An ML Editor Based on Proofs-as-Programs

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

Alan Bundy
Richard Boulton
Division of Informatics
University of Edinburgh
Edinburgh EH1 1HN
`bundy,rjb@dai.ed.ac.uk`

Helen Lowe[†]
Dept of Computer Studies
Glasgow Caledonian University
Glasgow G4 0BA
`H.Lowe@gcal.ac.uk`

Abstract

C^YNTHIA is a novel editor for the functional programming language ML in which each function definition is represented as the proof of a simple specification. Users of C^YNTHIA edit programs by applying sequences of high-level editing commands to existing programs. These commands make changes to the proof representation from which a new program is then extracted. The use of proofs is a sound framework for analysing ML programs and giving useful feedback about errors. Amongst the properties analysed within C^YNTHIA at present is termination. C^YNTHIA has been successfully used in the teaching of ML in two courses at Napier University, Scotland. C^YNTHIA is a convincing, real-world application of the proofs-as-programs idea.

1. Introduction

Current programming environments for novice functional programming (FP) are inadequate. This paper describes ways of using mechanised theorem proving to improve the situation, in the context of the language ML [11], a strongly-typed FP language with type inference. Datatypes in ML are defined by a number of constructors which can be used to write patterns which define a function. The most common way to write ML programs is via a text editor and compiler (such as the Standard ML of New Jersey compiler). Such an approach is deficient in a number of ways. Program errors, in particular type errors, are generally difficult to track down. Much conventional debugging of runtime errors is replaced by dealing with compile time error reports which, although one of the strengths of FP, can be frustrating for a new user and can form a barrier to learning FP concepts [16].

[*] formerly University of Edinburgh

[†] First author supported by EPSRC studentship and 519-50-32 NASA Code R Aero IT Base Program, SIPS, Program Synthesis. Third author EPSRC grant GR/L14381. Other authors EPSRC grant GR/M45030

C^YNTHIA is an editor for a subset of ML that provides improved support for novices. Programs are created incrementally using a collection of correctness-preserving editing commands. Users start with an existing program which is adapted by using the commands. This means fewer errors are made. In addition, C^YNTHIA's improved error-feedback facilities enable errors to be corrected more quickly. Specifically, C^YNTHIA provides the following correctness guarantees:

1. syntactic correctness;
2. static semantic correctness, including type correctness as well as checking for undeclared variables or functions, or duplicate variables in patterns etc.;
3. well-definedness — all patterns are exhaustive and have no redundant matches;
4. termination.

Note that, in contrast to the usual approach, correctness-checking is done incrementally. Violations of (1), (3) and (4) can never be introduced into C^YNTHIA programs. (2) may be violated as in general it is impossible to transform one program into another without passing through states containing such errors. However, all such errors are highlighted to the user by colouring expressions in the program text. The incremental nature of C^YNTHIA means that as soon as an error is introduced, it is indicated to the user, although the user need not change it immediately.

In C^YNTHIA, each ML function definition is represented as a proof of a specification of that function, using the idea of proofs-as-programs [6]. As editing commands are applied, the proof is developed hand-in-hand with the program, as shown in Figure 1. The user starts with an existing program and a corresponding initial proof (from an initial library). The edits are actually applied to the proof, giving a new partial proof which may contain gaps or inconsistencies. C^YNTHIA attempts to fill these gaps and resolve inconsistencies. Any which cannot be resolved are fed back to the user as program errors.

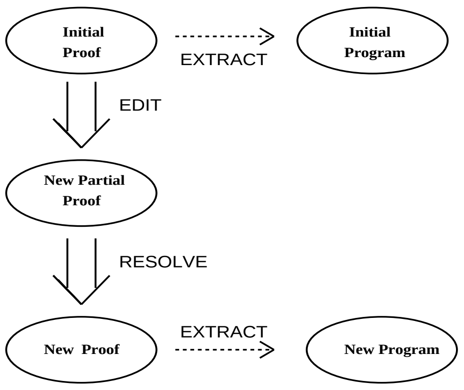


Figure 1. Editing Programs in $C^{Y}NTHIA$.

$C^{Y}NTHIA$'s proofs are written in *Oyster* [3], a proof-checker implementing a variant of Martin-Löf Type Theory [9]. *Oyster* specifications (or conjectures) may be written to any level of detail, but to make the proof process tractable in real-time, $C^{Y}NTHIA$ specifications are restricted severely. Specifications state precisely the type of the function and various lemmas needed for termination analysis (see §3.3). Proofs of such specifications provide guarantees (1)-(4) above. Given this restriction, all theorem proving can be done automatically.

The type systems of *Oyster* and ML are not quite the same. In particular, in ML type-checking is decidable which is not true of *Oyster*. However, it is possible to restrict to a subset of *Oyster*'s types which resembles that of ML very closely. We only consider a functional subset of the Core ML language [16]. In addition, we exclude mutual recursion and type inference. Mutual recursion could be added by extending the termination checker. We made a conscious decision to insist that the user provide type declarations. This is because the system is primarily intended for novices and investigations have shown that students find type inference confusing. Given that edits are done incrementally anyway, providing a type declaration is not too burdensome. A possible future project is to extend $C^{Y}NTHIA$ for expert users. This version would include type inference.

This paper concentrates on the underlying proof framework of $C^{Y}NTHIA$. More details on the editing commands can be found in [15] and details on empirical user evaluations are in [14].

2. An Example of Using $C^{Y}NTHIA$

Consider the task of writing a function, *count*, to count the number of nodes in a binary tree. `itree` is defined by:

```
itree = leaf of int | node of int * itree * itree;
```

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[1] can then use *length* as a starting point. Below we give the definition of *length* preceded by its type:

```
'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*. The user may indicate any occurrence of *length* and invoke the RENAME command to globally change *length* to *count*:

```
'a list -> int
 fun count nil = 0
 |   count (x::xs) = 1 + (count xs);
```

We want to count nodes in a tree so we need to change the type of the parameter. Suppose the user selects `'a list` and invokes CHANGE TYPE to change the type to `itree`. $C^{Y}NTHIA$ propagates this change by changing `nil` to `leaf n` and changing `::` to `node`:

```
itree -> int
fun count (leaf n) = 0
| count (node(x,xs,ys)) = 1 + (count xs);
```

Note that the new patterns are well-defined. A new variable `ys` of type `itree` has been introduced. It remains to alter the results for each pattern. `0` is easily changed to `1` and `1` to `1 + (count ys)` using CHANGE TERM. Note that the recursive call `count ys` was automatically validated as termination-preserving in the application of CHANGE TYPE. It can therefore be selected from a drop-down menu. The final program is:

```
itree -> int
fun count (leaf n) = 1
|   count (node(x,xs,ys)) = 1 +
                 (count ys) + (count xs);
```

$C^{Y}NTHIA$ has other commands too. MAKE PATTERN replaces a variable by a number of patterns — one for each constructor of the datatype. In this way, arbitrarily complex patterns can be built-up and are guaranteed to be well-defined. ADD RECURSIVE CALL allows the user to construct functions with new recursion schemes. $C^{Y}NTHIA$ keeps (and displays) a list of so far valid recursive calls — i.e. recursive calls which may be used in the program without compromising termination. The user may add to this by applying ADD RECURSIVE CALL. $C^{Y}NTHIA$ then checks that this new call maintains termination and if so, makes it available during editing.

3. Representing ML Definitions as Proofs

This section presents the underlying proof engine in $C^{Y}NTHIA$. Note that all the theorem proving is completely hidden from the user so that the user of $C^{Y}NTHIA$ requires no specialised knowledge of logic or proof. We

[1] 'he' will be used throughout to denote a male or female user.

will use an ongoing example to illustrate the ideas — *qsort*, illustrated in Figure 2.[2]

```
(int * int -> bool) -> int -> int list
                            -> int list
fun split f k nil = nil
| split f k (h::t) = if f(h,k)
                     then h::split f k t
                     else split f k t;

int list -> int list
fun qsort nil = nil
| qsort (h::t) = (qsort (split (op <) h t))
      @ [h] @ (qsort (split (op >=) h t));
```

Figure 2. A Version of Quicksort.

We exploit the proofs-as-programs idea to represent each ML function definition as a proof in *Oyster*. The key idea with proofs-as-programs is that, in a constructive proof, the expression $t : T$ can be read in three ways — t has type T, t is a proof for T and t is a program for the task specified by T. This is what enables us to extract programs from proofs. Proofs in *Oyster* are carried out in a goal-oriented way — the specification is the top-level goal. Applications of inference rules produce zero or more sub-goals that are themselves solved by further rule applications. Each inference rule has an associated *extract term*. Gathering together all inference rules gives an extract term for the entire proof.

3.1. Why Use Proofs?

Proofs seem to be a good framework for designing correctness-checking editors. Another possible framework is that of attribute grammars [1, 12], which attach annotations to a language's grammar so that properties can be propagated throughout the abstract syntax tree. Proofs-as-programs wins in two main ways. First, proofs-as-programs gives a sounder theoretical underpinning. The correctness of programs in C^YNTHIA comes from the underlying proof. The soundness of the proof rules is easy to check. In contrast, however, it would be a massive, if not impossible, undertaking to check the correctness of an attribute grammar implementing a C^YNTHIA-like editor. Second, proofs-as-programs seems more suited for functional programming. The proof structure localises the relevant parts of the program — for instance, an induction rule encapsulates the kind of recursion. This means that information is localised rather than being spread across the grammar.

[2] `::` is the ML cons operator for lists. `@` is append.

3.2. Specifications in C^YNTHIA

In general, specifications may specify arbitrarily complex behaviour about a function. However, C^YNTHIA specifications are deliberately rather weak. This is so that the theorem proving task can be automated and so that programs with very different behaviours can be edited one into another without requiring the user to carry out a complicated editing of the specifications. Each C^YNTHIA function specification states precisely the type of the function along with lemmas required for termination analysis. The specification for *split* in Figure 2 is:

$$P : (\forall z_1 : (int * int \to bool) \cdot \forall z_2 : int \cdot \forall z_3 : int\ list \cdot (f\ z_1\ z_2\ z_3) : int\ list\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3) \quad (1)$$

where f represents the name of the function.

There are three parts to this specification. P is a variable representing the definition of the ML function (the extract term). P gets instantiated as the inference rules are applied. The second part of the specification merely states the existence of a function of the given type. Clearly, there are an infinite number of proofs of such a specification. The particular function represented in the proof is given by the user, however, since each editing command application corresponds to the application of a corresponding inference rule. In addition, many possible proofs are outlawed because the proof rules (and corresponding editing commands) have been designed in such a way as to restrict to certain kinds of proofs, namely those that correspond to ML definitions. The final part of the specification states *bounding lemmas* that hold for the function. In this case, there is only one bounding lemma. Bounding lemmas are part of Walther Recursion analysis [10]. They express upper bounds on a function, f, based on a fixed size ordering, w. These upper bounds are useful because they enable us to reason about the measure of a recursive call involving f. In this case, w is the length of a list, so this lemma states that the length of the result of *split* is never greater than the length of its third argument.

C^YNTHIA comes with an initial library of functions and corresponding proofs. The idea is that the user always begins with an existing definition from this library but is free to add any function created using C^YNTHIA.

The specifications are, in fact, dynamic, in the sense that the type may change, or bounding lemmas may be added or removed. A change of type is made in direct response to the application of the CHANGE TYPE command. Bounding lemmas are added (removed) automatically depending on which edits are applied. Fortunately, it turns out that the only command which can affect the validity of the current bounding lemmas is CHANGE TERM so bounding lemma revision (which involves a fresh analysis of the entire function) is only required then.

$$\text{WREFL}\quad \frac{}{H \vdash x \le_w x} \qquad \text{WCONS1}\quad \frac{H \vdash u_i \le_w t}{H, (f \ldots x_i \ldots) \le_w x_i \vdash (f \ldots u_i \ldots) \le_w t}$$

$$\text{WRED}\quad \frac{H \vdash u_i \le_w t}{H, (f \ldots c(\ldots, x_i, \ldots) \ldots) \le_w x_i \vdash (f \ldots c(\ldots, u_i, \ldots) \ldots) \le_w t}$$

$$\text{WCONS2}\quad \frac{\begin{array}{rcl}(\forall i \in R_c)\;\; H & \vdash & u_i \le_w t_i \\ (\forall i \in \{1, \ldots, n\}) & \vdash & i \notin R_c \to (u_i = t_i)\end{array}}{H \vdash c(u_1, \ldots, u_n) \le_w c(t_1, \ldots, t_n)}$$

$$\text{WCONS3}\quad \frac{\begin{array}{rcl}H & \vdash & u \le_w t_i \\ & \vdash & i \in R_c\end{array}}{H \vdash u \le_w c(\ldots, t_i, \ldots)} \qquad \text{WSUBST}\quad \frac{H \vdash (u \le_w t) \bullet \{x_2/x_1\}}{H, Y : x_1 = x_2 \vdash u \le_w t}$$

Figure 3. Rules for Walther Recursion.

3.3. Bounding Lemmas

One of the main correctness guarantees provided by C^YNTHIA is termination. Termination is in general undecidable. One approach would be to provide the user with a pre-defined set of well-founded induction schemes. To use a scheme not specified in this set, the user would then specify an ordering and prove that this ordering is well-founded. Since C^YNTHIA is meant for programmers, not logicians, the user cannot be expected to carry out such tasks. The difficulty in designing C^YNTHIA then is to find a decidable subset of terminating programs that is large enough to include most definitions a (novice) ML programmer might want to create. C^YNTHIA is restricted to such a set, the Walther Recursive functions [10], which includes primitive recursive functions over an inductively-defined datatype, nested recursive functions and some functions with previously defined functions in a recursive call, such as *qsort*. Walther Recursion assumes a fixed size ordering, with a semantics defined by the rules in Figure 3. Intuitively, this ordering is defined as follows: $w(c(u_1, \ldots, u_n)) = 1 + \sum_{i \in R_c} w(u_i)$ where c is a constructor and R_c is the set of recursive arguments of c. In the case of lists, this measure is length. If $c(u_1, \ldots, u_n)$ has type T then the *recursive* arguments of c are the i such that u_i has type T. Other arguments are *non-recursive*.

There are two parts to Walther Recursion — bounding lemma analysis (BLA) and measure argument analysis (MAA). BLA calculates bounding lemmas. MAA checks that each recursive call is measure decreasing. Each time a new definition is made, bounding lemmas are calculated for the definition. These place a bound on the definition based on the fixed size ordering. To guarantee termination, it is necessary to consider each recursive call of a definition and show that the recursive arguments decrease with respect to this ordering. Since recursive arguments may in general involve references to other functions, a measure decrease is guaranteed by utilising previously derived bounding lemmas. There are two kinds of bounding lemmas — reducer and conserver lemmas. Conserver lemmas are of the form $f\ x_1 \ldots x_n \le_w x_i$ for some i. Reducer lemmas are the same but the inequality is strict. $u \le_w t$ iff u and t are well typed, inhabit the same type and the measure of u, $w(u)$, is no larger than $w(t)$. Strict inequality is defined similarly.

The function $split$ satisfies the conserver lemma:

$$split\ f\ k\ z \le_w z$$

This can be proved automatically in C^YNTHIA using the rules given in Figure 3 and induction. In Figure 3, f is an arbitrary function, whereas c is a constructor function.

3.4. Proofs in C^YNTHIA

Oyster's inference rules are somewhat low-level. This makes it awkward to develop the proof and program together because the two entities are at different levels of abstraction. In addition, there is some ambiguity in the correspondence when considering real ML programs. The same ML construct could be implemented in the proof by a variety of different combinations of inference rules. In order to obtain a cohesion between proof and program, derived rules and tactics were added to *Oyster* which both raise the level of discourse within the proof and also make design choices which resolve ambiguities.

We present (part of) the proof for *split*. We introduce the derived rules as they are needed. The choice of which rule to

1. Find measure arguments, M, for f by considering each x_i in turn and applying the rules in Figure 3;

2. **if** $M = \{\}$, termination analysis fails.
   else for each recursive call, $f\ u_1 \ldots u_n$, try to find an $m \in M$ such that $u_m <_w x_m$ — i.e. if x_m is a constructor term $c(\ldots, r_j, \ldots)$, we need $u_m \leq_w r_j$ for some j.
   if this can be done for all recursive calls, then f terminates.
   else termination analysis fails

Figure 4. Procedure for Checking Termination.

apply is taken by the user, although indirectly, by choosing which editing command to apply. Each editing command corresponds to applying (or modifying) a set of inference rules. For example, the MAKE PATTERN command modifies an induction rule.

The first step in the proof of *split* is to apply the I-∀ rule (a standard *Oyster* rule) backwards to the specification (1). After three applications of this rule, the goal looks like:

$$z_1 : (int * int \rightarrow bool), z_2 : int, z_3 : int\ list \vdash$$
$$P_1 : ((f\ z_1\ z_2\ z_3) : int\ list \wedge (f\ z_1\ z_2\ z_3) \leq_w z_3)$$

where P has been instantiated to $\lambda z_1 \cdot \lambda z_2 \cdot \lambda z_3 \cdot P_1$.

The pattern matching in *split* would have been introduced by applying MAKE PATTERN to the third argument. In proof terms, this corresponds to an induction over z_3. In C^YNTHIA, induction rules are dynamically created as they are needed. This process is driven by the user via the application of the MAKE PATTERN and ADD RECURSIVE CALL commands. Each ML datatype definition gives rise to a well-founded "primitive" induction rule which is added to C^YNTHIA when the datatype is defined. This only allows proofs based on this primitive induction rule. Non-primitive inductions can be added using MAKE PATTERN. Recall that MAKE PATTERN splits a variable into cases, one for each constructor of the datatype. Internally, MAKE PATTERN has a similar effect on the current induction rule in a proof. For example, if MAKE PATTERN is used to convert patterns `nil` and `h::t` into `nil`, `h::nil` and `h::h1::t`, then the corresponding induction rule will now have three antecedents — two base cases and one step case. Termination of this new induction rule is guaranteed because the induction hypotheses (corresponding to recursive calls in the program) are over a strict subterm of the induction term (pattern over which the function is defined).

More interesting recursive calls, not necessarily based on the structure of the defining patterns, are introduced by ADD RECURSIVE CALL. The effect on the induction rule is to add an induction hypothesis corresponding to the new recursive call. Since arbitrary recursive calls can be input by the user, there is no guarantee that the resulting induction rule is well-founded. This is where the second part of Walther Recursion (measure argument analysis) comes in. We digress from the proof of *split* to describe measure argument analysis.

3.5. Measure Argument Analysis

Definition 1 *Given a function f, defined over arguments $x_1, \ldots, x_n$, the set of measure arguments is the set of i such that for every recursive call $f\ u_1 \ldots\ u_n$ of f, $u_i \leq_w x_i$.*

MAA involves showing that the measure decreases over each recursive call. To check for termination, the procedure in Figure 4 is adopted.

For our quicksort example, Figure 2, there are two applications of ADD RECURSIVE CALL each introducing a recursive call. In attempting to derive $u_m <_w x_m$, it is necessary to use previously defined bounding lemmas. In this example $M = \{1\}$, since $split(op\ \ <)\ h\ t \leq_w t$ and $split\ (op\ \ >=)\ h\ t \leq_w t$. Since $t \leq_w h :: t$, termination is proved. These additional proof obligations are factored into the induction rule. Hence, in the general case, the induction rule also captures the proof obligations for measure argument analysis. The induction rule is well-founded as long as these obligations can be proved.

In [10], Walther Recursion was described for a small functional language with a syntax and semantics different to that of ML. We made extensions to encompass the subset of ML supported by C^YNTHIA. The major changes are as follows: in the language in [10] definitions are made using destructors. It is more natural to use constructors in ML. Therefore, the rules were recast in constructor-fashion; [10] suggests a forward application of the rules. C^YNTHIA is based on a backwards style so our system sets up subgoals for each possible lemma and then applies the rules in a backwards fashion; a function defined by an exhaustive pattern cannot be a reducer because the measure of the base case argument cannot be reduced. [10] forces the user to make an additional definition, restricted to non-base-cases. It is naive to expect programmers to go through this process of making additional definitions. Our solution is to automatically place side-conditions on reducer lemmas that rule out base cases. This allows the user to write definitions as normal; [10] does not include ML `case` expressions or local function declarations. It does allow local variable declarations but only of the form $dec = exp$ where dec is a variable. In $C^{Y}NTHIA$, dec may be a pattern.

We now return to our proof of *split*. By applying the primitive list induction rule for lists, P_1 is instantiated to:

$$ind(z_3, a_b, \lambda h \cdot \lambda t \cdot a_s)$$

where ind is a function returning its second argument if z_3 is an empty list and its third argument otherwise. The induction rule gives rise to two subgoals. Consider the base case first:

$$\ldots \vdash a_b : ((f\ z_1\ z_2\ nil) : int\ list \wedge (f\ z_1\ z_2\ nil) \leq_w nil)$$

The base case continues by applying the WITNESS rule which instantiates a_b to nil. The role of WITNESS is to provide explicit instantiations of extract terms. This instantiation is given by the user using the command CHANGE TERM. The WITNESS rule is defined as follows:

$$\frac{\begin{array}{lcl} H & \vdash & e : T_0 \\ H & \vdash & e \in \Sigma \\ H & \vdash & A \bullet \{e/(f\ x_1 \ldots x_n)\} \end{array}}{\begin{array}{lcl} H & \vdash & e : ((f\ x_1 \ldots x_n) : T_0 \wedge A) \end{array}}\ \text{WITNESS}$$

where Σ is the set of static semantically valid expressions. WITNESS is a derived rule added to *Oyster* to raise the level of the proof. The application of WITNESS instantiates e with nil giving three subgoals:

$$\ldots \vdash nil : int\ list \qquad \ldots \vdash nil \in \Sigma \qquad \ldots \vdash nil \leq_w nil$$

The first two subgoals are proved easily using tactics for type-checking and semantics-checking respectively. The third is proved using WREFL.

The step case subgoal is as follows:

$$\begin{array}{l} \ldots, h : int, t : int\ list, \\ \quad (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \qquad \vdash a_s : ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge \\ \qquad\qquad (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)) \end{array}$$

The definition of *split* includes a conditional statement. The corresponding proof notion is captured by another of $C^{Y}NTHIA$'s derived rules, IF:

$$\frac{\begin{array}{rcl} H & \vdash & e_1 : bool \\ H, C : e_1 & \vdash & e_2 : A \\ H, C : \neg e_1 & \vdash & e_3 : A \\ H & \vdash & e_1 \in \Sigma \end{array}}{\begin{array}{rcl} H & \vdash & (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) : A \end{array}}\ \text{IF}$$

The user provides the condition over which the casesplit is made with the command ADD CONSTRUCT(IF THEN ELSE). This instantiates a_s to `if` $z_1(h, z_2)$ `then` E_2 `else` E_3 and gives four subgoals. Type-checking and semantics-checking are done easily. The other two subgoals correspond to each branch of the conditional split. Let us consider the first branch only. The subgoal in this branch is:

$$\begin{array}{l} \ldots, C : z_1(h, z_2), (f\ z_1\ z_2\ t) : int\ list, \\ \quad X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \quad \vdash E_2 : ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge \\ \qquad (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)) \end{array}$$

Now we apply WITNESS, instantiating E_2 to $h :: (f\ z_1\ z_2\ t)$. Again, type-checking and semantics-checking are dealt with easily. The remaining subgoal is:

$$\begin{array}{l} \ldots, C : z_1(h, z_2), (f\ z_1\ z_2\ t) : int\ list, \\ \quad X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \quad \vdash (h :: (f\ z_1\ z_2\ t)) : int\ list\ \wedge \\ \qquad (h :: (f\ z_1\ z_2\ t)) \leq_w (h :: t) \end{array}$$

There are two conjuncts to prove. The first is trivial. The second needs to be proved using the rules for Walther Recursion and an induction hypothesis. First, apply WCONS2. This gives the subgoal:

$$\ldots \vdash (f\ z_1\ z_2\ t) \leq_w t$$

which is proved by the induction hypothesis, X_1.

The second branch of the conditional statement can be proved similarly. Collecting together all the instantiations, P has been instantiated to:

$$\begin{array}{l} \lambda z_1 \cdot \lambda z_2 \cdot \lambda z_3 \cdot ind(z_3, nil, \lambda h \cdot \lambda t \cdot \\ \qquad \texttt{if}\ z_1(h, z_2) \\ \qquad \texttt{then}\ h :: (f\ z_1\ z_2\ t) \\ \qquad \texttt{else}\ (f\ z_1\ z_2\ t)) \end{array}$$

A simple translation, along with a mechanism for keeping track of variable names, gives the program *split*.

3.6. Replaying Proofs According to User Edits

The previous section gave the complete proof representation for $split$. The user develops the structural parts of such proofs using the editing commands. $C^{Y}NTHIA$ automatically carries out the correctness checking parts of the proof which includes some search. The user never has to develop an entire program (proof) from scratch since a library component is chosen as the starting point. As editing commands are applied to this starting definition, the proof is updated and maintained (e.g., some bounding lemmas may need to be revised).

Definition 2 *The Abstract Rule Tree (ART) of a proof is the tree of rule applications, where the hypotheses list, goal etc. have been omitted.*

The procedure for editing the proof is as follows. The user highlights the position in the program where he wishes to make a change. $C^{Y}NTHIA$ calculates the corresponding position, pos, in the proof tree. Let the synthesis proof be denoted by P_t and the proof subtree below pos by P_s. $C^{Y}NTHIA$ abstracts P_s into an ART A_s. $C^{Y}NTHIA$ then makes changes to A_s to give $\phi(A_s)$. $\phi(A_s)$ is then unabstracted or replayed to give the new proof subtree $\phi(P_s)$. The complete new proof tree is then P_t with P_s replaced by $\phi(P_s)$. Note that $C^{Y}NTHIA$ abstracts only P_s and not the whole proof tree P_t. This saves effort because, due to the refinement nature of the proofs, any rules not in P_s will be unaffected.

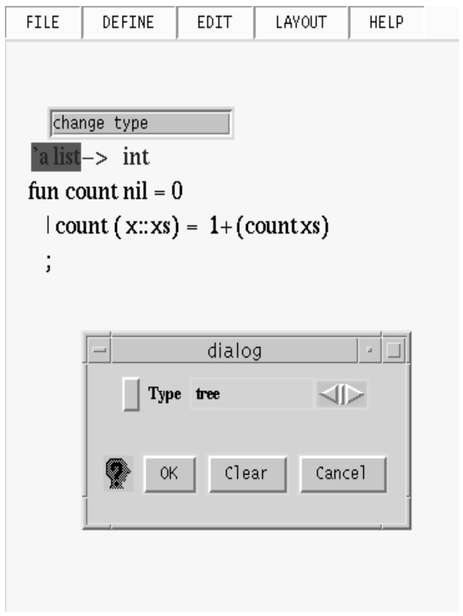
The replay of the ART is the main method for propagating changes throughout the proof. In addition, some commands also require a change to the specification. For example, ADD CURRIED ARGUMENT adds an additional type to the specification. The ART captures the dependencies between remote parts of the program and the replay of the ART updates these dependencies in a neat and flexible way. Changes to the program will mean that some of the previous subproofs no longer hold. In some cases, the system can produce a new proof. However, it may be that a subgoal is no longer true. Such subgoals correspond directly to errors in the program. The replay of the ART is a powerful mechanism for identifying program errors and highlighting them to the user. During the replay, if a rule no longer holds, a gap will be left in the proof. This corresponds to a position in the ML program and so the program fragment corresponding to where the proof failed can be highlighted to the user. This failed proof rule usually denotes a type error or other kind of semantic error (e.g. unbound variable). The replay of the ART and the resolution of inconsistencies is given by the RESOLVE step in Figure 1.

Various optimizations have been implemented to improve the efficiency of the ART replay. Correctness-checking rules can be time-consuming and so $C^{Y}NTHIA$ selectively replays these rules. $C^{Y}NTHIA$ automatically decides which correctness-checking rules need to be replayed according to which editing command was applied.

4. Evaluating $C^{Y}NTHIA$

$C^{Y}NTHIA$ has been successfully evaluated in two trials at Napier University. The first trial involved a group of 40 postgraduates learning ML as part of a course in Formal Methods. The second trial involved 29 Computer Science undergraduates. Full results of these trials can be found in [16]. Although some semi-formal experiments were undertaken, most analysis was done informally.

It is important to note that $C^{Y}NTHIA$ is generally fairly easy to use. The user requires no knowledge of the underlying proof framework to edit programs — the interface is similar to a structure editor (see Figure 5) in which all the proof rules are completely hidden.



```
'a list-> int
fun count nil = 0
  | count (x::xs) = 1+(countxs)
  ;
```

Figure 5. GUI for $C^{Y}NTHIA$

The evaluations suggested the following trends:

- Students make fewer errors when using $C^{Y}NTHIA$ than when using a traditional text editor.
- When errors are made, users of $C^{Y}NTHIA$ locate and correct the errors more quickly. This especially applies to type errors.
- $C^{Y}NTHIA$ discourages aimless hacking. The restrictions imposed by the editing commands mean that students are less likely, after compilation errors, to blindly change parts of their code.
- $C^{Y}NTHIA$ encourages a certain style of programming. This style is generally considered to be a good starting point for learning functional programming. The editing commands correspond to FP concepts and hence discourage, for example, attempts to program procedurally.

5. Related Work

The idea for $C^{Y}NTHIA$ grew out of work on the *recursion editor* [2], an editor for Prolog that only allows terminating definitions. The recursion editor was severely restricted, however, to a much smaller class of terminating programs. It also had $C^{Y}NTHIA$-like transformations but these were stored as complex rewrite rules, the correctness of which had to be checked laboriously by hand. The use of a proof to check correctness eliminates the possibility of error in such soundness-checking.

Attribute grammars were mentioned in §3.1 as an alternative framework for designing editors. No ML editors have been produced using attribute grammars. A couple of other ML editors have recently become available, however. MLWorks [5] and CtCaml [13] have different objectives than C^YNTHIA. MLWorks is an integrated environment for ML with no structure-editing facilities or advanced correctness-checking. CtCaml is a structure editor for another dialect of ML. Its structure editing is primitive, however, in contrast to C^YNTHIA's specially designed commands. C^YNTHIA offers incremental correctness-checking whereas MLWorks users must compile their programs to receive feedback.

Two proof environments are worth mentioning. ALF [8] has a similar design to C^YNTHIA. Proofs are derived by structure editing whilst maintaining termination and pattern exhaustivity. ALF is meant to be a proof environment rather than a programming environment, however. So the goals of C^YNTHIA and ALF are somewhat different. The fact that C^YNTHIA was to be used by novices meant that any proof details must be hidden. This is not the case with ALF. Extended ML [7] is a framework for the formal development of correct programs in the Standard ML language. It is a simple extension of Standard ML in which, in addition to code, the user may also express properties of this code in a logic which is a superset of Standard ML. The use of Extended ML requires knowledge of formal specification techniques because the proof is not hidden from the user.

6. Conclusions

This paper has presented C^YNTHIA, a novel environment for writing ML programs, primarily aimed at novices. The user writes ML programs by applying correctness-preserving editing commands to existing programs. Each ML definition is represented as the proof of a simple specification which guarantees various aspects of correctness, including termination. The use of an underlying proof provides a sound framework in which to analyse and provide feedback on users' programs. The proof checking is fully automatic and hidden from the user. C^YNTHIA has been successfully tested on novice ML students.

C^YNTHIA provides a framework for carrying out more sophisticated analysis than is done at present. This could be done by expressing additional properties in the specification of the proof. Clearly, the proof of such specifications could be arbitrarily hard, but the proofs could still be done automatically if only certain properties or restrictions were considered and proof strategies for these were implemented.

References

[1] H. Alblas and B. Melichar. Attribute grammars, applications and systems. In *International Summer School*, Prague, June 1991. Springer-Verlag. LNCS v. 545.

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

[3] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th CADE*, pages 647–648. Springer-Verlag, 1990. LNAI 449.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, 1982.

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

[6] 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.

[7] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Computer Science*, 173:445–484, 1997.

[8] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer Verlag, 1994. LNCS 806.

[9] 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, 1982.

[10] David McAllester and Kostas Arkoudas. Walther recursion. In M. A. McRobbie and J. K. Slaney, editors, *13th CADE*, pages 643–657. Springer Verlag LNAI 1104, July 1996.

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

[12] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

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

[14] J. Whittle. Improving functional programming environments. In *INTERACT-99, IFIP Conference on Human-Computer Interaction*, 1999.

[15] J. Whittle, A. Bundy, and H. Lowe. An editor for helping novices to learn Standard ML. In *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 389–405, 1997.

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