An ML Editor Based on Proofs-as-Programs

Jon Whittle[1], Alan Bundy[1], Richard Boulton[1], and Helen Lowe[2]

[1] Division of Informatics, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland.
[2] Dept of Computer Studies, Glasgow Caledonian University, City Campus, Cowcaddens Road, Glasgow G4 0BA, Scotland.
jonathw@dai.ed.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.

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 [9]. ML is a strongly-typed FP language with type inference [4]. ML incorporates extensive use of pattern matching. Datatypes 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. For novices, the lack of debugging support forms a barrier to learning FP concepts [14].

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 the using the commands. This means fewer errors are made. 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 mutually exhaustive and have no redundant matches;
4. termination.

Note that, in contrast to the usual approach, correctness-checking is done incrementally. Errors (1), (3) and (4) can never be introduced into C^YNTHIA programs. (2) may be introduced 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 program 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 given in Fig. 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.

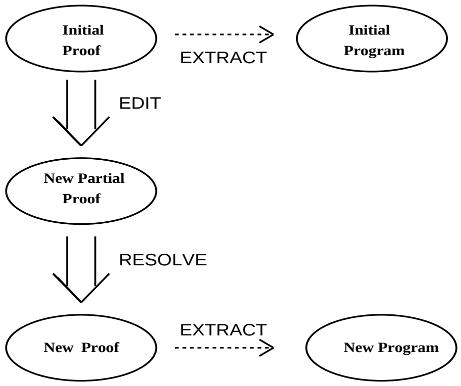

Fig. 1. Editing Programs in C^YNTHIA.

C^YNTHIA's proofs are written in *Oyster* [3], a proof-checker implementing a variant of Martin-Löf Type Theory [7]. *Oyster* specifications (or conjectures) may be written to any level of detail, but to make the proof process tractable in real-time, C^YNTHIA specifications are restricted severely. Specifications state precisely the type of the function and various lemmas needed for termination analysis (see §3.1). 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 [14]. 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 [14]. 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.

2 An Example of $C^{Y}NTHIA$ in Action

Fig. 2 shows an example of an interaction with $C^{Y}NTHIA$. The datatypes `exp` and `statement` and the function `unparse_exp` are already defined. They represent the abstract syntax of a simple imperative programming language. `unparse_exp` is an unparser for expressions. Suppose the user wishes to modify this function into a function, `unparse_st`, to unparse statements. `unparse_st` can be generated by applying a sequence of $C^{Y}NTHIA$'s edits to `unparse_exp`.

The first thing to do is to apply RENAME to any occurrence of `unparse_exp`. The user specifies a new name, `unparse_st`, and $C^{Y}NTHIA$ carries out a global rename. More interesting is the command CHANGE TYPE. In general, when changing type from T_1 to T_2, $C^{Y}NTHIA$ finds a mapping between the constructors of T_1 and those of T_2. In this example, $C^{Y}NTHIA$ finds the mapping:

`Var` $\mapsto$ {`Empty`}, `Const` $\mapsto$ {`Assign`}, `Op` $\mapsto$ {`Cond, While, Block`}

Many possible mappings could have been found, but $C^{Y}NTHIA$ restricts to mappings which map (non-)recursive constructors to (non-)recursive constructors. In addition, each constructor of type T_2 must have a pre-image. This guarantees that the new patterns produced by CHANGE TYPE are well-defined. Note how $C^{Y}NTHIA$ produces a well-defined set of patterns for `statement`.

$C^{Y}NTHIA$ finds a similar mapping for the arguments of each constructor. In some cases, fresh variables may have to be introduced (e.g. the clause for `Assign`), or variables may be dropped (e.g. the clause for `While`).

After the application of CHANGE TYPE, the definition of `unparse_st` contains errors. $C^{Y}NTHIA$ highlights these to the user in different colours. In this paper, boxes denote type errors and circles denote other semantic errors. The user may now use these annotations as a guide to finish the definition. Consider the `While` clause, immediately after CHANGE TYPE is applied.

`unparse_st (While(s,e1))=unparse_st e1 ^“ “ ^` $\boxed{\texttt{s}}$ `^ “ “^ unparse_st e2`

$C^{Y}NTHIA$ tells the user that there are two errors here. By using $C^{Y}NTHIA$'s type inspection facility, the user may highlight `s` and discover that the reason for the type error is that `s` has type `exp`. To rectify this, the user applies CHANGE TERM to replace the boxed occurrence of `s` with `unparse_exp s`. `e2` is circled because it is not declared. In response, the user invokes CHANGE TERM to replace it by `e1`. The expression now contains no errors but to give the correct result, the user replaces `unparse_st e1` by “`while` “ and introduces “`do` “.

The user may add further ML constructs by using the command ADD CONSTRUCT. The final stage of writing `unparse_st` involves using this command twice — once to add a local variable declaration and once to add a conditional statement. The user specifies the parameters to `let val` and `if` and then uses CHANGE TERM to make any further modifications.

```
datatype exp = Var of string | Const of string | Op of exp * string * exp;
datatype statement = Empty | Assign of string * exp | Cond of exp * statement * statement |
                     While of exp * statement | Block of statement * statement;

     exp -> string
     fun unparse_exp (Var t) = t
     |  unparse_exp (Const t) = t
     |  unparse_exp (Op(e1,s,e2)) = unparse_exp e1 ^ " " ^ s ^ " " ^ unparse_exp e2;
```

RENAME,
CHANGE TYPE

```
        statement -> string
        fun unparse_st Empty = t
        |  unparse_st (Assign(t,e)) = t
        |  unparse_st (Cond(s,e1,e2)) = unparse_st e1 ^ " " ^ s ^ " " ^ unparse_st e2
        |  unparse_st (While(s,e1)) = unparse_st e1 ^ " " ^ s ^ " " ^ unparse_st e2
        |  unparse_st (Block(e1,e2)) = unparse_st e1 ^ " " ^ s ^ " " ^ unparse_st e2;
```

CHANGE TERM (multiple times)

```
        statement -> string
        fun unparse_st Empty = ""
        |  unparse_st (Assign(t,e)) = t
        |  unparse_st (Cond(s,e1,e2)) = unparse_st e1 ^ " " ^ s ^ " " ^ unparse_st e2
        |  unparse_st (While(s,e1)) = "while " ^ unparse_exp s ^ "do " ^ unparse_st e1
        |  unparse_st (Block(e1,e2)) = "begin " ^ unparse_st e1 ^ "; " unparse_st e2 ^ " end";
```

ADD CONSTRUCT (LET VAL),
ADD CONSTRUCT (IF),
CHANGE TERM

```
  statement -> string
  fun unparse_st Empty = ""
  |  unparse_st (Assign(t,e)) = t ^ " := " ^ unparse_exp e
  |  unparse_st (Cond(s,e1,e2)) = let val (ss:string) = "if " ^ unparse_exp s ^ "then " ^ unparse_st e1
                                  in if e2 = Empty then ss
                                                   else ss ^ "else " ^ unparse_st e2
                                  end
  |  unparse_st (While(s,e1)) = "while " ^ unparse_exp s ^ "do " ^ unparse_st e1
  |  unparse_st (Block(e1,e2)) = "begin " ^ unparse_st e1 ^ "; " unparse_st e2 ^ " end";
```

Fig. 2. An Unparser for Statements.

CYNTHIA 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. *CYNTHIA* keeps (and displays) a list of currently 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 and specifying a new recursive call. *CYNTHIA* then checks that this new call maintains the termination property and if so, makes it available during editing. For further details about *CYNTHIA*'s editing commands, see [13].

3 Representing ML Definitions as Proofs

This section presents the underlying proof engine in C^YNTHIA. Note that all the theorem proving is completely hidden from the user so that the user of C^YNTHIA requires no specialised knowledge of logic or proof. We will use an ongoing example to illustrate the ideas — the representation of `qsort`, illustrated in Fig. 3.[1]

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



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

Fig. 3. A Version of Quicksort.

3.1 Termination Analysis

One of the main correctness guarantees provided by C^YNTHIA is termination. Termination is in general undecidable. Hence, the usual approach is 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 must specify an ordering and prove that this ordering is well-founded. Since C^YNTHIA is meant for programmers, not logicians, the user must not be expected to carry out such theorem proving. 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 may want to create. The set of Walther Recursive functions [8] is such a set. C^YNTHIA restricts the user to this set which includes primitive recursive functions over an inductively-defined datatype, multiple recursive functions, nested recursive functions and functions that reference 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 Fig. 6. 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[2]. In the case of lists, this measure is just length.

There are two parts to Walther Recursion — reducer / conserver (RC) analysis and measure argument (MA) analysis. Every time a new definition is made, reducer / conserver lemmas are calculated for the definition. These place a bound

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

[2] If $c(u_1, \ldots, u_n)$ has type T then the recursive arguments of c are the i such that u_i also has type T. A constructor is a step constructor if at least one of its arguments is recursive, and is a base constructor otherwise.

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 RC lemmas. The distinction between reducer and conserver lemmas is given as follows. First, define the semantics of the inequality operator.

Definition 1. *$u \leq_w t$ if the following conditions hold:*
- *If u is well typed then t is well typed.*
- *If u is well typed then the top level constructor of u is either a base constructor or the same as the top level constructor of t.*
- *If u is well typed then the measure of u, $w(u)$, is no larger than the measure of t, $w(t)$.*

Define strict inequality in a similar way.

Reducer / Conserver Analysis

Definition 2. *A function f is a reducer on its ith argument if*

$$f\ x_1 \dots x_n <_w x_i \tag{1}$$

and a conserver on its ith argument if

$$f\ x_1 \dots x_n \leq_w x_i \tag{2}$$

To simplify the analysis, $<_w$ can be eliminated by rewriting (1) as:

$$f\ x_1 \dots c_j(\dots, r_{j,k}, \dots) \dots x_n \leq_w r_{j,k} \tag{3}$$

where c_j is a constructor and $r_{j,k}$ is a recursive argument of c_j. This means that only one form of inequality is ever present.

RC analysis is done each time a definition is made.

Consider `partition`. It satisfies the conserver lemma:

$$\texttt{partition f k z} \leq_w \texttt{z} \tag{4}$$

This is proved by the rules in Fig. 6 and induction.

Measure Argument Analysis

Definition 3. *Given a function f, defined over arguments $x_1, \dots, x_n$, the set of measure arguments is the set of i such that for every recursive call $f\ u_1 \dots u_n$ of f, $u_i \leq_w x_i$.*

1. Find measure arguments, M, for f by considering each x_i in turn and applying the rules in Fig. 6;
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

Fig. 4. Procedure for Checking Termination.

Measure argument (MA) analysis involves showing that the measure decreases over each recursive call. To check for termination, the procedure in Fig. 4 is adopted.

In attempting to derive $u_m <_w x_m$, it may be necessary to use previously defined RC lemmas. Consider `qsort`. In this example $M = \{1\}$, since `partition (op <) h t` $\leq_w$ `t` and `partition (op >=) h t` $\leq_w$ `t`. Since `t` $\leq_w$ `h::t`, termination is proved.

It is worth pointing out that for the measure argument analysis to guarantee termination, the function must be defined by a well-defined pattern.

In [8], 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 $C^Y\!NTHIA$. The major changes were:

- In the language in [8] definitions are made using destructors. It is more natural to use constructors in ML. Therefore, the rules were recast in constructor-fashion.
- McAllester suggests a forward application of the rules. $C^Y\!NTHIA$ 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. McAllester 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. A better solution is to place side-conditions on reducer lemmas that rule out base cases. This allows the user to write definitions as normal.
- [8] 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.

3.2 Specifications

Each ML function is represented by a proof with specification (i.e. top-level goal) that is precisely the type of the function along with lemmas required for termination analysis. In general, such specifications may specify arbitrarily complex behaviour about the function. However, $C^Y\!NTHIA$ specifications are deliberately rather weak so that the theorem proving task can be automated. C^YNTHIA specifications are defined as follows.

Definition 4. *A C^YNTHIA specification of an ML function is of the form:*

$$\begin{aligned} P : (\forall z_1 : T_1. \ \ldots \forall z_n : T_n. \ & (f\ z_1 \ldots\ z_n) : T_0 \ \wedge \\ & (f\ z_1 \ldots z_n) \leq_w z_{i_1} \wedge \ldots \wedge (f\ z_1 \ldots z_n) \leq_w z_{i_r} \ \wedge \\ & (f\ z_1 \ldots c_{j_1}(\ldots, r_{j_{1,k}}, \ldots) \ldots z_n) \leq_w r_{j_{1,k}} \wedge \ldots \\ & \ldots \wedge (f\ z_1, \ldots, c_{j_s}(\ldots, r_{j_{s,k}}, \ldots) \ldots z_n) \leq_w r_{j_{s,k}}) \end{aligned} \tag{5}$$

where:
f represents the name of the function[3]*;*
$T_1 \to \ldots \to T_n \to T_0$ is the type of the function;
P is a variable representing the definition of the ML function. P gets instantiated as the inference rules are applied. A complete proof instantiates P to a complete program. This is a standard approach to extracting programs from proofs;
$c_{j_1}, \ldots, c_{j_s}$ are constructors;
$i_1, \ldots, i_r \in \{1, \ldots, n\}$.

The first part of the specification merely states the existence of a function of type $T_1 \to \ldots \to T_n \to T_0$. 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 second part of the specification states RC lemmas that hold for the function.

In the example, the specification for `partition` is:

$$\begin{aligned} P : (\forall z_1 : (int * int \ \to bool). \forall z_2 : int. \forall z_3 : int\ list. \\ (f\ z_1\ z_2\ z_3) : int\ list \ \wedge \ (f\ z_1\ z_2\ z_3) \leq_w z_3) \end{aligned}$$

C^YNTHIA specifications are in fact dynamic — in the sense that as edits are applied, the specification may be changed to reflect the modifications.

3.3 Inference Rules

Each ML function definition is represented by a proof of the relevant specification. There are three kinds of inference rules used in these proofs. Fig. 5 gives rules that mirror the structure of the ML definition. Each program construct has a corresponding inference rule. When the user introduces a construct using the editing commands, the appropriate inference rule is applied to the current goal in the proof. Fig. 5 omits the rules for the ML constructs `fn` and `case` —

[3] In this paper, f is given in curried fashion. Either curried or uncurried is allowed.

see [14]. As each rule is applied, the variable which represents the program (P in (5)) is gradually instantiated. Rules are written in sequent calculus fashion.

WITNESS is similar to the usual $\exists$R rule. LET FUN introduces a local function into the program. In proof terms, this corresponds to a lemma stating the existence of a function f of type $T_1 \to \ldots \to T_n \to T_0$ satisfying certain RC lemmas. IND is a super-rule setting up an induction corresponding to the recursion in the program and also setting up an induction to show the termination of this recursion scheme. $a_{b_1}, \ldots, a_{b_n}$ are base cases. $u_1, \ldots, u_n$ are therefore non-recursive arguments. For the sake of clear presentation, each constructor c_{b_i} is restricted to have only one argument. $a_{s_1}, \ldots, a_{s_n}$ are step cases. Each v_{ij} is a recursive argument. Again, we restrict to just two arguments.

There are two things going on with the IND rule. Firstly, subgoals are set up to carry out measure argument analysis — i.e. check that the recursive calls $R_{s_{ij}}$ are measure decreasing. This is true as long as each $R_{s_{ij}}$ is measure preserving on a strict subexpression of the pattern over which recursion is defined. Secondly, IND carries out an induction to show that the RC lemmas in the specification hold. The induction scheme is based on the patterns over which the ML function is defined. For each pattern $c_{s_i}(v_{i1}, v_{i2})$, the induction hypotheses state that the property A holds for v_{i1} and v_{i2}.

Once a proof is completed, the ML program represented by it can be extracted easily. For rules WITNESS, IF, LET VAL and LET FUN, the extract is precisely the instantiation of P. For IND, we need a simple translation from the *ind* function to an ML function definition using patterns.

The second kind of rules are rules for type-checking and checking that a term inhabits Σ. The third kind are rules for Walther Recursion analysis. These are given in Fig. 6. WSUBST is needed to make substitutions of local variables. The equality on the LHS, below the line, is introduced by the LET VAL rule. C^YNTHIA actually includes a more general version of WSUBST where equalities of the form $(x_1, x_2) = (u_1, u_2)$ are decomposed into $x_1 = u_1$ and $x_2 = u_2$.

An example of rule application may be illustrative. Consider the `partition` example again. After the usual $\forall$R rule has been applied to the specification a number of times, the goal looks like:

$$\begin{gathered} z_1 : (\mathit{int} * \mathit{int} \to \mathit{bool}), z_2 : \mathit{int}, z_3 : \mathit{int\ list} \vdash \\ P_1 : ((f\ z_1\ z_2\ z_3) : \mathit{int\ list}\ \wedge\ (f\ z_1\ z_2\ z_3) \leq_w z_3) \end{gathered}$$

where P has been instantiated to $\lambda z_1.\lambda z_2.\lambda z_3.P_1$. IND now applies. In this case, the form of the IND rule used is as follows:

$$\frac{\begin{array}{l} H \vdash a_{b_1} : ((f\ z_1\ z_2\ \mathtt{nil}) : \mathit{int\ list}\ \wedge\ (f\ z_1\ z_2\ \mathtt{nil}) \leq_w \mathtt{nil}) \\ H, h : \mathit{int},\ t : \mathit{int\ list}, (f\ z_1\ z_2\ t) : \mathit{int\ list}, X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \quad \vdash a_{s_1} : ((f\ z_1\ z_2\ (h :: t)) : \mathit{int\ list}\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t) \end{array}}{H, z_3 : \mathit{int\ list} \vdash (\mathit{ind}(z_3, a_{b_1}, \lambda h.\lambda t.\lambda X_1.a_{s_1}(t))) : (f\ z_1\ z_2\ z_3) \wedge (f\ z_1\ z_2\ z_3) \leq_w z_3}$$

This rule mirrors the structure of the patterns in the definition of `partition` — i.e. there is a case for `nil` and a case for `h::t`. It checks that the recursive call is measure decreasing ($t \leq_w t$). It also tries to prove the RC lemma by induction.

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

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

$$\frac{\begin{array}{r} H \vdash e_1 : T \\ H \vdash e_1 \in \Sigma \\ H, v : T, X : (v = e_1) \vdash e_2 : A \end{array}}{H \vdash (\texttt{let val}\ (v : T) = e_1\ \texttt{in}\ e_2\ \texttt{end}) : A} \text{ LET VAL}$$

$$\frac{\begin{array}{l} H \vdash e_1 : (\forall v_1 : T_1. \ldots \forall v_n : T_n.(f\ v_1 \ldots\ v_n) : T_0 \wedge (f\ v_1 \ldots v_n) \leq_w v_{i_r} \wedge \\ \qquad \ldots \wedge (f\ v_1 \ldots c_j(\ldots, r_{j_k}, \ldots) \ldots v_n) \leq_w r_{j_k}) \\ H, v_1 : T_1, \ldots, v_n : T_n, f : (T_1 \to \ldots \to T_n \to T_0), (f\ v_1 \ldots v_n) \leq_w v_{i_r}, \\ \qquad \ldots, (f\ v_1 \ldots c_j(\ldots, r_{j_k}, \ldots) \ldots v_n) \leq_w r_{j_k} \vdash \\ \qquad e_2 : A \end{array}}{H \vdash (\texttt{let fun}\ f\ (v_1 : T_1) \ldots (v_n : T_n) = (e_1 : T_0)\ \texttt{in}\ e_2\ \texttt{end}) : A} \text{ LET FUN}$$

$$\frac{\begin{array}{l} H, u_1 : \psi(c_{b_1}, 1) \vdash a_{b_1} : (f(c_{b_1}(u_1)) : T_0 \wedge A(c_{b_1}(u_1))) \\ \qquad \vdots \\ H, u_n : \psi(c_{b_n}, 1) \vdash a_{b_n} : (f(c_{b_n}(u_n)) : T_0 \wedge A(c_{b_n}(u_n))) \\ H, v_{11} : \psi(c_{s_1}, 1), v_{12} : \psi(c_{s_1}, 2), \\ f(R_{s_{11}}) : T_0, \ldots, f(R_{s_{1p_1}}) : T_0, \\ X_{11} : A(v_{11}), X_{12} : A(v_{12}) \vdash \\ \qquad a_{s_1} : (f(c_{s_1}(v_{11}, v_{12})) : T_0 \wedge A(c_{s_1}(v_{11}, v_{12})) \wedge \\ (R_{s_{11}} \leq_w v_{11} \vee R_{s_{11}} \leq_w v_{12}) \wedge \ldots \wedge (R_{s_{1p_1}} \leq_w v_{11} \vee R_{s_{1p_1}} \leq_w v_{12})) \\ \qquad \vdots \\ H, v_{n1} : \psi(c_{s_n}, 1), v_{n2} : \psi(c_{s_n}, 2), \\ f(R_{s_{n1}}) : T_0, \ldots, f(R_{s_{np_n}}) : T_0, \\ X_{11} : A(v_{n1}), X_{12} : A(v_{n2}) \vdash \\ \qquad a_{s_n} : (f(c_{s_n}(v_{n1}, v_{n2})) : T_0 \wedge A(c_{s_n}(v_{n1}, v_{n2})) \wedge \\ (R_{s_{n1}} \leq_w v_{n1} \vee R_{s_{n1}} \leq_w v_{n2}) \wedge \ldots \wedge (R_{s_{np_n}} \leq_w v_{n1} \vee R_{s_{np_n}} \leq_w v_{n2})) \end{array}}{\begin{array}{l} H, L : B \vdash (ind(L, \lambda u_1.a_{b_1}, \ldots, \lambda u_n.a_{b_n}, \\ \qquad \lambda v_{11}.\lambda v_{12}.\lambda X_{n1}.\lambda X_{n2}.a_{s_1}(R_{s_{11}}, \ldots, R_{s_{1p_1}}), \\ \qquad \vdots \\ \qquad \lambda v_{n1}.\lambda v_{n2}.\lambda X_{n1}.\lambda X_{n2}.a_{s_n}(R_{s_{n1}}, \ldots, R_{s_{np_n}}))) : (f(L) : T_0 \wedge A(L)) \end{array}} \text{ IND}$$

$t : T$	t has type T;
$t \in \Sigma$	t is (statically) semantically valid (e.g. no undeclared variables or functions);
$\psi(c, n)$	returns the type of the nth argument of constructor c;
$f(X)$	replace the distinguished argument of f (given by context) by X;
$R_{s_{ij}}$	are the recursive call arguments over which the function is defined;
L	is the induction variable (we restrict to a single induction variable here).

Fig. 5. Structure Rules for $C^{Y}NTHIA$ (1).

$$\frac{}{H \vdash x \leq_w x}\ \textsc{wrefl}$$

$$\frac{H \vdash u_i \leq_w t}{H, (f \ldots x_i \ldots) \leq_w x_i \vdash (f \ldots u_i \ldots) \leq_w t}\ \textsc{wcons1}$$

$$\frac{H \vdash u_i \leq_w t}{H, (f \ldots c_j(\ldots, x_i, \ldots) \ldots) \leq_w x_i \vdash (f \ldots c_j(\ldots, u_i, \ldots) \ldots) \leq_w t}\ \textsc{wred}$$

$$\frac{(\forall i \in R_c) \quad H \vdash u_i \leq_w t_i \qquad (\forall i \in \{1, \ldots, n\}) \quad \vdash i \notin R_c \rightarrow (u_i = t_i)}{H \vdash c(u_1, \ldots, u_n) \leq_w c(t_1, \ldots, t_n)}\ \textsc{wcons2}$$

$$\frac{H \vdash u \leq_w t_i \quad H \vdash i \in R_c}{\vdash u \leq_w c(\ldots, t_i, \ldots)}\ \textsc{wcons3}$$

$$\frac{H \vdash (u \leq_w t) \bullet \{x_2/x_1\}}{H, Y : x_1 = x_2 \vdash u \leq_w t}\ \textsc{wsubst}$$

Fig. 6. Rules for Walther Recursion.

By applying IND, P_2 is instantiated to:

$$ind(z_3, a_{b_1}, \lambda h.\lambda t.\lambda X_1.a_{s_1}(t))$$

The IND rule gives rise to two subgoals. Consider the base case first:

$$\ldots \vdash a_{b_1} : ((f\ z_1\ z_2\ \texttt{nil}) : int\ list\ \wedge\ (f\ z_1\ z_2\ \texttt{nil}) \leq_w \texttt{nil})$$

The base case continues by applying WITNESS where a_{b_1} is instantiated to `nil`. This instantiation is in general provided by the user and is the one used here because it is the result in the base clause in the definition of `partition`. WITNESS gives us three subgoals:

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

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:

$$H, h : int, t : int\ list, (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t$$
$$\vdash a_{s_1}((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t)$$

Instantiating a_{s_1} to `if` $z_1(h, z_2)$ `then` E_2 `else` E_3, we can apply IF. This 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:

$$\ldots, X : z_1(h, z_2), (f\ z_1\ z_2\ t) : int\ list, X_1 : (f\ z_1\ z_2\ t) \leq_w t$$
$$\vdash E_2 : ((f\ z_1\ z_2\ (h :: t)) : int\ list\ \wedge\ (f\ z_1\ z_2\ (h :: t)) \leq_w (h :: t)\ \wedge\ t \leq_w t)$$

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:

$$\begin{array}{c} \ldots, X : z_1(h, z_2), (f\ z_1\ z_2\ t) : \mathit{int\ list}, X_1 : (f\ z_1\ z_2\ t) \leq_w t \\ \vdash (h :: (f\ z_1\ z_2\ t)) : \mathit{int\ list}\ \wedge\ (h :: (f\ z_1\ z_2\ t)) \leq_w (h :: t)\ \wedge\ t \leq_w t \end{array}$$

There are three 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. The third conjunct is easily proved using WREFL.

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

$\lambda z_1.\lambda z_2.\lambda z_3.\mathit{ind}(z_3, \mathtt{nil}, \lambda h.\lambda t.\lambda X_1.\mathtt{if}\ z_1(h, z_2)\ \mathtt{then}\ h :: (f\ z_1\ z_2\ t)\ \mathtt{else}\ (f\ z_1\ z_2\ t))$

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

3.4 Replaying Proofs According to User Edits

When the user applies an editing command to the current program, *CYNTHIA* must apply a corresponding edit to the current synthesis proof. Typically, this edit will make an isolated change to the proof. *CYNTHIA*'s replay mechanism then propagates this change through to the rest of the proof.

Definition 5. *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. *CYNTHIA* 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. *CYNTHIA* abstracts P_s into an ART A_s. *CYNTHIA* 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 *CYNTHIA* 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.

Some commands also require a change to the specification. For example, ADD CURRIED ARGUMENT adds an additional type to the specification.

The replay of the ART is the main method for propagating changes throughout the proof. 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).

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. As an example, consider type-checking rules. In some cases, expressions within the ML program will not need to be type-checked during the replay. Consider applying the ADD CONSTRUCT command to introduce a conditional `if then else` statement into the program. This will copy the highlighted expression, `E`, to each branch of the condition to give: `if C then E else E` where `E` has been copied. Clearly, there is no point type-checking `E` during the replay as its status will be unchanged. Some commands will require that `E` is type-checked, however. If CHANGE TYPE is used to change the top-level signature, then the target synthesis proof may require `E` to inhabit some new type. We must apply type-checking to see if this holds.

4 Why Use Proofs?

The use of proofs to represent ML programs is a flexible framework within which to carry out various kinds of analyses of the programs. 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.

The use of a proof is a natural way to provide detailed feedback on program errors. When an editing command is applied, any errors correspond directly to failed proof obligations. No extra effort is required to look for new errors — the edit is just applied and then the proof is replayed as far as possible.

In addition, $C^{Y}NTHIA$ 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. $C^{Y}NTHIA$ could also be extended to incorporate optimizing transformations such as those in the KIDS [12] system. The proof framework is also a very natural one for this purpose.

5 Evaluating C^YNTHIA

C^YNTHIA 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 [14]. Although some semi-formal experiments were undertaken, most analysis was done informally. However, the following trends were noted:

- Students make fewer errors when using C^YNTHIA than when using a traditional text editor.
- When errors are made, users of C^YNTHIA locate and correct the errors more quickly. This especially applies to type errors.
- C^YNTHIA 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^YNTHIA 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.

6 Related Work

Proofs-as-programs seems to be a good framework for designing correctness-checking editors. Another possible framework is that of attribute grammars [1, 10], 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.

No ML editors have been produced using attribute grammars. A couple of other ML editors have recently become available, however. MLWorks [5] and CtCaml [11] 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 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.

7 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.

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 International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
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. 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.
8. 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.
9. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
10. T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.
11. L. Rideau and L. Théry. An interactive programming environment for ML. Rapport de Recherche 3139, INRIA Sophia Antipolis, March 1997.
12. Douglas R. Smith. KIDS — a knowledge-based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. AAAI/MIT Press, 1991.
13. 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*, 1997. LNCS v. 1292. Also available from the Dept of Artificial Intelligence, University of Edinburgh.
14. J.N.D. Whittle. *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor*. PhD thesis, Division of Informatics, University of Edinburgh, 1999.