An Environment for Proving and Programming

Theodore S. Norvell and Zhikai Ding
Memorial University of Newfoundland

theo@engr.mun.ca, jding97@yahoo.com

Abstract

We present an integrated development environment for the editing, and development of programs, proofs, and proofs of programs. The environment provides not only WYSYWIG editing of proofs and program developments, but also support for the checking of proof steps and program development steps, the ability to apply theorems, and the ability to suggest the next step in a proof.

The so called "calculational" proof style pioneered by Dijkstra and others is used, with enhancements to allow hierarchical proofs. This style has shown itself to be suitable both for traditional mathematics and for verified program development.

An initial implementation covers basic Boolean, number, and set theory. Future implementations will extend the mathematical language used to include sequential programs and program modules. We represent both programs and their specifications as predicates on the initial and final observable state. Modules are specified in terms of abstract implementations.

1 Overview

This paper describes a system for program and proof development currently under design and development at Memorial University of Newfoundland. The system is called SIMPLE.

Common integrated development environments such as Microsoft's Visual Studio allow for increased programmer productivity by tightly coupling tools such as program editors, compilers, linkers, rapid development tools, documentation browsers, and debuggers. The SIMPLE environment adds to this mix editing of proofs —in particular proofs about programs— and integration with proof assistants and theorem provers. Our preliminary implementation integrates only a proof/program editor with a proof assistant, but is intended to be extensible, so that in the future it will accommodate compilers, automatic theorem provers, and other tools. A block diagram of the preliminary implementation is shown in Figure 1.

The main contributions and features of the full implementation of SIMPLE are:

- WYSIWYG editing of programs and proofs.
- Tight coupling with a proof assistant, for immediate feedback.
- A proof assistant that can suggest or check proof steps.
- Calculational proof style and calculational development of software components.
- Specification and programming languages embedded within a broader mathematical language.
- Imperative and object-based programming.
- A flexible polymorphic type system.
- An extensible system architecture.
- Easy exchange, browsing, and object embedding, via XML.

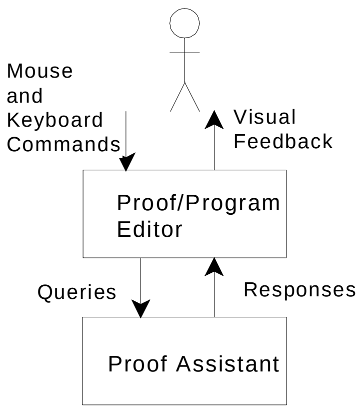

Figure 1

Theorem $(a \wedge b \Rightarrow c) = (a \Rightarrow (b \Rightarrow c))$
for all a, b, c
Proof

$$\begin{array}{ll} & a \wedge b \Rightarrow c \\ = & \quad \text{Material implication} \\ & \neg(a \wedge b) \vee c \\ = & \quad \text{DeMorgan} \\ & \neg a \vee \neg b \vee c \\ = & \quad \text{Material implication} \\ & a \Rightarrow (\neg b \vee c) \\ = & \quad \text{Material implication} \\ & a \Rightarrow (b \Rightarrow c) \end{array}$$

Figure 2

2 Mathematical language

The proof/program editor is a WYSIWYG text editor with a number of unusual features. We follow the Literate Programming paradigm, introduced by Knuth [1], that the formal content be embedded in the informal documentation. The formal text is a sequence of definitions and theorems.

Definitions are used to give names to mathematical expressions. The general form of a definition is: Definition N [: T] ::= E. (The underlined braces indicate an optional element.) The most general type for the expression will be computed by the proof assistant, using a Hindley-Milner style type inference system [2]. If a type, T, is specified, it must be an instance of the most general type; if no type is specified, the most general type is used instead.

Definitions can not be recursive and thus represent conservative extensions to underlying theory.

Expressions are built using common boolean, arithmetic, and set-theoretic operators and notations. We also use lambda expressions for creating functions, and function application. However the notion of mathematical expression is expanded to include imperative programming statements and modules, as explained in section 4.

Theorems are of the general form: Theorem E [for all $VSeq$] [where $ESeq$] [Proof $Proof$]. The expression must be boolean; its free variables (if any) must be listed in a list of variables, $VSeq$; any side conditions must be given in a list of boolean expressions, $ESeq$. The editor will keep track of whether or not each theorem has a valid proof or not. This information is indicated to the user by means of colour. When a theorem is changed, it becomes invalid, as do any theorems whose proofs rely on the theorem. Once a theorem has been entered, it may be used in future proofs.

Proofs are calculational in style [3]. Each proof consists of a sequence of expressions separated by relations and hints. For example, we can prove the shunting theorem as shown in Figure 2. There is no need to state types for a, b, and c as it can be inferred from the theorem statement that they are all boolean.

Each pair of adjacent lines and a relation constitute a proof step. Each proof step is independently checkable, within its context.

The hints are purely documentary and have no influence on the checking of the steps.

Subcalculations, as in [4], allow proofs by cases and induction.

3 Proof Assistant

The proof assistant is responsible for the following activities:

- Syntax checking.

- Type inference and checking.
- Checking proof steps.
- Making syntactic manipulations, such as commuting operands or reassociating.
- Suggesting steps.
- Filling in hints.
- Generating proof obligations.

The proof assistant is tightly integrated with the editor. Requests are typically sent to the proof assistant by placing the cursor on a part of the text or selecting a part of the text, and selecting a command via a menu or keyboard shortcut. The editor then sends a request to the proof assistant, embedding the relevant part of the text, and the relevant context information. The proof assistant makes syntax and type checks on the text sent, formulates a reply and sends the reply. The reply is communicated to the user by adding an additional step to the proof under development, or by signalling that a previously unchecked step has been checked.

4 Application to Program Derivation

Although the SIMPLE environment is intended to be usable for editing and checking proofs in traditional mathematical domains, it is in software development and verification that it is primarily intended to be used. We extend the mathematical language with the constructs of an imperative programming language. Thus we would be able to prove, for example:

$$(x, y := y, x) = (\mathbf{var}\, t := x \cdot x := y; y := t)$$

provided x, y, and t are distinct variable names. Such a theorem is generic over its variables and is applicable regardless the actual variable names and types. However, program equality is of limited use as program specifications are rarely written as programs themselves. Thus the programming language is broadened into a specification language by the addition of specification constructs. The statement **ens** Q, where Q is boolean expression about the initial state and the final state, behaves in such a way as to ensure the boolean expression holds. In such a boolean expression we use unprimed variables for values in the initial state and primed values for variables in the final state. For example,

$$\mathbf{ens}\; x \geq 0 \Rightarrow x - 10^{-5} \leq x'^2 \leq x + 10^{-5}$$

specifies a computation that, provided x is initially positive, calculates the square root of x, leaving the result in x. Although such statements can not be compiled, they are valuable as specifications. Instead of equality, the appropriate relation between specifications, including compilable statements, is refinement [5]; $S \sqsubseteq T$ means T is as specific as S. Programming constructs such as **if-then-else** and **while-do** may be applied equally to specifications or compilable statements. Thus programs can be developed proof-step by proof-step from their original specifications.

Using data-refinement [6], we can apply similar ideas to the development of modules, so that they too can be proved to implement an abstract specification.

The use of a polymorphic type system allow one to prove results not only about concrete specifications and programs, but also to develop program theory itself. Extensions to programming theory may be as simple as a factoring rule like

$$(\mathbf{if}\, E\, \mathbf{then} (S; U)\, \mathbf{else} (T; U)) = (\mathbf{if}\, E\, \mathbf{then}\, S\, \mathbf{else}\, T; U)$$

or may summarize algorithmic strategies like dynamic programming or binary search of a solution space. These theorems, once proved, may be applied as appropriate in the software development process.

5 Progress Report

The preliminary implementation of SIMPLE implements editing of definitions, theorems, proofs and informal explanatory text. The editor is written in Java, while the proof assistant is written in Haskell. All communication between the editor and the proof checker uses XML as does the saving of files. This allows texts to be easily converted to HTML for further publication and browsing; it also allows XML data of all sorts to be embedded in documents. The proof assistant implements syntax checking and parsing, type checking and type inference, and checking of proof steps. It supports reasoning about Booleans, integers, functions, and sets. There are three types of queries: check a proof step, apply a given rules, and find applicable rules.

Proof checking is based on pattern match in the abstract syntax trees of formulae. Monotonicity rules are applied implicitly in order to allow rules to be applied under operators; for example, if we know $b < c$, then a monotonicity rule allows us to conclude that $a - b > a - c$. Operators can also supply additional context; for example, in showing that $A \wedge B \Rightarrow A \wedge C$, we can use A as context while searching for a rule to show $B \Rightarrow C$.

Theorems are automatically preprocessed into rules, which can be applied to single steps in future proofs. Thus, proofs must be divided into steps that are small enough to be solved by a single rule.

6 Conclusions

The SIMPLE system and integrated development environments like it are unlikely to be used extensively in run of the mill programming. However, in areas where formalization is appropriate, such as safety critical applications, we feel that creating an easy to use, interactive environment with a highly expressive language can go a long way toward making formal software development more common and ultimately increasing the quality, reliability, and trustworthiness of software.

7 References

1. D. E. Knuth, ‘Literate programming’, *The Computer Journal*, 27(2):97–111, May 1984.

2. R. Milner, ‘A theory of type polymorphism in programming’, *Journal of Computer System Sciences*, 17(3), 1978.

3. E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

4. T. S. Norvell, ‘Induce-statements and induce-expression: constructs for inductive programming’, in *Foundations of Software Technology and Theoretical Computer Science*, 13th Conference, Bombay, Lecture Notes in Computer Science 761, 1993.

5. E. C. R. Hehner, *A Practical Theory of Programming*, Springer-Verlag, 1993.

6. D. Gries and J. Prins. ‘A new notion of encapsulation,’ In *Symposium on Language Issues in Programming Environments.* SIGPLAN, June 1985.