Window Inference in Isabelle

Mark Staples

10 August 1995

Abstract

Window inference is a transformational style of reasoning that provides an intuitive framework for managing context during the transformation of subterms under transitive relations. This report describes the design for a prototype window inference tool in Isabelle, and discusses possible directions for the final tool.

1 Introduction

Window inference is a transformational style of reasoning that provides an intuitive framework for managing the transformation of subterms. Like natural deduction or the sequent calculus, window inference is a generic style of inference applicable across a wide variety of logics.

A simple window inference mechanism for the Isabelle theorem prover [14] has been developed. It is based on a notion of proof state similar to that of Isabelle's subgoal package. It uses Isabelle's meta variables to effect subterm rewriting, and Isabelle's meta logic to represent window transformation and window opening rules.

A brief outline of window inference is given in Section 2. Later sections will serve to further explain window inference, but for a full explanation, see [15, 9] Section 3 details the core of the design of the prototype window inference tool for Isabelle. The window inference style of reasoning is similar to the standard presentation of program refinement [2, 11, 10] proofs. Because of this, window inference is gaining popularity as a basis for the construction of refinement tools in theorem provers — refinement tools [8, 17, 3, 4] constructed with the Hol and Ergo theorem provers are based on their window inference facilities. The author's intention for a window inference tool in Isabelle is to use it as the basis of a program refinement tool. Hence in Section 4, the relevant setting of refinement as window inference is used to present a worked example explaining window inference.

2 Window Inference

Window inference was proposed by Robinson and Staples [15] as a 'hierarchical, problem-reduction style of reasoning'. Window inference maintains a hierarchy of subproblems at any one point in a proof — while a problem can be readily decomposed into smaller problems, the information present in the original problem isn't lost. The first versions of window inference gave a central role to equivalence transformations (both logical equivalence and other equivalence relations) which allowed it to be used to reason about non-formulae terms. Window inference also originally provided access to contextual information during the transformation of subproblems, with contextual hypotheses present behind an object logic implication. Window inference was first implemented in the theorem prover demo2, and is now the main style of reasoning in its descendant theorem prover Ergo [16].

Window inference was later generalised by Grundy [7, 9], who relaxed the requirement on relations to preorder (reflexive and transitive) relations. However, Grundy accessed context through derivability

rather than implication. The main reason for this seems to be that he was then able to take advantage of the facilities of the HOL theorem prover [6] for manipulating hypotheses in the proof of theorems.

Window inference is about the transformation of a term of interest (the *focus*) F, under some preorder relation R. A *window*

$$R \ * F \ \ !H$$

states an intention to prove $H \vdash R(F, F')$ for some resulting F'.[1] Theorems of the form $R(F, F')$ can be used to transform a focus F under a relation R to a focus F'. The transformation of subterms of a focus is justified by the use of *opening rules* of the form:

$$\frac{H \vdash r(f, f')}{\vdash R(F[f], F[f'])}$$

The transformation of the subterm f under additional contextual hypotheses H also takes place using window inference. The new subwindow representing this transformation is pushed onto a *window stack*. The window stack thus records the state of the problem decomposition in the window proof, with the top window of the stack describing the current problem, and the bottom window of the stack describing the initial problem. Thus, although window inference provides problem decomposition, it doesn't lose information about the original form of the problem.

So, there are four main operations in window inference.

make an initial window Start the window proof for a specific focus F and relation R.

transform a window Transform a window using a transformation theorem as described above.

open a window Open a window at a selected subterm. The child window is pushed on the window stack. The new focus is the selected subterm of the parent window, and the relation in the subwindow is governed by the opening rule used. Contextual hypotheses of the parent window are inherited by the child window. Opening rules may also justify the addition of extra contextual hypotheses in the child window.

close a window Close back to a parent window. The focus of child window replaces the original subterm in the parent window which was selected for opening. The child window is popped from the window stack.

Part of the 'user interface' to window inference concerns hiding the choice of opening rules from the user. It should not be necessary to specify a specific opening rule in order to open a window. Rather, the position of a specific subterm is given, and the window opening mechanism uses that position information to select among the possible opening rules to apply. More importantly, a user may specify the position of a deep subterm (i.e. a subterm not directly accessible via the available window opening rules), and the window inference mechanism will implicitly compose many opening rules to open at that position, increasing the size of the window stack by one.

3 Window Inference In Isabelle

The normal representation of the state of a window inference proof in Isabelle can be viewed as a state in a subgoal proof. In fact the interface to window inference in the prototype Isabelle window inference tool is through the subgoal package. However, later versions of window inference may provide an alternative interface capable of remembering other state information.

[1] We follow Grundy [7] in putting contextual hypotheses behind derivation rather than implication, for much the same reasons — there is more support in Isabelle for manipulating hypotheses rather than left-hand sides of object logic implications.

The core of the representation of window inference in Isabelle is based (in the terminology of the Isabelle subgoal package) on the following two ideas:

- a window with focus F respecting a relation R is represented by a subgoal with conclusion $R(F, ?Goal)$ where $?Goal$ is some scheme variable; and
- after performing n window openings, the first n subgoals correspond to the window stack of size n, with the first subgoal being the top of the window stack. Subgoals after the first n are unproved side conditions of some kind.

Winsets (window sets) are used to maintain collections of known relations (for which there must be reflexivity and transitivity theorems) and opening rules. Winsets also contain information used to effect rule re-use.

The four main operations in window inference: making the initial window, window transformation, window opening, and window closing, are described below.

3.1 Initial Window

The prototype window inference command for starting a window proof is as follows.

`win_proof` *thy ws str* starts a subgoal proof of *str* in theory *thy*, and remembers the winset *ws* to be implicitly used throughout the proof. The string *str* should parse in theory *thy* to a term with conclusion $R(F, ?G)$ for some relation R, initial focus F and scheme variable $?G$. The relation R should be present in the winset *ws*.

There are three kinds of support for each of the main activities of window transformation, window opening and window closing.

1. Tactics which require the user to supply the winset to use, the number of the subgoal to affect and possibly auxiliary arguments. These tactics are called `WIN_TRANS_TAC`, `WIN_OPEN_TAC` and `WIN_CLOSE_TAC`.
2. Tactics which require the user to supply auxiliary arguments. It affects subgoal number 1 (the top of the window stack), and uses the current winset set by `win_proof`. These tactics are called `win_trans`, `win_open`, and `win_close`.
3. Commands equivalent to `by <the previous tactic>`. These commands are called `wt`, `wo`, `wc` in keeping with the spirit of existing Isabelle commands such as `br`, etc.

3.2 Window Transformation

A window represented by a subgoal with conclusion $R(F, ?Goal)$ is transformed by using a theorem

$$[\![H_1; ...; H_n]\!] \Rightarrow R(F_u, F1)$$

where F and F_u are unifiable. This theorem is used in conjunction with the transitivity theorem for R to construct a rule

$$[\![H_1; ...; H_n; R(F1, ?G)]\!] \Rightarrow R(F_u, ?G)$$

Resolving this rule against the window subgoal results in a transformed window represented by a subgoal with conclusion $R(F1', ?Goal)$ where $F1'$ is the result of applying the unifier of F and F_u to $F1$. If a constructed transformation rule has any hypothesis H_i, it will appear as an individual subgoal H_i' in the subgoal stack after the subgoals representing the window stack.

The prototype window inference commands for window transformation are:

`WIN_TRANS_TAC` *ws thl n* is a tactic which transforms subgoal n by using the transitivity theorems in the winset ws, and the transformation theorems in the list *thl*.

`win_trans` *thl* is a tactic which transforms the top of the window stack by using the theorems in the list *thl*.

`wt` *th* is equivalent to `by(win_trans [th])`

Windows can also be transformed by simplification using the Isabelle simplifier.[2]

3.3 Window Opening

A window represented by a subgoal with conclusion $R(F[f], ?Goal)$ is opened at f by using a theorem of the form[3]

$$[\![[\![C_1; ...; C_n]\!] \Rightarrow r(f_u, ?f1); H_1; ...; H_n]\!] \Rightarrow R(F[f_u], F[?f1])$$

where f and f_u are unifiable. This theorem is used in conjunction with the transitivity theorem for R to construct a rule[4]

$$[\![[\![C_1; ...; C_n]\!] \Rightarrow r(f_u, ?f1); H_1; ...; H_n, R(F[?f1], ?G)]\!] \Rightarrow R(F[f_u], ?G)$$

Resolving this rule against the window subgoal results in a new window on the window stack with conclusion $r(f', ?f1)$ (where f' is the result of unifying f and f_u) and extra contextual hypotheses C_i'. The old window (now the second on the stack) will be $R(F[?f1], ?Goal)$. If a constructed transformation rule has any hypothesis H_i, these will appear as individual subgoals H_i' in the subgoal stack after the offset indicating the number of window openings. If a parent window has any meta level context (i.e. meta level assumptions or meta forall bound variables), this will be inherited by the subwindow through the normal Isabelle resolution mechanism.

The window opening rules resemble congruence rules in the Isabelle simplifier. However, window inference can respect relations other than meta equality. Being essentially user-directed, it can serve purposes other than simplification. Window opening rules also resemble monotonicity theorems, but are different in that the relation in a subwindow can be different to the relation in its parent window.

The prototype window inference tool is not dependent upon a specific representation for the position of subterms. The only requirement of a representation of position is that the positions can be composed to indicate the position of a deep subterm. Specific kinds of winsets can be defined which utilise different position conventions. Positions are paired with opening rules in winsets in order to index rules which provide access to specific subterms. It is the user's responsibility to ensure that the position paired with an opening rule does in fact correspond under some convention to the subterm to be transformed.

There may be many opening rules in a winset that are applicable given a particular position in a window. However there will usually be a 'best' opening rule to use, and users do no wish to exert themselves discarding weak or inappropriate opening rules. The problem of multiple opening rules is exacerbated by the presence of derived opening rules (see Section 3.5 below) that may merely be weaker versions of other explicit opening rules. Grundy's window inference package for HOL used a heuristic to select a single 'best' window opening rule to apply. The approach taken in the prototype Isabelle window inference tool follows the Isabelle subgoal package — all applicable window opening

[2] However, the scheme variable in the goal position of the window to be simplified needs to be frozen during simplification.

[3] Like Isabelle's elimination rules, the 'major' premise of an opening rule (i.e. the premise representing the subwindow) should be the first premise.

[4] Actually, in order to restrict the possible unifiers during the composition (by resolution) of opening rules for deep subterms, the right-hand side of the constructed rule is constrained to be the conclusion of the parent window's subgoal.

rules are used. Users can then backtrack over the lazy list of resulting proof states. However, it is still beneficial to present 'the most useful' opening rules first. Currently, the presentation of constructed opening rules is sorted by the the number of base opening rules used to construct them, so that rules which open 'deeper' are presented first. This heuristic for sorting is a simplified version of the heuristic for rule selection used in Grundy's tool.[8]

As many different opening rules may be applied throughout backtracking, this implementation of window inference partially addresses the issue of tree-structured proofs mentioned in [15, p53]. However, the multiple branches of backtracking here are independent of each other, so these trees are not the 'more interesting and flexible, but more difficult' trees alluded to there.

The prototype window inference commands for window transformation are:

`WIN_OPEN_TAC` *ws pos n* is a tactic which uses the transitivity theorems and opening rules in the winset *ws* to open a window at position *pos* in subgoal number *n*.

`win_open` *pos* is a tactic which opens at position *pos* of the top window.

`wo` *pos* is equivalent to `by(win_open pos)`.

3.4 Window Closing

A window with conclusion $r(f', ?SubGoal)$ is closed by resolving against the subgoal with the reflexivity theorem for r. This solves the subgoal and instantiates $?SubGoal$ to f'. Its parent window would have been $R(F[?SubGoal], ?Goal)$, and so we are left with a transformed parent window $R(F[f'], ?Goal)$. Note that we can complete a window proof by closing the initial window, because this instantiates the top scheme variable and solves the last subgoal on the window stack.

The prototype window inference commands for window transformation are:

`WIN_CLOSE_TAC` *ws n* is a tactic which closes the window at subgoal *n*, using reflexivity theorems in winset *ws*.

`win_close` *()* is a tactic which closes the top window.

`wc` *()* is equivalent to `by(win_close())`

3.5 Rule Re-Use

Grundy [8] proposes various ways of implementing rule re-use, including using base rules, relation strength and lifted relations.[5] The Ergo theorem prover's window inference facility uses relation strength, relation inverses, and also provides a wide variety of possible ways to matching against rules.

The present prototype window inference tool in Isabelle only provides relation strength facilities for opening rule re-use. Given a relation strength theorem $R(a, b) \Rightarrow R'(a, b)$, the opening rule

$$[\![r(f, f'); H_1; ...; H_n]\!] \Rightarrow R(F[f], F[f'])$$

can be re-used as:

$$[\![r(f, f'); H_1; ...; H_n]\!] \Rightarrow R'(F[f], F[f'])$$

We could similarly resolve against the major assumption, but this has not been implemented in the prototype tool.

[5] Grundy also rightly considers opening rule composition (for opening at deep subterms) a form of rule re-use. However, this paper presents it as integral to the window opening facility, as it depends upon the convention for specifying the position of subterms.

3.6 Window Side Conditions, and Windows on Context

The form of the various side conditions H_i referred to in previous subsections has not been constrained. In particular, they have not been constrained to resemble windows. It would perhaps be desirable to be able to prove these side conditions within the window inference framework. Indeed, in other implementations of window inference, such a a uniform interface is the norm. This facility is not supported in the current prototype.

Another feature of the window inference style of reasoning is the ability to open windows on contextual information. This allows users to transform existing contextual information, or to derive new contextual information. However, this facility is not supported in the current prototype.

It should be possible to support both of these activities in a slightly more sophisticated prototype. Such a prototype would require both a logic-generic way of satisfying a subgoal (or transforming a meta level assumption) by transforming a window to true, and support for opening windows at places other than in the focus of the top window of the stack.

3.7 Winsets

The type `winset` has a predefined value `empty_ws`, and the following infix operations.

`addrels` *(ws,rl)* updates the winset *ws* by adding to it the theorems in the list of pairs of theorems *rl*. Each element of *rl* is a pair of reflexivity and transitivity theorems for a relation.

`addopens` *(ws,ol)* updates the winset *ws* by adding to it the opening rule information in *ol*. Each element of *ol* is a pair containing an opening rule and a specification of the position of the subterm to be transformed. Unknown opening rules derivable from relation strength information present in the winset are also added to the winset.

`addstrongs` *(ws,sl)* updates the winset *ws* by added to it the relation strength theorems in *sl*. Unknown opening rules derivable from existing opening rules and the theorems in *sl* are also added to the winset.

4 Case Study: Program Refinement

This case study has been kept trivial for the purposes of highlighting the separation of derivation and proof of side conditions. Carrington et. al. [5] highlights the importance of the separation of concerns between program transformation and the proof of side conditions in the refinement methodology. Window inference is naturally suited to supporting this aspect of the refinement methodology.

The refinement theory used in this case study is in the style of Agerholm's [1] theory of program semantics and verification in hol88. This kind of embedding is also used in von Wright's refinement theory and tool [17] in hol88 and hol90. The theory here is essentially the same as in [17], but also features specification statements and logical constants in the style of [10]. The theory is presented in Appendix A.

4.1 Example

A specification statement $[p,\ q]$ is a non-executable command where p and q are pre and post conditions. (Respectively, p and q are predicates defining possible initial and allowable final states.) So, the specification statement $[\lambda u.\mathsf{True},\ \lambda u.u = 1 \vee u = 2]$ works for all initial values of the state and terminates by making the state variable equal to one of the allowable values 1 or 2.

We can refine this to some more directly executable program by transforming it using window inference. The transformation makes use of a theory of program refinement (`MRTprog.thy` is provided in Appendix A.1), and the theory of natural numbers, so we first create a dummy theory `nat_prog`:

```
> val nat_prog = merge_theories (MRTprog.thy, Nat.thy);
Building new grammar...
val nat_prog = {ProtoPure, Pure, HOL, Ord, Set, Prod, MRTprog, Lfp, Trancl,
  WF, Nat} : theory
```

4.2 Initial Window

We start the refinement by calling

```
> win_proof nat_prog MRTprog_ws "[%u. True, %u. (u=0) | (u=Suc(0))] ref ?G";
```

which starts a window inference proof in the theory `nat_prog` using the window set `MRTprog_ws` (which contains theorems provided in Appendix A.2) with the initial focus $[\lambda u.\mathsf{True},\ \lambda u.u = 0 \vee u = 1]$, relation $\sqsubseteq$ and scheme variable goal $?G$. This results in an initial window stack which in the subgoal package appears as:

```
Level 0
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. [%u. True , %u. u = 0 | u = Suc(0)] ref ?G
```

4.3 Introduce Sequential Composition

The theorem `spec_seq`

$$[?p,\ ?q] \sqsubseteq [?p,\ ?r];\ [?r,\ ?q]$$

can be used to transform this window. The command `wt spec_seq` transforms the window by resolving against it with the rule

```
[?p1 , ?r1]; [?r1 , ?q1] ref ?C ==> [?p1 , ?q1] ref ?C
```

which is automatically constructed using the argument theorem `spec_seq` and transitivity of $\sqsubseteq$. There is exactly one premise to this constructed transformation rule, so there are no side conditions.

Note that we could have instantiated the mid condition $?r$ when applying this theorem. We have instead chosen to leave it uninstantiated for the moment. We can either explicitly (using `read_instantiate_tac`) or implicitly (through the application of some theorem) instantiate this scheme variable at any later stage in the proof. It also worth noting that Isabelle's support of meta variables should allow the development of a refinement tool incorporating ideas like those presented in [12].

The resulting window stack is represented in the subgoal package as:

```
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. [%u. True , ?r2]; [?r2 , %u. u = 0 | u = Suc(0)] ref ?G
```

4.4 Open on the First Command

In order to transform the first branch of the sequential composition, we first restrict our attention to it by performing a window opening at that position by calling the command `wo In_seq_left`. `In_seq_left` is an ML variable whose value and type is `[Rator, Rand] : pos list`. The datatype

`pos = Rand | Rator | Body` is used to describe the position convention for the winset used here. This kind of position specification is useful because it is readily compositional (using list append) and could be used as a basis for mechanically extracting subterms.

Using information in the top window, the theorem

$$?c1 \sqsubseteq ?c'1 \Rightarrow ?c1;\ ?c2\ \sqsubseteq ?c'1;\ ?c2$$

is selected as an appropriate opening rule from the rules in the current winset. This rule, the transitivity of $\sqsubseteq$, and the current subgoal, are implicitly used to construct a rule:

```
[| ([%u. True , ?r2] ref ?c'1); (?c'1; [?r2 , %u. u = 0 | u = Suc(0)] ref ?G) |] ==>
  [%u. True , ?r2]; [?r2 , %u. u = 0 | u = Suc(0)] ref ?G
```

which is used to resolve against the window. There are exactly two premises to this constructed opening rule, so there are no side conditions.

This results in the window stack represented in the subgoal package as:

```
Level 2
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. [%u. True , ?r2] ref ?c'4
 2. ?c'4; [?r2 , %u. u = 0 | u = Suc(0)] ref ?G
```

There are now two subgoals, because we have opened our initial window once. The top of the window stack is the first subgoal. Although we won't close the top window at this stage, note that if we did so by applying reflexivity of $\sqsubseteq$, the scheme variable `?c'4` would be instantiated to `[%u. True, %u. u = 0 | u = Suc(0)]`, resulting in a subgoal stack with one subgoal identical to the one prior to opening.

4.5 Introduce Assignment

We now wish to transform the specification statement in the focus to an assignment statement. The basis of our transformation is the theorem `spec_assign`:

$$\forall u.?p(u) \rightarrow ?q(?e(u)) \Rightarrow [?p,\ ?q]\ \sqsubseteq\ \mathsf{assign}(?e)$$

Using the transitivity of $\sqsubseteq$, we can implicitly derive a transformation theorem:

```
[| ! u. ?p1(u) --> ?q1(?e1(u)); assign(?e1) ref ?C |] ==>
  [?p1 , ?q1] ref ?C
```

Note that there is an extra premise, which will become a side condition in the window stack.

Calling `wt spec_assign` would leave the assignment statement $?e$ uninstantiated. For our purposes we wish to fix it, so we call `SPEC_ASSIGN "%u. 0"`[6] where `SPEC_ASSIGN` is an auxiliary function which transforms the top window by instantiating the assignment expression $?e$ in `spec_assign` with the certified term constructed using the string argument and the signature of the current proof state.

The resulting window stack is:

```
Level 3
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. assign(%u. 0) ref ?c'4
 2. ?c'4; [?r2 , %u. u = 0 | u = Suc(0)] ref ?G
 3. ! u. True --> ?r2(0)
```

[6] i.e. $u := 0$ in a state with a single variable u.

Note that the first two subgoals are part of the window stack (as we have opened once), while the third subgoal is an as yet unproved side condition. Note also that the side condition involves the still uninstantiated scheme variable $?r$ which represents the mid-condition.

4.6 Close back to the Sequential Composition

The command `wc()` uses the reflexivity of $\sqsubseteq$ to solve the top subgoal and instantiate the scheme variable in its range (`?c'4`) to the term in its domain `assign(%u. 0)`. This results in the window stack:

```
Level 4
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. assign(%u. 0); [?r2 , %u. u = 0 | u = Suc(0)] ref ?G
 2. ! u. True --> ?r2(0)
```

4.7 Open on the Second Command

The command `wo In_seq_right` will open in the second position of the sequential composition. However, this window opening uses the theorem $[\![c2 \sqsubseteq c2'; \mathsf{monotonic}\ c1]\!] \Rightarrow c1;\ c2 \sqsubseteq c1;\ c2'$. This contains an extra premise which becomes a side condition in the resulting window stack:

```
Level 5
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. [?r2 , %u. u = 0 | u = Suc(0)] ref ?c'9
 2. assign(%u. 0); ?c'9 ref ?G
 3. ! u. True --> ?r2(0)
 4. monotonic(assign(%u. 0))
```

Here the first two subgoals constitute the window stack, with the remaining two subgoals being as yet unproved side conditions.

4.8 Introduce Skip

The command `wt spec_skip` uses the theorem $[?p,\ ?p] \sqsubseteq \mathsf{skip}$ to transform the top window. There are no new side conditions, but the use of this rule does instantiate the mid condition $?r$ to `(%u. u = 0 | u = Suc(0))`. This finally fixes the assignment side condition.

```
Level 6
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. skip ref ?c'9
 2. assign(%u. 0); ?c'9 ref ?G
 3. ! u. True --> 0 = 0 | 0 = Suc(0)
 4. monotonic(assign(%u. 0))
```

4.9 Close back to the Sequential Composition

The command `wc()` will close back to the top window, resulting in the window stack:

```
Level 7
[%u. True , %u. u = 0 | u = Suc(0)] ref ?G
 1. assign(%u. 0); skip ref ?G
 2. ! u. True --> 0 = 0 | 0 = Suc(0)
 3. monotonic(assign(%u. 0))
```

4.10 Finish the Refinement

We can now finish the refinement by closing the top window. The command `wc()` will use the reflexivity of $\sqsubseteq$ to solve the top window's subgoal, and instantiate the top goal's scheme variable $?G$, indicating that the refinement has been completed. At this stage, there are still two side conditions:

```
Level 8
[%u. True , %u. u = 0 | u = Suc(0)] ref assign(%u. 0); skip
 1. ! u. True --> 0 = 0 | 0 = Suc(0)
 2. monotonic(assign(%u. 0))
```

We can leave this as a conditional refinement by calling `topthm()` to get:

$$[\![(\forall u.\mathsf{True} \rightarrow 0 = 0 \vee 0 = 1); \mathsf{monotonic}(\mathsf{assign}(\lambda u.0))]\!] \Rightarrow [\lambda u.\mathsf{True},\ \lambda u.u = 0 \vee u = 1] \sqsubseteq \mathsf{assign}(\lambda u.1);\ \mathsf{skip}$$

Alternatively, we can prove the side conditions. The trivial assignment side condition could be automatically be proved by the command `by(fast_tac HOL_cs 1)`, and the monotonicity side condition could then automatically proved using the command `by(simp_tac MRTprog_mono_ss 1)`. The monotonicity simpset `MRTprog_mono_ss` will prove the monotonicity of all normal commands, and in particular contains the theorem $\mathsf{monotonic}(\mathsf{assign}(?e))$.

5 Discussion

This implementation of window inference is not complete (future work is outlined in the next section), but nonetheless it is sufficiently complete to compare it to other implementations, and to use it as a platform to raise some interesting issues.

5.1 Backwards and Forwards Window Inference

The development of window inference here contrasts to Grundy's [9] in that this implementation is more like backward proof, while Grundy's is more forward in nature. Grundy uses reflexivity to justify an initial theorem which is associated with a window. The window theorem is transformed in a forward manner using transitivity, and closing rules (rather than opening rules) are used to justify the transformation of parent windows based upon the resulting theorems proved in child windows.

The scheme variables present in Isabelle facilitate the backwards style of implementation given here. Scheme variables are instantiated across subgoals, allowing the implicit transformation of parent windows and side condition subgoals.

5.2 Relations: Equivalence, Preorder, Transitive, ...?

Window inference was originally proposed to use only equivalence relations [15]. A later implementation allowed for preorders [7], and recently Robinson has suggested that reflexivity need not be a condition of relations used in window inference.

Reflexive relations are not needed for window inference. In the setting described above, a window $R(F, ?G)$ can been transformed to a window $R(F', ?G)$ by the use of a theorem $R(F, F')$. This was done by using the transitivity of R to implicitly construct a theorem $R(F', ?G) \Rightarrow R(F, ?G)$. We then used the reflexivity of R to close the resulting window; satisfying the subgoal, and instantiating $?G$ to F'. However, to close a window without using reflexivity, it is enough to use the raw transformation theorem $R(F, F')$ on the initial window; satisfying the subgoal, and instantiating $?G$ to F'. If we wanted to close the window after not performing any window transformation in the subwindow, then the subwindow needn't have been opened in the first place!

The only remaining condition on relations is that they must be transitive. Do window inference relations need to be transitive? Perhaps surprisingly, in general the answer is 'no'; although the relation in the initial window of a window proof must be transitive if more than one change to the top term is to be justified. Transitivity is currently used both to justify the transformation of a parent window by the transformation of a subwindow, and also to transform each subwindow. However, this latter use of transitivity is unnecessary — similarly to the technique for avoiding the need of reflexivity described above, a subwindow could be transformed once, thus immediately closing the subwindow, and the same opening rule could be applied again to re-open a new subwindow at the same position. Although to the user it would appear that they were repeatedly transforming subterms, they would in fact merely be performing individual transformations of different terms in the same position of a repeatedly transformed parent term.

Using transitivity and reflexivity simplifies the development of a window inference tool, and as most of the relations considered in the immediate application of the tool are preorders[7], these conditions on relations have remained in the prototype tool.

5.3 Extended Notions of Context

The use of window inference for reasoning about program logics places new demands on window inference, and especially on the notion of context. For example, when we open inside the 'then' branch of a conditional statement

$$\mathsf{if}\ (\lambda s.s = 1)\ \mathsf{then}\ \mathsf{assign}(\lambda s.s + 1)\ \mathsf{else}\ \mathsf{assign}(\lambda s.2)\ \mathsf{fi}$$

we would like to know that the guard $(\lambda s.s = 1)$ is true. However, this essentially modal hypothesis is not true in all contexts — it is only true in the context of the 'then' branch of the conditional. The guard is of the wrong type to be a meta-level assumption, and we cannot say that it is true of all s, because that would be equivalent to False.

Part of this problem arises because we have followed Grundy in placing contextual hypotheses behind derivation rather than implication. The kinds of context structuring proposed in [5] provide access to both sorts of hypotheses, by using derivability and modal object logic implications. So, instead of a window transformation being represented by a proof of $H \vdash R(F, F')$, it could be represented by a proof of $H \vdash H_c \Rrightarrow R(F, F')$, with contextual hypotheses like the guards of conditionals present as the conjunction of terms in H_c.

The context structuring suggested in [5] would provide many different forms of windows. Windows of each form would have access to different kinds of contextual hypotheses, and the transformation of each kind of window would be represented by the proof of different kinds of formulae. This idea could be implemented in the style of window inference describes here by having something like winsets local to specific windows (and hence referred to within window opening rules), rather than having a winset global to all windows in a window proof. Winsets would then not only describe reflexivity

[7] One notable exception arises in the proof of loop termination for imperative programs, where conditions arise involving the transitive irreflexive relation $<$.

and transitivity theorems and opening rules, but would also describe the structure and meaning of various kinds of context.

6 Conclusions and Future Directions

The work so far constitutes a design prototype for window inference in Isabelle. While it is complete enough to demonstrate that Isabelle's meta variables and meta logic provide an elegant basis for the construction of a useful tool, it does not provide all the sophistication of a complete tool.

Obvious areas for improvement are the provision of:

- An alternative interface to the subgoal package. This should at least hide the scheme variables in the range of windows, hide lower levels of the window stack, and provide easier access to side conditions. It may also provide a framework for heuristics in the style of the Ergo theorem prover.
- Better heuristics for the sorting of the presentation of multiple possible opening rules.
- A facility to open windows on the context of a window.
- A GUI with selectable subterms for window opening.
- A framework for motivational goals and goal directed tactics.
- A method for manipulating lemmas and unused conjectures.

More fundamental changes could include:

- dropping the reflexivity requirement on relations, and
- implementing more sophisticated notions of context in the manner of [13]. This could be done by providing access to different sorts of context behind object level implication and meta level implication, and by providing winset-like facilities at the level of windows and not window stacks.

7 Acknowledgements

Thanks to Jim Grundy for commenting on this paper, and to Donald Syme and John Staples for proof reading an earlier draft.

References

[1] S. Agerholm. Mechanizing program verification in HOL. Master's thesis, Computer Science Department, Aarhus University, April 1992.

[2] R.J.R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, February 1981.

[3] M. Butler, T. Långbacka, and R. Rukšėnas. *Refinement Calculator Tutorial and Manual*, April 20, 1995.

[4] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Requirements for a program refinement engine. Technical Report 94-43, Software Verification Research Centre, The University of Queensland, November 1994.

[5] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A review of existing refinement tools. Technical Report 94-8, Software Verification Research Centre, The University of Queensland, February 1994.

[6] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[7] J. Grundy. Window inference in the HOL system. In M. Archer and et. al., editors, *Proceedings of the International Tutorial and Workshop on the HOL Theorem Proving System and its Applications*, pages 177–189, Los Alamitos, California, 1991. IEEE Computer Society Press.

[8] J. Grundy. *A Method of Program Refinement*. PhD thesis, Computer Laboratory, University of Cambridge, 1993. Also available as TR318.

[9] J. Grundy. Transformational hierarchical reasoning. June 1995.

[10] C. Morgan. *Programming from Specifications*. Prentice-Hall International, 1990.

[11] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[12] R.G. Nickson and L.J. Groves. Metavariables and conditional refinements in the refinement calculus. Technical Report 93-12, Software Verification Research Centre, The University of Queensland, 1993.

[13] R.G. Nickson and I. Hayes. Program window inference, 1994.

[14] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[15] P. J. Robinson and J. Staples. Formalising the hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1), February 1993.

[16] M. Utting and L. Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, The University of Queensland, February 1994.

[17] J. von Wright. Program refinement by theorem prover. In D. Till and R.C.F. Shaw, editors, *Sixth Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1994.

A Small Program Refinement Theory

No recursion or looping constructs have been defined in this theory, but this could done as in [1].

A.1 Theory Definition

```
MRTprog = Prod +

types
 'a           pred    = "'a => bool"
 ('a, 'b)     ptrans  = "'b pred => 'a pred"
 ('a, 'b, 'c) eptrans = "('a * 'c, 'a * 'b) ptrans"

consts
 refines_to   :: "[('a,'b) ptrans, ('a,'b) ptrans] => bool \
\                 "                                    ("(_ ref/ _)" [7,8] 60)

 monotonic    :: "('a,'a) ptrans => bool"

 seq          :: "[('a,'b) ptrans, ('b,'c) ptrans] => ('a,'c) ptrans \
\                 "                                      ("(_;/ _)" [9,8] 50)
 abort        :: "('a,'b) ptrans"
 assert       :: "'a pred => ('a,'a) ptrans"                    ("{_}" [8] 9)
 guard        :: "'a pred => ('a,'a) ptrans"                    ("[_]" [8] 9)
 spec         :: "['a pred, 'a pred] => ('a,'a) ptrans" ("[(_ ,/ _)]" [0,0] 52)
 assign       :: "('a => 'b) => ('a,'b) ptrans"
 skip         :: "('a,'a) ptrans"
 cond         :: "['a pred, ('a,'b) ptrans, ('a,'b) ptrans] => ('a,'b) ptrans \
\                 "                      ("(if _/ then/(3 _)/ else/(3 _) /fi)")
 Con          :: "('a => ('b,'b) ptrans) => ('b,'b) ptrans" (binder "con " 10)
 block        :: "('a,'b,'b) eptrans => ('b,'b)ptrans"


defs
refines_to_DEF "c1 ref c2     == (!q u. c1(q,u) --> c2(q,u))"

monotonic_DEF  "monotonic(f)  == (!p q.(!u.p(u)-->q(u))-->(!u.f(p,u)-->f(q,u)))"

abort_DEF      "abort         == (%q u. False)"
skip_DEF       "skip          == (%q u. q(u))"
assert_DEF     "{p}           == (%q u. p(u) & q(u))"
guard_DEF      "[b]           == (%q u. b(u) --> q(u))"
spec_DEF       "[p, q]        == (%r u. p(u) & (!v. q(v) --> r(v)))"
seq_DEF        "c1; c2        == (%q. c1(c2(q)))"
assign_DEF     "assign(e)     == (%q u. q(e(u)))"
cond_DEF       "cond(g,c1,c2) == (%q u. (g(u) & c1(q,u)) | (~g(u) & c2(q,u)))"
con_DEF        "Con(C)        == (%q u. ? x. C(x,q,u))"
block_DEF      "block(c)      == (%q u. ! x. c(%u. q(snd(u)), <x,u>))"

end
```

A.2 Theorems

```
(* Transformation Theorems: *)

spec_assign "! u. ?p(u) --> ?q(?e(u)) ==> [?p , ?q] ref assign(?e)"
spec_seq    "[?p , ?q] ref [?p , ?r]; [?r , ?q]"
spec_skip   "[?p , ?p] ref skip"
spec_block  "[?p , ?q] ref block([%u. ?p(snd(u)) , %u. ?q(snd(u))])"
spec_con    "! u. ?p(u) --> (? x. ?p'(x, u)) ==> [?p , ?q] ref con X. [?p'(X) , ?q]"

fix_initial "[?p , ?q] ref con X. [%u. X = ?x(u) & ?p(u) , ?q]"
remove_con  "con X. ?c ref ?c"
simp_con    "(con X. ?c) = ?c"

skip_seq_left_ident  "skip; ?c ref ?c"
skip_seq_right_ident "?c; skip ref ?c"
simp_skip            "?c; skip = ?c"


(* Relations --- present in MRTprog_ws *)
refines_to_refl  "?A ref ?A"
refines_to_trans "[| ?A ref ?B; ?B ref ?C |] ==> ?A ref ?C"


(* Opening Rules --- present in MRTprog_ws *)
spec_weaken     "(!!u. ?p(u) --> ?p'(u)) ==> [?p , ?q] ref [?p' , ?q]"
spec_strengthen "(!!u. ?q'(u) --> ?q(u)) ==> [?p , ?q] ref [?p , ?q']"
mono_seq_left   "?c ref ?c' ==> ?c; ?c2.0 ref ?c'; ?c2.0"
mono_seq_right  "[| ?c ref ?c'; monotonic(?c1.0) |] ==> ?c1.0; ?c ref ?c1.0; ?c'"
mono_cond_then  "?c ref ?c' ==>
                   if ?b then ?c else ?c2.0 fi ref if ?b then ?c' else ?c2.0 fi"
mono_cond_else  "?c ref ?c' ==>
                   if ?b then ?c1.0 else ?c fi ref if ?b then ?c1.0 else ?c' fi"
mono_con_body   "(!!X. ?c(X) ref ?c'(X)) ==> con X. ?c(X) ref con X. ?c'(X)"


(* Monotonicity Theorems --- present in MRTprog_mono_ss *)
mono_skip       "monotonic(skip)"
mono_assert     "monotonic({?p})"
mono_guard      "monotonic([?p])"
mono_spec       "monotonic([?p , ?q])"
mono_assign     "monotonic(assign(?e))"
mono_seq        "[| monotonic(?c1.0); monotonic(?c2.0) |] ==>
                   monotonic(?c1.0; ?c2.0)"
mono_cond       "[| monotonic(?c1.0); monotonic(?c2.0) |] ==>
                   monotonic(if ?b then ?c1.0 else ?c2.0 fi)"
mono_con        "(!!x. monotonic(?c(x))) ==> monotonic(con X. ?c(X))"
```