Formalization and 'Literate' Programming

Hugh Anderson
Department of Computer Science, NUS
hugh@comp.nus.edu.sg

Abstract

The 'literate' programming model is extended to include a concept of mechanical transformation. A prototype tool, FLP (Formal Literate Programming tool), has been developed which uses this extended 'literate' programming model in both a formal program proof setting, and within a formal (refinement) program development setting. In both settings, FLP provides history, access to tools, and an easy-to-use interface. FLP is a system with

- *a tree structured revision control system allowing easy access to an entire software development history,*
- *a unifying semi-formal model encompassing both program proof and refinement, and*
- *a single simple mechanism for managing both formal transformations on programs (proofs, tests, refinements) and informal transformations (explanations).*

In this paper, we outline the underlying semi-formal model for this extended 'literate' programming tool, briefly show the system architecture, and demonstrate the tool's use during a sample program development.

1. Introduction

The FOLDOC[1] dictionary of computing defines literate programming as:

> *"Combining the use of a text formatting language such as TEX and a conventional programming language so as to maintain documentation and source together. The program is sometimes marked to distinguish it from the text, rather than the other way around as in normal programs (the inverse comment convention)".*

[1]The Free-On-Line-Dictionary-Of-Computing is found at http://wombat.doc.ic.ac.uk/foldoc/index.html.

The 'literate' programming model ([9],[10]) allows code to be structured independently of the explanation of the code. In normal use a LaTeX document is written, which describes the program, along with in-line segments of code (there is no requirement for LaTeX, but it is commonly used). The in-line code segments are extracted with a tool to produce source-files [1]. The end result is a single document describing the code, which in addition can automatically produce the code as and when needed. This model of programming has some advantages - in the 'literate' programming community, it is common to associate the model with a notion of *correctness*:

> *From a purist standpoint, a program could be considered a publishable-quality document that argues mathematically for its own correctness. [13]*

In practice, the model does not achieve this - it only provides a convenient way to develop programs along with their documentation - there is not much of a *mathematical* argument involved.

However, software development often includes formal (or *mathematical*) techniques along with the informal ones, and the 'literate' programming model may be extended to support this. For example, a software developer might perform any of the following actions:

1. exhaustively test an existing piece of code,
2. argue that a code segment always ensures that some invariant is preserved,
3. convert one code segment into another using some rule that preserves the intent of the original code (unraveling a loop, reordering some assignments), or
4. use stepwise refinement to develop a segment of code from a specification.

Each of the above actions is normally considered to be distinct from the others - requiring different tools, skills and notations, and results are often lost forever when the software developer moves on to the next task. Paige develops a

unifying model in [12] focusing on the use of heterogeneous notations, but the focus here is different. In this extended 'literate' programming model, each activity is considered to be equal - a *transform* from an existing document to a new document.

The prototype revision-controlled 'literate' programming tool FLP allows each *transform* to be undertaken in a controlled and verifiable way. First the existing document is saved, and archived in a read-only form, and then (for each action given above), a relevant tool is applied directly to mechanically transform the (new) literate document.

At any later stage, the complete history can be validated, as all intermediate development steps are documented and retained.

Note that traditional revision control systems provide another way of preserving the history of a development, but normally make no distinction between the code before a *test* or *proof* and the code afterwards. From the software developer's point of view though, these two identical code segments are different - the goal of software development is not only to produce software, but also to produce assurance that the code is correct.

Each of these topics is expanded in the following sections, beginning with a semi-formal treatment for LP, continuing with explanations of interfaces used in FLP, and concluding with a brief description of the architecture and use of the prototype tool.

2. The extended LP model

This section introduces a single framework combining LP and a wide range of program development tools and techniques. It provides steps towards a formal justification for the model, emphasizing that the principal goal of software engineering is *assured software* - not just *software*.

Our model of formal LP program development begins with a notion of the 'document' D which, in addition to free text, embodies components of both the specification S, the implementation I, and the proof that I satisfies S. During intermediate steps of the development of D, it may also contain incomplete subcomponents of either a suggested refinement, or the proof.

The initial document D_1 may contain only a specification S, but a later document D_n may contain a full implementation I, as well as the assurance that this code is correct - a 'proof' that it matches the specification. In this model, no differentiation is made between a specification and an implementation - each are considered to be *programs* at different levels of abstraction.

We express each document D_i as an aggregation of elements $D_i \equiv \{T_i, P_i\}$ representing the text T at some stage, and the associated program P for that document. The text T_i may be ignored for the purposes of this treatment. Our programs P_i have an ordering imposed on them.

1. If our document transformation step is a refinement, then we use the refinement ordering. If P_{i+1} satisfies any correctness assertion that P_i satisfies, then we write $P_i \sqsubseteq P_{i+1}$ - P_i is-refined-by P_{i+1}.
2. If our document transformation step is a proof, then an association between some new specification S_i and our program P_i was made during the step, and a proof performed that P_i satisfies S_i. Our new program segment P_{i+1} may be identical to the original, but we have new knowledge that $S_i \sqsubseteq P_i$.

We restrict our transformations of the document D_i to those that preserve this relationship between the P_i components. This gives us a similar model as that found in refinement, and borrowing the notation - we have:

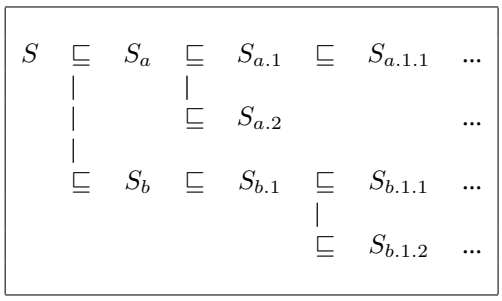
$$D_1 \sqsubseteq D_2 \sqsubseteq ... \sqsubseteq D_i \sqsubseteq D_{i+1} \sqsubseteq ... \sqsubseteq D_n$$

Various normally disjoint methods may now be used within this model.

2.1. Refinement calculus

The refinement calculus falls naturally within the extended LP model - any refinement performed always satisfies the specification that was refined, and all other parts of the document are unchanged.

The history of a program refinement may be documented by a sequence of LP documents. However at each stage of the process, we have a choice of possible refinements - (with not all refinements leading to an implementation) - and so during the development of a refinement, we may have a tree structure:



In addition, incomplete refinements may lead to additional proof obligations that must at some time be discharged. In FLP, such incomplete proof trees may be explicitly represented within a tree document structure - clarifying each of the subcomponents of the proof. Window inference [6] provides a formal basis for these styles of reasoning, allowing us to focus on a particular area of our proof and work on it independently of other areas.

While developing a proof, a tree like this may have many branches. Some of these branches may represent failed attempts at proofs/refinements, others may just be incomplete. At the completion of the proof, the tree represents all the steps of the proof.

The management and organization of these proof trees is a matter of some concern. In the HOL proof tool [3], incomplete proof steps are called subgoals, and listed in a series. The following display shows an intermediate display of a HOL proof, which has two incomplete subgoals:

```
OK..  2 subgoals:
> val it =
     A[j']<=A[P-1]
    ---------------          (First Subgoal)
     0.  !i j.  i<=P-1 /\ j<=P-1 /\ i<j
            ==> A[i]<=A[j]
     1.  j'<=P-1
     2.  ~(j'<P-1)

     A[j']<=A[P-1]
    ---------------          (Second subgoal)
     0.  !i j.  i<=P-1 /\ j<=P-1 /\ i<j
            ==> A[i]<=A[j]
     1.  j'<=P-1
     2.  j'<P-1
```

Schubert and Biggs [4] describe a graphical tree based interface for use with HOL which explicitly shows all subgoals, but due to the underlying architecture of HOL, cannot show two in-progress (incomplete) attempts at the same proof. Grundy's "Refinement Calculator" [2] and Nickson [5] have adopted similar ways of representing incomplete refinements, but neither system retains incomplete attempts to build a refinement. With these systems, at any one time there exists only a single (possibly partially completed) refinement sequence.

2.2. Testing, proof and transformation

Each of these software development activities may be treated in the same way as refinement, with some limitations on the use of testing tools:

Testing: Testing tools may be used to check the behavior of code segments. Model checkers such as the spin/PROMELA tool [11], can do exhaustive state-space analysis of surprisingly large code segments, and may be used to transform an LP document within our model. The use of one of these tools may generate an assurance that, for example, some (partial) implementation I_{i_a} implies some part of the specification S_{i_a} without affecting any other part of the document.

Proof: A similar argument to that given above demonstrates that program proof techniques such as the application of weakest-precondition calculus may be used to transform an LP document. The new document has an extra component - the assurance that some I_{i_a} implies some part of the specification S_{i_a} - in exactly the same way that exhaustive testing does.

Program transformation: Finally, we consider the application of a program transformation that preserves the meaning of the original code. In this case our two documents D_i and D_{i+1} are identical (at least as far as the model is concerned), and so we have that D_{i+1} satisfies D_i.

3. The 'literate' programming prototype

The following paragraphs give an introduction to the specification of the literate programming tool. The FLP program behaves simply - it has the following responsibilities:

1. It maintains a tree of literate program documents,
2. uses LyX to give a flexible user interface, and
3. allows the developer to apply an external (formal) tool to selected fragments of a document, resulting in the mechanical creation of a new descendant document.

The preceding section identified the utility of explicit proof and refinement trees, and justified the use of tree structured revision control in the 'literate' programming tool. A property of this part of FLP is that, in order to improve the consistency of the tree, if a document contains descendants, then it is marked as *no longer editable*. In this way, we cannot later change the antecedent of a document (which might make all its descendants invalid).

LyX [8] is an open source document processor which uses LaTeX, and supports 'literate' programming directly. It is used to provide the main interface for FLP. A small application manipulates the LyX window, applying selected transforms to the current document. The resultant documents are stored in child directories of the original document, but this is not normally apparent to the user of the tool.

3.1. Tree interface and consistency

FLP implements a graphical interface to a development tree, in which the branches represent specific transformations. For example - a development tree may look like that shown in Figure 1. The specific advantages of this interface are that:

- it is extremely easy to select any intermediate step of a development project, and access it, and

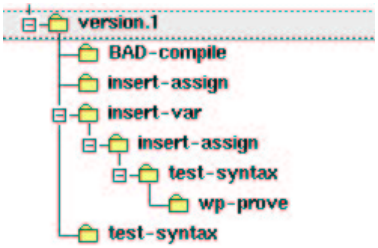


Figure 1. In-progress FLP development tree.

- traditional revision control processes have a natural *graphical* analog. For example - if you wish to undo a step, you may just select a transform further up the tree.

When a new descendant document is created, the original parent document must remain static in the area of the transform. Otherwise we may develop a situation where a descendant document is relying on something that has since been changed in a parent document. Unfortunately, the tool does not ensure this *tree consistency*, but it does attempt to help, by manipulating the file permissions to inhibit further modification.

3.2. Generic transform interface

FLP has a consistent view of program development activities - treating refinement, code conversion, proof, testing and informal justifications in a consistent manner - textually. In use, the system feels natural, and can be demonstrated in a few minutes. When the developer applies a tool to a marked section, the result always produces a transformed document in a subdirectory, with the parent marked as *no longer editable*.

The notions of *success* or *failure* of a transform are irrelevant in this model - both act in the same way, although the transform tool interface names the subdirectories differently dependent on the results of the transform. Failed transforms are no longer editable, and are just kept for informative purposes.

A single configuration file specifies the transform tools, which are small interface scripts. Each script processes the marked text into a suitable form before passing it to the associated tool.

The result from the tool is processed into a literate programming fragment which is returned to FLP along with a name for the transform. This is so that we can differentiate between differing results from the tool. We might perhaps use the names **BAD-proof** and **proof** to represent a failed and successful proof respectively.

3.3. Client/server architecture

LyX has a server mode of operation, in which the editor may be remotely manipulated. This suggested its use as a front end for FLP. The editor is used as a 'literate' programming editor, with a client program that controls its behavior. The client/server protocol is simple - there are only three types of messages:

- client->LyX: "LYXCMD:<client>:<command>:<argument>"
- LyX->client (answer): "INFO:<client>:<command>:<data>"
- LyX->client (notify): "NOTIFY:<key-sequence>"

The NOTIFY message type is used by LyX for asynchronously signaling client programs. Since the prototype tool had no need of this, we have a simple tool architecture as shown in Figure 2. In this view, there are several points of interest:

1. The tool is displaying a tree. This tree represents a tree-structured group of documents, any one of which may be selected by clicking on the tool display.
2. A segment of code has been highlighted, and we are about to select a particular transformation to apply to the highlighted code.

Transformations are instigated from the prototype tool window. The results of the transformation generate a new document, with the highlighted segment replaced with the transformed segment. This new document is automatically generated and loaded back into the editor, with the cursor positioned at the same place in the new document. The original document is write-protected to contribute to tree consistency.

3.4. Example use of FLP

In this section, we show how a small piece of code may be tested using a simple weakest-precondition analysis proof tool. This tool is a modified version of an example program-prover distributed with Harrison's **hol-light** [7], which tests correctness (with respect to a Hoare-style specification) for a simple imperative language. In this application, the **hol-light** program-prover runs as a background process, communicating via named pipes with the FLP tool. A literate program segment written in a C-like language may be selected, before an interface script converts it to a suitable form before sending it to the program-prover.

Within the editor, we mark the program text with a mouse and then select the C-proof menu, and the Wp-prove item in the FLP tool (see Figure 3). The selected text is an annotated C fragment which is supposed to calculate the factorial y of any integer n. If you look carefully in the code

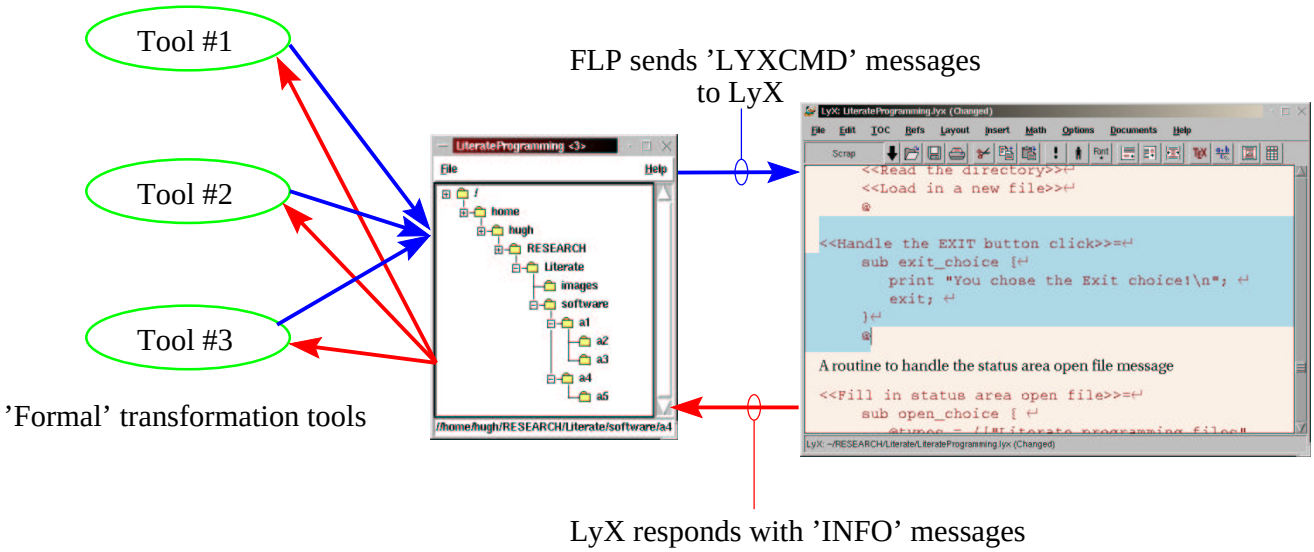


Figure 2. FLP tool architecture.

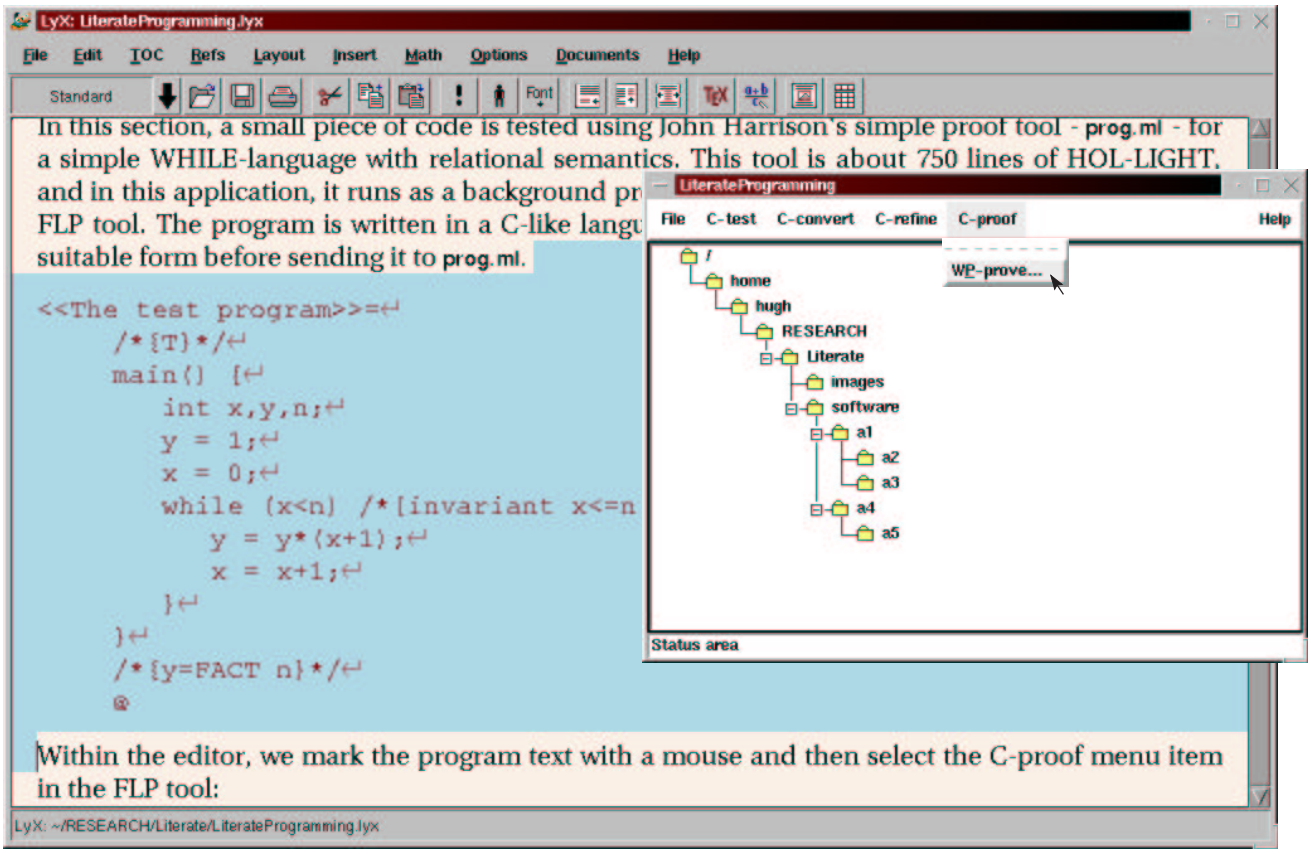


Figure 3. FLP in use.

in the diagram, you will notice that n is never assigned a value, but the theorem prover confirms that the final value of the variable y is FACT n.

The marked text is submitted to the proof tool, and the highlighted text on screen is replaced with new text. In this case the original source is restored, but also including the following text which reports the results of the test. In this case, reporting that the C fragment is correct - matching the specification $x, y, n : [T, y = \text{FACT}\, n]$ given as an annotation in the original code:

> The preceding code segment was submitted to wp-prove.pl on Tue Nov 21 17:56:53 GMT-8 2000. The result was:
>
> ```
> #EXAMPLE_43 : thm = |- correct (\(x,y,n). T)
>     (Assign (\(x,y,n). x,1,n) Seq
>     Assign (\(x,y,n). 0,y,n) Seq
>     Awhile (\(x,y,n). x <= n /\ (y = FACT x))
>           (measure (\(x,y,n). n - x))
>     (\(x,y,n). x < n)
>     (Assign (\(x,y,n). x,y * (x + 1),n) Seq
>     Assign (\(x,y,n). x + 1,y,n)))
>     (\(x,y,n). y = FACT n) #
> ```

The program developer can now continue with the development of the program, either editing the new document, or selecting sections for further transformation.

4. Conclusion

The development of this tool has identified various improvements and modifications necessary. Further work is needed in the following areas:

Use with multiple users: The model cannot support multiple users.

Interfaces to more tools: The model has a very small set of transformations.

Enforced consistency: FLP cannot guarantee consistency, as we allow documents to be edited immediately after a transform - a compromise solution to make FLP more usable. However a nice solution to this problem has presented itself, involving partial locks on documents.

In summary, FLP is a system which

- provides a simple revision control system allowing easy access to an entire software development history,
- provides a unifying semi-formal model encompassing both program proof and refinement, and
- provides a single simple mechanism for managing both formal transformations on programs (proofs, tests, refinements) and informal transformations (explanations).

The prototype tool is easy to use, and may provide a useful new tool in the software developer's tool-box.

Acknowledgments: Thanks to the anonymous referees for their helpful comments.

References

[1] David B. Thompson, "The Literate programming FAQ," http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lit_prog/faq.html.

[2] Michael Butler, Jim Grundy, Thomas Långbacka, Rimvydas Rukšėnas, and Joakim von Wright, "The Refinement Calculator: Proof Support for Program Refinement," in Formal Methods Pacific 97

[3] M.J.C. Gordon and T. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher-order logic." Cambridge University Press, 1993.

[4] Tom Schubert & John Biggs, "A tree based graphical interface for large proof development." International Workshop on the HOL Theorem Proving System and its Applications, September 1994.

[5] Ray Nickson, "Tool Support for the Refinement Calculus." PhD Dissertation, Victoria University of Wellington, 1994.

[6] Jim Grundy, "A Method of Program Refinement." PhD Dissertation, University of Cambridge, November 1993.

[7] HOL-light, available at http://www.cl.cam.ac.uk/users/jrh/hol-light/.

[8] LyX, available at http://www.lyx.org/.

[9] Donald E. Knuth, "Literate programming." *The Computer Journal*, 27(2), pages 97-111, May 1984.

[10] C.A. Lins, "A First Look at Literate Programming." *Structured Programming*, vol. 10, no. 1, pg 60-62.

[11] Gerard J. Holzmann, "Design and Validation of Computer Protocols." Prentice Hall, November 1990.

[12] R. Paige, "Formal Method Integration via Heterogeneous Notations." PhD Dissertation, University of Toronto, November 1997.

[13] Christoper Lee, "Literate Programming – Propaganda and Tools," http://www.cs.cmu.edu/~vaschelp/Programming/Literate/literate.html.