Mollusc
A General Proof-Development Shell for Sequent-Based Logics

Bradley L. Richards[*], Ina Kraan[†], Alan Smaill[‡], and Geraint A. Wiggins[§]

Abstract

This article describes an interactive proof development shell, Mollusc [Richards 93], which can be used to construct and edit proofs in sequent-based logics. Conceptually, Mollusc may be thought of as a logic-independent successor to Oyster [Bundy *et al* 90]. However, where Oyster was tied to a variant of Martin-Löf type theory, Mollusc can be used with any sequent-based logic for which a suitable definition is provided. Although developed in a research environment, Mollusc should also be suitable for use in classroom exercises. In addition to proof editing facilities, Mollusc supports the definition of new logics, includes a proof-planner interface, and provides for automated proof construction through a tactic language and interpreter.

1 Introduction

Mollusc is a general proof-development shell, which was designed as a logic-independent successor to Oyster. The Oyster system is an interactive proof checker for a variant of Martin-Löf type theory, developed as a rational reconstruction of Nuprl [Constable *et al* 86]. It is used principally to execute proof

[*]Artificial Intelligence Laboratory, Swiss Federal Institute of Technology, Lausanne, bradley@lia.di.epfl.ch

[†]Department of Computer Science, University of Zurich, inak@ifi.unizh.ch2

[‡]Supported by SERC grant GR/H/23610; Department of Artificial Intelligence, University of Edinburgh, smaill@aisb.ed.ac.uk

[§]Supported by ESPRIT Basic Research Project #6810, "Compulog II"; Department of Artificial Intelligence, University of Edinburgh, geraint@aisb.ed.ac.uk

plans developed by the proof planner CLAM [Bundy *et al* 90]. Recently, there has been a strong interest in applying CLAM to a variety of different logics, and this created a need for proof-checking support in these logics. Rather than creating a new proof-checking system for each logic, we decided to create a single shell which would work with a variety of logics, and which would assist the user in defining new logics. In addition to checking proof plans, the shell would also be meant for interactive proof development; thus, it had to include a friendly user-interface with on-line help.

These requirements formed the specification of the Mollusc proof development shell, which has been developed and implemented over the past two years. Mollusc consists of two components: the proof development shell itself and a logic-definition utility.

The proof development shell allows the user to choose a logic to work in, and then supports the user in exploring, constructing, and editing proofs in that logic. Mollusc supports backward inference only, i.e., moving from goals toward axioms. It provides a library mechanism and a means of tracking dependencies among proofs and definitions, as well as a tactic mechanism for automating the construction of a proof. Mollusc also provides a generic interface to proof planners, and can directly execute proof plans represented as tactics. The tactic language is a superset of that used by Oyster, and includes tacticals such as *complete, try, repeat, then, or*, as used in the LCF system [Gordon *et al* 79].

The logic-definition utility allows the user to formally specify the syntax of a new logic, and automatically produces both a parser and a set of "access functions" which allow the user to compose and decompose logical expressions in a declarative manner. The parser and access functions can then be used to write inference rules for the logic. Unlike the approaches taken in, e.g., [Huet & Plotkin 91, Paulson 89], where a variety of logics are represented in a single meta-logic, Mollusc does not imlpement a meta-logic, and generates a distinct theorem-proving system for each logic. The uniform presentation, however, does allow efficient reuse of inference procedures.

Mollusc has been used to implement propositional logic, many-sorted first-order predicate logic with equality, Martin-Löf type theory [Martin-Löf 79], Edinburgh Logical Framework [Harper *et al* 87], and a decidability logic for logic program synthesis [Wiggins 92]. Mollusc has been used most extensively in many-sorted first-order logic, where it was used to execute proof plans generated by CLAM to verify synthesized logic programs [Kraan 94]. The proof trees for such verifications were up to 0.75 megabytes in size; the tactic execution times for proofs of that size were under 40 seconds of CPU time on a Sparc station 10 using Quintus Prolog Release 3.1.4.

The rest of this article is organized as follows. Section 2 describes the proof development shell itself. Section 3 discusses the logic-definition utility. Section 4 discusses extensions that are planned for future releases. Finally, Section 5 describes how to obtain a copy of Mollusc.

2 The Mollusc Proof Development Shell

Mollusc provides an interactive environment for creating and editing proofs. Users can develop proofs manually or use the methods Mollusc provides to partially or completely automate the proof. Mollusc supports both scripts and tactics. A script is a file containing a series of Mollusc commands; this can be useful, for example, to set up the Mollusc environment to work with a particular set of proofs and definitions. Tactics, on the other hand, are programs representing a particular sequence of proof steps to be followed.

Although a complete description of Mollusc is beyond the scope of this paper, the paragraphs below describe three of the most important capabilities it provides: library support, proof manipulation, and tactics.

Library support. Mollusc maintains a library for each logic. A library may be distributed; normally, for example, there will be one or more central, shared repositories of definitions and lemmas, plus each user's local library containing work in progress. A user's start-up script file lists the shared libraries which should be searched, plus the user's local library. Mollusc loads proofs or definitions from all libraries listed, but saves work in the user's own library.

A library contains directories for proofs, tactics, and each type of definition in the logic. These items are all manipulated using the same basic set of commands (e.g., *load*, *save*). In addition, the user can create a file defining the dependencies of items in the library. For example, if the definition of *plus* should only be loaded after the definition of natural numbers, this can be specified in the dependency file. Mollusc will then automatically load all required definitions in the proper order.

Proof manipulation. The ultimate purpose of Mollusc is to allow the user to interactively create, edit, and display proofs. Mollusc provides an extensive set of navigation and display commands, allowing the user to move about a proof tree and inspect it. The display format is defined in a user-alterable file, so that the output format can be tailored to a particular logic. New display commands can also be added; this is useful, for example, if the logic has some additional feature such as extract terms.

Inferences are carried out in three ways. First, the user may explicitly invoke an inference rule. Mollusc can advise the user on possible inferences if the inference rules are written declaratively. Second, the user can create and execute tactics, as described below. Finally, the user can set an "autotactic", a tactic that is applied after every successful proof step. This helps eliminate many of the tedious steps in a proof, such as well-formedness goals.

Tactics. Tactics are essentially programs that describe how to perform a particular set of inferences. The tactic language is a superset of the language used in Oyster. At one end of the spectrum, the user might write a tactic that performs some common sequence of proof steps. At the other extreme, a tactic may include all steps necessary to completely recreate a proof. The user has control over the "grafting" of tactics. When a tactic is executed, all proof steps are normally represented explicitly in the proof tree, just as though they had been done manually. If a tactic is "grafted", then all intermediate steps are removed from the proof tree, and the tactic execution looks like a monolithic inference rule. Grafting can substantially shorten and simplify the proof tree.

Tactics may be created in three ways. First, of course, the user may write the tactic manually. Second, a tactic may be extracted from an existing proof or portion of a proof. Finally, Mollusc provides an interface to proof planners (e.g., CLAM [Bundy *et al* 90]), so that a proof planner can create a tactic for Mollusc to execute.

3 The Logic Definition Utility

When starting Mollusc, the first thing the user does is choose a logic to work in. Mollusc then loads a parser, a set of inference rules, a substitution algorithm, and a pretty printer for sequents in the logic. When a user wants to create a new logic, all of these must be created. While they can be written manually, much of the work is tedious and repetitive. To make the process of creating a logic easier, Mollusc provides standard templates for some files (e.g., the pretty printer) and a *define_logic* utility to help create a parser.

The *define_logic* utility allows the user to specify the syntax of the logic in a BNF-like language. The utility uses this specification to create a parser for the logic. It can also create a set of "access functions" which compose and decompose terms in the logic. These access functions make the process of writing declarative inference rules considerably easier.

4 Limitations and Planned Extensions

As with any new system, there are a number of areas where Mollusc could be enhanced. The major improvements being considered are:

- **Complete logic independence.** Mollusc currently assumes that target logics are sequent-based. This assumption could be lifted, making the shell completely logic independent. The principal difficulty is that Mollusc currently manipulates and displays hypothesis lists directly. In a completely logic-independent shell, utility predicates for these functions would have to be provided by each external logic definition.

- **Custom input parser.** Mollusc currently allows Prolog to parse all input items. This means that syntax errors result in uninformative messages. It also requires the user to end all entries with a period, and enclose descriptive items in single quotation marks. Providing Mollusc with its own input parser would eliminate these problems.

- **Custom display predicates for proof trees.** Currently, the logic designer can create custom display predicates for sequents, but not for the entire proof tree.

- **Automatic dependency determination.** In principle, Mollusc should determine dependencies automatically, by observing when definitions are used in the course of a proof. Adding this capability would simplify or even eliminate the manually constructed dependency file.

5 Obtaining Mollusc

Mollusc is available from the Mathematical Reasoning Group at the Department of Artificial Intelligence, University of Edinburgh. The contact person is Alan Smaill, smaill@aisb.ed.ac.uk.

Mollusc is written in Quintus Prolog, and runs under versions 3.1 or later. While it has not been ported to other Prolog implementations, this should not be difficult as long as the target Prolog supports modules. As of this writing, one user is porting Mollusc to SICSTUS Prolog.

References

[Bundy *et al* 90] 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.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[Harper *et al* 87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. of the Second Symposium on Logic in Computer Science*, 1987.

[Huet & Plotkin 91] G. Huet and G.D. Plotkin. *Logical Frameworks.* CUP, 1991.

[Kraan 94] I. Kraan. *Proof Planning for Logic Program Synthesis.* Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994. Submitted February 1994.

[Martin-Löf 79] 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.

[Paulson 89] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Richards 93] B. L. Richards. Mollusc user's guide version 1.1. DAI Technical paper 23, University of Edinburgh, September 1993.

[Wiggins 92] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M.I.T. Press, Cambridge, MA, 1992.