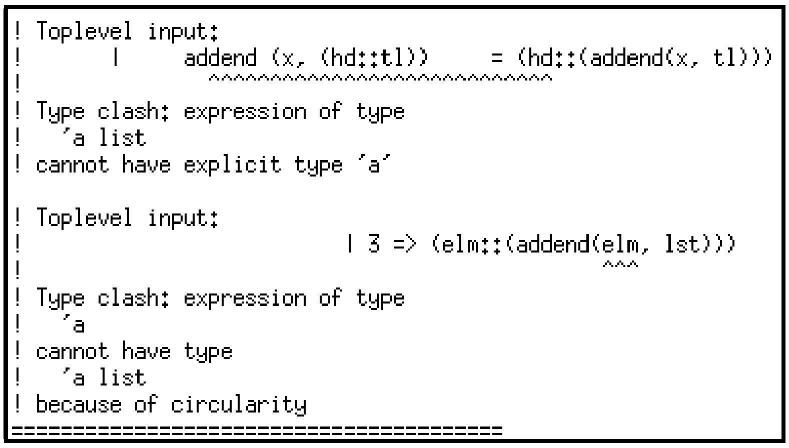
It can now be seen that the type of “elm” here is not the type that is required, however it is one of the types mentioned in the error reported by the compiler. We have already encountered the other type reported by the type error during the previous test on the “elm” which comes subsequently in the code. Now the error begins to make more sense. We have discovered both that the type checker has encountered problems in the attempt to give the same variable two incompatible types, and where this occurred. The problem is to understand why it occurred, and where the cause of the error lies.

In an attempt to locate the cause of the error, the user would now give the “elm” expression highlighted the type that she expected it to be in the first place, i.e. α, see figures 5.8a and 5.8b (over page).



```
let
      fun addend (x, [])          = x
      |    addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
      fn (x, elm, lst) => case x of
                     1 => addend(elm, lst)
                   | 2 => (elm::lst)
                   | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.8a – GUI result of changing the type of the highlighted expression.



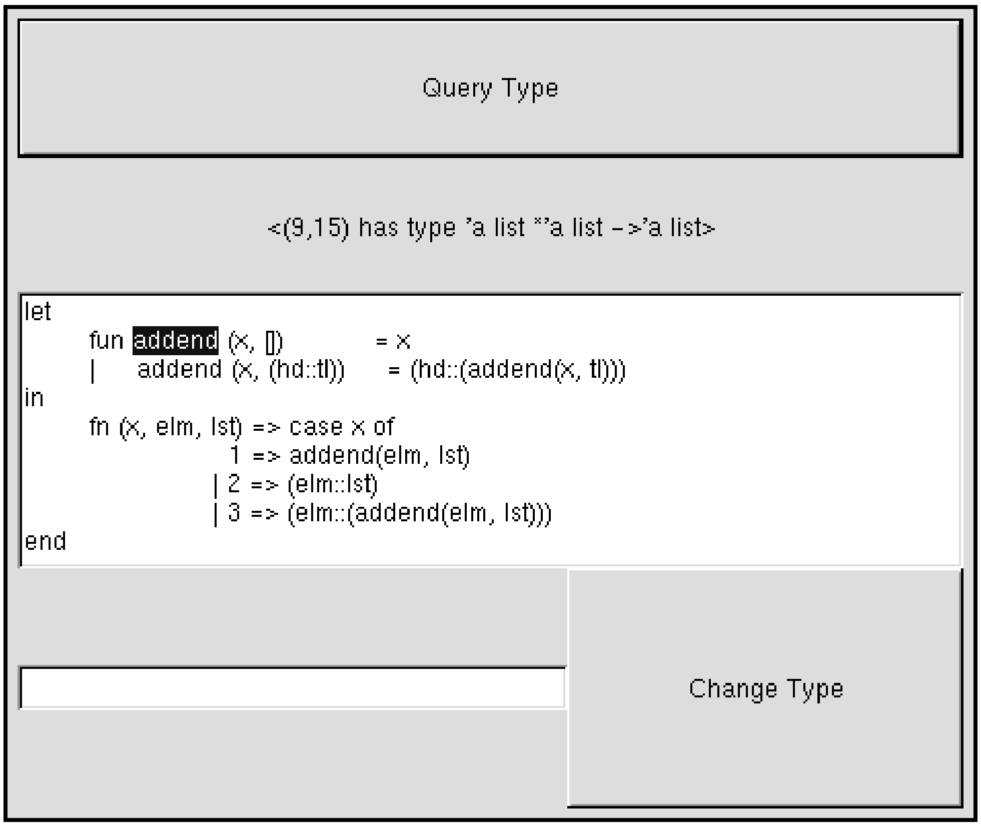
```
! Toplevel input:
!       |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
!               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
! Type clash: expression of type
!   'a list
! cannot have explicit type 'a'

! Toplevel input:
!                           | 3 => (elm::(addend(elm, lst)))
!                                                    ^^^
! Type clash: expression of type
!   'a
! cannot have type
!   'a list
! because of circularity
=======================================
```

Figure 5.8b – Compiler response to the change of the type.

Figure 5.8a shows the GUI after firstly typing in “`a” (α) into the bottom text box, then pressing the “Change Type” button. The resultant errors printed out by the type checker are displayed in figure 5.8b. These errors are displayed directly below the previous type errors in the shell window being used by the compiler, and as previously stated, are separated by the lines of “=” signs. Note that the circumflexes underlining the error are not sufficiently precise, in that they do not point to the sub-expression in which the error exists; this is a result of the method by which the compiler locates the error in question. It is assumed that the user will ignore these.

The user would now note that the type error has been changed by this change of type. The new type error now points to the error being located in the function in the let part of the expression. This then should change the focus of the investigation to the addend function, see figure 5.9.

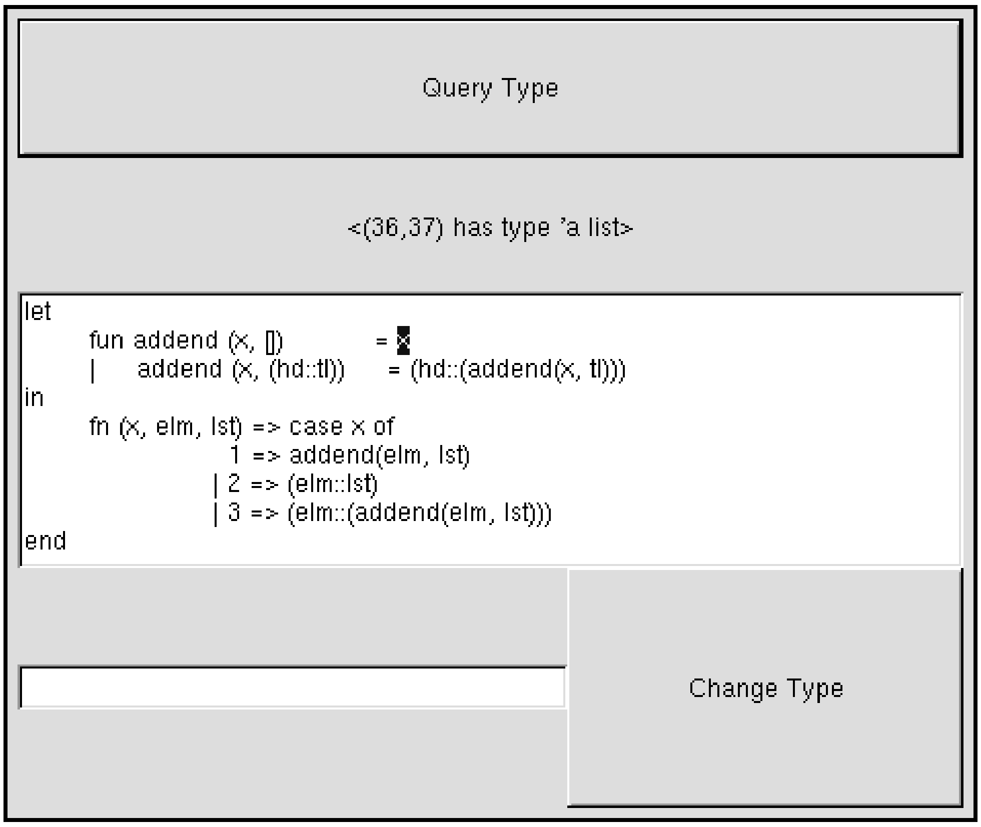


<(9,15) has type 'a list *'a list ->'a list>

```
let
	fun addend (x, [])	= x
	|	addend (x, (hd::tl))	= (hd::(addend(x, tl)))
in
	fn (x, elm, lst) => case x of
		  1 => addend(elm, lst)
		| 2 => (elm::lst)
		| 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.9 - Result of querying the type of the highlighted expression.

The user can now see (figure 5.9) that the type of the function here is not what was expected as it should be α * α list → α list. At this point she is expected to look at the code and recognise that the type of x is incorrect, which should be obvious from the function type but can be checked simply enough by highlighting the x argument and pressing the Query Type button. The user can then be expected to note that the expression after the "=" is not correct, which can be verified by highlighting the "x" in question and querying its type, figure 5.10 (over page).



```
<(36,37) has type 'a list>
```

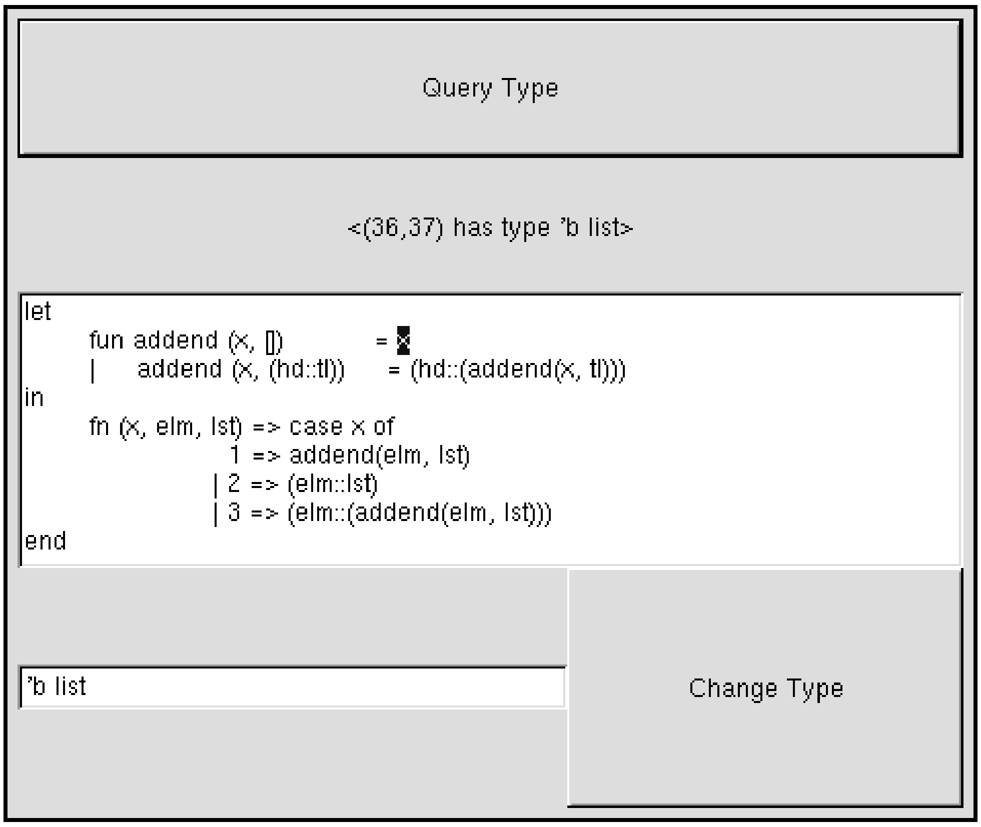
```
let
        fun addend (x, [])          = x
        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
        fn (x, elm, lst) => case x of
                          1 => addend(elm, lst)
                        | 2 => (elm::lst)
                        | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.10 - Result of querying the type of the highlighted expression.

At this point, the user has a choice; she could make the change to the source code required, i.e. turn the code from figure 5.1a into that of 5.1b and attempt to compile it again, in the expectation that the types for the rest of the program correctly match. Alternatively, she could assign a new type to the incorrectly typed variable, re-type check the code and note any further type errors in the code.

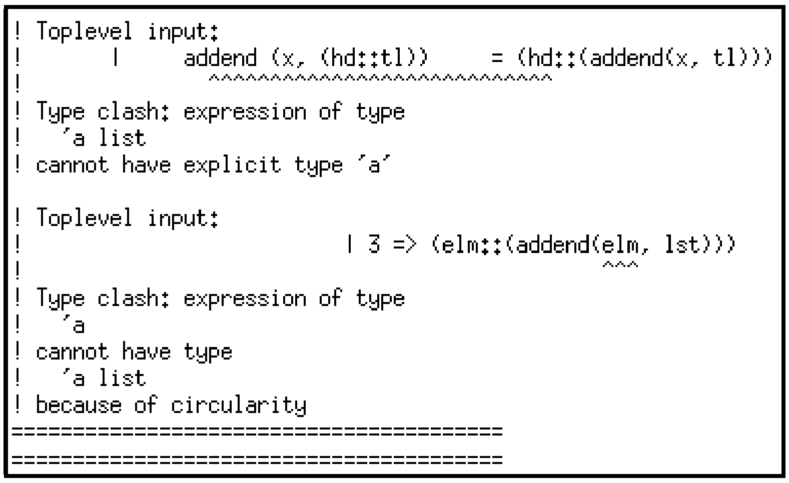
Taking the second option, the user may well give the expression in question the type β list. As mentioned earlier, this change effectively removes the expression "x" from the type checking – allowing the user to apply any type they wish. When this is done, the responses from the program are as in figures 5.11a and 5.11b (over page). Note that in figure 5.11b, the type errors present are not from this change, but are from the previous type change, but have been left in the figure to display the fact that nothing else is output. The code now completely type checks as is signified by the lack of errors between the two lines of "=" signs, the second of which is the only line to have been displayed upon pressing the Change Type button on this occasion. Hence, the user has successfully located the error in question, and may now correct it in the source code, reapplying the compiler to the newly revised code.

There is no need for the user to have followed precisely these steps for the solving of this problem. She for example, may have tested the types of other expressions in the code before coming to a conclusion about where the error lay. However, it is hoped that the tool should help to guide the user to the true source of the error. Note also that in following this example the second type error was ignored completely and the initial error was followed until the true cause was located. Alternatively, the user may investigate any of the type errors, and knowing the cause of one of the errors, may investigate others. The second type error, n this case, had th e same cause as the first, and so may have provided extra pointers for the user as to what could be investigated.



```
let
      fun addend (x, [])          = x
      |    addend (x, (hd::tl))     = (hd::(addend(x, tl)))
in
      fn (x, elm, lst) => case x of
                   1 => addend(elm, lst)
                 | 2 => (elm::lst)
                 | 3 => (elm::(addend(elm, lst)))
end
```

Figure 5.11a - GUI result of changing the type of the highlighted expression.



```
! Toplevel input:
!        |     addend (x, (hd::tl))     = (hd::(addend(x, tl)))
!                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
!
! Type clash: expression of type
!   'a list
! cannot have explicit type 'a'

! Toplevel input:
!                        | 3 => (elm::(addend(elm, lst)))
!                                                ^^^
! Type clash: expression of type
!   'a
! cannot have type
!   'a list
! because of circularity
=========================================
=========================================
```

Figure 5.11b – Compiler response to the change of the type.

## 5.4 Conclusions from the Example

The example used here is rather simple in both its error and its functionality; nevertheless, it demonstrates adequately the usefulness that such a tool can have. As with all tools, it relies on the user employing it correctly, and having some degree of knowledge about possible problems that could have caused the error.

As should be clear from the example, the tool makes the location of such an error much simpler than previously possible. While the programmer could have been fortunate enough to spot the error straight away and corrected it without any need for such a tool, in a complex example, this may not have been possible, and such a tool could be an invaluable asset to the programmer.

Even if it were the case that the tool did not have the ability to alter the types of expressions in the code, it would still be of use as often the key to locating a type error is verifying what type expressions actually have. During the creation of the tool itself such a tool would have been helpful, since there were occasionally type errors that seemed to make little sense, the resolution of which required a tedious trial and error approach. This approach required changes in the source code, then recompilation after each change, to see if the last change affected the error in any way. Hence, the error location process was long and complicated, and more ofte n than not, the error was caused by a simple mistake, which could easily have been noticed had the true location of the problem been known.

# 6. Conclusion

## 6.1 Is The Project Finished?

In order to assess whether or not the project is finished, it is ne cessary to ask several questions.

### 6.1.1 Does It Meet the Initial Requirements

In the course of the project, there were two aims:

- to integrate a tool for investigating type errors into an existing ML compiler;
- to provide an effective user interface to the tool

The first of these aims translated into two goals:

- the creation of a tool to report the type information of a given expression in an SML program.
- the creation of a tool that would allow the type information of a given expression to be changed to a type defined by the user.

The second of the aims became the third goal, namely:

- to provide an effective user interface to the tool.

The first goal was reasonably easily achieved. Simply adding some extra information to the AST, which was then instantiated during the type-checking phase, was enough to allow the type of any expression to be discovered.

The second goal was much more difficult to achieve, and remains partially unsatisfied. The initial requirements of this part of the tool were that the user was able to change the type of a given expression to any type that she chose. This has been largely achieved although currently it is only possible to change the type of expression where the code would normally allow a let…in…end expression to be placed, hence function arguments in the functions definition are not able to have their type changed.

The third goal was also difficult to reach and remains the least complete of the three. As was noted earlier [Section 3.3] most GUIs are created hurriedly and wit h little thought as to the end user. Unfortunately, the GUI for this tool also fell foul of this trap, due to the difficulty experienced in the use of mGtk.

When mGtk was chosen as the tool for the creation of the GUI, it was mainly because there seemed no other viable alternative at the time. This was unfortunate as it transpired mGtk was a difficult tool kit to learn and use, primarily due to its lack of documentation. Hence, the GUI for this project is really only a placeholder for a proper GUI. It was however, all that was possible given the time constraints imposed and therefore, the interface falls short of the requirements laid out in chapter 3.

## 6.1.2 Its Successes and Failures

The project as it is currently has many successes and failures when reviewed next to the initial requirements as has just been noted. Some of the success, however, is not in the tool itself but in the realising that firstly, the approach taken to the implementation of the tool will never be able to work fully, and secondly there is a much better way to approach the problem [Section 6.2]. Although these are successes in some ways, they are failures in others; they are failures in that the current tool does not work as required.

Some of the major failings of this project cause it to be of very limited use in its current form, and without a great deal of work, it would serve little purpose to present the tool to the current ML user base. It is important to note that these failures are not in the project itself as a program, but are failures of the approaches used to solving the problem as presented. More specifically, and with reference to the original requirements, the failings of this project are:

- the lack of ability to change the type of any arbitrary expression in an ill-typed ML programs; currently only expressions is particular places are allowed to have their type changed. (That is, only expressions where it is allowable to exchange the expression with a let…in…end expression). This is a significant problem, as one of the major requirements of the project was that it should be possible to specify the type of an arbitrary expression in an ill-typed ML program.

- the lack of a good user interface – this is very unfortunate as one of the major goals of the project was to improve on the user interface provided by Alison Keane's project. However, for reasons previously stated [Section 4.2.2] this was not possible.

The successes of the project are:

- the discovery that this particular approach to the problem of changing the type of an expression is flawed.

- the conception of a new approach that should allow such a tool to be built with a minimum of problems.

- and the creation of a tool that will tell the user the type of any expression with in an ML program – note this part works (and is useful in itself) even though the rest of the tool is not completely satisfactory.

## 6.2 Future Work

The tool as designed had, as noted, several shortcomings. There are several places in which improvements could be made, many of these where not carried out due to time constraints. Others have only become apparent after the project was completed.

### 6.2.1 Additional Functionality

There is a great deal that could be added to the project as it is, some of which would improve the tool's usefulness and the rest of which involves streamlining the process the user must employ to discover the type errors.

Any additions to the functionality must first focus on the missing key functionality, which is included in the requirements. This includes functionality such as; the ability to alter the type of any arbitrary (complete) phrase within an ill-typed ML program, and the ability of certain changes to affect more than a single location in a program. For instance, if a request is made to change the type of a variable, then all occurrences of that variable after the point concerned (but in the variables current scope) should also have their type changed. Although the last point is not explicitly stated in the requirements it is may be intuitive to the way the user may expect such a tool to operate.

An element that would be useful to have, but is not present in this tool, is focusing, or zooming; this is functionality present in Alison Keane's tool and proves to be very useful. Zooming would allow the programmer to take any expression from the program and investigate type errors in that expression locally, that is without any of the changes affecting the global environment. Once an erroneous expression is located, the user could then zoom into that expression, and test sub-expressions of that expression, without having anything in the main code affected by any of the changes made in these sub-expressions. In addition, the zooming would be able to take any type the user had requested for the expression being zoomed in on , and would use this type as the type with which the attempt is made to unify the sub -expression. This latter functionality would be very difficult if not impossible to accomplish without revising the current approach to the way the types are altered in the AST.

More additional functionality that would be preferable, although is not essential to the workings of the tool, would be to have some sort of undo facility either for the whole tool, locally or both. That is, an undo for the whole tool would undo the last action sequentially in the reverse order to that in which they occurred. A local undo would allow highlighted expressions to have their types reverted to a previous type – if this type was the original type of the expression, then any additional cod e added by the tool to this expression would be removed – thereby reverting the expression to its original state.

Other functionality that may be of use would be for there to be a procedure for the programmer to receive an ordered list of the changes she made to the code – to avoid her having to remember all the details themselves. A redo facility might also be useful, although perhaps less so than any of the other mentioned functionality. It may also be useful if the user were provided with an automatic way of building up types to be applied to an expression. For instance, she could be provided with radio boxes, listing all of the possible in-built types and user defined types in the code currently, and using one of them would simply be a case of selecting the appropriate check box and applying the selection. An alternative could be a drag and drop style interface where the types are provided as icons, that can be dragged on to a text area where the can be built up, using the standard type operators, into whatever type the user desires.

### 6.2.2 Improve User Interface

As noted earlier, the user interface of this tool could be very much improved. Although it provides access to all the functionality available, it does not provide it in a pleasing manner. It is also lacking in some minor functionality, for instance it does not have a quit button. It also currently does not provide the user with adequate feedback since the label that tells the user the type of the given expression, only refers to the expression by its position in the code. In addition, if the type of an expression is quite large, then the label will be overrun with text, and the user will not be able to make out what the actual type is.

Other functionality missing from the user interface is that it does not provide a safe environment for the user. That is, none of the data provided by the user is checked before it is used. What is more, if the user enters a type that is not recognised then the user interface crashes. It is arguable that the user s hould take care not to enter text that is not correct, but realistically the interface should not crash in such a situation – especially since it is likely to be quite a common occurrence.

### 6.2.3 Efficiency Considerations

Due to the manner in which this project was written, much of the code in the original compiler is duplicated in a slightly modified form for the tool. As such, there is much that is redundant in the compiler and which could be removed through a few small changes to part of the code. Many of the functions could be used for both the normal compiler and the tool, with only minor modifications.

For instance, currently, there exist two copies of the type checker in the compiler. One is the version that is normally in the compiler, and the other is modified slightly to add extra type information to the AST. The reason these were made separate was to avoid introducing bugs into the normal compiler execution and to allow more freedom in the changes that were performed to the type checking code. Had more time been available for the project, the type checkers would probably have been combined into a single piece of code. Another example of duplicated functionality is in the printing functions. These too had to be copied and modified in a similar way to the type checker; again, had more time been available these would have been combined with the original versions.

The final efficiency consideration is the positioning of the additional type information in the AST. Initially the idea was to put the inf ormation in every possible location that might have a type associated with it, however, in retrospect this was a bit too much. It would simply be enough to provide extra type information in the same places that the location information is available. This is due to the fact that the location information has been put in the AST in all the places necessary to uniquely define an expression in a ML program, hence if any expression can be located this way it is enough to put the type information here too. It may even be possible, and beneficial to simply extend the location data type to contain the type information itself – this could make implementation much easier, and would possibly avoid many of the changes to the compiler that were made in the course of this project.

### 6.2.4 Write a New Type Checker from Scratch

Understandably, the Moscow ML compiler was not designed with a tool such as the one created here in mind; it is unlikely that any compiler would be. This causes great difficulty in implementing such a tool, since much of it requires functionality of the compiler that is not immediately available, and is often hard to access.

It would be much more appropriate to build a new type checker designed specifically for this type of tool although perhaps time consuming. Such a type checker could be integrated into Moscow ML simply by replacing all calls to the existing type checker with calls to the new type checker, as long as it was ensured that the new type checker returns the same values as the old type che cker. Note that side effects need to be carefully considered too, since inducing the correct side effects happen is key to making the whole compiler function identically after the new type checker is installed. However, so long as the new type checker causes the same side effects to occur, and returns the required values correctly, then it is free to do any thing at all. Probably the best way to go about creating such a new type checker would be as follows.

The first thing to do would be to take a copy ofthe AST; this new copy would be a slight modification of the original AST and would be very similar to, if not the same as, the AST used by this project. The key difference between the original AST and the new one would be that the new one would include Ty pe Option Reference at each node where type information may be available.

This new AST would then be passed to the new type checking functions. These new functions would type check as normal, but they would also instantiate the Type Option Reference at each node to be the correct type of that node of the tree, which would be of the form Ref SOME X, where X is the type of that node, much in the way that the tool for this project did. The type checker would then assume that the types of any sub nodes were the types as held in their specific Type Option variable. Then, in the case where the program is correctly typed, the compiler continues as normal (with the newly type checked AST being converted back to the old type of AST – this simply means removing the Type option from each node)[1]. In the other case, however, the compiler would behave more like the compiler created by this project.

[1] Note that strictly speaking the compiler could ignore the extra information in the AST, however to allow the tool to be integrated with current compilers with the minimum of effort it would be better for the type checking functions to remove the extra information, in the case when it type checks correctly.

Initially, the compiler would note that a type error had occurred, and its location (using the location information). Then a GUI window would be displayed, telling the user that a type error had been discovered, and displaying the text of the error in question. The user could then use the tool, much as the tool designed for this project can be used.

Once the user decides that She actually wishes to use the tool, she can highlight any expression desired in the program, receiving its type information. She can then request a change in type of the given expression. At this point, the tool would firstly check the validity of the type requested for the expression using a small type checking function specifically designed for these cases. The tool, will then (assuming the type is properly typed) store both the location of the expression and the type in a small table of all the type changes that a user wished to make to the program. If the expression having its type changed were an atomic expression then the table would also note the expression, this allows a change in one part of the program to affect all variables of the same name.

The user having made as many changes as were necessary, could request that the program be re-type checked, taking into account the new types. The type checker then proceeds to re-type check the AST, however this time the type checker looks up the table of type changes at each node it encounters. The type checker will assume that the node has the type as given by the user, and stored in the lookup table if these conditions are fulfilled:

- the location information of the current node matches any of the entries in the tree, or;

- the current node is an atomic node that appears in the table, and the current nodes location information is greater than that of the node in the table (i.e. the node in the table occurs before the current node in the program) and the sco pe of the two nodes is the same.

When the type checker has finished type checking, the lookup table would be added to the top of the undo table, in the reverse order from the order that the changes were requested – that is, the last type change requested would be at the top of the table. Note that the lookup table would have elements removed from it only when they are also removed from the undo table, and in fact, these could be the same table if wished. The user would then be informed of whether or not the type checking was successful. In either event, the user will have the option to continue to use the tool, or to exit the tool, and return to the previous state in the compiler.

This approach to the tool has many benefits over the approach taken for this project. One such benefit is that the type of any phrase, that has a type, can be changed (that is, not simply expressions, but patterns too), and this change will affect the rest of the program (most notably in the case when the expression is atomic). In addition, the undo or redo facilities of such a tool would be much easier to implement.

The two most interesting and useful benefits of this projected tool over the tool as created for this project are; that changing the type of a sub-expression of an expression that has already has had its type altered would be both possible, and be effective. That is, if the sub-expression were an atomic expression then any occurrences of that atomic expression later in the program would have this new type. Further more, a list of the changes that the user applied to the program whilst using the tool could be saved to a file, allowing the user to see what was done during any session using the tool, hence all steps taken need not be remembered, as they would be recorded.

The action of changing the type of all occurrences of any atomic expression and having this change reflected throughout identical expressions in the program is not only useful but is what a user might naturally expect to happen in certain cases. This would allow errors to be located more quickly and easily.

Other advantages of this approach are that zooming would be easier to implement in this way (although as Alison Keane [Keane1999] discovered, this is not a trivial task). In addition, with only minor revisions to the AST and the type checker, such a tool has the potential to be incorporated easily into most, if not all, SML compilers.

Other minor benefits of such a tool are; firstly, it is much more efficient than the tool presented in this report, due to the lack of repetition of code this tool would have. Secondly, it would be easier to maintain and keep up to date with the compiler as it changes. Furthermore, given that the tool simply takes the AST the compiler creates, and then returns a type checked AST, as the compiler requires, any changes to the compiler should be easy to incorporate into the tool. In fact, as long as the AST remains the same, any changes to the compiler can be made without affecting the tool at all.

## 6.3 Overall Conclusion

The initial aim for this project was the creation of a tool to assist in the location and correction of type errors in ill-typed ML programs. The problem was approached in a similar way to that which was taken by Alison Keane, and an attempt was made to improve on several aspects of her work. These were; firstly integrating the tool into an existing ML compiler, secondly extending the coverage of the language by the tool, and thirdly creating a new and better user interface. The first two of these goals were achieved simultaneously, so then the aims became twofold; to integrate the project she provided into an existing ML compiler and to improve the user interface.

Although this project failed to fully meet the aim initially laid out, nevertheless there have been several positive results from the research. Firstly, it has shown that the approach used, of changing the type of a given expression, is not the best approach to the problem, and would be difficult to carry through to completion. Secondly, it has shown a new way of creating a tool, fulfilling the initial goal of this project. Lastly, it has given a detailed account of how such a tool could be created, what it should do, and how best to do it. As such, future work in the same area should be much more successful due to the discoveries of this project, and the preceding work.

In that case, this project could then be considered a success, since although it did not accomplish all that it set out to do, it does provide a good basis for the future work that will no doubt follow on from it. A lot of work is still required in this area however, and, as Alison Keane’s project showed, a tool such as the one attempted here would be of great use in the location of type errors in programs. The creation of such a tool would definitely ease the introduction of programmers to ML – allowing her to discover what is going on inside the programs she writes, and why the errors the compiler reports are errors and what the real problem is or where it lies.

# Appendices

# Appendix A: Grammar for Moscow ML

A more detailed description of the grammar can be found in [mosref].

| | | |
|---|---|---|
| *exp* | ::= | *infexp* |
| | | *exp* : *ty* |
| | | $exp_1$ andalso $exp_2$ |
| | | $exp_1$ orelse $exp_2$ |
| | | *exp* handle *match* |
| | | raise *exp* |
| | | if $exp_1$ then $exp_2$ else $exp_3$ |
| | | while $exp_1$ do $exp_2$ |
| | | case *exp* of *match* |
| | | fn *match* |
| *infexp* | ::= | *appexp* |
| | | $infexp_1$ *id* $infexp_2$ |
| *appexp* | ::= | *atexp* |
| | | *appexp atexp* |
| *atexp* | ::= | *scon* |
| | | <op> *longvid* |
| | | {<exprow>} |
| | | # *lab* |
| | | () |
| | | ($exp_1$, ... $exp_n$) |
| | | [$exp_1$, ... , $exp_n$] |
| | | #[$exp_1$, ... , $exp_n$] |
| | | ($exp_1$; ... ; $exp_n$) |
| | | let *dec* in $exp_1$; ... ; $exp_n$ end |
| | | [ structure *modexp* as *sigexp* ] |
| | | [ functor *modexp* as *sigexp* ] |
| | | ( *exp* ) |
| *exprow* | ::= | *lab* = *exp* <, *exprow*> |
| *match* | ::= | *mrule* < \| *match* > |
| *mrule* | ::= | *pat* => *exp* |
| *dec* | ::= | val *tyvarseq valbind* |
| | | fun *tyvarseq fvalbind* |
| | | type *typbind* |
| | | datatype *datbind* <withtype *typbind*> |
| | | datatype *tycon* = datatype *tyconpath* |
| | | abstype *datbind* <withtype *typbind*> *abstype* with *dec* end |
| | | exception *exbind* |
| | | local $dec_1$ in $dec_2$ end |
| | | open *longstrid1...longstridn* |
| | | structure *strbind* |
| | | functor *funbind* |
| | | signature *sigbind* |
| | | *(empty declaration)* |
| | | *dec1* <;> $dec_2$ |

```
                   infix <d> id_1...id_n
                   infixr <d> id_1...id_n
                   nonfix id_1...id_n

valbind      ::=   pat = exp < and valbind >
                   rec valbind

fvalbind     ::=   <op> var atpat_11 ... atpat_1n <:ty> = exp_1
                   | <op> var atpat_21 ... atpat_2n <:ty> = exp_2
                   | ...
                   | <op> var atpat_m1 ... atpat_mn <:ty> = exp_m
                                            <and fvalbind>

typbind      ::=   tyvarseq tycon = ty <and typbind>

datbind      ::=   tyvarseq tycon = conbind <and datbind>

conbind      ::=   <op> vid <of ty> <| conbind>

exbind       ::=   <op> vid <of ty> <and exbind>
                   <op> vid = op longvid <and exbind>

tyconpath    ::=   longtycon
                   longtycon where strid = modexp

ty           ::=   tyvar
                   { <tyrow> }
                   tyseq tyconpath
                   ty_1 * ... * ty_n
                   ty_1 ? ty_2
                   [ sigexp ]
                   ( ty )

tyrow        ::=   lab : ty <, tyrow>

atpat        ::=   _
                   scon
                   <op> longvid
                   { <patrow> }
                   ()
                   (pat_1, ... , pat_n)
                   [pat_1, ... , pat_n]
                   #[pat_1, ... , pat_n]
                   ( pat )

patrow       ::=   ...
                   lab = pat <, patrow>
                   lab <:ty> <as pat> <, patrow>

pat          ::=   atpat
                   <op> longvid atpat
                   pat_1 vid pat_2
                   pat : ty
                   <op> var <:ty> as pat

modexp       ::=   appmodexp
                   modexp : sigexp
```

| | | |
|---|---|---|
| | | *modexp* :> *sigexp* |
| | | functor ( *modid* : *sigexp* ) => *modexp* |
| | | functor *modid* : *sigexp* => *modexp* |
| | | rec ( *strid* : *sigexp* ) *modexp* |
| *appmodexp* | ::= | *atmodexp* |
| | | *appmodexp atmodexp* |
| *atmodexp* | ::= | struct *dec* end |
| | | <op> *longmodid* |
| | | let *dec* in *modexp* end |
| | | ( *dec* ) |
| | | ( *modexp* ) |
| *strbind* | ::= | *strid* <*con*> = *modexp* <and *strbind*> |
| | | *strid* as *sigexp* = *exp* <and *strbind*> |
| *funbind* | ::= | *funid* $arg_1$ ... $arg_n$ <*con*> = modexp <and *funbind*> |
| | | *funid* ( spec ) <*con*> = *modexp* <and *funbind*> |
| | | *funid* as *sigexp* = *exp* <and *funbind*> |
| *sigbind* | ::= | *sigid* = *sigexp* <and *sigbind*> |
| *con* | ::= | : *sigexp* |
| | | :> *sigexp* |
| *arg* | ::= | ( *modid* : *sigexp* ) |
| | | *modid* : *sigexp* |
| *sigexp* | ::= | sig *spec* end |
| | | *sigid* |
| | | *sigexp* where *typreal* |
| | | functor ( *modid* : *sigexp* ) ? *sigexp* |
| | | functor *modid* : *sigexp* ? *sigexp* |
| | | rec ( *strid* : *sigexp* ) *sigexp* |
| *typreal* | ::= | type *tyvarseq longtycon* = *ty* <and *typreal*> |
| spec | ::= | val *tyvarseq valdesc* |
| | | type *typdesc* |
| | | type *typbind* |
| | | eqtype *typdesc* |
| | | datatype *datdesc* <withtype *typbind*> |
| | | datatype *tycon* = datatype *tyconpath* |
| | | exception *exdesc* |
| | | structure *strdesc* |
| | | functor *fundesc* |
| | | signature *sigbind* |
| | | include *sigid1 ... stridn* |
| | | local *lspec* in *spec* end |
| | | *(empty)* |
| | | *spec* <;> *spec* |
| | | spec sharing type $longtycon_1$ = ... = $longtycon_n$ |
| | | *spec* sharing *longstrid1* = ... = *longstridn* |
| | | infix <d> $id_1$ ... $id_n$ |
| | | infixr <d> $id_1$ ... $id_n$ |

nonfix *id*$_1$ ... *id*$_n$

*lspec* ::= open *longstrid*$_1$ ... *longstrid*$_n$
type *typbind*
local *lspec* in *lspec* end
*(empty)*
*lspec* <;> *lspec*

*valdesc* ::= *vid* : *ty* <and *valdesc*>

*typdesc* ::= *tyvarseq* *tycon* <and *typdesc*>

*datdesc* ::= *tyvarseq* *tycon* = *condesc* <and *datdesc*>

*condesc* ::= *vid* <of *ty*> <| *condesc*>

*exdesc* ::= *vid* <of *ty*> <and *exdesc*>

*strdesc* ::= *strid* : *sigexp* <and *strdesc*>

*fundesc* ::= *funid* : *sigexp* <and *fundesc*>

# Appendix B: Moscow ML Abstract Syntax Tree

Note these are the main definitions only, for several of the types used here definitions are omitted for the sake of clarity. The entire AST representation can be found in the Asynt and related modules in the Moscow ML compiler.

```
type LocString = Location * string;

type VId = LocString;
type TyCon = LocString;

type ModId = LocString;
type FunId = LocString;
type SigId = LocString;

type LongModId = IdInfo;
type LongModIdInfo = IdInfo * ((Environment option) ref);
type LongVId = IdInfo;
type LongTyCon = IdInfo;

type TyVar = IdInfo;
type TyVarSeq = TyVar list;

datatype TyConPath' =
    LONGtyconpath of LongTyCon
  | WHEREtyconpath of LongTyCon * ModId * ModExp

and Ty' =
    TYVARty of TyVar
  | RECty of Ty Row
  | CONty of Ty list * TyConPath
  | FNty of Ty * Ty
  | PACKty of SigExp
  | PARty of Ty


and InfixPat =
    UNRESinfixpat of Pat list
  | RESinfixpat of Pat
and Pat' =
    SCONpat of SCon * Type option ref
  | VARpat of LongVId
  | WILDCARDpat
  | NILpat of LongVId
  | CONSpat of LongVId * Pat
  | EXNILpat of LongVId
  | EXCONSpat of LongVId * Pat
  | EXNAMEpat of Lambda.Lambda
  | REFpat of Pat
  | RECpat of RecPat ref
  | VECpat of Pat list
  | INFIXpat of InfixPat ref
  | PARpat of Pat
  | TYPEDpat of Pat * Ty
  | LAYEREDpat of Pat * Pat
```

```
and RecPat =
    RECrp of Pat Row * RowType option
  | TUPLErp of Pat list
and VIdPathInfo =
    RESvidpath of LongVId
  | OVLvidpath of LongVId * OvlType * Type
and InfixExp =
    UNRESinfixexp of Exp list
  | RESinfixexp of Exp
and Exp' =
    SCONexp of SCon * Type option ref
  | VIDPATHexp of VIdPathInfo ref
  | RECexp of RecExp ref
  | VECexp of Exp list
  | LETexp of Dec * Exp
  | PARexp of Exp
  | APPexp of Exp * Exp
  | INFIXexp of InfixExp ref
  | TYPEDexp of Exp * Ty
  | ANDALSOexp of Exp * Exp
  | ORELSEexp of Exp * Exp
  | HANDLEexp of Exp * Match
  | RAISEexp of Exp
  | IFexp of Exp * Exp * Exp
  | WHILEexp of Exp * Exp
  | FNexp of Match
  | SEQexp of Exp * Exp
  | STRUCTUREexp of ModExp * SigExp * (ExMod option) ref
  | FUNCTORexp of ModExp * SigExp * (ExMod option) ref

and RecExp =
    RECre of Exp Row
  | TUPLEre of Exp list

and MRule = MRule of (Pat list ref) * Exp

and FunDec =
    UNRESfundec of TyVarSeq *  (FValBind list)
  | RESfundec of Dec

and Dec' =
    VALdec of TyVarSeq * (ValBind list * ValBind list)
  | PRIM_VALdec of TyVarSeq * (PrimValBind list)
  | FUNdec of FunDec ref
  | TYPEdec of TypBind list
  | PRIM_TYPEdec of TyNameEqu * TypDesc list
  | DATATYPEdec of DatBind list * TypBind list option
  | DATATYPErepdec of TyCon * TyConPath
  | ABSTYPEdec of DatBind list * TypBind list option * Dec
  | EXCEPTIONdec of ExBind list
  | LOCALdec of Dec * Dec
  | OPENdec of LongModIdInfo list
  | STRUCTUREdec of ModBind list
  | FUNCTORdec of FunBind list
  | SIGNATUREdec of SigBind list
  | EMPTYdec
  | SEQdec of Dec * Dec
  | FIXITYdec of InfixStatus * string list
```

```
and ValBind = ValBind of (Pat ref) * Exp

and FClause = FClause of (Pat list ref) * Exp

and ConBind = ConBind of IdInfo * Ty option

and ExBind =
    EXDECexbind of IdInfo * Ty option
  | EXEQUALexbind of IdInfo * IdInfo

and ModBind = MODBINDmodbind of ModId * ModExp
            | ASmodbind of ModId * SigExp * Exp
and FunBind =
              FUNBINDfunbind of FunId * ModExp
            | ASfunbind of FunId * SigExp * Exp
and SigBind = SIGBINDsigbind of SigId * SigExp
and FunctorSort =
    Generative of bool (* true if conforms to SML 97 *)
  | Applicative
and ModExp' =
    DECmodexp of Dec
  | LONGmodexp of LongModId
  | LETmodexp of Dec * ModExp
  | PARmodexp of ModExp
  | CONmodexp of ModExp *  SigExp
  | ABSmodexp of ModExp *  SigExp
  | FUNCTORmodexp of FunctorSort *  ModId * (IdKindDesc ref) * SigExp
* ModExp
  | APPmodexp of ModExp * ModExp
  | RECmodexp of ModId * (RecStr option) ref * SigExp * ModExp
and ModDesc = MODDESCmoddesc of ModId * SigExp
and FunDesc = FUNDESCfundesc of FunId * SigExp
and SigExp' =
    SPECsigexp of Spec
  | SIGIDsigexp of SigId
  | WHEREsigexp of SigExp * TyVarSeq * LongTyCon * Ty
  | FUNSIGsigexp of FunctorSort * ModId * SigExp * SigExp
  | RECsigexp of ModId * SigExp * SigExp
and Spec' =
    VALspec of TyVarSeq * ValDesc list
  | PRIM_VALspec of TyVarSeq * (PrimValBind list)
  | TYPEDESCspec of TyNameEqu * TypDesc list
  | TYPEspec of TypBind list
  | DATATYPEspec of DatBind list * TypBind list option
  | DATATYPErepspec of TyCon * TyConPath
  | EXCEPTIONspec of ExDesc list
  | LOCALspec of Spec * Spec
  | OPENspec of LongModIdInfo list
  | EMPTYspec
  | SEQspec of Spec * Spec
  | INCLUDEspec of SigExp
  | STRUCTUREspec of ModDesc list
  | FUNCTORspec of FunDesc list
  | SHARINGTYPEspec of Spec * LongTyCon list
  | SHARINGspec of Spec * (Location * LongModId list)
  | FIXITYspec of InfixStatus * string list
  | SIGNATUREspec of SigBind list
```

```
and Sig =
    NamedSig of {locsigid : SigId, sigexp: SigExp}
  | AnonSig of Spec list
  | TopSpecs of Spec list

and Struct =
    NamedStruct of {locstrid : ModId, locsigid : SigId option,
                decs : Dec list}
  | Abstraction of {locstrid : ModId, locsigid : SigId,
                decs : Dec list}
  | AnonStruct of Dec list
  | TopDecs of Dec list

withtype TyConPath = Location * TyConPath'
and Ty = Location * Ty'
and Pat = Location * Pat'
and Exp = Location * Exp'
and ModExp = Location * (ModExp' * (ExMod option) ref)
and SigExp = Location * SigExp'
and Spec = Location * Spec'
and ValDesc = IdInfo * (Location * Ty')
           (* IdInfo * Ty *)
and ExDesc = IdInfo * (Location * Ty') option
          (* IdInfo * Ty option *)
and LocString = Location * string
and Match = MRule list
and Dec = Location * Dec'
and PrimValBind = IdInfo * (Location * Ty') * int * string
               (* IdInfo * Ty * int * string *)
and FValBind = Location * FClause list
and TypBind = TyVarSeq * TyCon * (Location * Ty')
           (* TyVar list * IdInfo * Ty *)
and TypDesc = TyVarSeq * TyCon
and DatBind = TyVarSeq * TyCon * ConBind list
end;
```

# Bibliography

## Book and journal references:

[Appel1998] Appel, A. W. 1998. *Modern Compiler Implementation.* Cambridge University Press

[Dix1998] Dix, A., Finlay, J., Abowd, G. and Beale, R. 1998. *Human-Computer Interaction* Prentice Hall

[Keane1999] Keane, A. 1999. *A tool for investigating type errors in ML program* The University of Edinburgh

[McAdam] McAdam B. J. *Generalising Techniques for Type Debugging* Laboratory for the Foundations of Computer Science, The University of Edinburgh

[Paulson1996] Paulson, L. C. 1996.*ML for the Working Programming (Second Edition).* Cambridge University Press 1996

[Sethi1989] Sethi, R. 1989.*Programming Languages: Concepts and Constructs.* Addison-Wesley

## Internet references:

| | |
|---|---|
| [mosmlgl] mosmlgl: | http://www.home.gil.com.au/~mthomas/ |
| [mGtk] mGtk: | http://mgtk.sourceforge.net/ |
| [sml-tk] sml-tk: | http://www.informatik.uni-bremen.de/~cxl/sml_tk/ |
| [mosml] Moscow ML: | http://www.dina.kvl.dk/~sestoft/mosml.html |
| [SML/NJ] SML/NJ: | http://cm.bell-labs.com/cm/cs/what/smlnj/ |
| [mosman] Moscow ML Owners Manual: | ftp://ftp.dina.kvl.dk/pub/mosml/doc/manual.pdf |
| [mosref] Moscow ML Language Refference: | ftp://ftp.dina.kvl.dk/pub/mosml/doc/mosmlref.pdf |