TkWinHOL:
A Tool for Window Inference in HOL

Thomas Långbacka, Rimvydas Rukšėnas, Joakim von Wright

Åbo Akademi University
Department of Computer Science
Lemminkäinengatan 14–18
SF–20520 Åbo, Finland

Abstract. Window inference is a method for contextual rewriting and refinement, supported by the HOL Window Inference Library. This paper describes a user-friendly interface for window inference. The interface permits the user to select subexpressions by pointing and clicking and to select transformations from menus. The correctness of each transformation step is proved automatically by the HOL system. The interface can be tailored to particular user-defined theories. One such extension, for program refinement, is described.

1 Introduction

Though the original purpose of the HOL system [10] was as a tool for hardware verification, it has become popular also as a basis for reasoning about software (see for example [1, 7, 8]). However, theories built for supporting the software development process are normally difficult to use, especially if one does not have any previous detailed knowledge of the HOL system. In order to make such theories available to a general audience, it is essential that users have support in the form of a user interface, either a general HOL interface such as [19] or a specialised tool such as [2]. This observation has motivated us to design a tool that supports development of provably correct programs within the framework of the Refinement Calculus [3, 4, 6].

Our work is based on a number of layers. The starting point is the HOL system and its Window Inference Library [11, 12]. Window inference (originally described in [17]) supports the notion of contextual refinement which is an important ingredient in the refinement calculus as well. We also use the HOL theory for the Refinement Calculus [21]. On top of these packages we have designed a graphical user interface that itself is designed in a layered fashion. The basic user interface is called TkWinHOL, and it offers users an X-windows based graphical interface to the window inference system.

TkWinHOL is built using the Tcl/Tk programming system [15], which is a general purpose environment for building user interfaces. Tcl/Tk applications are normally memory conserving, which is useful when building interfaces to large systems such as HOL. TkWinHOL is purposely designed to be easy to extend and modify. As an extension to the TkWinHOL system, we are developing a tool for the Refinement Calculus.

2 Background

Our original motivation for developing a tool, such as the one presented in this paper, was to provide an environment supporting development of provably correct programs within the framework of the Refinement Calculus [3,4]. This calculus is a formalization of the stepwise refinement method for program construction, based on the weakest precondition calculus of Dijkstra [9]. The basic idea is to start from a high level specification and refine it through a sequence of correctness preserving steps into an executable program. The basic notion in the Refinement Calculus is the refinement relation between program statements. Statement S' is said to be a correct refinement of statement S if it preserves the total correctness of the latter. The refinement relation is a preorder (i.e., it is reflexive and transitive). This fact justifies the stepwise refinement method. Another important aspect of the Refinement Calculus is its top-down approach to program development; a part of a larger program can be refined independently of its context.

A starting point for our work was the Centipede environment [5]. Centipede was a mouse and menu driven environment that supported stepwise development of programs within the Refinement Calculus. However, Centipede did not support mechanical verification of refinement steps. Thus, we wanted to develop a similar tool with this added capability.

The choice of HOL as the proof engine underlying the system was simple, because of previous work on formalizing the Refinement Calculus in HOL, which has spanned several years [20,21]. Furthermore, the Window Inference Library of the HOL system provides support for doing refinement transformations easily in HOL. In window inference, terms (or subterms) of a logic are transformed, preserving some preorder. The user indicates what transformation is desired and provides information describing the path to the subterm that should be transformed. Computing paths is tedious for an ordinary user, but it can be automated within a user interface. In fact, using the mouse to select the data (in this case subterms) one wants to perform operations on is a well established principle in modern user interfaces. Thus window inference lends itself well as the basis for a modern user interface to HOL. Most of the features that one wants from such an interface are not specific to program refinement but are desirable for arbitrary transformational reasoning using window inference. TkWinHOL is an attempt to provide such a general user interface. The basic idea is to provide facilities for manipulating any object that is already present in the system just by pointing and clicking with the mouse. New functionality needed for program refinement or any other kind of transformational reasoning can then be obtained by extending the basic TkWinHOL interface.

There were initial attempts to use both the Centaur system (which is used also in Théry's chol system [19]) and the Synthesizer Generator [16]. Both these systems make it possible to build syntax directed editors (in fact the Centipede system also had a syntax directed editor as it's base) for constructing programs in some user defined language. Both systems also allow one to define transformation rules that operate on the programs (or more generally some syntactically correct term in a given language). However, the executables that these systems produce are very big. Keeping in mind that HOL90 itself is a big system this is not a nice feature. Furthermore, both systems have built-in assumptions that can be restrictive, e.g., regarding the ways in which syntax can be manipulated.

To implement the interface we have chosen the Tcl/Tk programming system. This choice has turned out to be a good one. Tcl/Tk is a general purpose system that is well suited for building user interfaces for existing UNIX applications (HOL in our case). We have managed to keep TkWinHOL small and efficient. And given the freedom provided by a general purpose system we have also been able to add certain 'unorthodox' features to the system (e.g., dealing with associative operators as explained in Section 4).

3 Window inference

In window inference, the user transforms a term so that a preorder (a reflexive transitive relation) is preserved. The user can also restrict attention to a subterm of the term that is being transformed. A transformation of the subterm is at the same time a transformation of the whole term, provided that certain context-dependent monotonicity conditions hold.

In the HOL Window Inference Library there is built-in support for window reasoning under equality, implication and reverse implication. Furthermore, the user can add other preorders. In HOL, window reasoning is handled as follows. Assume that $E[t]$ is a term with subterm t. Our starting point is a *window stack* containing the window

$$R \;\star\; E[t]$$

The *focus* of the window is the term $E[\ldots t \ldots]$ and the relation that we want to preserve (the *window relation*) is R (the star simply separates these two). In order to set up such a stack, the user must first load the HOL Window Inference Library and then type the appropriate command for beginning a stack.

The theorem associated with this window (the *window theorem*) is

$$\vdash \; E[\ldots t \ldots] \; R \; E[\ldots t \ldots]$$

which holds because R is reflexive.

Opening subwindows. We can *open a subwindow* on the subterm t. This is done by entering a command of the form `DO(OPEN_WIN path)`, where `path` is a list of path elements that indicate how we navigate in the abstract syntax tree of E in order to get to the subterm t. In view of the simple syntax of higher order logic, the possible path elements are `RATOR`, `RAND` and `BODY` (we assume that the reader is familiar with higher order logic and has some experience of the HOL system). The result of the opening command is the following window:

$$R' \;\star\; t$$

with the corresponding window theorem (again by reflexivity)

$$\vdash \; t \; R' \; t$$

Here R' may be the same relation as R, but it can be different. For example, R can be implication and R' equality (this would be the case if the focus was $x+1-1 \leq y$, the relation was implication and we opened a subwindow on $x+1-1$). We now transform t under the relation R', using available transformation rules (e.g., rewriting, beta-conversions, matching with pre-proved theorems etc). Assume that the result of these transformations is t'. The new window theorem is then

$$\vdash \; t \; R' \; t'$$

(the system uses transitivity of R' to draw this conclusion). We can now close the subwindow by entering the command `DO CLOSE_WIN`.

Closing and window rules. At this point, the window inference system uses a collection of *window rules* (ML functions that perform HOL proofs) to do the following inference:

$$\frac{\vdash \; t \; R' \; t'}{\vdash \; E[t] \; R \; E[t']}$$

Essentially, window rules embody monotonicity properties of contexts, with respect to the relations involved. The system chooses suitable window rules automatically; the choice is made when the subwindow is opened. The rules are stored anonymously in a database so the user does not have to know names or other details of these rules. The result of closing is the new window

$$R \star E[t']$$

and the window theorem shows the result of the the whole sequence of steps:

$$\vdash \; E[t] \; R \; E[t']$$

We can now store this theorem in a theory file or we can continue and transform $E[t']$ further.

Assumptions. The window inference system keeps tracks of what assumptions can be used in a transformation. Consider opening a subwindow on the right conjunct of the following window:

$$\Rightarrow \; \star \; \mathsf{x}=1 \wedge \mathsf{x}>0$$

The new window has x=1 as an *assumption*:

$$\begin{array}{ll} ! & \mathsf{x}=1 \\ \Rightarrow \; \star & \mathsf{x}>0 \end{array}$$

The assumption can be used to transform the focus to `T` (truth). After this we can close and rewrite; this transforms the original focus to x=1.

Extensibility. The Window Inference library lets the user add a new relation R into the system, once R has been proved reflexive and transitive. An example is the refinement relation on programs which is used in the example in Section 6. The user can also add window rules. In the refinement example, we have added one rule for each argument of each statement constructor (e.g., one rule for closing after opening a subwindow on the left component of a sequential composition, one rule for the body of a loop, etc.).

4 TkWinHOL

TkWinHOL is a graphical user interface to the Window Inference Library. It runs on UNIX based systems under X-windows. It requires the Tcl/Tk programming system and the Expect [14] extension to Tcl/Tk. The HOL environment supported is HOL90.

TkWinHOL is designed to simplify the work a user has to do while transforming an initial focus towards the desired goal. One of the main features of TkWinHOL is that it gives the user the possibility to use the mouse to point out subterms that need to be operated on. The user is spared from the task of typing in explicit path information which can be tedious if operating on a large focus. The system has other interesting features as well (see below) and we have tried to build in as much graphical support as possible.

Using Tcl/Tk for implementing TkWinHOL makes the system small yet powerful. It also makes the system 'open' in the sense that it can easily be modified and extended. Indeed, one of the basic ideas of TkWinHOL is that it should be possible to use as a basic building block for the more specialised system discussed in Section 6. Tcl/Tk has turned out to be a good choice also because there is work going on using Tcl/Tk for interfacing to other aspects of the HOL system that we have not considered (for example theorem retrieval) [18], and there is a great potential for cooperation.

4.1 The basic appearance of TkWinHOL

When TkWinHOL is started three windows appear on the screen. First there is a simple text editor window that offers a small set of menu commands for file and edit operations. The text editor window also offers a subset of the ordinary Emacs key sequences for moving the insertion point etc. However, the user does not have to use this editor (this is explained below).

Executing HOL commands. The second window (the HOL window) is similar to a terminal window (like Xterm). Commands can be typed directly into the HOL window and executed. Alternatively, users can execute any HOL command they wish by selecting (using the mouse) a text string containing the command and pressing an execute button that is associated with the HOL window. The text string is sent verbatim to the HOL process running in the background and the reply given by HOL is presented in the HOL window. The selected text can be

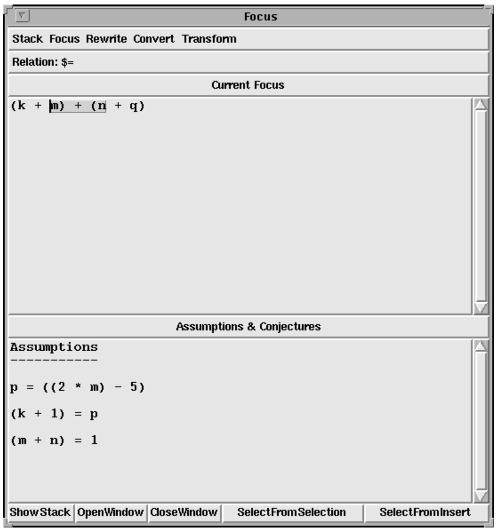

Fig. 1. A sample view of the focus window of TkWinHOL

---

in the text editor window but it can also be in a separate window not opened by the TkWinHOL application. This window can, e.g., be running Emacs provided the Emacs version used exports the selection.

Window inference in TkWinHOL. The third window is the heart of the application. This window (called the focus window) displays the status of the current window stack. As can be seen in Figure 1 the window is divided into parts. At the top of the window there is a menubar from which the user can select operations that transform the appearance of the the current focus. Below the menubar there is a field that presents the relation that is currently begin preserved, (in Figure 1 this relation is equality). The top subwindow presents a pretty printed version of the current focus. The pretty printer associated with the environment supplies not only the pretty-printed text, but also information about the paths to subterms and also the starting and ending positions on the screen of subterms. Using this information, TkWinHOL can access subterms. Without an interface, manipulating a subterm would mean typing the operation to be performed and supplying the path to the subterm in question, which can be a tedious task. In TkWinHOL, double-clicking the right mouse button at some position in the term will highlight the smallest enclosing subterm. After this, the user can select the desired operation from the menus (or a button in some cases); no path information is required. A typical operation is opening a window. The user selects, by clicking, the subterm to open a window on and then press the button labeled OpenWindow (which can be seen at the bottom of the focus window in Figure 1).

TkWinHOL then sends the corresponding command string to HOL and waits for a reply (which is either new pretty-printing information or an error message).

To illustrate how this works we can again refer to Figure 1. If the user clicks the `k` in the focus then only the `k` is selected (and highlighted). This is because `k` is a subterm on which it would make sense to open a subwindow. If the user clicks the leftmost `+` in the focus, then the selected subterm is `(k + m)`. Similarly, if the user clicks the middle `+` the whole focus is selected.

Handling associative operators. In a sense TkWinHOL is, as for example chol [19], built around the structure of terms. However, this is not the only way to operate on the focus. Often there are situations where the structure of the focus is such that one cannot open a window on the part of the focus that one wants to, without modifying the focus first. Typically, such situations occur with associative operators where the standard association may not be what one wants. To simplify the user's work in such situations TkWinHOL provides a way to select (by dragging, not clicking) a segment of the string representing the focus, and then opening a window on that selection although it may not be accessible using a path in the ordinary sense.

This is made possible by a routine that automatically reassociates the focus according to what the user wants and then opens a window on the subterm the user has selected. There are restrictions on the use of this feature; the text segment selected by dragging has to be of the correct form[1] in order for reassociation to work.

We again refer to Figure 1 to illustrate how this works. As seen from the figure, the segment that has been selected (by dragging) is `m) + (n`. This is not a subterm in itself and cannot be selected by double-clicking as described above, but it is still possible to press the OpenWindow button which will result in a new window with the focus `m + n`.

Working with assumptions. The subwindow at the bottom of the focus window contains the assumptions (assumptions, conjectures and lemmas in window inference jargon) of the current stack. Operations that work on the assumptions are supported by TkWinHOL; users can click on the term that they want to operate on after which the term is highlighted. This term can then be used (as a theorem) as an argument to any command chosen from a menu. This saves the user from the trouble of typing the term (and additional type information).

Supplying parameters to commands. In the operations described above, TkWinHOL constructs HOL commands that use only information that is present by default in the focus window. Sometimes, however, the user has to supply additional parameters to some menu command chosen (e.g. the command REWRITE_WIN in the Rewrite menu could be passed a list of names of theorems to be used for the

[1] This means that the selection must cover two subterms (having the same type) separated by an associative operator (disregarding parentheses denoting current association)

rewrite operation). Such parameters are passed via dialogue boxes or by selecting a string for example in the text editor window. Strings can also be selected in an emacs window if the user prefers to work in such a way. It is also always possible to modify the current window stack by issuing commands directly to HOL as described above. In this case the result of the issued command appears in the HOL window. The state of the focus window can be updated by pressing the ShowStack button.

Proof scripts. The commands that are issued via the focus window while transforming a given stack are recorded as a script. This is a simple process which records every command sent to HOL, separating them with semicolons. The complications that arise when trying to store a proof script from a session with the subgoal package (e.g. lists of tacticals to handle subgoals) do not exist in window inference.

For the moment being, the system does not record commands that change the current stack but are issued directly via the HOL window. This means that in order to record the whole proof process, all commands that actually operate on the stack must be issued via the focus window. Likewise, small lemmas etc. that are needed in the proof process have to be constructed via the focus window.

4.2 Implementing TkWinHOL

As mentioned in previous sections, the main motivation for implementing TkWinHOL is to use it as a building block for something larger; an environment supporting formal program derivation in the Refinement Calculus framework. This means that the interface layer must be reasonably small and efficient. Tcl/Tk applications are interpreted but that does not generally affect their efficiency substantially. In fact, the overhead produced by adding TkWinHOL on top of HOL90 is negligible. Tcl/Tk applications are also easy to extend since the language is easy to learn and additions are simply loaded into the application, so no recompilation or linking is needed.

To connect the interface to HOL we use an extension to the Tcl/Tk system called Expect. This extension provides a set of functions lacking from Tcl/Tk, like job control over several processes running in the background simultaneously, and communicating with standard command line based applications like HOL. Using Expect makes it easy to run, for example, a computer algebra system simultaneously with HOL under the same interface. There are situations where this possibility could prove useful [13].

TkWinHOL relies on a special purpose pretty printer implemented in SML. Besides pretty printing terms this pretty printer also supplies information about the subterms of the pretty printed term. This information is needed to implement the selection of subterms by clicking the mouse.

4.3 Customising TkWinHOL

When customising TkWinHOL for some specific HOL theory, there are basically three things one might want to do: modify the basic pretty printer supplied with

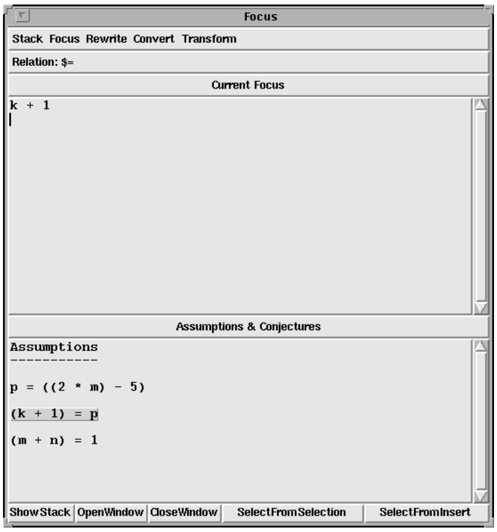

Fig. 2. An other sample view of the focus window of TkWinHOL

the environment, produce a parser suited for the theory used and add new choices to the menus or even entire new menus. A parser is useful in the situations where one has to enter terms as arguments to menu choices.

As mentioned above the format of the the pretty printer is fairly simple. It has to supply the TkWinHOL interface with information about the start and end point of subterms, using a specific row + column format and path information, in addition to the actual pretty printing. In general, it is not necessary to replace the basic pretty printer. It is only in the case that one wants the output to look 'nice' that a new pretty printer is needed. A parser is also needed only in case one is not satisfied with the 'HOL-syntax' of the theory one uses.

Adding new choices in the menus or altogether new menus to the menubar is done using Tcl/Tk programming.

5 Using TkWinHOL – an example

We shall now illustrate how TkWinHOL supports transformations in a simple and intuitively appealing way. In particular we want to show features dealing with selection and regrouping of expressions. Selection is always essential and automatic regrouping substantially simplifies derivations that handle associative operators.

Suppose that in a derivation we have opened a subwindow with the current focus and assumption list seen in the Figure 1. While preserving equality the focus

```
(k + m) + (n + q)
```

can be simplified to

```
p + q
```

just by exploiting context information present in the assumption subwindow (the lower subwindow of the focus window). In Figure 1, the only applicable assumption `(m + n) = 1` can not be used directly since association of the variables `m` and `n` in the focus is different from the one in the assumption. Using window inference without an interface, one would need to rewrite the focus twice using the associativity theorem in order to get an appropriate association. Here, however, we can proceed just by dragging the pointer and highlighting the desired expression (see Figure 1). Pressing the OpenWindow button then automatically rearranges the current focus before opening and as a result the new focus is `m + n`. Now we select (by double-clicking) the first assumption `(m + n) = 1` in the assumption list and use the REWRITE_WIN menu entry. TkWinHOL then rewrites the focus with this assumption. Closing the window with the button CloseWindow gives us the new focus

```
k + (1 + q)
```

Once again, by dragging the pointer we can highlight `k + 1`, then regroup the whole expression (with the OpenWindow button) and focus on

```
k + 1
```

Now double-clicking on the second assumption `(k + 1) = p` selects it for the next rewriting (see Figure 2). Note that in this way we can easily choose the assumptions to be used in a transformation and avoid unnecessary (and possibly looping) rewriting which occurs when directly using functions from the Window Inference Library such as ASM_REWRITE_WIN (in our example we prevented variable `p` from being rewritten using the first assumption `p = ((2 * m) - 5)`).

Having closed the window we achieved our goal and transformed the initial expression into `p + q`. This fact is expressed by the window theorem:

```
|- (k + m) + (n + q) = p + q
```

which shows the result of the part of the derivation described here.

6 The Refinement Calculator

The Refinement Calculator, a customized version of TkWinHOL, is a tool for program refinement within the framework of the Refinement Calculus. The language used to express programs and specifications is essentially Dijkstra's language of guarded commands, with some extensions. In the Refinement Calculus, assertions can be used as statements, to indicate facts that are known to hold at certain points in the program text. There is also a nondeterministic assignment, which is used for specification purposes. Furthermore, blocks with local variables are included.

The Refinement Calculator extends TkWinHOL with a number of menu alternatives (in the Transform menu) for refinement transformations. For each transformation, HOL proves the correctness using the underlying theory of refinement. The example below illustrates a few basic transformations. More advanced transformations are also available, such as introduction of blocks with local variables, recursion introduction and data refinement.

The Refinement Calculator contains a parser (written by Michael Butler) which parses program statements entered by the user into the abstract syntax of the HOL system. It also contains an extension of the TkWinHOL pretty-printer to handle program statements.

6.1 How programs are handled

The HOL theory of refinement is a *shallow embedding*, meaning that the theory manipulates semantic objects, rather than the actual syntax of the Refinement Calculus. The semantics used is a *weakest precondition semantics*, where every program statement denotes a predicate transformer. A predicate transformer is a function of type `pred→pred` from predicates to predicates, where a predicate is a function of type `state→bool`. The type of states is in the general theory a polymorphic type. For any given program, this type is instantiated to a product with one component for each program variable. If, for example, a program works on two natural number variables `x` and `y` and a boolean variable `b`, then the state space has type `num×num×bool`. The names of program variables are handled using a `let`-construction; in this case the term representing the program is of the form

```
let x = FST in let y = FST o SND in let b = SND o SND in ...
```

Thus program variables are *projection functions*; they indicate positions in the state tuple.

One main reason for using a shallow embedding is that this permits reuse of HOL data types in programs. When a program variable ranges over the natural numbers, we can use the HOL theory of natural numbers directly when doing transformations (e.g., to simplify a subexpression of a statement). Programs can also work on lists, sets, real numbers etc, relying on built-in theories and libraries. If a user adds a new data type to HOL, then that data type can immediately be used in programs. If we had decided on a deep embedding rather than a shallow one, then we would have had to embed the type structure in the theory of refinement. This would have fixed the available data types for program variables, making it impossible to introduce new types later without extending and rebuilding the whole HOL theory.

Even though the HOL theory only manipulates semantic objects, the TkWinHOL interface always presents programs and statements according the the syntax of the extended guarded command language. The correct syntactic form can always be computed from the semantic object and the list of current program variables. This list is given by the user for the initial specification from which a derivation starts, and it is then updated every time local variables are added.

Consider the following subset of the statement syntax:

$$S ::= \texttt{skip} \mid \{\texttt{b}\} \mid \texttt{x} := \texttt{e} \mid S_1; S_2 | \ldots$$

where `b` is a boolean term, `x` is a program variable and `e` is an expression of the same type as `x`. Given a tuple of current program variables `u` (containing `x`), each of these statements denotes a predicate transformer, as follows. The `skip` statement denotes the `skip` predicate transformer. The assertion {`b`} denotes `assert(λu.b)`, the assignment `x:=e` denotes `assign(λu.e)`, and the denotation of the sequential composition is the result of applying the operator `seq` to the denotations of S_1 and S_2.

The definitions in higher order logic of the semantic operators are as follows:

```
skip q = q
assert p q = λs. p s ∧ q s
assign f q = λs. q(f s)
seq S1 S2 q = S1(S2 q)
```

The right hand side shows the predicate which is the result when the predicate transformer in question is applied to the predicate `q`. For details on the weakest precondition semantics and its embedding in the HOL system, see [9, 20, 21].

Every time a transformation is applied to the current (syntactic) focus, the interface sends a command to ML which first performs the transformation on the semantic level (using window inference) and then computes (using the pretty-printer) the syntactic form of new focus. This new focus is returned to the interface and presented on the screen.

6.2 A small refinement example

We illustrate the Refinement Calculator by showing a derivation of a program that computes a factorial. We begin a program derivation by entering the initial specification

```
"program ex var x,y:num . {y=N}; x,y:=x',y'. x' = FACT N"
```

A special menu alternative for beginning derivations sends the string to a parser for the Refinement Calculus notation. The Window Inference system then begins a new stack and returns the program term which is pretty-printed in the focus window. The specification has name `ex`, its state has type `num×num`, the program variables are `x` and `y` and the specification contains an assertion and a nondeterministic assignment. The assertion gives the precondition `y=N` and the nondeterministic assignment says the final value of `x` is `N!` while the final value of `y` is not constrained. Here `N` is a specification constant, not a program variable (in fact, an initial specification need not mention any specification constants; they can be introduced during the derivation).

Our first transformation step is to add an initialisation of `x`. We open a subwindow (by pointing and clicking) on the nondeterministic assignment and select the refinement-specific menu alternative ADD ASSIGNMENT with argument `"x:=1"`. After this, we close and then use a special menu alternative PUSH ASSERT to propagate the assertion through the assignment. The focus is then the following:

```
{ y = N };
x := 1;
{ (x = 1) /\ (y = N) };
x,y := x',y' . x' = (FACT N)
```

So far, the correctness of each refinement step has been proved automatically by the Refinement Calculator (i.e., by HOL in the background). The rule for adding an initialisation states that

`x,y:=x',y'. b` *is refined by* `y:=e; x,y:=x',y'. b`
provided `b` *does not mention* `y`.

The rule for pushing an assertion essentially says that

`{p};S` *is refined by* `S;{sp(S,p)}`

where `sp(S,p)` is the *strongest postcondition* of `S` with respect to precondition `p`.

Now we want to replace the combination of an assertion and a nondeterministic assignment with a do-loop. We open a subwindow on this statement pair and select a LOOP INTRODUCTION menu alternative from the Transform menu and give the arguments that the interface prompts for:

guard:	`"0<y"`
body:	`"x,y:=x*y,y-1"`
invariant:	`"FACT N = x*(FACT y)"`
termination function:	`"y"`

Figure 3 shows what the focus window looks like when the menu alternative for loop introduction is selected.

The Refinement Calculator replaces the focus with the statement

```
do 0<y -> x,y := x*y,y-1 od
```

and adds three proof obligations for the refinement. These proof obligations say that the invariant is true initially, that the invariant is preserved (and the termination function is decreased) and that the postcondition is established when the loop terminates. They can either be discharged (i.e., proved) immediately or later, also using the Refinement Calculator (there is a special menu alternative for setting up such a proof obligation as a focus, with the aim of transforming

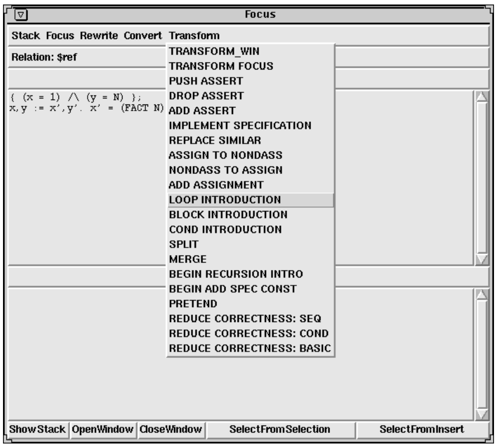

Fig. 3. The Refinement Calculator

it to truth while preserving backward implication). In this simple example, one is proved automatically, one requires only trivial arithmetic and one is a simple total correctness assertion which can be proved using basic properties of the factorial function.

Finally, we can close and use a special menu alternative to drop assertions from the program text (again, correctness proofs of the transformations are automatic). This leaves us with the final program for computing a factorial

```
x := 1;
do 0<y -> x,y := x*y, y-1 od
```

Once we have discharged the proof obligations, we can store the current window theorem, which states that the final program is in fact a correct refinement of the original specification.

7 Conclusions

We have described a graphical interface to the HOL Window Inference library. The main inspiration for developing it was to provide a user friendly tool for transformational reasoning. Using the interface, the user can indicate what transformations should be applied and what subterms they should be applied to. One important aim was to automate certain simple but nevertheless tedious steps needed in window inference, such as selecting a subexpression or regrouping an expression into appropriate form. We have provided facilities which allow these steps to be carried out in a simple way. An expression can be selected by pointing to an appropriate part of it and clicking a mouse button. A part of an expression can be regrouped using associativity properties of the operators just by selecting the segment of the text that one is interested in.

This simple way to select 'interesting' subexpressions in the focus window allows easy selection of the objects that are to be manipulated in some transformation (for instance, selecting an assumption to be used in rewriting). In the future, a multiple selection facility (allowing discontinuous selections) will make this feature even more powerful, e.g. when regrouping expressions that contain operators that are commutative and associative. Note that the user neither needs to enter HOL terms nor supply any extra information (types, numbers of assumptions, etc.) concerning the objects of interest.

We have also provided an example of how TkWinHOL can be used for more specific purposes: it serves as a basis for an environment supporting formal program derivation within the Refinement Calculus framework.

The choice of the Tcl/Tk programming system (with the Expect extension) for implementing the tool turned out to be a good one. It made our interface customisable and easily extensible with new functionality and new menu choices. The Refinement Calculator is a good illustration of this.

Acknowledgements

We wish to thank Michael Butler who implemented the parser for the Refinement Calculator, as well as Jim Grundy who implemented the Window Inference Library and also commented on a draft version of this paper.

References

1. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, Lyngby, 1992.
2. Flemming Andersen, Kim Dam Petersen, and Jimmi S. Petterson. A Graphical Tool for Proving UNITY Progress. In T. F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications – 7th International Workshop. Valletta, Malta, September 1994*, volume 859 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
3. R.J.R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts.* Mathematical Centre, Amsterdam, 1980.
4. R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
5. R. J. R. Back, J. Hekanaho and K. Sere. Centipede — a Program Refinement Environment, Reports on Computer Science and Mathematics, Ser. A 139, Åbo Akademi University, 1992.
6. R.J.R. Back and J. von Wright. Refinement calculus, part I: Sequential programs. In *REX Workshop for Refinement of Distributed Systems,* volume 430 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, 1989. Springer–Verlag.
7. R. J. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.

8. A. Camillieri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.
9. E.W. Dijkstra *A Discipline of Programming*. Prentice–Hall, 1976.
10. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.
11. J. Grundy. A window inference tool for refinement. In Jones et al, editor, *Proc. 5th Refinement Workshop*, London, Jan. 1992. Springer–Verlag.
12. J. Grundy. HOL90 window library manual. 1994.
13. John Harrison and Laurent Théry. Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications – 6th International Workshop, HUG '93 Vancouver, B. C., Canada, August 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184. Springer Verlag, 1993.
14. Don Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, 1994.
15. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison–Wesley, 1994.
16. Thomas W. Reps and Tim Teitelbaum, editors. *The Synthesizer Generator. A System for Constructing Language-Based Editors*. Springer-Verlag, 1988.
17. P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. Techn. Rep. 138, Key Centre for Software Technology, University of Queensland, Australia, 1990.
18. Donald Syme. A New Interface for HOL - Ideas, Issues and Implementation. submitted to the HUG95.
19. Laurent Théry. A Proof Development System for the HOL Theorem Prover. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications – 6th International Workshop, HUG '93 Vancouver, B. C., Canada, August 1993*, volume 780 of *Lecture Notes in Computer Science*, pages 115–128. Springer Verlag, 1993.
20. J. von Wright, J. Hekanaho, P. Luostarinen and T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in Systems Design*, 3:49–81, 1993.
21. J. von Wright. Program refinement by theorem prover. In *BCS FACS Sixth Refinement Workshop – Theory and Practise of Formal Software Development. 5th – 7th January, City University, London, UK.*, 1994.