An Editor for Helping Novices to Learn Standard ML

Jon Whittle[1] and Alan Bundy[1] and Helen Lowe[2*]

[1] Dept. of Artificial Intelligence, University of Edinburgh
80 South Bridge, Edinburgh EH1 1HN, UK.
[2] Dept. of Computer Studies, Napier University
Craiglockhart, 219 Colinton Road, Edinburgh
EH14 1DJ, UK.
Email: jonathw@dai.ed.ac.uk

Abstract. This paper describes a novel editor intended as an aid in the learning of the functional programming language Standard ML. A common technique used by novices is programming by analogy whereby students refer to similar programs that they have written before or have seen in the course literature and use these programs as a basis to write a new program. We present a novel editor for ML which supports programming by analogy by providing a collection of editing commands that transform old programs into new ones. Each command makes changes to an isolated part of the program. These changes are propagated to the rest of the program using analogical techniques. We observed a group of novice ML students to determine the most common programming errors in learning ML and restrict our editor such that it is impossible to commit these errors. In this way, students encounter fewer bugs and so their rate of learning increases. Our editor, C^YNTHIA, has been implemented and is due to be tested on students of ML from September, 1997.

Keywords: Programming Language Learning, Learning Environments, Analogy

1 Introduction

Functional programming languages such as LISP, ML and Haskell are increasingly being used in academe and industry. Many universities now teach functional languages as a key part of their software engineering programme. However, the teaching of such languages presents problems. Functional languages involve abstract concepts such as recursion which are difficult to learn ([1]). Many experiments have been carried out that suggest that students overcome these difficulties by using analogy in the early stages of programming [16, 18]. Given a program to write, novices refer to similar programs they have written before or seen in the course literature. They then use the old program as a basis to construct the new one. We have conducted our own informal experiment with a group of 30 novice ML students which involved observations of the students over the course of a semester and in-depth interviews with two of the students. This provided additional evidence of programming by analogy [19].

ML is a typed, functional language incorporating extensive use of pattern matching and recursion. We have implemented a program editor, C^YNTHIA, for Standard ML that supports programming by analogy. Programs are constructed in C^YNTHIA by transforming an existing program from an available library. The user is provided with a collection of editing commands. Each command makes an isolated change to the current program, such as adding an extra argument to a function definition. The effects of this change are then propagated automatically throughout the rest of the program. By applying a sequence of editing commands, previously constructed programs can be easily transformed into new ones. In addition, programs produced using C^YNTHIA are guaranteed free of certain kinds of bugs. To illustrate the idea, consider the task of writing a function, *count*, to count the number of nodes in a binary tree, where the definition of the datatype *tree* is given in ML as:[3]

```
datatype tree = leaf of int | node of tree * tree;
```

Suppose the user recognises that a function, *length*, to count the number of items in an integer list, is similar to the desired function. He[4] can then use *length* as a starting point. Below we give the definition of *length* preceded by its type[5].

---

[*] The first author was supported by an EPSRC studentship. The second and third authors are supported by EPSRC grant GL/L/11724
[3] *int* is the built-in datatype integers.
[4] Throughout this document, I refer to the user by the pronoun 'he' although the user may be male or female.
[5] :: is the ML list operator cons

```
'a list -> int

fun length nil = 0
|   length (x::xs) = 1 + (length xs);
```

Note that *'a list* is the polymorphic list type. We show how *length* could be edited into *count*. This example is taken from [20].

1. The user may indicate any occurrence of *length* and invoke the RENAME command to change *length* to *count*. C^YNTHIA then changes all other occurrences of *length* to *count*:
   ```
   'a list -> int

   fun count nil = 0
   |   count (x::xs) = 1 + (count xs);
   ```
2. We want to count nodes in a tree so we need to change the type of the parameter. Suppose the user indicates *nil* and invokes CHANGE TYPE to change the type to *tree*. C^YNTHIA propagates this change by changing *nil* to (*leaf n*) and changing :: to *node*:
   ```
   tree -> int

   fun count (leaf n) = 0
   |   count (node(xs,ys)) = 1 + (count xs);
   ```
   Note that the program no longer contains x. Instead, a new variable *ys* of type *tree* has been introduced. In addition, (*count ys*) is made available for use as a recursive call in the program.
3. It remains to alter the results for each pattern. 0 is easily changed to 1 using CHANGE TERM. If the user then clicks on 1 in the second line, a list of terms appear which include (*count ys*). Selecting this term produces the final program:
   ```
   tree -> int

   fun count (leaf n) = 1
   |   count (node(xs,ys)) = (count ys) + (count xs);
   ```

The editing commands available can be divided into two types: low- and high-level commands. Low-level commands make only very small changes to the existing program, such as changing 0 to 1 in (3) above. High-level commands affect the overall structure of the program, e.g. changing the top-level type in (2). C^YNTHIA encourages the use of high-level commands first to set up a 'shell' for the definition. Low-level commands can then be used to fill in the details.

We aim our system primarily at novices. However, C^YNTHIA is general enough to allow complex, practical programs to be produced. It is unlike many tutoring systems (e.g. [2]) that are restricted to a small number of toy examples. This means the novice has the freedom to experiment and enables continued support once the novice has become more expert.

2 The Design of C^YNTHIA

We wish C^YNTHIA programs to be guaranteed correct in some respects. It is natural, therefore, to base the design around established techniques from logic and proof theory which give us a flexible and powerful way of reasoning about the correctness of programs. [9] identifies the necessary machinery to set up a one-to-one correspondence between functional programs and mathematical proofs in a constructive logic. Note that under this correspondence, recursion in a functional program is dual to mathematical induction in a proof. Hence, changing the recursion scheme corresponds to changing the induction scheme. This idea has been used as the basis for program verification and synthesis. For instance, within the paradigm of program verification, given a program, we can prove it correct by proving the corresponding theorem in the constructive logic. As an example, given a program to append two integer lists together, we could formulate a theorem[6]

$$\forall x : list(int)\ \forall y : list(int)\ \exists z : list(int)\ (\forall e : int\ e \in z \leftrightarrow e \in x \lor e \in y) \tag{1}$$

[6] $X : T$ means X is of type T

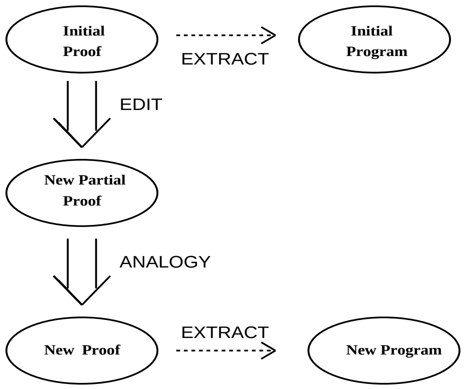

Fig. 1. Editing Programs in C^YNTHIA

This theorem or *specification* states the existence of a list z that contains all elements of x and y and no others. Hence, a possible z is x @ y where @ is the append operator[7]. Suppose we have a proof of this specification in a constructive logic. We can extract a functional program from this proof such that the program is guaranteed to be correct with respect to the specification – i.e. it will compute x @ y. This is called the *proofs-as-programs* paradigm. It enables us to construct programs that are correct in some way.

We use a restricted form of this idea where the specification does not describe the full behaviour of the corresponding program but instead states the number and the type of input arguments and the type of the output argument. For example, *append* would have a spec $list(int) \rightarrow list(int) \rightarrow list(int)$. Every program is associated with a corresponding specification and *synthesis proof*. Our proofs are written in the proof editor *Oyster* [8] which is based on a constructive logic known as Martin-Löf's Type Theory[8] [12]. The synthesis proof essentially guarantees the correctness of the program extracted from it (with respect to the specification) . The more detailed the specification, the more we can guarantee about the program. Our simple specifications prove that C^YNTHIA programs are syntactically correct, well-typed, well-defined and terminating – see §3.2 for more details.

The design of C^YNTHIA is depicted in Figure 1. Note that editing commands directly affect the synthesis proof and only affect the program indirectly. C^YNTHIA is equipped with an interface that hides the proof details from the user. As far as the user is aware, he is editing the program directly. In this way, the user requires no knowledge of logic and proof. The user begins with an initial program and a corresponding synthesis proof. These may be incomplete. Editing commands make changes to a particular part of the synthesis proof. This yields a new partial proof which may contain gaps or inconsistencies. To fill in these gaps and resolve inconsistencies, we use an analogical mechanism. This mechanism *replays* the proof steps in the original (source) proof to produce a new (target) proof. During this replay, the changes induced by the editing command are propagated throughout the proof. Once gaps in the target proof have been bridged, a new program is extracted. This program incorporates the user's edits and is guaranteed correct with respect to the weak specification.

Let us explain how the analogy works in a little more detail. A tactic is a combination of a number of inference rules. Tactics can be written that allow us to apply a frequently used sequence of inference rules in one step. We have implemented a number of tactics that allow us to synthesise ML programs. There are five types of tactics in C^YNTHIA:

- A tactic for pattern matching.
- Tactics for constructing an ML program fragment. Each ML construct (e.g. `if..then..else`, `case`) has a corresponding tactic. Applying the tactic in *Oyster* adds extra branches to the proof tree.

[7] In fact, *append* is just one program that would satisfy the specification. We can write the specification to any level of detail.

[8] The type systems of ML and Martin-Löf, although very similar, are not the same, so some minor translating is done between the two. We chose *Oyster* in which to build our constructive proofs because there is a body of work in synthesising programs in *Oyster*.

- Result tactics for giving the result of a case in the definition (e.g. giving 0 as the result in the *nil* case of the definition of *length*).
- Tactics for type-checking.
- Tactics for checking termination.

By applying tactics, a synthesis proof is constructed. This synthesis proof is represented as a tree where each node in the tree has slots for the hypotheses, the current goal and a tactic to be applied. In our case, the current goal will be of the form $\vdash T$ where T is the output type of the function being defined. This goal should be interpreted as meaning that an object of type T is required to satisfy the goal.

The analogical mechanism works as follows. When an editing command is invoked, C^YNTHIA abstracts away unnecessary information from the proof tree to produce an *abstract rule tree* (ART) which is essentially a tree of tactic applications. If the editing command is applied at a position, P, in the ML program, C^YNTHIA calculates the position corresponding to P in the ART and then makes changes in the ART. This may involve adding, removing or modifying tactics. This process yields a new ART which is then replayed by systematically applying the tactics that make it up, hence producing a new synthesis proof. Note that the changes made by the editing commands are propagated during the replay of the ART. There may be tactics that can no longer apply – in this case, a gap is left in the proof tree which is signalled to the user at the corresponding point in the ML program. These gaps must be 'filled in' before the program is accepted. For an example of this, see §4.

Refer to the example in §1. We briefly explain how the analogy works in (2). The user has selected *nil* and indicated a change of type. This induces a change of the type of *length* from `'a list -> int` to `tree -> int`. Hence, the proof tree is abstracted to an ART with a new specification $tree \rightarrow int$. The ART is now replayed. During this replay, the tactic that implements pattern matching is modified so that the definition is based on a new pattern with constructors $leaf$ and $node$. This also changes the induction scheme in the proof hence making the recursive call (*count ys*) available for future use. The two result tactics are replayed without modifications. The new program can be then be extracted from the new proof tree.

In general, constructing proofs by analogy is a difficult task [15]. Because we are restricted to specifications involving a limited amount of detail, the proofs are simpler and so the analogy becomes a viable option in a practical, real-time system such as ours.

3 Increasing the Learning Rate

Over a period of three months we conducted observations of a group of 30 novice ML students from Napier University to ascertain what problems presented themselves when learning ML. The students were observed writing programs during weekly tutorial sessions. In addition to informal observations, their interactions with ML were scripted and analysed. The students completed questionnaires relating their experiences and at the end of the course, two students were interviewed in depth. We describe how ML programs can be built up quickly and easily using C^YNTHIA. We also point out the problems that the students had with ML and how C^YNTHIA can help to overcome them. The students used version 0.93 of New Jersey ML and so our comments refer to this version.

3.1 Program Transformation

To provide the maximum support for programming by analogy, the editing commands in C^YNTHIA are structured into low-level commands for making very small changes and high-level commands for changing the overall program structure. Not only does this approach constitute a powerful way of transforming programs but it also encourages the novice to follow a top-down approach to programming – deciding on the high-level structure first and then filling in the details.

Low-Level Commands These are commands that only affect an isolated part of the program. They do not affect the datatype of the function being defined. Nor do they affect the recursion the function is defined by. This means that the analogy needed to produce a new program is fairly straightforward. The following are the available low-level commands, with a brief description of each:

- ADD CONSTRUCT: add a construct at the current point in the program. The ML constructs currently supported are `if..then..else`, `case`, `fn`, `let val` and `let fun`.

– CHANGE TERM: change a sub-expression in a term at the current point in the definition only.

High-Level Commands We now present the high-level commands available. First, we give a list of the commands where the analogy is relatively simple. Then we go into more complicated high-level commands.

- RENAME: change the name of a variable name or function name throughout the definition.
- ADD ARGUMENT: add an additional argument to a function definition.
- ADD COMPONENT: if an argument is a pair, add an extra component to the pair. If the argument is not already a pair, make it into one. For example, consider applying ADD COMPONENT to *nil* in the following program.

```
'a list -> int

fun length nil = 0
|   length (x::xs) = 1 + (length xs);
```

  would go to

```
'a list * 'b -> int

fun length (nil,y) = 0
|   length (x::xs,y) = 1 + (length (xs,y));
```

- MOVE ARGUMENTS: swap the positions of two arguments in a definition.

Figure 2 gives an idea of some commands used to transform, *rev* a function for reversing lists, into *delete* for deleting an element from a list. The commands are in upper case. The first step renames the function and adds an extra argument. ADD ARGUMENT is invoked by indicating an occurrence of *rev* and then analogy gives all other occurrences of *rev* an additional argument too. ADD IF..THEN..ELSE places a case-split at the designated position, duplicating whatever is below the current position in the original program. CHANGE TERM is used to edit the result for one of the patterns – e.g. to remove @ in $(delete\ xs\ e)$ @ $[x]$ giving $(delete\ xs\ e)$.

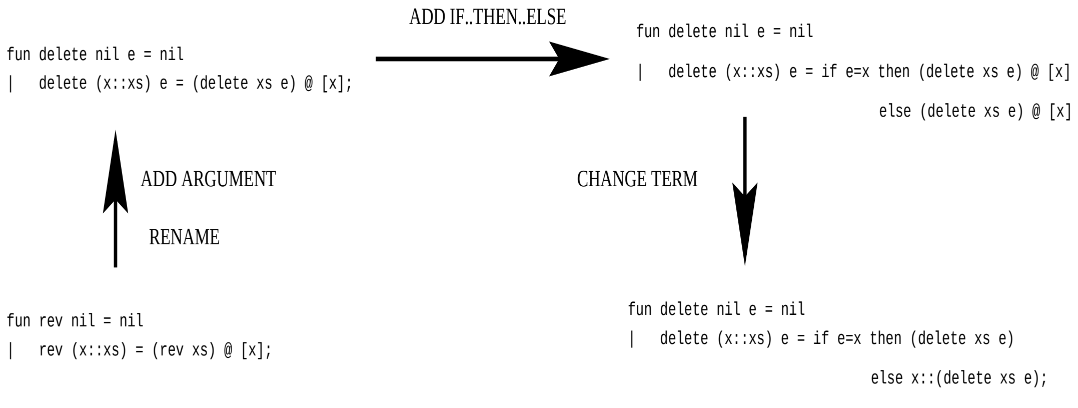

Fig. 2. Low-Level Commands

More complicated high-level commands are for changing the recursion and changing the type of an argument.

Definition by Patterns By *definition by patterns* we mean the common practice as used in ML whereby functions are defined by pattern matching (see *rev* for example in §3.1). We observed that novices often have difficulty in deciding upon the correct definition by patterns for a function. They are capable in simple cases where the function has only one argument that is pattern matched against, but become lost when more than one argument is pattern matched or when the pattern used is non-standard. A simple function that pattern matches multiple arguments would be an *nth* function to return the nth element in a list.

We have a command MAKE PATTERN which allows the user to build up non-standard patterns by combining a number of standard ones. We have implemented a version of a technique used in ALF [5]. The user can highlight an object of a certain datatype. The application of the MAKE PATTERN command splits the object into a number of patterns - one for each constructor function used to define the datatype. Hence, MAKE PATTERN on *x:list* below

```
fun f(x,l)=..
```

produces two patterns:

```
fun f(nil,l)=..
|   f(h::t,l)=..
```

Non standard patterns can be defined by applying the command a number of times. Highlighting t and applying MAKE PATTERN gives:

```
fun f(nil,l)=..            (1)
|   f(h::nil,l)=..         (2)
|   f(h::h2::t1,l)=..      (3)
```

This can be done for any datatype by using the definition of the type as encoded in ML. Suppose l is of type *tree* then we can split l in pattern number (2) to give:

```
fun f(nil,l)=..
|   f(h::nil,(leaf x))=..
|   f(h::nil,(node(l1,l2)))=..
|   f(h::h2::t1,l)=..
```

We do not use the same underlying theory as ALF but use the constructive logic already available in our proof system. The result is the same, however. This command can also be applied to patterns introduced by the ML construct `case`[9].

Recursion Recursion is well-known to be a difficult concept to learn. Novices can have considerable difficulty with even primitive recursion schemes. However, an introductory course will also introduce non-standard schemes involving accumulators, multiple recursion, course-of-values recursion and nested recursion. To help novices to learn non-standard recursions, the commands ADD RECURSIVE CALL and REMOVE RECURSIVE CALL encourage them to think about which recursive calls are needed for the task at hand. $C^{Y}NTHIA$ maintains a list of recursive calls that are currently available to the user. When the user is required to enter a term, these recursive calls are among the options presented to the user. He can pick one using the mouse without any need for further typing. The user can change the list of recursive calls by the commands mentioned above. The idea is that the user first decides upon what kind of recursion he should use. He can then use these commands to set up the basic structure within which to use them. Other commands can be used to fill in the details.

As an example of how the commands can be used, consider trying to produce the function *zip*[10]:

```
fun zip f nil nil = nil
|   zip f nil (i::u) = nil
|   zip f (x::xs) nil = nil
|   zip f (x::xs) (i::u) = f(x,i)::zip f xs u;
```

[9] `case` splits a term into patterns within a definition allowing nested patterns. It does not allow recursive calls

[10] A common definition of *zip* would omit the second and third cases. This is disallowed in $C^{Y}NTHIA$ because we restrict to well-defined programs. The current version of $C^{Y}NTHIA$ does not support exceptions.

Figure 3 shows the edits needed to produce *zip* from *rev*. The ideal way to proceed is to decide upon the program structure to begin with by applying ADD ARGUMENT twice and then MAKE PATTERN twice. The user then introduces the recursive call (*zip f xs u*) as necessary[11]. To avoid restricting the user, he is not forced to produce programs in this top-down fashion. The result is generally independent of the order of execution of commands.

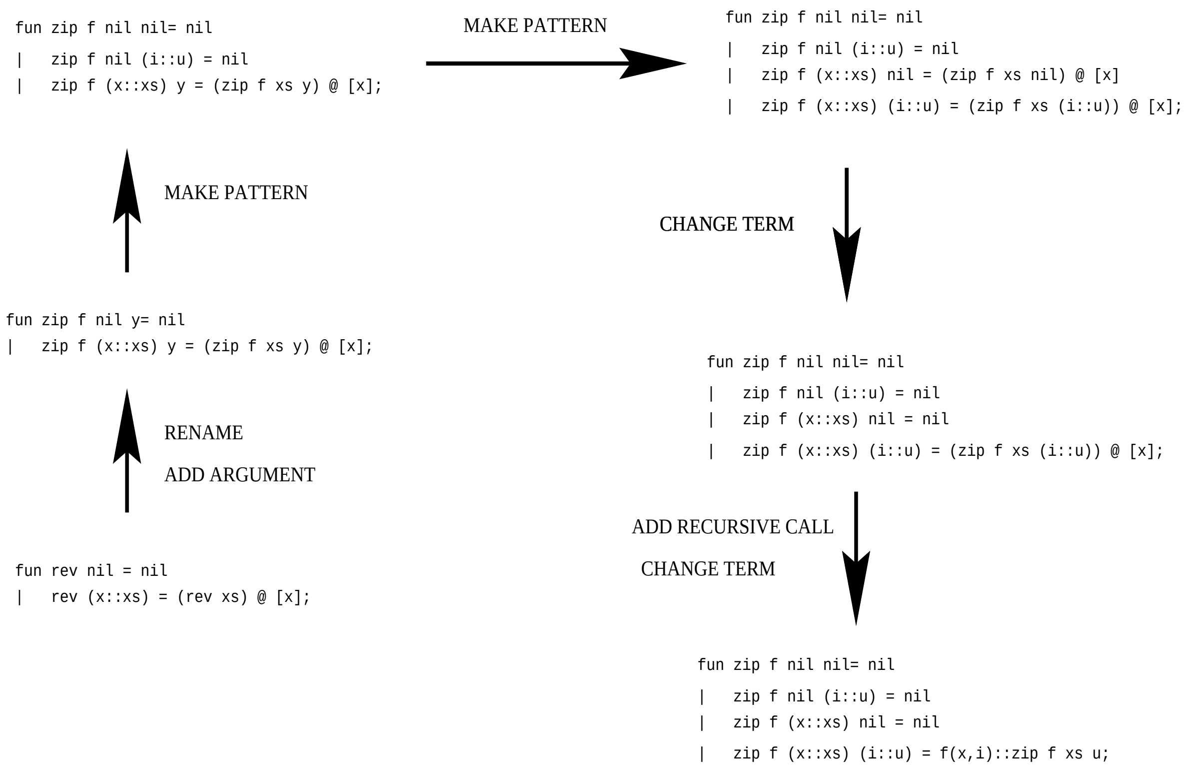

Fig. 3. Writing *zip*

Changing Type A major drawback of current learning processes is that novices can ignore datatypes. ML is equipped with a type inference engine which automatically derives (if possible) the type of the top-level function. Although advantageous in that users need not explicitly state the type of each term, novices can ignore types and be unaware of type inconsistencies which may arise. This results in unhelpful error messages from the compiler and confusion. For instance, in the function:

```
fun length nil = nil
|   length (x::xs) = 1 + (length xs);
```

we have an example of a simple but commonly-made error whereby the output type in the first line is list but is an integer in the second. It can often be difficult to pinpoint exactly where the error has occurred.

For this reason, we insist that the user declares the type of a function before anything else. This forces the novice to think about the types hence reducing type errors when writing the rest of the program. Once the top-level type has been given, the types of terms in most other parts of the program are determined and hence need not be given by the user.

[11] Note that when this recursive call is introduced into the program, the program is checked for termination. In this way, the user is restricted to recursive calls that preserve termination – see §3.2.

During the course of a program the user may realise he has stated the top-level type incorrectly. Or he may want to change the top-level type of an old program to produce a new one. C^YNTHIA provides quite advanced facilities for doing this. Changing the output type of a function or changing the type of a non-recursive argument does not present too many problems. It can result in inconsistencies in the proof (ill-typed terms) but this can be dealt with – see §4. The real challenge comes when changing the type of an argument that is pattern matched against since the pattern needs to be changed. If we wanted to change the type of the first argument in *length* from list to tree, we can invoke CHANGE TYPE to edit *length* into

```
fun length (leaf n) = nil
|   length (node(xs,ys)) = 1 + length xs;
```

In this case, C^YNTHIA will also add the recursive call *length ys* to the list available. This could then be used by the user. More complicated examples arise when MAKE PATTERN has been applied more than once. If our original program had been:

```
fun app2 nil l2 = l2
|   app2 (x::nil) l2 = (x::l2)
|   app2 (x1::x2::xs) l2 = x1::(app2 xs l2);
```

it is not clear what the new pattern should be after a change of type of the first argument to *tree*. C^YNTHIA looks for a mapping between the old and new datatype definitions and uses heuristics to select a mapping if necessary[12]. This mapping is then applied to the old pattern definition to produce a new one. A full description of these heuristics is beyond the scope of this paper. The general idea is to map base (step) constructors for the source type to target base (step) constructors. For a datatype, t, we say that a constructor is a base constructor if none of its arguments are of type t. A step constructor has at least one such argument. When mapping individual constructors, we have to map the arguments of the constructors. In this case, we map (non-)recursive arguments to (non-)recursive arguments. An argument is recursive if it is of type t and is non-recursive otherwise. To illustrate, consider the datatype definitions for *list*[13] and *tree*:

```
datatype 'a list = nil | :: of 'a * 'a list;

datatype tree = leaf of int | node of tree * tree;
```

For these definitions, *nil* is a base constructor, *leaf* is a base-constructor with a single non-recursive argument, :: is a step constructor with one non-recursive and one recursive argument, and *node* is a step constructor with two recursive arguments. The mapping that C^YNTHIA selects in this case is $nil \mapsto (leaf\ n)$ and $x :: xs \mapsto node(xs, ys)$, where n, ys are fresh variables. ys is introduced in preference to mapping $x :: xs$ to $node(x, xs)$ because x is not of type *tree*. Applying this mapping to the program $app2$ above produces:

```
fun app2 (leaf n) l2 = l2
|   app2 (node((leaf n),ys)) l2 = (x::l2)
|   app2 (node(node(xs,ys),zs)) l2 = x1::(app2 xs l2);
```

Note that x, $x1$ appear on the RHS of the equalities but not on the LHS. Rather than try to replace these terms with a suitable alternative, we prefer to highlight this fact to the user and let him deal with it. In this way, the user is aware of exactly what effect his change of type has had. If x and $x1$ had been replaced it would be difficult to know what to replace them with. In addition, the user might not notice the change.

3.2 Reducing the Number of Programming Errors

One of the main purposes of our experiment was to identify the kinds of programming errors that novice ML users encounter. Our results suggest that the learning rate is severly affected by these errors. The

[12] Although the user may override the heuristics and choose an alternative mapping if necessary.
[13] This is a built-in definition in ML.

most common errors were syntax errors and type errors. C^YNTHIA disallows such errors in its programs. The students found it particularly difficult to pinpoint the source of a type error. Although ML does type checking at compile time and spots type inconsistencies, the system messages provide little or no help in rectifying the problem. C^YNTHIA also incorporates a type checker. The ML type checker is not invoked until compile time. In C^YNTHIA, however, the type checker is called as each new term is entered. Hence, the user receives immediate feedback on whether a term is well-typed. In addition, given the type of the top-level function that the user has already supplied, C^YNTHIA can tell the user what the type of a term should be *before* he enters it. In this way, the number of type errors is reduced considerably. All programs in C^YNTHIA are guaranteed to be well-typed.

A major source of errors in recursive programs is non-termination [4]. An example of a non-terminating function is

```
fun gcd (x:int) y = if x=y then x
                     else gcd (x-y) y;
```

This will not terminate for the call $gcd(2,3)$. Termination errors are less frequent than type errors but are usually more serious. C^YNTHIA restricts the user to producing terminating programs[14]. Checking the termination of function definitions is undecidable. Hence, C^YNTHIA restricts the user to a decidable subclass known as Walther Recursive functions [14]. It is easy to check if a function definition falls into this class and yet the class is wide enough to be of real use to novice (and indeed more experienced programmers). The class includes most commonly occurring examples of course-of-values, nested and multiple recursions. The idea behind Walther Recursion is that when a definition is made, we attempt to place a bound on the size of the output of the function. The size measure used is based on the size of constructor terms where $w(c(u_1,\ldots,u_n)) = 1 + \sum_{i \in R_c} w(u_i)$ if c is a constructor, and R_c is the set of its recursive arguments. Bounding lemmas derived for previous definitions are then used to show that each recursive call is measure-decreasing, and hence that the definition terminates on all inputs. Walther Recursion is sufficiently wide-ranging to allow the following definition of *msort* modified from the example in [14]. Note that this is not the most natural definition of *msort* but is the definition that is easiest to produce using the editing commands in C^YNTHIA.

```
fun msort nil = nil
|   msort (x::nil) = x::nil
|   msort (x::h::t) = merge (msort (evenl (x::h::t)))
                            (msort (x::(evenl (h::t))));
```

evenl returns the elements in a list at even positions. *merge* joins two lists by repeatedly taking the smaller of the two heads. Given the previous bounding lemma, $|evenl\ (z :: zs)| \leq |zs|$ for non-nil inputs, we can derive that both recursive calls decrease the measure.

As mentioned earlier, a common way of defining ML programs is by pattern matching. A source of errors in our analysis was that students wrote programs that were not well-defined – i.e. the pattern for an argument did not exhaustively cover the domain of the datatype with no redundant matches. ML spots such errors at compile time displaying a warning message. Although ML does not consider ill-definedness as an error, it is commonly believed that it is good programming practice to write well-defined programs. Otherwise, there can be serious run-time errors. Students were found to ignore the warnings given by ML because they are not explicitly flagged as errors. We feel, however, that students would make fewer errors if their programs were all well-defined. Hence, in C^YNTHIA the editing commands guarantee well-definedness.

In addition to these guarantees, any program defined in C^YNTHIA is syntactically correct.

4 The User Interface

We are currently developing a graphical user interface for C^YNTHIA written in Tcl/Tk. At present, C^YNTHIA supports a functional subset of the core language of Standard ML, and does not yet support exceptions. The user starts off with a library of function definitions to choose from which have been taken

[14] Occasionally, non-terminating programs can be useful. One could envisage, however, a facility for overriding termination restrictions in this small number of cases.

from the course notes at Napier University. The user can select one of these functions to transform and can add his own functions to the library.

ML programs are presented to the user in a window and 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. 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. CHANGE TERM can be applied to a subterm of an expression as well as the whole expression. This is possible because when the mouse is moved over a function symbol, the subterm which has that function symbol as top-level functor is highlighted and it is this subterm that is changed. Clicking on the right mouse button when a term is highlighted will display the type of the highlighted expression. This is an invaluable way of providing type feedback to the user during program development. Figure 4 is a screenshot of one stage during the transformation of *length*.

By using a graphical interface, the user is completely unaware of the proof machinery that lies behind C^YNTHIA. As far as he is aware, he is editing the ML program.

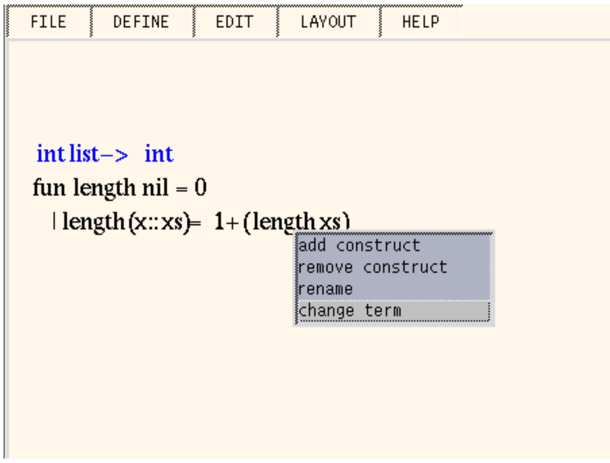

Fig. 4. Graphical user interface to C^YNTHIA

A further development in the interface has been to provide feedback to the user about the next stage of editing. Although the aim is, in general, to produce a valid program at each stage of editing, this is not always possible. Some editing commands will invalidate parts of the program. There are two main ways this can happen. We have already given an example of the first – see $app2$ in §3.1. If a section, S of a program is deleted and a subsequent part of the program, P depends on S then P can no longer be valid. This is what happened when changing type in $app2$ – x and $x1$ were present on the RHS of the equality but not the LHS. The solution we choose to deal with this is to leave such terms in the program but highlight them (by displaying in a different colour) to the user. The user may then inquire why they are highlighted and will be told that unless they change the terms, the program will not be syntactically valid. We prefer this approach to removing the terms because the highlighting process retains as much as possible of the program so that frustrating retyping is not necessary whilst it additionally tells the user exactly which parts of the program must be changed next.

The other situation where this is used is when an editing command causes the program to become ill-typed. In C^YNTHIA every term that is entered is type-checked at entry time. If the term is not of the required type, the user will be told and will not be allowed to enter the term. More generally, if an application of the CHANGE TYPE command makes some part of the program ill-typed, C^YNTHIA will not only highlight the offending term to alert the user but will also tell the user what the type should be so that when he changes the highlighted term he is told what type the new term should belong to. A simple example of this phenomenon is when changing the output type of a program. Consider the *length* example again:

```
'a list -> int

fun length nil = 0
|   length (x::xs) = 1 + (length xs);
```

Suppose the user applies CHANGE TYPE to change *int* to *int list*. Then 0 and $1 + (length\ xs)$ will no longer be of the same type as the output type. Again, rather than removing these terms or attempting to change them automatically, we highlight them to the user by a displaying them in a different colour. The user then has freedom either to change the terms immediately, or to make some other sequence of editing commands that will resolve the problem. As soon as the terms become well-typed, the colouring disappears.

Note that the proofs-as-programs paradigm is a natural way in which to implement this mechanism. No extra checks are needed to highlight terms. Highlighted parts of the program just correspond to proof rules that have failed to apply. Similarly, highlighting of ill-typedness means that a proof obligation to prove the well-typedness has failed to be proved and the user is alerted.

We are currently looking into further ways of providing instructive feedback to the user during program development. One obvious possibility is when using ADD RECURSIVE CALL. If the user enters a term that means that a program no longer terminates, rather than just refusing the command application, the user could be given feedback about why the command cannot be applied, and perhaps what he should do so that it could be.

5 Related Work

The work closest to our own is the *recursion editor* presented in [4]. In fact, this was one of the original inspirations. The recursion editor is an editor for writing terminating programs. Like our system, edits are made by invoking editing commands. $C^{Y}NTHIA$'s commands are more general than those in the recursion editor. In the recursion editor, only a very restricted subset of recursive programs could be produced. The recursion editor is very sensitive to the order in which commands are performed. If they are performed in the wrong order, it may be difficult or impossible to recover. Our proofs-as-programs design overcomes this by allowing greater flexibility because it keeps track of the dependencies within the same program and between different programs. Our proof design also allows us to locate errors in the program easily. [4] makes no consideration of datatypes.

[6] gives an alternative approach to the problem of understanding type errors. He presents a modification of the unification algorithm used in Hindley-Milner type inference which allows the decisions that led to a particular type being inferred to be recorded and fed back to the user. As I have not seen his work in action, I cannot comment on how useful his explanations are in discovering type errors. [13] also looks into ways of providing more information to the user about why particular types have been inferred.

Some work has been done on programming using schemata [10, 7]. This is similar in spirit to our low and high-level commands as the user follows a top-down approach. However, previous attempts are limited to a small range of programs. Our editor is much more general providing a range large enough to be of real practical use. The techniques editor TED [3] features a program transformation perspective but it has no strong theoretical foundations and is therefore much less powerful than $C^{Y}NTHIA$.

We do not address the problem of retrieving a previous example – see [17] which indicates that students tend to solve problems in analogy to the most recent problem they have attempted even though this may not be the best starting point. Although we do not address this issue, our system is at least general enough such that a poor choice of base problem should not prevent a correct, albeit sub-optimal, transformation sequence leading to a solution. As yet, the use of metacognitive tools forcing students to think about their problem solving process has not been very effective [17]. Although $C^{Y}NTHIA$ encourages students to think about such issues, they retain control to explore unconventional paths.

There exist many editors that guarantee syntactic correctness (e.g. [11]). We are aware of no editor that provides the additional guarantees that we do.

6 Conclusion

This paper has presented an editor for producing correct functional programs. It builds upon ideas in [4]. The editor is intended to be a suitable vehicle for novices to learn the language ML. Its high-level commands provide guidance to the user and the user is prevented from making certain kinds of programming error.

Our work can be seen on a number of levels. First, as an educational aid, it provides support for novices learning a language by reducing the effort needed to produce correct programs but without restricting the user to text book solutions. Second, as a support tool for ML, it is a way to quickly edit existing programs without introducing unnecessary bugs. Third, it is an interesting application of ideas from the field of automated reasoning.

C^YNTHIA is due to be tested on ML students at Napier University from September 1997 onwards.

Acknowledgements: We are grateful to Andrew Cumming for help and discussions on the experiment with his ML students. We also thank Paul Brna and Dave Berry for insightful comments on this paper.

References

1. J. R. Anderson, P. Pirolli, and R. Farrel. Learning to program recursive functions. *The Nature of Expertise*, pages 153–183, 1988.
2. S. Bhuiyan, J. Greer, and G. I. McCalla. Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2):115–139, 1994.
3. 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.
4. A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.
5. Th. Coquand. Pattern matching with dependent types. In *Proceedings from the logical framework workshop at Bästad*, June 1992.
6. D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.
7. T.S. Gegg-Harrison. Adapting instruction to the student's capabilities. *Journal of AI in Education*, 3:169–181, 1992.
8. C. Horn and A. Smaill. Theorem proving and program synthesis with Oyster. In *Proceedings of the IMA Unified Computation Laboratory*, Stirling, 1990.
9. 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.
10. M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University, 1989.
11. A. Kohne and G. Weber. Struedi: A lisp-structure editor for novice programmers. In *Human-Computer Interaction (INTERACT 87)*, pages 125–129, 1987.
12. Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.
13. B.J. McAdam. Adding BigTypes to ML. Tech.rep, August 1996. Available at http://www.dcs.ed.ac.uk /home /bjm /summer96 /tech.ps.
14. David McAllester and Kostas Arkoudas. Walther recursion. In *CADE-13*, pages 643–657. Springer Verlag, July 1996.
15. E. Melis and J. Whittle. External analogy in inductive theorem proving. In *Proceedings of KI97, 21st German Conference on Artificial Intelligence*, 1997.
16. P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming. *Canadian Journal of Psychology*, 39:240–272, 1985.
17. G. Weber. Individual selection of examples in an intelligent learning environment. *Journal of AI in Education*, 7(1):3–31, 1996.
18. G. Weber and A. Bögelsack. *Representation of Programming Episodes in the ELM model.* Ablex Publishing Corporation, Norwood, NJ, 1995.
19. J. Whittle. An analysis of errors encountered by novice ML programmers. Tech.rep, University of Edinburgh, 1996. In http: //www.dai.ed.ac.uk/ daidb/ students/ jonathw/ publications.html.
20. J. Whittle, A. Bundy, and H. Lowe. Supporting programming by analogy in the learning of functional programming languages. In *Proceedings of the 8th World Conference on AI in Education*, 1997. Also available from Dept. of Artificial Intelligence, University of Edinburgh.