Evaluating Environments for Functional Programming

Jon Whittle[1] and Andrew Cumming[2]

[1] Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland.
[2] Dept of Computer Studies, Napier University, 219 Colinton Road, Edinburgh EH14 1DJ, Scotland.
jonathw@dai.ed.ac.uk

Abstract. This paper presents a novel program editor, C^YNTHIA, for the functional programming language ML. Motivated by the current lack of good programming environments for functional programming, C^YNTHIA incorporates *programming by analogy* — whereby users write programs by applying abstract transformations to existing programs — and sophisticated correctness-checking techniques such as checking for semantic errors and non-termination. C^YNTHIA has been used in two introductory courses in ML and the results of these evaluations are presented here. It was found that students using C^YNTHIA committed fewer errors than students using a compiler / text editor approach. In addition, errors that were made could be corrected more easily.

1 Introduction

This paper investigates what kinds of programming environments could most benefit novice functional programmers, with particular reference to the language ML [Paulson 91, Michaelson 95, Ullman 94]. We claim that students are being held back by the primitiveness of current environments which inadequately relay errors to the user resulting in an arduous edit-compile-edit loop whereby users incorrectly patch their programs. Although there now exist many advanced, integrated environments for traditional languages like C, functional programming (FP) languages have not been given the same attention (although recent environments such as MLWorks [Har96] and CtCaml [Rideau & Théry 97] alleviate this to some extent). We believe that superior environments would free the novice from trivial debugging tasks allowing them to proceed to more difficult problems and hence speeding up the learning process. The kind of environments we have in mind for novices would ultimately be useful for experts too.

The most common way for students to write programs is at a command-line interface or in a text editor followed by compilation of the file. Current compilers for ML (e.g. SML of New Jersey (SML-NJ) and CamlLight) give only crude error information and only deal with complete definitions. There are two ways in which we address the current lack of functionality in FP environments. Firstly, we propose a more incremental approach whereby the users' programs are checked for errors *as* they are written[1] 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.

[1] Writing incremental compilers is hard. Our task is easier as we only consider the non-imperative aspects of ML.

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.

Secondly, we investigate ways in which students can be prevented from making errors in the first place. Traditional syntax-directed editors (e.g. [Hansen 71, Teitelman 75, Teitelman & Reps 84]) 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, real-time programming environment.

The concrete result of this research is the system $C^{Y}NTHIA$, an editor for ML. $C^{Y}NTHIA$ combines advanced correctness-checking with the idea of programming by analogy. $C^{Y}NTHIA$ has been implemented and supports a subset of ML. It has been used extensively in two introductory courses on ML, the results of which are presented in §4. The interface to $C^{Y}NTHIA$ is currently based around a structure editor but future versions could be hybrid, providing both structure and text editing facilities.

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.* 90] which is the most widely used. We do not attempt to describe functional programming here but refer the unfamiliar reader to [Bird & Wadler 88]. The syntax of ML is best illustrated by example. The following program is a piece of ML code to calculate the length of a list:

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

where `nil` denotes the empty list and `::` is the *cons* operator for constructing lists. Note the use of recursion as is common in functional programming. The features of ML relevant to this paper are as follows:

- **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. `length` above is such a definition. Its input is a list of numbers and its output is another list of numbers. For example, if we evaluated `length (1::2::nil)`, the answer returned would be `2`. As ML is functional, it makes extensive use of recursion where procedural languages would use a loop. ML is an impure functional language — i.e. it contains a small number of imperative features included for practical reasons.
- **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 `length` has type `'a list -> int`. `int` is the type of integers. `'a list` is a *polymorphic* list — i.e. the type of elements of the list is unspecified. Polymorphism allows code to be shared between different data structures while retaining the security associated with strong typing.
- **Higher order programming**. This means that ML functions can take other functions as arguments. In this case, the function is a *higher-order* function or *functional*. Although procedural languages such as Fortran may allow functions to be passed

as arguments to another function, few procedural languages allow function values to play a full role in data structures [Paulson 91].

$C^{Y}NTHIA$ is concerned only with a purely functional subset of Core[2] 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 Functional Programming Environments

Current FP environments are generally difficult for novices to use. Students find FP languages difficult because they involve a lot of foreign concepts and because the environments do not sufficiently support the programming process. Students transferring from a procedural background find the going particularly tough. Current environments typically expect the user to edit their programs in a standard text editor and then compile. Any errors found during compilation are relayed to the user but in a cryptic and often unhelpful way. As a result, students tend to ignore many error messages and instead blindly hack away at their code, possibly changing parts of the code which were originally correct. As such, a disproportionate amount of time is spent locating and correcting trivial syntactic or semantic errors — time which could be spent more profitably in the learning of the more difficult concepts.

Strongly-typed languages, such as ML, present particular problems for the novice [Jun & Michaelson 98]. ML compilers can automatically infer the types of definitions using a type inference algorithm. Although this frees the user from writing type declarations, it is difficult to report on inconsistencies discovered during type inference. Compilers often report type errors in the wrong place. Type inference algorithms may not correspond well to how people understand the types of programs. The addition of polymorphic types makes the process of reporting type errors even more complicated. The kind of reasoning required to decipher type messages — unravelling a long chain of deductions and type instantiations — is non-trivial to master. The other problem with type inference is that students are encouraged to ignore types. This may mean that they do not acquire the skills necessary for debugging type error messages.

To get some idea of the extent of students' difficulties with typing, we carried out an informal experiment [Whittle 99] to measure the type errors encountered by a group of 14 ML students using the SML of New Jersey (SML-NJ) compiler. Students were monitored over a two hour tutorial session. Any syntax, semantic or type errors were noted. 193 type errors were found, compared with 114 other semantic errors and 70 syntax errors. The dominance of type errors suggests that improved type feedback facilities would be useful. In questionnaires, students indicated that type errors were more difficult to correct than syntax errors. Most students also said that they received only a slight amount of help from the compiler.

Other problems discovered during the experiment include students' use of pattern matching (for example, the `length` function given earlier was defined for the pattern when the list was `nil` and for when it was `h::t`, i.e. non-empty). ML makes extensive use of pattern matching. 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

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

— for example, if a call of `f nil` is made and there is no case for the `nil` pattern. Observations of students showed that they had a good deal of difficulty in formulating well-defined patterns (i.e. those that exhaustively cover the datatype and do not contain redundant cases). In the case of overlapping patterns, ML compilers resolve the ambiguity by considering patterns in a top-bottom, left-right order. Note, however, that this may not give the result the programmer intended. 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.

ML also involves extensive use of recursion. Recursion is well-known to be a difficult concept to grasp [Anderson *et al.* 88]. Although languages such as C do support recursive definitions, the more usual approach is to use iteration which is often said to be more natural. There are few everyday analogies for recursion. Many approaches have been made to explain recursion to novices in a way which they will understand but none of them have been entirely successful.

One of the most common attempts to support students learning recursion is the template-based approach [Bhuiyan *et al.* 94, Gegg-Harrison 91, Kirschenbaum *et al.* 89, Bowles & Brna 93]. 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.* 91] is one of the better examples of these systems. However, the problem here is that, although a selection 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 his 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.

3 Overview of $C^{Y}NTHIA$

As well as embodying the idea of programming by analogy, $C^{Y}NTHIA$ provides superior error prevention and detection facilities. Firstly, the fact that programs are constructed from pre-defined commands means that fewer errors are made. Secondly, $C^{Y}NTHIA$'s feedback facilities mean that it is easier to correct those errors that are made. $C^{Y}NTHIA$ provides stronger guarantees of correctness than any of the environments mentioned in the previous section or any current ML environments. Specifically, these are:

- **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.

- **Type correctness**. Because $C^Y NTHIA$ is primarily aimed at novices, all function definitions must explicitly be given a type declaration. This is in contrast to the usual approach in ML whereby type inference is used. In the `length` example on p. 2, the user would have to give the declaration `'a list -> int`. We believe that type inference confuses students and that by giving a declaration the student takes more notice of types and hence makes fewer errors. Because type declarations are edited incrementally, the user never needs to write down an entire declaration. Hence, the extra burden of including declarations is not unacceptable.
  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 program into another without passing through intermediate, ill-typed programs. 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 96]. This is because the type inference algorithm breaks down when a type is derived that is inconsistent with a previously derived type. However, expressions early on may have a type derived that is correct in the local context but incorrect when the program is considered as a whole. This is not detected until later on. Hence, type errors may be reported at locations distant from their actual source. $C^Y NTHIA$ 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 $C^Y NTHIA$ can do better is because large fragments of code are guaranteed well-typed *a priori* because they are introduced by editing commands.
- **Static Semantic correctness**[3]. User entry is accepted only if devoid of static semantic errors (e.g. undeclared variables, undeclared functions). 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).
- **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 definition 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 definition is over-defined[4]:

  ```
  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

[3] Type errors are strictly static semantic errors but we make a distinction for purposes of presentation.
[4] `tl` returns the tail of a polymorphic list.

not pick up the expected value. For this reason, C^YNTHIA restricts the user to well-defined functions. In `length` above, `x` would have to be replaced by `(h::t)`. Well-definedness is guaranteed in the program because pattern matching in the proof must be well-defined.
It is impossible to create ill-defined patterns in C^YNTHIA as all patterns are built up incrementally using the editing command MAKE PATTERN which is guaranteed correct.
- **Termination**. All programs in C^YNTHIA are checked to be terminating. Termination checking is an undecidable problem so C^YNTHIA restricts the user to a decidable subset of terminating programs. This is the set of Walther Recursive programs [McAllester & Arkoudas 96]. This set 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 editor. 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.

3.1 Writing Programs in C^YNTHIA

C^YNTHIA is implemented in SICStus Prolog v.3 and Tcl/Tk. Upon startup, the user may select a program from an initial library. The user may edit programs and save them to the library. He[5] will therefore build up his own customised library over time. There are, at present, no facilities for indexing this library in any way.

Figure 1 shows C^YNTHIA's display. There are three main parts to this. The upper part of the display contains the definition being edited. Note that only one definition is displayed at a time. Other definitions may be selected from the EDIT menu. 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. The entries in the menu differ depending on which part of the program has been selected — i.e. only those commands that are currently applicable are given as an option. After selecting a command, the user is presented with a dialog box for him to enter any necessary parameters for the command. He can either enter these parameters as text or select them from a menu of suitable options. When the mouse is moved over a function symbol, the subterm which has that function symbol as top-level operator is highlighted and it is this subterm that is acted upon.

The middle section of the display is used for system messages. The lower part lists all *valid* recursive calls that are currently available for insertion into the program. Valid recursive calls are ones that would not introduce non-termination if used in the definition. C^YNTHIA uses this list of recursive calls for checking termination of definitions.

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^YNTHIA as a way of transforming one function definition into another. The example does not illustrate the full capabilities of C^YNTHIA but gives a typical situation in which it is used. 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

[5] Throughout this paper, we will use 'he' as a pronoun representing either a male or female.

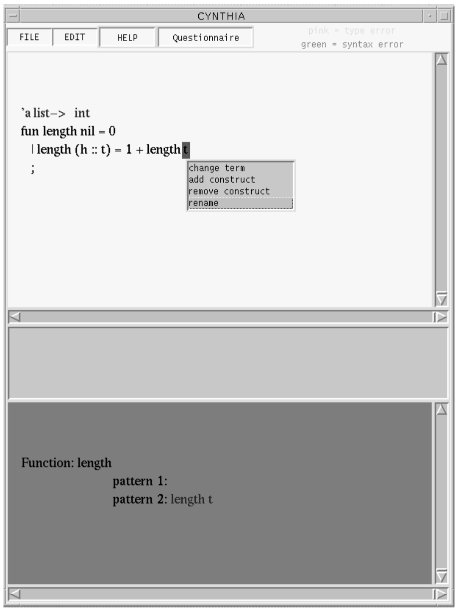

Fig. 1. Graphical user interface to *CYNTHIA*

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.

hasentry 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 values[6]. 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 first thing to do is to rename this function.

[6] This 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.

If the user selects the command RENAME at the indicated point[7], the definition will change to:

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

Note how all of the occurrences of `length`, not just the highlighted one, have been changed. Next, the student changes the output type of the definition using the command CHANGE TYPE:

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

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:

```
'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), giving[8]:

```
'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. Exceptions are a way of leaving a result undefined. The above constitutes a reasonable definition of `hasentry`. The student now proceeds to define `findentry` and notices the similarity between the two definitions.

[7] Throughout this paper, program code enclosed in boxes denotes the point at which the user has applied an editing command.

[8] Strictly, 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.

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
| 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:

```
'a -> 'a -> 'a list -> 'a list
fun addentry k v nil = k::v::nil
| addentry k v (h::nil) = raise excep
| addentry k v (h::h1::t) = if k=h then k::v::t
                            else h::h1::addentry k v t;
```

At this point, the student may decide that he has not chosen the best representation for tables. A better choice would have been a list of pairs. *CYNTHIA* can be used to transform his 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 function. 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 *CYNTHIA* 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 acts on labelled, binary trees. 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;
```

CHANGE TYPE is one of *CYNTHIA*'s most powerful commands. It can make a transformation between patterns of a wide variety of types[9] 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.

3.2 The Design of *CYNTHIA*

From a technical point of view, the novel aspect of *CYNTHIA* 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 80]. A description lies beyond the scope of this paper but can be found in [Whittle 99]. 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 *CYNTHIA*'s interface means that the user requires no knowledge of the underlying proof technology and indeed is unaware of its existence.

3.3 The Scope of *CYNTHIA*

CYNTHIA supports a purely functional subset of the core SML language. It is meant to be an environment for editing function definitions and as such does not support evaluation of expressions. Currently, datatype declarations are not included. References and record types have also been omitted. The scope turns out to be sufficient for a typical first course in ML. The main omissions of the core language are that mutually recursive functions cannot be defined and user-defined types are not supported.

4 Evaluation of *CYNTHIA*

This section reports on empirical studies undertaken with two groups of students (SG1 and SG2) at Napier University to assess if *CYNTHIA* provides a better environment for FP. The main research questions being asked are:

1. How successful was *CYNTHIA* as an editor for ML that guarantees correctness?
   (a) How does the quantity of errors made using *CYNTHIA* compare to text editors?
   (b) How does *CYNTHIA*'s error feedback compare to the compiler's?
   (c) How does the user's productivity rate using *CYNTHIA* compare to text editors?
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?

[9] defined explicitly in [Whittle 99]

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:

– Logging interaction (both SG1 and SG2). The students' sessions with $C^{Y}NTHIA$ were recorded as they worked. For SG1, interactions with the compiler — New Jersey of SML (SML-NJ) — were also recorded.
– A 'crossover' experiment (SG1). In a two hour session, students were given a set of six functions to write. SG1 was split into two groups, X and Y. X attempted part A of the test with $C^{Y}NTHIA$ and part B without. Y attempted part A without and part B with $C^{Y}NTHIA$. Both A and B contained tasks of similar difficulty. Hence, each student attempted similar problems both with and without $C^{Y}NTHIA$. This meant that individual students' performance could be compared.
– A group of 4 SG1 students were videoed attempting four short exercises (see Appendix A). They were asked to attempt some questions with $C^{Y}NTHIA$ and some without. In this way, individual students' performance could again be compared.
– Informal observations were taken by both authors. Students were observed during their tutorial sessions interacting with and without $C^{Y}NTHIA$.

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

Table 1. Research Questions — Criteria and Measures.

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] and could ask questions of the tutors. The ML course is divided into eight short tutorials consisting of the introduction 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) [NJS96] 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.
Lesson 8: Accumulators in recursion, mutual recursion, nested definitions.

Fig. 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 undergraduate 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 not possible given the experimental setup. 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 $C^Y NTHIA$ to only one half of the students would have been unethical. Even if $C^Y NTHIA$ 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 $C^Y NTHIA$ 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 $C^Y NTHIA$ outside the tutorials.
- *Interdependency of $C^Y NTHIA$'s features.* The different aspects of $C^Y NTHIA$ 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 majority of experiments undertaken were at an informal level. The crossover experiment was the only (semi-)formal experiment which involved splitting students into two groups. The crossover design was preferred because it gave each student a chance to use both systems.

4.3 Answering the Research Questions

1. How successful was C^YNTHIA as an editor for ML that guarantees correctness?

C^YNTHIA 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.

1.a. How does the quantity of errors made using C^YNTHIA compare to text editors?

Throughout this chapter, C^YNTHIA will be compared to a traditional text editor. The text editor approach will usually be referred to as TEA, but may also be called the compiler-only approach. Recall that two groups of students were evaluated and SG2 was given a slightly more developed version of C^YNTHIA. Recall also that the programming experience of the two groups differed significantly and that for SG2, C^YNTHIA was more integrated into the course.

One way to compare the number of errors made is to look at logs of C^YNTHIA-interaction and make a count of the errors. For SG1, a count was made of errors made during the crossover experiment. The crossover experiment took place over one hour of intense programming. SG2 errors were counted over 8 weeks during which C^YNTHIA 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 96]. Figure 3 gives this classification.

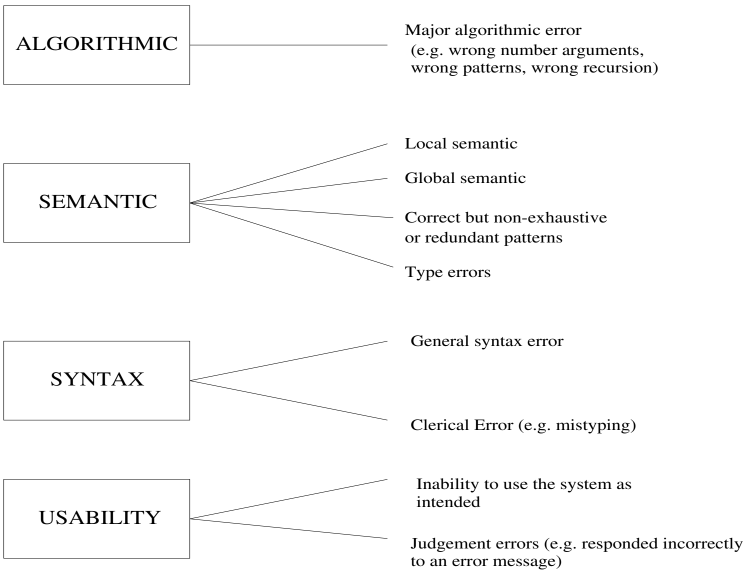

Fig. 3. Classification of Programming Errors.

I briefly explain the motivation behind this classification. A full list of each error in each class is given in [Whittle 99].

- **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. C^YNTHIA 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 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 C^YNTHIA was 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 C^YNTHIA 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 C^YNTHIA, 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 `;` 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^YNTHIA and non-C^YNTHIA users. The results are only meant to be suggestive.

The difference in the total error count is not as high as expected. However, the kinds of errors committed are *different* for C^YNTHIA and non-C^YNTHIA users.

	C^YNTHIA	non-C^YNTHIA
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

Table 2. SG1 Errors.

Syntax errors were almost eliminated when using C^YNTHIA. 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^YNTHIA 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^YNTHIA-specific errors). This is a disappointingly large figure. It suggests first that C^YNTHIA'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^YNTHIA users. The judgement errors were meant to measure this sort of thing, but the results do not back up the informal observations. It is worth some closer inspection to see why this is the case. Table 3 gives a more fine-grained analysis of the judgement errors. See Appendix B for definitions of J1-J5.

	C^YNTHIA	non-C^YNTHIA
J1	8	0
J2	0	53
J3	35	0
J4	4	0
J5	3	0
Total	50	53

Table 3. Judgement Errors made during Crossover Test.

All of the SML-NJ judgement errors fall into the same category – J2. J2 occurs when the student has misunderstood an error message and has changed the wrong part of the program in response to this message. The fact that all SML-NJ judgement errors are of type J2 shows that the SML-NJ error messages are cryptic. These are fairly serious errors. As expected, 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 are similar to J2 errors but are concerned with dialog boxes. Users may be asked to type in text as a parameter to an editing command. If there is an error, they will be told and should edit the text. A J3 error is when the student edited the wrong part of this text in response to $C^{Y}NTHIA$'s error message. Close analysis of the logs show that about 95% of the time, J3 errors occurred because the user tried to type in an entire conditional expression rather than just the actual condition. They might type in `if h=x then delete x t else h::delete x t` and get a syntax error because the `if` keyword is not recognised by $C^{Y}NTHIA$ (it expected the user to simply type `h=x`). The student would then try and make changes such as changing to `h>x` or `h::(delete x t)`. In fact, one student made 13 such mistakes. Clearly, the real problem here is that the student has not correctly understood the functionality of the command needed.

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**. It was assumed, wrongly, in the first evaluation that the editing commands were simple enough for their functionality to be obvious. Hence, the only documentation was limited on-line help, a short tutorial and a Web page describing each command in detail. For the second evaluation, the documentation of $C^{Y}NTHIA$ was more closely integrated into the Napier 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 $C^{Y}NTHIA$. The intention was that if the students could relate more easily to the set of commands, they would commit fewer errors.
- **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, as in the example above, the dialog box would prompt the user with

  `if` $\boxed{\quad}$ `then` `else`

  rather than just a box. This made it more obvious what was required.

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. The error counts for SG1 were made during a timed session. For SG2, all errors over the entire course were collected. Table 4 gives the error count for Evaluation 2. To enable a tentative comparison with SG1, the second column of Table 4 gives the increase from experiment 1 to experiment 2 expressed as a percentage of experiment 2 over experiment 1. It gives a very cautious estimate of the change in relative importance of each kind of error. All things being equal, we would expect each error class to increase by the same amount. The third column gives another way of comparing the different error categories — it gives a measure of the number of errors made *per edit*, again expressed as a percentage.

	No. errors	% relative to expt 1	% Error rate
Local Semantic	39	195	1.19
Global Semantic	41	293	1.26
Patterns	0	0	0
Type Errors	59	295	1.8
General Syntax	0	0	0
Clerical	32	356	0.98
Incorrect Usage	74	140	2.3
Judgement	27	54	0.8
Total	323	175	
Total no. Edits	3266	690	

Table 4. SG2 Errors.

The total number of edits increased roughly seven-fold. However, most error categories increased by much less. This suggests (as expected) that as users became more familiar with how $C^{Y}NTHIA$ worked, they committed fewer and fewer errors. Incorrect Usage errors increased much less than Semantic and Syntax errors. This is evidence that the improved documentation in the second evaluation meant that students were more comfortable with using the system. Judgement errors are the only error that have reduced in absolute terms. This is mainly due to the reduction of J3 errors. In the first experiment, J3 accounted for 70% of the total Judgement Errors. In the second experiment, it is only 16%. It seems then that Evaluation 2 provides evidence that the initial usability problems were teething problems that could be filtered out. Hence, the saving in other type errors is not offset by a gain in usability errors.

It would be useful, in future versions of $C^{Y}NTHIA$, to distinguish between different kinds of error messages in $C^{Y}NTHIA$. There are basically two kinds of error: type A errors where there really is an error, e.g. in syntax, typing. And type B errors where $C^{Y}NTHIA$ gives an error because the user is trying to do something that is not possible — e.g. apply MAKE PATTERN to a polymorphic variable. It seemed that the students were often unable to make the distinction between these two things.

The kinds of errors encountered seemed to change between the two evaluations. In the first evaluation, the majority of errors were type A errors. In the second evaluation, there were relatively few type A errors but more type B errors. This suggests that users were held back less in the early days and managed to progress to deal with more interesting concepts.

1.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?

The question will be answered on a qualitative level. Evidence from videoing, observation and communication with students seems to suggest very positively that $C^{Y}NTHIA$

does better than TEA. First, the kinds of errors that students make are different when using $C^{Y}NTHIA$. 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, I 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 $C^{Y}NTHIA$ 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 $C^{Y}NTHIA$. 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 well-defined. 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.

When dealing with type errors, students in SG1 immediately noticed, thanks to $C^{Y}NTHIA$'s highlighting, that there was a problem and knew which part of the program to change. They did not always know the correct changes to make, however. Student 2, in the videoing session, trying `maxlist`, had as part of his program:

```
if h > [h::t] then ....
```

Although `h > [h::t]` is highlighted in pink by $C^{Y}NTHIA$, the student did not try to change this immediately. He noticed something was wrong but was not capable of making a correction just yet. He went off and edited other parts of the program before finally coming back to it. This seems to be a relatively common phenomenon. Students might ignore the highlighting but would have to address it sooner or later. When student 2 came back to address the type error, he verbalised: "Why is this this colour?" He tried to change the condition to `h > (h::t)` but it remained pink. "`h::t` is a list and what I want is just a number." So, student 2 correctly identified the cause of the type error but was not immediately aware of the correct expression (i.e. which integer) with which to replace `h::t`. After a little more thought, he found the correct answer.

This observation is typical. Although students could easily locate the source of an error, they often needed help to be able to replace the offending expression with a correct one. With SML-NJ, students regularly do not know which part of the program is wrong. They make a half-hearted guess and then re-compile the program, only to find more error messages. This process can go on for some time. In $C^{Y}NTHIA$, students can just change the pink expression and see what happens rather than having to re-compile everything. It seems then that $C^{Y}NTHIA$ succeeded in giving the students a more focussed way of correcting their type errors.

There are situations where $C^{Y}NTHIA$ can be misleading with regards to type errors. This happens when type inference would succeed on a definition but type checking in $C^{Y}NTHIA$ 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 $C^{Y}NTHIA$ does not. Consider the example:

```
'a -> 'b -> 'a
fun f x y = if g x then x else [y] ;
```

The user gets a type error in *CYNTHIA* at the indicated point until he changes `'b` to `'a`. It may be possible for *CYNTHIA* to automatically update such type declarations (or at least suggest updates). In any case, it would be worth working at incorporating type inference in *CYNTHIA*.

1.c. How does the user's productivity rate using *CYNTHIA* compare to text editors?

We answer this question with specific reference to the video experiment. The four subjects that took part in the videoing each worked through a maximum of three examples of increasing difficulty. Some of these were attempted using *CYNTHIA* and some without *CYNTHIA*. Table 5 gives the number of examples and timings for each student. Times are given in minutes and seconds. C denotes that the example was attempted using *CYNTHIA*. S denotes that the student did not use *CYNTHIA*. Students were given help if they asked for it. This was minimal except for student 3 who was given a substantial amount. The examples involved writing simple recursive functions. Since the experiment was undertaken only two weeks after recursion had been introduced, the students understandably had some difficulty. Student 4 failed to finish `addlist`. Students 2 and 3 did not have time to attempt it.

Student	`leading0s`	`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 because it involves multiple recursion. Students 1, 2 and 4 seem to have performed better with *CYNTHIA*. On the first two tasks, all of these students took less time when using *CYNTHIA* – on average, 35% less. Student 3 took longer when using *CYNTHIA* but the amount of help given was so great that a comparison is meaningless. 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 *CYNTHIA*. 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. All of these examples are relatively simple. An interesting question is whether *CYNTHIA* would provide a saving on more complex examples where implementation is more mixed in with trying to decide on the abstract algorithm required.

Clearly, too many conclusions should not be drawn from the timings given above. The number of subjects studied is very small and so can only be suggestive of a trend. Further investigation is needed.

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

2.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 well-understood, was their function clear, etc.? Green et al [Green & Petre 96] 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 I will not mention all of the dimensions here, they serve 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 99].

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 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, a wide variety of programs can be produced (much wider than comparable systems such as [Brna & Good 96, Bundy *et al.* 91]). However, the high number of Incorrect Usage errors in Tables 2 and 4 show that the commands caused some confusion. As already mentioned, this is partly due to practical issues such as documentation. In addition, the abstractness of the editing commands seemed difficult to learn. Some commands are based on functional programming concepts — for example ADD ARGUMENT, MAKE PATTERN, CHANGE TYPE — and if the student has not got to grips with functional programming, he is likely to be baffled by the command. 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. Some students commented that they did not find the naming of the commands very intuitive. 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 $C^{Y}NTHIA$. 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 B). Four errors, U12, U14, U16 and U7, contribute over 60% of the total count.

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

Table 6. Incorrect Usage Errors.

Three of these errors are in response to events that the user has every right to assume should be possible. U12 says that MAKE PATTERN cannot be applied to integers (integers were dealt with in a sub-optimal way in the evaluation version. Because integers are not based on finite constructor definitions, it is non-trivial to allow the pattern edits to work with integers. The evaluation version required the user to start with an integer recursion if it was needed). U14 is similar — the result of CHANGE TYPE needs to introduce a pattern definition on integers. U16 is again something that $C^{Y}NTHIA$ should support. U16 errors occur when the user fails to use ADD RECURSIVE CALL to introduce a new recursive call, but attempts to change an existing recursive call directly using CHANGE TERM. All of U12, U14 and U16 are areas where $C^{Y}NTHIA$'s functionality is lacking in some way. Note that the other kinds of errors — which crop up when the wrong parameters are introduced etc. — are very few in number. The remaining major error, U7, is also understandable — it concerns confusion between RENAME and CHANGE TERM. CHANGE TERM is meant to modify a specific occurrence of an expression, whereas RENAME modifies all occurrences in the definition.

The four errors mentioned show 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. The first evaluation threw up an example of a hidden dependency in $C^{Y}NTHIA$ to do with $C^{Y}NTHIA$'s type feedback mechanism. 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 $C^{Y}NTHIA$ 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 $C^{Y}NTHIA$ 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. Arguably, the student learns from the experience. The student is made more aware by solving the error that a nested list type is being utilised. This could have gone unnoticed using TEA.

One solution to the problem would be to highlight also the part of the top-level type declaration that is responsible for the error. In fact, this would be good even if the error is in the program — to help the user understand the error. An expert version of $C^{Y}NTHIA$ should incorporate type inference so that it conforms more closely to the standard notion of programming in ML.

Premature commitment

This dimension concerns the extent to which the user is forced to make a decision before the information is available. In the context here, this manifests itself as 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.* 91], 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. To what extent is this also true of $C^{Y}NTHIA$?

For the most part, the order of editing commands in $C^{Y}NTHIA$ is irrelevant. If the user applies an incorrect edit, and only realises this much later, it is easy to undo the edit by applying a short sequence of recovery commands. This contrasts starkly with the recursion editor, where the user can easily get stuck down an incorrect route from which the only way to recover is to re-start from the beginning. In the worst case, the recovery commands in $C^{Y}NTHIA$ will unpack those edits applied so that the user retraces his steps back to the start, but in the vast majority of cases, the recovery commands form a short route back onto the correct path.

There are a few examples, however, where $C^{Y}NTHIA$ requires some form of premature commitment. In all of these cases, premature commitment merely makes life easier — it does not prevent the user from doing something. An example is where it is useful to decide upon the patterns that define a function initially, but where it can be awkward to revise this choice. Suppose the user is writing a function, `app`, to append two lists together and begins by splitting the second argument:

```
fun app l nil = l
|   app l (x::xs) = x :: app l xs;
```

It is at this point that the user realises he should have split the first argument instead. Ideally, there would be a command to transfer the patterns from the second to the first argument. Currently, however, the user must apply REMOVE PATTERN to give:

```
fun app l l2 = l;
```

and then MAKE PATTERN on `l`, giving:

```
fun app nil l2 = nil
|  app (x::xs) l2 = 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 PATTERN[10]. Hence, although the user can achieve the desired goal, he needs to go a long way round to get there. Another example of this is when the user has initially chosen the wrong programming construct — suppose he introduced a conditional statement of the form `if (x:nat) = zero` into his program but now wants to use the `case` construct on `x` to introduce patterns for `x=zero` and `x= s n`. Although there are two cases for both `if` and `case`, there is no direct way to achieve this. The user must first remove the conditional statement then introduce `case` and probably re-implement parts of the program that had been deleted. This phenomenon is a consequence of the fact that the number of editing commands was deliberately kept to a minimum. More advanced users could easily find this sort of thing frustrating, however, and so commands such as one that transfers patterns from one argument to another would be of great use.

Observations of the students showed that occasionally, they found it easier to start from scratch rather than try to find the correct sequence of recovery commands. There are two cases when this can happen. The first is when the user applies an incorrect edit and cannot see how to correct it easily. The second is when the user gradually realises that he has chosen a sub-optimal starting example and so goes back and starts with a different example. There is sometimes a fine line between persevering with the current source example or choosing an alternative. Note that it can be a valuable learning experience to revise choices in this way. The student is refining their concept of similarity of functional programs. This will help them to develop an improved strategy for writing functions later (whether using C^YNTHIA or not). This strategy is an essential part of an expert programmer's repertoire and hence, We claim that C^YNTHIA is encouraging the novice user to think in this kind of way.

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^YNTHIA improves on ML compilers 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 gets constant feedback about semantic errors during the programming process.

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

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.

2.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 96, Runciman & Toyn 91]. In $C^{Y}NTHIA$'s case, however, no library search functions are provided. This is for two reasons. First, since $C^{Y}NTHIA$ 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 recently[11]. 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, a student wrote `combine`, which has four definition clauses, 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 $C^{Y}NTHIA$ 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 `leading0s`. 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

[11] This backs up the claim made in [Weber 96].

looked for a function with 2 lines of code as source for `leading1s`. He chose `doublist` even though this has a result type of `int list` not the required `int`.

A few students asked how they should go about choosing source functions. The students were probably trying to make the task more difficult than it was. Given that there was definitely a choice to make, these students were overly cautious and expected there to be some hidden set of rules for making these choices which they might be tested on later. It was as though the students could not believe that their initial choice was so unimportant and so they assumed they must be missing out on something.

In a more advanced, expert-oriented version of C^YNTHIA, it might be useful to have some sort of library retrieval mechanism. The number of functions in the database would be much larger. A very simple, but potentially effective, solution would be to implement a library browser which annotated the function name with the type of the function and perhaps the kind of recursion scheme, or a sentence (provided by the user) of its meaning.

5 Conclusion

This paper has examined the shortcomings of current editors for functional programming languages, in particular ML. It was discovered that students have particular difficulties when programming functionally — principally because of types and recursion. C^YNTHIA is an editor which attempts to overcome some of these problems in two ways. First, C^YNTHIA provides editing commands for allowing old programs to be transformed into new ones. Second, C^YNTHIA provides correctness checks not found in any other comparable editor. Two trials were conducted to see what effect the use of C^YNTHIA has in teaching functional programming. Tentative conclusions can be drawn:

- Students generally make fewer errors when using C^YNTHIA than when using a traditional text editor.
- C^YNTHIA's highlighting of errors means that students locate and correct type errors more easily.
- C^YNTHIA tends to discourage aimless hacking and encourage good programming style.

Further investigation is needed to examine these issues in more detail. Specifically, it would be interesting to see how useful students find advanced correctness-checking facilities such as termination, and to see what are the implications for more expert users.

Our experiments with C^YNTHIA suggest a new trend for functional program editors. C^YNTHIA-like transformations could be very useful in maintaining existing programs. Software often needs to be updated by making very small changes. A tool like C^YNTHIA could make these changes but guarantee that no errors are introduced as a by-product. Another interesting avenue would be to extend the correctness guarantees that C^YNTHIA offers. Although a challenging task, we believe that C^YNTHIA's underlying framework could allow the user to make assertions about the behaviour of their functions and then to have these assertions proved. The ideas behind C^YNTHIA are also applicable to non-functional programming — [Blewitt 98] describes a C^YNTHIA-like tool for Java.

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

[Aitken 96] S. Aitken. An analysis of errors in interactive proof attempts. Technical report, Dept. of Computer Science, Glasgow University, June 1996.

[Anderson *et al.* 88] J. R. Anderson, P. Pirolli, and R. Farrel. Learning to program recursive functions. *The Nature of Expertise*, pages 153–183, 1988.

[Bhuiyan *et al.* 94] S. Bhuiyan, J. Greer, and G. I. McCalla. Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2):115–139, 1994.

[Bird & Wadler 88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[Blewitt 98] A. Blewitt. A Java editor based on proofs-as-programs. Unpublished M.Sc. thesis, Department of Artificial Intelligence, Edinburgh, Scotland, September 1998.

[Bowles & Brna 93] A. Bowles and P. Brna. Programming plans and techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proceedings of the World Conference on Artificial Intelligence in Education*, pages 378–385. AIED, August 1993.

[Brna & Good 96] P. Brna and J. Good. Searching for examples: An evaluation of an intermediate description language for a techniques editor. In P. Vanneste, K. Bertels, B. de Decker, and J-M. Jaques, editors, *Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group*, pages 139–152. 1996.

[Bundy *et al.* 91] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[Duggan & Bent 96] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[Gegg-Harrison 91] T.S. Gegg-Harrison. Learning Prolog in a schema-based environment. *Instructional Science*, 20:173–192, 1991.

[GIML] Department of Computing Studies, Napier University, Craiglockhart campus, 219 Colinton Road, EH14 1DJ. *A Gentle Introduction to ML*. http://www.dcs.napier.ac.uk/course-notes/sml/manual.html.

[Green & Petre 96] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[Hansen 71] W.J. Hansen. Graphic editing of structured text. In R.D. Parslow and R.E. Green, editors, *Advanced Computer Graphics*. Plenum Press, 1971.

[Har96] *MLWorks*. Harlequin, Inc., 1996.

[Howard 80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[Jun & Michaelson 98] Y. Jun and G. Michaelson. A visualisation of polymorphic type checking. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt University, March 1998.

[Kirschenbaum *et al.* 89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University, 1989.

[McAllester & Arkoudas 96] David McAllester and Kostas Arkoudas. Walther recursion. In M.A. McRobbie and J.K. Slaney, editors, *13th International Conference on Automated Deduction (CADE13)*, pages 643–657. Springer Verlag LNAI 1104, July 1996.

[Michaelson 95] G. Michaelson. *Elementary Standard ML*. UCL Press, November 1995.

[Milner *et al.* 90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NJS96] Standard ML of New Jersey, version 0.93. Tech.rep, AT&T Bell Laboratories, 1996. Available at http://www.dcs.napier.ac.uk/course-notes/sml/BASE.ps.

[Paulson 91] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Rideau & Théry 97] L. Rideau and L. Théry. An interactive programming environment for ML. Rapport de Recherche 3139, INRIA Sophia Antipolis, March 1997.

[Runciman & Toyn 91] C. Runciman and I. Toyn. Retrieving reusable software components by polymorphic type. *Journal of Functional Programming*, 1(2):191–211, April 1991.

[Teitelman & Reps 84] T. Teitelman and T. Reps. The Cornell program synthesizer: a syntax-directed programming environment. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 97–116. McGraw-Hill, 1984.

[Teitelman 75] W. Teitelman. *INTERLISP Reference Manual*. XEROX, 1975.

[Ullman 94] J. D. Ullman. *Elements of ML programming*. Prentice-Hall International, Englewood Cliffs, N.J., 1994.

[Weber 96] G. Weber. Individual selection of examples in an intelligent learning environment. *Journal of AI in Education*, 7(1):3–31, 1996.

[Whittle 99] J.N.D. Whittle. *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor*. Unpublished PhD thesis, Division of Informatics, University of Edinburgh, 1999.

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

3) Write a function which takes two lists of integers and adds
together corresponding elements of the lists.
Examples: addlist [1,2,3] [4,5,6] = [5,7,9]
          addlist [1,2,3] [] = [1,2,3]
```

Appendix B

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