# 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

# 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 H. 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".

# Contents

# 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, Math∫pad, 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 computer 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

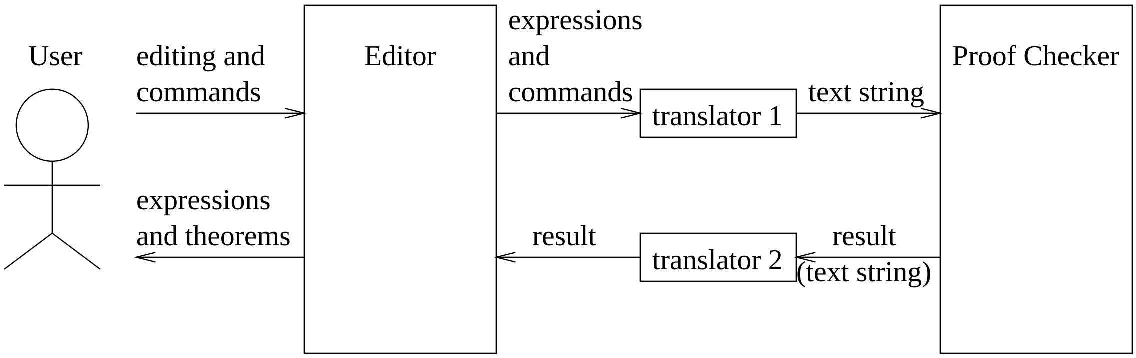


Figure 1.1: Block Diagram of the Tool

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.

  - 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:

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

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

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

## 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 *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, *ok* is then defined as

$$x' = x \wedge y' = y \wedge z' = z$$

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

$$y' = x \wedge x' = x \wedge z' = z$$

Another programming construct is *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

$$P \,.\, Q$$

is defined as

$$\exists x'', y'', z'' \cdot P[x''/x', y''/y', z''/z'] \wedge Q[x''/x, y''/y, z''/z]$$

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 $x$.

$$\begin{aligned}
& y{:=}\,x \,.\, z{:=}\,y \\
= \;& (y' = x \wedge x' = x \wedge z' = z) \,.\, (z' = y \wedge x' = x \wedge y' = y) \\
= \;& \exists x'', y'', z'' \cdot (y'' = x \wedge x'' = x \wedge z'' = z) \wedge (z' = y'' \wedge x' = x'' \wedge y' = y'') \\
= \;& x' = x \wedge y' = x \wedge z' = x
\end{aligned}$$

The following is an instance of the *substitution law*. Often we have a dependent composition $x{:=}\,e \,.\, P$ 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 $P[e/x]$:

$$\begin{aligned}
& x{:=}\,e \,.\, P \\
= \;& (x' = e \wedge y' = y \wedge z' = z) \,.\, P
\end{aligned}$$

$$\begin{aligned} &= \exists x'', y'', z'' \cdot (x'' = e \wedge y'' = y \wedge z'' = z) \wedge P[x''/x, y''/y, z''/z] \\ &= P[e/x] \end{aligned}$$

Iteration is discussed in the next section.

## 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 *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:

$$x' = 0 \Leftarrow \textbf{if } x = 0 \textbf{ then } ok \textbf{ else } (x{:=}\, x - 1 \,.\, x' = 0)$$

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 $P \Leftarrow A \,.\, B$, then we have $P \Leftarrow A' \,.\, B'$

   (b) If we have $P \Leftarrow$ **if** $b$ **then** $A$ **else** $B$, then we have $P \Leftarrow$ **if** $b$ **then** $A'$ **else** $B'$

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

2. Refinement by parts:

   (a) If we have $P_1 \Leftarrow (A_1 \,.\, B_1)$ and $P_2 \Leftarrow (A_2 \,.\, B_2)$, then we have $P_1 \wedge P_2 \Leftarrow (A_1 \wedge A_2 \,.\, B_1 \wedge B_2)$

   (b) If we have $P_1 \Leftarrow$ **if** $b$ **then** $A_1$ **else** $B_1$ and $P_2 \Leftarrow$ **if** $b$ **then** $A_2$ **else** $B_2$, then we have $P_1 \wedge P_2 \Leftarrow$ **if** $b$ **then** $A_1 \wedge A_2$ **else** $B_1 \wedge B_2$

   (c) If we have $P_1 \Leftarrow A_1$ and $P_2 \Leftarrow A_2$, then we have $P_1 \wedge P_2 \Leftarrow A_1 \wedge A_2$

3. Refinement by cases: We have

   $$P \Leftarrow \textbf{if } b \textbf{ then } A \textbf{ else } B$$

   if and only if we have $P \Leftarrow b \wedge A$ and $P \Leftarrow \neg b \wedge B$

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 *ok* is the function

$$\lambda x, x', y, y' \cdot x' = x \wedge y' = y$$

   More examples are illustrated in the following table:

| Editor | Proof Checker |
|---|---|
| $ok$ | $\lambda x, x', y, y' \cdot x' = x \wedge y' = y$ |
| $x < x'$ | $\lambda x, x', y, y' \cdot x < x'$ |
| $x < x' \,.\, y < y'$ | $\lambda x, x', y, y' \cdot \exists x'', y'' \cdot$ <br> $(\lambda x, x', y, y' \cdot x < x')\, x\, x''\, y\, y'' \wedge (\lambda x, x', y, y' \cdot y < y')\, x''\, x'\, y''\, y'$ |
| $x{:=}\,x+1$ | $\lambda x, x', y, y' \cdot ok\; x\, (x+1)\, y\, y'$ |

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 $ok$, ., and := are handled and simplified by the editor, which is programmed with known laws about them such as:

   (a) $ok \,.\, P = P$

   (b) $P \,.\, ok = P$

   (c) $x{:=}\,e \,.\, P = P[e/x]$

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.,

| Editor | Proof Checker |
|---|---|
| $ok$ | $x' = x \wedge y' = y$ |
| $x < x'$ | $x < x'$ |
| $x < x' \;.\; y < y'$ | $\exists x'', y'' \cdot x < x'' \wedge y'' < y'$ |
| $x{:=}\,x + 1$ | substitute $x + 1$ for $x$ in $x' = x \wedge y' = y$ |

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

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

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.

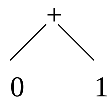


Figure 3.1: An Expression Tree Inside the Editor

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

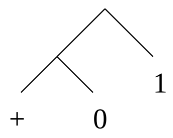


Figure 3.2: An Expression Tree Inside HOL

nodes represent values or functions. (This is to accomodate higher-order expressions.)

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 straightfoward: 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:

```
$==> (ok) ($:= (x : num) ((x : num) + (0) + (0)))
```

This example illustrates two aspects of our translation. First, since HOL accepts prefix notation as long as the `$` sign is prepended to an operator, we take advantage of it to simplify our programming. So for example, HOL accepts both `A ==> B` (infix form) and `$==> 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) /\ (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:

```
~(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 = ~ ==>
```

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 = . . ~ . ==> . .
```

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:

if _ then _ 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.)

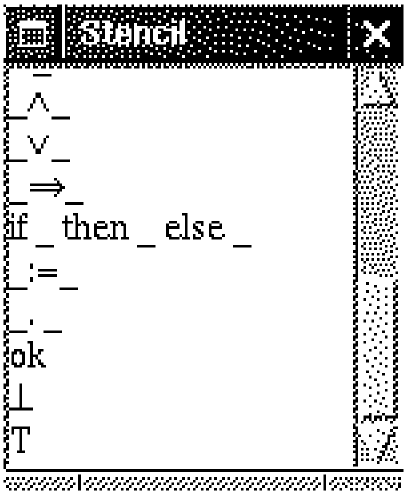


Figure 4.1: The Stencil Window

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

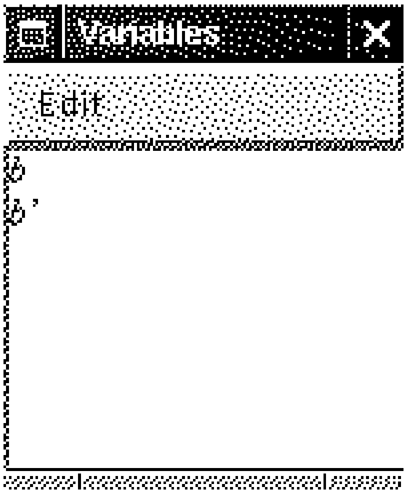


Figure 4.2: The Variables Window (with variable $b$ added)

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.

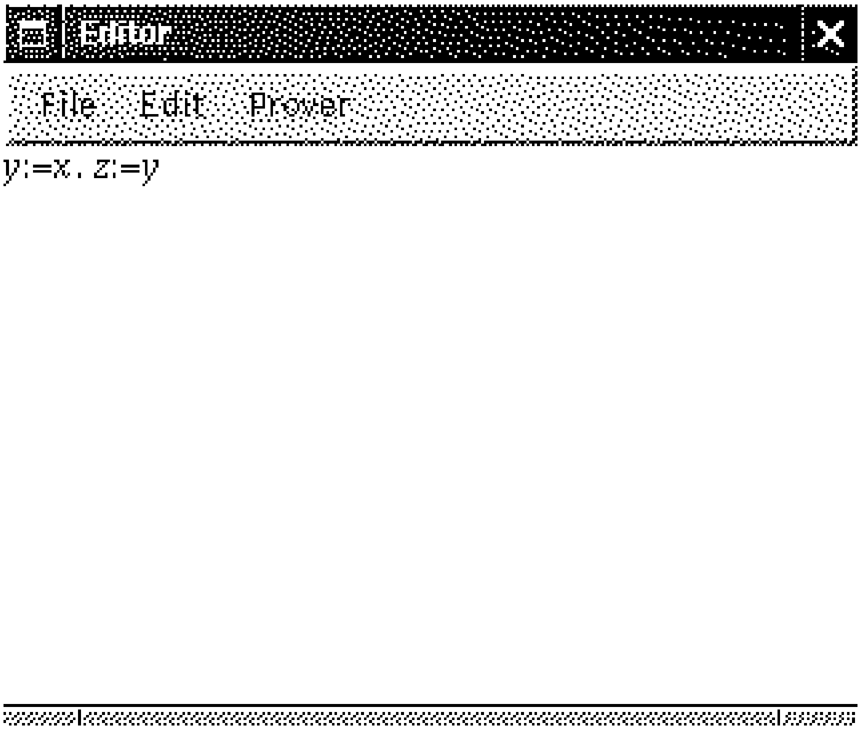


Figure 4.3: The Editor Window (with $y{:=}\,x$ . $z{:=}\,y$ entered)

The method of expression editing follows that of Math∫pad [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 tree structure.

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 $y{:=}x\ .\ z{:=}y$, 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:

$$y{:=}\,x\ .\ \_$$

6. enter $z$:

$$y{:=}\,x\ .\ z$$

7. enter the assignment operator

$$y{:=}\,x\ .\ z{:=}\,\_$$

8. enter $y$:

$$y{:=}\,x\ .\ z{:=}\,y$$

The editor recognizes associative commutative operators, such as $+$ and $\vee$, 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

$$y{:=}\,x\ .\ z{:=}\,y$$

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:

$$x' = x \wedge (y' = x \wedge z' = x)$$

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:

$$(y{:=}\,x\ .\ z{:=}\,y) = (x' = x \wedge y' = x \wedge z' = x)$$

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.

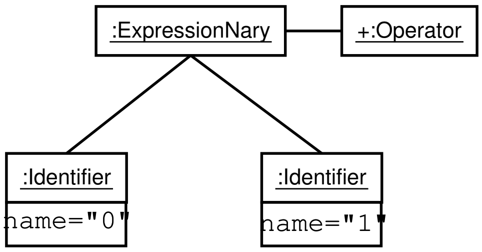


Figure 4.4: An Expression Tree

`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:

$$n \geq 1 \;\Rightarrow\; 2^{x'} \times y' = n \;\wedge\; \neg\, y' \bmod 2 = 0$$

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 `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:

$$\begin{aligned} & x{:=}\,0 \;.\; y{:=}\,n \;.\; 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \;\wedge\; \neg\, y' \bmod 2 = 0) \\ \Rightarrow \;\; & n \geq 1 \Rightarrow 2^{x'} \times y' = n \;\wedge\; \neg\, y' \bmod 2 = 0 \end{aligned}$$

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 peforms the equivalent of the substitution law, reduces the antecedent $2^0 \times n = n$ 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

$$2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)$$

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:

$$\begin{array}{ll}
& \textbf{if } \neg\, y \bmod 2 = 0 \\
& \textbf{then } ok \\
& \textbf{else } y \bmod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)) \\
\Rightarrow & 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)
\end{array}$$

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.

$$\begin{array}{ll}
& \textbf{if } \neg\, y \bmod 2 = 0 \\
& \textbf{then } \neg\, y \bmod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)) \\
& \textbf{else } y \bmod 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)) \\
\Rightarrow & 2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)
\end{array}$$

2.

$$\begin{aligned} & ok \\ \Rightarrow \ & \neg\, y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \ \wedge\ \neg\, y' \text{ mod } 2 = 0)) \end{aligned}$$

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 $ok$ to replace all other occurences 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:

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

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:

$$\begin{aligned} & y{:=}\, y \text{ div } 2\ .\ x{:=}\, x + 1\ .\ (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \ \wedge\ \neg\, y' \text{ mod } 2 = 0)) \\ \Rightarrow \ & y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \ \wedge\ \neg\, y' \text{ mod } 2 = 0)) \end{aligned}$$

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:

$$\begin{aligned} & (2^{x+1} \times y \text{ div } 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \ \wedge\ \neg\, y' \text{ mod } 2 = 0)) \\ \Rightarrow \ & y \text{ mod } 2 = 0 \Rightarrow (2^x \times y = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \ \wedge\ \neg\, y' \text{ mod } 2 = 0)) \end{aligned}$$

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 $(y \operatorname{div} 2) \times 2 + y \bmod 2$ 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 $y \bmod 2$ can be discarded thanks to the antecedent $y \bmod 2 = 0$. So we get:

$$
\begin{array}{rl}
& (2^{x+1} \times y \operatorname{div} 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)) \\
\Rightarrow & y \bmod 2 = 0 \Rightarrow (2^x \times (y \operatorname{div} 2) \times 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0))
\end{array}
$$

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 $2^x \times (y \operatorname{div} 2) \times 2$ as $2^{x+1} \times y \operatorname{div} 2$ by arithmetic, yielding:

$$
\begin{array}{rl}
& (2^{x+1} \times y \operatorname{div} 2 = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0)) \\
\Rightarrow & y \bmod 2 = 0 \Rightarrow (2^{x+1} \times (y \operatorname{div} 2) = n \Rightarrow (n \geq 1 \Rightarrow 2^{x'} \times y' = n \wedge \neg\, y' \bmod 2 = 0))
\end{array}
$$

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


[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, second edition, 1997.

[2] R. J. Back, J. Grundy, and J. von Wright. Structured calculational proof. *Formal Aspects of Computing*, 9:469–483, 1997.

[3] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[4] R. C. Backhouse, R. Verhoeven, and O. Weber. Math∫pad: A system for on-line preparation of mathematical documents. *Software – Concepts and Tools*, 18:80–89, 1997.

[5] Roland Backhouse and Richard Verhoeven. *Math∫pad: Ergonomic Document Preparation.* Department of Mathematics and Computing Science, Eindhoven University of Technology, June 1996. WWW Home Page: http://www.win.tue.nl/cs/wp/mathpad/.

[6] Roland Backhouse and Richard Verhoeven. Math∫pad: Tool support for the calculational method. Downloadable Paper, February 1996. http://www.win.tue.nl/cs/wp/mathpad/CalculationMethod.ps.gz.

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

Steve Reeves, editors, *Formal Methods Pacific '97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer.

[8] Michael Butler and Thomas Långbacka. Program derivation using the refinement calculator. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 93–108, Turku, Finland, August 1996. Springer.

[9] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[10] Jim Grundy. Transformational hierarchical reasoning. *The Computer Journal*, 39(4):291–302, 1996.

[11] Eric C. R. Hehner. A practical theory of programming. *Science of Computer Programming*, 14(2–3):133–158, October 1990.

[12] Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer, 1993.

[13] The HOL system. WWW Page. http://www.cl.cam.ac.uk/Research/HVG/HOL/.

[14] Andrzej Kondracki. Zastosowanie środowiska PVS w metodzie kart. Master's thesis, Institute of Informatics, Warsaw University, Poland, 1997.

[15] T. Långbacka, R. Rukšėnas, and J. von Wright. TkWinHOL: A tool for doing window inference in HOL. In E. Thomas Schubert, Philip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 245–260, Aspen Grove, Utah, USA, September 1995. Springer.

[16] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[17] The PVS specification and verification system. WWW Page. http://pvs.csl.sri.com/.

[18] Richard Verhoeven and Roland Backhouse. Interfacing program construction and verification. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - Formal Methods*, volume 1709 of *Lecture Notes in Compute Science*, pages 1128–1146, Toulouse, France, September 1999. Springer.

[19] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.