Program Derivation Using the Refinement Calculator

Michael Butler[1] and Thomas Långbacka[2]

[1] Dept. of Electronics & Computer Science, University of Southampton
[2] Dept. of Computer Science, University of Helsinki

Abstract. The refinement calculus provides a theory for the stepwise refinement of programs and this theory has been formalised in HOL. TkWinHOL is a powerful graphical user interface (GUI) that can be used to drive the HOL window Library. In this paper, we describe a tool called the Refinement Calculator which combines TkWinHOL and the HOL Refinement Calculus theory, to provide support for formal program development. The tool improves the usability of the HOL Refinement Calculus theory considerably through its window-inference based GUI and by supporting a conventional programming syntax.

1 Introduction

The refinement calculus [2, 3, 14, 15] is a formalisation of the stepwise refinement method of program construction. The required behaviour of a program is specified as an abstract, possibly non-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 are usually long and error-prone. Therefore the use of a proof assistant such as HOL to increase the level of trust in proofs is a natural step. Work on using HOL for constructing refinement calculus proofs has been carried out for number of years at Åbo Akademi University:

- Work by Back on the refinement calculus [2, 3].
- Work on formalising the refinement calculus theory in HOL [4, 21, 22]
- More recently work on building a software environment supporting program development using the refinement calculus formalisation in HOL.

Although much work has been done on formalising programming logics (e.g. [1, 6]), working with complex objects like programs using a purely textual interface to HOL is difficult, especially for non-HOL users. As well as having a sound proof engine, an important goal of this work on providing tool support for the refinement calculus was that it could be used by people who may be familiar with program refinement, but not so experienced in using HOL.

Usually a linear calculational style is used when deriving programs by hand in the refinement calculus. A specification is transformed in a series of steps to an executable program by applying transformation rules which preserve a refinement relation between the stages. Many of the steps involve sub-derivations on sub-components. It is precisely this style of reasoning that is supported by

Grundy's HOL window Library [10]. But the HOL window Library is difficult to work with directly especially since accessing sub-components of large programs is quite cumbersome.

TkWinHOL [13] is a GUI tool that can be used to drive the HOL window Library. Rather than having to provide complex parameters in order to access sub-terms when starting a sub-derivation, the user can focus on a sub-term using the mouse. It also supports the menu-driven application of transformation rules to terms.

In this paper, we describe a tool called the Refinement Calculator which combines TkWinHOL and the Refinement Calculus formalisation (refcalc for short), to provide support for formal program development. As well as providing all the features of TkWinHOL, the Refinement Calculator supports a conventional programming syntax by providing a parser and pretty-printer layer between the user and the syntax of the refcalc theory. This is especially important since it allows the user to work with program variables, as is usual in the refinement calculus, whereas program variables don't appear in the syntax of the refcalc theory. Another feature of the refcalc theory is that it is a *shallow embedding*, i.e., programming constructs are defined semantically rather than through the syntax. This means any existing theories about data-types can be readily used in the Refinement Calculator. For example, in Section 4, we show how an existing theory of arrays can be used to work with arrays in program refinement.

2 Background

In this section we will give a brief description of those parts of the refinement calculus and its formalisation in HOL that are relevant to this paper.

2.1 The Refinement Calculus

The refinement calculus is based on Dijkstra's weakest precondition semantics for programs [8]. Dijkstra's work is extended by introducing a refinement relation between programs. Another change to Dijkstra's original work is the (partial) relaxation of the required *healthiness conditions* of program statements, to allow the introduction of useful (yet unimplementable in practice) specification statements into the set of statements.

The programming notation used in this paper is basically Dijkstra's guarded commands language, extended with *assertions* and *nondeterministic assignment*. An assertion has the form $\{p\}$, where p is a predicate on the program state; occurrence of an assertion in a program allows us to assume that p holds at that point in the program. A nondeterministic assignment has the form $x := x' \bullet p$ and specifies that x is assigned some value x' satisfying predicate p.

The refinement relation is defined in terms of the weakest preconditions of the related programs. For program S and postcondition P, $wp(S,P)$ represents the weakest precondition under which S is guaranteed to terminate in a state satisfying P. Program S_0 is refined by S_1, denoted $S_0 \sqsubseteq S_1$, iff

$$\forall P \cdot wp(S_0,P) \Rightarrow wp(S_1,P)$$

which states that program S_1 must preserve the total correctness of program S_0. The refinement relation is a preorder. Thus programs can be developed in a *linear* fashion as in the following sequence

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

which establishes the refinement $S_0 \sqsubseteq S_n$, because of the transitivity of the relation.

An important property is that one can refine sub-components of programs without affecting the total correctness of the whole program. Formally this means that if we have proven that $T \sqsubseteq T'$ then we have in fact established the refinement $S[T] \sqsubseteq S[T']$ provided the program context $S[\ldots]$ is monotonic.

2.2 Formalising The Refinement Calculus in HOL

Program state When formalising a (state-based) programming logic in HOL an important decision one has to make is how to represent the state of a program. Since refcalc is a shallow embedding, one natural way to deal with states is to represent them as tuples. On the general level, states are defined using the polymorphic type. In individual programs, each program variable is represented as one component in the tuple. This means that program variables are anonymous.

This could be seen as a disadvantage of the approach, making the practical use of the theory unmanageable. The Refinement Calculator described in section 3, hides the state representation under a surface syntax allowing program variables to be used. In the translation, program variables are modelled using *projection functions*. Assume the state has type $num \times bool$, then the variables indicate positions in the state tuple as follows:

$$let\ x = FST\ \ in\ let\ y = FST \circ SND\ in...$$

The main advantage in using a shallow embedding is that one can reuse HOL data types in the state. So program variables ranging over, e.g., natural numbers can be treated by means of existing theorems and proof procedures without difficulty. In case new data types are defined, these can also be used without having to worry about incompatibility (see section 4 for an example of this). In a *deep embedding* data types have to be embedded into the theory every time new types are to be used.

Predicates, predicate transformers and refinement The semantics used is weakest precondition semantics where the meaning of programs are defined using predicates over the program state. Thus, predicates (*pred*) in refcalc are functions of type $state \to bool$, where the state is represented as described above.

Program statements are *predicate transformers*, i.e., functions of type $pred \to pred$. For predicate transformer S and predicate q, $S\ q$ corresponds to $wp(S,q)$ in Dijkstra's formalism. Assuming q is a predicate, s is a program state, f is a state function ($state \to state$) and $S1$ and $S2$ are predicate transformers, we can define the following three predicate transformers:

$$
\begin{aligned}
&\mathit{skip}\ q = q\\
&\mathit{assign}\ f\ q = \lambda s \cdot q(f\ s)\\
&\mathit{seq}\ S1\ S2\ q = S1(S2\ q)
\end{aligned}
$$

representing *skip*, *variable assignment* and *sequential composition*. An assignment statement in the surface syntax of the Refinement Calculator having the form $x := e$, would be translated into the form $\mathit{assign}(\lambda x \cdot e)$.

The refinement relation is defined as follows (here *implies* is HOL implication lifted to the level of predicates):

$$S1 \sqsubseteq S2 \ = \ \forall q \cdot (S1\ q)\ \mathit{implies}\ (S2\ q).$$

2.3 The use of window inference as proof engine

The refinement calculus has characteristics that make window inference [16] and more specifically the implementation of window inference in HOL, the HOL window Library [10, 11] an attractive environment to work in when carrying out proofs. In this section we will very briefly describe the HOL window Library and discuss why it is a useful proof engine in the context of the refcalc theory and the Refinement Calculator. We will also very briefly describe the steps taken in [21] to adapt the refcalc theory for use with the Window Library.

Window inference In window inference, one transforms a term preserving a *preorder*. An expression p transformed to q preserving the relation R, gives a proof of $p\ R\ q$.

A transformation of a sub-term is a transformation of the whole term, if certain *context-dependent monotonicity* conditions hold. For example, in the expression $A \wedge B$, the sub-term A can be transformed under assumption B (assuming implication is the relation preserved) since

$$\frac{B \vdash A \Rightarrow A'}{\vdash (A \wedge B) \Rightarrow (A' \wedge B)}$$

The HOL implementation supports window reasoning under *implication*, *equality* and *reverse implication*. New relations can be added to the system by proving that the relation is *reflexive* and *transitive* (i.e., a preorder) and adding rules that govern how windows can be opened and closed.

Users work with so called *window stacks*. Windows consist of a *focus* (the term being transformed), the relation preserved and a *context* (assumptions, lemmas and conjectures). Windows also hold a *window theorem* which records the current proof state. Windows can be opened on sub-terms of the focus using the operators *rand*, *rator* and *body* to identify the sub-term one is interested in. The activity of proof is carried out using two main categories of operations. One can either change the scope of interest by opening or closing windows or one can transform the current focus using pre-proved theorems, conversions, rewrite rules etc[3].

[3] The HOL window Library also contains a hybrid command *AT* which can be used to perform a transformation of a sub-term as if a new window had been opened.

Why window inference? From the refinement point of view the HOL window Library is well suited as the basic inference mechanism since:

- The refinement relation is a preorder.
- Refinement steps are often refinements of sub-components of programs (i.e. refinement in context)
- In a program derivation one doesn't necessary know exactly in what direction the program development might go so being able to reason transformationally, as in window inference, is an advantage.

Thus, the window inference mechanism offers a suitable environment for structuring proofs within the refcalc theory.

The HOL window Library has other attractive features:

- Window inference encompasses both *forward* and *backward* reasoning.
- The user has full control over at which level of detail to reason.
- The transformations one performs to the current focus are usually simple (in HOL terms); typically they are simple rewrites.
- Contextual information that might prove useful in transformations is made available when windows are opened.

Furthermore, as Grundy phrased it in [9], "The window inference system suggests a graphical user interface"; in particular, it suggests the use of a mouse to select sub-terms and access contextual information, and the use of menus, buttons and dialogs to enter commands.

Refinement using window inference In order to use window inference in program development, one has to prove that the refinement relation is transitive and reflexive, and add rules for opening and closing of windows. The first part is trivial.

A program may be specified with a statement of the form $\{pre\}; v := v' \bullet post$. 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 sub-programs as, for example, the following 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 refinement calculus provides a set of rules for transforming the current focus in order to introduce extra program structure to the focus, or to simplify the focus. The refinement rules supported by the refinement calculator are all derived from the HOL semantics of programs and refinement described above. Some of these rules are listed in Section 3.

Assertion statements are used to carry context information around. This context can be used directly when we focus on the right hand side of an assignment or the postcondition of a nondeterministic assignment:

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

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

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 backwards implication.

Context information is propagated using theorems such as the following:

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

$\vdash\ \ x := e\ \ \sqsubseteq\ \ x := e;\{x = e\}$

3 The Refinement Calculator

The Refinement Calculator consists of:

- TkWinHOL[12, 13] – an extendable general purpose interface to the HOL window Library .
- Extensions to TkWinHOL dealing with refinement specific issues.

As mentioned above we have tried to make the base tool TkWinHOL general and extendable in nature. Unlike the work in e.g. [18, 19] we have not attempted to build a general purpose user interface to the HOL system rather we restricted ourselves to window inference. Also our approach has all the time been aiming at a specific extension, the Refinement Calculator.

In [18] Syme also discusses HCI aspects of user interface building as an influence on his work. Our approach is less structured though it is strongly influenced by the calculational style of proof and by window inference. We have tried to build a tool with a user interface that looks and feels like any "typical" modern graphic user interface. Our aim is to minimize the amount of typing a user has to do and allow the user to interact in a "select – operate" type of way.

3.1 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 presents a structured view of the current state of the active window stack. Thus different types of information (i.e., focus, context, window theorem) about the current stack is displayed in different regions of the window.

Working with TkWinHOL Accessing sub-terms using the mouse instead of textual path information (when opening windows) is an important feature in a system such as this. In fact the selection of parts of the focus stretches beyond this. One can select a segment of the focus that wouldn't normally be accessible without reordering the associations in the focus, and have the interface perform the required reordering automatically.

The commands of the HOL window Library are bound to buttons and menus. For many commands the information needed is typically made available using the mouse to select relevant information already present on the screen. If that is not enough, information is entered through dialogs. The dialogs in TkWinHOL are implemented in such a way that they don't have to be destroyed (e.g. by clicking the OK button) before data in other windows can be selected. Thus, it is possible to use information already present on the screen to fill some of the fields in any given dialog.

Customisability Emphasis has been placed on making TkWinHOL easily customisable (for the purpose of building the Refinement Calculator). When using TkWinHOL together with a specific theory (as in the Refinement Calculator) one usually wants to use a higher level notation for the interaction. This can be achieved through two steps

- By constructing a translator from the higher level notation to the corresponding HOL representation.
- By constructing a pretty-printer doing the opposite. Actually the pretty-printer has to do more since all output from the HOL window Library commands have to be augmented with some instructions to the interface on which sub-terms of the focus are to be selectable etc. Because of this the base TkWinHOL system has a default pretty-printer although there is no translator.

Another way in which one wants to customise TkWinHOL is by adding theory specific menu choices bound to specialised commands for transformations etc. To support this there is a very general prompting procedure (prompting is done through dialogues) built into TkWinHOL. The idea is that through this prompting procedure one can easily build the glue by which to tie an add-on menu choice to a command built on the HOL level.

For parameterless commands such glueing is not necessary (the menu choice is bound more or less directly to the command) but for any command needing parameters, it is convenient to have some method to specify how this parameter passing should take place, and not force the user to patch the user interface source code directly.

The way this is accomplished is through binding add-on menu choices to a call to the above mentioned prompting procedure. As parameters to this procedure one can describe how many fields the prompting dialogue should have and possibly what individual fields have as default value. To construct the actual commands to be sent to HOL one provides so called command specifications as additional parameters to this procedure. These are simply text strings where one can refer to the contents of the different fields (as well as the current screen selection) through symbolic names. Once the dialogue is ok'd by the user, a substitution takes place. One can provide several command specifications to the procedure. Which one is used is determined by where on the screen the selection sits (i.e., if information from the context of the focus is selected do one thing, if a part of the focus is selected do something else etc).

3.2 The Refinement Calculator Extension

The Refinement Calculator supports a customised program notation with the following syntax:

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

Here, variable name v may represent a list of variable names, *Type* represents any HOL type, *Term* represents any HOL term, and *BTerm* represents any boolean-valued HOL term. Thus the standard HOL notation is embedded in the program-specific notation. The last construct, $|[\ \textbf{var}\ v : Type \cdot Com\ ]|$, represents a block with local variable v.

Cond Introduction:

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

Block Introduction:

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

Loop Introduction:

$$\frac{\begin{array}{l} \vdash\ pre\ \Rightarrow\ inv \\ \vdash\ (inv\ \wedge\ G\ \wedge\ E = e) \ll Body \gg (inv\ \wedge\ 0 \leq E < e) \\ \vdash\ inv\ \wedge\ \ G\ \Rightarrow\ post[v' := v] \\ v \text{ not free in } post \end{array}}{\vdash\ \{pre\}; v := v' \bullet post\ \sqsubseteq\ \textbf{do}\ G \to Body\ \textbf{od}}$$

Assignment Introduction:

$$\frac{\vdash\ pre\ \Rightarrow\ post[v' := E]}{\vdash\ \{pre\};\ v := v' \bullet post\ \sqsubseteq\ v := E}$$

Leading Assignment Introduction:

$$\frac{x \text{ not free in } post}{\vdash\ v := v' \bullet post\ \sqsubseteq\ x := E; v := v' \bullet post}$$

Fig. 1. Refinement Transformations

The Refinement Calculator provides a menu of transformations for refining programs; some of these are represented as rules in Figure 1. Each of the rules requires the user to provide some arguments before the transformation is applied; for example, the *Cond-Introduction* rule requires the guard G to be supplied. Some of the rules have side-conditions (about free variables) and assumptions. The assumptions are added to the current window as conjectures that can be established later. A transformation will fail if its side-conditions are not satisfied.

Note that a term of the form $pre \ll prog \gg post$, as used in the *Loop-Introduction* rule of Figure 1, 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 also provides a transformation for propagating context information as assertion statements and for applying rewrite theorems to the focus. All the transformations supported by the calculator are provided by glueing menus to the appropriate HOL command as outlined previously.

The role of the rules of Figure 1 will be described more clearly in the next section where they are applied to some examples. The full program syntax and list of refinement rules supported by the Refinement Calculator may be found in [5].

4 Some Example Derivations

We present three example derivations that may be carried out using the refinement calculator: a program that finds the maximum of two numbers, a program that finds the maximum value in an array of numbers, and a program that sorts an array of numbers. In the following, we write a window as

$R \; * \; f$

where R is the relation to be preserved and f is the term to be transformed. Also, we write context assumptions as ! p and conjectures as ? p.

4.1 Finding the Maximum of Two Numbers

This first example illustrates the use of a structure introduction rule, refinement of program sub-components, and the use of context information in refinement.

Assume the operator *max*, is defined by:

$m \; max \; n = if \; (m \geq n) \; then \; m \; else \; n$

The program we wish to derive is then specified as follows:

$\textbf{program} \; maximum \; \textbf{var} \; m,n,x : num. \quad x := m \; max \; n$

The derivation commences by opening a window on this specification, with $\sqsubseteq$ as the relation to be preserved:

$\sqsubseteq \; * \; \textbf{program} \; maximum \; \textbf{var} \; m,n,x : num. \quad x := m \; max \; n$

Let us assume that the *max* operator is not available in our target programming language so that we have to implement this specification by comparing m and n. We proceed by introducing an if-statement: using the Refinement Calculator, we focus on the assignment statement and then apply the *Cond-Introduction* transformation with $m \geq n$ (i.e., the guard) as an argument. This results in the focus:

$\sqsubseteq \; * \; \textbf{if} \; m \geq n \; \textbf{then} \; x := m \; max \; n \; \textbf{else} \; x := m \; max \; n \; \textbf{fi}$

When refining the first branch of this statement, we can assume that $m \geq n$ holds. To make this assumption available in the branches, the guard and it's negation are propagated as assertions:

$\sqsubseteq \; * \; \textbf{if} \; m \geq n \; \textbf{then} \; \{m \geq n\}; \; x := m \; max \; n \; \textbf{else} \; \{\neg(m \geq n)\}; \; x := m \; max \; n \;\; \textbf{fi}$

Focusing on the first branch yields:

$\sqsubseteq\ *\ \{m \geq n\};\ x := m\ max\ n$

Now, focusing on the right hand side of the assignment yields the window (see rule R3, page 6):

$!\ m \geq n$
$=\ *\ m\ max\ n$

Notice that the relation to be preserved on this new focus is equality. Using the definition of *max*, the focus may be rewritten to

$=\ *\ if\ (m \geq n)\ then\ m\ else\ n$

Selecting the assumption $m \geq n$, and rewriting the focus results in

$=\ *\ m$

Closing the window yields:

$\sqsubseteq\ *\ \{m \geq n\};\ x := m$

Since the assertion is no longer required, it can be dropped resulting in:

$\sqsubseteq\ *\ x := m$

Closing this window yields:

$\sqsubseteq\ *\ \textbf{if}\ m \geq n\ \textbf{then}\ x := m\ \textbf{else}\ \{\neg(m \geq n)\};\ x := m\ max\ n\ \ \textbf{fi}$

The second branch[4] may be refined in a similar manner, so that we end up with the program

$\sqsubseteq\ *\ \textbf{if}\ m \geq n\ \textbf{then}\ x := m\ \textbf{else}\ x := n\ \textbf{fi}$

Thus, we have constructed the theorem:

$\vdash\ x := m\ max\ n \sqsubseteq \textbf{if}\ m \geq n\ \textbf{then}\ x := m\ \textbf{else}\ x := n\ \textbf{fi}$

4.2 Finding the Maximum of an Array of Numbers

Let s represent a set of numbers, i.e., $s : num \rightarrow bool$. The maximum element of s is selected by *Max*:

$Max\ s = (\varepsilon\ m \cdot s\ m\ \wedge\ (\forall n \cdot s\ n\ \Rightarrow\ m \geq n))$

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:

$asize : (\alpha)array \rightarrow num$
$lookup : (\alpha)array \rightarrow num \rightarrow \alpha$
$swap : (\alpha)array \rightarrow num \rightarrow num \rightarrow (\alpha)array$

An array a is indexed from 0 to $(asize\ a) - 1$. 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.

For array a, $(elems\ a\ n)$ represents the set of values of a in the range $0..n-1$:

[4] Note that the two branches could have been refined in either order.

$elems\ a\ n = (\lambda x \cdot \exists i \cdot i < n \ \wedge\ (lookup\ a\ i) = x)$

A program that finds the maximum value of a non-empty array is specified as follows:

$\textbf{program}\ Maximum\ \textbf{var}\ a : (num)array;\ m : num\cdot$
$\quad \{(asize\ a) > 0\};\ m := Max\ (elems\ a\ (asize\ a))$

We will implement this using a loop that traverses the array starting at position zero. The array will be traversed using an indexing variable $k : num$. In order to apply the *Loop-Introduction* rule, we require a loop guard (G), a loop body (*Body*), a loop invariant (*inv*), and a loop variant (E), as follows:

$Guard:\quad k < (asize\ a)$
$Body:\quad m,k := m',k' \bullet m' = Max(elems\ a\ k')\ \wedge\ k' = k+1$
$Invariant: k \leq (asize\ a)\ \wedge\ m = Max\ (elems\ a\ k)$
$Variant:\quad (asize\ a) - k$

The invariant says that m is the maximum value in the array slice $a[0..k-1]$. The body increments k and re-establishes the invariant.

Before introducing the loop, we introduce k as a local variable by applying *Block Introduction* to the specification:

$\sqsubseteq\ *\ |[\ \textbf{var}\ k : num \cdot m,k := m',k' \bullet m' = Max(elems\ a\ (asize\ a))\ ]|$

Both m and k will have to be initialised to establish the invariant, so we add an assignment before the body of this block (using *Leading Assignment Introduction*):

$\sqsubseteq\ *\ |[\ \textbf{var}\ k : num \cdot m,k := (lookup\ a\ 0), 1;$
$\qquad m,k := m',k' \bullet m' = Max(elems\ a\ (asize\ a))\ ]|$

Propagating assertions through appropriately and focusing on the second statement of the block yields:

$\sqsubseteq\ *\ \{(asize\ a) > 0\ \wedge\ m = (lookup\ a\ 0)\ \wedge\ k = 1\};$
$\quad m,k := m',k' \bullet m' = Max(elems\ a\ (asize\ a))$

Now applying the *Loop-Introduction* rule with the guard, body, invariant, and variant described above as arguments, yields the focus

$\sqsubseteq\ *\ \textbf{do}\ k < (asize\ a) \rightarrow$
$\qquad k < (asize\ a)\ \wedge\ m = Max(elems\ a\ k);$
$\qquad m,k := m',k' \bullet m' = Max(elems\ a\ k')\ \wedge\ k' = k+1\ \textbf{od}$

This step also generates a number of conjectures. These state respectively that the invariant should hold initially (1), the body should preserve the invariant and decrease the variant (2), and the invariant and the negated guard should establish the original postcondition (3):

$$? \quad \forall a,m,k \cdot (asize\ a) > 0\ \wedge\ m = (lookup\ a\ 0)\ \wedge\ k = 1 \\ \Rightarrow\ k \leq (asize\ a)\ \wedge\ m = Max(elems\ a\ k) \tag{1}$$

$$? \ \forall k,a,m,e \cdot \\ k < (asize\ a)\ \wedge\ m = Max(elems\ a\ k)\ \wedge\ ((asize\ a) - k) = e \\ \ll m,k := m',k' \bullet m' = Max(elems\ a\ k')\ \wedge\ k' = k+1 \gg \\ k \leq (asize\ a)\ \wedge\ m = Max(elems\ a\ k)\ \wedge\ 0 \leq ((asize\ a) - k) < e \tag{2}$$

$$? \forall k,a,m \cdot \; (k < (asize\ a)) \;\wedge\; k \leq (asize\ a) \;\wedge\; m = Max(elems\ a\ k) \\ \Rightarrow \;\; m = Max(elems\ a\ (asize\ a)) \tag{3}$$

At this stage in the derivation, we can either attempt to discharge the conjectures[5], or continue refining the body of the program. Conjectures such as above may be established by opening windows on the respective conjectures with backward implication as the relation to be preserved, and then transforming the focus to true. The second conjecture above can be simplified by applying a transformation which reduces a correctness assertion (of the form *pre* $\ll$ *assignment* $\gg$ *post*) to a boolean term. When proceeding with the loop body above, we focus on the sub-term $Max(elems\ a\ k')$ of the assignment statement yielding the window:

$$\begin{array}{l} !\;\; k < (asize\ a) \quad !\;\; m = Max(elems\ a\ k) \quad !\;\; k' = k+1 \\ = \;*\; Max(elems\ a\ k') \end{array}$$

We can rewrite the focus using the assumption $k' = k+1$ to

$$= \;*\; Max(elems\ a\ (k+1))$$

Using a theorem about Max and the assumption $k < (asize\ a)$, we rewrite this to

$$= \;*\; (Max\ (elems\ a\ k))\ max\ (lookup\ a\ k)$$

and using the assumption $m = Max(elems\ a\ k)$, this is rewritten to

$$= \;*\; m\ max\ (lookup\ a\ k)$$

Closing the current window and focusing on the assignment statement, we now have:

$$\sqsubseteq \;*\; m,k := m',k' \bullet m' = m\ max\ (lookup\ a\ k) \;\wedge\; k' = k+1$$

This is easily transformed to

$$\begin{array}{l} \sqsubseteq \;*\; m := m\ max\ (lookup\ a\ k); \\ \qquad\; k := k+1 \end{array}$$

In the manner of the previous section, the first of these may be refined to

$$\sqsubseteq \;*\; \mathbf{if}\ m < (lookup\ a\ k)\ \mathbf{then}\ m := (lookup\ a\ k)\ \mathbf{else}\ skip\ \mathbf{fi}$$

By closing several windows we have arrived at the program

$$\begin{array}{l} \sqsubseteq \;*\; |[\ \mathbf{var}\ k : num \cdot m,k := (lookup\ a\ 0),1; \\ \qquad\quad \mathbf{do}\ k < (asize\ a) \rightarrow \\ \qquad\qquad \mathbf{if}\ m < (lookup\ a\ k)\ \mathbf{then}\ m := (lookup\ a\ k)\ \mathbf{else}\ skip\ \mathbf{fi}; \\ \qquad\qquad k := k+1\ \mathbf{od}\ ]| \end{array}$$

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

4.3 Sorting an Array

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

$$
\begin{array}{l}
\mathit{sorted}\ (a : (\mathit{num})\mathit{array})\ (r : \mathit{num} \to \mathit{bool}) = \\
\quad (\forall i \cdot (r\ i)\ \Rightarrow\ i < (\mathit{asize}\ a))\ \wedge \\
\quad (\forall i j \cdot (r\ i)\ \wedge\ (r\ j)\ \wedge\ i < j\ \wedge\ j < (\mathit{asize}\ a)\ \Rightarrow\ (\mathit{lookup}\ a\ i) \leq (\mathit{lookup}\ a\ j))
\end{array}
$$

$$
\begin{array}{l}
\mathit{perm}\ (a1 : (\mathit{num})\mathit{array})\ (a2 : (\mathit{num})\mathit{array}) = \\
\quad (\mathit{asize}\ a1) = (\mathit{asize}\ a2)\ \wedge \\
\quad (\exists f \cdot (\mathit{injective}\ f)\ \wedge \\
\qquad (\forall i \cdot i < (\mathit{asize}\ a1)\ \Rightarrow\ (f\ i) < (\mathit{asize}\ a1)\ \wedge \\
\qquad\qquad (\mathit{lookup}\ a1\ i) = (\mathit{lookup}\ a2\ (f\ i))))
\end{array}
$$

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

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

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 one step at a time ensuring that the array slice $0..k-1$, where k is the current position, is sorted. This suggests the following guard, invariant, and variant for the outer loop:

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

Before transforming the specification using loop introduction, k is introduced as a local variable, and $a0$ (the initial value of a) is introduced using an assertion:

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

Now, using

$$
\begin{array}{l}
\{k < (\mathit{asize}\ a)\ \wedge\ (\mathit{sorted}\ a\ (\lambda i \cdot i < k))\ \wedge\ (\mathit{perm}\ a0\ a)\}; \\
a, k := a', k' \bullet (\mathit{sorted}\ a'\ (\lambda i \cdot i < k'))\ \wedge\ (\mathit{perm}\ a0\ a')\ \wedge\ k' = k + 1
\end{array}
$$

as the body of the outer loop, application of loop introduction yields:

$$
\begin{array}{l}
\sqsubseteq\ *\ \textbf{do}\ k < (\mathit{asize}\ a) \to \\
\qquad\quad \{k < (\mathit{asize}\ a)\ \wedge\ (\mathit{sorted}\ a\ (\lambda i \cdot i < k))\ \wedge\ (\mathit{perm}\ a0\ a)\}; \\
\qquad\quad a, k := a', k' \bullet (\mathit{sorted}\ a'\ (\lambda i \cdot i < k'))\ \wedge\ (\mathit{perm}\ a0\ a')\ \wedge\ k' = k + 1\ \textbf{od}
\end{array}
$$

This step generates some conjectures similar to those shown in the previous example derivation.

The body of this loop is easily transformed to:

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

Thus, under the assumption that the array is sorted between 0 and $k-1$, this body 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:

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

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. Loop introduction yields

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

Closing several windows gives us the implementation:

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

5 Conclusions

We have described the Refinement Calculator, a tool for the derivation of provably correct programs. The Refinement Calculator makes it feasible to do program refinement using HOL by providing a powerful GUI and a high level surface syntax on top of a Refinement Calculus HOL formalisation and the HOL window Library. The window inference mechanism supports the transformational style of reasoning used in program refinement. The ability to perform sub-derivations by simply focusing using the mouse makes the tool scalable to larger programs.

An important feature of the system is that it effectively hides the underlying representation of the program state, thus minimizing the possible drawbacks of using a shallow embedding of the formalism. That way one can also make use of the advantages of shallow embeddings by using existing theories of data types. This was demonstrated in the examples provided in section 4.

Independently of the tool described here, a refinement tool called PRT has been developed by a group at the University of Queensland [7]. PRT is built on top of the Ergo theorem prover [20] which also supports the window inference style of reasoning. This tool is quite similiar 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. In contrast, the Refinement Calculator uses only a conservative extension of the HOL logic giving us a higher degree of confidence in its soundness. However, having a closer match between the programming syntax and the underlying logic, as PRT has, may also have advantages, and the comparison between the approaches deserves a more thorough evaluation.

Much work can still be done on improving the Refinement Calculator. The underlying HOL theory still needs more proof support. There is also a need to extend the theory (and consequently the high level language) to support data refinement and action systems (to deal with reactive programs). Furthermore, the interface itself needs to be improved upon to deal, for example, with theorem retrieval, re-running proofs, etc.

Acknowledgments

Jim Grundy, Rymvidas Rukšėnas, and Jockum von Wright were also directly involved in the development of the Refinement Calculator.

References

1. F. Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, Lyngby, 1992.
2. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.
3. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
4. R. Back and J. von Wright. Refinement concepts formalized in higher order logic. *Formal Aspects of Computing*, 2:247–272, 1990.
5. M. Butler, T. Långbacka, R. Rukšėnas, and J. von Wright. Refinement Calculator tutorial and manual. Draft – available upon request.
6. A. Camillieri. Mechanizing CSP trace theory in Higher Order Logic. *IEEE Transactions on Software Engineering*, 16(9):993–1004, 1990.
7. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A tool for developing correct programs by refinement. For presentation at *7th BCS–FACS Refinement Workshop*, July 1996.
8. E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.
9. J. Grundy. Window inference in the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, pages 177–189, University of California at Davis, August 1991. ACM-SIGDA, IEEE Computer Society Press.
10. J. Grundy. A window inference tool for refinement. In Jones et al, editor, *Proc. 5th Refinement Workshop*, London, Jan. 1992. Springer–Verlag.
11. J. Grundy. HOL90 window library manual. 1994.
12. T. Långbacka. TkWinHOL users guide. Draft – available upon request.
13. T. Långbacka, R. Rukšėnas, and J. von Wright. TkWinHOL: A tool for doing window inference in HOL. In Schubert et al. [17], pages 245–260.

14. C.C. Morgan. *Programming from Specifications (2nd Edition)*. Prentice–Hall, 1994.
15. J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comp. Prog.*, 9(3):298–306, 1987.
16. P.J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47–61, February 1993.
17. E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors. *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, Aspen Grove, Utah, September 1995. Springer-Verlag.
18. D. Syme. A new interface for HOL – ideas, issues and implementation. In Schubert et al. [17], pages 324–339.
19. L. 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. M. Utting and K. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, University of Queensland, 1994.
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.
22. J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanising some advanced refinement concepts. *Formal Methods in Systems Design*, 3:49–81, 1993.

This article was typeset using the LaTeX macro package with the LLNCS2E class.