Evaluating Environments for Functional Programming

Jon Whittle* Recom Technologies, NASA Ames Research Center Moffett Field, CA 94035 jonathw@ptolemy.arc.nasa.gov

Andrew Cumming Dept of Computer Studies, Napier University, 219 Colinton Road, Edinburgh EH14 1DJ, Scotland. andrew@dcs.napier.ac.uk

Abstract

Functional programming presents new challenges in the design of programming environments. In a strongly typed functional language, such as ML, much conventional debugging of runtime errors is replaced by dealing with compile time error reports. On the other hand, the cleanness of functional programming opens up new possibilities for incorporating sophisticated correctness-checking techniques into such environments. $C^{Y}NTHIA$ is a novel editor for ML that both addresses the challenges and explores the possibilities. It uses an underlying proof system as a framework for automatically checking for semantic errors such as non-termination. In addition, $C^{Y}NTHIA$ embodies the idea of *programming by analogy* — whereby users write programs by applying abstract transformations to existing programs. This paper investigates $C^{Y}NTHIA$'s potential as a novice ML programming environment. We report on two studies in which it was found that students using $C^{Y}NTHIA$ commit fewer errors and correct errors more quickly than when using a compiler / text editor approach.

1 Introduction

Functional programming (FP) is now taught widely in universities as introductory computing. However, despite the emergence of some excellent beginners' texts [Ullman, 1994, Michaelson, 1995], only limited attention has been paid to the design of programming environments that augment the learning process. This paper investigates what features of environments could most benefit novice functional programmers, with particular reference to the language ML [Paulson, 1991]. Note that we are concerned here with environments that support *real* programming, not Intelligent Tutoring Systems that are limited to a small number of pre-defined examples.

^{*}Formerly University of Edinburgh

Most students of FP write programs at a command-line interface or in a text editor the contents of which are then compiled. Current compilers for ML (e.g. SML of New Jersey (SML-NJ) and CamlLight) give only crude error information and can only compile complete definitions. The primitiveness of this approach can result in an arduous edit-compile-edit loop in which users incorrectly patch their programs because of the lack of clear feedback [Whittle, 1999]. Note that FP presents new difficulties because languages like ML, with polymorphic static type checking, replace much conventional runtime debugging with debugging of the type conflicts which inevitably arise during compilation. The design of FP environments, therefore, must be undertaken with a new set of considerations in mind.

 $C^{Y}NTHIA$ is a novel editor for a functional subset of ML which overcomes some of these difficulties. It incorporates an incremental approach to programming whereby the users' programs are checked for errors as they are written and errors are flagged to the user immediately (although they need not necessarily be corrected straight away). This means both that incomplete programs can be checked for errors and that many errors can be trapped early on, thus avoiding sometimes dire consequences later. These ideas are captured in the notion of *programming by analogy* — whereby the user transforms existing function definitions using a sequence of abstract editing commands. These commands are mostly correctness-preserving and in the case that errors are introduced, the errors can be highlighted easily to the user rather than providing cryptic system messages.

Traditional syntax-directed editors (e.g. [Teitelman & Reps, 1984, Hansen, 1971, Teitelman, 1975]) prevent the user from writing syntactically incorrect programs by forcing the user to build programs from pre-defined templates. We extend this approach in two ways — by defining a set of editing commands or transformations that generalise the template approach and are oriented towards FP, and by providing sophisticated semantic guarantees such as termination checking. It is an interesting question to decide which semantic guarantees are potentially the most useful to programmers. There are many techniques available for carrying out automated analysis of programs but they all require a high level of expertise to be used effectively. We consider a few particular ideas from Formal Methods and restrict them in such a way that the analysis can be done completely automatically and hence can be used as part of an everyday, programming environment.

2 Novice Functional Programming

2.1 A Brief Introduction to ML

Before going on to describe $C^{Y}NTHIA$, we will acquaint the reader with ML. There are a number of different dialects of ML. Throughout the rest of the paper, we confine attention to the recognised dialect Standard ML [Milner *et al.*, 1990] which is the most widely used. We do not attempt to describe functional programming here but refer the unfamiliar reader to [Bird & Wadler, 1988]. The syntax of ML is best illustrated by example:

fun map f nil = nil
| map f (h::t) = (f h) :: map f t;

where nil denotes the empty list and :: is the *cons* operator for constructing lists. This example illustrates many features of ML:

- ML is functional. Each program in ML consists of a set of definitions. Each definition is a set of equations that define a particular function. map is a function whose inputs are another function, f, and a list, and whose output is the list formed by applying f to each element of the list. For example, if we evaluated map (op +) [(1,2),(3,2)], the result returned would be [3,5]¹.
- ML uses recursion and pattern matching. Recursion is used extensively where procedural languages would use a loop. Functions are often defined by pattern matching in *map*, nil and h::t are patterns used to define the function, where h::t represents a non-empty list with head h and tail t. Pattern matching gives us a way of performing a case analysis on a datatype.
- ML is strongly typed. Every object in the language belongs to a type such as list, integer or tree. ML employs type inference to automatically infer types at compile time. This frees the user from declaring types in most cases. For example, an ML compiler would deduce that map has type ('a -> 'b) -> 'a list -> 'b list. 'a and 'b are polymorphic types. 'a list is a polymorphic list type i.e. the type of elements of the list is unspecified but the elements must all have the same type. Polymorphism allows code to be shared between different data structures while retaining the security associated with strong typing.

¹[x,y] is ML shorthand for x::y::nil

- **Higher order programming**. This means that ML functions can take other functions as arguments, results can be functions, and data structures can contain functions.
- ML is impure. It contains a small number of imperative features included for practical reasons, mainly for input/output.

 $C^{Y}NTHIA$ is concerned only with a purely functional subset of Core² ML. This is on the grounds that purely functional definitions are easier to analyse and also that the impure features are less likely to be used by novices than by expert programmers.

2.2 Requirements for a Functional Programming Editor

Numerous studies (see IEEE Standard 1044, for example) examine what kinds of errors programmers make. However, there have been very few studies in the context of FP. [Bental, 1995] reports on experiences with the Ceilidh system at Heriot-Watt University, Scotland. 60 students used Ceilidh as part of a course on ML programming. The students sent email to human tutors when problems were encountered and these email queries were classified. Some of the major problems noted were recursion (recursion is widely accepted to be difficult to learn [Anderson *et al.*, 1988]), pattern matching and type errors. Our own study at Napier University [Whittle, 1999] backs up these claims. Over a two hour period, 14 novice ML students, in the fourth week of a course on ML, were monitored during a normal tutorial session. Their interactions with the SML-NJ compiler were logged and any errors were noted. 193 type errors were found in total, compared with 114 other semantic errors and 70 syntax errors. The dominance of type errors suggests that improved type feedback facilities would be useful. [Jun & Michaelson, 1998] also found that students have difficulties understanding types and suggests a graphical representation of type errors to overcome this.

In our study, errors were classified according to the compiler's error message. Because error classification was done automatically, some errors may have been classified incorrectly. One problem with ML is that an error that is essentially a syntax error can have an alternative parse and will therefore only show up as a type error. Such incorrect classifications can be tolerated, however, as these situations are especially confusing to the user and so we are essentially weighting particular kinds of errors.

Note that the problem with type errors cannot be solved by merely rephrasing the error messages. The underlying problem is that the type inference algorithm is a

²ML has a module system for structuring large programs. Core ML excludes the module system.

complex one and there are subtle interactions between polymorphic variables occurring in distant parts of the program. As a simple example, consider the code from $[Duggan \& Bent, 1996]^3$:

...
$$F y \dots; \dots y=(3,x) \dots; \dots F(z,4.5)$$

The kind of reasoning needed to determine why x has type real is highly non-trivial, especially if the three expressions occur far apart in the input file. To provide non-cryptic messages would require an explanation of the interactions, [Duggan & Bent, 1996], but these explanations tend to be lengthy and complex. C^YNTHIA addresses such problems in the following way. Programming by analogy raises the level at which programs are written. Rather than writing low-level code, programs are constructed in pieces. In addition, since the editing commands are correctness-preserving, the user is prevented from making many errors (for instance, syntax errors can never be introduced into C^YNTHIA programs). The kinds of correctness that C^YNTHIA checks for are as follows.

2.2.1 Syntactic Correctness

All programs must be syntactically correct. Expressions are checked for syntax errors as they are entered (rather than waiting until compilation). Hence, incorrect syntax is *never* introduced into a program.

2.2.2 Type correctness

Type declarations usually need not be given in ML because type inference can automatically infer them at compile time. However, as noted above, subtle interactions mean that type inference leads to confusing error messages. C^YNTHIA insists that users give a type declaration which helps to clarify these interactions because the user is forced to make his/her intentions explicit. By removing type inference, we remove one of the key advantages of ML — that the user need not write type declarations. However, because functions are only ever written by making modifications to existing functions, the user never writes down a full type declaration for a function, but merely adds to it in piece-meal fashion. Hence, the extra burden is not unacceptable. And the advantage is that type feedback can be much more focussed.

Syntax errors are never allowed in a $C^{Y}NTHIA$ definition. In contrast, type errors can be introduced. This is because it is generally impossible to transform one well-typed

 $^{^{3}\,;}$ is the ML notation for connecting sequences of expressions, which should be evaluated in left to right order.

program into another without passing through intermediate, ill-typed states. Any type errors that do occur are highlighted to the user in a different colour. In general, it is very difficult to highlight the *actual* source of a type error [Duggan & Bent, 1996]. This is because the type inference algorithm breaks down when a type is derived that is inconsistent with a previously derived type. However, expressions far apart in the input file may have inconsistent types. Hence, type errors may be reported at locations distant from their actual source. C^YNTHIA is less susceptible to this problem because each definition is given a type declaration. The highlighting is then made with respect to this definition. The other reason why C^YNTHIA can do better is because large fragments of code are guaranteed well-typed *a priori* because they are introduced by editing commands. Hence, there are fewer type inconsistencies that need to be reported.

2.2.3 Static Semantic correctness

User entry is accepted only if devoid of static semantic errors (e.g. undeclared variables, undeclared functions)⁴. Such errors can be introduced by editing commands, however. This usually happens if a command removes an object from the definition — for instance, suppose the program contains the definition of a local variable. If the construct that introduced this variable is removed, the remaining program fragment may still contain references to that variable. Such static semantic errors are highlighted to the user (in a colour different than type errors).

2.2.4 Well-definedness

A well-defined function definition is one that is neither over- nor under-defined. In terms of ML, this means that all patterns must exhaustively cover the datatype that they are defined over and must contain no overlapping patterns. The following function is under-defined:

fun addlist (x::xs) (y::ys) = (x:int) + y :: addlist xs ys;

Note that there is no pattern for when either of the input lists is empty. Under-defined functions are allowed in ML (they are flagged as warnings at compile-time) but they can lead to run-time errors. In this example, a call to *addlist* will always produce an error. The following function is over-defined⁵:

 $^{{}^{4}}$ Type errors are strictly static semantic errors but we make a distinction for purposes of presentation.

 $^{^{5}}tl$ returns the tail of a polymorphic list.

fun length x = 1 + length (tl x)
 length nil = 0;

If tl is defined such that tl nil gives an exception, then length nil will also produce an exception. Swapping the order of the two clauses would work as expected because ML imposes a top-to-bottom ordering on the clauses (essentially ignoring the ambiguity). In general, ML's top-bottom ordering could lead to errors. The user may be unaware that for a certain input the function is defined twice and ML may not pick up the expected value. For this reason, CYNTHIA restricts the user to well-defined functions. In *length* above, x would have to be replaced by (h::t).

Students tend to understand the most commonly occurring patterns fairly well, as they are stressed in courses, but patterns can become arbitrarily complex and must be defined over other datatypes. It is at this point that novices become confused. To define a function via pattern matching, each expression of the relevant datatype must match exactly one pattern. Otherwise, run-time errors can occur. Observations of students showed that they had a good deal of difficulty in formulating well-defined patterns. Although non-exhaustive functions can occasionally be useful (and are therefore signalled only as warnings, not errors, by compilers), it is generally considered a good idea to write total functions wherever possible, especially when learning the language.

It is impossible to create ill-defined patterns in $C^{Y}NTHIA$ as all patterns are built up incrementally using the editing command MAKE PATTERN which is guaranteed correct.

2.2.5 Termination

Students' lack of understanding of recursion can lead to the writing of programs containing infinite loops. In our observations of students [Whittle, 1999], we found that this kind of error is particularly difficult for students to uncover. $C^{Y}NTHIA$ explicitly checks for termination. If the user attempts to write a non-terminating program, they will be told immediately and banned from making this edit. Termination checking is an undecidable problem so $C^{Y}NTHIA$ restricts the user to a decidable subset of terminating programs. This is the set of Walther Recursive programs [McAllester & Arkoudas, 1996], which contains a wide variety of recursive programs sufficient for use in real programming situations, such as multiple recursions, nested recursions, recursion with accumulators etc. The only other programming environment of which the authors are aware that checks for termination is the recursion ed*itor* [Bundy *et al.*, 1991]. However, this editor is severely restricted in the functions which can be accepted. Rather than having a general termination checker to hand, the *recursion editor* relies on the syntactic nature of the programs and hence realistic programming is not possible.

2.3 Other Editors

One of the most common attempts to support students learning recursion is the templatebased approach Bhuiyan et al., 1994, Gegg-Harrison, 1991, Kirschenbaum et al., 1989, Bowles & Brna, 1993]. Rather than writing recursive (or indeed non-recursive) algorithms from scratch, users call up templates. These may be schematic representations that need to be filled in, concrete programs that are transformed using special commands, or a combination of both. These tools are generally inadequate, however. Too many templates are needed to cover the whole of the target language. This leads to two extremes. Either a very large number of (often very specific) templates are provided that cover a wide subset of the language but which makes remembering which template to use difficult, or a small number of less-specific templates are provided in which case the subset of the language supported is overly restricted. The recursion editor [Bundy et al., 1991] is one of the better examples of these systems. However, the problem here is that, although a set of transformational commands is provided, the order of application of these commands is crucial, leading to a situation where the user has written a partial program but cannot complete it without undoing a large number of previous steps (or not be able to complete it at all). $C^{Y}NTHIA$ was inspired by the *recursion editor* and looks to overcome some of the difficulties associated with template-based systems. We claim that $C^{Y}NTHIA$ provides a small, compact set of commands supporting a sufficiently large subset of the ML language whilst not being sensitive to the order of application of commands. This has been possible to achieve partly because of the careful design of the commands and partly because of the uniformity of the functional style.

It has already been stated that most functional programmers use a text editor followed by compilation of the text file. Recently, two editors have become available which attempt to provide advanced programming support. MLWorks [Har1996] provides an integrated environment for Standard ML that incorporates debugging facilities, profiling and offers a graphical view of structured SML items. CtCaml [Rideau & Théry, 1997] is a structure editor for CamlLight [Leroy, 1995] designed using the Centaur [Aikins, 1980] framework. As well as structure editing facilities, CtCaml provides more advanced features such as the highlighting of type errors and type explanations. However, these environments have somewhat different objectives to C^YNTHIA . MLWorks is aimed at providing a fully integrated programming environment in the same way as has been provided for languages like C. Hence, it does not contain sophisticated correctness-checking or programming by analogy. CtCaml is an attempt to apply the Centaur technology to functional programming but its structure editing is based on syntax rather than semantics. CtCaml incorporates the type explanation algorithms of [Duggan & Bent, 1996]. These provide an alternative way of improving type error feedback but the explanations produced tend to be lengthy and difficult and their worth is yet to be realized cf. [Rideau & Théry, 1997], "On the one hand it [type explanation] appears to be too complex a tool for being used by a ML newcomer. (...) On the other hand, experts usually find the explanation too detailed to be of real help."

2.4 Scope

 $C^{Y}NTHIA$ supports a functional subset of Core ML. The main exclusions are mutually recursive programs and type inference. The former are disallowed because our termination checking technique, Walther Recursion, does not cover it. We also exclude the imperative features of ML which are used mainly for input/ouput. [Whittle, 1999] addresses how these omissions could be incorporated.

3 Overview of *C*^Y*NTHIA*

3.1 Writing Programs in CYNTHIA

 $C^{Y}NTHIA$ is implemented in SICStus Prolog v.3 and Tcl/Tk. Upon startup, the user selects a program from an initial library. which may then be edited and saved to the library. In this way, a user-customised library can be built up. Figure 1 shows $C^{Y}NTHIA$'s display. The user may highlight any part of the program by positioning the mouse over it. Clicking on the left mouse button brings up a menu of editing commands that could be applied at this point. Only those commands that are currently applicable are given as an option. After selecting a command, the user is presented with a dialog box to enter any necessary parameters for the command. The lower part of the display lists all *valid* recursive calls that are currently available for insertion into the program. Valid recursive calls are ones that would not introduce infinite loops if used in the definition. This list gives the user a ready reminder of which calls may be

used at a given time, and the list may be added to using the command ADD RECURSIVE CALL.

-			CYNTHIA		• •
FILE	EDIT	HELP	Questionnaire		
-	1			green = syntax error	100
'a list fun le ¦ler ;	—> int ength nil 1gth (h ::	= 0 t) = 1 + le	ngth change term add construct remove construct rename		
Func	tion: lenț	gth pattern pattern	1: 2: length t		

Figure 1: Graphical user interface to $C^{Y}NTHIA$

The following example is a typical task which might be given to students in the first half of a course on functional programming. The student is asked to write a number of table-accessing functions. Each of these functions has a very similar structure. It is natural, therefore, to use $C^{Y}NTHIA$ as a way of transforming one function definition into another. We give a textual representation of the interface here. The task is described as follows:

A "table" can be thought of as a store of data-items (the "values") where each entry is indexed by some data-item (the "keys"). The idea is that a value can be retrieved from the table by using the appropriate key, and that there is at most one entry for each key. The assumption is that tables will not be very large, i.e. there is no serious problem concerning the efficiency of search through a table. Choose a suitable representation for such tables and implement the following functions in a consistent manner:

newtable: a new table containing no entries.

addentry k v d: returns a new table which is the same as table d except that the

value entered for the key k is v; this value replaces any entry that might be there for k.

findentry k d: given a table d and key k, returns the value entered in d against k. Raises an exception if there is no such entry.

has entry k d: checks if the table d contains an entry for key k.

Let us assume that the student decides to implement the table as a polymorphic list where the odd elements are the keys and the even elements are the values⁶. Let us also assume that the student has already defined *newtable* and now wishes to define *hasentry*. The first thing that the student must do is to decide upon a starting definition. Since the tables will be represented by lists, the student chooses *length*:

'a list -> int
fun length nil = 0
| length (h::t) = 1 + length t;

The RENAME command can be used to carry out a global rename of this function. Selecting this command at the indicated $point^7$, gives:

Next, the student changes the output type of the definition using the command CHANGE TYPE:

The definition is now ill-typed (the underlined expressions are ill-typed with respect to the type declaration and are highlighted in CYNTHIA). CYNTHIA highlights all type inconsistencies in this way. The highlighting serves as a warning to the user but the errors need not be corrected immediately.

To access the keys, the program will need to recurse in two steps. To achieve this, the student invokes MAKE PATTERN at the boxed point. This command will replace t with two cases – when t is empty and non-empty:

⁶This assumes that the keys and values are of the same type but one could imagine a novice making this sort of assumption. Later, we give an alternative representation that would overcome this problem.

 $^{^7\}mathrm{Throughout}$ this paper, program code enclosed in boxes denotes the point at which the user has applied an editing command.

'a list -> bool
fun hasentry nil = 0
| hasentry (h::nil) = 1 + hasentry nil
| hasentry (h::h1::t) = 1 + hasentry t;

In the third clause, a new variable, h1, has been introduced. In addition, a recursive call using this new variable — namely, hasentry (h1::t), has been added to the list of valid calls. It can now be introduced into the program if required. The system knows that any definition involving this new recursive call will terminate. The definition is still missing a parameter for the key to search for. This can be introduced using the command ADD CURRIED ARGUMENT, which adds a parameter *throughout* the definition:

'a -> 'a list -> bool
fun hasentry k nil = 0
| hasentry k (h::nil) = 1 + hasentry k nil
| hasentry k (h::h1::t) = 1 + hasentry k t;

The user gives a name and type for the new argument and the type declaration is updated automatically. Finally, the user needs to change the output in each case. This can be done using the commands CHANGE TERM and ADD CONSTRUCT(IF THEN ELSE), giving⁸:

'a -> 'a list -> bool
fun hasentry k nil = false
| hasentry k (h::nil) = raise excep
| hasentry k (h::h1::t) = if k=h then true else hasentry k t;

excep is a previously defined exception. The above constitutes a reasonable definition of *hasentry*. The student now proceeds to define *findentry* and notices the similarity between the two definitions. To correctly define *findentry*, the user need only invoke RENAME, CHANGE TYPE and CHANGE TERM twice, to give:

```
'a -> 'a list -> 'a
fun findentry k nil = raise excep
| findentry k (h::nil) = raise excep
```

⁸Strictly, the second argument should now have type ''a list. ''a is a polymorphic type over which equality may be defined. CYNTHIA does not yet make any distinction between 'a and ''a. This is a minor oversight that should be corrected soon.

findentry k (h::h1::t) = if k=h then h1 else findentry k t; To construct addentry is almost as easy. The user needs to invoke ADD CURRIED ARGUMENT, CHANGE TYPE and CHANGE TERM:

At this point, the student may decide that a better representation for tables would have been a list of pairs. $C^{Y}NTHIA$ can be used to transform the definitions to suit this new representation. For example, applying REMOVE PATTERN to *hasentry* gives:

'a -> 'a list -> bool
fun hasentry k nil = false
| hasentry k (h::t) = if k=h then true else hasentry k t;
Applying CHANGE TYPE at the indicated point gives:

'a -> ('a * 'b) list -> bool
fun hasentry k nil = false
| hasentry k (h::t) = if k=h then true else hasentry k t;
Applying MAKE PATTERN at the point indicated gives a correct solution:

'a -> ('a * 'b) list -> bool
fun hasentry k nil = false

hasentry k ((h,h1)::t) = if k=h then true else hasentry k t;

To evaluate any of these definitions, the user must load them into a compiler.

The editing commands were designed with the intention that any definition may be transformed to any other definition (within the subset of ML supported). Of course, it makes sense to choose as a starting function a definition that is close to the target. However, the user will not be overly disadvantaged by making a sub-optimal choice. The commands fit together in such a way that it is easy to recover from an incorrect application of an editing command, even if other edits have been applied since. The intention also was to keep the set of commands as small as possible. This means that the commands are easy to learn and that very little experience of C^YNTHIA is needed before one can start editing programs. A wider range of commands could have been included, but it is thought that these would have added confusion to the system whilst only providing limited increases in functionality.

The other main advantage of the editing commands is that some programming tasks can be completed in a single step. The CHANGE TYPE command is a good example. Consider the *length* function again:

```
'a list -> int
fun length nil = 0
| length (h::t) = 1 + length t;
```

Suppose the user wishes to write a similar function but which counts the leaf nodes of a labelled, binary tree. Usually, this would mean changing the input patterns to reflect the constructors for trees. All of this can be achieved in one step by applying CHANGE TYPE and specifying a new tree type. The application of this command would result in:

'a tree -> int
fun length (leaf n) = 0
| length (node(h,t,t2)) = 1 + length t;

This definition can then be easily completed. Note that many editing commands make an arbitrary decision about which variable to use in an expression — e.g. to use length t rather than length t2 in this example. It is impossible to second guess the users' intentions and so it is necessary to make such an arbitrary choice, but these choices can be changed easily. The count function could be completed by changing the results and name of the function appropriately. CHANGE TYPE is one of C^YNTHIA 's most powerful commands. It can make a transformation between patterns of a wide variety of types⁹ in such a way that the target patterns are well-defined. It also provides a list of new recursive calls which the user may introduce, and these recursive calls are guaranteed not to violate termination. CHANGE TYPE works by finding a mapping between the old and new datatype definitions in such a way that (non-)recursive constructors are mapped to (non-)recursive constructors. Full details are beyond the scope of this paper and can be found in [Whittle, 1999].

⁹defined explicitly in [Whittle, 1999]

3.2 The Design of C^YNTHIA

From a technical point of view, the novel aspect of C^YNTHIA is that all programs are represented as proofs of a (weak) specification which correspond to programs in a way defined by the *proofs-as-programs* notion [Howard, 1980]. A description lies beyond the scope of this paper but can be found in [Whittle *et al.*, 1999]. Basically, the type of each function, along with lemmas needed for termination analysis, forms the specification of a so-called synthesis proof. This specification is a theorem, proved by the system and the user (by means of the editing commands). Each proof step corresponds to a program construct or correctness check. Hence, programs may be extracted from their proofs. This framework gives a flexible, sound way of representing functional programs and reasoning about their correctness. Note that C^YNTHIA 's interface means that the user requires no knowledge of the underlying proof technology and indeed is unaware of its existence.

4 Evaluation of C^YNTHIA

This section reports on empirical studies undertaken with two groups of students (SG1 and SG2) at Napier University, Scotland. The aim of the studies was to assess C^YNTHIA 's usefulness as a novice programming environment for ML. "Usefulness" is in the sense defined by the research questions below. As a point of comparison, C^YNTHIA was judged against the way in which students on previous courses had programmed — using the text editor TextEdit and compiling programs using the Standard ML of New Jersey compiler (henceforth, this approach will be called the TEA approach). The main research questions being asked by this study are:

- 1. How useful was $C^{Y}NTHIA$ as an editor for ML that guarantees correctness?
 - (a) How does the quantity of errors made using $C^{Y}NTHIA$ compare to TEA?
 - (b) How does $C^{Y}NTHIA$'s error feedback compare to TEA?
 - (c) How does the user's productivity rate using $C^{Y}NTHIA$ compare to TEA?
- 2. How does programming by analogy compare to writing a program from scratch?
 - (a) Are the editing commands well-designed?
 - (b) How easy is it to choose a starting (source) example?

These are summarised in Table 1. The criteria expand upon what is being asked. The measures describe which data were used to answer each question. The following methods of data collection were employed:

- I (SG1 and SG2). The students' sessions with C^YNTHIA were logged as they worked. For SG1, interactions with the compiler New Jersey of SML (SML-NJ)
 were also recorded. Interactions were logged during normal tutorial sessions in which students worked through a series of examples on a Web-based course [GIML, 1998]. These examples typically involved the student writing a function consisting of a few lines using pattern matching and recursions.
- II (SG1) Students were asked to take two tests, A and B, each consisting of three questions requiring the writing of a simple list recursive function (see Appendix B). Both A and B contained tasks of similar difficulty and were each allotted half an hour. SG1 was split into two groups, X and Y. X attempted part A of the test using C^YNTHIA and part B using TEA. Y attempted part A using TEA and part B using C^YNTHIA . The experiment allows a comparison of errors made by those using C^YNTHIA and those using TEA (see Table 2).
- **III** (SG1) A group of 4 students were videoed attempting four short exercises (see Appendix A). They were asked to attempt some questions with $C^{Y}NTHIA$ and some without and were given a time limit of 45 minutes. In this way, individual students' performance could be compared. The students were asked to verbalise what they did at each stage and were instructed that they could ask questions if they got really stuck.
- IV (SG1 and SG2) Informal observations were taken by both authors. Students were observed during their tutorial sessions interacting with and without $C^{Y}NTHIA$.

4.1 The Experimental Subjects

Both subject groups studied ML as part of a Formal Methods course at Napier University. The course lasted 14 weeks of which approximately 9 weeks was on ML. The students were given lectures each week and were then expected (although not forced) to attend a two hour, supervised tutorial session during which they would work through examples from a Web-based course [GIML, 1998] and could ask questions of the tutors. The ML course is divided into eight short tutorials consisting of the introduction

Question	Criteria	Measures
1a)	What kind of errors can be made?	I, II, IV
	How many errors are made?	
1b)	Are errors located easily?	I, III, IV
	Are they corrected quickly?	
1c)	Did students get through more examples?	III, IV
2a)	Were the commands easy to understand?	I, IV
	Were they at the right level of abstraction?	
	Were they consistent?	
2b)	How were examples chosen?	III, IV
	Is a library-indexing system necessary?	

Table 1: Research Questions — Criteria and Measures.

of new concepts and then exercises that the students could work through — Figure 2. In previous years, students had used the New Jersey SML compiler (version 0.93) [NJS1996] to compile their programs. In the early stages of the course, students tend to write programs directly into the New Jersey interpreter or cut and paste program fragments from the course notes. Later on, they would write programs in a text editor and then compile the program. Assessment on the course was by examination and also practical coursework. All experiments involving a compiler were done using the SML-NJ compiler.

Lesson 1: Expressions and simple functions.
Lesson 2: Types, bindings, pattern matching and lists.
Lesson 3: Simple recursion on integers.
Lesson 4: List processing including recursion.
Lesson 5 : Partial functions, overlapping patterns, anonymous functions, more complex recursion.
Lesson 6: Higher-order programming.
Lesson 7: User-defined types.
${\bf Lesson} \ {\bf 8}: \ {\bf Accumulators} \ in \ recursion, \ mutual \ recursion, \ nested \ definitions.$
Figure 2: A Gentle Introduction to ML (course structure).

The subjects in the first evaluation were 40 postgraduates following a one-year Software Technology course. $C^{Y}NTHIA$ was introduced in the second week of the course. The students were told that $C^{Y}NTHIA$ was the result of a research project and that they could use it as much or as little as they wished. $C^{Y}NTHIA$ was only mentioned in passing in lectures.

The subjects in the second evaluation were 29 students in year 4 or 5 of an under-

graduate course in Computer Science. $C^{Y}NTHIA$ was introduced as one of the main teaching tools in this course. The students were not told that $C^{Y}NTHIA$ was part of a research project. $C^{Y}NTHIA$ was introduced more fully in the lectures, although details of editing commands and functionality were only taught in the tutorials.

4.2 Informal versus Formal Evaluation

Most of the evaluation of $C^{Y}NTHIA$ was informal. The formal approach of splitting the subject group into a control and an experimental group was unwise given the experimental conditions. The reasons for this are as follows:

- Ethical considerations. The students in the subject group in both evaluations were following courses which would directly contribute to their degree mark. Therefore, giving CYNTHIA to only one half of the students would have been unethical. Even if CYNTHIA had no overall effect, psychological factors could affect performance. For example, the control group could feel that they had been unfairly treated since they were denied access to the tool.
- Controlling the experimental setup. Dividing the students fairly into a control and experimental group was impossible. Randomization would mean that in a given tutorial some students would be using C^YNTHIA and some would not. This could lead to the psychological factors mentioned above. The first course had two separate tutorial groups but we could not simply use one of these as the control as timetabling factors meant that the abilities of the two groups were vastly different.
- Controlling the running of the experiment. Even given two groups of equal ability, leakage between these groups would be a major complication. Since students are all following the same course, they would communicate with each other between tutorial sessions, and there was no way to stop the control group using CYNTHIA outside the tutorials.
- Interdependency of C^YNTHIA's features. The different aspects of C^YNTHIA tend to interact with each other. Therefore, it would have been difficult to isolate any effects of an experiment, even given perfect conditions.

For these reasons, the following results do not include any detailed statistical analysis. We do include comparisons of error counts but the results of these should be interpreted with care. Note that the crossover experiment (measure II) is some kind of control/experimental group design but the design was used so that individual students could be compared both using and not using C^YNTHIA . The intention was not to make a statistical comparison.

4.3 Answering the Research Questions

Q1. How useful was $C^{Y}NTHIA$ as an editor for ML that guarantees correctness?

 $C^{Y}NTHIA$ was designed as an editor that includes sophisticated correctness-checking techniques. This question asks whether it is useful to have a such an editing environment.

Q1.(a) How does the quantity of errors made using $C^{Y}NTHIA$ compare to text editors?

It was found that the number of errors made by $C^{Y}NTHIA$ -users is less than that of TEA-users.

One way to compare the number of errors made is to look at logs of $C^{Y}NTHIA$, TEA-interaction and make a count of the errors. For SG1, a count was made of errors made during the crossover experiment (measure II). Recall that the crossover experiment involved students using both the $C^{Y}NTHIA$ and the TEA approach, and took place over one hour of intense programming. SG2 errors were counted over 8 weeks during which $C^{Y}NTHIA$ was used regularly by a large number of students for two hours each week (although less intensely). A classification of all errors made was developed, inspiration being drawn from [Aitken, 1996]. Figure 3 gives this classification.

I briefly explain the motivation behind this classification. A full list of each error in each class is given in [Whittle, 1999].

- Algorithmic errors suggest a major algorithmic flaw in the program, such as giving the wrong condition in a conditional statement. These errors arise when the user has misunderstood the problem or is unable to design a solution. *CYNTHIA* was not primarily designed to help with this kind of error (although analogy may provide some help), so I do not include a count of these (they were roughly the same).
- Semantic errors are split into four categories. Local semantic errors arise in response to a misunderstanding of part of ML's semantics such as trying to define the type int string or overloading a variable. Global semantic errors are where



Figure 3: Classification of Programming Errors.

the error is dependent on some other part of the definition — for example, the use of an unbound variable or undefined function. Although type errors could be seen as global semantic errors, they are given a separate category for emphasis. The same is true of pattern errors (i.e. patterns are overlapping or non-exhaustive). Although a program with pattern errors will successfully compile, they are included because they can be a source of run-time errors and because CYNTHIAwas designed to forbid them.

- Syntax errors include clerical errors where, for example, the student clearly mistyped a name or missed off a bracket. General Syntax errors are slightly more serious and suggest a cause more than mere carelessness : examples are using a syntax that ML does not support such as return 0, or using the wrong syntax for a conditional statement.
- Usability errors are when the student could not work the system properly. I make no comment whether it is the student's or the system's fault. An example is not being able to find the right editing command in *CYNTHIA* or entering the wrong parameters in a dialog box. Usability errors also include judgement errors: where the system feedback was misunderstood by the student causing him to make an incorrect change to the program. In *CYNTHIA*, this could happen if, for instance, a syntax error is given in a dialog box and the student changes

the wrong part of the entry. In TEA, the user might change a clause in the definition which was perfectly correct because he did not realise which clause the error appeared in.

A brief note is needed on how the errors were counted. The errors were counted manually from the logs obtained during the experiment. It has to be decided at which point in the logs errors will be counted. In SML-NJ, each time the student typed a semicolon to evaluate a program attempt, any errors present were noted. In C^YNTHIA , the program was checked after each editing command was applied, and any errors in the program were counted. In some cases, multiple commands may be required to get the program into a "consistent" state. For instance, if the user wishes to change a base case output to nil rather than 0, he must do two things: make the actual change, which results in a type error, and then change the result type in the declaration which will eliminate the type error. "Intermediate" errors like this were not counted. Some commands require text to be entered into a dialog box. Any errors present in such textual input were also counted.

The results of each evaluation are presented separately.

Evaluation 1

Table 2 gives the error count for the SG1 crossover experiment for both $C^{Y}NTHIA$ and TEA users. The total number of edits row represents the number of editing commands applied. This figure cannot be applied to TEA users. The difference in the total error

	$C^{Y}NTHIA$	TEA
Local Semantic	20	33
Global Semantic	14	21
Patterns	0	6
Type Errors	20	36
General Syntax	0	8
Clerical	9	40
Incorrect Usage	53	0
Judgement	50	53
Total	166	197
Total no. Edits	473	n/a

Table 2: SG1 Errors.

count is not as high as expected. However, the kinds of errors committed are different for $C^Y NTHIA$ and TEA users.

Syntax errors were almost eliminated when using $C^{Y}NTHIA$. In particular, the number of clerical errors has reduced by 78% for SG1.

The number of **semantic errors** was also less for C^YNTHIA -users. Note particularly that the number of type errors made using C^YNTHIA was just over half that for non- C^YNTHIA users. This is as expected. The raw figures do not tell us how easy it was to correct type errors under each system. However, anecdotal evidence suggests it was much easier using C^YNTHIA – see question 1.(b).

Both the number of local and global semantic errors are fewer with C^YNTHIA than without. Again, this is as expected. Editing commands introduce semantically valid expressions so there is less scope for making errors. We would also expect these errors to be located more quickly (although the figures cannot tell us this). Local semantic errors can only occur in freely typed text — and C^YNTHIA traps these as soon as they are made. Global semantic errors may make it through to the main program text but C^YNTHIA will highlight them making them easier to correct.

For SG1, $C^{Y}NTHIA$ seems to score pretty badly on **usability errors**. In particular, 53 Incorrect Usage errors were introduced that obviously could not occur when using TEA (because they are $C^{Y}NTHIA$ -specific errors). This is a disappointingly large figure. It suggests first that $C^{Y}NTHIA$'s interface is difficult to use and second that students do not read documentation – for most of the errors they committed could have been avoided if they had read the documentation.

Surprisingly, there is no real difference in the numbers of judgement errors. Anecdotal evidence suggests that students using TEA spend much more time trying to locate errors than $C^{Y}NTHIA$ users. The judgement errors were meant to measure this sort of thing, but the results do not back up the informal observations. Table 3 gives a more fine-grained analysis of the judgement errors. See Appendix C for definitions of J1-J5.

	J1	J2	J3	J4	J5	Total
CYNTHIA	8	0	35	4	3	50
TEA	0	53	0	0	0	53

Table 3: Judgement Errors made during Crossover Test.

All of the SML-NJ judgement errors fall into the same category -J2. J2 states that the student has misunderstood an error message and has changed the wrong part of the program in response to this message. J2 errors are fairly serious. The fact that all SML-NJ judgement errors are of type J2 shows that the SML-NJ error messages are cryptic. These kinds of errors do not occur with $C^{Y}NTHIA$ because the nature of the editor tells you much more clearly where the error is. However, there are a large number of J3 errors for $C^{Y}NTHIA$ users. J3 errors occur when the user types in a parameter to an editing command in a dialog box but the parameter given was of an incorrect form. Detailed inspection of the logs showed that 95% of the J3 errors occurred when adding a conditional statement. $C^{Y}NTHIA$ expects the user to type in the condition but often students typed in the entire expression — i.e. if h=x then delete x t else h::delete x t instead of just h=x. The real problem here is that the functionality of the command has not been adequately explained. Given that it is stated precisely in the documentation, our expectation was that by integrating $C^{Y}NTHIA$ more closely into the teaching course, these kinds of errors could be eradicated easily.

Evaluation 2

Evaluation 1 showed that there were generally fewer errors when using $C^Y NTHIA$ but that there were problems with the usability of the system. We expected that most of the judgement errors and many of the incorrect usage errors could be eradicated by making changes to the system and by encouraging students to read documentation. To test out this claim, some changes were made to $C^Y NTHIA$ before the second evaluation. These were:

- **Documentation**. The documentation of *CYNTHIA* was more closely integrated into the course notes each time a new concept was introduced, the corresponding editing command was introduced also and the student was taken through a couple of examples which specifically used *CYNTHIA*.
- **Dialog Boxes**. The main change to the actual interface was to give keywords in the dialog boxes that provided strong hints as to what input was required. Hence, when adding a conditional statement, the dialog box would prompt the user with

if	then	else

rather	than	inst	а	hov
raunci	unan	Juou	a	DUA

A further change in Evaluation 2 is that $C^{Y}NTHIA$ was introduced as one of the main teaching tools. This was meant to give us an opportunity to assess $C^{Y}NTHIA$'s usefulness in a teaching environment. This did mean, however, that for SG2 there was no control group.

Table 4 gives a tentative comparison of SG1 and SG2 errors. SG1 errors were counted during a timed session, whereas SG2 errors were counted throughout the course. Hence, the number of edits / errors in SG2 is much larger. However, all things being equal, we would expect the relative frequency of each error kind to be the same. The second column of Table 4 tells us whether this is the case. We would like to see by how much the quantity of each kind of error has increased from SG1 to SG2 and compare it to the increase expected. Since there is a roughly seven-fold increase in the total number of edits (from 473 to 3266), we would expect a seven-fold increase in each kind of error. This will not be true, in general, though as students will make fewer errors the more experienced they become. We therefore use the increase in Clerical Errors as a basepoint comparison. This is based on the assumption that the changes in the experimental setup from SG1 to SG2 had no effect on the number of Clerical Errors. The second column expresses how the other kinds of errors have increased compared with the Clerical Errors, or, put in another way, it shows the percentage of the expected errors that actually occurred.

	No. errors	% increase relative to Clerical
Local Semantic	39	55
Global Semantic	41	82
Patterns	0	0
Type Errors	59	83
General Syntax	0	0
Clerical	32	100
Incorrect Usage	74	39
Judgement	27	15
Total	323	
Total no. Edits	3266	

Table 4: SG2 Errors.

Most error categories increased by much less than expected. This is probably due in part to the improved documentation / integration. Note particularly that Judgement Errors actually reduced in absolute terms. This backs up our hypothesis that C^YNTHIA 's poor showing regarding Judgement Errors in Evaluation 1 was due to the lack of reading of documentation, not due to any inherent major fault with C^YNTHIA . Hence, the saving in other kinds of errors when using C^YNTHIA is not offset by a gain in usability errors.

Q1.(b) How does $C^{Y}NTHIA$'s error feedback compare to the compiler's?

The previous question showed that the quantity of errors made using $C^{Y}NTHIA$ is generally less than without $C^{Y}NTHIA$. This question concerns not the amount of errors made, but given that an error has occurred, how easily could the students identify it and correct it? It was found that errors are generally easier to correct in $C^{Y}NTHIA$ than TEA.

The evidence for this is at the qualitative level. Evidence from videoing, observation and communication with students seems to suggest very positively that CYNTHIAdoes better than TEA. First, the kinds of errors that students make are different when using CYNTHIA. Trival errors (e.g. clerical errors) have generally been filtered out so students are less infuriated and so try to work out the problem rather than hacking at their code. In stark contrast to users of the SML-NJ compiler, we noticed students paying attention to error feedback and trying to work through the problem. They did not always succeed, of course, but undoubtedly learnt something along the way. Another point is that it is much easier in CYNTHIA to distinguish what kind of error is occurring. This is because the error feedback is different for different categories of errors. One of the problems with ML syntax is that its succinctness means that syntax errors can often have knock-on effects meaning that they show up as type errors during compilation. This happens less in CYNTHIA. The divide between errors is more clear-cut. Type errors are always shown by pink highlighting. Global semantic errors are shown by green highlighting, and syntax errors can only occur in dialog boxes.

Second, students are editing smaller chunks of program at a single time and hence the range over which the error could have occurred is far less. With TEA, students write an entire function before attempting compilation. With $C^{Y}NTHIA$, however, as each sub-expression is entered into a dialog box, the text is checked for errors immediately. This means that the student need only look over very small chunks at a time and need worry less about dependencies with other code fragments. In addition, the user knows that some parts of the program are guaranteed correct — for example, any patterns will have been built up using MAKE PATTERN and therefore the patterns must be welldefined. Since some code is generated automatically, there is no reason for the user to suspect an error there. Hence, $C^{Y}NTHIA$ allows the user to narrow his field of vision when looking for errors.

Although, in general, $C^{Y}NTHIA$ users seem to find it much easier to locate the

source of type errors, there are a few situations where C^YNTHIA can be misleading. This happens when type inference would succeed on a definition but type checking in C^YNTHIA fails because of extra restrictions placed (unwittingly) on the definition by the type declaration. An example is where the compiler would automatically unify two polymorphic variables but C^YNTHIA does not. Consider the example:

The user gets a type error in C^YNTHIA at the indicated point until he changes 'b to 'a. It may be possible for C^YNTHIA to automatically update such type declarations (or at least suggest updates).

Q1.(c) How does the user's productivity rate using $C^{Y}NTHIA$ compare to text editors?

We answer this question with specific reference to the video experiment (measure III). The four subjects that took part in the videoing each worked through a maximum of three examples (see Appendix A) of increasing difficulty. Some of these were attempted using $C^{Y}NTHIA$ and some without $C^{Y}NTHIA$. Table 5 gives the number of examples and timings (in minutes and seconds) for each student. C denotes that the example was attempted using $C^{Y}NTHIA$. S denotes that the student did not use $C^{Y}NTHIA$. Students were given help if they asked for it. This was comparable and minimal except for student 3 who was given a substantial amount. Student 4 failed to finish addlist. Students 2 and 3 did not have time to attempt it.

$\operatorname{Student}$	leading 0s	maxlist	addlist
1	11:31 (C)	20:39~(S)	17:70 (C)
2	27:00 (S)	20:10 (C)	_
3	16:30 (C)	12:20~(S)	_
4	16:10~(S)	10:30~(C)	18:46 (S)

Table 5: Student Performance on Three Examples.

The general level of ability of the students seemed to be in the order: student 1 (best), student 4, student 2, student 3. maxlist is slightly harder than leading0s. addlist is more difficult again because it involves multiple recursion. Students 1, 2 and 4 seem to have performed better with $C^{Y}NTHIA$. On the first two tasks, all of these students took less time when using $C^{Y}NTHIA$ – on average, 35% less. Note also that student 1 actually took less time on addlist than on the easier problem maxlist.

One reason for this could be that the student used C^YNTHIA for addlist (note, for instance, that student 4 did not use C^YNTHIA for addlist). The same phenomenon occurs when comparing student 2 and 4's performances on maxlist and leading0s.

A 35% gain would seem to agree with informal observations. For this level of task difficulty, the student often starts with a good idea of the required program behaviour and can describe this behaviour fairly accurately. Most time is taken up trying to implement this algorithm – for example, correcting syntax and type errors and perhaps adjusting their algorithm slightly. Hence, 35% represents the gain in implementation time achieved by C^YNTHIA . Student 3 was unable to describe the algorithm in abstract terms and hence took more time in non-implementational work. This explains why the 35% decrease is not experienced in this case.

Clearly, the number of students involved in the video experiment is small. Further investigation is needed to back up these results.

Q2. How does programming by analogy compare to writing a program from scratch?

Q2.(a) Are the editing commands well-designed?

This question concerns the transformation of a source program using $C^{Y}NTHIA$'s collection of editing commands. Specifically, was the structure of the commands wellunderstood, was their function clear, etc.? Green et al [Green & Petre, 1996] introduce the notion of 'cognitive dimension', a broad-brush evaluation technique for interactive devices and non-interactive notations. Green describes thirteen high-level criteria for discussing the design of a system. The idea is that they will form a common point of discourse for evaluating interactive systems. Although we will not mention all of the dimensions here, they serve as a useful framework for discussing the design of the editing commands and for evaluating how easily the editing commands can be learnt and applied. The following considers the most relevant of these dimensions and evaluates the set of editing commands on each. More dimensions can be found in [Whittle, 1999].

Abstraction Gradient

Each editing command is essentially an abstraction, grouping together common sequences of editing operations. But are they at the right level of abstraction? As Green says, "learning to think in abstract terms is a high educational achievement". The natural question to ask therefore is if the students using $C^Y NTHIA$ could understand the editing commands. Does the abstract nature of the editing commands benefit them in the long run?

The main result we found was that C^YNTHIA 's editing commands tend to correspond to functional programming concepts. For example, ADD RECURSIVE CALL emphasises the role of termination checking and CHANGE TYPE emphasises the use of types. In particular, C^YNTHIA discourages a procedural style of coding.

The original aim when designing the editing commands was to make the set as small as possible whilst keeping the meaning of the commands transparent to a new user. As far as the former goes, the goal was certainly achieved — with as few as 11 commands (appendix D), a wide variety of programs can be produced (much wider than comparable systems such as [Brna & Good, 1996, Bundy *et al.*, 1991]). However, the high number of Incorrect Usage errors in Tables 2 and 4 show that the commands caused some confusion. In some cases, the abstractness of the editing commands seemed difficult to learn precisely because they correspond to FP concepts. There is a chicken and egg situation here — learning the commands is easier if functional programming is understood, but use of the commands can help the understanding of functional concepts. MAKE PATTERN is an example. Consider the following code:

fun combine x nil = nil
| combine x (h::t) = ...

To pattern match on the indicated x, students would try to rename x to nil. This would be disallowed by *CYNTHIA*. The problem here is that students do not think in terms of making a pattern or such like, but in terms of adding another line of code. One way to overcome this would be to recast the commands in terms of very obvious code-writing operations, such as ADD LINE OF CODE. However, MAKE PATTERN does more than just adding a line of code. There is a specific reason why the code is being added and MAKE PATTERN cannot be used to add just any line of code. It is unclear whether it is better to design the commands in terms of functional programming concepts or not. The ideal solution is probably to incorporate the editing commands into the teaching material.

Consistency

This dimension asks: when some of the language has been learnt, how much of the rest can be inferred. In this context, the question concerns the consistency of the operation of the editing commands. It is instructive to look more closely at the Incorrect Usage errors committed by SG2: see Table 6 for a breakdown (the categories are explained in Appendix C).

	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10	U11
Errors	0	4	3	0	1	1	11	3	2	2	4
	U12	U13	U14	U15	U16	U17	U18	U19			
Errors	12	1	11	4	11	1	2	1			

Table 6: Incorrect Usage Errors.

Four errors, U12, U14, U16 and U7, contribute over 60% of the total count. U12 states that the user tried to apply MAKE PATTERN to an integer variable. This is disallowed in C^YNTHIA because MAKE PATTERN can only be applied to types with a finite number of constructors. Since each integer can be viewed as a constructor, the finiteness restriction does not hold. U14 is when an application of CHANGE TYPE would require that a polymorphic variable be split into patterns. Clearly, this cannot be done. U16 is where the user attempted to use CHANGE TERM instead of ADD RECURSIVE CALL to add a new recursive call. U7 concerns the use of RENAME rather than CHANGE TERM to make a local change.

Three of these errors, U12, U14 and U16, are in response to events that the user has every right to assume should be possible. This shows a slight lack of consistency in the set of editing commands. Students have attempted operations that should be possible based on their knowledge so far of the commands. This should not be frowned upon too much, however, for each of the errors could be fixed relatively easily in a future version of $C^{Y}NTHIA$. Hence, the evaluation has pinpointed areas where $C^{Y}NTHIA$ could be improved.

Hidden Dependencies

A hidden dependency is a relationship between two components such that one is dependent on the other but that dependency is not fully visible. An example would be a spreadsheet — a formula in a cell tells which other cells it takes its value from, but does not tell which other cells take their value from it.

Generally, there are very few hidden dependencies in $C^{Y}NTHIA$ compared to TEA.

For instance, type inference introduces a very significant hidden dependency — type error messages often point to a point of the program far removed from the actual source of the error. In contrast, in $C^{Y}NTHIA$, the pink highlighting of type errors tends to be much closer to the source of the problem. In fact, [Whittle, 1999] pages 250-251, makes a brief comparison of type highlighting in $C^{Y}NTHIA$ and MLWorks and finds that $C^{Y}NTHIA$'s highlighting is closer to the source.

Sometimes a compiler will accept a function which is not accepted by $C^{Y}NTHIA$. Consider the *flatten* function:

```
fun flatten nil = nil
    flatten (h::t) = h @ flatten t;
```

This is accepted by ML compilers and 'a list list -> 'a list is inferred as the type for *flatten*. Unfortunately, certain interactions with CYNTHIA can lead to the above definition but with an incorrect type displayed. Students would edit an old function, such as *doublist* which has type int list -> int list. They would correctly edit *doublist* into the *flatten* definition above. However, they would not edit the type declaration and so CYNTHIA would give a type error because the type declaration means that h is an integer and so cannot be appended onto flatten t.

This kind of situation was a cause of great confusion for students who encountered it. The students had an implicit assumption that the type declaration was correct and hence would scrutinise the program itself for errors. This implicit assumption is what makes type errors easier to locate in $C^{Y}NTHIA$. The situation given here, however, is an example where $C^{Y}NTHIA$ introduces an error that ordinarily would not occur.

Premature commitment

This dimension concerns the extent to which the user is forced to make a decision before the information is available. Hence, premature commitment is generally a bad thing. Type inference reduces the need for premature commitment since decisions about the typing of an expression can be delayed. At first glance, it seems that the same cannot be said of C^YNTHIA since a type declaration must be given. However, it is extremely easy to change the type declaration using CHANGE TYPE so premature commitment is not really needed.

Another context in which premature commitment manifests itself is the degree to

which the order of application of the editing commands matters. One of the main criticisms of the *recursion editor* [Bundy *et al.*, 1991], which is also a transformation-based editor, is that the order of commands is critical to success and so the user must think about the order before delving into the programming task.

This is not true of CYNTHIA. Design decisions taken in the early stages can easily be corrected later using a short sequence of editing commands. Suppose the user is writing a function, app, to append two lists together and begins by splitting the second argument:

It is at this point that the user realises the first argument should have been split instead. In C^YNTHIA , it is easy to revise this decision. The user applies REMOVE PATTERN to give:

fun app 1 12 = 1;

and then MAKE PATTERN on 1, giving:

fun app nil 12 = nil
| app (x::xs) 12 = x::xs;

Note that the user must also apply CHANGE TERM in the second clause to re-introduce the program fragment that was lost during the application of REMOVE PATTERN¹⁰. Hence, although no premature commitment is required, the fix is non-optimal. A better solution might be to provide a specialised command for transferring the pattern definition to the first argument.

Progressive evaluation

Progressive evaluation means that programs can be evaluated by the user at frequent intervals during their development, not just once the program is completely finished. $C^{Y}NTHIA$ improves on TEA in a significant way here. Although any program must be finished before it is executed (for it still must be accepted by the compiler), the user

¹⁰One variation would be to allow the user to specify which clause is kept so that $x::app \ l \ xs$ need not be re-typed.

gets constant feedback about semantic errors during the programming process. This is achieved by the use of the highlighting mechanism for pointing out type errors etc. The key point is that $C^Y NTHIA$'s feedback merely notifies the user of a problem, it does not enforce them to change it immediately. Hence, the user retains the freedom to experiment but the existence of any errors is always in the back of his mind.

Viscosity

Viscosity means resistance to local change. $C^{Y}NTHIA$ is sometimes less viscous than TEA and sometimes more viscous. Operations can be carried out in $C^{Y}NTHIA$ that would have to be done laboriously by hand in TEA. An example is changing the type of a function and seeing the effects of this propagated throughout the program (having the pattern definition changed automatically). In this case, productivity is increased in $C^{Y}NTHIA$ because a "conceptual" change can be achieved in one edit. Such conceptual changes require many more keystrokes in a text editor, though. On the other hand, it can sometimes be easier to use a standard text editor to affect a change. Recall the *app* example above. Without a specialised editing command to transfer the pattern definition from the second to the first argument, we lose some program text which has to be re-entered.

Q2.(b) How easy is it to choose a starting (source) example?

Programming by analogy introduces an additional overhead for the user, namely, the decision about which source example should be chosen. Some work has been done in the area of software re-use to provide sophisticated library systems that allow the user to quickly select the best example [Weber, 1996, Runciman & Toyn, 1991]. In C^YNTHIA 's case, however, no library search functions are provided. This is for two reasons. First, since C^YNTHIA is currently being used in a novice environment it was considered unnecessary. The students would be dealing with relatively easy examples and probably not building up too large a database of source functions. Second, the choice of a source is not as critical as in other systems because the editing commands are very flexible and so it is easy to recover from a sub-optimal choice. The evaluation phase gave an opportunity to test out these decisions.

One way of answering the question is to consider how students decided upon a source example. There seems to be three main ways — recency, familiarity and closeness.

Most students pick the function they have used most recently¹¹. In many cases, this is a perfectly reasonable approach. For instance, when working on the on-line tutorial, examples within a tutorial tend to be similar (and get increasingly more complex) so that such an ordering is very natural. It is not quite so natural in a real situation. For example, in the crossover experiment (measure II), a student wrote *combine* (see Appendix B), which involves a complex recursion scheme, then used *combine* as the starting point for a primitive list recursion example. Most students do modify their strategy in this sort of situation, however.

Another very common way of choosing a source is to choose a familiar example. In the version of C^YNTHIA that students were given, 6 examples were pre-defined. Two of these were primitive list recursion examples, *sum* and *doublist*. These were familiar examples as they were used in the tutorial material on list recursion. Students were quick to pick one of these as a starting point rather than something they had defined themselves, even if their definitions were closer to what they needed. This is because the students are more familiar with the built-in functions and so need not waste time understanding them. It should be said, however, that because the nature of functional programming means that most functions are relatively small, understanding the source example is rarely a time-consuming task. Also, students don't start to think about the task in hand until they have something on the canvas. Only once they have a function in the edit area do they start to think about the current task. By bringing something up in the edit area straight away, they feel as though they are part way to their goal.

The more able students do think more deeply about which source example to choose. This was brought out during the videoing. Student 1 said: "I'm looking for a function with two lines in it." when trying *leading* θ s. Student 4 said, whilst looking at the definitions available: "So, I want to get something closest to *maxlist*. I don't know what half of these are unfortunately." He then selected a couple and decided they were not close enough until he eventually chose *sum*. The two main measures of similarity used in these circumstances were: the type of the variable being recursed upon, and how many patterns (or lines of code) were in the function. They did not seem concerned with the result type of the function or with the type of non-recursive input types. Student 1 looked for a function with 2 lines of code as source for *leading*1s. He chose *doublist* even though this has a result type of **int list** not the required **int**.

¹¹This backs up the claim made in [Weber, 1996].

5 Conclusion

It is our belief that the design of editors for novice functional programmers requires special attention. Novice students generally find functional concepts like recursion and higher-order functions difficult to learn. Debugging is in some sense more tricky in a functional language because much runtime debugging is replaced by compile time resolution of type conflicts. These conflicts must be resolved before the program can be executed and so program traces cannot be used as a debugging guide. In addition, the complex nature of type inference over polymorphic types means that type errors can be awkward to locate.

As an antidote to these effects, this paper has presented $C^{Y}NTHIA$. In $C^{Y}NTHIA$, the program is checked for errors incrementally as it is written. Due to its underlying proof framework and the insistence on a type declaration, $C^{Y}NTHIA$ can highlight the source of type errors more accurately than compilers. On a presentation level, the highlighting of errors (in the form of a change of colour of a program expression) is a non-intrusive form of feedback which the user must at all times be aware of but may choose to ignore temporarily. On the other hand, compilers can fill entire screens with cryptic type error messages which both must be dealt with immediately and also have a negative impact on the morale of the student.

At a deeper level, C^{YNTHIA} 's abstract editing commands reduce the number of errors which students make. Students construct programs by modifying existing, understood code fragments. The majority of commands are correct-by-construction in the sense that they cannot introduce errors. Some commands will allow the user to introduce errors but these are flagged once again in a non-intrusive manner. Unlike previous approaches to transformation-based programming, the emphasis in C^{YNTHIA} is on semantic rather than syntactic transformations. The commands also tend to encapsulate key functional programming concepts, such as the addition of a recursive call or the change of a set of patterns resulting from a change of type, and as such, they encourage the student to program functionally rather than procedurally. Occasionally, the functionality of a command can be misunderstood precisely because the student does not understand the underlying concept fully. Whilst this can hamper usability of the system, it forces the student to get help to understand first the concept and then the command.

 $C^{Y}NTHIA$'s commands form a small set that allow a wide variety of programs to be constructed in an order-independent fashion. This is in contrast to traditional template and schema methods of programming where either there are far too many schemas so that choosing the appropriate one is an arduous task, or only a very restricted subset of the language can be used. We believe that the modular nature of functional programming is what allows the balance to be achieved in $C^{Y}NTHIA$.

There are two main results of this paper. First, the use of transformation-based programming as embodied in $C^{Y}NTHIA$ can reduce the number of errors that novice programmers make. It was shown that syntax errors can be reduced the most dramatically but the real gain is in the reduction of type errors as these errors are most difficult to locate and correct. Infinite loops are also eradicated in $C^{Y}NTHIA$. Further study is needed to investigate how much of a gain it is to outlaw non-terminating programs. Students tend to introduce fewer termination errors than, say, type errors but termination errors can be more serious.

The second main result is that the transformation approach can increase the productivity of novice programmers. Commands like CHANGE TYPE propagate the effects of small changes throughout a program, which may result in significant changes that would otherwise have to be laboriously typed in by hand. The video experiment showed that students took less time to complete harder problems in C^YNTHIA than they did to complete easier problems with the traditional approach. Admittedly, the video experiment involved only four students, but the results were observed more generally during tutorial sessions with the other students.

One question that has not been fully addressed in this paper is the worth of programming by analogy as a learning aid. Analogy is a powerful technique in learning concepts such as recursion [Pirolli & Anderson, 1985] but can also lead to problems if students make incorrect analogies, resulting in incorrect solutions [Escott & McCalla, 1988]. We have shown that the use of an existing program as an analogy to a desired one is one important way of overcoming the "blank page" problem. We believe also that analogies between functional programs are much easier to make than between procedural programs and hence the issue of incorrect analogies is less likely to arise.

We see that $C^{Y}NTHIA$ -like transformations could be useful in a much wider functional programming context, for example, in maintaining existing programs. Software often needs to be updated by making very small changes. A tool like $C^{Y}NTHIA$ could make these changes but guarantee that no errors are introduced as a by-product. Clearly, this would require extending $C^{Y}NTHIA$ to the full ML language. The two major problems with doing this would be to extend termination checking and to deal with imperative features of ML. As for the former, we envisage integrating a collection of specialised termination checkers into $C^{Y}NTHIA$ — for example, a checker based on recursive path orderings [Dershowitz, 1985] could deal with mutual recursion — but ultimately, the user may wish to deliberately write a non-terminating program and so the facility would be provided to switch off termination-checking. The underlying proof framework of $C^{Y}NTHIA$ could be adapted to deal with imperative features. In fact, work is underway to produce a $C^{Y}NTHIA$ -like editor for the procedural language Java [Blewitt, 1998]. A further requirement of a fully blown $C^{Y}NTHIA$ system would be the inclusion of type inference. We show in [Whittle, 1999] how this could be incorporated without losing anything (and indeed there are gains to be made, as $C^{Y}NTHIA$ provides a nice framework for *partial* type inference, whereby a type can be inferred based only on the information given so far).

Another interesting avenue would be to extend the correctness guarantees that $C^{Y}NTHIA$ offers. Although a challenging task, we believe that $C^{Y}NTHIA$'s underlying framework could allow the user to make assertions about the behaviour of their functions and then to have these assertions proved. Clearly, the greater the correctness guarantees, the more difficult the theorem proving required. We expect that a series of specialised theorem proving tactics could be formed that would enable much of the theorem proving to be automated, given that the assertions are in a suitably restricted form.

Acknowledgments The first author was supported by an EPSRC studentship and computing facilities were provided by EPSRC grant GR/L/11724. The authors would like to thank Alan Bundy, Helen Lowe and Richard Boulton for discussions throughout the duration of this research.

References

[Aikins, 1980]	Aikins, J. (1980). Prototypes and production rules: a knowl- edge representation for computer consultations. Unpublished Ph.D. thesis, Stanford University, Available as computer sci- ence report number STAN-CSD-80-814.
[Aitken, 1996]	Aitken, S. (June 1996). An analysis of errors in interactive proof attempts. Technical report, Dept. of Computer Science, Glasgow University.
[Anderson et al., 1988]	Anderson, J. R., Pirolli, P. and Farrel, R. (1988). Learning to program recursive functions. In Chi, M.T.H, Glaser, R. and

	Farr, M.J., (eds.), <i>The Nature of Expertise</i> , pages 153–183, Hillsdale, NJ. L. Erlbaum.
[Bental, 1995]	Bental, D. (1995). Why doesn't my program work? : re- quirements for automated analysis of novices' computer pro- grams. In <i>Proceedings of the Workshop on Automated Un-</i> <i>derstanding of Novice Programs, World Conference on AI</i> <i>and Education (AIED)</i> , Washington DC, USA. Available at http://www.cs.mdx.ac.uk /staffpages /DBental /aiedpa- per.html.
[Bhuiyan et al., 1994]	Bhuiyan, S., Greer, J. and McCalla, G. I. (1994). Supporting the learning of recursive problem solving. <i>Interactive Learning</i> <i>Environments</i> , 4(2):115–139.
[Bird & Wadler, 1988]	Bird, Richard S. and Wadler, Philip. (1988). Introduction to Functional Programming. Prentice-Hall.
[Blewitt, 1998]	Blewitt, A. (September 1998). A Java editor based on proofs- as-programs. Unpublished M.Sc. thesis, Department of Arti- ficial Intelligence, Edinburgh, Scotland.
[Bowles & Brna, 1993]	Bowles, A. and Brna, P. (August 1993). Programming plans and techniques. In Brna, P., Ohlsson, S. and Pain, H., (eds.), <i>Proceedings of the World Conference on Artificial Intelligence</i> <i>in Education</i> , pages 378–385. AIED.
[Brna & Good, 1996]	Brna, P. and Good, J. (1996). Searching for examples: An evaluation of an intermediate description language for a techniques editor. In Vanneste, P., Bertels, K., de Decker, B. and Jaques, J-M., (eds.), <i>Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group</i> , pages 139–152.
[Bundy et al., 1991]	Bundy, A., Grosse, G. and Brna, P. (1991). A recursive techniques editor for Prolog. <i>Instructional Science</i> , 20:135–172.
[Dershowitz, 1985]	Dershowitz, N. (1985). Synthetic programming. Artificial Intelligence, 25:323-373.
[Duggan & Bent, 1996]	Duggan, D. and Bent, F. (1996). Explaining type inference. Science of Computer Programming, 27:37-83.
[Escott & McCalla, 1988]	Escott, J. A. and McCalla, G. I. (1988). Problem solv- ing by analogy: A source of errors in novice LISP program- ming. In <i>Intelligent Tutoring Systems</i> , pages 312–319, Mon- treal, Canada.
[Gegg-Harrison, 1991]	Gegg-Harrison, T.S. (1991). Learning Prolog in a schema- based environment. Instructional Science, 20:173-192.
$[\mathrm{GIML},\ 1998]$	Department of Computing Studies, Napier University, Craiglockhart campus, 219 Colinton Road, EH14 1DJ. (1998). A Gentle Introduction to ML, http://www.dcs.napier.ac.uk/course-notes/sml/manual.html.
[Green & Petre, 1996]	Green, T.R.G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. <i>Journal of Visual Languages and Computing</i> , 7:131–174.
[Hansen, 1971]	Hansen, W.J. (1971). Graphic editing of structured text. In Parslow, R.D. and Green, R.E., (eds.), <i>Advanced Computer</i> <i>Graphics</i> . Plenum Press.
$[\operatorname{Har} 1996]$	(r1996). MLWorks. Harlequin, Inc.

[Howard, 1980]	Howard, W. A. (1980). The formulae-as-types notion of con- struction. In Seldin, J. P. and Hindley, J. R., (eds.), To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479-490. Academic Press.
[Jun & Michaelson, 1998]	Jun, Y. and Michaelson, G. (March 1998). A visualisation of polymorphic type checking. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt Univer- sity.
[Kirschenbaum et al., 1989]	Kirschenbaum, M., Lakhotia, A. and Sterling, L.S. (1989). Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University.
[Leroy, 1995]	Leroy, X. (1995). The CAML Light system (release 0.7). Projet cristal, INRIA Sophia Antipolis.
[McAllester & Arkoudas, 1996]	McAllester, David and Arkoudas, Kostas. (July 1996). Walther recursion. In McRobbie, M. A. and Slaney, J. K., (eds.), 13th International Conference on Automated Deduc- tion (CADE13), pages 643-657. Springer Verlag LNAI 1104.
$[{\rm Michaelson},\ 1995]$	Michaelson, G. (November 1995). <i>Elementary Standard ML</i> . UCL Press.
[Milner <i>et al.</i> , 1990]	Milner, R., Tofte, M. and Harper, R. (1990). The Definition of Standard ML. MIT Press.
[NJS1996]	(S1996). Standard ML of New Jersey, version 0.93. Tech.rep, AT&T Bell Laboratories, Available at http://www.dcs.napier.ac.uk/course-notes/sml/BASE.ps.
[Paulson, 1991]	Paulson, L.C. (1991). <i>ML for the Working Programmer</i> . Cambridge University Press.
[Pirolli & Anderson, 1985]	Pirolli, P. L. and Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming. <i>Canadian Journal of Psychology</i> , 39:240–272.
[Rideau & Théry, 1997]	Rideau, L. and Théry, L. (March 1997). An interactive pro- gramming environment for ML. Rapport de Recherche 3139, INRIA Sophia Antipolis.
[Runciman & Toyn, 1991]	Runciman, C. and Toyn, I. (April 1991). Retrieving reusable software components by polymorphic type. <i>Journal of Func-</i> tional Programming, 1(2):191–211.
[Teitelman & Reps, 1984]	Teitelman, T. and Reps, T. (1984). The Cornell program synthesizer: a syntax-directed programming environment. In Barstow, D.R., Shrobe, H.E. and Sandewall, E., (eds.), <i>Inter-</i> <i>active Programming Environments</i> , pages 97–116. McGraw- Hill.
[Teitelman, 1975]	Teitelman, W. (1975). INTERLISP Reference Manual. XE-ROX.
[Ullman, 1994]	Ullman, J. D. (1994). <i>Elements of ML programming</i> . Prentice- Hall International, Englewood Cliffs, N.J.
[Weber, 1996]	Weber, G. (1996). Individual selection of examples in an intelligent learning environment. <i>Journal of AI in Education</i> , 7(1):3-31.
[Whittle, 1999]	Whittle, J.N.D. (1999). The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor. Unpub- lished Ph.D. thesis, Division of Informatics, University of Ed- inburgh.

```
[Whittle et al., 1999] Whittle, J., Bundy, A., Boulton, R. and Lowe, H. (1999).
An ML editor based on proofs-as-programs. In Proceedings
of the 14th Conference on Automated Software Engineering
(ASE99), Cocoa Beach, Florida.
```

Appendix A

```
1) Write a function that takes a list of zeros and ones and returns
the number of consecutive zeros (if any) at the front of the list.
```

Examples: leading0s [0,0,0,1,0] = 3 leading0s [1,0,0,0] = 0

2) Write a function that takes a list of non-negative integers and returns the maximum integer in the list.

Example: maxlist [1,3,2,5,3] = 5

Appendix B

Test X

```
1) Write a function addl which takes 2 arguments: an integer, n, and a list of integers. The function returns a list of pairs. The first component of the pair is the original element. The second component is the element added to n.
```

e.g. addl 5 [1,2,3] = [(1,6), (2,7), (3,8)]

2) Write a function alessnum which tests element by element whether a list of integers is numerically less than another list. alessnum should return true if and only if every element in the first list is less than the corresponding element in the second list.

e.g. alessnum [1,2,3] [2,3,4] = true

alessnum [1,2,3] [2,3,2] = false

3) Write a function that takes a list of integers and produces a new list. The nth element in the new list is the sum of the elements of the old list up to position n.

e.g. ilist [1,2,3,4] = [1,3,6,10]

Test Y

1) Write a function delete which deletes all occurrences of an element from a list.

e.g. delete 3 [1,2,3,3] = [1,2]

2) Write a function combine which takes two lists and returns a new list with corresponding elements combined into a pair.

combine [1,2,3] [4,5,6] = [(1,4), (2,5), (3,6)]

3) Write a function pairlist which takes a list of pairs of integers and returns a list of the alternate components of these pairs.

e.g. pairlist [(1,2),(3,4),(5,6)] = [1,4,5]

Appendix C

J1	Wrong sequence of editing commands because could not get the right command to work
J2	Misunderstood error messages so changed wrong part of program
J3	Error message caused user to incorrectly change part of input in dialog box in $C^{Y}NTHIA$
J4	Wrong structure in definition caused by bad source example choice
J5	Misunderstood pink highlighting so changed expression to another ill-typed one

Table 7: Judgement Errors

U1	Omitted entry in dialog box
U2	Gives pattern, not variable name, to ADD ARGUMENT
U3	Typed entire conditional statement in IF THEN ELSE box
U4	Gave function name for new variable name in ADD ARGUMENT
U5	Used CHANGE TERM instead of IF THEN ELSE
U6	Used CHANGE TYPE to add an argument
U7	Used RENAME instead of CHANGE TERM
U8	Tried to remove an argument that had been split into patterns
U9	Gave top-level type in dialog box for ADD ARGUMENT
U10	Used RENAME instead of MAKE PATTERN
U11	Used ADD ARGUMENT with nil instead of MAKE PATTERN
U12	Tried to apply MAKE PATTERN to integers
U13	Tried to add a variable not in use but used internally
U14	Applied CHANGE TYPE to introduce a split of an integer or polymorphic variable
U15	REMOVE CONSTRUCT instead of REMOVE PATTERN
U16	CHANGE TERM instead of ADD RECURSIVE CALL
U17	Use of case instead of if then else
U18	Only gave arguments, not entire recursive call, as parameter to ADD RECURSIVE CALL
U19	Wrong parameter given to RENAME

 Table 8: Incorrect Usage Errors

Appendix D

Make pattern	Remove pattern
Add argument	Remove argument
Add curried argument	Remove curried argument
Add recursive call	Remove recursive call
Change term	Change type
Rename	