A TOOL FOR A FORMAL REFINEMENT METHOD

by

Albert Y. C. Lai

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2000 by Albert Y. C. Lai
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non-exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-49738-0
Abstract

A Tool for A Formal Refinement Method

Albert Y. C. Lai

Master of Science

Graduate Department of Computer Science
University of Toronto
2000

A formal refinement method is a method of constructing correct programs and proving programs correct: one constructs a program the way one would perform stepwise refinement, except that specifications are formal and refinements are justified by theorems. Thus, the program and its proof of correctness are developed in parallel, which is mentally easier than developing the program and the proof separately.

A major problem with the actual use of formal refinement methods is that, when carried out on paper, they could be mundane, tedious, error-prone, and hard to edit.

We build a computer tool for using Hehner's theory, a formal refinement method. It combines the advantages of both the syntax-directed editor Math/pad of Backhouse et al. and the proof checker driven Refinement Calculator of Butler et al.: an editor aware of the recursive operator-operand structure of expressions, with computer proof assistant support behind. The computer will do the mundane chore, and correcting mistakes and making changes will be convenient.
Acknowledgements

First of all, I must thank my supervisor, Eric C. R. Hehner, for his supervision, enlightenment, and patience; for every question I have, he has a revealing answer or hint. He also introduced me to the fascinating idea of symbolic, algebraic reasoning, better known as calculational proofs. Next, I thank Marsha Chechik, another faculty member researching into formal methods, for being intensely interested in and highly critical to this work, and being the second reader of this thesis. In addition, I thank my friends and schoolmates, Bryan P. K. Chan and Aaron M. S. Ng, for sharing my enthusiasm and seeing preliminary demonstrations of the tool. Retrospectively, I thank two of my teachers of mathematics in secondary school, Mrs. Y. K. Tsang and Mrs. L. N. Tse, for their early instilment of a sense of mathematical rigour: they demonstrated and insisted on precision in definitions, axiomatizations, and reasoning—they taught mathematics itself, not just a curriculum entitled “mathematics”.

iii
# Contents

1 **Introduction**  
   1.1 Formal Refinement Methods ............................................. 1  
   1.2 Using Formal Refinement Methods ........................................ 2  
   1.3 A Tool for a Formal Refinement Method ................................. 3  
   1.4 Organization of the Thesis .............................................. 6  

2 **Hehner’s Theory**  
   2.1 Specifications .............................................................. 7  
   2.2 Programs ................................................................. 8  
   2.3 Refinement .............................................................. 10  

3 **Translations to and from the Proof Checker**  
   3.1 Embedding Hehner’s Theory in Logic .................................... 12  
      3.1.1 Mathematics vs. Logic .............................................. 12  
      3.1.2 Translating Mathematics into Logic ................................ 13  
   3.2 Implementation of the Translation ...................................... 16  
      3.2.1 From the Editor to the Proof Checker .............................. 17  
      3.2.2 From the Proof Checker to the Editor .............................. 19  

4 **The Editor**  
   4.1 Using the Editor ........................................................... 21
4.1.1 The Stencil Window ........................................ 21
4.1.2 The Variables Window ..................................... 22
4.1.3 The Editor Window ......................................... 23
4.2 The Design of the Editor .................................... 29
  4.2.1 Representing Expressions ............................... 29
  4.2.2 Displaying Expressions ................................. 30
  4.2.3 Traversing the Expression Tree ....................... 31
  4.2.4 Operators .................................................. 32
  4.2.5 Other Parts of the Editor .............................. 32

5 An Example of Refinement ..................................... 34

6 Conclusion ......................................................... 39

Bibliography .......................................................... 42

A Source Code Listing ................................................. 45
Chapter 1

Introduction

1.1 Formal Refinement Methods

Among the many formal methods proposed for writing specifications and proving programs correct with respect to them, several belong to the class of formal refinement methods, such as those of Back [3], Hehner [11, 12], and Morgan [16].

A formal refinement method is essentially a formalization of the idea of stepwise refinement [19]. In stepwise refinement, one develops a program in a top-down fashion: state that the specification can be refined by a combination of several new, more concrete specifications, and recursively repeat for each new specification, until all specifications are concrete enough to write code for.

In a formal refinement method, the notions of code and specification are unified and formalized, and refinement is defined mathematically. Thus, one still carries out a process of top-down stepwise refinement: begin with a specification, recursively refine each specification by a combination of new, more concrete specifications, until all specifications reach the level of code. The difference is that now each refinement is justified using the given formalization, rather than by informal arguments.

We believe that formal refinement methods are easier to use than most of the other
formal methods because the burden of proof is spread over many points of the development, and therefore at each point the proof is small and easy to do, sometimes easy enough for a computer to resolve. In addition, immediately after conceiving a refinement, one still remembers the intuition behind it, and therefore one has a better chance of proving it with ease. Lastly, the entire process of refinements and associated proofs serves as a sort of internal documentation as well as a complete proof of the correctness of the final program.

1.2 Using Formal Refinement Methods

A problem with formal refinement methods is that they are mundane to do on paper. While most steps are relatively easy, there are many steps, and there is a lot of writing and copying of mathematical formulae; in addition, one may often make mistakes and need to backtrack, which is agonizing when done on paper manually. A solution to this problem is twofold: using an editor to aid the writing and copying, and using a proof checker to automate the mundane parts of the proofs.

The first aspect of the solution is given by Backhouse and Verhoeven [6, 5, 4]. Their project, Mathpad, is about an editor designed for performing mathematical work. It is called a syntax-directed editor, in the following sense. It recognizes the syntactical structure of a formula, i.e., it considers a formula as an operator attached to operand expressions (recursively), and aids the user in manipulating it as such. The user can easily navigate through various subexpressions and select one for editing, copying, and replacement. There had been no computer aid for proof behind this editor in the past, although adding it has been a long-term goal of the authors, and recently the editor has been extended to send candidate proofs to a proof checker for validation [18].

The second aspect of the solution is given by Butler et al. [15, 8, 7]. Their Refinement Calculator is a menu-driven tool for performing Back’s refinement calculus [3] with com-
puter aid for proof. It consists of a text editor, a proof checker equipped with theorems in the refinement calculus, and a menu that lists such theorems. The user can enter a specification and then apply a theorem from the menu to it, obtaining a validated refinement. However, the editor is not aware of the syntax of the expressions and does not aid the user in navigating through the subexpression structure; in addition, it presents expressions using the notation of the proof checker, which is not necessarily natural.

1.3 A Tool for a Formal Refinement Method

The contribution of this work is a tool that combines these two complementary approaches. We create a syntax-directed editor with computer aid for proof; it will ultimately provide these functions:

1. Syntax-directed editing:
   
   (a) The tool displays expressions in familiar forms, using familiar symbols and conventions.
   
   (b) In the display of an expression, the amount of space before and after each subexpression is proportional to its complexity (subtree height), so that the syntactical structure of the overall expression is more apparent.
   
   (c) The user edits an expression by manipulating subexpressions. This way, the tool treats an expression as a tree of subexpressions, although the expression is displayed in a single line.

2. Proof aid:

   (a) The tool contains a proof checker as a backend.

   (b) Using the proof checker, the tool can simplify an expression.

   (c) Using the proof checker, the tool can prove a refinement by applying inference rules and theorems chosen by the user.
(d) The tool can suggest some applicable inference rules and theorems based on the form of the refinement to be proved. The user may then choose from the suggested list of inference rules and theorems.

(e) Among all of the refinements entered, the tool can locate those that are not yet proved and bring the user's attention to them when the user asks for it.

(f) The tool keeps and displays the proofs, i.e., the inference commands used and the intermediate results obtained. This is a form of documentation for the correctness of the refinements and the resulting program.

3. Programming aid: The tool can check whether all specifications are refined down to the level of code. In case they are, the tool can translate them into a popular programming language (such as C or Standard ML) for compilation and execution. Otherwise, the tool can bring the user's attention to those specifications that need to be further refined.

Thus, we gain the advantages and eliminate the disadvantages of a standalone syntax-directed editor and a standalone proof aid.

Currently, we have only implemented 1, 2a, and 2b. Refinement in Hehner's theory [11, 12] is supported, with the restriction that the only data types are the booleans and the natural numbers. The other functions and more data types will be added in the future.

Our tool consists of two major parts: the editor as the frontend and the proof checker as the backend. Figure 1.1 shows their relationship. The user would enter a refinement into the editor and possibly edit it; then the user would attempt to prove the refinement using the commands provided by the editor, such as the functions 2b and 2c listed above. The editor is responsible for providing an environment for editing and proving; user commands for proving are handed to the proof checker. The proof checker is responsible for the actual inferences and calculations; it communicates with the outside world via a
bi-directional text stream using its own syntax, so we build two translators to convert expressions between the editor's representation and the proof checker's syntax.

The proof checker is chosen based on these basic considerations:

- We would like to have a proof checker that uses higher-order logic. Although Hehner's theory only requires first-order logic, it does not preclude higher-order logic, and higher-order functions could be useful in some uses of Hehner's theory; we would like the proof checker to retain this flexibility.

- We would like to have a programmable proof checker; this makes the proof checker easier to use. For example, if we find that we frequently perform a certain sequence of actions on the proof checker, we can turn this sequence into a new command and add it to the proof checker.

Two prominent higher-order, programmable proof checkers or theorem provers are HOL [9, 13] and PVS [17]. Their respective advantages are:

- HOL:
  - Its programming language (Standard ML) is strongly typed, and theorems are represented in the language as a type. This makes it easy to distinguish theorems from ordinary expressions.
Chapter 1. Introduction

- It uses less disk space and memory in recent versions—about one-fifth of that of PVS.

- PVS:
  - It is more mature in automatic proof search.
  - It has a proof format for presenting the intermediate steps taken in proofs.

Although PVS has better automation, HOL is improving in this aspect as researchers implement decision procedures and simplifiers for it. The proof presentation format provided by PVS is of little benefit because we will probably use some other format in our tool: PVS uses trees of sequent deductions, whereas most formal refinement methods assume calculational formats. Therefore, the advantages of HOL are more significant and we choose it over PVS.

1.4 Organization of the Thesis

This thesis consists of three parts. The first part is this document. The second part is the tool itself, available both on CD-ROMs attached to original copies of this document and at the WWW URL http://www.cs.utoronto.ca/~trebla/scphEditor/; the source code listing is also provided as an appendix to this document. The third part is a demonstration of the tool given to the two evaluators of this thesis.

The rest of this document is organized as follows. Chapter 2 is a short introduction to Hehner's theory. Chapter 3 describes the two translators. Chapter 4 describes the editor: both how it is used and how it is designed. Chapter 5 contains an example of using the tool to perform refinement in Hehner's theory, followed by a discussion of the desirable types of proof aid. Chapter 6 is the conclusion, which summarizes the contribution of this thesis and the plan for future work.
Chapter 2

Hehner's Theory

This chapter is an introduction to Hehner's theory; full expositions are in [11, 12].

Hehner's theory is a formal refinement method for developing, or deriving, programs that correctly implement specifications; the development is through refinement, starting with a specification and ending with a program. As a result, the development process also documents a correctness proof.

In order to do this, the notions of specification and program, as well as their relation (i.e., under what conditions we can say that one refines or implements the other), must be formalized. We will formalize specification; then program will be a special case. Afterwards, we will describe refinement.

2.1 Specifications

We assume that there are some state variables, say $x$, $y$, and $z$ of the natural number type, which the specification mentions and which the program can modify. Before a program statement is executed, these state variables have some initial values, and we still name them $x$, $y$, and $z$, respectively; after the program statement is executed, they have some final values, and we name them $x'$, $y'$, and $z'$, respectively. (This is an imperative computation model in which inputs are initial values and outputs are final values.)
A specification is a predicate relating the initial values with the final values. For example, a specification for taking an input value from $x$ and outputting the square of $x$ in $y$ is:

$$y' = x^2$$

Specifications can be non-deterministic, as predicates can be. The above specification does not dictate what $x'$ and $z'$ must be, so it is in fact non-deterministic in this regard. As another example, the following specifies the output $y'$ to be a number between $x$ and $x + 10$ inclusive, and any one will do:

$$x \leq y' \land y' \leq x + 10$$

The following specification, written in implicit form, asks for a solution to an inequality; any solution will do.

$$x \leq 2 \times y' + 4 \land 2 \times y' + 4 \leq x + 10$$

### 2.2 Programs

A program is a special case of a specification, in the sense that a program specifies the relation between input and output, albeit algorithmically.

For example, the program statement $\text{ok}$ in Hehner's theory is the equivalent of "no operation" in most programming languages, and it specifies that all variables keep their values. Assuming again that the state variables are $x$, $y$, and $z$, of the natural number type, $\text{ok}$ is then defined as

$$x' = x \land y' = y \land z' = z$$

An assignment statement such as

$$y := x$$

specifies that the final value $y'$ be set to the initial value $x$ and the final values of other variables stay the same as their respective initial values. Thus, it is defined in Hehner's
theory as

\[ y' = x \land x' = x \land z' = z \]

Another programming construct is \textit{dependent composition}, which represents sequential execution. In most programming languages it is denoted by a semicolon ";", while in Hehner's theory it is denoted by a period ".". Its definition is as follows: suppose \( P \) and \( Q \) are specifications (or program statements), then the dependent composition

\[
P \cdot Q
\]

is defined as

\[
\exists x'', y'', z''. \; P[x''/x', y''/y', z''/z'] \land Q[x''/x, y''/y, z''/z]
\]

Informally, this predicate says that the output values of \( P \) are propagated as input values to \( Q \). This definition can be better understood from a few examples:

The following copies the value of \( x \) to \( y \), then copies the value of \( y \) to \( z \). At the end all variables should contain the original value of \( x \).

\[
y := x \cdot z := y
\]

\[
\begin{align*}
&= (y' = x \land x' = x \land z' = z) \cdot (z' = y \land x' = x \land y' = y) \\
&= \exists x'', y'', z''. \; (y'' = x \land x'' = x \land z'' = z) \land (z'' = y'' \land x' = x'' \land y' = y'') \\
&= x' = x \land y' = x \land z' = x
\end{align*}
\]

The following is an instance of the \textit{substitution law}. Often we have a dependent composition \( x := e \cdot P \) of an assignment statement \( x := e \) (where \( e \) is an expression involving no post-value of the program variables) followed by a specification \( P \). The substitution law states that it can be simplified to \( P[e/x] \):

\[
x := e \cdot P
\]

\[
\begin{align*}
&= (x' = e \land y' = y \land z' = z) \cdot P
\end{align*}
\]
\[
\begin{align*}
\text{Iteration is discussed in the next section.}
\end{align*}
\]

\subsection{2.3 Refinement}

We now need a criterion to determine if a certain specification or program "implements" another specification or program. Our formalization of this concept is called \textit{refinement}, and it is based on the following idea. A specification is a predicate \( P \) describing how the initial values and final values are related. If we purport that another specification \( Q \) implements \( P \), then at least we must ensure that \( Q \) does what \( P \) does, i.e., the input-output relation in \( Q \) satisfies that in \( P \). Thus, we define "\( P \) is refined by \( Q \)" as logical implication:

\[
P \leftarrow Q
\]

As a simple example, we have:

\[
y' = x^2 \leftarrow y := x^2
\]

If exponentiation for natural numbers is supported by the underlying computer, we may declare to have finished the programming. Otherwise, we may go one step further:

\[
y := x^2 \leftarrow y := x \times x
\]

Often we wish to refine a specification by an iterative or recursive algorithm. For example, to set \( x \) to 0, we may decrease it until it hits 0. (Since \( x \) is a natural number variable here, we do not need to worry about negative numbers.) This is expressed in Hehner's theory by the recursive refinement:

\[
x' = 0 \Leftarrow \text{if } x = 0 \text{ then } \text{ok} \text{ else } (x := x - 1 . x' = 0)
\]
One can consider $x' = 0$ to be the name of a procedure that sets $x$ to 0; the above is then interpreted as implementing the procedure by tail recursion. Thus, loops in Hehner's theory are expressed as recursions, in particular tail recursions.

The followings are some theorems about the notion of refinement we have just defined. Since this notion is just logical implication, the proofs of these theorems are fairly simple, using essentially the partial order properties of implication.

1. Refinement by steps: Assume $A \leftarrow A'$ and $B \leftarrow B'$. Then the following hold:

(a) If we have $P \leftarrow A \cdot B$, then we have $P \leftarrow A' \cdot B'$

(b) If we have $P \leftarrow \text{if } b \text{ then } A \text{ else } B$, then we have $P \leftarrow \text{if } b \text{ then } A' \text{ else } B'$

(c) If we have $P \leftarrow A$, then we have $P \leftarrow A'$

2. Refinement by parts:

(a) If we have $P_1 \leftarrow (A_1 \cdot B_1)$ and $P_2 \leftarrow (A_2 \cdot B_2)$, then we have $P_1 \land P_2 \leftarrow (A_1 \land A_2 \cdot B_1 \land B_2)$

(b) If we have $P_1 \leftarrow \text{if } b \text{ then } A_1 \text{ else } B_1$ and $P_2 \leftarrow \text{if } b \text{ then } A_2 \text{ else } B_2$, then we have $P_1 \land P_2 \leftarrow \text{if } b \text{ then } A_1 \land A_2 \text{ else } B_1 \land B_2$

(c) If we have $P_1 \leftarrow A_1$ and $P_2 \leftarrow A_2$, then we have $P_1 \land P_2 \leftarrow A_1 \land A_2$

3. Refinement by cases: We have

$$P \leftarrow \text{if } b \text{ then } A \text{ else } B$$

if and only if we have $P \leftarrow b \land A$ and $P \leftarrow \neg b \land B$

Expositions of other aspects of Hehner's theory, including execution time, data structures, etc., can be found in [11, 12].
Chapter 3

Translations to and from the Proof Checker

This chapter describes the translation of expressions between the editor, which uses notations in Hehner’s theory, and the proof checker, which uses notations based on higher-order logic. The first section describes the translation of Hehner’s theory into logic in general. The second section describes the implementation of the translation.

3.1 Embedding Hehner’s Theory in Logic

While the embedding of Hehner’s theory into the proof checker seems to be trivialized by the fact that Hehner’s theory requires only first-order logic, there are some catches as we will see. This section describes the problems and the solution.

3.1.1 Mathematics vs. Logic

In Hehner’s theory, symbols such as “ok” (no operation), “.” (dependent composition), and “:=” (assignment) represent different expressions in different contexts, depending on the program variables present, say x and y (and therefore x’ and y’ as well). One may
like to think of them as meta-symbols.

To a mathematician, this is fine. We often do things like: let \( s = t^2 \), then the derivative of \( s \) is \( \dot{s} = ds/dt = 2t \). We say that \( t \) is an independent variable, and \( s \) is a variable dependent on \( t \): whenever we write \( s \), we make a mental note that it involves \( t \) once expanded. Likewise, when we write \( x := 1 \), we make a mental note that the variables \( x', y, \) and \( y' \) are also involved.

But to a logician or a contemporary proof checker, there is no such thing as independent variable or dependent variable. The orthodox way is: let \( t \) be a dummy variable and \( s \) be a function, with \( s = \lambda t \cdot t^2 \); then we have the derivative \( \dot{s} = \lambda t \cdot 2t \).

The difference in these two views is fundamental. A mathematician knows what he/she is doing, and at the same time seeks concise notations; therefore, it is fine and in fact desirable to create two free variables \( x \) and \( y \), and declare that \( y \) means an expression involving \( x \). A logician, however, performs proofs in a purely syntactic manner, relying heavily on substitution; therefore, hidden relations between free variables are intolerable, as they complicate substitution.

### 3.1.2 Translating Mathematics into Logic

Since a proof checker resembles a logician, we need to handle "ok", "=" and "\(:=\)" specially. In the following we list several options we have. For the purpose of illustration, let us assume there are two state variables \( x \) and \( y \).

1. Lift to higher orders, e.g., the proof checker is told that \( \text{ok} \) is the function

   \[
   \lambda x, x', y, y' \cdot x' = x \land y' = y
   \]

   More examples are illustrated in the following table:
So each boolean expression in Hehner's theory is translated to a function determined by the state variables. At a grander scale, each boolean expression becomes a family of functions in which members differ by state variables, and the appropriate member is chosen at the time of translation.

Lifting seems to be a natural approach, from the perspective described in previous paragraphs. However, many of the useful pre-proved theorems and inference rules in the proof checker are geared towards first-order logic; lifting would then become an obstacle to practical usage: existing theorems that a user wishes to use need to be lifted and re-proved.

2. Deal with meta-symbols without the proof checker. Expressions involving \(ok\), \(\cdot\), and \(:=\) are handled and simplified by the editor, which is programmed with known laws about them such as:

(a) \(ok \cdot P = P\)

(b) \(P \cdot ok = P\)

(c) \(x := e \cdot P = P[e/x]\)

This leads to two significant systems of logic—one of the proof checker and the other of the editor. This is unnecessary work and, in a sense, re-inventing the wheel.
3. Expand to predicates. Expressions involving the meta-symbols are expanded to plain first-order formulae before they are presented to the proof checker, e.g.,

<table>
<thead>
<tr>
<th>Editor</th>
<th>Proof Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>$x' = x \land y' = y$</td>
</tr>
<tr>
<td>$x &lt; x'$</td>
<td>$x &lt; x'$</td>
</tr>
<tr>
<td>$x &lt; x' \land y &lt; y'$</td>
<td>$\exists x'', y''. x &lt; x'' \land y'' &lt; y'$</td>
</tr>
<tr>
<td>$x := x + 1$</td>
<td>substitute $x + 1$ for $x$ in $x' = x \land y' = y$</td>
</tr>
</tbody>
</table>

Here the problem is that a lot of useful laws will not be expressible; any use of such laws by the user must be translated to an ad hoc proof in the proof checker.

We choose the last option because the problem is readily solved by a general-purpose simplifier that comes with the proof checker. So even though the useful laws are not encoded as-is anywhere in the editor or the proof checker, the expected results of applying the laws can be obtained with equal ease with the simplifier. For example, suppose the following is entered:

$$x := 1 . \ y' = x$$

Then it will be translated to

$$\exists x'', y''. (\text{substitute } 1 \text{ for } x \text{ in } x'' = x \land y'' = y) \land y' = x''$$

This looks messy but is trivial to the simplifier, which knows how to make use of the conjunction of the equations to eliminate the existential quantifier. It is simplified to

$$y' = 1$$

which is the correct result, and it is as good as that of applying the substitution law $x := e . P = P[e/x]$. 
3.2 Implementation of the Translation

The editor and the proof checker, HOL, use different ways of presenting and representing expressions. In the editor (which is further described in Chapter 4), an expression is displayed in ordinary mathematical notations, such as:

\[ 0 + 1 \]

Inside the editor, this is represented as a tree in which operators sit in parent nodes and operands are child nodes, as shown in Figure 3.1.

```
+  \
0  1
```

Figure 3.1: An Expression Tree Inside the Editor

In HOL, an expression is displayed using almost ordinary mathematical notation, except that it is done in ASCII text, so some mathematical symbols are replaced by ASCII symbols. Nonetheless, the syntax remains largely the same. For example, the same expression above is displayed in HOL as:

\[ 0 + 1 \]

Inside HOL, this is represented as a tree as well, but in a different shape, as shown in Figure 3.2. The idea is that the internal nodes represent function applications and the leaf nodes represent values or functions. (This is to accommodate higher-order expressions.)

```
+  \
0  1
```

Figure 3.2: An Expression Tree Inside HOL

The editor and HOL communicate via a bi-directional text stream. This means that each side must flatten a tree structure into a text string and send it to the other side
where another tree structure is constructed. The following two subsections describe how this is done.

### 3.2.1 From the Editor to the Proof Checker

When the editor sends an expression to HOL for simplification or inference, the following two tasks must be accomplished in order for HOL to process the expression properly:

1. The expression needs to be translated into the HOL syntax to form an HOL formula.

2. Names such as "ok" need to be expanded into their first-order logic counterparts, as explained in Section 3.1.2, and the expansion depends on which program variables are present.

These are independent tasks, and we perform them separately.

The first task is fairly straightforward: the editor traverses an expression top-down and produces a string containing HOL symbols corresponding to the operators and variables used in the expression. For example, the editor translates this expression:

\[ ok \Rightarrow (x := x + 0 + 0) \]

into this string:

\[ \$\Rightarrow (ok) \ (\$:= (x : num) \ ((x : num) + (0) + (0))) \]

This example illustrates two aspects of our translation. First, since HOL accepts prefix notation as long as the $ sign is prepended to an operator, we take advantage of it to simplify our programming. So for example, HOL accepts both \( A \Rightarrow B \) (infix form) and \( \$\Rightarrow A \ B \) (prefix form) for what we know as \( A \Rightarrow B \), and we use the prefix form. However, we still use the infix notation for associative, commutative operators such as the addition in the above example.

Second, expansions of "ok", ":=", and "." are not done in this stage. They belong to the second task, which is explained now.
We write an ML module, *embedding.sml*, to perform the expansion. It contains a function `downcast` that takes an HOL formula, scans top-down for occurrences of `ok`, `:=`, and `;;` (for ".;"), and expands them. In order to perform the expansion according to the program variables present, it remembers the correct expansions internally. As an example, suppose the program variable is only `x`; then the expansions are:

- `ok` is expanded to `x'=x`; the latter is stored for use by `downcast`.
- `$ := (x) (E)` is expanded to `let x=E in x'=x`; this procedure is stored as a function to be called by `downcast`.
- `(P) ;; (Q)` is a bit convoluted: we take `P` out and replace `x'` by `x''` in it; and we take `Q` out and replace `x` by `x''` in it. Then we conjoin the results and prepend the conjunction by the existential quantifier `?x''` to it. For example,

\[(x' = x + 1) ;; (x' = x + 2)\]

becomes

\[?x''. \ (x'' = x + 1) \land (x' = x'' + 2)\]

This convoluted procedure is again stored as a function to be called by `downcast`.

Now, when the user adds program variables (section 4.1.2), these expansions need to be modified. This is done by the function `setme`: when it is called with a list of program variables, it re-computes and stores the new expansions. The editor calls this function whenever the user adds a new program variable.

Thus, the entire translation process is as follows. To pass an expression to HOL, the editor first produces a string that conforms to HOL syntax, then lets HOL parse it into a HOL formula (this is necessary because `downcast` expects a formula, not a string), and finally calls `downcast` with the parsed HOL formula as an argument to obtain the
expansion. After the expansion, we get a HOL formula that is ready for simplification or inference in HOL.

We round up this subsection with a simple example. Suppose \( x \) is the only program variable and we have this expression in the editor:

\[
ok \Rightarrow (x := x + 0 + 0)
\]

The editor computes the string

\[
$==>(ok) ($:= (x : num) ((x : num) + (0) + (0)))
\]

The editor then embeds this string into the command:

\[
downcast (--'$==>(ok) ($:= (x : num) ((x : num) + (0) + (0)))'--)
\]

This is equivalent to:

\[
downcast (--'ok ==> (x := x + 0 + 0)'--)
\]

After downcast, the result is:

\[
(x' = x) ==> (let x = x + 0 + 0 in x' = x)
\]

3.2.2 From the Proof Checker to the Editor

When HOL finishes a simplification or inference and produces a result like:

\[
\neg (x = 0) ==> b
\]

the editor needs to translate it and display the corresponding expression:

\[
\neg x = 0 \Rightarrow b
\]

We would like to avoid context-free grammar parsing in this process. Like our editor, HOL represents a formula by a tree internally as well (though in a different form); if we are to flatten it, we should try to preserve the tree structure as much as possible, for example by using postfix notation with spaces delimiting tokens:
\[ b \ 0 \ x = ^\neg \Rightarrow \]

Here the order of the operands are reversed for implementational reasons. Assuming that we know the arities of the operators, we can reconstruct the expression tree. Although this assumption holds currently, it would be nice to augment this scheme to support higher-order curried functions in the future, in which case arities no longer disambiguate subexpression boundaries. Thus, we insert extra dots after each operator, with the number of dots indicating how many arguments the operator should take:

\[ b \ 0 \ x = \ldots ^\neg \ldots \Rightarrow \ldots \]

The generation of this string is precisely the purpose of the ML module `termlist.sml`. It has a function `postfix` that takes a HOL formula, uses HOL built-in functions to extract operators and operands recursively, and re-order them by post-order to yield a string of the above form.

Thus, translating a HOL formula back to the editor works as follows. The editor does not read the formula directly. Instead, it lets the `postfix` function intercept the formula and produce the postfix string; then the editor tokenizes the string and creates an expression according to these tokens. This is much simpler to implement than parsing a context-free grammar.
Chapter 4

The Editor

This chapter describes the user side of the tool: the editor. The first section is essentially a user guide of the editor. The second section summarizes the design of the editor.

4.1 Using the Editor

This section is a brief user guide to the tool for formal refinement method, as a complement to the demonstration given to the two readers of this thesis.

The editor has three windows: Stencil, Variables, and Editor.

4.1.1 The Stencil Window

The Stencil window displays a list of available operators (such as negation) and constants (such as true and false). It is shown in Figure 4.1. The operators are displayed as if they are appearing in an expression, except that the operands are blank. For example, the conditional operator is displayed like:

\[
\text{if } \_ \text{ then } \_ \text{ else } \_
\]

This shows the relative positions of the operator and its operands. (It is also easier to implement: it can share a significant portion of code with that of the Editor window.)
Clicking on an operator inserts it into the Editor window.

Currently, only the operators for natural numbers and booleans are supported. In the future, operators for other types such as lists will be added.

4.1.2 The Variables Window

The Variables window displays a list of the currently available variables. It is shown in Figure 4.2. Initially, it is empty. To create a variable, select Edit then Add, then
enter the name and the type of the desired variable in the dialog box. (Currently, only two HOL types, num for natural numbers and bool for booleans, are supported.) Two variables will be created: the “pre” variable and the “post” variable. For example, if \( b \) is entered as the name, then both \( b \) and \( b' \) will be created. Afterwards, clicking on a variable in this window inserts it into the Editor window.

4.1.3 The Editor Window

The Editor window is the place for editing expressions and performing proofs. It is shown in Figure 4.3.

![Figure 4.3: The Editor Window (with \( y := x \) . \( z := y \) entered)](image)

The method of expression editing follows that of MathJax [6, 5, 4]. Essentially, it is syntax-directed: the editor treats expressions as trees and the user edits them by tree operations. In addition, they are displayed with proportional spacing: a longer subexpression has more space before and after it, which aids the user in recognizing the
The basic operation of editing is navigating through existing expressions. Clicking with the left mouse button on an operator will select the (sub-)expression rooted at it. (As a degenerate case, clicking on a variable or constant will select it.) When an expression is selected, it will be highlighted in black and stay that way until another expression is selected. To unselect an expression without selecting any other expression, click on any large area of empty space in the window.

The cursor keys on the keyboard are also useful in the navigation when an expression is already selected. The up arrow key selects the parent, the down arrow key selects the first child, the right arrow key selects the sibling to the right, and the left arrow key selects the sibling to the left. All four of the above statements should be suffixed by the proviso "if any".

The user begins entering an expression in one of the following ways:

- clicking on an operator from the Stencil window, which enters the operator and blanks for the operands
- clicking on a variable from the Variables window, which enters the variable
- selecting Add under the Edit menu, which enters a blank

Then the user can proceed in one of the following ways:

- filling in a blank: select the blank, then choose an operator from the Stencil window or a variable from the Variable window, which will be used to fill the blank
- subsuming an expression: select the expression (click on the topmost operator or use cursor keys), then choose an operator from the Stencil window, thus entering the operator and making the expression the first operand

For example, to enter $y := x \cdot z := y$, we may follow these steps:
1. add \( x \), \( y \), and \( z \) to the Variables window

2. enter \( y \) (choose it from the Variables window):
   
   \[ y \]

3. enter the assignment operator (choose it from the Stencil window):
   
   \[ y := \]

4. enter \( x \):
   
   \[ y := x \]

5. select the entire expression (click on the assignment operator or press the up arrow key), then enter the dependent composition operator:
   
   \[ y := x \cdot \]

6. enter \( z \):
   
   \[ y := x \cdot z \]

7. enter the assignment operator
   
   \[ y := x \cdot z := \]

8. enter \( y \):
   
   \[ y := x \cdot z := y \]

The editor recognizes associative commutative operators, such as \(+\) and \(\lor\), and treats them a bit specially. An associative commutative operator can have two or more operands, whereas other operators have a fixed number of operands each. To add operands, just perform normal entering as one would guess: while an operand or a blank is selected, choose the operator from the Stencil window. This will keep the original operand as well as insert a blank and one more copy of the operator. To remove
operands, use the Delete menu item described below as usual: select an operand, then apply Delete once or twice to remove it. (The reason for needing to apply Delete twice will become apparent when the menu items are described.)

The rest of this section describes the menus in the Editor Window: File, Edit, and Prover.

The File Menu

The File Menu contains the following items:

Print Prints the content of the Editor window.

New Window Opens a new Editor window. The program can maintain more than one Editor window.

Close Closes the Editor window. If there is no other Editor window, the program exits.

Exit Exits the program.

The Edit Menu

The Edit menu contains the following items:

Add Inserts a blank expression at the end of the window.

Constant Replaces the selection with a constant, or adds a constant at the end of the window if there is no selection. This is mainly used for entering numeric constants, which cannot be practically made available in the Stencil window. The editor will display a dialog box to prompt for the constant.

Delete Deletes the selected expression, if any. If the selection was not a blank, it now becomes a blank. If, on the other hand, the selection was a blank and it is sensible to remove it (it has no parent, or it is an operand of an associative commutative operator), it is now totally removed.
Copy Makes a copy of an expression. Select the source with the right mouse button (this highlights it in blue), and select the target with the left mouse button (this highlights it in black). Now click on Copy. The target will be replaced by a copy of the source.

Swap Swaps two expressions. Select one expression with the left mouse button, and select the other with the right mouse button. Now click on Swap. The two will switch places.

Group This and the next item are for expressions using associative commutative operators. The Group item groups several consecutive terms into a subexpression, effectively putting a pair of parentheses around them. For example, suppose we have this expression:

\[ s + t + u + v \]

This expression has four operands: \( s, t, u, \) and \( v \). Now select \( t \) with the left mouse button, and \( u \) with the right mouse button. Then select the Group menu item. The expression will become:

\[ s + (t + u) + v \]

A little bit of navigation reveals that now the operands are: \( s \), the whole of \( (t + u) \), and \( v \).

Ungroup Undoes the effect of Group. Using the above example, select the subexpression \( (t + u) \), then select Ungroup. Now it is restored to

\[ s + t + u + v \]

Split Breaks the selected expression into multiple lines; the line breaks are at the operator.

Flat Undoes the effect of Split: restores the selected expression back to a single line if it was splitted.
The Edit menu can be "torn off" on supporting platforms, e.g., most X Window systems, but not available on Windows. This means that when the menu pops up, the user may click near the dotted line between the menu bar and the menu, and the menu will be detached as a separate window. Now the user may move it to a convenient place and use it at will, not needing to click on Edit every time he/she wants an item in the menu.

The Prover Menu

The Prover menu contains one item:

Simplify Requests the proof checker to simplify the selected expression (if any). The result of the simplification is added back to the Editor window.

When more proof aids are added to the tool in the future, they will generally go under the Prover menu as well.

Like the Edit menu, the Prover menu can be "torn off" on supporting platforms.

As an example of the use of the Simplify item, suppose the expression

\[ y := x \cdot z := y \]

has been entered as in page 24. Now select the entire expression, and then select the Simplify menu item. The result then appears in the Editor window:

\[ x' = x \land (y' = x \land z' = x) \]

which is the same as that on page 9 in Section 2.2. To get rid of the parentheses, use the Ungroup function described in the previous subsection.

Simplification can be used as a means for doing simple proofs: to prove a statement, the user can request Simplify to work on it, and see if it reduces to truth (displayed as T). For example, enter this statement:

\[ (y := x \cdot z := y) = (x' = x \land y' = x \land z' = x) \]
To prove this statement, select it and then select the Simplify menu item. The result will be

\[ T \]

A bigger example of using Simplify in the context of program refinement is in Chapter 5.

4.2 The Design of the Editor

Object-oriented design and programming are employed to give the editor and its various components a modular, extensible, and adaptable structure. The Java programming language [1] is chosen as the implementation language for its garbage collection memory management scheme, which relieves us from a lot of error-prone chore, and its simple library of graphical user interface, which eliminates a substantial amount of mundane coding.

4.2.1 Representing Expressions

Expressions are the most important objects in the editor, as it is a syntax-directed editor keeping track of expression structures. They are represented by the class hierarchy of Operand:

- **Operand**: the abstract superclass for all expressions
  - **Identifier**: leaf node in the tree: constant and variable
    * **Variable**: typed variable (we do not need to keep track of the types of constants)
  - **Expression**: internal node in the tree: expression consisting of an **Operator** and at least one **Operand**
    * **ExpressionNary**: n-ary expression, with variable number of operands
To illustrate the purposes of these classes, consider the expression \(0 + 1\). It is represented by the object structure in Figure 4.4.

![Figure 4.4: An Expression Tree](image)

Operand is in turn a subclass of OperandVector, motivated by the following consideration. Suppose the user selects an expression and wishes to delete it. This requires us to locate the parent of the selected expression and remove the child from the parent. Now, the parent could be an expression or an editor window containing expressions. In order for deletion and similar editing operations to work uniformly across both cases, we put expressions and other expression-containing types under the common superclass OperandVector—container of expressions.

### 4.2.2 Displaying Expressions

The classes for displaying expressions are distinct from those in the last subsection representing expressions; this is an application of the model-view-controller approach to graphical user interface programming: one class hierarchy is responsible for representing data inside the program (model), another for displaying data to the user (view), and the third for manipulating data according to user input (controller). Thus, classes in the Operand hierarchy are model classes.

The following classes are the view classes; except for AtomView, they are all subtypes of the OperandView interface for uniformity:

- **AtomView**: displays a single identifier or operator symbol
- **NullView**: displays an empty operand

- **IdentifierView**: displays a single identifier

  * **VariableView**: displays a single variable; differs from IdentifierView by the font used

- **ExpressionView**: displays an expression with an operator and at least one operand; therefore, it contains other OperandView objects as children

- **ExpressionNaryView**: displays an n-ary expression

Evidently, the above class hierarchy mirrors the Operand hierarchy closely. The dynamic object structures are also very similar: a tree of OperandView objects corresponds to an expression tree of Operand objects being displayed.

There is also an OperandVectorView interface to go with the OperandVector class.

### 4.2.3 Traversing the Expression Tree

Traversing an expression tree is a frequent operation in the program: it must be done when creating a tree of view objects corresponding to the expression tree and when communicating the expression to the proof checker. Complicating this traversal is the fact that different actions must be performed when different types of nodes are encountered. For example, to display an expression, we do the following: when we see an Expression node, we create an ExpressionView object; when we see an Identifier node, we create an IdentifierView object, etc.

The OperandVisitor interface defines a uniform framework for performing different operations on different types of nodes. It specifies one method for each type of node, e.g., visitExpression for working on an Expression node, visitIdentifier for working on an Identifier node, etc. Thus, to display an expression, we implement the class OperandViewFactory, which subclasses OperandVisitor. Its visitExpression
method creates an ExpressionView object, its visitIdentifier method creates an IdentifierView object, etc. Similarly, the class TermBuilder is responsible for translating expressions to proof checker notation, and it achieves this by being a subclass of OperandVisitor and implementing the translation as operations over different types of nodes.

### 4.2.4 Operators

Operators in the program are represented by subtypes of the Operator interface. Each operator has attributes, such as precedence, associativity, commutativity, and arity, that affect the behaviour of Expression objects during runtime, e.g., an Expression object determines the number of children it ought to have by looking at the arity of its Operator object.

Display formats of the operators are encoded in the Embedding class. An Embedding object knows the display name of an operator as well as the display positions of operands, and provides these information to an ExpressionView object for proper rendering. The subclasses of Embedding are: EmbeddingFixed, for operators with fixed numbers of operands; and EmbeddingNary, for operators with variable numbers of operands. The mapping from Operator to Embedding is maintained by the EmbeddingFactory class.

Names of operators to be used when communicating with the proof checker are maintained by two mappings, one for each way of lookup, in the TermBuilder class.

### 4.2.5 Other Parts of the Editor

Work is a subclass of OperandVector that contains expressions to be displayed in an application window. Workspace is the view class for Work, responsible for the actual display. Evidently, each Stencil, Variables, or Editor window contains a Workspace object and gives it the bulk of the display area.

Palette is an application window that displays template expressions or variables for
the user to pick from. When the user picks one, a copy is made and inserted into the Editor window, thus implementing the behaviour described in Sections 4.1.1 and 4.1.2. Stencil and Variables are subclasses of Palette implementing the Stencil window and the Variables window, respectively.

Editor is the main editor window for editing expressions and giving commands to the proof checker. It implements the bulk of the functionalities described in section 4.1.3. ProofMenu, with the help of ProverFrame and ProverShell, implements the commands under the Prover menu.

Both Palette and Editor display expressions, or more technically, the OperandView objects for the expressions. Each view object receives user input that are mouse and keyboard events, and, depending on to which window it belongs, it must respond in a certain fashion. For example, clicking on an expression in the Stencil window inserts it into the Editor window, but clicking on an expression in the Editor window just selects it. The problem is that modularity would be broken if this behaviour were implemented inside the view classes.

This is where the controller in the model-view-controller approach comes into play: each view object is given a controller object, and when the view object receives user input, it simply passes the input message to the controller object, which will respond to the input. (This structure is supported and encouraged by Java.) View objects inside a Stencil or Variables window are given controller objects of the PaletteOperandControl class (an inner class of Palette), and view objects inside an Editor window are given controller objects of the EditorOperandControl class (an inner class of Editor). Both controller classes are subclasses of OperandControl for uniformity. Now the expressions respond to input correctly, and new responses can be implemented without affecting the view classes.
Chapter 5

An Example of Refinement

This chapter presents and briefly discusses an example of constructing a program by refinement using the tool. To keep it down to reasonable length, we do not describe every user operation in detail; instead, we just say “enter this expression”, “use Simplify”, “the result is this”, etc. (These user operations are described generally in the user guide in Section 4.1.)

A positive natural number is always the product of an odd number and a power of 2; we wish to find them. Thus, given \( n \), we have the specification:

\[
n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0
\]

so that \( y' \) has the odd factor and \( x' \) has the correct index of 2 at the end. So our program variables are \( n \), \( x \), and \( y \), all of type \texttt{num}. We add these program variables to the Variables window. Then we enter the expression above to the Editor window.

Our strategy is to start with \( x = 0 \) and \( y = n \); then we successively pull out factors of 2 from \( y \) and increment \( x \) at the same time, maintaining the relation \( 2^x \times y = n \). So at the end, when \( y \) becomes odd, we are done.

Thus the first refinement is:

\[
x := 0 \quad y := n \quad 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)
\]

\[
\Rightarrow n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0
\]
We add this expression to the Editor window. (Note that a large part of it is already entered previously, so we would minimize our effort by maximizing copying.) It is readily proved by applying Simplify: it performs the equivalent of the substitution law, reduces the antecedent $2^x \times n = n$ to T, and finds that both sides of the implication are the same. The result of applying Simplify to this refinement is T; thus the refinement is proven.

Next, we refine

$$2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0)$$

by a conditional statement: if $y$ is odd, we are done; otherwise, we have more work to do, but we can do it using the knowledge that $y$ is even. So the refinement is:

```plaintext
if \neg y \text{ mod } 2 = 0
then ok
else y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0))
\Rightarrow 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0)
```

But if we apply Simplify to this refinement, the result is essentially the same expression, not T, meaning that Simplify is unable to prove it. Simplify is just an equational rewriting system with some decision procedure for linear arithmetic, but not an inference engine. To prove the above refinement, however, one needs to recognize that there are two steps in the reasoning:

1.

```plaintext
if \neg y \text{ mod } 2 = 0
then \neg y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0))
else y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0))
\Rightarrow 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \text{ mod } 2 = 0)
```
2.

\[ \text{ok} \]

\[ \Rightarrow \neg y \mod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

Now we can enter each of the above two refinements into the Editor window (with the help of copying, again), apply Simplify, and get it proved automatically. In the first case, Simplify has a rewrite rule that precisely reduces if-then-else expressions like the one on the left hand side to the expression on the right hand side. In the second case, Simplify uses the equations \( x' = x \) and \( y' = y \) provided by \text{ok} to replace all other occurrences of \( x' \) and \( y' \), and then use antecedents to simplify subexpressions in consequences to \( T \) whenever they are identical.

Lastly, we refine the other branch of the conditional statement:

\[ y \mod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

Here we are sure that \( y \) is even, so we can divide it by 2 and update \( x \) to preserve the relation \( 2^x \times y = n \). Afterwards, a tail recursion will finish the job. So the refinement is:

\[ y := y \div 2 \cdot x := x + 1 \cdot (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

\[ \Rightarrow y \mod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

Now, if we enter this refinement and apply Simplify to it, we find that Simplify is unable to prove it by reducing it to \( T \). The proof of this refinement requires several intelligent steps and is beyond the capability of Simplify. We first apply Simplify anyway to perform the equivalent of the substitution law, obtaining:

\[ (2^{x+1} \times y \div 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

\[ \Rightarrow y \mod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \land \neg y' \mod 2 = 0)) \]

The rest of the proof is mostly manual; this means that the user goes ahead to edit expressions at will, and the tool offers no aid to automation or verification.
There are many ways to finish the proof. Here, we choose a course of action that is most likely to be supported by a computer proof checker. On the right hand side, in $2^x \times y = n$, rewrite $y$ as $(y \text{ div } 2) \times 2 + y \text{ mod } 2$ by a theorem of natural number division. This is a complexification step: it rewrites a subexpression to a longer equivalent. There are two characteristics of complexification steps:

- the rewrite rule used is non-terminating, so the number of applications is carefully chosen

- the term to be rewritten is also carefully chosen

The choices made require judgement using knowledge and techniques in the problem domain, e.g., number theory in this case. Because of this, complexification steps are best performed by humans, and the desirable role of the computer in these situations is to provide proof checking.

Next, notice that the term $y \text{ mod } 2$ can be discarded thanks to the antecedent $y \text{ mod } 2 = 0$. So we get:

$$(2^{x+1} \times y \text{ div } 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^x \times y' = n \land \neg y' \text{ mod } 2 = 0))$$

$\Rightarrow y \text{ mod } 2 = 0 \Rightarrow (2^x \times (y \text{ div } 2) \times 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^x \times y' = n \land \neg y' \text{ mod } 2 = 0))$

To many humans, this is now obviously a theorem. To a computer or a human who demands more detail, it is probably not “obvious” yet, but two extra small steps will prove it. Rewrite $2^x \times (y \text{ div } 2) \times 2$ as $2^{x+1} \times y \text{ div } 2$ by arithmetic, yielding:

$$(2^{x+1} \times y \text{ div } 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^x \times y' = n \land \neg y' \text{ mod } 2 = 0))$$

$\Rightarrow y \text{ mod } 2 = 0 \Rightarrow (2^{x+1} \times (y \text{ div } 2) = n \Rightarrow (n \geq 1 \Rightarrow 2^x \times y' = n \land \neg y' \text{ mod } 2 = 0))$

Now this is a tautology of the form $P \Rightarrow (Q \Rightarrow P)$. We can apply Simplify to it, obtaining the result $T$, thus proving it. Alternatively, if we had tautology checkers for propositional or first-order logic, we could use them as well.
So Simplify is capable of resolving linear arithmetic, constant folding, and contextual simplification, but is inadequate for more interesting proofs requiring complexification and advanced techniques in the theory of the problem domain (such as the way the division theorem of number theory is used in our example). For this reason, other functions should be provided for validating a proof developed by the user if the user breaks it down into sufficient detail:

- Marking an expression as a theorem after it is proved.

- Inference commands to apply a theorem (built-in or user-proven) to transform an expression or a subexpression. This is for proving that the old expression is equal to, implies, or is implied by the new expression, depending on the inference command and the theorem used. Window inference of Grundy [10, 2] combined with Simplify is a likely choice for this.

- Verification commands to check tautologies in propositional logic and first-order logic.

These will be added in the future. A reason they are not done now is that the way these should be presented to the user needs to be carefully thought out—the last thing we want is a mere translation of HOL commands, most of which are unnatural to use, to menu items in our tool.
Chapter 6

Conclusion

We have built a tool for a formal refinement method, namely, Hehner's theory, consisting of a syntax-directed editor and a proof checker. The user can:

1. Edit expressions and proofs that are pretty-printed in familiar forms and symbols, and the editing is done in terms of subexpression manipulation.

2. Use the simplifier provided by the proof checker to simplify expressions as a means of proof or, when that fails, as a small step involved in a proof.

In order for the tool to fully support formal refinement methods, we still need to add these functions:

1. Provide inference rules, theorems, and other automated and semi-automated proof methods (such as tautology checkers) besides simplification to the user, and allow the user to use a theorem he/she has proved in addition to built-in theorems, so that the suite of proof aids is more complete and capable of verifying all valid proofs developed by the user.

The proof checker we use, HOL, has most of the desired capabilities. We need to find a natural way of interfacing them with the user via the editor. We are considering some unification of goal-directed proofs (backward inference), supported by HOL.
and popular among the HOL users, and structured calculational proof [2] as a possible framework for this.

The editor also needs to mark proved theorems as proved and remember them.

2. Suggest useful theorems for use in a proof, upon user request.

This requires the development or adoption of some matching algorithm based on the method of resolution proofs in the first-order logic.

3. Identify unproved refinements upon user request.

This requires the editor to identify refinements, and distinguish proved theorems from unproved claims.

4. Keep and display the proofs, i.e., the inference commands used and the intermediate results obtained.

The storing and the display of proofs are closely related to the actions performed in developing the proofs. Thus, the framework for proving adopted in item 1 above will determine the format of storing and displaying proofs. In particular, structured calculation proof [2] is both a proof system and a proof format, and it may be adopted here.

5. Translate completed refinements into program code, or identify incomplete refinements, upon user's request.

This requires the editor to identify refinements and distinguish concrete (code-level) specifications from other specifications.

Other less critical but nonetheless useful functions worth implementing are: Save and Load functions to store and retrieve previous work of the user; a way for the user to enter comments and explanations; a way for the user to enter operators and variables from the keyboard (by typing in \TeX-like names, for example) in case he/she does not want to
use the mouse; and translation of the work of the user into $\LaTeX$ code for inclusion into other documents.
Bibliography


[7] M. Butler, J. Grundy, T. Långbacka, R. Rukšenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In Lindsay Groves and


Appendix A

Source Code Listing
package scphEditor;
import java.awt.*;
import java.awt.event.*;

/**
 * This AWT component is the view of a simple string, e.g., the name
 * of an Identifier, a part of an operator. The string is considered
 * immutable, therefore this class is not an observer. However,
 * protected methods are provided to access this string, so subclasses
 * can be written to be observers.
 *<p>
The default font is null; i.e., follows the parent container
 * dynamically.
 */

public class AtomView extends Component
{

private String text;
private Image printimage;
private boolean simpleprint;

/**
 * Creates an AtomView object that displays the given string.
 */
public AtomView(String textOf)
{  
super();
text = textOf;
printimage = null;
}

/**
 * Gives the string displayed by this view.
 */
protected String getText()
{  
return text;
}

/**
 * Changes the string displayed by this view. And redraws itself,
 * of course.
 */
protected void setText(String newText)
{  
text = newText;
printimage = null;
invalidate();
// needs to validate() the ancestors
repaint();
}

/**
 * Returns the display size of this AWT component. Both
 * getMinimumSize() and getPreferredSize() return the same thing.
 *<p>
 * @param g the graphics context in which the drawing is done.
 */
public void paint(Graphics g)
{  
Dimension size = getSize();
g.setColor(getBackground());
g.fillRect(0, 0, size.width, size.height);
g.setColor(getForeground());
g.fillRect(0, 0, size.width, size.height);
g.setColor(getForeground());
g.fillRect(0, 0, size.width, size.height);
}

/**
 * Returns display size of this component. And redraws itself,
 * of course.
 */
protected void paint(Graphics g)
{  
Dimension size = getSize();
g.fillRect(0, 0, size.width, size.height);
g.fillRect(0, 0, size.width, size.height);
g.fillRect(0, 0, size.width, size.height);
}

private boolean simpleprint()
{  
for (int i = 0; i < text.length(); i++)
{  
if (text.charAt(i) >= '\u0100') return false;
}
return true;
}

/*
 * @(#)AtomView.java
 */

package scphEditor;

import java.util.Vector;
import java.util.Observable;
import java.util.Enumeration;
import java.util.Observable;

/**
 * This class re-implements java.util.Observable so it can be cloned safely.
 * But there can be two meanings for cloning an observable:
 * <ul>
 * <li>The clone has no observers to begin with</li>
 * <li>The clone has all of the observers of the original to begin with</li>
 * </ul>
 * You can choose which one you want when you call the constructor.
 * (The default is the first meaning; empty list.) In addition,
 * you can change your mind any time in the program flow with the
 * cloneObserverList() method.
 * // This class does not implement java.lang.Cloneable! Reason: you might
 * not want to support cloning in some of your subclasses. But this class
 * provides a correctly-written protected clone() method, so your
 * subclasses and the subclasses of your subclasses can rely on it. In
 * other words, this class does not open up access until you want to,
 * and provides proper support when you do open up access.
 * // This class subclasses Observable, so that Observer objects cannot
 * tell the difference. But all Observable methods accessing the observer
 * list are overridden and re-implemented from scratch. Only the change
 * methods are retained.
 * @see CloneableObservable#cloneObserverList(boolean)
 * @see java.util.Observable
 */

public class CloneableObservable extends Observable
{
    private boolean cloneObservers;
    private Vector observers;

    /**
     * Default constructor.
     */
    public CloneableObservable()
    {
        this(false);
    }

    /**
     * Constructor specifying the copy policy for the observer list.
     * @param cloneObserverList true if you want a clone to duplicate the
     * observer list of the original; false if you want the observer
     * list of a clone to be empty.
     */
    public CloneableObservable(boolean cloneObserverList)
    {
        super();
        observers = new Vector();
        cloneObservers = cloneObserverList;
        }

    protected synchronized Object clone()
    {
        CloneableObservable c;
        try {
            c = (CloneableObservable) super.clone();
        }
        catch (CloneNotSupportedException e) {
            // should not happen
            throw new InternalError(e.toString());
        }

        if (cloneObservers) {
            c.observers = (Vector) c.observers.clone();
        } else {
            c.observers = new Vector();
        }

        return c;
    }

    /**
     * Gives the observer-list cloning policy.
     */
    public synchronized boolean cloneObserverList()
    {
        return cloneObservers;
    }

    /**
     * Changes the observer-list cloning policy.
     * @param newPolicy true if you want a clone to duplicate the
     * observer list of the original; false if you want the observer
     * list of a clone to be empty.
     * @return the old policy. (Same semantics as the parameter.)
     */
    public synchronized boolean cloneObserverList(boolean newPolicy)
    {
        boolean r = cloneObservers;
        cloneObservers = newPolicy;
        return r;
    }

    // The following re-invents the Observable.

    /**
     * Adds an observer. No duplication check!
     * @param o the observer.
     */
public synchronized void addObserver(Observer o) {
    observers.addElement(o);
}

/**
 * Deletes an observer.
 * @param o the observer.
 */
public synchronized void deleteObserver(Observer o) {
    observers.removeElement(o);
}

/**
 * Clears the observer list.
 */
public synchronized void deleteObservers() {
    observers.removeAllElements();
}

/**
 * If this object is marked as changed (hasChanged() returns true),
 * mark itself as unchanged (with clearChanged()),
 * and then notify all of its observers.
 * Observers are notified by calling their update() methods with two parameters:
 * this object, and null.
 * @see CloneableObservable#hasChanged
 * @see CloneableObservable#clearChanged
 * @see java.util.Observer#update
 */
public void notifyObservers() {
    notifyObservers(null);
}

/**
 * If this object is marked as changed (hasChanged() returns true),
 * mark itself as unchanged (with clearChanged()),
 * and then notify all of its observers.
 * Observers are notified by calling their update() methods with two parameters:
 * this object, and the parameter arg.
 * @param arg the second parameter to be passed to update() methods of observers.
 * @see CloneableObservable#hasChanged
 * @see CloneableObservable#clearChanged
 * @see java.util.Observer#update
 */
public void notifyObservers(Object arg) {
    // this implementation is potentially thread-unsafe. But it does
    // not introduce deadlock. Any modification to make it thread-safe
    // should avoid the deadlock problem, which is subtle.
    if (hasChanged()) {
        clearChanged();
        for (Enumeration i = observers.elements(); i.hasMoreElements();) {
            Observer ob = (Observer) i.nextElement();
            ob.update(this, arg);
        }
    }
package scphEditor;
import java.awt.*;
import java.awt.event.*;
import java.util.Stack;
import java.util.Properties;

/**
 * Editor is the top-level window to display and edit the refinement work. It sets up the menu, obviously. Since this class has nothing else better to do, we combine both the view and the control here.
 * <p>It also defines the control class, as an inner class, appropriate for operands in an editor window.
 */
public class Editor extends Frame implements ActionListener, WindowListener {

    private static int countEditors = 0;
    private static Editor recentFocus = null;

    private Component selected, selected2;
    private Workspace myWorkspace;

    /**
     * Default Constructor.
     */
    public Editor()
    {
        super();
        setTitleEditors++;
        addWindowListener(this);

        myWorkspace = new Workspace(new WorkO, new EditorOperandControlO);
        ((RowLayout) myWorkspace.getLayoutO),
        setSpace(myWorkspace.getFontMetrics(myWorkspace.getFont()).getSpace());

        selected = null;
        selected2 = null;

        // the menu bar
        MenuBar mb = new MenuBar();
        setMenuBar(mb);

        Menu m; // scratch space
        // menu File
        m = new Menu("File");
m.add(newMenuItem("Exit", "Exit") );
        mb.add(m);
m.addActionListener(this);
        m.add(newMenuitem("Add", "Expr") );
        mb.add(newMenuitem("Constant", "Constant") );
        m.add(newMenuitem("Delete", "Del") );
        m.add(newMenuitem("Copy", "Copy") );
        m.add(newMenuitem("Swap", "Swap") );
        m.add(newMenuitem("Ungroup", "Ungroup") );
        m.add(newMenuitem("Group", "Group") );
        m.add(newMenuitem("Split", "Split") ); // provisionally here
        m.add(newMenuitem("Flat", "Flat") ); // provisionally here

        // menu Edit
        m = new Menu("Edit");
        mb.add(m);
m.addActionListener(this);
        m.add(newMenuItem("Add", "Expr") );
        m.add(newMenuItem("Constant", "Constant") );
        m.add(newMenuItem("Delete", "Del") );
        m.add(newMenuItem("Copy", "Copy") );
        m.add(newMenuItem("Swap", "Swap") );
        m.add(newMenuItem("Ungroup", "Ungroup") );
        m.add(newMenuItem("Group", "Group") );
        m.add(newMenuItem("Split", "Split") ); // provisionally here
        m.add(newMenuItem("Flat", "Flat") ); // provisionally here

        // menu Prover
        m = new Menu("Prover");
        mb.add(m);
m.addActionListener(new ProofMenu());
        m.add(newMenuItem("Send", "Send") );
        m.add(newMenuItem("Simplify", "Simplify") );

    /**
     * Helper function for the constructor. Creates a menu item.
     * @param label displayed name of menu item.
     * @param command internal name (command) of menu item for use by action events.
     */
    private MenuItem newMenuItem(String label, String command)
    {
        MenuItem mi = new MenuItem(label);
        mi.setActionCommand(command);
        return mi;
    }

    public void actionPerformed(ActionEvent e)
    {
        String cmd = e.getActionCommand();
        if (cmd.equals("Close") )
        {
            closeEditor();
        }
        else if (cmd.equals("New") )
        {
            newEditor();
        }
        else if (cmd.equals("Print") )
        {
            doPrint();
        }
        else if (cmd.equals("Exit") )
        {
            doExit();
        }
        else if (cmd.equals("Ungroup") )
        {
            ungroup();
        }
        else if (cmd.equals("Group") )
        {
            group();
        }
        else if (cmd.equals("Constant") )
        {
            constant();
        }
        else if (cmd.equals("Expr") )
        {
            deselect();
            insert(null);
        }
        else if (cmd.equals("Constant") )
        {
            constant();
        }
        else if (cmd.equals("Expr") )
        {
            deselect();
            insert(null);
        }
private void doPrint()
{
    final Properties print_prop = new Properties();

    PrintJob job = getToolkit().getPrintJob(this, "scphEditor", print_prop);
    Graphics page = job.getGraphics();
    int resolution = job.getPageResolution();
    page.translate(resolution / 2, resolution / 2); // 1/2 inch
    page.setColor(Color.black);
    page.print(page);
    page.dispose();
    job.end();
}

private void closeEditor()
{
    countEditors--;
    setVisible(false);
    dispose();
    if (countEditors == 0) doExit();
}

private void doExit()
{
    System.exit(0);
}

/**
 * Helper function. Opens a new Editor window, sets up its size, * shows it.
 */
private static void newEditor()
{
    Editor e = new Editor();
    e.setSize(640, 480);
    e.show();
    recentFocus = e;
}

private void doPrint()
{
    final Properties print_prop = new Properties();

    PrintJob job = getToolkit().getPrintJob(this, "scphEditor", print_prop);
    Graphics page = job.getGraphics();
    int resolution = job.getPageResolution();
    page.translate(resolution / 2, resolution / 2); // 1/2 inch
    page.setColor(Color.black);
    page.print(page);
    page.dispose();
    job.end();
}

/*
 * Helper function. Closes an Editor window, and if this is the * last, exit the entire program.
 */
private void closeEditor()
{
    countEditors--;
    setVisible(false);
    dispose();
    if (countEditors == 0) doExit();
}

private void doExit()
{
    System.exit(0);
}

/**
 * Helper function. Opens a new Editor window, sets up its size, * shows it.
 */
private static void newEditor()
{
    Editor e = new Editor();
    e.setSize(640, 480);
    e.show();
    recentFocus = e;
}

private void doPrint()
{
    final Properties print_prop = new Properties();

    PrintJob job = getToolkit().getPrintJob(this, "scphEditor", print_prop);
    Graphics page = job.getGraphics();
    int resolution = job.getPageResolution();
    page.translate(resolution / 2, resolution / 2); // 1/2 inch
    page.setColor(Color.black);
    page.print(page);
    page.dispose();
    job.end();
}

// may belong to Expression
public void void insert(Operand o)
{
    if (selected == null) { // then easy
        OperandVector target = myWorkspace.getOperandVector();
        target.addChild(o, -1);
        Component c = myWorkspace.getOperandView(target.countChildren() - 1);
        myWorkspace.validate();
        selectAfterInsert(c);
    } else {
        OperandContext context = new OperandContext(selected);
        // could deselect() here but theoretically unnecessary
        context.parent.setChild(o, context.index);
        if (o.countChildren() > 0 && o.getChild(0) == null) {
            o.addChild(context.self, 0);
        }
        // ok, ugly to resort to instanceof, but ...
        if (context.parent instanceof ExpressionNary) {
            ((ExpressionNary) context.parent).ungroup(context.index);
            myWorkspace.validate();
            selectAfterInsert(((OperandVectorView) context.parentview).getOperandView(context.index));
        }
    }
}

// ungroup at the selection
public void ungroup()
{
    if (selected == null) return;
    OperandContext context = new OperandContext(selected);
    if (context.parent instanceof ExpressionNary) return;
    context.parentview.ungroup(context.index);
    deselect();
    myWorkspace.validate();
    myWorkspace.repaint();
}

public void group()
{
    if (selected == null || selected2 == null) return;
    OperandContext context = new OperandContext(selected);
    OperandContext context2 = new OperandContext(selected2);
    if (context.parent instanceof ExpressionNary) return;
    if (context.parent == context2.parent) return;
    if (context.index >= context2.index) return;
    // ok, finally :
    ((ExpressionNary) context.parent).group(context.index, context2.index - context.index + 1);
    deselect();
    deselect2();
    myWorkspace.validate();
    myWorkspace.repaint();
}

// delete the operand at the selection
public void delete()
{
    if (selected != null) {
        OperandContext context = new OperandContext(selected);
        if (context.self == null) {
            context.parent.delChild(context.index);
        } else if (context.parent instanceof ExpressionNary &&
        operand = context.parent.getChild(context.index);
        if (operand instanceof ExpressionNary) {
            ExpressionNary expression = (ExpressionNary) operand;
            List<Operand> children = expression.getChildren();
            int index = children.indexOf(operand);
            if (index != -1) {
                expression.removeChild(index);
            }
        }
    }
}

// insert, at the selection point or the end, a new expression // with the specified operator.
public void insert(Operand o)
{
    if (selected == null) {
        // easy
        OperandVector target = myWorkspace.getOperandVector();
        target.addChild(o, -1);
    } else {
        OperandContext context = new OperandContext(selected);
        // could deselect here but theoretically unnecessary
        context.parent.setChild(o, context.index);
        if (o.countChildren() > 0 && o.getChild(0) == null) {
            o.addChild(context.self, 0);
        }
        // ok, ugly to resort to instanceof, but ...
        if (context.parent instanceof ExpressionNary) {
            ((ExpressionNary) context.parent).ungroup(context.index);
            myWorkspace.validate();
            selectAfterInsert(((OperandVectorView) context.parentview).getOperandView(context.index));
        }
    }
}

// ungroup at the selection
public void ungroup()
{
    if (selected == null) return;
    OperandContext context = new OperandContext(selected);
    if (context.parent instanceof ExpressionNary) return;
    context.parentview.ungroup(context.index);
    deselect();
    myWorkspace.validate();
    myWorkspace.repaint();
}

public void group()
{
    if (selected == null || selected2 == null) return;
    OperandContext context = new OperandContext(selected);
    OperandContext context2 = new OperandContext(selected2);
    if (context.parent instanceof ExpressionNary) return;
    if (context.parent == context2.parent) return;
    if (context.index >= context2.index) return;
    // ok, finally :
    ((ExpressionNary) context.parent).group(context.index, context2.index - context.index + 1);
    deselect();
    deselect2();
    myWorkspace.validate();
    myWorkspace.repaint();
}

// delete the operand at the selection
public void delete()
{
    if (selected != null) {
        OperandContext context = new OperandContext(selected);
        if (context.self == null) {
            context.parent.delChild(context.index);
        } else if (context.parent instanceof ExpressionNary &&
    operand = context.parent.getChild(context.index);
    if (operand instanceof ExpressionNary) {
        ExpressionNary expression = (ExpressionNary) operand;
        List<Operand> children = expression.getChildren();
        int index = children.indexOf(operand);
        if (index != -1) {
            expression.removeChild(index);
        }
    }
}

// insert, at the selection point or the end, a new expression // with the specified operator.
public void insert(Operand o)
{
    if (selected == null) {
        // easy
        OperandVector target = myWorkspace.getOperandVector();
        target.addChild(o, -1);
    } else {
        OperandContext context = new OperandContext(selected);
        // could deselect here but theoretically unnecessary
        context.parent.setChild(o, context.index);
        if (o.countChildren() > 0 && o.getChild(0) == null) {
            o.addChild(context.self, 0);
        }
        // ok, ugly to resort to instanceof, but ...
        if (context.parent instanceof ExpressionNary) {
            ((ExpressionNary) context.parent).ungroup(context.index);
            myWorkspace.validate();
            selectAfterInsert(((OperandVectorView) context.parentview).getOperandView(context.index));
        }
    }
}

// ungroup at the selection
public void ungroup()
{
    if (selected == null) return;
    OperandContext context = new OperandContext(selected);
    if (context.parent instanceof ExpressionNary) return;
    context.parentview.ungroup(context.index);
    deselect();
    myWorkspace.validate();
    myWorkspace.repaint();
}

public void group()
{
    if (selected == null || selected2 == null) return;
    OperandContext context = new OperandContext(selected);
    OperandContext context2 = new OperandContext(selected2);
    if (context.parent instanceof ExpressionNary) return;
    if (context.parent == context2.parent) return;
    if (context.index >= context2.index) return;
    // ok, finally :
    ((ExpressionNary) context.parent).group(context.index, context2.index - context.index + 1);
    deselect();
    deselect2();
    myWorkspace.validate();
    myWorkspace.repaint();
}

// delete the operand at the selection
public void delete()
{
    if (selected != null) {
        OperandContext context = new OperandContext(selected);
        if (context.self == null) {
            context.parent.delChild(context.index);
        } else if (context.parent instanceof ExpressionNary &&

public void split(boolean s)
{
    if (selected == null) return;
    OperandContext context = new OperandContext(selected);
    if (context.self == null) return;
    context.self.setSplit(s);
    context.selfView.invalidate();
    myWorkspace.validate();
    myWorkspace.repaint();
}

public void copy()
{
    if (selected == null || selected2 == null) return;
    OperandContext src, dst;
    src = new OperandContext(selected);
    dst = new OperandContext(selected);
    deselect3();
    deselect2();
    if (src.self == null) {
        dst.parent.setChild(null, dst.index);
    } else {
        dst.parent.setChild((Operand) src.self.clone(), dst.index);
    }
    validate();
}

public void swap()
{
    if (selected == null || selected2 == null) return;
    if (selected == selected2) return;
    OperandContext s, t;
    s = new OperandContext(selected);
    t = new OperandContext(selected);
    if (s.self == t.self) return;
    Operand s_ = t.self, t_ = s.self;
    s.parent.removeChild(s_, s.index);
    t.parent.removeChild(t_, t.index);
    deselect3();
    deselect2();
    validate();
}

private class ConstantDialog extends Dialog {
    boolean cancelled = false;
    TextField field = new TextField(5);
    Button cancel = new Button("Cancel");
    ConstantDialog()
    {
        super(Editor.this, "Enter Constant", true);
        setLayout(new FlowLayout());
        add(field);
        add(cancel);
        field.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                cancelled = false;
                ConstantDialog.this.dispose();
            }
        });
        cancel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                cancelled = true;
                ConstantDialog.this.dispose();
            }
        });
    }

    public void constant()
    {
        ConstantDialog constdialog = new ConstantDialog();
        constdialog.pack();
        Point p = getLocation();
        constdialog.setLocation(p.x + 10, p.y + 10);
        constdialog.show();
        if (!constdialog.cancelled) {
            String c = constdialog.field.getText();
            Identifier n = Identifier.lookup(c);
            if (n == null) n = new Identifier(c);
            insert(n);
        }
    }

    // c is recently inserted
    private void selectAfterInsert(Component nov)
    {
        // look for root of tree
        Stack path = new Stack();
        Component r = nov;
        while (r.getParent() != null && r.getParent() instanceof OperandView) {
            path.push(r);
            r = r.getParent();
        }
        // post-condition: r is root, path is path of view objects leading
        // from r to nov, not including r (even if r == nov)
        // select first null child or nov itself
        Component firstnull = findFirstNull(r, path);
        if (firstnull != null)
            select(firstnull);
        else
            select(nov);
    }

    // destroys the path along the way
    private Component findFirstNull(Component root, Stack path)
    {
        if (path.empty() && root instanceof NullView) return root;
OperandVectorView r = (OperandVectorView) root;
OperandVector op = r.getOperandVector();
int start = path.empty() ? 0 : r.operandIndexOf((Component) path.pop());
for (int i = start; i < op.countChildren(); i++) {
    Component c = findFirstNull(r.getOperandView(i), path);
    if (c != null) return c;
} return null;

public void deselect() {
    if (selected != null) {
        selected.setForeground(null);
        selected.setBackground(null);
        selected.repaint();
        selected = null;
    }
    myWorkspace.requestFocus();
}

public void select(Component c) {
    if (selected != null) {
        selected.setForeground(null);
        selected.setBackground(null);
        selected.repaint();
        selected = null;
    }
    c.setForeground(Color.white);
    c.setBackground(Color.black);
    c.repaint();
    selected = c;
    c.requestFocus();
}

public void deselect2() {
    if (selected2 != null) {
        selected2.setForeground(null);
        selected2.setBackground(null);
        selected2.repaint();
        selected2 = null;
    }
    c.setForeground(Color.white);
    c.setBackground(Color.black);
    c.repaint();
    selected2 = c;
    c.requestFocus();
}

public void keyReleased(KeyEvent e) {
    int key = e.getKeyCode();
    OperandContext context = new OperandContext(e.getComponent());
    switch (key) {
    /*
    * cursor keys
    */
        case KeyEvent.VK_LEFT:
            if (context.index == 0) break;
            select(((OperandVectorView) context.parentview).getOperandView(context.index - 1));
            break;
        case KeyEvent.VK_RIGHT:
            if (context.index == context.parent.countChildren() - 1) break;
            select(((OperandVectorView) context.parentview).getOperandView(context.index + 1));
            break;
        case KeyEvent.VK_UP:
            if (!context.parent instanceof Operand) break;
            select(context.parentview);
break;
case KeyEvent.VK_DOWN:
    if (context.self == null || context.self.countChildren() == 0) break;
    select(((OperandView) context.selfview).getOperandView(0));
    break;
    /*
     * editing keys
     */
case KeyEvent.VK_DELETE:
    delete();
    break;
}
/**
 * Returns the Workspace instance owned by this Editor instance.
 */
public Workspace getWorkspace()
{
    return myWorkspace;
}
/**
 * Returns the most recently focused/activated Editor instance.
 */
public static Editor recent()
{
    return recentFocus;
}
/**
 * Main method: sets up and opens the minimal windows.
 */
public static void main(String[] args)
{
    /*
     * set up the stencil
     */
    Palette stencil = new Stencil();
    /*
     * set up the variables palette
     */
    Palette variables = new Variables();
    /*
     * Prover
     */
    ProverFrame pf = ProverFrame.instance();
    /*
     * open all windows
     */
    stencil.setSize(160, 300);
    stencil.show();
    variables.setSize(160, 300);
    variables.show();
    // pf.setSize(640, 480);
    // pf.show();
    newEditor();
}
package scphEditor;

/**
 * An Embedding object encapsulates presentation styles, in AWT terms,
 * of an operator and its operands. They are needed by an
 * ExpressionView object that is responsible for displaying the
 * operator and the operands. They include:
 */

<ol>
<li>layout manager</li>
<li>component indexes of operands</li>
<li>component indexes of operator symbols, and the operator symbols
 themselves</li>
</ol>

E.g., consider the if-then-else operator.

<ul>
<li>layout manager: nLULayout2</li>
<li>the three operands go to components 1, 2, and 3;</li>
<li>the operator symbols go to components 0, 2, and 4; they are
 "if", "then", and "else".</li>
</ul>

This class is only an abstract class. It specifies the layout
creation, indexing, and symbol services, but does not implement
them. The concrete class EmbeddingFixed implements them for
operators with fixed numbers of operands, and the concrete class
EmbeddingNary implements them for n-ary AC operators.

@see EmbeddingFixed
@see EmbeddingNary

public abstract class Embedding
{
/**
 * Creates an Embedding object.
 */
public Embedding()
{
}

/**
 * Returns a new instance of the preferred layout manager.
 */
public abstract ExpressionLayoutManager getNewLayout();

/**
 * Maps operand index to component index. If the operand index is
 * invalid (less than zero, or beyond arity), this method may choose
 * to throw an exception.
 */

@param operandIndex the operand index.
@return the component index.
@exception java.lang.ArrayIndexOutOfBoundsException when it is blatantly
 obvious to me that the operand index is invalid.
*/
public abstract int operandIndex(int componentIndex);

/**
 * Gives the operator symbol at the component index, or null if
 * there should be an operand at the component index. If the
 * component index is invalid (less than zero, or beyond component
 * count), this method may choose to throw an exception.
 */

@param componentIndex the component index.
@return the operator symbol at that index, or null.
@exception java.lang.ArrayIndexOutOfBoundsException when it is blatantly
 obvious to me that the component index is invalid.
*/
public abstract String symbol(int componentIndex);

/**
 * Gives the component count of this embedding. In other words, how
 * many components an ExpressionView should prepare to contain, in
 * order to accommodate the operands and the operator symbols. Since
 * this may (or may not) depend on the actual number of operands,
 * this method requires that information. Implementors are allowed
 * to garbage-in-garbage-out in case the operand count is invalid,
 * e.g., negative.
 */

@param operandCount the actual, present number of operands.
@return the number of components needed.
*/
public abstract int componentCount(int operandCount);
}
package scphEditor;

import java.util.Hashtable;

/**
 * This singleton factory class maintains a collection of Embedding objects as flyweights, and maps Operator objects to Embedding objects, i.e., you give me an Operator and I give you the corresponding Embedding.
 */
public class EmbeddingFactory
{
    /**
     * Maps operators to embeddings.
     */
    private Hashtable embeddings;

    private static EmbeddingFactory singleton = new EmbeddingFactory();

    /**
     * Constructor.
     */
    private EmbeddingFactory()
    {
        embeddings = new Hashtable();
    }

    /**
     * Gives the single, global instance of this class.
     */
    public static EmbeddingFactory instance()
    {
        return singleton;
    }

    /**
     * Gives the embedding that corresponds to a operator, or null if it does not exist.
     *
     * @param op the operator.
     * @return the embedding or null.
     */
    public Embedding get(Operator op)
    {
        return (Embedding) embeddings.get(op);
    }

    /**
     * Enters or replaces an operator-embedding correspondence into me.
     *
     * @param op the operator
     * @param e the corresponding embedding
     */
    public void set(Operator op, Embedding e)
    {
        embeddings.put(op, e);
    }
}
package scphEditor;

import java.awt.LayoutManager;

/**
 * An EmbeddingFixed object encapsulates presentation styles, in AWT terms, of an operator and its operands. The operator is assumed to have a fixed number of operands, e.g., if-then-else, unary operators.
 */
public class EmbeddingFixed extends Embedding {
  private String[] template;
  private int[] map_c2o; // index map: component to operand
  private int[] map_o2c; // index map: operand to component

  /* Creates an EmbeddingFixed object. Requires a template for the display of the operator. The template specifies where and what symbols are displayed, and also where operands occur. Examples:
   * <ul>
   * <li>unary prefix ++ operator: <code>("++", null)</code>
   * <li>unary postfix ++ operator: <code>({null, "++")</code>
   * <li>tertiary if-then-else operator: <code>("if", null, "then", null, "else", null)</code>
   * </ul>
   * In general, the template is an array of strings: null where operands should occur, non-null strings where operator symbols should occur. Operands are sequentially placed, e.g., if operand#0 then operand#1 else operand#2.
   * @param template The template that specifies operator symbols and operand positions.
   */
  public EmbeddingFixed(String[] template) {
    template = template;
    buildMap();
  }

  /**
   * Takes the template field, builds map_o2c and map_c2o.
   */
  private void buildMap() {
    int arity = 0;
    int[] o2c = new int[template.length]; // will copy to map_o2c
    map_o2c = new int[template.length];
    for (int i = 0; i < template.length; i++) {
      if (template[i] != null) {
        map_c2o[i] = -1;
      } else {
        map_c2o[i] = arity;
        o2c[arity] = i;
        arity++;
      }
    }
    map_o2c = new int[arity];
    System.arraycopy(o2c, 0, map_o2c, 0, arity);

    public String symbol(int componentIndex) { return template[componentIndex]; }
    public int componentIndex(int operandIndex) { return map_o2c[operandIndex]; }
    public int operandIndex(int componentIndex) {
      if (map_c2o[componentIndex] == -1) throw new ArrayIndexOutOfBoundsException();
      return map_c2o[componentIndex];
    }
    public int componentCount(int operandCount) { return template.length; }
    */
  }
}

/**
 * Gives a new instance of MLMLayout2, the preferred layout manager here.
 */
public ExpressionLayoutManager getNewLayout() {
  return new MLMLayout2();
}

package scphEditor;

import java.awt.LayoutManager;

/**
 * An EmbeddingNary object encapsulates presentation styles, in AWT terms, of an operator and its operands. The operator is assumed to be n-ary and infix - the indexing methods will compute indexes accordingly.
 */

public class EmbeddingNary extends Embedding {
    private String symbol;

    /**
     * Creates an EmbeddingNary object. Requires a string containing the operator symbol.
     * @param symbolOf the operator symbol.
     */
    public EmbeddingNary(String symbolOf) {
        symbol = symbolOf;
    }

    /**
     * Gives a new instance of MLMLayout2, the preferred layout manager here.
     * @see MLMLayout2
     */
    public ExpressionLayoutManager getNewLayout() {
        return new MLMLayout2();
    }

    public String symbol(int componentIndex) {
        if (componentIndex < 0)
            throw new ArrayIndexOutOfBoundsException();
        // symbol occurs at odd indexes
        return (componentIndex & 1) == 1 ? symbol : null;
    }

    public int componentIndex(int operandIndex) {
        if (operandIndex < 0)
            throw new ArrayIndexOutOfBoundsException();
        // operands go to even indexes
        return 2 * operandIndex;
    }

    public int operandIndex(int componentIndex) {
        if (componentIndex < 0 || (componentIndex & 1) == 1)
            throw new ArrayIndexOutOfBoundsException();
        return componentIndex >> 1;
    }

    public int componentCount(int operandCount) {
        return operandCount == 0 ? 0 : operandCount * 2 - 1;
    }
}
package asphEditor;

import java.util.Vector;
import java.util.Enumeration;

/**
 * This class represents compound expressions. A compound expression
 * has an operator and a list of subexpressions, i.e., children, each
 * child being an Operand object. E.g., -x is modelled as an Expression
 * object, with the unary - operator and the child x.
 *<p>The arity of an expression is the same as that of its operator;
 * in case the operator is n-ary, though, the expression becomes
 * binary. Use the subclass ExpressionNary instead if you want true
 * n-ary services. For example, 1 + 2 * 3 is better modelled by an
 * ExpressionNary object.
 */
public class Expression extends Operand {

/**
 * [Rooting] Operator in this expression.
 */
protected Operator op;

/**
 * List of child operands.
 */
protected Vector children;

/**
 * True iff this operand is parenthesized.
 */
protected boolean paren;

/**
 * True iff the view should split this expression across lines.
 */
protected boolean splitted;

/**
 * Constructs an Expression object with the given operator. The
 * given operator will become the operator of this compound
 * expression permanently. As many children as the operator arity
 * specifies are set up, and initialized to null. In the case of an
 * n-ary operator, 2 children are set up.
 *
 * @param opOf the operator object for this Expression object.
 */
public Expression(Operator opOf) {
    super();

    op = opOf;
    children = new Vector();
    splitted = false;
    paren = false;

    // now we need empty children
    if (op.arity() == Operator.NARY) {
        // make 2 children
        children.ensureCapacity(2);
    }
}

/*  */
public Object clone() {
    // need to copy deeply
    Expression c = (Expression) super.clone();
    c.children = (Vector) c.children.clone();
    for (int i = 0; i < c.children.size(); i++) {
        Operand e = (Operand) c.children.elementAt(i);
        if (e != null) e = (Operand) e.clone();
        c.children.setAtElement(e, i);
    }
    return c;
}

/**
 * Gives the operator of this expression.
 */
public Operator getOperator() {
    return op;
}

/**
 * Adopts the given new child as my child at the given index,
 * replacing the previous child at the same index.
 *
 * @param newValue the new child
 * @param atIndex the index to where newValue goes
 * @throws java.lang.ArrayIndexOutOfBoundsException if atIndex
 * is an invalid index, i.e., atIndex violates my arity.
 */
public void setChild(Operand newValue, int atIndex) {
    try {
        children.setAtElement(newValue, atIndex);
    } catch (ArrayIndexOutOfBoundsException e) {
        e.fillInStackTrace();
        throw e;
    }

    if (newValue != null) newValue.ensureParen(this);
    setChanged();

    // now we need empty children
    if (op.arity() == Operator.NARY) {
        // make 2 children
        children.ensureCapacity(2);
    }
}

public void setChild(Operand newValue) {
    try {
        children.addElement(newValue);
    } catch (ArrayIndexOutOfBoundsException e) {
        e.fillInStackTrace();
        throw e;
    }

    if (newValue != null) newValue.ensureParen(this);
    setChanged();

    // now we need empty children
    if (op.arity() == Operator.NARY) {
        // make 2 children
        children.ensureCapacity(2);
    }
}
}

Expression.java

// children.addElement(null);
children.addElement(null);
}

}
Gives my child at the given index.

```java
public Operand getChild(int atIndex) {
    try {
        return (Operand) children.elementAt(atIndex);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        e.printStackTrace(); // pretend we threw it
    }
}
```

This implementation does not do anything.

```java
public void addChild(Operand child, int atIndex) {
    ...
    // This implementation returns null and does not do anything else.
}
```

Enumerates all of my children, in ascending order of indexes.

```java
public Enumeration getChildren() {
    return children.elements();
}
```

Counts children.

```java
public int countChildren() {
    return children.size();
}
```

Add/Remove Parentheses.

```java
protected void setParent()
{
    paren = true;
    setChanged();
    notifyObservers();
}
protected void setParent()
{
    paren = false;
}
```

Expression.java

```java
setChanged();
notifyObservers();
}
public boolean getIsParent() {
    return isParent;
}
public boolean getIsSplit() {
    return isSplit;
}
public void setIsParent(boolean isParent) {
    isParent = split;
    setChanged();
    notifyObservers();
}
/**
 * Calls v.visitExpression(this).
 */
public void acceptVisitor(OperandVisitor v) {
    v.visitExpression(this);
}
/**
 * A Replacement object represents a substituational change made to
 * an Expression object. Essentially, a change involves first
 * deleting some children, then inserting some new children.
 */
public class Replacement {
    ...
    // The range of replacement begins at this index.
    public int at;
    // The replacement involves deleting these many children. Note
    // the semantics: delete, then insert.
    public int deleteCount;
    // The replacement involves inserting these many children. Note
    // the semantics: delete, then insert.
    public int insertCount;
    // Creates a Replacement object.
    // @param a becomes at.
    // @param dn becomes deleteCount.
    // @param iN becomes insertCount.
    public Replacement(int a, int dN, int iN) {
        at = a;
        deleteCount = dN;
        insertCount = iN;
    }
```
package scphEditor;

import java.awt.LayoutManager2;

public interface ExpressionLayoutManager extends LayoutManager2
{
    public void setSplit(boolean s);
}
package acphEditor;
import java.util.Vector;

/**
 * This class represents n-ary compound expressions.
 *
 * @param opOf the operator object for this Expression object.
 * @param atIndex the given index.
 * @return 0 if no ungrouping performed (e.g., operators are different or not associative); otherwise the number of new children in the expansion.
 * @see ExpressionNary#addChild
 * @see ExpressionNary#ungroup
 */
protected int ungroup_impl(int atIndex)
{
  Operand child;
  try {
    child = (Operand) children.elementAt(atIndex);
    catch (ArrayIndexOutOfBoundsException e) {
      e.fillInStackTrace();
      throw e; // pretend we threw it
    }
    children.removeElementAt(atIndex);
    setChanged();
    Replacement change = new Replacement(atIndex, 0, 1);
    notifyObservers(change);
    return oldChild;
  }
  catch (Exception e) {
    e.fillInStackTrace();
    throw e; // pretend we threw it
  }
  if (op != child.getOperator() || op.associative() != Operator.ASSOC) {
    return 0;
  }
  else {
    oldchild = (Operand) children.elementAt(atIndex);
    catch (ArrayIndexOutOfBoundsException e) {
      e.fillInStackTrace();
      throw e; // pretend we threw it
    }
    children.removeElementAt(atIndex);
    setChanged();
    Replacement change = new Replacement(atIndex, 1, n);
    notifyObservers(new Replacement(atIndex, 1, n));
  }
}

public void group(int atIndex, int n) {
  if (op.associative() == Operator.ASSOC) return;
  if (n < 2) return;
  if (n == countChildren()) return;
  ExpressionNary e = new ExpressionNary(op);
  e.children.removeAllElements();
  for (int j = 0; j < n; j++) {
    Operand o = (Operand) children.elementAt(atIndex);
    children.removeElementAt(atIndex);
    e.addChild(o, -1);
  }
  children.insertElementAt(e, atIndex);
  e.ensureParen(this);
  setChanged();
  notifyObservers(new Replacement(atIndex, n, 1));
}

public void ungroup(int atIndex) {
  if (children.elementAt(atIndex) == null) return;
  int n = ungroup_impl(atIndex);
  if (n > 0) {
    setChanged();
    notifyObservers(new Replacement(atIndex, 1, n));
  }
}

public void addChild(Operand newChild, int atIndex) {
  super.addChild(newChild, atIndex);
  for (int j = 0; j < atIndex; j++) {
    e.addChild(o, -1);
  }
  children.insertElementAt(e, atIndex);
  e.ensureParen(this);
  setChanged();
  notifyObservers(new Replacement(atIndex, 1, n));
}

public void setChild(Operand newChild, int atIndex) {
  super.setChild(newChild, atIndex);
}

public void delChild(int atIndex) {
  Operand oldChild;
  try {
    oldChild = (Operand) children.elementAt(atIndex);
    catch (ArrayIndexOutOfBoundsException e) {
      e.fillInStackTrace();
      throw e; // pretend we threw it
    }
    children.removeElementAt(atIndex);
    setChanged();
    Replacement change = new Replacement(atIndex, 0, 1);
    notifyObservers(change);
    return oldChild;
  }
  catch (Exception e) {
    e.fillInStackTrace();
    throw e; // pretend we threw it
  }
  if (newChild != null) newChild.ensureParen(this);
  setChanged();
  notifyObservers(new Replacement(atIndex, 0, 1));
}

public class ExpressionNary extends Expression {
  // ...
children.removeElementAt(atIndex);
int count = child.countChildren();
for (int i = 0; i < count; i++) {
    Operand opnd = child.getChild(i);
    children.insertElementAt(opnd, i + atIndex);
    if (opnd != null) opnd.ensureParen(this);
}
return count;
}

/**
 * Calls v.visitExpressionNary(this).
 */
public void acceptVisitor(OperandVisitor v) {
    v.visitExpressionNary(this);
}
package scphEditor;

import java.util.Observable;

/**
 * This AWT component is the view class for ExpressionNary objects.
 * It displays the expression recursively. It automatically updates
 * itself whenever the expression changes (child manipulations), by
 * way of the Observable-Observer mechanism.
 *
 * @see ExpressionNary
 */
public class ExpressionNaryView extends ExpressionView
    // implements Observer, OperandView
{
    protected ExpressionNary myClient;

    /**
     * Constructor.
     * @param exOf the ExpressionNary object to view.
     * @param c the control object (event listener) for me.
     */
    public ExpressionNaryView(ExpressionNary exOf, OperandControl c)
    {
        super(exOf, c);
        myClient = exOf;
    }

    public void update(Observable o, Object arg)
    {
        Expression.Replacement change = (Expression.Replacement) arg;
        if (change != null) {
            int atComponentIndex = myClient.componentIndex(change.at) + (paren ? 1 : 0);

            if (change.deleteCount > 0) {
                // delete so many children and the operators inside them:
                // deleteCount * 2 - 1 of them
                int c = change.deleteCount * 2 - 1;
                for (int i = 0; i < c; i++) remove(atComponentIndex);
                // delete one more operator if not inserting later
                if (change.insertCount == 0) {
                    if (change.at == 0) remove(atComponentIndex);
                    else remove(atComponentIndex - 1);
                }
            }
            if (change.insertCount > 0) {
                OperandViewFactory f = new OperandViewFactory(myControl);
                // insert extra operator if not deleting earlier
                if (change.deleteCount == 0) {
                    if (change.at == 0)
                        add(new AtomView(myEmbedding.symbol(1)), this, atComponentIndex);
                    else
                        add(new AtomView(myEmbedding.symbol(1)), this, atComponentIndex - 1);
                }
                // insert the children and inside operators:
                // insertCount * 2 - 1 of them
                int c = change.insertCount * 2 - 1;
                for (int i = 0, ci = atComponentIndex, ci = change.at;
                    i < c;
                    i++, ci++) {
                    // insert one more operator if not inserting later
                    if (change.insertCount == 0) {
                        if (change.at == 0) remove(ci);
                        else remove(ci - 1);
                    }
                }
            }
        }
        // may need validation here
        repaint();
        parenthesize();
        ((ExpressionLayoutManager) getLayout()).setSplit(myClient.getSplit());
    }
}
package scphEditor;

import java.awt.*;
import java.util.Observer;
import java.util.Observable;
import java.util.Enumeration;

/**
 * This AWT component is the view class for Expression objects. It
 * displays the expression recursively. It automatically updates
 * itself whenever the expression changes (child manipulations), by
 * way of the Observable-Observer mechanism.
 * @see Expression
 */
public class ExpressionView extends Container implements Observer, OperandView
{
    /**
     * The expression viewed by me.
     */
    protected Expression myClient;
    /**
     * The Embedding for the Operator of my expression.
     */
    protected Embedding myEmbedding;
    /**
     * True iff I am showing parentheses.
     */
    protected boolean paren;
    /**
     * Remember my control object (event listener), so I can pass it on
     * to children.
     */
    protected OperandControl myControl;
    /**
     * Constructor.
     * @param exOf the Expression object to view.
     * @param c the control object (event listener) for me.
     */
    public ExpressionView(Expression exOf, OperandControl c)
    {
        super();
        myClient = exOf;
        myEmbedding = EmbeddingFactory.instance().get(myClient.getOperator());
        ExpressionLayoutManager l = myEmbedding.getNewLayout();
        setLayout(l);
        l.setSplit(myClient.getSplit());
        paren = false;
        myControl = c;
        addMouseListener(c);
        addKeyListener(c);
        makeComponents();
    }
    /**
     * Creates my components - views for child operands as well as
     * operator symbols and parentheses. This implementation works
     * correctly for n-ary operators and expressions, by calling the
     * most abstract methods possible.
     */
    protected void makeComponents()
    {
        OperandViewFactory f = new OperandViewFactory(myControl);
        int arity = myClient.countChildren();
        int cc = myEmbedding.componentCount(arity);
        Enumeration e = myClient.getChildren();
        for (int i = 0; i < cc; i++)
        {
            if (myEmbedding.symbol(i) == null)
            {
                add(new AtomView( myEmbedding.symbol(i) ), this);
            }
            else {
                Operand p = (Operand) e.nextElement();
                add(f.make(p));
            }
        }
        parenthesize();
    }
    /**
     * Helper function. Modifies my parentheses according to my
     * Expression object.
     */
    protected void parenthesize()
    {
        if (paren && !myClient.getParen())
        {
            remove(-1);
            remove(0);
            paren = false;
        }
        else if (!paren && !myClient.getParen())
        {
            add(new AtomView("*", 0));
            add(new AtomView("*", -1));
            paren = true;
        }
    }
    public void addNotify()
    {
        myClient.addObserver(this);
        super.addNotify();
    }
    public void removeNotify()
    {
        myClient.deleteObserver(this);
        super.removeNotify();
    }
    /**
     * Returns the Embedding for the Operator of my Expression.
     */
    public Embedding getEmbedding()
    {
        return myEmbedding;
    }
    public Operand getOperand()
    {
        return myClient;
    }
    public OperandVector getOperandVector()
    {
        return myClient;
    }
}
public int operandIndex(Component child)
{
    return myEmbedding.operandIndex(viewIndex(child) - (paren ? 1 : 0));
}

public Component getOperandView(int opindex)
{
    return getComponent(myEmbedding.componentIndex(opindex)
                        + (paren ? 1 : 0));
}

/**
 * Helper function. Replaces the child view object at the given index.
 * @param opindex the index.
 * @param view the new view object.
 * @return the replaced, old view object.
 */
protected Component setOperandView(int opindex, Component view)
{
    int viewindex = myEmbedding.componentIndex(opindex) + (paren ? 1 : 0);
    Component old = getComponent(viewindex);
    remove(viewindex);
    add(view, viewindex);
    return old;
}

/**
 * Given a child view object, looks up the Component index it
 * belongs to in me.
 * @param child the child view object.
 * @return its Component index in me.
 */
public int viewIndex(Component child)
{
    // linear search, yikes
    int n = getComponentCount();
    for (int i = 0; i < n; i++)
    {
        if (getComponent(i) == child) return i;
    }
    return -1;    // or throw exception?
}

public void update(Observable o, Object arg)
{
    Expression.Replacement change = (Expression.Replacement) arg;
    if (change != null) {
        OperandViewFactory f = new OperandViewFactory(myControl);
        // assume change.deleteCount == 1 and change.insertCount == 1
        setOperandView(change.at, f.make(myClient.getChild(change.at)));
        parenthesize((ExpressionLayoutManager) getLayout()).setSplit(myClient.getSplit());
        // may need validation here
        // repaint();
    }
}

public void paint(Graphics g)
{
    Dimension size = getSize();
    g.setColor(getBackground());
    g.fillRect(0, 0, size.width, size.height);
    super.paint(g);
}
package scphEditor;

import java.util.Hashtable;

/**
 * This class models terminal expressions, such as variables and constants
 * like x and 10. An Identifier object has a name, and this is precisely
 * the x or 10 part. It could also have an alternative name to be used
 * with the theorem prover.
 */

/**
 * It implements Operator, because variables and constants are degenerate
 * cases of functions.
 */

/**
 * It is a subclass of Operand as well. It is an operand that has no
 * child and an operator - itself.
 */

/**
 * Identifier objects are designed to be flyweights.
 * You can use the lookup method to look up an existing Identifier object
 * by its name.
 */

public class Identifier extends Operand implements Operator {

  /**
   * dictionary from name to identifier object
   */
  private static Hashtable dictionary = new Hashtable();

  private String name;  // e.g., x, 10
  private String pname;  // name known to prover

  /**
   * Looks up an Identifier object from the dictionary.
   * @param name The [prover] name of the desired identifier.
   * @return The corresponding identifier or null.
   */
  public static Identifier lookup(String name) {
    return (Identifier) dictionary.get(name);
  }

  /**
   * Creates an identifier with the given name.
   * @param name The name.
   */
  public Identifier(String nameOf) {
    this(nameOf, nameOf);
  }

  /**
   * Creates an identifier with the given name and prover name.
   * @param name The name.
   * @param pname The prover name.
   */
  public Identifier(String nameOf, String pnameOf) {
    name = nameOf;
    pname = pnameOf;
    dictionary.put(pname, this);
  }

  /**
   * Changes the name and prover name of this identifier.
   * @param nameOf The new name.
   */
  public void setName(String nameOf)
  {
    dictionary.remove(pname);
    name = nameOf;
    pname = nameOf;
    dictionary.put(pname, this);
    setChanged();
    notifyObservers();
  }

  /**
   * Returns the name of this identifier.
   */
  public String getName() {
    return name;
  }

  /**
   * Returns the prover name of this identifier.
   */
  public String getProverName() {
    return pname;
  }

  /**
   * Returns the precedence of this identifier (as an operator). Always 0.
   */
  public int precedence() {
    return 0;
  }

  /**
   * Returns the associativity of this identifier (as an operator).
   * Always Operator.NONASSOC.
   */
  public int associative() {
    return Operator.NONASSOC;
  }

  /**
   * Returns the commutativity of this identifier (as an operator).
   * Always false.
   */
  public boolean commutative() {
    return false;
  }

  /**
   * Returns the arity of this identifier (as an operator). Always 0.
   */
  public int arity() {
    return 0;
  }

  /**
   * 'Clones' this identifier. Since identifiers are flyweights,
   * there is no cloning; the object itself is returned.
   */
```java
* @return this
*/
public Object clone()
{
    return this;
}

/**
 * Gives the operator in this identifier operand, which is the
 * identifier itself.
 *
 * @return this
 */
public Operator getOperator()
{
    return this;
}

/**
 * Calls v.visitIdentifier(this).
 */
public void acceptVisitor(OperandVisitor v)
{
    v.visitIdentifier(this);
}
```
package scphEditor;
import java.awt.*;
import java.util.Observer;
import java.util.Observable;

/**
 * This AWT component is the view class for Identifier objects. It
 * displays the identifier name. It automatically updates itself
 * whenever the identifier name changes, by way of the
 * Observable-Observer mechanism.
 * @see Identifier
 */
public class IdentifierView extends AtomView implements Observer, OperandView {
    /**
     * My Identifier object.
     */
    private Identifier myClient;

    /**
     * Creates an IdentifierView object that views the given Identifier.
     * @param idOf the Identifier object to view.
     * @param c the control object (event listener) for me.
     */
    public IdentifierView(Identifier idOf, OperandControl c) {
        super(idOf.getName());
        // should setFont() here
        addMouseListener(c);
        addKeyListener(c);
        myClient = idOf;
    }

    public void addNotify() {
        myClient.addObserver(this);
        super.addNotify();
    }

    public void removeNotify() {
        myClient.deleteObserver(this);
        super.removeNotify();
    }

    public Operand getOperand() {
        return myClient;
    }

    public OperandVector getOperandVector() {
        return myClient;
    }

    public int operandIndex(Component child) {
        return -1;    // or throw exception?
    }

    public Component getOperandView(int opindex) {
        return null;    // or throw exception?
    }

    /**
     * Called by my Identifier object automatically when it changes its
     * name. This method will then:
     * @param o Check that the parameter o is indeed my Identifier object.
     * @param arg if not, skip.
     * @param arg Obtain the new identifier name and display it.
     */
    public void update(Observable o, Object arg) {
        if (o == myClient) {
            setText(myClient.getName());
        }
    }
}
package scphEditor;

import java.awt.*;
import java.util.Hashtable;

/**
 * This layout manager provides depth-proportional spacing to
 * components. If you add() a component with a constraint parameter
 * (any non-null string or object will do), the component will obtain
 * depth-proportional spacing around its left and right. Otherwise
 * the component does not get the extra space.
 *<p>
 * All of minimumLayoutSize(), preferredLayoutSize(), and
 * maximumLayoutSize() return the same value.
 */
public class MLMLayout2 implements ExpressionLayoutManager
{
    /**
     * Padding space = GLUE * depth.
     */
    private int GLUE = 1;
    /**
     * In splitting mode or not.
     */
    private boolean split;
    /**
     * Remembers which components are operators.
     */
    private Hashtable operators;
    /**
     * Default constructor.
     */
    public MLMLayout2()
    {
        split = false;
        operators = new Hashtable();
    }

    public void addLayoutComponent(String s, Component com)
    {
        if (s != null) operators.put(com, com);
    }

    public void addLayoutComponent(Component com, Object con)
    {
        if (con != null) operators.put(com, com);
    }

    public void removeLayoutComponent(Component com)
    {
        operators.remove(com);
    }

    public Dimension minimumLayoutSize(Container con)
    {
        return preferredLayoutSize(con);
    }

    public Dimension preferredLayoutSize(Container con)
    {
        if (split) return preferredLayoutSizeSplit(con);
        else return preferredLayoutSizeFlat(con);
    }

    private Dimension preferredLayoutSizeFlat(Container con)
    {
        int space = findDepth(con) * GLUE;
        Dimension ps = new Dimension(0, 0);
        Component[] coms = con.getComponents();
        for (int i = 0; i < coms.length; i++)
        {
            if (operators.containsKey(coms[i]))
                ps.width += space;
            Dimension s = coms[i].getPreferredSize();
            ps.width += s.width;
            if (operators.containsKey(coms[i]))
                ps.width += space;
            if (ps.height < s.height) ps.height = s.height;
        }
        // if (ps.width > 0) ps.width -= space;
        return ps;
    }

    private Dimension preferredLayoutSizeSplit(Container con)
    {
        Dimension[] sizes = sizesSplit(con);
        return new Dimension(sizes[0].width + sizes[1].width,
                             Math.max(sizes[0].height, sizes[1].height));
    }

    private Dimension[] sizesSplit(Container con)
    {
        Dimension optrsize = new Dimension(0, 0);
        Dimension opndsize = new Dimension(0, 0);
        Component[] coms = con.getComponents();
        for (int i = 0; i < coms.length; i++)
        {
            Component com = coms[i];
            Dimension s = com.getPreferredSize();
            if (operators.containsKey(com))
                optrsize.height += s.height;
            else
                opndsize.height += s.height;
            optrsize.width = Math.max(optrsize.width, s.width);
        }
        return new Dimension[] { optrsize, opndsize };
    }

    public Dimension maximumLayoutSize(Container con)
    {
        return preferredLayoutSize(con);
    }

    public void layoutContainer(Container con)
    {
        if (split) layoutContainerSplit(con);
        else layoutContainerFlat(con);
    }

    private void layoutContainerFlat(Container con)
    {
        int space = findDepth(con) * GLUE;
        int cx = 0, cy = 0;
        Component[] coms = con.getComponents();
        for (int i = 0; i < coms.length; i++)
            layoutComponentFlat(coms[i], cx, cy, space);
    }
for (int i = 0; i < coms.length; i++) {
    if (operators.containsKey(coms[i]))
        cx += space;
    Dimension s = coms[i].getPreferredSize();
    coms[i].setSize(s);
    coms[i].setLocation(cx, cy);
    cx += s.width;
    if (operators.containsKey(coms[i]))
        cx += space;
}

private void layoutContainerSplit(Container con) {
    Dimension[] sizes = sizeSplit(con);
    Component[] coms = con.getComponents();
    int cx = 0, cy = 0;
    for (int i = 0; i < coms.length; i++) {
        Component com = coms[i];
        Dimension s = com.getPreferredSize();
        com.setSize(s);
        if (operators.containsKey(com)) {
            com.setLocation(cx, cy);
        } else {
            com.setLocation(cx + sizes[0].width, cy);
            cy += s.height;
        }
    }
}

public float getLayoutAlignmentX(Container con) {
    return 0.0f;
}

public float getLayoutAlignmentY(Container con) {
    return 0.0f;
}

public void invalidateLayout(Container con) {
}

/**
 * Recursively computes the depth of the container.
 * @param con the container.
 * @return depth of the container.
 */
private int findDepth(Container con) {
    Component[] coms = con.getComponents();
    int max = -1;
    for (int i = 0; i < coms.length; i++) {
        if (coms[i] instanceof Container) {
            int d = findDepth((Container)coms[i]);
            if (max < d) max = d;
        }
    }
    return max + 1;
}

public void setSplit(boolean s) {
    split = s;
}
package scphEditor;

import java.awt.Component;

/**
 * This is the view class/object for a null Operand.
 */
public class NullView extends AtomView implements OperandView
{

    /**
     * Creates NullView object.
     * @param c the control object (event listener) for me.
     */
    public NullView(OperandControl c)
    {
        super("_");
        addMouseListener(c);
        addKeyListener(c);
    }

    public Operand getOperand()
    {
        return null;
    }

    public OperandVector getOperandVector()
    {
        return null;
    }

    public int operandIndex(Component child)
    {
        return -1; // or throw exception?
    }

    public Component getOperandView(int opindex)
    {
        return null; // or throw exception?
    }
}
package scphEditor;

/**
 * This abstract class models an operand. An operand may be a terminal
 * expression like x or 10; or a compound expression like x+10. This
 * abstract class is the superclass of both, defining common services
 * and default implementations; this is the base class in a composite pattern.
 */

/** The common services include child management. A compound expression
 * has a number of child operands, e.g., x and 10 are the child operands
 * in x+10. Each child is referred to by a zero-based index; thus, in x+10,
 * x is child #0 and 10 is child #1. Child management, then, consists of
 * methods for:
 * <ul>
 * <li>setting (adopting) a new child at an index
 * <li>getting a child at an index
 * <li>enumerating all children
 * </ul>
 * Naturally, the default implementations of these methods are for leaf
 * nodes (terminal expressions), for which setting and getting children
 * does not make sense; therefore, they always throw exceptions.
 */

/** A compound expression class must override this and other behaviours
 * and implement them for real. This is done by the Expression class
 * hierarchy, for example.
 */

/** See Expression
 *
 * public abstract class Operand extends OperandVector implements Cloneable
 *
 *  /**
 *  * Default constructor.
 *  */
 *  public Operand()
 *  { super(); }
 *
 *  /**
 *  * Clones this operand.
 *  * @return A clone of this operand
 *  */
 *  public Object clone()
 *  { return super.clone(); }
 *
 *  /**
 *  * Gives the operator in this subexpression. Yes, this is supposed to
 *  * make sense even for terminals like x and 10. The current plan is
 *  * that a terminal is both an operand and an operator; but there is also
 *  * a chance that a terminal returns some kind of empty operator.
 *  *
 *  * @return the Operator object of this operand/subexpression.
 *  *
 *  */
 *  public abstract Operator getOperator();
 *
 *  /**
 *  * Determines if I need to be parenthesized, assuming I am a child
 *  * of the given operand. I need parentheses if my operator binding
 *  * is looser than my parent's, up to associativity. Unary operators
 *  * are treated as right-associative.
 *  *
 *  */
 *  protected boolean needParen(Operand owner)
 *  { if (owner == null) return false;
 *    int diff = owner.getOperator().precedence() - getOperator().precedence();
 *    if (diff < 0) return true;
 *    if (diff > 0) return false;
 *    int assoc = getOperator().associative();
 *    if (owner.getChild(owner.countChildren() - 1) == this)
 *      return false;
 *    if ((assoc == Operator.RIGHT && owner.getChild(0) == this)
 *      && owner.getOperator().arity() == 1)
 *      return false;
 *    else
 *      return true;
 *  }
 *
 *  /**
 *  * Parenthesizes me, if I need to be. Calls needParen() to
 *  * determine the need and addParen() to do it. This method is
 *  * normally only used by setChild() of compound expression classes.
 *  *
 *  * @param owner my hypothetical or real parent, or null.
 *  * @param needParen
 *  * @param addParen
 *  * @param needParen2
 *  * @param addParen2
 *  * @param needParen3
 *  * @param addParen3
 *  */
 *  protected void ensureParen(Operand owner)
 *  { if (needParen(owner)) addParen(); }
 *
 *  /**
 *  * Parenthesizes me.
 *  *
 *  */
 *  public void addParen()
 *  { }
 *
 *  /**
 *  * De-parenthesizes me. This method should only be called by
 *  * internal code such as removeParen().
 *  *
 *  */
 *  public void removeParen()
 *  { }
 *
 *  /**
 *  * See Operand#removeParen
 *  */
 *  protected void deParen()
 *  { }
 *
 *  /**
 *  * De-parenthesizes me, if I do not need parentheses. Calls
 *  * needParen() to determine the need and deParen() to do it.
 *  *
 *  */
 *  public void deParen()
 *  { }
 *
 *  /**
 *  * @param owner my hypothetical or real parent, or null.
 *  * @param needParen
 *  * @param needParen2
 *  * @param needParen3
 *  */
 *  protected void maybeParen(Operand owner)
 *  { }
 */

Operand.java
public void removeParen(Operand owner) {
    if (!needParen(owner)) deParen();
}

/**
 * Tells if I am already parenthesized.
 *<p>This default implementation is for leaf nodes. Compound
 * expressions should override this properly.
 *<p>@return true iff I am parenthesized. (False for leaf nodes.)
 */
public boolean getParen()
{
    return false;
}

/**
 * Gets my split status. In other words, whether or not I should be
 * splittted across the line when displayed by a view, like, <tt>b
 * = a</tt> becomes:
 *<pre>
 * b
 * = a
 * </pre>
 *<p>Of course, this is just a hint to the view.
 *<p><p>This default implementation is for leaf nodes, which never splits.
 * Compound expressions should override this properly.
 *<p>@return true iff I should be displayed as splittted. (False for
 * leaf nodes.)
 */
public boolean getSplit()
{
    return false;
}

/**
 * Sets my split status.
 *<p>This default implementation is for leaf nodes, which never splits.
 * Compound expressions should override this properly.
 *<p>@param split true if you want me to split, false otherwise.
 */
public void setSplit(boolean split)
{

/** Interface for OperandVisitor objects to visit this operand
 * concretely. (Visitor pattern.) Absolutely no traversal in any
 * implementation of this method, i.e., this method by itself does
 * not recurse into children of this operand.
 *
 */
public abstract void acceptVisitor(OperandVisitor v);
}
package scphEditor;

import java.awt.Component;
import java.awt.Container;

/**
 * This class is a simple record containing the context of an operand,
 * i.e., itself, its position index in its parent, and the parent.
 */
public class OperandContext {
    public Operand self;
    public OperandVector parent;
    public Component selfview;
    public Container parentview;
    public int index;

    public OperandContext (Component oview) {
        selfview = oview;
        self = ((OperandView) oview).getOperand();
        parentview = oview.getParent();
        parent = ((OperandVectorView) parentview).getOperandVector();
        index = ((OperandVectorView) parentview).operandIndex(selfview);
    }
}
package scphEditor;
import java.awt.Component;
import java.awt.event.*;

public class OperandControl implements MouseListener, KeyListener
{
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
    public void keyTyped(KeyEvent e) {}
}
package acphEditor;

import java.util.Enumeration;
import java.util.NoSuchElementException;
import java.io.Serializable;

/**
 * This class models a vector of operands. Operands and Work are
 * vectors of operands.
 * @see Operand
 * @see Work
 */
public abstract class OperandVector
    extends CloneableObservable
    implements Cloneable, Serializable {
    public OperandVector() {
        super();
    }

    /**
     * Adopts the given new child as my child at the given index,
     * replacing the previous child at the same index.
     * @param newChild the new child
     * @param atIndex the index to where newChild goes
     * @exception java.lang.ArrayIndexOutOfBoundsException if atIndex
     * is an invalid index. (For leaf nodes, all indexes are invalid.)
     */
    public void setChild(Operand newChild, int atIndex) {
        throw new ArrayIndexOutOfBoundsException("No child exists");
    }

    /**
     * Gives my child at the given index.
     * @param atIndex the index at where a child will be sought.
     * @exception java.lang.ArrayIndexOutOfBoundsException if atIndex
     * is an invalid index. (For leaf nodes, this does not make
     * sense.)
     * @return the child at the index. (For leaf nodes, this does not make
     * sense.)
     */
    public Operand getChild(int atIndex) {
        throw new ArrayIndexOutOfBoundsException("No child exists");
    }

    public void addChild(Operand newChild, int atIndex) {
        throw new ArrayIndexOutOfBoundsException("No child exists");
    }

    public Operand delChild(int atIndex) {
        throw new ArrayIndexOutOfBoundsException("No child exists");
    }

    /**
     * Enumerates all of my children, in ascending order of indexes.
     * @return an Enumeration that iterates over my children in ascending
     * order of indexes. (For leaf nodes, returns an "empty" Enumeration.)
     */
    public Enumeration getChildren() {
        return new Enumeration() {
            public boolean hasMoreElements() { return false; }
            public Object nextElement() {
                throw new NoSuchElementException("No child in Operand");
            }
        };
    }

    /**
     * Gives the actual, present number of my children.
     * @return the present number of my children. (Zero for leaf nodes.)
     */
    public int countChildren() {
        return 0;
    }
}
package scphEditor;

import java.awt.Component;

public interface OperandVectorView {
    OperandVector getOperandVector();
    int operandIndex(Component child);
    Component getOperandView(int opindex);
}
package scphEditor;

public interface OperandView extends OperandVectorView
{
    Operand getOperand();
}
package scphEditor;
import java.awt.Component;

/**
 * This class is a factory of view objects for Operand objects. It is mainly a helper of the ExpressionView and ExpressionNaryView classes, when they recursively create views for children. Just call make() with the operand and you are done.
 * 
 * It does its job by being a visitor of concrete Operand objects. Each visit method creates an appropriate view object. In case the given Operand object is null, an appropriate view object is created without visiting.
 * 
 * @see OperandViewFactory#make
 */
public class OperandViewFactory implements OperandVisitor {
    private Component view;
    private OperandControl control;

    /**
     * Constructor.
     */
    public OperandViewFactory(OperandControl c) {
        view = null;
        control = c;
    }

    /**
     * Creates and returns a view object for the given Operand object.
     * It works for null too.
     * 
     * @param p the operand.
     * @return the view object.
     */
    public Component make(Operand p) {
        if (p == null) {
            makeNullView();
        } else {
            p.acceptVisitor(this);
        }
        return view;
    }

    public void visitVariable(Variable v) {
        view = new VariableView(v, control);
    }

    /**
     * Visits an identifier and creates an IdentifierView object.
     * 
     * @param id the visited identifier.
     */
    public void visitIdentifier(Identifier id) {
        view = new IdentifierView(id, control);
    }

    public void visitExpression(Expression e) {
        view = new ExpressionView(e, control);
    }

    public void visitExpressionNary(ExpressionNary e) {
        view = new ExpressionNaryView(e, control);
    }

    private void makeNullView() {
        view = new NullView(control);
    }
}
package scphEditor;

/**
 * This is the "visitor" interface, as in the Visitor pattern, for
 * visiting Operand trees. Since the acceptVisitor() methods in the
 * Operand class does not traverse or recurse into children, you have
 * do the traversal yourself, but it is up to you how (but please
 * document). All this interface specifies is just the names (entry
 * points) of the callbacks.
 *
 * @see Operand
 */
public interface OperandVisitor {
    /**
     * Called by Variable.acceptVisitor().
     *
     * @param id the Variable object that calls me.
     * @see Variable#acceptVisitor
     */
    void visitVariable(Variable v);

    /**
     * Called by Identifier.acceptVisitor().
     *
     * @param id the Identifier object that calls me.
     * @see Identifier#acceptVisitor
     */
    void visitIdentifier(Identifier id);

    /**
     * Called by Expression.acceptVisitor().
     *
     * @param e the Expression object that calls me.
     * @see Expression#acceptVisitor
     */
    void visitExpression(Expression e);

    /**
     * Called by ExpressionNary.acceptVisitor().
     *
     * @param e the ExpressionNary object that calls me.
     * @see ExpressionNary#acceptVisitor
     */
    void visitExpressionNary(ExpressionNary e);
}
package acphEditor;

/**
 * This interface models an operator. An Operator object is an
 * immutable object (so sorry, no clone), and should be used as
 * singletons and flyweights. You can query an Operator object
 * for its properties, and the whole point of this interface is
 * to define which properties can be queried.
 *<ul>
 *<li>The addition operator should be associative,
 * commutative, n-ary, and have a precedence lower (larger)
 * than the multiplication operator.
 *<li>The negation operator should have a high (small) precedence,
 * be non-associative, non-commutative, and unary.
 *<li>If you consider a variable or constant as an operator too,
 * it should have a very high precedence (say, above negation),
 * be non-associative, non-commutative, and 0-ary.
 *<ul>
 *<p>You can use the OperatorImpl class to create concrete operators,
 * or you can write your own implementation.
 *
 */
public interface Operator
{
    /**
     * If arity() returns nARY, this operator is an n-ary operator,
     * without an upper limit for the number of operands, though the lower
     * limit is 2. Examples are plus, times, and, or.
     *
     * @see Operator#arity
     */
    public final static int nARY = -2;

    /**
     * If associative() returns NONASSOC, this operator is non-associative.
     * @see Operator#associative
     */
    public final static int NONASSOC = 0;
    /**
     * If associative() returns LEFT, this operator is left-associative.
     * @see Operator#associative
     */
    public final static int LEFT = 1;
    /**
     * If associative() returns RIGHT, this operator is right-associative.
     * @see Operator#associative
     */
    public final static int RIGHT = 2;
    /**
     * If associative() returns ASSOC, this operator is (two-way) associative.
     * @see Operator#associative
     */
    public final static int ASSOC = 3;

    /**
     * Returns the precedence, as a number, of this operator. Larger number
     * means lower precedence (looser binding), smaller number means higher
     * precedence (tighter binding).
     */
    public int precedence();
package scphEditor;

/**
 * This concrete class provides a convenient implementation of the Operator
 * interface. You provide the operator properties (arity etc.) to the
 * constructor, and these properties will be used by the query services.
 */
public class OperatorImpl implements Operator
{
    // private String text;
    private int arity;
    private int precedence;
    private int associative;
    private boolean commutative;

    /**
     * Constructs an OperatorImpl object with the given properties.
     * @param arityOf arity of the operator
     * @param precOf precedence of the operator
     * @param assocOf associativity of the operator
     * @param commOf commutativity of the operator
     */
    public OperatorImpl(int arityOf, int precOf, int assocOf, boolean commOf)
    {
        arity = arityOf;
        precedence = precOf;
        associative = assocOf;
        commutative = commOf;
    }

    /**
     * Returns the precedence of this operator.
     */
    public int precedence()
    {
        return precedence;
    }

    /**
     * Returns the associativity of this operator.
     */
    public int associative()
    {
        return associative;
    }

    /**
     * Returns the commutativity of this operator.
     */
    public boolean commutative()
    {
        return commutative;
    }

    /**
     * Returns the arity of this operator.
     */
    public int arity()
    {
        return arity;
    }
}
package scphEditor;
import java.awt.*;
import java.awt.event.*;

public class Palette extends Frame
{
    private ScrollPane scroller;
    private Workspace myWorkspace;
    private Component selection;

    public Palette(String title) {
        super(title);
        selection = null;
        Work w = new Work();
        myWorkspace = new Workspace(w, new PaletteOperandControl());
        scroller = new ScrollPane();
        add(scroller, "Center");
        scroller.add(myWorkspace);
    }

    public Workspace getWorkspace() {
        return myWorkspace;
    }

    public synchronized void select(Component c) {
        if (c == selection) return;
        if (selection != null) {
            selection.setBackground(null);
            selection.setForeground(null);
            selection.repaint();
        }
        if (c != null) {
            c.setBackground(Color.black);
            c.setForeground(Color.white);
            c.repaint();
        }
        selection = c;
    }

class PaletteOperandControl extends OperandControl {
    public void mouseEntered(MouseEvent e) {
        Component c = e.getComponent();
        if (c instanceof NullView) c = c.getParent();
        select(c);
    }

    public void mouseExited(MouseEvent e) {
        Component c = e.getComponent();
        if (c instanceof NullView) c = c.getParent();
        if (c == selection) select(null);
    }

    public void mouseClicked(MouseEvent e) {
        Component c = e.getComponent();
        if (c instanceof NullView) c = c.getParent();
        if (c == selection) {

package scphEditor;

import java.awt.*;
import java.awt.event.*;

public class ProofMenu implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String cmd = e.getActionCommand();
        Component selected = Editor.recent().getSelection();
        if (selected == null) return;
        OperandContext c = new OperandContext(selected);
        if (cmd.equals("Send")) send(c);
        else if (cmd.equals("Simplify")) simplify(c);
    }

    public void send(OperandContext c)
    {
        if (c == null) return;
        TermBuilder b = new TermBuilder();
        ProverFrame.instance().execute("embedding.downcast ---* + b.make(c.self) + '*-');
    }

    public void simplify(OperandContext c)
    {
        if (c == null) return;
        ProverFrame pf = ProverFrame.instance();
        String output = pf.execute("val _ = (print o termlist.postfix o rhs o concl o simp
lib.SIMP_CONV bossLib.arith_as []) o embedding.downcast (---* + new TermBuilder().make(c.
self) + '*-');
        Editor e = Editor.recent();
        e.getWorkspace().getWork().addChild(TermBuilder.make(output), -1);
        e.validate();
    }
}
package scphEditor;

import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import java.util.Observer;
import java.util.Observable;

public class ProverFrame extends Frame implements Observer, ActionListener
{
    private static final String proverName = "hol98 -I scphEditor scphEditor/start.ML";
    private static final Font myFont = new Font("Monospaced", Font.PLAIN, 12);
    private Shell ps;
    private TextArea display;
    private TextField commandline;
    private StringBuffer output = new StringBuffer();
    private boolean ready = false;

    private static ProverFrame singleton;

    public static ProverFrame instance()
    {
        if (singleton == null) singleton = new ProverFrame();
        return singleton;
    }

    private ProverFrame()
    {
        super("Prover Monitor");
        display = new TextArea();
        display.setEditable(false);
        display.setFont(myFont);
        commandline = new TextField();
        commandline.addActionListener(this);
        commandline.setFont(myFont);

        try {
            ps = new Shell(proverName);
            ps.addObserver(this);
            ps.start();
        } catch (IOException e) {
            ps = null;
            display.setText("Unable to start prover
" + e.toString());
            commandline.setEditable(ps != null);
        }
        add("Center", display);
        add("South", commandline);

        addWindowListener(new WindowAdapter() {
            public void windowClosed(WindowEvent e) {
                System.out.println("ProverFrame closed");
                if (ps != null) ps.kill();
            }
        });
    }

    public void actionPerformed(ActionEvent a)
    {
        execute(commandline.getText());
        commandline.setText("*");
    }

    public void update(Observable o, Object arg)
    {
        String cmd = (String)arg;
        display.append(cmd);
        synchronized (output) {
            if (cmd.equals("-- ")) {
                ready = true;
                output.notify();
            } else if (cmd.endsWith("\n- ")) {
                ready = true;
                output.append(cmd.substring(0, cmd.length() - 3));
                output.notify();
            } else {
                output.append(cmd);
            }
        }
    }

    public String execute(String cmd)
    {
        if (ps == null) return "*";
        display.append(cmd + "\n");
        output.setLength(0);
        ready = false;
        try {
            ps.execute(cmd);
        } catch (IOException e) {
            display.append("Lost connection to prover\n");
            return "*";
        } synchronized (output) {
            try {
                while (!ready) output.wait();
            } catch (InterruptedException e) {
            }
        }
        return output.toString();
    }

    public void kill()
    {
        if (ps != null) ps.kill();
    }
}
package scphEditor;
import java.awt.*;

public class RowLayout implements LayoutManager
{
    private int gap;
    public RowLayout()
    {
        gap = 0;
    }
    public RowLayout(int g)
    {
        gap = g;
    }
    public int getGap()
    {
        return gap;
    }
    public void setGap(int g)
    {
        gap = g;
    }
    public void addLayoutComponent(String s, Component com)
    {
    }
    public void removeLayoutComponent(Component com)
    {
    }
    public Dimension minimumLayoutSize(Container con)
    {
        return preferredLayoutSize(con);
    }
    public Dimension preferredLayoutSize(Container con)
    {
        Dimension ps = new Dimension(0, 0);
        Component[] coms = con.getComponents();
        for (int i = 0; i < coms.length; i++) {
            Dimension s = coms[i].getPreferredSize();
            ps.height += s.height;
            if (ps.width < s.width) ps.width = s.width;
            if (i < coms.length - 1) ps.height += gap;
        }
        return ps;
    }
    public void layoutContainer(Container con)
    {
        int cx = 0, cy = 0;
        Component[] coms = con.getComponents();
        for (int i = 0; i < coms.length; i++) {
            Dimension s = coms[i].getPreferredSize();
            coms[i].setSize(s);
            coms[i].setLocation(cx, cy);
            cx += s.width + gap;
            cy += s.height + gap;
        }
    }
}
package scphEditor;

import java.io.*;
import java.util.Observable;

public class Shell extends Observable implements Runnable {
    private String processName;
    private Process process;
    private BufferedReader processOut; // hehe the senses are reversed
    private Writer processIn; // this too
    private Thread waiter;
    private boolean alive;

    public Shell(String pName) throws IOException {
        processName = pName;
        process = Runtime.getRuntime().exec(processName);
        // processIn = new BufferedWriter(new OutputStreamWriter(process.getOutputStream()));
        // processOut = new BufferedReader(new InputStreamReader(process.getInputStream()));
        waiter = new Thread(this);
        alive = false;
    }

    public void start() {
        waiter.start();
    }

    public void execute(String cmd) throws IOException {
        processIn.write(cmd);
        if (!cmd.endsWith("\n")) processIn.write("\n");
        processIn.flush();
    }

    public boolean isAlive() {
        return alive;
    }

    public void run() {
        alive = true;
        try {
            readChars();
        } catch (IOException e) {
            System.out.println(e.getMessage());
            alive = false;
        }
    }

    private void readChars() throws IOException {
        int len;
        char[] buf = new char[128];
        for (;;) {
            len = processOut.read(buf);
            if (len == -1) break;
            setChanged();
            notifyObservers(new String(buf, 0, len));
        }
    }

    public void kill() {
        try {
            processIn.close();
        } catch (IOException e) {
        }
        try {
            processOut.close();
        } catch (IOException e) {
        }
        // process.destroy();
    }

    private void readLines() throws IOException {
        String s;
        for (;;) {
            s = processOut.readLine();
            if (s == null) break;
            setChanged();
            notifyObservers(s + "\n");
        }
    }
}
package scphEditor;

public class Stencil extends Palette {
    public Stencil() {
        super("Stencil");
        Work stencilwork = getWorkspace().getWork();
        EmbeddingFactory f = EmbeddingFactory.instance();
        Operator op;
        Embedding em;
        // exponentiation
        op = new OperatorImpl(2, 2, Operator.RIGHT, false);
        em = new EmbeddingFixed(new String[] { null, "\u2297", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "EXP");
        // multiplication
        op = new OperatorImpl(Operator.NARY, 3, Operator.ASSOC, true);
        f.set(op,
                new EmbeddingNary("\u00d7"));
        stencilwork.addchild(new ExpressionNary(op), -1);
        TermBuilder.set(op, "**");
        // division
        op = new OperatorImpl(2, 3, Operator.LEFT, false);
        em = new EmbeddingFixed(new String[] { null, "\u2215", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "DIV");
        // modulus
        op = new OperatorImpl(2, 3, Operator.LEFT, false);
        em = new EmbeddingFixed(new String[] { null, "\u2216", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "MOD");
        // addition
        op = new OperatorImpl(Operator.NARY, 4, Operator.ASSOC, true);
        f.set(op,
                new EmbeddingNary("*"));
        stencilwork.addchild(new ExpressionNary(op), -1);
        TermBuilder.set(op, "+");
        // subtraction
        op = new OperatorImpl(2, 4, Operator.LEFT, false);
        em = new EmbeddingFixed(new String[] { null, "\u2212", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "-");
        // equality
        op = new OperatorImpl(2, 7, Operator.NONASSOC, true);
        em = new EmbeddingFixed(new String[] { null, "==", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "==");
        // <
        op = new OperatorImpl(2, 7, Operator.NONASSOC, false);
        em = new EmbeddingFixed(new String[] { null, "<", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "<");
        // >
        op = new OperatorImpl(2, 7, Operator.NONASSOC, false);
        em = new EmbeddingFixed(new String() { null,">", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, ">");
        // =
        op = new OperatorImpl(2, 12, Operator.NONASSOC, false);
        em = new EmbeddingFixed(new String[] { null, "==", null });
        f.set(op, em);
        stencilwork.addchild(new Expression(op), -1);
        TermBuilder.set(op, "==");
        // (seq comp)
        op = new OperatorImpl(Operator.NARY, 13, Operator.ASSOC, false);
        em = new EmbeddingNary("*");
        f.set(op, em);
        stencilwork.addchild(new ExpressionNary(op), -1);
        TermBuilder.set(op, "*");
        stencilwork.addchild(new Identifier("ok"), -1);
        stencilwork.addchild(new Identifier("\u2265", "F"), -1);
        stencilwork.addchild(new Identifier("\u2264", "T"), -1); // should be \u2264
package scphEditor;

import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Stack;
import java.util.StringTokenizer;

public class TermBuilder implements OperandVisitor{
    // from Operator to String
    private static Hashtable terms = new Hashtable();
    // from String to Operator
    private static Hashtable operators = new Hashtable();

    private StringBuffer buffer;

    public static void set(Operator op, String s)
    {
        terms.put(op, s);
        operators.put(s, op);
    }

    public static String get(Operator op)
    {
        return (String) terms.get(op);
    }

    public static Operator get(String s)
    {
        return (Operator) operators.get(s);
    }

    public TermBuilder()
    {
        buffer = new StringBuffer();
    }

    public static Operand make(String s)
    {
        StringTokenizer tokens = new StringTokenizer(s);
        Stack stack = new Stack();
        while (tokens.hasMoreTokens())
        {
            String t = tokens.nextToken();
            if ("beginEps\(\)".equals(t)) break;
        }
        while (tokens.hasMoreTokens())
        {
            String t = tokens.nextToken();
            if (t.equals("\") )
            {
                Operand parent = (Operand) stack.pop();
                Operand child = (Operand) stack.pop();
                int i = 0;
                while (parent.getChild(i) != null) i++;
                parent.setChild(child, i);
                stack.push(parent);
            }
            else if (t.equals("\") )
            {
                Operand x;
                Operator o = null;
                x = Identifier.lookup(t);
                if (x == null) {
                    o = get(t);
                    if (o == null) {
                        x = new Identifier(t);
                    } else 
                    {
                        if (o.arity() == Operator.nARY) {
                            x = new ExpressionNary(o);
                        } else
                        {
                            x = new Expression(o);
                        }
                    }
                }
            }
            else {
                stack.push(x);
            }
        }
        return stack.empty() ? null : (Operand) stack.pop();
    }

    public String make(Operand p)
    {
        buffer.setLength(0);
        if (p == null) {
            // hmm...
        } else {
            p.acceptVisitor(this);
        }
        return buffer.toString();
    }

    public void visitVariable(Variable v)
    {
        buffer.append(v.getName() + ":");
        buffer.append(v.getType());
    }

    public void visitIdentifier(Identifier id)
    {
        buffer.append(id.getName());
    }

    public void visitExpression(Expression e)
    {
        buffer.append("*");
        buffer.append(get(e.getOperator()));
        buffer.append(" * ");
        Enumeration i = e.getChildren();
        while (i.hasMoreElements())
        {
            Operand p = (Operand) i.nextElement();
            buffer.append(" * ");
            p.acceptVisitor(this);
            buffer.append(" * ");
        }
    }

    public void visitExpressionNary(ExpressionNary e)
    {
        String opname = get(e.getOperator());
        Enumeration i;
        Operand p;
        switch (e.getOperator().associative()) {
            case "}
                break;
            case "}
                break;
            default:
                break;
        }
    }
}
case Operator.LEFT:
    for (int j = 1; j < e.getChildren().size(); j++) {
        buffer.append("(* */");
    }
    i = e.getChildren().get(0);
    p = (Operand) i.nextElement;
    buffer.append("(* */");
    p.acceptVisitor(this);
    buffer.append("(* */");
    while (i.hasMoreElements()) {
        p = (Operand) i.nextElement;
        buffer.append(opname);
        buffer.append("(* */");
        p.acceptVisitor(this);
        buffer.append("(* */");
    }
    break;

case Operator.RIGHT:
    break;

case Operator.ASSOC:
    i = e.getChildren().get(0);
    p = (Operand) i.nextElement;
    buffer.append("(* */");
    p.acceptVisitor(this);
    buffer.append("(* */");
    while (i.hasMoreElements()) {
        p = (Operand) i.nextElement;
        buffer.append("(* */");
        buffer.append(opname);
        buffer.append("(* */");
        p.acceptVisitor(this);
        buffer.append("(* */");
    }
    break;
}
package scphEditor;

public class Variable extends Identifier
{
  private String type;

  public Variable(String nameOf, String typeOf)
  {
    super(nameOf);
    type = typeOf;
  }

  public String getType()
  {
    return type;
  }

  /**
   * Calls v.visitIdentifier(this).
   */
  public void acceptVisitor(OperandVisitor v)
  {
    v.visitVariable(this);
  }
}
package svpEditor;

public class VariableView extends IdentifierView {
    public VariableView(Identifier idOf, OperandControl o) {
        super(idOf, ch);
        setLocationFrame(idOf, IDOWN);
    }
}
package scphEditor;
import java.awt.*;
import java.awt.event.*;
import java.util.Enumeration;
import java.util.Vector;

public class Variables extends Palette
{
  private Vector progVars;
  public Variables()
  {
    super("Variables");
    progVars = new Vector();
    MenuBar bar = new MenuBar();
    setMenuBar(bar);
    Menu edit = new Menu("Edit");
    bar.add(edit);
    MenuItem add = new MenuItem("Add");
    edit.add(add);
    add.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        addVariable();
      }
    });
    //   MenuItem del = new MenuItem("Del");
    //   edit.add(del);
  }

  private void addVariable()
  {
    ask = new Dialog(this, "Add Variable", true);
    ask.setLayout(new FlowLayout());
    TextField v = new TextField(6);
    v.setFont(new Font("Monospaced", Font.PLAIN, 20));
    TextField t = new TextField(6);
    ask.add(new Label("Name:", true));
    ask.add(v);
    ask.add(new Label("Type:", true));
    ask.add(t);
    Button ok = new Button("OK");
    ask.add(ok);
    Button cancel = new Button("Cancel");
    ask.add(cancel);
    ok.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        answer = true;
        ask.dispose();
      }
    });
    cancel.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        answer = false;
        ask.dispose();
      }
    });

    if (answer) {
      String vName = v.getText(), vType = t.getText();
      Variable vPre = new Variable(vName, vType);
      progVars.addElement(vPre);
      getWorkspace().getWork().addChild(vPre, -1);
      getWorkspace().getWork().addChildren(vPost, -1);
      validate();
      setVariableList();
    }
  }
}

private void setVariableList()
{
  StringBuffer cmd = new StringBuffer();
  cmd.append("embedding.setme " + vName + ");
  for (Enumeration i = progVars.elements(); i.hasMoreElements(); )
  {
    Variable v = (Variable) i.nextElement();
    cmd.append("--");
    cmd.append(v.getName());
    cmd.append(v.getType());
    cmd.append("--");
    if (i.hasMoreElements()) cmd.append("+");
  }
  cmd.append(";");
  proverFrame.instance().execute(cmd.toString());
}
package scphEditor;
import java.util.Vector;
import java.util.Enumeration;

/** *
 * This class models an scphEditor document, a list of operands and
 * stuff.
 * *
 * <p>When a Work object changes, it will call notifyObservers() with an Integer object as a parameter:
 * *
 * <ul>
 * <li>If the value is n, n+1, then new stuff is inserted at index n-1.
 * <li>If the value is -n, n+1, then old stuff at index n+1 is deleted.
 * </ul>
 * */

public class Work extends OperandVector {
    /** *
     * List of operands and stuff.
     */
    private Vector content;

    /** *
     * Constructor.
     */
    public Work() {
        content = new Vector();
    }

    public void setChild(Operand newChild, int atIndex) {
        // content.setElementAt(newChild, atIndex);
        // setChanged();
        // notifyObservers(new Integer(atIndex+1));
        delChild(atIndex);
        addChild(newChild, atIndex);
    }

    public void addChild(Operand newChild, int atIndex) {
        if (atIndex == -1) atIndex = content.size();
        content.insertElementAt(newChild, atIndex);
        setChanged();
        notifyObservers(new Integer(atIndex+1));
    }

    public Operand delChild(int atIndex) {
        Operand o = (Operand) content.elementAt(atIndex);
        content.removeElementAt(atIndex);
        setChanged();
        notifyObservers(new Integer(-atIndex-1));
        return o;
    }

    public Operand getChild(int atIndex) {
        return (Operand) content.elementAt(atIndex);
    }

    public Enumeration getChildren() {public int countChildren() {
        return content.size();
    }
}
package apchEditor;

import java.awt.*;
import java.awt.event.*;
import java.util.Observable;
import java.util.Observer;

// Workspace holds the formulae

public class Workspace extends Panel implements Observer, OperandVectorView
{
    public static Font IDPONT = new Font("Serif", Font.ITALIC, 14);
    public static Font OPPONT = new Font("Serif", Font.PLAIN, 14);
    public static Font UNDEFPONT = new Font("Serif", Font.BOLD, 14);
    public static Font TEXTFONT = OPPONT;

    private Work myClient;
    private OperandControl childControl;

    public Workspace(Work w, OperandControl c)
    {
        super();
        setLayout(new RowLayout());
       setFont(TEXTFONT);
        setForeground(Color.black);
        setBackground(Color.white);
        WorkspaceControl wc = new WorkspaceControl();
        addMouseListener(wc);
        addKeyListener(wc);

        myClient = w;
        myClient.addObserver(this);
        childControl = c;

        OperandViewFactory f = new OperandViewFactory(c);
        for (int i = 0; i < myClient.countChildren(); i++)
        {
            add(f.make(myClient.getChild(i)));
        }
    }

    public Work getWork()
    {
        return myClient;
    }

    public OperandVector getOperandVector()
    {
        return myClient;
    }

    public int operandIndex(Component c)
    {
        Component[] cs = getComponents();
        for (int i = 0; i < cs.length; i++)
        {
            if (cs[i] == c) return i;
        }
        return -1; // or throw exception?
    }

    public Component getOperandView(int opindex)
    {
        return GetComponent(opindex);
    }

    public void update(Observable w, Object a)
    {
        if (myClient != w) return;
        int n = ((Integer)a).intValue();
        if (n >= 1)
        {
            OperandViewFactory f = new OperandViewFactory(childControl);
            add(f.make(myClient.getChild(n-1)), n-1);
        }
        else
        {
            remove(-n-1);
        }
    }

    // public static Workspace getWorkspace(Component c)
    // {
    //     for (Component i = c; i != null; i = i.getParent())
    //     {
    //         if (i instanceof Workspace)
    //         {
    //             return (Workspace) i;
    //         }
    //     }
    //     return null;
    // }
}

public void update(Observable w, Object a)
{
    if (myClient != w) return;
    int n = ((Integer)a).intValue();
    if (n >= 1)
    {
        OperandViewFactory f = new OperandViewFactory(childControl);
        add(f.make(myClient.getChild(n-1)), n-1);
    }
    else
    {
        remove(-n-1);
    }
}

// public static Workspace getWorkspace(Component c)
// {
//     for (Component i = c; i != null; i = i.getParent())
//     {
//         if (i instanceof Workspace)
//         {
//             return (Workspace) i;
//         }
//     }
//     return null;
// }
package scphEditor;
import java.awt.*;
import java.awt.event.*;

public class WorkspaceControl implements MouseListener, KeyListener
{
    public void mouseClicked(MouseEvent e)
    {
        if ((e.getModifiers() & InputEvent.BUTTON1_MASK) != 0) {
            Container c = e.getComponent().getParent();
            if (c instanceof Editor) ((Editor) c).deselect();
        }
    }

    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
    public void keyTyped(KeyEvent e) {}
}
structure embedding = struct
local
  open HolKernel basicHOLLib
infix |->
  fun prime_var v = let
    val (Name = n, Ty = t) = dest_var v
  in
    mk_var (Name = n ^ 't', Ty = t)
  end
in
  fun gimme_prog_vars = let
    val logic_postvars = map prime_var prog_vars
    val logic_vars = prog_vars @ logic_postvars
    (* val prog_var_types = map type_of prog_vars *)
    (* ok *)
    val free_ok = list_mk_conj (ListPair.map (fn (v,vp) => mk_eq (lhs=vp, rhs=v)) (prog_vars, logic_postvars))
    (* assignment *)
  fun mk_asgn equality = let
    val (lhs = var, rhs = expr) = dest_eq equality
  in
    mk_let (arg = expr, func = mk_abs (Bvar = var, Body = free_ok))
  end
  (* dependent composition *)
  fun depcomp S R = let
    val avoid = (all_vars S) @ (all_vars R)
    val variants = map (variant avoid) prog_vars
    fun bind vl = ListPair.map (op |->) (vl,variants)
  in
    list_mk_exists(variants, mk_conj {conj1 = subst (bind logic_postvars) S, conj2 = subst (bind prog_vars) R})
  end
in
  {free_ok, mk_asgn, depcomp}
end
val ok = ref (¬'T'¬)
val asgn = ref (fn n : term => ¬'T'¬)
val dc = ref (fn s : term => fn r : term => mk_conj {conj1 = s, conj2 = r})

fun setme_prog_vars = let
  val {x, y, z} = gimme_prog_vars
in
  ok := x; asgn := y; dc := z
end

fun downcast t = let
  fun d (VAR r) = mk_var r
  | d (CONST (Name = "ok", Ty = _)) = !ok
  | d (CONST r) = mk_const r
  | d (LAMB (Bvar = v, Body = b)) = mk_abs (Bvar = downcast v, Body = downcast b)
  | d (COMB (Rator = f, Rand = x)) = let
      val (Rator = g, Rand = w) = dest_comb f
      (* so the term is "g w x" *)
      val (Name = n, ...) = dest_const g
      in
        if n = "=" then !asgn (mk_eq (lhs = w, rhs = x))
        else if n = "::" then !dc (downcast w) (downcast x)
        else mk_comb (Rator = downcast f, Rand = downcast x)
      end
    end
  handle HOL_ERR _ => mk_comb (Rator = downcast f, Rand = downcast x)
structure termlist = struct
  local open Term Parse in

    fun postfix t = let
      fun postfix_2 (COMB (Rator = x, Rand = y)) L
        = postfix_2 (dest_term y) (postfix_2 (dest_term x) (["\", "\"] @ L))
        | postfix_2 (LAMB (Bvar = v, Body = b)) L
        = postfix_2 (dest_term b) (postfix_2 (dest_term v) (["\", "\"] @ L))
        | postfix_2 (CONST (c as (Name = n, Ty = t))) L = [n, ""] @ L
        | postfix_2 (VAR (v as (Name = n, Ty = t))) L = [n, ""] @ L

      in
        String.concat (["begin ": postfix_2 (dest_term t) ["\n"]])
      end

    end

(* the following are cool examples
val it = postfix ("x:bool = y"--); val it = postfix ("x. x ==> T"--); val it = postfix ("(\x. x) y:bool = y"--); val it = postfix ("(\y. x) x / y (x:bool = y) T"--); val it = postfix ("f (x \ y) : bool"--); *)