The Refinement Calculator: Proof Support for Program Refinement

Michael Butler,[1] Jim Grundy,[2] Thomas Långbacka,[3,4]
Rimvydas Rukšėnas,[5,4] and Joakim von Wright[5,4]

[1] Dept. of Electronics & Computer Science, University of Southampton
[2] Dept. of Computer Science, The Australian National University
[3] Dept. of Computer Science, University of Helsinki
[4] Turku Centre for Computer Science (TUCS)
[5] Dept. of Computer Science, Åbo Akademi University

Abstract. We describe the Refinement Calculator, a tool which supports the application of the refinement calculus to program development. The tool uses a general mechanism for transformational reasoning with HOL as the underlying proof system. A graphical user interface provides the user with menus of transformations and the ability to select and focus on subcomponents of a specification using simple mouse operations. The refinement-oriented transformations are illustrated with a case study.

1 Introduction

The *refinement calculus* [2, 20, 22] is a formalisation of the stepwise refinement method of program construction. The required behaviour of a program is specified as an abstract, possibly executable, program which is then refined by a series of correctness-preserving transformations into an efficient, executable program.

When using formalisms like the refinement calculus, the derivations that transform an initial specification to an executable program are usually long and error-prone. Even small refinement steps typically generate large, and therefore difficult to manage, proof conditions. In order to make the task manageable in practice, tools are needed. In this paper, we present a tool called the *Refinement Calculator* which we have developed to support the application of refinement transformations and the proof of verification conditions introduced by individual refinement steps.

Our aim has been to develop a tool that is both user-friendly and reliable. User-friendliness is hard to measure, but it typically means providing a suitable graphic user interface (GUI). GUIs often work in a fashion where the user selects (using the mouse) the data to manipulate and then chooses the desired operation from a menu (or the other way around). Typically, applications from different domains are similar in appearance. This is desirable, since it makes moving between application domains easy. In our tool, proofs are carried out primarily in the way described above, i.e. by selecting information using the mouse, and then applying menu commands to operate on the selected data. Little typing is required of the user, contrary to current practice with most proof tools.

An important aspect of the Refinement Calculator is its reliance on *transformational* reasoning. In transformational reasoning a proof is constructed by transforming an initial expression, via a series of relation preserving steps, to a final expression that has some desired property. Program refinement is a typical example of such a proof activity: refinement being the relation preserved, and executability or efficiency being the property desired. In effect, this paper will describe a mechanisation of a general method of transformational reasoning, including a graphical user interface, and its subsequent specialisation for use with program refinement. An overview of the Refinement Calculator has been presented in an earlier paper [5]. Here we discuss more of the development history of the tool and the rationale for various design decisions as well. We also give a more detailed view of the tool from a user's perspective.

The remainder of the paper is organised as follows: After a general introduction to the refinement calculus in Sect. 2, we discuss some of the design considerations in the development of the tool in Sect. 3. We then give an overview of HOL and window inference in Sect. 4. Sect. 5 describes the graphical user interface and the general support it gives for transformational reasoning, while Sect. 6 describes the refinement specific features of the Refinement Calculator. Sect. 7 illustrates the use of the Refinement Calculator with an example program derivation.

2 Refinement Calculus

Stepwise refinement is a method for developing programs from high-level specifications into efficient implementations. In this approach, the development of a program includes the proof of its correctness. This can be compared with program verification, where the program is first developed, using informal methods, and then checked for desired correctness properties. The refinement calculus is based on the predicate transformer (weakest precondition) calculus of Dijkstra [8]. The calculus was originally developed for the refinement of sequential programs, but was later extended to deal with parallel and distributed programs through the action system formalism [3]. It is a calculus of program transformations that preserve total correctness. If S and S' are statements (program fragments), then the refinement $S \sqsubseteq S'$ holds if and only if S' satisfies every total correctness assertion that S satisfies. Thus, if we have proved the refinement formally, then S' is guaranteed to be a correct implementation of S. The refinement relation $\sqsubseteq$ is a preorder (reflexive and transitive). Furthermore, the ordinary control structures of programming (sequential composition, conditional composition and iteration) are monotonic with respect to refinement. Transitivity implies that we can do refinements stepwise, and monotonicity implies that we can focus on substatements when we do refinement (if the refinement $S \sqsubseteq S'$ holds, then $C[S] \sqsubseteq C[S']$ holds, i.e., we can replace S by S' in any program context $C[\cdot]$). The refinement calculus supports a transformational style of program development; starting from an initial specification S_0, we make a series of refinement steps,

$$S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$$

Each individual refinement step $S_i \sqsubseteq S_{i+1}$ is justified by reference to a *refinement rule*. The application of a refinement rule typically involves three activities:

- selecting the substatement that is to be transformed;
- matching the selected statement to the rule;
- verifying conditions associated with the rule.

Some rules operate on large program components (e.g., high-level rules that replace local variables in a block) while others make only small changes (e.g., rewriting the right hand side of an assignment). Rules are often very general, so parameters must be instantiated before they can be applied. Finally, many rules have side conditions (e.g., restricting free occurrences of a variable or requiring equivalence between two expressions) that need to be checked.

Program development in the refinement calculus is creative work but it also involves tedious proof details. To make program refinement more practical, we need tools that take care of the details. One such kind of tool is a *refinement editor*, which keeps track of applicability and side conditions [11, 28]. A more advanced tool is a *refinement calculator*, where the application of a refinement rule leads to a formal proof of the refinement step in a mechanised logic (i.e., the logic of a theorem proving system). By representing the semantics of the refinement calculus in the logic and proving soundness of the refinement rules, we can get a complete guarantee that the tool permits only valid refinement steps.

3 Design Considerations

In this section we discuss the overall architecture of the Refinement Calculator tool. We also discuss some of the major design decisions that were made while developing the tool.

3.1 A Layered Design

The Refinement Calculator has been built in a number of layers. Fig. 1 illustrates the different layers in the design. The bottom layer is the HOL system, described further in Sect. 4.1. The HOL window Library, described in more detail in Sect. 4.2, forms the next layer.

The Refcalc theory is a formalisation of the refinement calculus in HOL. The development of this theory was originally based only on the 'pure' HOL system (as indicated in Fig. 1). Quite soon it became evident the theory was too difficult to use as intended. Later, when the HOL window Library implemented for the window inference system of transformational reasoning in HOL, the Refcalc theory was quickly adapted for use with it.

Another layer, called TkWinHOL, is a GUI to the HOL window Library. TkWinHOL is a tool in its own right and can be used to develop HOL proofs under window inference. It was, however, designed with the intention of being used as a basis for the Refinement Calculator. Although use of the HOL window

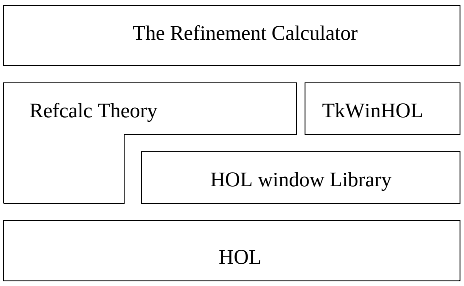

Fig. 1. The layered design of the Refinement Calculator

Library simplified the development of refinement proofs in HOL, it was soon clear that a more sophisticated tool was needed. Certain aspects of working with window inference in HOL are tedious. Fortunately these aspects can be simplified significantly by using a GUI such as TkWinHOL.

Since TkWinHOL was designed with the Refinement Calculator in mind, it was designed to be extendible (more information of how the tool can be extended is given in Sect. 5.3). For example, one can add new menu choices and bind these to HOL commands. This is largely what the Refinement Calculator does, i.e. it adds a number of menu choices that are bound to HOL commands that have been defined in the Refcalc theory. This is indicated in Fig. 1 by placing the Refinement Calculator layer on top of both the Refcalc and TkWinHOL layers. Actually the layered design stretches beyond this as well. One can add further layers above the refinement calculator by defining new menus and binding them to new HOL commands etc. The *data refinement* support discussed in Sect. 6.4 is an example of such an additional layer.

3.2 Design Choices

While developing the Refinement Calculator we had to make a number of choices concerning how to proceed. Below we summarise the major design choices we were faced with:

– **To use an existing theorem prover or build our own:** The choice here was whether or not to use an existing theorem prover, and if so, which one. We chose to use an existing system to avoid the obvious difficulty of building a sufficiently powerful theorem prover of our own, particularly one with the high degree of reliability deemed necessary for the project to have been worth while.
  Among existing theorem provers, HOL was selected for its reliability and flexibility, but primarily because earlier work by members of the team had placed us in the possession of a HOL formalisation of the refinement calculus. Sect. 4.1 gives a more detailed description of the HOL system itself, further

illustrating both why HOL was appropriate for this project, and why HOL was chosen for the earlier formalisation of the refinement calculus.

- **To support any HOL inference style or just window inference:** In contrast to its many virtues, the HOL system has a justified reputation for being difficult to learn. However, our aim with the Refinement Calculator was to build a tool that was easy to use. We therefore chose to avoid the full generality of supporting the various HOL inference mechanisms, and to concentrate on supporting just window inference, because a great deal of reasoning can be accomplished with window inference simply by selecting subterms of an expression and applying one of a relatively small set of transformations. These actions are handled naturally in a GUI by selection with the mouse and choosing options from a menu.

  Furthermore, the structure imposed on proofs by this restricted discipline of transformation results in proofs that are easy to browse. The recording and browsing of proofs is described in Sect. 5.2.
- **Choosing an interface building tool:** We began by examining an earlier window inference based GUI for HOL built using the Centaur tool [26], and then by experimenting with other interface tools including the Cornell Synthesiser Generator [24] and Tcl/Tk [23]. In the end, we adopted Tcl/Tk as the implementation vehicle for our own graphical interface.

  Tcl/Tk consists of a general purpose scripting language (Tcl), together with a powerful set of widgets (Tk). The main strengths of Tcl/Tk are its light weight, generality, and ease of use. Unlike the other tools examined, Tcl/Tk offers no particular enhancements for manipulating syntax trees. This lack of specialisation, however, proved in some ways to be a blessing. For example, it was relatively easy for us to implement a freely re-associating selection mechanism (described in Sect. 5.1), which necessarily ignores the syntax tree of the underlying HOL term.

  The Expect [18] library for Tcl is used to manage the communication between HOL and the user interface. At present we use Expect only for coupling the interface and the theorem prover, but this feature may prove useful in future versions of the refinement calculator for interfacing the system with external oracles. The use of oracles to guide HOL proofs has already been demonstrated by a system integrating HOL and a computer algebra system [16].

4 HOL and Window Inference

The previous section described the overall design of the Refinement Calculator. The use of the HOL theorem proving system [9] and its accompanying window Library [12] form a major part of that design. Together they make up the underlying engine for the formal manipulation of expressions. In this section we describe HOL and the window Library in more detail, giving a more complete rational for their choice as components of the Refinement Calculator.

4.1 The HOL System

The HOL theorem prover has a number of features that distinguish it from competing systems and make it particularly well suited for use as a component in a tool like the Refinement Calculator. One of these is the architecture of the HOL system. For reasons explained below, this architecture allows us to place a particularly high degree of trust in tools built using HOL. We felt that the effort required to build the Refinement Calculator would only be justified if it was possible to confidently trust the results proved with it.

HOL is built using the *LCF architecture* for theorem provers, named for LCF [10], the first system of this kind. Tools with the LCF architecture are built around a small, trusted core implementing the abstract data-type of theorems in the logic of the tool. The axioms of the logic are constants of the data-type, while the inference rules are represented by functions that return elements of the data-type. Modus ponens, for example, would be implemented as a function taking two theorems of the form 'P' and '$P \Rightarrow Q$', and returning a theorem of the form 'Q'. The data-type may also include functions to make new theorems that extend the logic, for example by defining new constants. No other ways of creating a theorem are included in the signature of the data-type. The remainder of the system is built around this core. The architecture ensures our freedom to further extend the system with features needed for the Refinement Calculator without jeopardising its soundness. Any errors we might make can result only in a failure to produce a theorem, or in the production of a theorem other than the one intended, they can not result in the production of a 'nontheorem'.

Another relevant strength of HOL was the existing experience of embedding languages in the system. This gave us sources of both inspiration and comparison. The two common approaches to embedding languages in HOL are known as 'deep embedding' and 'shallow embedding'. In a *deep embedding*, the syntax of the language to be embedded is used to define a type of terms in that language. A function is then defined to map terms to their meaning. Deep embeddings offer opportunities to prove abstract and theoretical properties of a language that shallow embeddings do not. In a *shallow embedding* no new type of terms of the language to be embedded is defined. Instead, terms in the embedded language are identified with terms in the logic by extra-logical parsing and pretty-printing functions. We chose a shallow embedding for the practical reason that it allows us to identify types in the embedded language with types in the HOL logic. This means we can make the programming language of our refinement system strongly typed by inheriting the HOL type system rather than constructing our own. It also means that we can reuse existing HOL theories describing numbers, arrays, lists, and other data-types for the types of variables in our language.

The logic of the HOL system (Church's simple theory of types [7]) makes an excellent choice for formalising program refinement for several reasons. Firstly, the predicate transformer semantics commonly associated with program refinement is naturally modelled in higher-order logic. Predicate transformers are functions from predicates to predicates. It is possible to define such things in higher-order logics, but not in first-order logics. As a result, we are able to safely define the predicate transformer semantics of our target programming language as a definitional extension of higher-order logic; while similar tools based on first order logic, like PRT [6], need extend their logic with new axioms to achieve the same effect. Secondly, to avoid limiting our tool to trivial applications, we needed a logic with sufficient abstraction mechanisms to specify both complex refinement problems and the complex data-types needed to solve them. Part of our motivation for selecting the HOL system was that the power of the abstraction mechanisms of its logic had already been well demonstrated in the field of hardware verification [19]. Temporarily setting aside our already noted preference for a higher-order (and hence typed) logic; we observe that when choosing an expressive logic, a decision must be made between using a set theory or a type theory. The programming language used with the Refinement Calculator is typed, as is the HOL logic, and – as noted before – by opting for a shallow embedding we were able to avoid modelling the types of the programming language by inheriting the types of the logic. If we had chosen a theorem prover based on a set theoretic logic, we would also have had to model the type system of the target programming language within the logic ourselves.

4.2 Window Inference

Window inference is a transformational style of reasoning proposed by Robinson and Staples [25], and later generalised by Grundy [14]. A transformational proof begins with an term E, and proceeds by applying transformations that preserve some desired relationship, R. The proof ends when E has been transformed into another term E' that has some required property. The result is a theorem of the form $\vdash E \; R \; E'$. It is often the case with such proofs that the form of the solution E' is not known at the outset, but is discovered as part of the proof. Program refinement, for example, is an instance of transformational reasoning where E is a specification, R is refinement, and the property that E' must have is executability.

In contrast to this, most mechanised proof assistants support a style of reasoning that is best described as *goal directed*. In such systems a solution is proposed at the beginning of the proof, and the proof is a check that the proposed solution is valid. Although logical variables can be used with some systems to avoid the need to state the solution at the outset, the transformational approach of window inference would seem to be a closer fit to the program refinement process we want to support. As with most theorem provers, the usual interface to HOL is goal directed. However, the LCF architecture of HOL has allowed the implementation of an additional window inference based interface in the form of the HOL window Library [12]. Furthermore, the window Library is designed to be extended with databases of rules to support reasoning in various problem domains. This facility has been exploited to provide direct support for reasoning in the refinement calculus.

The distinguishing feature of window inference is that it allows users to transform a term by restricting attention to a subterm and transforming it. The remainder of the term is left unchanged. While transforming a subterm, it is possible to make assumptions based on its context. For example, if we want to transform the term $A \wedge B$; this may be done by transforming the subterm A under the assumption of B. We may assume B, because if it were false the entire term would be false regardless of A. It is also possible to select and transform assumptions to derive new ones which can then be used in subsequent transformations.The HOL implementation of window inference also allows the temporary conjecture of additional assumptions. If used, these *conjectures* must be proved – by transforming them to true – before the proof as a whole is considered complete.

Reasoning in window inference is conducted with a stack of windows. Each window has a *focus*, F, which is the term to be transformed; a set of formulae, Γ, called the *assumptions*, which can be assumed in the context of the window; and a *relation*, R, which is the relation to be preserved between the focus and any term to which it is transformed. (In the HOL implementation of window inference R must be a preorder.) Such a window will be written as follows:

$$\begin{array}{c} \quad !\ \Gamma \\ R * F \end{array}$$

Four kinds of operations are permitted as part of a window inference proof:

1. A proof is begun by creating a window stack of depth one.
2. A transformation can be applied to the focus of the top window on the stack, providing it preserves the relationship associated with the window.
3. A new window can be pushed onto the stack. The focus of the new window must be a subterm of the focus or an assumption of the previous window. The assumptions and relation of the new window depend on the position of the new focus within the previous window. This operation is called *opening* a window.
4. The top window of a stack can be removed. The relationship established between the initial and final focus of that window is used to transform the window underneath. This operation is called *closing* a window.

Each level in the stack stores a theorem, called the *window theorem*, which records the reasoning done with that window.

5 TkWinHOL – A Tool for Transformational Reasoning

One of the design objectives of the Refinement Calculator was that it be easy to use. While the window Library made the stepwise derivation of programs with HOL a possibility, the tedious nature of the textual interface it provided ensured that the refinement of practical programming examples remained as challenging as ever. This section describes TkWinHOL, a graphical user interface for the window Library designed to help solve this problem. In TkWinHOL, work is carried out by selecting some information on the screen and then choosing an operation to perform through a menu selection or button click.

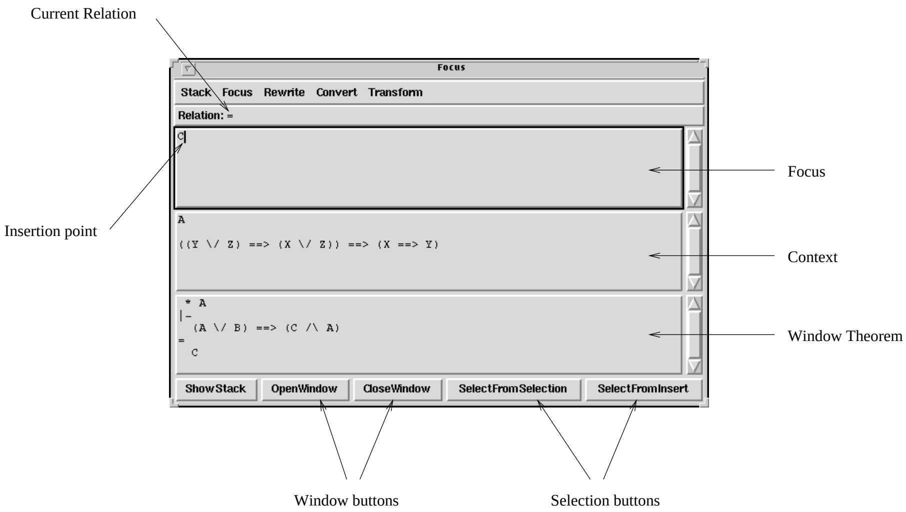

Fig. 2. The appearance of the focus (primary) window of TkWinHOL

5.1 Working with TkWinHOL

Once TkWinHOL is started the user is presented with three windows. Two of these are of less importance, offering a simple editor and a session window for entering commands directly to HOL. The third window (the focus window – see Fig. 2) presents a view of the current state of the active window stack. Thus different types of information (i.e. focus, context, window theorem and the relation preserved) about the current stack are displayed in different regions of the window.

Opening windows With TkWinHOL, a new window is opened on a subterm of the current focus by selecting the required subterm with the mouse, and then pressing the *OpenWindow* button. For further convenience, TkWinHOL can also deal with associativity when selecting subterms. For example, consider the focus

$$(A \wedge B) \wedge C$$

where the gray area denotes a text segment selected with the mouse. Pressing the *OpenWindow* button in this case causes the focus to be automatically re-associated and a new window to be opened with focus $B \wedge C$. These selection features greatly facilitate the use of window inference.

The selection mechanism is implemented by generating extra hidden information about the syntactic structure of the focus during pretty-printing. This information is then used by TkWinHOL to generate paths that access the HOL representation of selected subterms; such paths are required by the underlying HOL window Library.

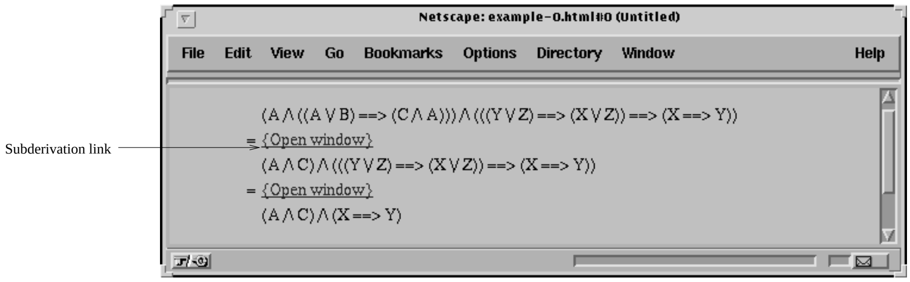

Fig. 3. An example of a browsable proof script

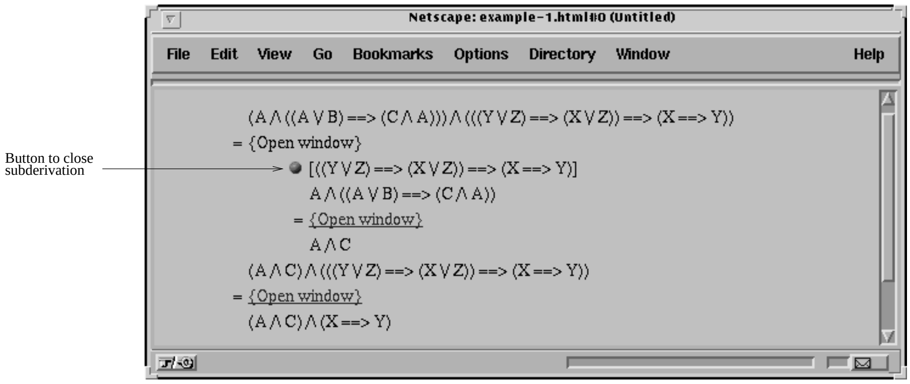

Fig. 4. A browsable proof script with a subderivation link expanded

Executing transformation commands The commands of the HOL window Library are bound to buttons and menus. For many parameters, the information required can be supplied by using the mouse to select relevant information already present on the screen. If that is not enough, information is entered through dialogs. The dialogs are not modal, thus it is possible to use information already present on the screen to fill some of the fields in any given dialog. The kind of information one typically has to fill in is the names of HOL theorems used for rewriting etc.

The use and role of some of the important transformation commands provided by the tool are described in Sects. 6 and 7.

5.2 Proof Visualisation

HOL proofs are not easy to read or understand. Therefore we have created a way of generating and presenting structured scripts that document proofs performed using TkWinHOL. Fig. 3 shows an example of such a proof script. The basic idea is to present proofs in a hierarchical and browsable format [13] (using HTML) so that the reader can focus on the parts of the proof that are of particular interest. For example, consider the logical term

$$\big(A \wedge ((A \vee B) \Rightarrow (C \wedge A))\big) \wedge \big(((Y \vee Z) \Rightarrow (X \vee Z)) \Rightarrow (X \Rightarrow Y)\big) \tag{1}$$

which can easily be shown to be equivalent to the term

$$(A \wedge C) \wedge (X \Rightarrow Y)$$

To prove the equivalence one could start by focussing on the left conjunct of (1) and transform that subterm into $(A \wedge C)$. After that, one could similarly transform the right conjunct of (1) to $(X \Rightarrow Y)$. This is exactly the proof strategy shown in Fig. 3. At the level of abstraction shown in Fig. 3 both the subderivations mentioned above are hidden under subderivation links. Hidden subderivations are created when the user opens a new window, i.e. every time the scope of interest is reduced by the user.

By clicking one of the links, the proof script is expanded to contain the subderivation chosen. Fig. 4 shows the appearance of the proof script once the upper subderivation of Fig. 3 has been expanded. Clicking the button indicated in Fig. 4 recreates the state of Fig. 3. Alternatively, clicking one of the subderivation links in Fig. 4 would expand the script even further.

The main use of proof scripts such as these is to record proofs in a readable and easily distributable (through the WWW) format for pedagogic purposes. A detailed description of the proof scripting mechanism is available as a TUCS technical report [15].

5.3 Extending TkWinHOL

Our original goal was to build a tool supporting mechanically verified program derivation. Thus, emphasis has been placed on making TkWinHOL easy to customise for the needs of particular HOL theories (like those for program refinement [29] or lattice theory [17]). In such specific theories one usually wants to use a higher-level notation (abstract syntax) for the interaction with HOL. This can be achieved in TkWinHOL by adding a theory specific parser and pretty printer. For example, when developing refinement proofs it is more convenient to use a programming language syntax (described in the next section) rather than the relatively clumsy syntax defined by the underlying HOL theory.

Users of TkWinHOL can further customise the tool by adding theory specific menu choices bound to specialised HOL commands. Theory specific menu choices, and parsers and pretty printers, can be packaged together with the theories they support in a system of loadable libraries called ‘extensions’. The Refinement Calculator is an example of such a TkWinHOL extension.

6 The Refinement Calculator

6.1 Programming Notation

The Refinement Calculator is an extension of TkWinHOL and supports a customised program notation with the following syntax:

$$
\begin{array}{rl}
\mathit{Prog} ::= & \textbf{program}\ \mathit{Name}\ \textbf{var}\ v : \mathit{Type} \cdot \mathit{Com} \\
\mathit{Com} ::= & \mathit{Com};\ \mathit{Com} \\
\mid & \{\mathit{BTerm}\} \\
\mid & v := \mathit{Term} \\
\mid & v := v' \bullet \mathit{BTerm} \\
\mid & \textbf{if}\ \mathit{BTerm}\ \textbf{then}\ \mathit{Com}\ \textbf{else}\ \mathit{Com}\ \textbf{fi} \\
\mid & \textbf{do}\ \mathit{BTerm} \rightarrow \mathit{Com}\ \textbf{od} \\
\mid & |[\ \textbf{var}\ v : \mathit{Type} \cdot \mathit{Com}\ ]|
\end{array}
$$

Here, v represents a (list of) variable name(s), *Type* represents a corresponding (list of) HOL type(s), *Term* represents any HOL term, and *BTerm* represents any boolean-valued HOL term. Because the tool uses a shallow embedding of the refinement calculus in HOL, the types and terms can be any such objects known to the HOL system when the derivation is being performed. In this way, the standard HOL notation for types and terms is embedded in the program-specific notation.

The programming notation is basically Dijkstra's guarded commands language extended with *assertions* and *nondeterministic assignment*. Occurrence of an assertion, $\{p\}$, in a program allows us to assume that p holds at that point in the program. Thus assertions can be used to carry context information in programs. A nondeterministic assignment, $x := x' \bullet p$, specifies that x is assigned some value x' satisfying predicate p. The final construct, $|[\ \textbf{var}\ v : \mathit{Type} \cdot \mathit{Com}\ ]|$, represents a block with local variable v.

6.2 Window Opening Rules for Refinement

A program may be specified with a statement $\{\mathit{pre}\}; v := v' \bullet \mathit{post}$. Derivation of a program satisfying this involves transforming the specification while preserving the refinement relation. Monotonicity of the program constructs allows us to focus on subprograms as, for example, the following window opening rules for sequential composition show:

$$
\frac{\vdash S \ \sqsubseteq\ S'}{\vdash S;T \ \sqsubseteq\ S';T}\ R1 \qquad\qquad \frac{\vdash T \ \sqsubseteq\ T'}{\vdash S;T \ \sqsubseteq\ S;T'}\ R2
$$

The context information provided by assertions can be used directly when we focus on the right hand side of an assignment or the postcondition of a nondeterministic assignment as shown by the following rules:

$$
\frac{p \vdash e1 = e2}{\vdash \{p\};\ x := e1 \ \sqsubseteq\ \{p\};\ x := e2}\ R3
$$

$$
\frac{p \vdash q1 \Leftarrow q2}{\vdash \{p\}; x := x' \bullet q1 \ \sqsubseteq\ \{p\}; x := x' \bullet q2}\ R4
$$

Cond Introduction:

$$\vdash \; S \;\sqsubseteq\; \textbf{if } G \textbf{ then } S \textbf{ else } S$$

Block Introduction:

$$\vdash \; v := v' \bullet post \;\sqsubseteq\; |[\; \textbf{var } x : T \cdot x := E;\; v, x := v', x' \bullet post \;]|$$

Loop Introduction 1:

$$\frac{\begin{array}{l}\vdash pre \Rightarrow inv \\ \vdash inv \wedge \neg G \Rightarrow post[v' := v]\end{array}}{\begin{array}{l}\vdash \{pre\}; v := v' \bullet post \;\sqsubseteq \\ \quad \textbf{do } G \rightarrow \{inv \wedge G\}; v := v'.inv[v := v'] \wedge E[v := v'] < E \textbf{ od}\end{array}} [v \text{ not free in } post]$$

Loop Introduction 2:

$$\frac{\begin{array}{l}\vdash pre \Rightarrow inv \\ \vdash (inv \wedge G \wedge E = e) \ll Body \gg (inv \wedge E < e) \\ \vdash inv \wedge \neg G \Rightarrow post[v' := v]\end{array}}{\vdash \{pre\}; v := v' \bullet post \;\sqsubseteq\; \textbf{do } G \rightarrow \{inv \wedge G\}; Body \textbf{ od}} [v \text{ not free in } post]$$

Trailing Assignment:

$$\begin{array}{l}\vdash v, x := v', x' \bullet post \wedge x' = E \\ \qquad \sqsubseteq\; v := v' \bullet post;\; x := E[v' := v]\end{array} \quad \begin{bmatrix} x' \text{ not free in } post, \\ v \text{ not free in } E \end{bmatrix}$$

Fig. 5. Refinement Transformations

Notice that, in the first case, the relation to be preserved on the expression is equality while, in the second case, the relation to be preserved on the postcondition is reverse implication.

The refinement calculator includes a complete set of window opening rules for selecting the subexpressions of the programming language defined in Sect. 6.1. The combination of these rules and the general window opening rules for navigating though terms in the HOL logic means that users of the refinement calculator can choose to focus their attention on any subterm of a program they wish to examine.

6.3 Refinement Transformations

The Refinement Calculator provides a menu of transformations for refining programs; some of these are represented as rules in Fig. 5. The refinement transformations are all derived from the HOL semantics of programs and refinement [1, 29, 31]. Some of the rules in Fig. 5 require the user to provide arguments (that possibly may not appear in the current focus) before the transformation is applied; for example, the *Cond Introduction* rule requires a guard G to be supplied while the *Loop Introduction 1* rule requires a guard G, an invariant inv and a variant E. When such a transformation is selected by the user of the tool, a dialogue box appears with input fields for each of the arguments; these must be filled out appropriately by the user before the tool applies the transformation to the focus.

Some of the rules have assumptions and side-conditions. When a rule with assumptions is applied, the assumptions will be instantiated appropriately and added to the current window context as conjectures to be established later. A transformation will fail if its side-conditions are not satisfied (failure of a transformation leaves the current window unchanged.)

Note from Fig. 5 that two *Loop Introduction* rules are used, the difference between them being that *Loop Introduction 1* determines the loop body from the invariant and the variant, while *Loop Introduction 2* requires the loop body to be supplied explicitly by the user. In the tool, both these rules are provided as a single command; if the *Body* field is left empty by the user in the dialogue box that appears, then *Loop Introduction 1* gets applied, otherwise *Loop Introduction 2* gets applied.

A term of the form $pre \ll prog \gg post$, as used in the *Loop Introduction 2* rule, describes a total-correctness assertion stating that *prog* is guaranteed to establish *post* when executed in initial state satisfying *pre*. The Refinement Calculator provides transformations for converting assertions of this form into boolean terms. The calculator provides a single command for propagating context information as assertion statements. This command uses theorems such as the following:

$$\vdash \;\; \mathbf{if}\ G\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi} \;\;\sqsubseteq\;\; \mathbf{if}\ G\ \mathbf{then}\ \{G\}; S\ \mathbf{else}\ \{\neg G\}; T\ \mathbf{fi}$$
$$\vdash \;\; x := E \;\;\sqsubseteq\;\; x := E; \{x = E\} \quad [x \text{ not free in } E]$$

The calculator also provides commands for rewriting the focus with equations selected from the assumptions, for transforming the focus with user-supplied HOL theorems, and for replacing a focus F (while preserving relation R) by a focus F', where F' is supplied by the user, generating the conjecture $F\ R\ F'$.

The role of the rules of Fig. 5 will be seen more clearly in Sect. 7 where they are applied to an example. The full program syntax and list of refinement rules supported by the Refinement Calculator may be found in its manual and tutorial [4].

6.4 Data refinement

Data refinement is a special case of refinement where abstract program variables are replaced with more concrete ones. Typically, 'more concrete' means more easily or efficiently implementable. Since all occurrences of the abstract variables are to be replaced, the transformation affects the entire scope of their declaration. In the Refinement Calculator, the scope of declaration is the block statement. Thus, for the abstract variables a and concrete variables c, data refinement is the following transformation:

$$|[\ \mathbf{var}\ a \cdot S\ ]| \;\sqsubseteq\; |[\ \mathbf{var}\ c \cdot T\ ]|$$

where the abstract statement S refers to the variables v, a, and the concrete statement T to v, c (v represents the state variables that are not data-refined and are thus common to both S and T). An *abstraction relation* R, relating the abstract and concrete variables, is required in order to perform a data refinement transformation. It has been shown [21, 29] that the concrete statement T can be calculated from the abstract statement S and the abstraction relation R. This allows us to define a function $\mathcal{D}_R$ such that

$$|[\ \mathbf{var}\ a \cdot S\ ]|\ \sqsubseteq\ |[\ \mathbf{var}\ c \cdot \mathcal{D}_R(S)\ ]|$$

The function is defined recursively over the structure of program notation as, for example, in the case of sequential composition:

$$\mathcal{D}_R(S_1; S_2)\ \widehat{=}\ \mathcal{D}_R(S_1); \mathcal{D}_R(S_2)$$

For the basic statements like assignment, $\mathcal{D}_R$ gives the corresponding concrete statement as follows (assuming that R is of the form $(\lambda a, c, v.abs)$):

$$\begin{array}{l} \mathcal{D}_R(v, a := v', a' \bullet post)\ \widehat{=} \\ \qquad v, c := v', c' \bullet (\forall a.abs\ \Rightarrow (\exists a'.abs[a, c, v := a', c', v']\ \wedge\ post)) \end{array}$$

In the Refinement Calculator, the *Data-Refinement* command implements the function $\mathcal{D}_R$. Given variables to be replaced, a, new variables, c, and the abstraction relation R, it calculates the new focus, $|[\ \mathbf{var}\ c \cdot T\ ]|$, from the old focus, $|[\ \mathbf{var}\ a \cdot S\ ]|$, and automatically proves the correctness of the refinement, possibly adding conjectures. Note that the transformation need not necessarily replace all variables declared in the block. Furthermore, if the list of variables to be replaced is empty, the rule implements *superposition refinement*, where only new variables are added under the relation R.

7 Example Derivation: Sorting an Array

We present the outline of an example derivation that may be carried out using the Refinement Calculator. The example involves the derivation of a program that sorts an array of numbers.

Arrays may be modelled by defining a new HOL type; assume $(\alpha)array$ is the type of arrays that are polymorphic on α, and that the following functions on arrays have been defined:

$$\begin{array}{l} size : (\alpha)array \to num \\ lookup : (\alpha)array \to num \to \alpha \\ swap : (\alpha)array \to num \to num \to (\alpha)array \end{array}$$

An array a is indexed from 0 to $(size\ a) - 1$. The i^{th} element of a is given by $(lookup\ a\ i)$, while $(swap\ a\ i\ j)$ represents the array a with the values at positions i and j swapped around.

Let *sorted* and *perm* be defined as follows:

$$\begin{array}{l} sorted\ (a : (num)array)\ (r : num \to bool) = \\ \quad (\forall i \cdot (r\ i)\ \Rightarrow\ i < (size\ a))\ \wedge \\ \quad (\forall ij \cdot (r\ i)\ \wedge (r\ j)\ \wedge\ i < j\ \Rightarrow (lookup\ a\ i) \le (lookup\ a\ j)) \\ perm\ (a1 : (num)array)\ (a2 : (num)array) = \\ \quad (size\ a1) = (size\ a2)\ \wedge \\ \quad (\exists f \cdot (injective\ f)\ \wedge \\ \qquad (\forall i \cdot i < (size\ a1)\ \Rightarrow\ (f\ i) < (size\ a1)\ \wedge \\ \qquad\qquad (lookup\ a1\ i) = (lookup\ a2\ (f\ i)))) \end{array}$$

The term $(sorted\ a\ r)$ states that that projection of array a whose indices satisfy r is sorted, e.g., $(sorted\ a\ (\lambda i \cdot 5 \leq i \leq 10))$ says that the array slice 5..10 of a is sorted. The term $(perm\ a1\ a2)$ specifies that array $a2$ is a permutation of array $a1$. The sorting program is then specified as:

$$\begin{array}{l} \textbf{program}\ Sort\ \textbf{var}\ a : (num)array \cdot \\ \quad \{a = a0\};\ a := a' \bullet (sorted\ a'\ (\lambda i \cdot i < (size\ a0)))\ \wedge (perm\ a0\ a') \end{array}$$

Here, $a0$ is a specification constant, not a program variable. This specification (in ASCII form) is provided as input to the tool in order to begin a program derivation; the specification will then appear in the focus region of the tool (see Fig. 2).

Using the mouse to focus on the highlighted body of the specification yields the following window:

$$\begin{array}{l} \sqsubseteq\ *\ \{a = a0\}; \\ \quad\quad a := a' \bullet (sorted\ a'\ (\lambda i \cdot i < (size\ a0)))\ \wedge (perm\ a0\ a') \end{array}$$

We shall implement this using an insertion sort algorithm which requires a pair of nested loops. The body of the outer loop will ascend the array ensuring that the array slice $0..k-1$, where k is the current position, is sorted. Before introducing the loop, we must introduce a local variable k that will be used to ascend the array. This is introduced and initialised by applying the *Block Introduction* transformation to the highlighted statement above, with $x : T$ instantiated by $k : num$ and E instantiated by 0. This results in the focus:

$$\begin{array}{l} \sqsubseteq\ *\ \{a = a0\}; \\ \quad\quad |[\ \textbf{var}\ k : num \cdot k := 0; \\ \quad\quad\quad a, k := a', k' \bullet (sorted\ a'\ (\lambda i \cdot i < (size\ a0)))\ \wedge (perm\ a0\ a') \\ \quad\quad]| \end{array}$$

Now, we execute the command that propagates assertions yielding:

$$\begin{array}{l} \sqsubseteq\ *\ |[\ \textbf{var}\ k : num \cdot k := 0; \\ \quad\quad\quad \{(k = 0)\ \wedge (a = a0)\}; \\ \quad\quad\quad a, k := a', k' \bullet (sorted\ a'\ (\lambda i \cdot i < (size\ a0)))\ \wedge (perm\ a0\ a') \\ \quad\quad]| \end{array}$$

Next, we select the highlighted part of this block using the mouse, open a window on it, and apply the *Loop Introduction* command. This causes a dialogue box to appear with fields for loop guard, body, invariant and variant. We supply the following values for the guard, invariant and variant and leave the *Body* field blank:

$$\begin{array}{ll} Guard: & k < (size\ a) \\ Invariant: & k \leq (size\ a)\ \wedge (sorted\ a\ (\lambda i \cdot i < k))\ \wedge (perm\ a0\ a) \\ Variant: & (size\ a) - k \end{array}$$

This transformation results in the following window:

$$
\begin{array}{l}
\sqsubseteq\ *\ \mathbf{do}\ k < (size\ a) \rightarrow \\
\qquad \{k < (size\ a)\ \wedge (sorted\ a\ (\lambda i \cdot i < k))\ \wedge (perm\ a0\ a)\}; \\
\qquad a, k := a', k' \bullet (sorted\ a'\ (\lambda i \cdot i < k'))\ \wedge (perm\ a0\ a')\ \wedge \\
\qquad\qquad ((size\ a') - k') < ((size\ a) - k) \\
\quad \mathbf{od}
\end{array}
$$

This step also generates two conjectures which are displayed in the context region of the tool. These state respectively that the invariant should hold initially and the invariant and the negated guard should establish the original postcondition:

$$
\begin{array}{l}
(\forall k, a \cdot (k = 0)\ \wedge (a = a0)\ \Rightarrow \\
\quad k \leq (size\ a)\ \wedge (sorted\ a\ (\lambda i \cdot i < k))\ \wedge (perm\ a0\ a)) \\
(\forall k, a \cdot \neg(k < (size\ a))\ \wedge\ k \leq (size\ a) \\
\qquad \wedge\ (sorted\ a\ (\lambda i \cdot i < k))\ \wedge (perm\ a0\ a) \\
\qquad\qquad \Rightarrow\ (sorted\ a\ (\lambda i \cdot i < (size\ a0)))\ \wedge (perm\ a0\ a))
\end{array}
$$

At this stage in the derivation, we can choose either to attempt to discharge the conjectures[6] or to continue refining the body of the loop. Conjectures such as this may be established by opening windows on the conjectures, with backward implication as the relation to be preserved, then transforming the focus to true.

We proceed with the refinement of the loop body:

$$
\begin{array}{l}
\sqsubseteq\ *\ \{k < (size\ a)\ \wedge (sorted\ a\ (\lambda i \cdot i < k))\ \wedge (perm\ a0\ a)\}; \\
\quad a, k := a', k' \bullet\ (sorted\ a'\ (\lambda i \cdot i < k'))\ \wedge (perm\ a0\ a')\ \wedge \\
\qquad ((size\ a') - k') < ((size\ a) - k)
\end{array}
$$

Focussing on the highlighted term yields:

$$
\begin{array}{l}
\Leftarrow\ *\ (sorted\ a'\ (\lambda i \cdot i < k'))\ \wedge (perm\ a0\ a')\ \wedge \\
\qquad ((size\ a') - k') < ((size\ a) - k)
\end{array}
$$

Notice that the relation to be preserved is reverse implication. We strengthen this term by replacing the highlighted subterm with $k' = k + 1$, yielding the following focus:

$$
\begin{array}{l}
\Leftarrow\ *\ (sorted\ a'\ (\lambda i \cdot i < k'))\ \wedge (perm\ a0\ a')\ \wedge \\
\qquad k' = k + 1
\end{array}
$$

along with the conjecture

$$k' = k + 1\ \Rightarrow\ ((size\ a') - k') < ((size\ a) - k)$$

This conjecture may discharged by opening a window on it and transforming it to true using appropriate arithmetic theorems (this also requires $(perm\ a0\ a)$ and $(perm\ a0\ a')$ which will appear as contextual assumptions). We skip these steps and continue with the main derivation. Focussing on the highlighted subterm of the previous focus yields

[6] If we postpone establishing the conjectures, then what we have is a refinement theorem with the conjectures as assumptions.

$$\Leftarrow * \;(\mathit{sorted}\ a'\ (\lambda i \cdot i < \boxed{k'}\,)) \wedge (\mathit{perm}\ a0\ a')$$

with $k' = k+1$ as a context assumption. Applying the *Rewrite* command to the focus using this assumption yields:

$$\Leftarrow * \;(\mathit{sorted}\ a'\ (\lambda i \cdot i < (k+1))) \wedge (\mathit{perm}\ a0\ a')$$

Closing (twice) gives us:

$$\begin{array}{ll} \sqsubseteq * & \{k < (\mathit{size}\ a) \wedge (\mathit{sorted}\ a\ (\lambda i \cdot i < k)) \wedge (\mathit{perm}\ a0\ a)\}; \\ & a, k := a', k' \bullet (\mathit{sorted}\ a'\ (\lambda i \cdot i < (k+1))) \wedge (\mathit{perm}\ a0\ a') \wedge \\ & \qquad\qquad k' = k+1 \end{array}$$

Applying the *Trailing Assignment* transformation yields:

$$\begin{array}{ll} \sqsubseteq * & \{k < (\mathit{size}\ a) \wedge (\mathit{sorted}\ a\ (\lambda i \cdot i < k)) \wedge (\mathit{perm}\ a0\ a)\}; \\ & a := a' \bullet (\mathit{sorted}\ a'\ (\lambda i \cdot i < (k+1))) \wedge (\mathit{perm}\ a0\ a'); \\ & k := k+1 \end{array}$$

So, under the assumption that the array is sorted between 0 and $k-1$, the highlighted specification must establish that the array is sorted between 0 and k. It will do this by shuffling the element of a at position k down the array to the appropriate position. The following arguments will be used to perform the introduction of the inner loop:

$$\begin{array}{ll} \mathit{Guard}: & l > 0 \wedge (\mathit{lookup}\ a\ l) < (\mathit{lookup}\ a\ (l-1)) \\ \mathit{Body}: & a := (\mathit{swap}\ l\ (l-1)\ a);\ l := l-1 \\ \mathit{Invariant}: & l \leq k \wedge (\mathit{sorted}\ a\ (\lambda i \cdot i < l \vee (i > l \wedge i \leq k))) \wedge \\ & l < k \Rightarrow (\mathit{lookup}\ a\ l) < (\mathit{lookup}\ a\ (l+1)) \wedge \\ & (\mathit{perm}\ a0\ a) \\ \mathit{Variant}: & l \end{array}$$

This invariant states that the array slice $0..k$, with the l^{th} position excluded, is sorted, and that the value at the l^{th} position is less than the value at the $(l+1)^{th}$ position. We focus on the highlighted part of the previous focus and introduce l as a local variable in the same way that k was introduced. We then apply the *Loop Introduction* command with the above arguments yielding the focus

$$\begin{array}{l} \sqsubseteq * \ \mathbf{do}\ l > 0 \wedge (\mathit{lookup}\ a\ l) < (\mathit{lookup}\ a\ (l-1)) \rightarrow \\ \qquad\quad a := (\mathit{swap}\ l\ (l-1)\ a); \\ \qquad\quad l := l-1 \\ \qquad \mathbf{od} \end{array}$$

along with some appropriate conjectures.

Closing several windows gives us an implementation of sorting:

$$
\begin{array}{l}
\sqsubseteq\ *\ |[\ \mathbf{var}\ k : \mathit{num} \cdot k := 0; \\
\qquad\quad \mathbf{do}\ k < (\mathit{size}\ a) \rightarrow \\
\qquad\qquad |[\ \mathbf{var}\ l : \mathit{num} \cdot l := k; \\
\qquad\qquad\quad \mathbf{do}\ l > 0\ \wedge (\mathit{lookup}\ a\ l) < (\mathit{lookup}\ a\ (l-1)) \rightarrow \\
\qquad\qquad\qquad a := (\mathit{swap}\ l\ (l-1)\ a) \\
\qquad\qquad\qquad l := l - 1 \\
\qquad\qquad\quad \mathbf{od}\ ]|; \\
\qquad\quad k := k + 1 \\
\qquad\quad \mathbf{od}\ ]|
\end{array}
$$

8 Conclusions

We have described the Refinement Calculator, a tool for the derivation of provably correct programs. The calculator is an extension of TkWinHOL, a tool that provides a graphical user interface to window inference proof in HOL. The architecture of TkWinHOL has allowed us to make use of several existing packages to reduce the effort involved in implementing the Refinement Calculator, i.e. HOL, the window Library and the Refcalc theory.

Our experience has shown that the window inference style of proof is well suited to program derivation using the refinement calculus. Furthermore, window inference does not preclude the use of the more traditional verification style of program derivation, rather both styles can be easily mixed. For example, the conjectures generated by the application of *Loop Introduction 2* rule of Fig. 5 are exactly those proof obligations used in loop verification. We claim several other advantages for our approach:

- Without the GUI described here, a detailed knowledge of HOL and its metalanguage (ML) is required in order to use the HOL window Library and the Refcalc theory. Even with such detailed knowledge, focussing on subterms is particularly difficult when working directly with the HOL window Library. The GUI improves the usability of the HOL window Library and the Refcalc theory considerably by providing visually-based access to subterms and context assumptions as well as an abstract programming language syntax, and menus and dialogue boxes for the application of transformations.
- The use of HOL as an underlying proof engine gives us a high degree of confidence in the reliability of the tool.
- Existing and future HOL libraries supporting various data structures or more automated proof can easily be used with the tool.
- The programmability of the tool allows us to abstract many similar transformation rules into single user commands. A simple example of this is the way the two loop-introduction rules of Fig. 5 are accessed by a single user command. This reduces the number of different commands that the user needs to be aware of.

One disadvantage of our approach is that defining new transformations requires knowledge HOL and ML. This is because transformations are not provided to the tool in the form of inference rules as used in Fig. 5 but rather are implemented as ML functions. It should be possible to overcome this to an extent by allowing a sequence of existing transformations to be combined into a single transformation. Another disadvantage is that the target programming language syntax is currently hardwired into the tool, making it impossible for users to extend the syntax. This shortcoming, however, is not integral to the tool. It should be possible for us to provide the same facilities via general purpose parser and pretty printer tools.

Independently of the tool described here, a refinement tool called PRT has been developed by a group at the University of Queensland [6]. PRT is built on top of the Ergo theorem prover [27], which also supports the window inference style of reasoning. This tool is similar to the Refinement Calculator though an important difference is that PRT uses a purpose-built logic in which commands, predicates, program variables, and logic variables are treated as separate syntactic classes avoiding the need for a parsing and pretty-printing layer between the user and the proof engine. The Refinement Calculator, on the other hand, uses only a conservative extension of the HOL logic, giving us a higher degree of confidence in its soundness.

Currently the Refinement Calculator supports the refinement of sequential programs, including the application of data refinement, as well as the browsable presentation of proofs. Future plans include extending the tool to support procedures and modules as well as the *action system* formalism [3], an extension of the refinement calculus dealing with parallel and distributed systems. We are also investigating ways of further increasing the usability and extendibility of the tool.

References

1. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
2. R.-J. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, Feb. 1981.
3. R.-J. R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.
4. M. Butler, T. Långbacka, R. Rukšėnas, and J. von Wright. Refinement Calculator tutorial and manual. Draft — available upon request.
5. M. J. Butler and T. Långbacka. Program derivation using the refinement calculator. In von Wright et al. [30], pages 93–108.
6. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A tool for developing correct programs by refinement. In H. Jifeng, editor, *BCS FACS 7th Refinement Workshop*. Springer-Verlag, July 1996.
7. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

8. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.
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. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
11. L. J. Groves, R. G. Nickson, and M. Utting. A tactic driven refinement tool. In C. B. Jones, B. T. Denvir, and R. C. F. Shaw, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272–297, London, Jan. 1992. BCS-FACS, Springer-Verlag.
12. J. Grundy. The HOL window library. In *The HOL System*, volume Libraries. SRI International, Cambridge Research Center, Millers Yard, Mill Lane, Cambridge CB2 1RQ, England, 2.01 edition, July 1991.
13. J. Grundy. A browsable format for proof presentation. *Mathesis Universalis*, 1(2), Spring 1996. Available from `http://www.pip.com.pl/MathUniversalis/2/grundy/mu.html`.
14. J. Grundy. Transformational hierarchical reasoning. *The Computer Journal*, 39(4):291–302, 1996.
15. J. Grundy and T. Långbacka. Towards a browsable record of HOL proofs. Technical Report 7, Turku Centre for Computer Science, Lemminkäisenkatu 14A, 20520 Turku, Finland, May 1996. Available from `http://www.tucs.abo.fi/publications/techreports/TR7.ps.gz`.
16. J. Harrison and L. Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184, Vancouver, Canada, Aug. 1993. Springer-Verlag.
17. L. Laibinis. Using lattice theory in higher order logic. In von Wright et al. [30], pages 315–330.
18. D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. O'Reilly & Associates, 1995.
19. T. F. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.
20. C. C. Morgan. *Programming from Specifications*. Prentice Hall, 2 edition, 1994.
21. C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27(6):481–503, 1990.
22. J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, Dec. 1987.
23. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.
24. T. W. Reps and T. Teitelbaum, editors. *The Synthesizer Generator. A System for Constructing Language-Based Editors*. Springer-Verlag, 1988.
25. P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, Feb. 1993.
26. L. Théry. An X-windows interface for the window inference system. In M. Gordon, ed., The Final Report on DSTO Grant: Improved Interface for HOL, Internal Report. University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, England, 1993.

27. M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Institute, University of Queensland, QLD 4072, Australia, Oct. 1993.
28. T. Vickers. An overview of a refinement editor. In *5th Australian Software Engineering Conference*, pages 39–44, Sydney, May 1990. IREE.
29. J. von Wright. Program refinement by theorem prover. In D. Till and R. C. F. Shaw, editors, *6th Refinement Workshop*, Workshops in Computing, pages 121–150, London, Jan. 1994. BCS-FACS, Springer-Verlag.
30. 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*, Turku, Finland, Aug. 1996.
31. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in Systems Design*, 3:49–81, 1993.