Proving as Programming with DrHOL: A Preliminary Design

Scott Owens and Konrad Slind
School of Computing, University of Utah

May 31, 2003

Abstract

We discuss the design of a new implementation of the HOL system aimed at improved graphical user interface support for formal proof. We call our approach *Proving as Programming*, since we believe that meta-language programming is a central aspect of proof construction. Thus we look to contemporary programming environments for inspiration on how to provide graphical support for proof. In particular, our implementation builds upon *DrScheme*, a popular programming environment for Scheme.

1 Proving as Programming

We have begun work on *DrHOL*, a new implementation of the HOL logic. DrHOL is systematically derived from HOL-4 [7] and aims at improving user interfaces in many aspects of work in HOL: development of proof procedures, construction of terms and definitions, interactive proof, and embedding of object languages are seen as candidates for better interface support. We believe that programmability is an essential part of all these activities. To support our view, we will discuss the ways in which we will adapt an advanced programming environment, DrScheme [2], into a proof environment for HOL-4. The main question being investigated—and it will take a while to obtain comprehensive answers—is : *How can support for programming also support proof*?

The issue of programmability in proof is interesting. One one hand, there is a long history of tactic proofs, stemming from the invention of tactics and tacticals in the original LCF system. Since tactics and tacticals are metalanguage programs, the construction of compound tactics to prove a goal can be considered programming. Thus we have a large body of historical evidence that programmability is *A Good Thing*. On the other hand, the *declarative* approach to proof has recently garnered much attention[1] [9, 5, 12, 8, 10], and is often presented as a way of abolishing programming from the activity of constructing proofs. A declarative proof is written in a fixed proof language, which may be

[1] Although its roots in the Mizar system are quite old by the standards of the field.

processed in a variety of ways. Importantly, the specification of a proof in the proof language is independent of the means of achieving the proof. This allows declarative proofs to be processed in more ways than tactics, which can only be executed. As a result, declarative proofs are more readable and maintainable than tactic proofs. However, declarative proofs can be more verbose than procedural proofs, and may contain more explicitly given terms. That the two approaches are not completely distinct is illustrated in [11], which demonstrates that declarative proof may be quite concisely implemented via tactics.

One of the early reasons for having programmability was *extensibility.* Since most of the basic tactics for LCF embodied quite small reasoning steps, composing them into larger steps was accomplished by tacticals and, often, by extended ML programming. Later work, especially in Isabelle, showed that long and intricate tactics could often be replaced by parameterized proof tools, such as first order provers and simplifiers. The amazing increases in computational power available to researchers have validated the move to more powerful proof tools. Interactive theorem proving has thus become much more efficient in terms of user time and effort, with the obvious *caveat* that when one gets well and truly stuck on a proof, no amount of automation will help.

However, in extended proof developments, we still find that highly automated tools are not completely adequate: the ability to write *ad hoc* proof procedures and custom term construction functions on the fly is a key facility. Similarly, when a sequence of steps re-occurs in proof, then programming is needed to avoid unnecessary repetition: this is just code reuse by procedural abstraction. To us, this implies that there is no real gap between online and offline construction of proof procedures. Thus a proof environment should support interactive construction of programs.

Proving as programming should not be confused with *Proofs as Programs*, a slogan associated with constructive logic. That slogan has specific and deep technical content behind it, while our slogan is more a methodological attitude.

2 DrScheme

DrScheme is a graphical program development environment for the Scheme programming language. DrScheme presents a single window with two nested windows, called the *definitions window* and the *interactions window*. The definitions window (the upper one) contains a program that users can execute, save to disk, and access in the interactions window. The interactions window (the lower one) provides a "read-eval-print loop" (REPL), in which users can experiment with their programs. Both windows implement the same programming language, use the same error messages, and report result values using the same syntax. When a user presses the "execute" button the interactions window is cleared, the program in the definitions window is executed and the resulting definitions are placed in the interactions window for the programmer to inspect and manipulate. The programmer can handle infinite loops with the "break" button. When pressed, the break button stops the currently executing program

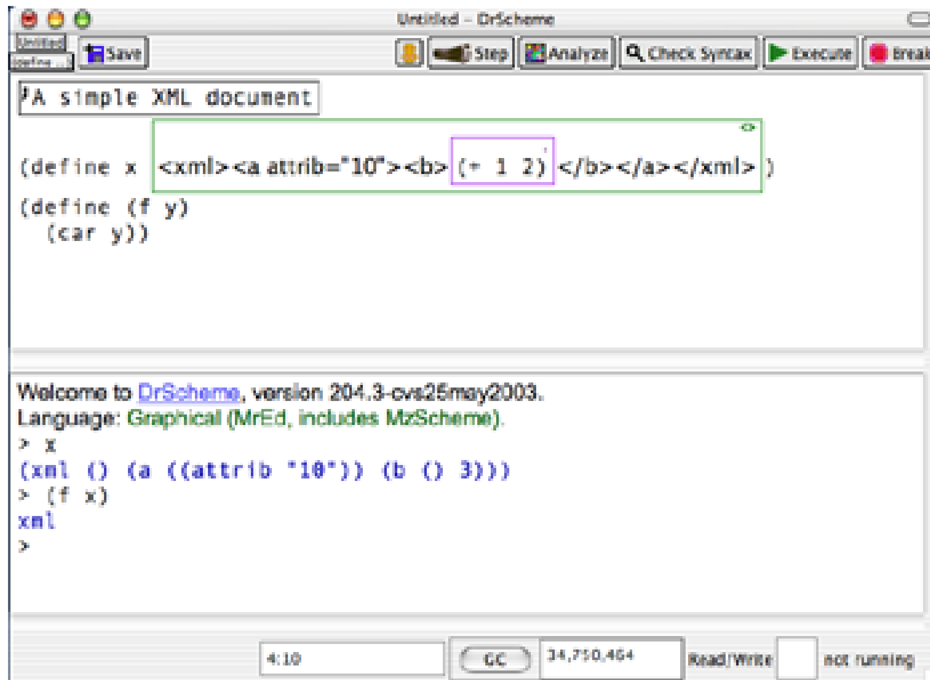

Figure 1: DrScheme

within either window. DrScheme provides the usual emacs-like program editing features in both the definitions and interactions window, such as auto-indenting and parenthesis matching. The definitions window has a *Check Syntax* feature that draws graphical arrows from a variable use to its binding when the user moves the mouse over the variable. Check Syntax also implements α-consistent variable renaming. While most program editors can graphically highlight the locations of compilation errors in the program's source text, DrScheme can also highlight the locations of runtime errors and uncaught exceptions when they occur.

To support both student and professional programmers, DrScheme provides several different language levels. A *Language Level* is a particular dialect of Scheme tailored to the needs of a certain class of students or professionals. Students usually work with a subset of Scheme that they can completely understand. This enables DrScheme to provide error messages that mention only concepts the students already understand. For example, the "Beginning Student" language level does not have anonymous functions and syntactically restricts the function position of an application expression to be a variable. Students often have difficulty at first in placing parentheses correctly and might write ((+ 1 2)). A Scheme system would report a runtime error indicating 3 is not a function. However, Beginner Scheme reports a compile-time error indicating that the expression (+ 1 2) appears in the function position of ((+ 1 2)) where

only a variable is allowed.

DrScheme is widely used to teach beginning programmers; the language levels are used to structure the course and have been proven and classroom tested over the past 8 years. The text [1] serves as the foundation of the course.

Each language level is implemented by a compiler that translates programs from the language level's dialect into the DrScheme's primitively supported Scheme dialect. For student language levels, these compilers primarily detect static errors and introduce code for better explanation of run-time errors.

DrScheme also supports an ALGOL60 language level and work is underway to support several Java and OCaml language levels for pedagogic purposes [4]. Although these language levels are technically the same type of thing (basically a compiler from their language to Scheme), they operate on a much grander scale. In particular, care must be taken to ensure that these languages interoperate nicely with Scheme programs.

The core of the DrScheme system is an interpreter called *mred* (pronounced 'Mister Ed'). mred interprets a dialect of Scheme that includes integrated graphical interface widgets. DrScheme itself is just a program written in this dialect that executes on the mred interpreter. DrScheme executes the user's programs directly in the mred interpreter, which supports sophisticated primitives that allow DrScheme to work robustly in the presence of misbehaving user programs [3]. Higher language levels can use the same facilities. DrScheme can run on the platforms that mred supports: Apple, Microsoft Windows and Unix with X-windows.

3 DrHOL

The HOL theorem prover is implemented in the SML programming language as a library of SML functions. The two main activities in HOL are constructing new logical terms and performing inference steps. Both of these activities require programming in SML. In the case of terms, the proper SML data constructors must be invoked to generate the term[2] and in the case of inference steps, the SML function that performs the inference step must be invoked. Occasionally a new inference rule or tactic needs to be written in terms of other existing inference functions. These facts lead us to the central conclusion that theorem proving in HOL is an inseparable activity from programming in SML. Hence our thesis that theorem proving in HOL should occur in a programming environment.

DrHOL is being implemented as a language level in DrScheme with a two-step approach.

1. An SML language level will be implemented via an SML to Scheme compiler. This will allow the existing body of HOL source code to be run inside the DrScheme environment and to interoperate with Scheme programs. Thus a HOL user will be able to use HOL in the DrScheme envi-

[2]Built in custom term parsers simplify this task

ronment as he does currently. However, he will have access to the helpful programming features of DrScheme.

2. HOL itself will be extended and modified to support greater integration into DrScheme. This step will allow more advanced user-interface features to be added to HOL, while maintaining a tight integration with the development environment.

3.1 SML to Scheme

We have chosen to provide programming environment support for HOL by creating a general purpose SML to Scheme compiler, although this is by no means the only possible approach. Because both HOL and DrScheme will be Scheme programs running on the mred Scheme interpreter, our approach will support a tight integration between the theorem prover and programming environment. Our SML to Scheme translation maps each externally visible SML construct into an equivalent Scheme construct, so that interoperation between Scheme and SML programs will not be difficult. For some SML constructs, such as structures, Scheme has no suitable built in construct. However, using Scheme's macro system, we can create new Scheme constructs without exposing implementation details to the programmer.

To simplify the task of building an SML compiler, we use the parser, overloading resolver and type checker from the Moscow ML compiler, all of which are written in SML. We currently invoke this front end in a separate process, but eventually intend to bootstrap the front end by translating these parts of the Moscow ML compiler into Scheme with our compiler.

We will briefly discuss several approaches that avoid building an SML to Scheme compiler and explain why we have not chosen them.

- *Use an SML interface widget package to build DrHol in SML* [6].
  Building a new extensible, production quality programming environment is a much more difficult and time consuming project than building an SML to Scheme compiler. Moreover, MoscowML, our current ML platform, doesn't support threads.

- *Translate HOL into a Scheme program manually.*
  While this might be feasible for some small core of HOL, the entire system is altogether too large. Furthermore, it's not *future-proof*: we want to be able to use future HOL libraries that will be written in SML.

- *Invoke a separate Moscow ML process from DrScheme to evaluate SML programs.*
  Two-process systems have been tried and found, in our opinion, to be less robust than desired. They have problems with dealing with undesired behaviour (breaking loops, interpreting error messages) and with interpreting returned values.

3.2 Safety

Via the SML type system, HOL guarantees that any theorem in the system has been proven through the application of a series of basic inference steps to some basic theorems. Knowing that theorems have been built using only a few simple and well-tested rules provides a HOL user with confidence that theorems produced in HOL are indeed correct. It is quite important that we provide this invariant in our Scheme system, not only when dealing with translated SML programs, but also when dealing with Scheme programs that use parts of the HOL system. Thus our compiler must preserve the observational equivalence relation for SML programs when put into Scheme contexts as well as SML contexts.

HOL implements theorems as an SML datatype and relies on the type system to ensure that no data constructed otherwise can be considered as a theorem. We ensure this property holds even when Scheme programs handle theorems by translating SML's datatype constructors into mred's *define-struct* form, which provides data abstraction facilities. HOL uses SML's structure system to restrict access to the theorem constructors, so that only the basic inference steps may see them. Since we translate SML structures into a Scheme implementation of structures, the theorem constructors are protected from Scheme programs just as they are protected from SML programs.

4 Benefits of DrHOL

HOL users will benefit from the integration of HOL into DrScheme almost immediately by taking advantage of the programming features of DrScheme. Beyond that, DrHOL will be extended to support more advanced capabilities ranging from modest extensions to ambitious projects.

4.1 Definitions and Interactions

The only existing way to interact with HOL is through an SML REPL. For many tasks, REPL interaction is ideal. Users can quickly try many different approaches to proving a theorem and get immediate responses from the system. DrScheme's interactions window provides a convient-to-use REPL whose robustness surpasses most existing interactive modes for emacs.

Once exploration is finished the user needs a record of construction of the expression (be it a type, term, theorem, or more complex entity). The record needs to be easily executed on demand, because the expression might be needed in the construction of another expression. Currently users must carefully look back through their interactions buffer and find the steps that made actually progress in the construction and manually copy them into a separate file. We will be able to assist the user in this process by providing automatic support for moving expressions from the interactions window into the definitions window.

4.2 Graphical Syntax

DrScheme supports the encapsulation of syntax in graphical containers. When a programmer adds a graphical container to the interactions or definitions window, the container is treated as a single character by the editor while the cursor is outside of the container. When the cursor is placed inside the container the programmer can edit the container's contents as though it were a separate window. These containers can be used to implement different syntaxes in much the same way as Moscow ML's antiquote or Lisp's quasiquote and unquote mechanisms. Compared to these techniques, graphical containers for syntax provide a much more easily recognized visual cue to the programmer that a different syntax is in use. Furthermore, none of the container's contents need to be prefixed with escape sequences since its extent is delimited graphically instead of textually.

DrScheme currently supports two different kind of containers: comment boxes and XML [3] boxes. The comment boxes contain comments whose contents are ignored. XML boxes contain literal XML text typed in directly. The contents of an XML box are converted to an s-expression when the box is encountered by the parser. The programmer can also insert a Scheme box into an XML box. The Scheme box's contents are evaluated as a Scheme program and the results placed into the XML box's resultant s-expression. The presence of the XML box lets DrScheme know exactly how the contents of the box should be treated. Inside an XML box, DrScheme automatically inserts the matching closing tag for each opening markup tag the programmer writes.

DrHol will provide HOL term boxes to allow the programmer to input and output HOL terms directly and in a natural syntax. For example, the HOL term box can automatically replace LaTeX like symbol commands (`\alpha`) with the actual symbol. It will also be able to color the term syntax to distinguish between logical constants, free variables and bound variables and, via Check Syntax, provide graphical arrows linking their definitions and uses.

4.3 Help System

We will be able to integrate the existing HOL help system with the programming environment. For example, the user will be able to select a HOL function with the mouse and open a new window with the documentation for that function. The user will also be able to select a logical constant from a HOL term and retrieve its definition. These facilities are more extensive and easier to use than the current HOL help system which either (a) dumps out a textual message to the interaction loop, thus obscuring the proof state; or (b) depends on an external web-browser.

[3] XML is the W3C's eXtensible Markup Language.

4.4 Goal Stack Management

HOL keeps a global state that tracks which obligations remain in proving some particular theorem. These obligations naturally form a tree, however for historical reasons HOL uses a *goalstack* interface that tracks only the leaves of the tree. Although this is one of the most heavily used tools in HOL, DrHOL will provide user-interface support for managing proof state directly as a tree. We expect that these additions will require moving beyond the two-window format that DrScheme currently supports, with an extra window that directly displays either the proof tree or goal stack. Then the user can graphically navigate the remaining proof obligations and DrHOL can automatically generate the final proof script from a completed proof tree.

4.5 Embedding other logics in HOL

A common activity in HOL is to embed computer languages and logics. This approach allows the user of the domain-specific logic to use all of HOL's tools and power when proving things in the logic as well as when constructing proof tools for the logic. Sometimes, however, the embedded language doesn't require such an extensive and complicated interface. Occasionally, the logic's only interface should be graphical. Our system would allow a domain-specific GUI extension to the DrHOL environment to be constructed alongside the logic's embedding. In particular, an embedding may require support for different syntax or even point-and-click proof tools.

5 Future Work

Although a future work section may seem strange in a design paper, we can see goals that will be achievable once our basic design has been implemented.

- We can implement a language level that supports only a subset of the HOL system. For example, one could envision a language level that introduced 'the essence of HOL' by restricting to forward inference using only primitive inference rules, as was done in Melham's successful HOL course. Another language level useful for teaching beginners would be tactic-based, along the line of Graham Birtwistle's *Ten Tactic HOL* method of teaching HOL. In such a level, a goal stack window would have a fixed number of buttons corresponding to the fixed repertoire of allowed tactics.

- With an integrated theorem prover, Scheme or SML programmers could have ready access to theorem proving for developing their programs. For example, one could envision a language level which enforced termination, via HOL proof, of each recursive function introduced by a user.

6 Conclusion

We have presented a preliminary design for an SML language level in DrScheme, and on top of that, a level for the HOL-4 implementation of the HOL logic. This system, DrHOL, will provide a basis for integrating and investigating user-interface support for interactive theorem proving. Underlying our design is the assumption that 'if proving is programming, then a good programming environment can be adapted to be a good theorem proving environment'. To give some substance to this viewpoint, we have discussed specific areas where the facilities of DrScheme can be used to improve proof development. We expect that many other such opportunities will arise as this work matures.

A major source of encouragement for us is the vigorous ongoing development of DrScheme by its talented group of developers. Just as we plan to adapt their insights about graphical support for programming, we hope our work on support for proof in DrHOL will provide useful ideas and challenges in the future development of DrScheme.

References

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, *How to design programs*, The MIT Press, Cambridge, Massachusetts, 2001, `http://www.htdp.org/`.

[2] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen, *DrScheme: A programming environment for Scheme*, Journal of Functional Programming **12** (2002), no. 2, 159–182, A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.

[3] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen, *Programming languages as operating systems (*or *revenge of the son of the Lisp machine)*, Proc. ACM International Conference on Functional Programming, September 1999, pp. 138–147.

[4] Kathryn E. Gray and Matthew Flatt, *ProfessorJ: A gradual intro to Java through language levels*, OOPSLA Educators' Symposium, October 2003.

[5] John Harrison, *A Mizar mode for HOL*, Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96 (Turku, Finland), Lecture Notes in Computer Science, no. 1125, Springer-Verlag, 1996, pp. 203–220.

[6] C. Lüth and B. Wolff, *Functional design and implementation of graphical user interfaces for theorem provers*, Journal of Functional Programming **9** (1999), no. 2, 167– 189.

[7] M. Norrish and K. Slind, *A thread of HOL development*, The Computer Journal **45** (2002), no. 1, 37–45.

[8] Donald Syme, *Three tactic theorem proving*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 203–220.

[9] M. Wenzel and F. Wiedijk, *A comparison of the mathematical proof languages Mizar and Isar*, Journal of Automated Reasoning **29** (2002), 389–411.

[10] Markus Wenzel, *Isar—a generic interpretative approach to readable formal proof documents*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 167–185.

[11] Freek Wiedijk, *Mizar Light for HOL Light*, Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001 (Edinburgh), Lecture Notes in Computer Science, no. 2152, Springer-Verlag, 2001, pp. 378–393.

[12] Vincent Zammit, *On the implementation of an extensible declarative proof language*, Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99) (Nice) (Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, eds.), LNCS, no. 1690, Springer-Verlag, 1999, pp. 185–202.