Proofs-as-Programs as a Framework for the Design of an Analogy-based ML Editor

Jon Whittle
NASA Ames Research Center,
ms 269-2,
Moffett Field,
CA 94040, USA
jonathw@ptolemy.arc.nasa.gov

Alan Bundy
Division of Informatics,
University of Edinburgh,
80 South Bridge,
Edinburgh, EH1 1HN,
Scotland
bundy@dai.ed.ac.uk

Richard Boulton
Dept of Computing Science,
University of Glasgow,
8-17 Lilybank Gardens,
Glasgow, G12 8QQ,
Scotland.
boulton@dcs.gla.ac.uk

Abstract

C^YNTHIA is a transformation-based editor for a functional subset of ML that lies somewhere between a structure editor and a framework for formal program development. Users construct programs from existing code by applying editing commands that make a semantic analysis of the program's behavior, e.g., whether it is terminating. All analysis is done using the *Oyster* system, which is an implementation of proofs-as-programs. We concentrate on identifying analyses that can be done fully automatically (e.g., using a decision procedure) and hence can be hidden from the user. As a result, C^YNTHIA is progress towards a goal of program editors that make an intelligent analysis of their code, but in a way that requires no extra input from the programmer.

1 Towards a New Kind of Editor

Imagine an editor that would automatically check code for errors as it is being written, and would either prevent the user from making certain errors, or warn of others on the understanding that they would be corrected later. Imagine too that the editor could deal not only with trivial syntactic errors but also much deeper errors, such as non-termination. The authors feel that such an editor would fill an important gap in current programming environments. At the one end of this gap is structure editing, whereby syntactically correct code, e.g., in the form of templates, is inserted into a program given a user's command. It is generally accepted, however, that structure editing is too restrictive for all except the most novice programmer. Simple syntactic changes to a program, made easily using a standard text editor, can become extremely awkward to make when forced to use a pre-defined set of transformations. At the other end of the gap is formal program development. The user may give logical specifications of a program's intended behavior which are then used to synthesize a program satisfying this behavior, either automatically (e.g., [ASG97, LPPU94]) or interactively (e.g., [MW93]). The problem here is that the user must have an intimate knowledge of both the specification language and, since full automation is well beyond state-of-the-art, theorem proving techniques.

The goal of C^YNTHIA is to begin to fill this perceived gap. C^YNTHIA is an editor for a functional subset of ML that makes a number of limited fully automatic analyses of the program in question. The intention has been to identify key properties or key synthesis tasks that could be fully automated using a restricted analysis routine (e.g., a decision procedure). In addition, C^YNTHIA is based on an extensible framework implemented in the Edinburgh proof checker *Oyster*. Proofs-as-programs [How80], whereby a functional program can be represented as a proof in a constructive logic, is used to both represent and reason about ML function definitions. Each function definition has a corresponding specification in *Oyster* concerning functionality of the function along with conjectures about its behavior (e.g., termination). *Oyster* was instrumented with special purpose strategies and decision procedures that enable these specifications to be proved automatically.

2 C^YNTHIA from a User's Perspective

Functional programming is a suitable testbed for developing new kinds of editors of the sort described above. Firstly, proofs-as-programs is directly applicable to functional programming. The essential idea is that given a logical specification and its proof, each proof step corresponds to an *extract term*. By composing the extract terms for all proof steps, an extract term for the entire proof is created. Since these extract terms are in fact functional λ-terms, translation to a functional language such as ML is easy. In principle, proofs-as-programs could be applied to imperative languages as well. In fact, a spin-off of the C^YNTHIA project investigated applying the idea to Java [Ble98].

Secondly, there has recently been increased interest in designing more advanced functional programming environments. MLWorks [Har96] provides an integrated environment for Standard ML that incorporates debugging facilities, profiling and browsing offering a graphical view of structured SML items. CtCaml [RT97] is a structure editor for CamlLight [Ler95] designed using the Centaur [BCD+88] framework. As well as structure editing facilities, CtCaml provides more advanced features such as the highlighting of type errors and type explanations. C^YNTHIA adds functionality that could eventually be integrated with the more traditional aspects of, for example, MLWorks.

One of the areas in which a new functional editor could make a real difference is in teaching. Despite the developments in the previous paragraph, the most common way to write ML code is still to use a standard text editor such as Emacs. This is a particulary inadequate method for novice functional programmers. It is well known that compilers provide error messages which may be unhelpful or confusing — cf., for instance, [JM00, Whi96]. The combination of this and the fact that functional programming contains concepts such as recursion which are difficult to learn anyway [APF88] suggests that there is scope for an editor that is specifically geared towards novice programming. The advantages that C^YNTHIA provides in teaching are twofold. In C^YNTHIA, programming is done by applying a pre-defined library of editing commands or transformations to an existing program. The underlying proof framework guarantees the correctness of these transformations in the sense that programs with certain properties (e.g., non-termination, redundant patterns) cannot be introduced. By restricting the student to correct programs in this way, the student needs to spend less time debugging trivial errors and can therefore graduate to the more challenging parts of the language. In addition, the editing commands in C^YNTHIA are reasonably abstract which means that students can compose functions from existing programs, for example in the course literature, that they understand well. This overcomes the "blank page" syndrome: faced with a program to write and a blank page, where does one start? Rather than hacking around in a text editor, C^YNTHIA encourages the student to think about the key properties of the intended program. Typically, the user will make some kind of high-level (possibly erroneous) consideration of the data and control structures needed and will then pull out a "similar" definition from a library. The editing commands can then be applied to make modifications to this program in a correctness-preserving way.

At each stage of the editing process, C^YNTHIA maintains a proof representation of the function. Editing is based around the collection of editing commands which propagate isolated changes to other parts of a function definition — for example, changing the type of an input argument automatically changes the patterns over which a function is defined. When the user applies an editing command, the edit is actually applied to the proof representation (see Figure 1). This results in a new partial proof which may contain gaps or inconsistencies. C^YNTHIA then attempts to resolve these by propagating the user's edits throughout the proof. Any which cannot be resolved are fed back to the user as program errors. Note that the user has no knowledge of the underlying proof representation and hence requires no knowledge beyond that normally required to program in ML.

The use of a proof to represent ML programs has a number of advantages, namely:

- It provides an extensible framework for the automated analysis of correct-

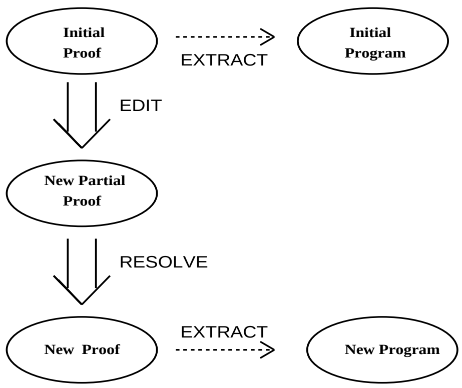

Figure 1: Editing Programs in $C^{Y}NTHIA$.

ness properties of ML programs. Any program written in $C^{Y}NTHIA$ is guaranteed correct with respect to the top-level specification of the corresponding proof. At present, $C^{Y}NTHIA$ specifications are deliberately rather weak. This is to make automated correctness-checking tractable. Any program written in $C^{Y}NTHIA$ is guaranteed to be free of the following kinds of errors:

1. syntax errors;
2. type errors;
3. non-exhaustive or overlapping patterns[1]. A program free of redundant or overlapping patterns will henceforth be known as *well-defined*;
4. other static semantic errors, e.g., no duplicate parameters;
5. non-termination errors.

Note that (1), (3) and (5) can never be introduced into $C^{Y}NTHIA$ programs because the design does not allow it. This is because editing is done by the application of commands which are themselves correctness-preserving. In the case of (2) and (4), this restriction is lifted since it is in general impossible to transform one function into another without passing through intermediate states containing, for example, type errors. In this case, the user is allowed to temporarily introduce ill-typed programs but is given immediate feedback as to the location of type errors. The framework is extensible in that it allows for the easy integration of analyses for other properties.

- The proof framework turns out to be a very appropriate one for specifying the style of editing embodied in $C^{Y}NTHIA$. The editing commands include transformations that allow the user to speedily manipulate existing

[1] We consider such patterns as errors as it is generally considered good style to avoid these patterns (especially for novices).

functions to modify them into new, similar functions. Implementing the editing commands requires that the dependencies between different program parts are maintained constantly. Proofs encode this knowledge in a flexible and principled way.

- The use of a proof also yields a degree of flexibility that turns out to be very useful in providing error feedback, e.g., type errors. Any type error in $C^{Y}NTHIA$ is highlighted to the user by displaying the offending expression in a different color. The proof of the specification of a function may be incomplete in the sense that there may be proof obligations which $C^{Y}NTHIA$ was unable to resolve. These incomplete obligations correspond directly to program errors. Hence, the proof framework allows errors to be highlighted easily — highlight the part of the program which corresponds to the failed proof obligation. Although MLWorks and CtCaml also have type highlighting capabilities, in our experiments it seems that $C^{Y}NTHIA$'s highlighting is more specific and hence provides more of a focus as to where the error lies.

2.1 A Brief Introduction to ML

There are a number of different dialects of ML. Throughout the rest of the paper, we confine attention to the recognised dialect Standard ML [MTH90] which is the most widely used. We do not attempt to describe functional programming here but refer the unfamiliar reader to [BW88]. The syntax of ML is best illustrated by example:

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

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

- **ML is functional**.
- **ML makes wide use of recursion and pattern matching**.
- **ML is strongly typed**. ML employs *type inference* to automatically infer types at compile time. For example, an ML compiler would deduce that *map* has type `('a -> 'b) -> 'a list -> 'b list`. `'a` and `'b` are polymorphic types.
- **ML is higher order**.
- **ML is impure**. It contains a small number of imperative features included for practical reasons, mainly for input/output.

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

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

2.2 Example

The best way to describe the specifics of C^YNTHIA is to give an example. Figure 2 gives an actual problem from a course in functional programming.

> *A "table" can be thought of as a store of data-items (the "values") where each entry is indexed by some data-item (the "keys"). The idea is that a value can be retrieved from the table by using the appropriate key, and that there is at most one entry for each key. The assumption is that tables will not be very large, i.e. there is no serious problem concerning the efficiency of search through a table. Choose a suitable representation for such tables and implement the following functions in a consistent manner:*
>
> *newtable: a new table containing no entries.*
>
> *addentry k v d: returns a new table which is the same as table d except that the value entered for the key k is v; this value replaces any entry that might be there for k.*
>
> *findentry k d: given a table d and key k, returns the value entered in d against k. Raises an exception if there is no such entry.*
>
> *hasentry k d: checks if the table d contains an entry for key k.*

Figure 2: Implementing Tables in ML.

Assume that a student decides to implement a table as a polymorphic list where the odd elements are the keys and the even elements are the values[3]. Assume also that the student has already defined *newtable* in C^YNTHIA and now wishes to define *hasentry*. Since editing in C^YNTHIA is based on incrementally modifying existing functions, the first thing that the student must do is to decide upon a starting definition. Since the tables will be represented by lists, the student chooses *length* (from C^YNTHIA's pre-defined library):

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

Note that all definitions in C^YNTHIA are accompanied by a type declaration. C^YNTHIA does not currently support type inference. Forcing the student to give an explicit declaration makes the student more aware of types and avoids the confusion that usually comes with type inference[4]. Note that there is little additional burden in providing type declarations, since programming in

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

[4]An expert version of C^YNTHIA could easily incorporate type inference — see [Whi99].

CYNTHIA is incremental anyway. The first thing to do to construct *hasentry* is to rename the length function. If the user selects the command RENAME at the indicated point[5] and types in `hasentry` at the prompt, the definition will change to:

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

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

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

The definition is now ill-typed with respect to the type declaration. As a result, *CYNTHIA* highlights the expressions which do not agree with the declaration (in reality, each expression turns a different colour, but in this paper ill-typed expressions will be underlined). The highlighting serves as a warning to the user but the errors need not be corrected immediately.

To access odd elements of the input list (i.e., the keys), the program will need to recurse in two steps. To achieve this, the student invokes MAKE PATTERN at the boxed point. This command will replace `t` with two cases – when `t` is empty and non-empty:

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

In the third clause, a new variable, `h1`, has been introduced. The definition is still missing a parameter for the key to search for. This can be introduced using the command ADD CURRIED ARGUMENT, which adds a specified parameter *throughout* the definition:

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

The user gives a name and type for the new argument and the type declaration is updated automatically. Finally, the user needs to change the output in each case. This can be done using the commands CHANGE TERM and ADD CONSTRUCT(IF THEN ELSE), giving[6]:

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

[6]Strictly, the second argument should now have type `''a list`. `''a` is a polymorphic type over which equality may be defined. *CYNTHIA* does not yet make any distinction between `'a` and `''a`.

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

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

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

To construct *addentry* is almost as easy. The user needs to invoke RENAME, ADD CURRIED ARGUMENT, CHANGE TYPE and CHANGE TERM:

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

At this point, the student may decide that the representation for tables is not the best one. A better choice would have been a list of pairs. $C^{Y}NTHIA$ can be used to transform the definitions to suit this new representation. For example, applying REMOVE PATTERN to *hasentry* gives:

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

Applying CHANGE TYPE at the indicated point gives:

```
'a -> ('a * 'b) list -> bool
fun hasentry k nil = false
| hasentry k (h::t) = if k=h then true else hasentry k t;
```

Applying MAKE PATTERN at the point indicated gives a correct solution:

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

The graphical interface of $C^{Y}NTHIA$ (not shown above) is given in Figure 3. Moving the mouse over the program highlights program subexpressions. Clicking on an expression brings up a menu of editing commands that can be applied. After a selection, the user types in any parameters in a pop-up dialog box. The command is then applied at the highlighted point, which may involve propagating the effects over the rest of the program. To evaluate any of these definitions, the user must compile them in the normal way.

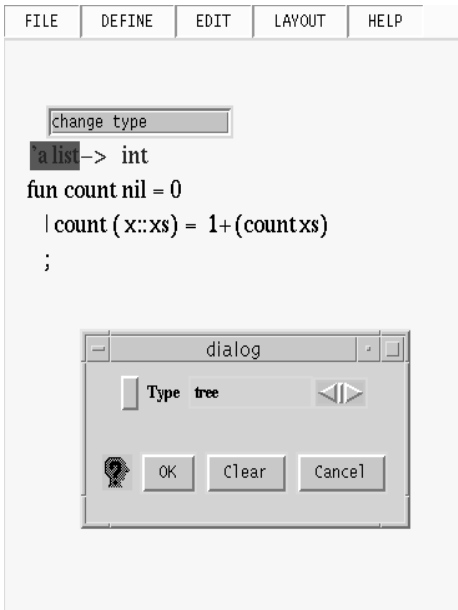

Figure 3: GUI for C^YNTHIA

2.3 Editing Commands

The full list of commands available in C^YNTHIA is given in Table 1. Of particular interest are the following.

2.3.1 CHANGE TYPE

CHANGE TYPE was used in the example but not in a situation where the argument was split into patterns. In such a case, C^YNTHIA automatically determines a set of patterns for the new datatype. As an example, consider changing `'a list` to `'a tree` in our first definition of *hasentry* († on page 7), where `'a tree` is defined as follows:

```
datatype 'a tree = leaf of 'a | node of 'a * 'a tree * 'a tree;
```

C^YNTHIA produces the following definition:

```
'a -> 'a tree -> bool
fun hasentry k (leaf n) = false
| hasentry k (node(h,leaf n,t1)) = raise excep
| hasentry k (node(h,node(h1,t,t1),t2)) =
                        if k=h then true else hasentry k t;
```

where `n`, `t1` and `t2` are fresh variables. Note that the target patterns are well-defined. More is said about CHANGE TYPE in Section 4.

Command	Description
ADD ARGUMENT REMOVE ARGUMENT	Globally add or remove a (non-curried) argument to a function definition
ADD CURRIED ARGUMENT REMOVE CURRIED ARGUMENT	Globally add or remove a curried argument
CHANGE TERM	Replace an expression by another one, entered as text
CHANGE TYPE	Globally change the type of an expression
MAKE PATTERN REMOVE PATTERN	Split a variable into cases
ADD RECURSIVE CALL REMOVE RECURSIVE CALL	Add or remove a (measure-decreasing) call to the list of valid calls
ADD CONSTRUCT REMOVE CONSTRUCT	Add or remove a `let val`, `let fun`, `if then else`, `case` or `fn` construct
RENAME	Global rename

Table 1: Editing Commands in $C^{Y}NTHIA$.

2.3.2 ADD RECURSIVE CALL

$C^{Y}NTHIA$ analyses programs for termination. During the editing process, it maintains a list of valid recursive calls – those which will not introduce non-termination – from which the user can choose one to be entered into the current program. Some recursive calls are added to this list automatically as a by-product of an editing command. In addition, the user may explicitly introduce a recursive call using the ADD RECURSIVE CALL command. The parameter to this command is a new recursive call. $C^{Y}NTHIA$ then checks the validity of this new call and, if it does not violate termination conditions, adds it to the list. CHANGE TERM is then used to introduce the recursive call into the program. All termination checking is done using Walther Recursion, see Section 3.3.

2.3.3 MAKE PATTERN

Our example showed an application of MAKE PATTERN. Note that the command may be applied at any level of nesting to build up arbitrarily complex patterns, which are guaranteed well-defined by construction. The proof editor ALF [Nor93] has a similar command.

Note that many commands make minor heuristic decisions when they are applied. For example, there may be more than one possible set of patterns after an application of CHANGE TYPE, or there may be a choice as to which source program fragment to duplicate in a MAKE PATTERN command. These heuristics are defined *a priori* and do not affect which functions can be defined in $C^{Y}NTHIA$.

2.4 Scope

At present, C^YNTHIA supports a purely functional subset of core SML. Some omissions were made due to lack of time, e.g., type inference and user-defined types are not supported (although some non-standard datatypes are built-in)[7]. Other omissions are due to the fact that the underlying proof system cannot deal with certain features. The main omission of this kind is that mutually recursive functions cannot be defined because the termination checker does not deal with mutual recursion. However, C^YNTHIA could be extended with another termination checker. One could imagine having a number of termination checkers available and control could be switched between them as necessary. Indeed, the termination check could be switched off altogether if the need arises.

3 C^YNTHIA Under The Hood

As previously mentioned, C^YNTHIA is based on the one-one correspondence between functional terms of the λ-calculus and constructive logic proofs. Each ML function definition is represented as a proof which enables us to reason about the correctness of functions. Proofs are written in the Edinburgh proof checker, *Oyster* [BvHHS90], which implements Martin-Löf Type Theory [ML79]. We give a brief introduction to this theory and proofs-as-programs in this section. The interested reader is referred to [How80] for more details.

A constructive logic is one in which "pure existence" proofs are disallowed. For example, a classical proof of the existence of an infinite number of primes might assume the converse and derive a contradiction. This is outlawed in constructive logic — the proof must instead compute a *proof object* or *witness*, such as a procedure for calculating the next prime in the sequence. In Martin-Löf Type Theory, each mathematical sentence is considered as a type, the elements of which are proofs of that sentence. A sentence is assumed to be true if and only if there is a proof of that sentence — i.e., the type is inhabited. The notation $obj : type$ is used to mean that obj inhabits the type $type$. Because of the fact that mathematical sentences are represented as types, the statement $obj : type$ can in fact be read in three ways:

- obj is an element of type $type$;
- obj is a proof of $type$;
- obj is a program for the task specified by $type$.

To represent a function with inputs I and outputs O the following specification is formulated:

$$\forall I \cdot \exists O \cdot spec(I, O)$$

[7]The CHANGE TYPE command is not restricted to built-in types but would work on user-defined types if they were available.

where *spec* specifies the function. This can be represented in sequent calculus form as:

$$\ldots \vdash P\!:\!(\forall I\!:\!i \cdot \exists O\!:\!o \cdot spec(I,O))$$

By applying inference rules to this sequent in a backwards fashion, P is instantiated as the proof is built up. A complete proof provides a full instantiation of P which corresponds to a program satisfying the specification. P will be a λ-calculus term and is often called an *extract term*. Hence, if the user provides a specification (which may be given to varying levels of detail) and then furnishes a proof of that specification, the extract term of the proof will be a program that is correct with respect to the given specification.

In C^YNTHIA, users are, without knowing it, editing a proof of a specification (albeit a very weak one) — see Figure 1, page 4. Each editing command makes an isolated change to part of a function definition corresponding to an isolated change in the corresponding proof. The change is then propagated automatically through the proof using an analogical mechanism that produces a new proof incorporating the changes. This is a two-step process. First, the isolated change is made which results in a new partial proof which may contain gaps or inconsistencies. C^YNTHIA then attempts to resolve these by formulating and proving new proof tasks. Any proof tasks which cannot be resolved are fed back to the user as program errors. Note that the extraction step translates the proof into Standard ML notation which is then displayed to the user. In this way, the user has no knowledge of the underlying proof representation and hence requires no knowledge beyond that normally required to program in ML.

In what follows, the definition of Quicksort given in Fig. 4 will be used as an example to illustrate key parts of C^YNTHIA's proof technology.

```
(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));
```

Figure 4: A Version of Quicksort.

3.1 Specifications in C^YNTHIA

Each ML definition in C^YNTHIA has a specification which is the top-level goal of the corresponding proof. In general, specifications may specify arbitrarily complex behavior about the 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 behaviors 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 *partition* in Fig. 4 is:

$$\begin{aligned} P : (\forall z_1 : (int * int \to bool).\ \forall z_2 : int.\ \forall z_3 : int\ list. \\ (\phi\ z_1\ z_2\ z_3) : int\ list \ \wedge\ (\phi\ z_1\ z_2\ z_3) \leq_w z_3) \end{aligned} \tag{1}$$

where ϕ represents the name of the function[8].

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 type $(int{*}int \to bool) \to int \to int\ list \to int\ list$. 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 the termination analysis. They express upper bounds on a function, ϕ, 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 ϕ. In this case, w is the length of a list, so this lemma states that the length of the result of *partition* is never greater than the length of the third argument.

These specifications (and their proofs) come from an initial library. A small library is provided with each release, but this library can be added to as the user sees fit. Any library component added by the user will also have a corresponding proof as long as it has been 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 (or removed) automatically depending on which edits are applied. This requires a fresh analysis of the entire function each time an edit is made. 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 is only required after the application of this command.

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

3.2 Proofs in C^YNTHIA

One of the central considerations in using proofs-as-programs within C^YNTHIA is to keep track of the translation between proof and program. Any ML function can be represented as a tree of inference rules in *Oyster*. However, *Oyster*'s primitive rules are somewhat low-level. This makes it awkward to develop the proof and program together. Editing commands are applied to the proof but if the proof is at a much lower level than the program then the exact point in the proof at which the command should be applied is unclear.

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. For example, the complex recursion scheme of Quicksort in Figure 4 could be implemented as a tree of induction rules in many different ways. Conversely, the same proof step could be interpreted in more than one programming sense. For example, a casesplit in a proof could be interpreted as a conditional statement or as pattern matching depending on the context.

In order to obtain a cohesion between proof and program, tactics[9] were added to *Oyster* which both raise the level of discourse within the proof and also make design choices which resolve ambiguities. The other advantage of increasing the abstraction in the proof is that it simplifies the algorithm for propagating the edits through the proof (see Section 3.5). This algorithm essentially produces a new proof by analogy with an existing one. It has been shown in [MW99] that analogy is more successful when working at a more abstract level.

Each ML programming construct is encoded in the proof as a tactic. Tactics were also designed for termination analysis, type checking[10], and other forms of analysis. In essence, each tactic corresponds to a small, domain-specific subproof, tailored towards the analysis of ML functions. As an example, C^YNTHIA guarantees termination of ML programs by providing well-founded induction rules. In fact, C^YNTHIA speculates induction rules that are then checked for termination by a technique known as Walther Recursion — see Section 3.3. This means that a complex recursion scheme can be implemented as a single induction rule rather than as a collection of multiple rules distributed over the proof tree.

3.3 Bounding Lemmas

One of the main correctness guarantees provided by C^YNTHIA is termination. Termination is in general undecidable. One 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,

[9]Tactics are small programs which control the application of inference rules.

[10]The type systems of *Oyster* and ML are not the same. In particular, ML type-checking is decidable which is not true in *Oyster*. Tactics were used to restrict *Oyster*'s types to a subset resembling those of ML very closely.

the user must not be expected to carry out such theorem proving. The difficulty in designing $C^{Y}NTHIA$ 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. The set of Walther Recursive functions [MA96] is such a set. $C^{Y}NTHIA$ restricts the user to this set 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, defined as $w(c(u_1, \ldots, u_n)) = 1 + \sum_{i \in R_c} w(u_i)$ where c is a constructor and R_c is the set of recursive arguments of c. In the case of lists, this measure is just length. Recursive arguments are defined as follows:

Definition 1 *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. The other arguments are* non-recursive.

There are two parts to Walther Recursion — bounding lemma analysis and measure argument analysis. Bounding lemma (BL) analysis calculates bounding lemmas. Measure argument (MA) analysis 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 applying previously derived bounding lemmas. There are two kinds of bounding lemmas — reducer and conserver lemmas. The distinction between reducer and conserver lemmas is given as follows. First, define the semantics of the inequality operator.

Definition 2 *$u \leq_w t$ if the following conditions hold:*
- *u and t are well typed and inhabit the same type;*
- *the measure of u, $w(u)$, is no larger than the measure of t, $w(t)$.*

Define strict inequality in a similar way. Conserver lemmas are of the form $\phi\ x_1 \ldots x_n \leq_w x_i$ for some i. Reducer lemmas are the same but the inequality is strict.

Consider the *partition* example. It satisfies the conserver lemma:

$$partition\ f\ k\ z \leq_w z \tag{2}$$

This can be proved automatically in $C^{Y}NTHIA$ using the rules for Walther Recursion (Figure 5) and induction. The rules in Figure 5 define the relation $\leq_w$. They work by decomposing the arguments of $\leq_w$ in a way that preserves the measure, w. WCONS1 and WRED apply previously derived reducer/conserver lemmas or an induction hypothesis. R_c is the set of recursive arguments of the constructor c. WCONS2 strips off the outermost constructors on each side of $\leq_w$. WCONS3 is similar but for the case when there is not an identical constructor on each side. WCONS4 applies a bounding lemma in a different way to WCONS1.

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

$$\frac{H \ \vdash \ u_i \leq_w t}{H, (f \ldots x_i \ldots) \leq_w x_i \ \vdash \ (f \ldots u_i \ldots) \leq_w t}\ \textsc{wcons}1$$

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

$$\frac{(\forall i \in R_c) \ \ H \vdash u_i \leq_w t_i \qquad (\forall i \in \{1, \ldots, n\}) \ \ \vdash i \notin R_c \rightarrow (u_i = t_i)}{H \vdash c(u_1, \ldots, u_n) \ \leq_w \ c(t_1, \ldots, t_n)}\ \textsc{wcons}2$$

$$\frac{H \vdash u \leq_w t_i \qquad H \vdash i \in R_c}{\vdash u \leq_w c(\ldots, t_i, \ldots)}\ \textsc{wcons}3$$

$$\frac{H \vdash u \leq_w (f \ldots t \ldots)}{H, (f \ldots x_i \ldots) \leq_w x_i \vdash u \leq_w t}\ \textsc{wcons}4$$

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

Figure 5: Rules for Walther Recursion.

3.4 Measure Argument Analysis

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

MA analysis involves showing that the measure decreases over each recursive call. To check for termination, the procedure in Figure 6 is adopted.

For our Quicksort example, Figure 4, 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 *partition* (*op* $<$) $h\ t \leq_w t$ and *partition* (*op* $>=$) $h\ t \leq_w t$. Since $t <_w h :: t$, termination is proved. Each application of ADD RECURSIVE CALL modifies the induction rule in the proof so that it incorporates proof obligations for showing measure reduction. The induction rule is well-founded as long as these obligations can be proved.

1. Find measure arguments, M, for ϕ by considering each x_i in turn and applying the rules for Walther Recursion;
2. **if** $M = \{\}$, termination analysis fails.
   else for each recursive call, $\phi\, u_1 \ldots u_n$, 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 ϕ terminates.
   else termination analysis fails

Figure 6: Procedure for Checking Termination.

In [MA96], Walther Recursion was described for a small, imaginary 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 [MA96] 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. *Oyster* 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.
- [MA96] 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.

3.5 Replaying Proofs According to User Edits

The user develops the structural parts of proofs using the editing commands. $C^{Y}NTHIA$ automatically carries out the correctness checking parts of the proof which include 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).

The procedure for editing the proof is as follows. First, we make the following definition.

Definition 4 *The Abstract Rule Tree (ART) of a proof is the tree of inference* *rule applications, where the hypotheses list, goal etc. have been omitted.*

The user highlights the position in the program where a change will be made. C^YNTHIA 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^YNTHIA abstracts P_s into an ART A_s. C^YNTHIA then makes changes to A_s to give $\psi(A_s)$. $\psi(A_s)$ is then unabstracted or replayed to give the new proof subtree $\psi(P_s)$. The complete new proof tree is then P_t with P_s replaced by $\psi(P_s)$. Note that C^YNTHIA 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, page 4.

Various optimizations have been implemented to improve the efficiency of the ART replay. Correctness-checking rules can be time-consuming and so C^YNTHIA selectively replays these rules. C^YNTHIA 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. Applying the ADD CONSTRUCT command to introduce a conditional `if then else` statement into the program will copy the highlighted expression, `E`, to each branch of the condition to give: `if C then E else E`. 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 then apply type-checking to see if this holds.

4 Changing Type in the Presence of Patterns

As well as being used for preserving correctness, editing commands can be useful in making some changes automatically, thus increasing productivity. One particularly interesting example of this is the CHANGE TYPE command. In many cases, an application of CHANGE TYPE will result in coloured parts of program text corresponding to type errors introduced by the edit. If the original program is defined using patterns, however, $C^{Y}NTHIA$ will do better than this and automatically change the patterns to respect the new datatype. There is a procedure in $C^{Y}NTHIA$ for making such changes which is guaranteed[11] to result in a set of patterns that are exhaustive and non-overlapping. To aid discussion, make the following definitions.

Definition 5 *Suppose a function ϕ has arguments $x_1, \ldots, x_n$ where these arguments may be curried or uncurried. Then x_i is said to be a* split argument *if ϕ is defined by pattern matching over x_i. Otherwise, x_i is an* unsplit argument.

Definition 6 *Suppose a data constructor c has type $T_1 * \ldots * T_n \rightarrow T_0$. Then c is a base constructor for T_0 if T_0 is distinct from each T_i, $1 \leq i \leq n$. Otherwise, c is a step constructor for T_0.*

If CHANGE TYPE is applied to modify the type of an unsplit argument, then no additional processing is required. The ART is merely replayed as in the previous subsection and ill-typed expressions remain in the proof as unsolved proof obligations. If the argument is split, however, $C^{Y}NTHIA$ will update the patterns that realize the split. This may involve adding clauses to or removing clauses from the function definition.

To motivate what follows, suppose the user is about to edit the following *member* function[12]:

```
nat -> [ nat list ] -> bool
fun member x nil = false
|   member x (h::t) = nateq(x,h) orelse member x t;
```

It may be that the user now wishes to produce a member function for natural number trees and so applies CHANGE TYPE at the indicated point. The application of this command produces:

```
nat -> nat tree -> bool
fun member x (leaf n) = false
|   member x (node(h,t,t2)) = nateq(x,h) orelse member x t;
```

There are two points to note here. First, $C^{Y}NTHIA$ has automatically introduced new variables, `n` and `t2`, to reflect the fact that the arities of the constructors are different in the source and target. Second, $C^{Y}NTHIA$ makes an analysis of the recursive structure of the target datatype and adds (or deletes) elements to the list of valid recursive calls. For example, `node` has two recursive arguments and so $C^{Y}NTHIA$ adds a recursive call for the new recursive argument, `t2`, namely `member x t2`. Even though this call is not in the program, it appears in the antecedent of an induction rule in the proof and so can be placed in the program at any time without further proving. $C^{Y}NTHIA$ knows

[11] A proof of this fact can be found in [Whi99].

[12] where `nateq` compares two natural numbers for equality

that any definition using this new recursive call will still terminate (since `t2` has measure strictly less than `node(h,t,t2)`).

The CHANGE TYPE command makes all of the above mentioned changes automatically. The algorithm to do this consists of the following steps:

1. Find a relation between the old and new datatype.
2. Update the relevant[13] source induction rule according to this relation. This may result in the addition or removal of antecedents in the rule.
3. Check the well-foundedness of the target induction rule (as before, using Walther Recursion).

Step (1) actually involves finding two relations — a relation between the source and target constructors (the *constructor relation*), and then for each source / target constructor pair, a relation between the arguments of the source and target constructor (the *argument relation*). We illustrate using *member* as an example. To derive these two relations, C^YNTHIA looks at the definitions of the source and target datatypes:

```
datatype nat list = nil | :: of nat * nat list;
datatype nat tree = leaf of nat | node of nat * nat tree * nat tree;
```

Note that these are instantiations of the polymorphic types `'a list` and `'a tree`. Of course, for a concrete CHANGE TYPE application, the instantiations may be different in source and target, but in this case, they are the same. The constructor relation associates source base (step) (as defined in Definition 6, page 19) constructors with target base (step) constructors. In this example, there is therefore only one possible relation, containing the elements $\langle \texttt{nil}, \texttt{leaf} \rangle$ and $\langle ::, \texttt{node} \rangle$. The argument relation associates source (non-)recursive arguments (Definition 1, page 15) and target (non-)recursive arguments. The elements of this relation for the example are: $\langle ::_1, \texttt{node}_1 \rangle$, $\langle ::_2, \texttt{node}_2 \rangle$ and $\langle ::_2, \texttt{node}_3 \rangle$, where c_i denotes the ith argument of constructor c.

Applying the relation to the current program (step (2)) is trivial — the source induction rule is modified so that if the constructor relation contains an element $\langle c_i, d_j \rangle$, then all occurrences of c_i are replaced by d_j. In the case that there is more than one possible d_j, an additional antecedent is introduced into the induction rule corresponding to this replacement. The procedure is similar for the argument relation.

In the example, the instantiations of the source and target polymorphic variables are the same. In general, this is not the case, and so the CHANGE TYPE algorithm must be applied recursively over the structure of the datatypes. This means that it may be necessary to derive a multiple number of constructor relations. For instance, if the source type had been `nat list` and the target type `nat list tree`, then C^YNTHIA would find two constructor relations, one

[13] Since the `case` construct may also introduce pattern matching, inductions may occur at any point within the proof.

between the constructors of `list` and `tree` and one between the constructors of `nat` and `nat list`.

Deriving a "reasonable" relation, given that there are in general many possible relations, is the most difficult part of the analysis. $C^{Y}NTHIA$ restricts these possibilities in two ways: restrict the relations to those that produce well-defined target patterns (*validity* restrictions), and use heuristics to decide between those that remain (*heuristic* restrictions). Often, this will not limit the choice to a unique relation. In such circumstances, $C^{Y}NTHIA$ will arbitrarily choose a relation from the restricted set of possibilities, which can be undone by the user if necessary (note that this rarely seems to be required, in practice).

4.1 The argument relation

The validity restrictions for the argument relation are as follows.

- If x_i is a recursive argument, and if $\langle x_i, y_j \rangle$ is in the argument relation, then y_j should also be a recursive argument. This ensures that recursive calls in the target recurse over recursive arguments.
- Similarly, if x_i is a non-recursive argument such that $\langle x_i, y_j \rangle$ is in the argument relation, then y_j should also be non-recursive.
- The duplication of arguments is not allowed — e.g., $\langle (\mathtt{h} :: \mathtt{t}), \mathtt{node}(\mathtt{h}, \mathtt{t}, \mathtt{t}) \rangle$ is disallowed because `node(h,t,t)` is not a valid ML pattern. Instead, a fresh variable is introduced, $\langle (\mathtt{h} :: \mathtt{t}), \mathtt{node}(\mathtt{h}, \mathtt{t}, \mathtt{t2}) \rangle$.

These constraints can always be satisfied.

The heuristic restrictions are:

- Introduce as few fresh variables as possible. This means that $\langle (\mathtt{s\ n}), \mathtt{e1} :: \mathtt{n} \rangle$, where `e1` is a fresh variable, is chosen over $\langle (\mathtt{s\ n}), \mathtt{e1} :: \mathtt{e2} \rangle$, where both `e1` and `e2` are fresh.
- For non-recursive arguments, choose relations that associate arguments of the same type. Hence, if two constructors are defined as:

  ```
  c : int * nat * 'a list -> 'a list
  d : nat * int * 'a tree -> 'a tree
  ```

  then $C^{Y}NTHIA$ would produce $\langle \mathtt{c}(\mathtt{h1}, \mathtt{h2}, \mathtt{t}), \mathtt{d}(\mathtt{h2}, \mathtt{h1}, \mathtt{t}) \rangle$. The motivation behind this is to minimize the appearance of ill-typed expressions in other parts of the definition.

4.2 The constructor relation

The validity restrictions for the constructor relation are:

- Base (step) constructors relate to base (step) constructors.

- All target constructors must have a pre-image in the source. Otherwise, ill-defined patterns would be produced. (Note that not all source constructors need have an image).

There is one instance where these constraints cannot be satisfied. This is when there are no source step constructors but there are some target step constructors. This means that the target step constructors cannot have pre-images that are also step constructors. Since the patterns produced by CHANGE TYPE must be well-defined, the target constructors without pre-images must be introduced explicitly. The parameters of these constructors are fresh variables.

For step constructors, the heuristics are:

- Relate constructors of the same name if possible.
- Preserve the type of constructors in the relation.
- Preserve the number of recursive arguments. This is a relaxed form of the previous heuristic.

In the case of mapping base constructors, only the first two heuristics still apply.

4.3 Example

Suppose `to`, `tn` are defined as:

```
to = ATOM of nat | NEG of to | AND of to * to;
tn = ATOM of nat | OR of tn * tn | LABEL of tn * nat | NEG of tn;
```

Consider the function definition[14]:

```
fun f (ATOM n) = n
|   f (NEG p) = (f p)
|   f (AND(p, q)) = plus ((f p), (f q));
```

When replacing `to` by `tn`, the constructor relation is given by $\langle$`ATOM`, `ATOM`$\rangle$, $\langle$`NEG`, `NEG`$\rangle$, $\langle$`AND`, `OR`$\rangle$, $\langle$`NEG`, `LABEL`$\rangle$ giving a new function:

```
fun f (ATOM n) = n
|   f (NEG p) = (f p)
|   f (LABEL(p, e1)) = (f p)
|   f (OR(p, q)) = plus((f p), (f q));
```

Note how the 1-many relation of `NEG` has increased the number of cases in the pattern definition. As a further illustration, consider the reverse: sending `tn` to `to`. In this case the relation is given by $\langle$`ATOM`, `ATOM`$\rangle$, $\langle$`NEG`, `NEG`$\rangle$, $\langle$`OR`, `AND`$\rangle$ and `LABEL` is "lost". The new function has no pattern for `LABEL`.

[14] where `plus` is addition over the natural numbers

4.4 Limitations

The CHANGE TYPE algorithm is based on a definition that states that a recursive argument of a constructor is one which has the same type as the result type of the constructor. However, there are datatypes that are recursive in nature but that do not satisfy this definition, such as the following datatype for terms:

```
datatype term = Var of string | Atom of string | Term of string * term list;
```

The CHANGE TYPE algorithm will not detect that `Term` is recursive and so will treat it as a base constructor, resulting in modified programs that are "surprising" to the user, even though they are well-defined. Further research is needed to extend the algorithm to such cases.

5 Evaluating C^YNTHIA

C^YNTHIA was designed to be an editor for novices to use. [WC00] reports on empirical studies undertaken with two groups of students at Napier University, Scotland, to assess if C^YNTHIA provides a good learning environment for functional programming. We give brief highlights of our findings here.

The first trial involved a group of 40 postgraduates learning ML as part of a course in Formal Methods (SG1). The second trial involved 29 Computer Science undergraduates (SG2). Although some semi-formal experiments were undertaken, most analysis was done informally, e.g., automatic recording of interactions with C^YNTHIA, videoing use of C^YNTHIA, interview sessions with students. Comparisons were made, however, between students using C^YNTHIA and students using a standard text editor. This was done by holding semi-formal experiments in which students were given problems and asked to attempt some tasks with and some without C^YNTHIA. In both cases, the SML of New Jersey (SML-NJ) compiler was used by the students.

Students tend to make fewer errors when using C^YNTHIA than when using a traditional text editor. We compared interaction logs of students using C^YNTHIA and using a text editor. A classification of errors was made and each kind of error was counted. Clearly, some kinds of errors (e.g., non-termination) were eliminated completely for C^YNTHIA users. More interestingly, the number of semantic errors, such as type errors, reduced significantly.

When errors are made, users of C^YNTHIA locate and correct the errors more quickly. Evidence from observations of students seems to suggest very positively that C^YNTHIA does better than a text-editor approach. Trival errors (e.g. clerical errors) have generally been filtered out so students are less infuriated and try to work out the problem rather than hacking at their code. Another point is that it is much easier in C^YNTHIA to distinguish what *kind* of error is occurring. This is because the error feedback is different for different categories of errors. One of the problems with ML syntax is that its succinctness means that syntax errors can often have knock-on effects meaning that they show up as type errors during compilation. The divide between errors in C^YNTHIA

is clear-cut. Type errors are always shown by pink highlighting. Other semantic errors are shown by green highlighting, and syntax errors can only occur in dialog boxes.

Since students are editing smaller chunks of program at a single time, the range over which the error could have occurred is far less. With $C^{Y}NTHIA$, each sub-expression is checked for errors immediately. In addition, the user knows that some parts of the program are guaranteed correct — for example, any patterns will have been built up using MAKE PATTERN and therefore must be well-defined.

$C^{Y}NTHIA$ may encourage 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 functional programming concepts and hence discourage, for example, attempts to program procedurally. Further research is needed to investigate this fully.

6 Related Work

$C^{Y}NTHIA$ fills an important niche in the way that editors for programming languages, in particular functional ones, are designed. The current implementation is committed to a structural style of editing on top of which the semantic framework is built. Longer term, however, such editors should be hybrid text / structure editors. It is for this reason that the CtCaml editor [RT97] for Caml-Light is not considered in the same class as $C^{Y}NTHIA$. The main motivation behind CtCaml was to apply the Centaur [BCD+88] environment generator which yields a structure editor for the given language. CtCaml does have some additional interesting features, such as type explanations, but does not provide advanced correctness checking such as termination. Type explanations [DB96] are an attempt to present the internal workings of the type checker in a way that is intelligible to the user as an aid to resolving type errors. As [RT97] says, type explanations tend to be too detailed and lengthy for reasonable use by a programmer, and they often hide internal specificities of the type checker which are necessary for a proper understanding. For this reason, the authors believes that type highlighting is a more promising approach. Certainly, in the trials conducted with $C^{Y}NTHIA$, type highlighting was found to be useful by the students. A recent approach to visualizing types [JM00] may also provide a degree of help.

The closest work in spirit to $C^{Y}NTHIA$ is the *recursion editor* [BGB91], an editor for a subset of Prolog that restricts to terminating predicates. In fact, the recursion editor was one of the original inspirations behind $C^{Y}NTHIA$. The two systems are fundamentally different, however. The recursion editor's transformations are based on pre-defined schemas, each expressing a particular recursion scheme. As such, it is limited in the same way as more traditional cliché programming [BP90], in that only a pre-defined number of schemas are allowed, and it takes a substantial effort to include more. The recursion editor has no underlying framework for checking termination but relies on the fact that the transformations preserve it. C^YNTHIA's approach, of including a general termination decision procedure, allows a much wider range of programs to be dealt with.

A similar line of research is that of developing programming techniques [BG96, KLS89, Rob91] for Prolog. Techniques are transformations that attempt to overcome the traditional problem with cliché programming (that the number of clichés required overwhelms the user) by making the transformations more abstract. However, the level of abstractness required can make the techniques counter-intuitive. Nor do techniques formally preserve any notion of correctness.

One of the most successful approaches to programming environment generation is the use of attribute grammars [RT89]. This results in editors that can propagate, for example, type information in a very similar way to C^YNTHIA. However, attribute grammars are an extremely general notation and as such, are not suited for implementing very complex correctness checks. Using a proof checker, correctness of implementation comes for free. In addition, proofs-as-programs is much more suited for dealing with functional programming. The proofs localise key parts of functions making analysis easier — for example, the induction rule encapsulates all information relevant to termination checking. This kind of mechanism would be very ugly to mimic as an attribute grammar.

7 Conclusion

The proofs-as-programs idea has been around for some time but it has not previously been used as part of a programming environment. In C^YNTHIA, proofs-as-programs has been used to represent novice ML programmers' function definitions in a way that the underlying proof framework has been hidden from the user. There have been very few other systems that use formal methods in this way. Systems using formal methods are generally for analysing large programs and hence require a great deal of expertise (in logic and automated theorem proving (ATP)) to be operated. One of the few systems which does hide the underlying framework is the AMPHION system [LVB97] in which users write graphical specifications of physical problems in the domain of space exploration and the system automatically extracts Fortran programs from a proof of these specifications. Although in a different domain than C^YNTHIA, AMPHION is similar in spirit in that the logic and proof framework is hidden from and need not be understood by the user. The idea in C^YNTHIA — to exploit proofs-as-programs in an editor for functional programming — is new.

C^YNTHIA stands in its own right as an original and state-of-the-art ML editor, containing various features that are either unavailable in other editors or are improved in C^YNTHIA. C^YNTHIA supports programming by analogy whereby users do not start from scratch but transform an existing program by applying a sequence of editing commands. No other ML editors support programming by analogy although editors for other languages (e.g. Prolog) do exist. A compact, small set of editing commands was developed. These editing commands are relatively intuitive and easy to learn but do not overly restrict the user to doing things in a particular way. Very few other programming environments check termination of programs. Those that do (e.g. the recursion editor [BGB91]) are severely restricted to a small subset of terminating programs. $C^{Y}NTHIA$ allows a much wider range of terminating programs to be constructed. $C^{Y}NTHIA$ also provides improved type error feedback over other editors — such as compilers which invariably give cryptic error messages.

It has been demonstrated that $C^{Y}NTHIA$ can be a useful teaching tool that can alleviate many of the traditional difficulties that novices have in learning functional programming. However, its use as an expert editor is currently limited. Some of these limitations stem from the fact that the full ML language is not supported. Whilst many of the omitted features could be added relatively easily (it would require tactics to be designed corresponding to each new construct), it would be problematic to extend some of the analysis features of $C^{Y}NTHIA$. The obvious example is termination checking. The authors feel that $C^{Y}NTHIA$ could be augmented with a variety of termination checkers. For example, mutual recursion could be dealt with using a checker based on recursive path orderings. Ultimately, of course, due to the undecidability of termination checking, the feature may have to be switched off in specific cases.

$C^{Y}NTHIA$ is currently based on structure editing. Whilst this is usually fine for novices, it would soon become frustrating for expert users. An expert version of $C^{Y}NTHIA$ should be a hybrid text/structure editor, in which the user can enter text freely, but has the option of invoking $C^{Y}NTHIA$'s editing commands during this process.

$C^{Y}NTHIA$ has demonstrated that proofs-as-programs can be used as a framework for implementing sophisticated editors. The technique seems particularly suited for functional programming languages. In particular, the use of proof rules localize information in the proof that is widely dispersed in the program. This makes the implementation of analysis methods easier than in a framework using attribute grammars, for example. It remains to be seen if the same advantages hold for other languages. Certainly, for other functional languages such as Haskell, the advantages will remain (see [Whi99] for a discussion). An initial investigation of using the same ideas to design an imperative language editor (for Java) have already been undertaken [Ble98].

References

[APF88] J. R. Anderson, P. Pirolli, and R. Farrel. Learning to program recursive functions. In M.T.H Chi, R. Glaser, and M.J. Farr, editors, *The Nature of Expertise*, pages 153–183, Hillsdale, NJ, 1988. L. Erlbaum.

[ASG97] Alessandro Armando, Alan Smaill, and Ian Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th*

IEEE International Automated Software Engineering Conference, pages 2–9, Lake Tahoe, Nevada, USA, 1997.

[BCD+88] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments*, Boston, USA, 1988.

[BG96] 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.

[BGB91] A. Bundy, G. Grosse, and P. Brna. A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172, 1991.

[Ble98] A. Blewitt. A Java editor based on proofs-as-programs. Master's thesis, Department of Artificial Intelligence, Edinburgh, Scotland, September 1998.

[BP90] D. Barker-Plummer. Cliché programming in Prolog. In *Proceedings of the META-90 Workshop*, 1990.

[BvHHS90] 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.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[DB96] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[Har96] *MLWorks*. Harlequin, Limited, 1996. Harlequin has discontinued sales and development of MLWorks.

[How80] 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.

[JM00] Y. Jung and G. Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.

[KLS89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for Prolog programming. Technical Report Tr-89-170, Case Western Reserve University, 1989.

[Ler95] X. Leroy. The CAML Light system (release 0.7). Projet Cristal, INRIA Sophia Antipolis, 1995.

[LPPU94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, Charlotte, North Carolina, October 1994.

[LVB97] M. Lowry and J. Van Baalen. Meta-Amphion: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering*, 4:199–241, 1997.

[MA96] 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.

[ML79] 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.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW93] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming*. Addison-Wesley, New York, N.Y., 1993.

[MW99] E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22:117–147, 1999.

[Nor93] B. Nordström. The ALF proof editor. In *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, 1993.

[Rob91] D. Robertson. A simple Prolog techniques editor. Technical Report 523, Department of Artificial Intelligence, University of Edinburgh, 1991.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.

[RT97] L. Rideau and L. Théry. An interactive programming environment for ML. Rapport de Recherche 3139, INRIA Sophia Antipolis, March 1997.

[WC00] J. Whittle and A. Cumming. Evaluating environments for functional programming. *International Journal of Human-Computer Studies*, 52:847–878, 2000.

[Whi96] J. Whittle. A quantative and qualitative analysis of errors encountered by novice ML programmers. Tech.rep, University of Edinburgh, 1996. In http: // ase.arc.nasa.gov/ whittle/ publications.html.

[Whi99] 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.