Transformational reasoning with incomplete information

O. Celiku and J. von Wright

Åbo Akademi University and Turku Centre for Computer Science,
Lemminkäisenkatu 14 A, 20520, Turku, Finland
{oceliku,jwright}@abo.fi

Abstract. Starting a proof without having complete information about the proof term can be beneficial. While the proof is carried out newly introduced constraints can make the information more precise. Window inference, a proof paradigm based on hierarchical term rewriting, is very well suited for general transformational reasoning and especially for reasoning about programs. The HOL implementation of window inference does not support proofs with uninstantiated terms since the HOL system does not have a formalized metalogic. We show how, without having to redo proofs, higher order variables (metavariables) can be used to perform proofs with uninstantiated terms in HOL window inference. We illustrate the uses of metavariables with a few examples related to program reasoning.

1 Introduction

In a metalogic, variables range over terms of an object logic. These metavariables can be used to describe schemes or inference rules of the object logic, or in general to perform generic derivations in the object logic. Their ability to generalize proofs can be used to perform proofs even without having complete information about the proof term.

The main motivation behind using metavariables in problem solving to stand for unknown information is that as we move toward a solution, we get more information about the problem and may be able to make the information more precise. The spectrum of their use is quite wide: they can be used in proof terms, proof rules, proof trees, etc. Here we will be particularly interested in their use in program development.

Window inference is a proof paradigm where a theorem is proved by stepwise transformation of an initial term, including transformations that change subterms while taking the context of these subterms into account. Window inference was originally developed for mathematical reasoning with equivalence relations, but it has proved a powerful method for general logical reasoning, and in particular for reasoning about correctness and refinement of programs.

Grundy formalized window inference in higher-order logic and implemented a tool [7] for the HOL theorem proving system [6]. His system translates proof steps of window inference to corresponding proof steps in higher-order logic, giving a guarantee that every proof is sound. Staples [15] has implemented window inference in the Isabelle system [12], in a "lightweight" way that reuses much of the built-in theorem-proving infrastructure of Isabelle. The Ergo system [2], a generic proof tool, has window inference as its main theorem proving paradigm.

Isabelle's scheme variables and Ergo's metalogical variables make it easy to use uninstantiated terms in derivations performed in their respective implementations of window inference. HOL has no support for metalogical variables, but we show how higher-order object-logic variables can be used to "simulate" metavariables. However, this does not fit well with the implementation of window inference, which works with a stack of windows, each characterized by a theorem describing how a subterm has been transformed and on which the transformation of the parent terms depends. This means that instantiating a term leads to a change in all window theorems of the stack in question. The procedure would be expensive and it would violate the basic idea of a stack (that only the top element can be accessed).

In this paper, we illustrate how derivations can be carried out with the HOL window inference system, as if one had access to metavariables. To simulate metavariables, we use higher order (object-logic) variables, with special naming conventions, to stand for unknown information. As the derivation proceeds and we get more information we can instantiate these variables by introducing constraints on them. These constraints become assumptions in the final theorem, but if they define the metavariable uniquely, then they can be discharged.

2 Metalogical variables

A metalogic is used to specify and reason about object logics. Variables in a metalogic (which stand for object-level terms) will from now on be called *metalogical variables*, and we will use the term *metavariables* for higher-order object-level variables used to stand for information that is to be instantiated later.

Metalogical variables are supported in several theorem proving systems, for example COQ [5], Ergo [2], Isabelle [12], Oyster [9]. In such systems, formal derivations with metalogical variables are possible, and instantiating a metalogical variable at some point during the derivation by a term of the object logic does not require redoing the proof.

2.1 Uses in program development

Examples of the use of metalogical variables are numerous. Here we concentrate on examples from program development.

Nickson and Groves [11] show how using metalogical variables in program refinement can help in separating the concerns of choosing the right rules to apply, and filling in the details of the proofs. Following the goal-directed approach, they suggest that rules, suggested by some pattern of the specification, be applied without instantiating all their scheme variables. In fact, the scheme variables that are meant here are the parametric ones – i.e., those that cannot be determined during pattern matching, but whose instantiation affects the result. The applicability conditions of the rules, if they contain metalogical variables, are added to the constraints on the later binding of the metalogical variables. As the development process continues, the constraints on the binding of metalogical variables make it easier to determine appropriate instantiations of these variables.

Typical examples of situations where it is desirable to apply refinement rules without instantiating all their scheme variables are development of guarded commands, such as the conditional if statement, the do loop statement etc., development of a sequence of commands etc. In the guarded statements case, it usually helps to determine the *guard* at a later stage, while in the sequence case the variable one would usually want to instantiate later is the *intermediate assertion*.

Another benefit from using metalogical variables in program refinement is that derived refinement rules can be developed [11]. If a refinement is performed on an initial specification that contains metalogical variables, once the refinement is over, the final theorem expressing the refinement of the initial specification by the final refinement, with constraints on the binding of the metalogical variables added as assumptions, will also contain metalogical variables. Such theorems can be used as refinement rules.

When proving programs correct with respect to a precondition and a postcondition, metalogical variables can be used to develop loop invariants incrementally [18]. We can start by an approximation of the loop invariant conjuncted to a metalogical variable which can be instantiated step by step, and whose instantiation strengthens our initial approximation of the invariant. Similarly, for weakening invariants metalogical variables can be used in disjunction to the initial approximations.

Basin et al. [1] note that in constructive programming, the activities of program synthesis and verification can be unified by allowing metalogical variables to appear in actual proofs and proof rules. When the term whose proof we are looking for is a ground term of the type theory in question, we have verification, when it is a metalogical variable, we have synthesis, and when it is a combination of both we have some hybrid system. The same kind of proof rule is used to build proofs regardless of what the case is.

3 Window inference in HOL

Window inference is well suited for reasoning about programs, and it has been used as a basis for the construction of several refinement tools [3, 16, 4]. We are interested in using the HOL-based Refinement Calculator [3] for program reasoning with metalogical variables.

In this section we briefly describe window inference and its implementation in the HOL system.

3.1 Window inference

Window inference is a method for doing transformational proofs, originally described by Robinson and Staples [14] and later extended by Grundy [7]. An *initial focus* E_0 is transformed step by step while preserving a preorder (i.e., a reflexive and transitive relation) R. The result of the sequence of transformations (called a derivation)

$$E_0 \; R \; E_1 \; R \cdots R \; E_n$$

is (by transitivity of R) that $E_0 \; R \; E_n$ holds.

A major feature of window inference is that we can at any time focus our attention on a subterm of the current focus (this is called *opening a window* on the subterm). The subterm is transformed as described above (in a *subderivation*) and when we *close the subwindow*, the result of the subderivation is substituted for the term that we focused our attention on.

The opening and the closing of subwindows is governed by *window rules* of the form

$$\frac{\Gamma' \vdash e \; r \; e'}{\Gamma \vdash E[e] \; R \; E[e']}$$

where r and R are preorders, $E[e]$ is a term with e occurring as a subterm and $E[e']$ is the same term but with e' substituted for e and Γ and Γ' are sets of formulas. Logically, this is an inference rule in a sequent formulation of a logic (in our case higher-order logic). As a window rule, it should be read starting in the lower left corner. The rule can be applied in a situation when we are preserving the relation R and focusing on the subterm e of the current focus $E[e]$ and when the assumptions that we are working under are Γ. The rule then states that we may assume Γ' and transform e while preserving r in the subderivation.

An example of a window rule is the following:

$$\frac{\Gamma, t_2 \vdash t_1 \Rightarrow t_1'}{\Gamma \vdash (t_1 \wedge t_2) \Rightarrow (t_1' \wedge t_2)}$$

3.2 HOL window inference

Grundy has implemented window inference [7] as an interface to the HOL system. The terms that are being transformed by window inference are HOL terms, and the inference steps are reduced to inference steps in the secure underlying HOL logic.

A derivation is handled by using a *stack of windows* in HOL's metalanguage. When a stack is created, a focus E, a relation to be preserved R, and a set of hypotheses Γ are entered into the first window in the stack. The window is initially characterized by the $\Gamma \vdash E \; R \; E$ theorem (recall R is a preorder, and this fact can be looked up in a database of relations that are proved to be preorders).

A transformation is applied by specifying a theorem in the HOL logic of the form $\Phi \vdash t \; R \; t'$ where E matches t and a subset of Γ matches Φ. The system then deduces $\Gamma \vdash E\ R\ E'$ and this becomes the new window theorem. After a transformation of the form $\Gamma \vdash E'\ R\ E''$ the window theorem is by transitivity $\Gamma \vdash E\ R\ E''$ and so on.

When focusing on a subterm e of the current focus, a suitable relation r and suitable assumptions Γ are produced from the chaining of suitable window rules and the *subwindow* is pushed onto the stack. The new focus can now be transformed, focused upon, etc. When a subwindow is closed at a point when the window theorem is $\Gamma \vdash e\ r\ e'$, the subwindow is popped from the stack and the window rule is applied to make $\Gamma \vdash E[e]\ R\ E[e']$ the window theorem of the parent window.

If assumptions are added in a transformation step, they need to be discharged by a proof. Assumptions of this kind are called *conjectures*, although logically they are not any different from the rest of the assumptions.

3.3 Metalogical variables and window inference

Staples's implementation [15] of window inference in Isabelle allows for a rule to be applied without having had all its scheme variables instantiated. This implementation makes use of Isabelle's scheme variables, which, once instantiated, are instantiated across subgoals (the implementation is in the backward proof style) implicitly updating the parent subgoals. Isabelle's scheme variables are logically equivalent to free metalogical variables, but they can be instantiated during unification.

The Ergo theorem prover [2] has window inference as its main proof paradigm. Ergo is implemented in Qu-Prolog [13], whose variables can be used as metalogical variables in proofs. Qu-Prolog's variables can be instantiated during unification, but they can also be frozen so that the user can have control over their instantiation, i.e., so that instantiation occurs only via explicit system commands. During proofs, new metalogical variables that did not appear in the original statement can be introduced, and these variables can be subject to freezing or thawing.

The HOL system does not have a formalized metalogic, and as a result no built-in support for metalogical variables. As we will show, metavariables (higher-order object-level variables) can be used instead, but even then, we would need to change all the window theorems of the stack in question once a metavariable was instantiated. This means that we must make a pass through the whole stack, updating every window and proving the correctness of the instantiation.

4 Adding metavariables to HOL window inference

Metalogical variables that stand for unknown pieces of terms are also known as place-holders, holes, etc. They have been theoretically approached and several calculi have been developed to deal with them [8, 10, 17]. It is not our intention to develop another such calculus. Our intention is to show a simple way of allowing for derivations with uninstantiated terms in the HOL implementation of window inference.

When metalogical variables are simulated by higher-order HOL variables, instantiation leads to the introduction of a conjecture rather than to redoing earlier parts of the proof. A conjecture of the form $X = t$, where X is the metavariable in question and t its instantiation, is introduced at the moment of instantiation. The proof is carried out using this conjecture, and the final window theorem will have $X = t$ as one of its assumptions, but X will not be free in the conclusion of the theorem.

A conjecture can be discharged in window inference only if it is proved, so the final window theorem will have $X = t$ as one of its assumptions. However, as a derivation is finished, we can step outside the window inference system and discharge the assumption. The result is a theorem without any occurrences of the metavariable X. This "trimmed" theorem, although not a window theorem, will be as useful as the window theorem that we could produce if the proof was redone. Since both the window inference and this final trimming is reduced to HOL inferences, we cannot produce false theorems (although the proof can fail if we try to do steps where side conditions are violated).

4.1 A small example

One common use of metavariables in general logic reasoning is when introducing Skolem functions in order to eliminate existential quantifiers. A simple derivation, performed in HOL window inference, follows. Note that we are working with pure HOL terms and theorems (no metalogical variables).

We would like to prove (by transforming to truth under backward implication)

$$\forall x \cdot \exists y \cdot y > x$$

First we focus on

$$\exists y \cdot y > x$$

The existential quantifier can be eliminated if we give a witness for y. Instead of the concrete witness we give as witness the application of metavariable Z to x. Z is an object-level function variable which is made dependent on x so that variable capturing can be avoided when Z is instantiated. The focus becomes

$$Z(x) > x$$

and the window theorem

$$\vdash (\exists y \cdot y > x) \Leftarrow (Z(x) > x)$$

A possible instantiation of Z is $(\lambda z \cdot z + 1)$ and after instantiating and performing beta reduction the focus becomes

$$x + 1 > x$$

and the window theorem

$$(Z = (\lambda z \cdot z + 1)) \vdash (\exists y \cdot y > x) \Leftarrow (x + 1 > x)$$

We simplify the focus to T, and close the subwindow. Note that the conjecture about Z does not cause any problem when closing the window since no variable capturing occurs. The new focus is T and the window theorem

$$(Z = (\lambda z \cdot z + 1)) \vdash (\forall x \cdot \exists y \cdot y > x) \Leftarrow \mathsf{T}$$

Since Z is not free in the conclusion of the theorem, we can get rid of the assumption about Z as follows:

1. $(Z = (\lambda z \cdot z + 1)) \vdash (\forall x \cdot \exists y \cdot y > x) \Leftarrow \mathsf{T}$
2. $\vdash (Z = (\lambda z \cdot z + 1)) \Rightarrow ((\forall x \cdot \exists y \cdot y > x) \Leftarrow \mathsf{T})$ by (1), discharge.
3. $\vdash \forall Z \cdot (Z = (\lambda z \cdot z + 1)) \Rightarrow ((\forall x \cdot \exists y \cdot y > x) \Leftarrow \mathsf{T})$ by (2), generalize.
4. $\vdash (\forall x \cdot \exists y \cdot y > x) \Leftarrow \mathsf{T}$ by (3), one-point rule.

4.2 Technical considerations

The same steps as illustrated in the example above are always taken when a conjecture about a metavariable is being discharged. These steps can be put together to form a more general discharging inference rule, which can be automatically repeated in order for a list of such conjectures to be discharged. When implementing such a rule and, in general, when providing support for metavariables, several decisions should be taken.

Variable capturing Variable capturing can occur if the metavariable appears in a subterm of a quantified or lambda abstracted term. The metavariable must then be a function of the quantified variable, to allow proper subwindow closing.

For example, consider

$$\forall x \cdot Z \geq x$$

If the current focus was $Z \geq x$, we could choose to instantiate Z by x. The result would be the window theorem

$$Z = x \vdash (Z \geq x) \Leftarrow (x = x)$$

However, if we try to close the window, the window inference will not let us produce an invalid theorem of the form

$$Z = x \vdash (\forall x \cdot Z \geq x) \Leftarrow (x = x)$$

In order to avoid variable capturing the metavariable should have been introduced as $Z(x)$ instead of just Z, that is, Z should be a function of x.

In general, a metavariable is potentially dependent on all variables that are free in the subterm where the metavariable is introduced. For example, metavariables that are introduced when reasoning about programs are dependent on the program state (i.e., on the program variables).

A simple solution which makes automatic typing of metavariables possible is to make each metavariable a function of all the free variables of the focus term in question. This solution would make metavariables unnecessarily dependent on variables that are free even in the initial focus. However, the price for a more careful inspection of constraints is high since that would require the examination of the whole stack. Another possibility is that the user makes such constraints on the metavariables explicit, when introducing them.

Order of discharging The discharging order is important when metavariables depend on each other. For example, if we want to discharge $X = t[Y]$ and $Y = t'$ (where X is not free in t') we must discharge $X = t[Y]$ before discharging $Y = t'$. Indeed, if we attempt to deal first with $Y = t'$, the *generalize* step will fail. In general, metavariables that are not depended on should be discharged first.

Conjectures other than those related to the instantiation of the metavariables can also depend on metavariables. These conjectures should also be inspected when the dischargeability of a metavariable is being considered.

Circular dependence Since metavariables can depend on each other, circular dependence can occur. Circular dependence would make it hard for the conclusion of a theorem to become free of the metavariables and would make the discharging of the instantiation conjectures impossible.

The detecting of circular dependence can be performed either during instantiation or during conjecture discharging. Detecting circular dependence during instantiation would require an analysis of dependences every time a metavariable is instantiated, while detecting it when discharging conjectures would make the part of the proof from instantiation on useless, and thus it would be necessary to go back to the moment of instantiation in order to be able to produce a meaningful theorem.

4.3 Implementation

We have implemented a general inference rule MV_ASM_DISCH that discharges a list of assumptions about instantiation of metavariables. It takes as an argument a theorem and returns a trimmed version of it if there are assumptions about metavariables that can be successfully discharged, or the theorem itself otherwise. Note that although this rule is targeted at window theorems it actually works outside window inference, and instead of establishing a new window theorem it produces a new ordinary theorem.

In the current implementation metavariables are recognized by special naming conventions, and an assumption related to a metavariable instantiation by its equational form. Such an assumption is considered dischargeable if its metavariable is not free in other assumptions, and then it is discharged as illustrated in the example above. The naming convention is needed, since there may be other assumptions of the form $x = t$ where x is not a metavariable. In fact, program variables are implemented as assumptions of this form in the Refinement Calculator.

This implementation establishes a right order of discharging by "brute force", since in each round of trimming an assumption about a metavariable is removed only if the metavariable is not depended on.

Circular dependence is detected when trying to discharge assumptions. It does not need any special treatment since neither of two mutually dependent metavariables can be considered dischargeable when the assumptions are inspected. Hence, MV_ASM_DISCH will not attempt to discharge mutually dependent metavariables.

Constraints on metavariables must be made explicit by the user, so avoiding variable capturing is the user's responsibility.

5 Examples

The examples have been carried out in the Refinement Calculator [3], which is one of the refinement tools that uses window inference as an interface to HOL. Here, the program state has been implemented as a tuple of values where every component corresponds to a program variable. To make programs more readable the projection functions are given names according to the corresponding program variables. Programs are identified with predicate transformers which are functions from state predicates to state predicates. The programming language has been shallowly embedded in the HOL logic, which means that the terms in the embedded language are identified with terms in the HOL logic by extra-logical parsing and pretty-printing functions. The shallow embedding allows the identification of types in the embedded language with types in the HOL logic. Hence, when types are mentioned they refer to HOL types.

5.1 Refining programs

As an example of the use of metavariables in program refinement we show how we derive a program that finds the greater of two numbers. This example was also used by Nickson and Groves [11]. The relation that is being preserved is *refinement* and the initial specification statement

$$z := z' | z' = \mathsf{max}(x, y)$$

where x, y, z are program variables. The specification statement is a nondeterministic assignment which specifies that z is assigned some value z' satisfying the predicate $z' = \mathsf{max}(x, y)$ which, in turn, is defined as

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

This predicate can also be regarded as the postcondition of the program.

The disjunction in the postcondition suggests that our specification can be refined by a conditional statement.

In order to introduce a conditional statement we need to provide the guard of the statement. However, we can postpone choosing the guard for when we know more about the program statements constituting the branches of the if statement. Instead of the concrete guard, we supply a metavariable B which is typed as a boolean function over the program state, i.e., $B : N \times N \times N \to Bool$.

$$\mathsf{if}\ B\ \mathsf{then}\ \{B\};\ z := z'|z' = \mathsf{max}(x, y)\ \mathsf{else}\ \{\neg B\};\ z := z'|z' = \mathsf{max}(x, y)\ \mathsf{fi}$$

The assertion statements that appear in the branches of the if statement make the context information explicit, i.e., that when focusing on the first branch of the conditional we can assume that the guard holds, and when focusing on the second branch we can assume the negation of the guard. Note that B is being applied to the program state although pretty-printed it does not appear so.

Again judging from the form of the postcondition, we proceed by refining either of the branches of the if statement by $z := x$. We focus on the first branch and refine the nondeterministic assignment by $z := x$.

$$\mathsf{if}\ B\ \mathsf{then}\ z := x\ \mathsf{else}\ \{\neg B\};\ z := z'|z' = \mathsf{max}(x, y)\ \mathsf{fi}$$

The refinement rule we just applied gives rise to the proof obligation

$$(\forall x\ y\ z \cdot B(x, y, z) \Rightarrow (x = \mathsf{max}(x, y)))$$

By using the definition of max we can simplify the proof obligation as

$$(\forall x\ y\ z \cdot B(x, y, z) \Rightarrow (x \geq y))$$

which cannot be simplified any further, unless we have more information about B. In fact, in order to prove this obligation, we make B express exactly what we want to prove.

We instantiate B to $(\lambda(x, y, z) \cdot x \geq y)$ (we could also instantiate B to $(\lambda(x, y, z) \cdot x > y))$.

$$\mathsf{if}\ x \geq y\ \mathsf{then}\ z := x\ \mathsf{else}\ \{y > x\};\ z := z'|z' = \mathsf{max}(x, y)\ \mathsf{fi}$$

$B = (\lambda(x, y, z) \cdot x \geq y)$ is added to the proof obligations. With the added information about B, the first proof obligation can be successfully discharged.

We now focus on the second branch of the conditional. Note that $\{y > x\}$ corresponds to $\{\neg B\}$. Refining the nondeterministic assignment by $z := y$ generates the following proof obligation:

$$(\forall x\ y\ z \cdot (y > x) \Rightarrow (y = \mathsf{max}(x, y)))$$

which can be easily proved by expanding the definition of max.

The final window theorem is

$$\begin{array}{l} B = (\lambda(x, y, z) \cdot x \geq y) \\ \vdash z := z'|z' = \mathsf{max}(x, y) \sqsubseteq \mathsf{if}\ x \geq y\ \mathsf{then}\ z := x\ \mathsf{else}\ z := y\ \mathsf{fi} \end{array}$$

and expresses the fact that the initial specification is refined by the implementation of the if statement provided that B is defined as $(\lambda(x, y, z) \cdot x \geq y)$.

Since B does not appear free in the conclusion of the window theorem, we can use MV_ASM_DISCH to derive a new theorem that is not conditioned on B

$\vdash z := z'|z' = \mathsf{max}(x, y) \sqsubseteq \mathsf{if}\ x \geq y\ \mathsf{then}\ z := x\ \mathsf{else}\ z := y\ \mathsf{fi}$

Recall that MV_ASM_DISCH operates outside window inference, so the new theorem is not established as a window theorem.

5.2 Verifying programs

This example shows how metavariables can be used to develop loop invariants incrementally while proving total correctness. An initial approximation of the loop invariant conjuncted to a metavariable can be used to generate verification conditions. The verification conditions are then simplified. The unproved parts usually give hints about possible strengthening of the invariant approximation, which are in fact hints about possible instantiation of the metavariable. The metavariable then is instantiated to the hinted predicate conjuncted to a fresh metavariable which is used for further strengthening the invariant. The cycle of simplification and instantiation is repeated till the instantiation of the metavariable of the turn to true can reduce the verification condition to true. Dually, metavariables disjuncted to an approximation of the invariant could be used to incrementally weaken an approximation that is too strong.

The loop divides x by y storing the quotient in q and the remainder in r

$$\mathsf{do}\ y \leq r \rightarrow r := r - y; q := q + 1\ \mathsf{od}$$

We want to prove its total correctness with respect to precondition $0 < y \wedge q = 0 \wedge r = x$ and postcondition $x = q \times y + r \wedge r < y$.

As invariant we use

$$x = q \times y + r \wedge X(x, y, q, r)$$

where X is a metavariable that will be used to strengthen the initial approximation of the invariant in case it proves too weak to prove the verification conditions. X is function of the program variables x, y, q, and r.

After reducing the correctness of the loop with r as termination function the proof obligation is

$$\begin{aligned}
&(\forall x\ y\ q\ r \cdot 0 < y \wedge q = 0 \wedge r = x \Rightarrow x = q \times y + r \wedge X(x, y, q, r)) \\
\wedge\ &(\forall y\ q\ r \cdot y < r \wedge X(r + q \times y, y, q, r) \\
&\qquad \Rightarrow r + q \times y = (r + (q + 1) \times y) - y \wedge X(r + q \times y, y, q + 1, r - y) \wedge r - y < r) \\
\wedge\ &(\forall x\ y\ q\ r \cdot r < y \wedge x = q \times y + r \wedge X(x, y, q, r) \Rightarrow x = q \times y + r \wedge r < y)
\end{aligned}$$

Simplified the verification condition becomes

$$\begin{aligned}
&(\forall y\ r \cdot 0 < y \Rightarrow X(r, y, 0, r)) \\
\wedge\ &(\forall y\ q\ r \cdot y \leq r \wedge X(r + q \times y, y, q, r) \Rightarrow X(r + q \times y, y, q + 1, r - y) \wedge r - y < r)
\end{aligned}$$

The consequent in the second conjunct shows that we need to prove $r - y < r$. The corresponding antecedent implies $y \leq r$, but we also need it to imply $0 < y$. The hint that we get is to instantiate X to $(\lambda x\ y\ q\ r \cdot 0 < y \wedge Y(x, y, q, r))$ where Y is a new metavariable. The first conjunct will not cause any problems since $0 < y$ is in its antecedent. Instantiating and simplifying we get

$$(\forall y\ r \cdot 0 < y \Rightarrow Y(r, y, 0, r))$$
$$\wedge (\forall y\ q\ r \cdot y \leq r \wedge 0 < y \wedge Y(r + q \times y, y, q, r) \Rightarrow Y(r + q \times y, y, q + 1, r - y))$$

We can see that "nothing" is left to be proved; the consequents of the both conjuncts consist of conditions on Y. Since we can control the instantiation of Y we can make these conditions true by instantiating Y to $(\lambda x\ y\ q\ r \cdot \mathsf{T})$. Both conjuncts are reduced to true and the proof is finished.

The final window theorem expresses that the correctness of the do loop with respect to the given precondition and postcondition follows from T with the conjectures about X and Y as the assumptions of the theorem. By using MV_ASM_DISCH on this theorem we can derive outside window inference a new theorem that does not have any assumption about X or Y. X will be the first to be discharged by MV_ASM_DISCH since Y does not depend on X but X depends on Y.

As a by-product we have the complete invariant $x = q \times y + r \wedge 0 < y$.

5.3 Metavariables in a program text

Our last example shows how a metavariable in a program text can be handled. Consider finding a least element in an array by a simple linear search (where i and j range over the natural numbers):

$$i, j := 0, 1; \mathsf{do}\ j < n \rightarrow \mathsf{if}\ a[j] < a[i]\ \mathsf{then}\ i, j := j, j + 1\ \mathsf{else}\ j := j + 1\ \mathsf{fi}\ \mathsf{od}$$

with precondition $n > 0$ and postcondition $i < n \wedge (\forall k < n \cdot a[i] \leq a[k])$. We are interested in finding out whether we can find a minimal element according to some partial order in the same way. Rather than asserting specific assumptions about the ordering, we simply start with a metavariable $R : \alpha \rightarrow \alpha \rightarrow \mathit{Bool}$ (standing for a relation on the type of array elements) and then we let the correctness proof show us what restrictions on R are necessary. Thus, the initial term of our derivation is a correctness assertion stating that the program

$$i, j := 0, 1; \mathsf{do}\ j < n \rightarrow \mathsf{if}\ R\ a[j]\ a[i]\ \mathsf{then}\ i, j := j, j + 1\ \mathsf{else}\ j := j + 1\ \mathsf{fi}\ \mathsf{od}$$

is correct with respect to precondition $n > 0$ and postcondition $i < n \wedge (\forall k < n \cdot \neg R\ a[k]\ a[i])$ (no other element is related to $a[i]$ by R). Using window inference (as in the preceding example), we apply transformations for reduction of correctness assertions, with loop invariant $i < j \leq n \wedge (\forall k < j \cdot \neg R\ a[k]\ a[i])$. After simplifying the verification conditions, the following two conditions remain:

$$\neg R\ a[0]\ a[0]$$
$$i < j < n \wedge (\forall k < j \cdot \neg R\ a[k]\ a[i]) \wedge R\ a[i]\ a[j] \ \Rightarrow\ \neg R\ a[j]\ a[i]$$

At this point, we can decide to instantiate R in any way we like. If we choose a suitable instantiation (such as $<$), then the remaining conditions can be proved and after that, MV_ASM_DISCH can be applied to the final theorem. However, we can also choose to constrain R rather than instantiate it. It is easily seen that if R is nonreflexive and antisymmetric then the remaining verification conditions are true. We introduce these assumptions about R in a separate derivation step and end up with a window theorem which states that the original correctness assertion is correct for any nonreflexive and antisymmetric relation R. After that, we can still make an instantiation of the form $R = t$ and then use MV_ASM_DISCH to get rid of all occurrences of R. Note that in this case, there is no need to assume transitivity. Thus, by working with a metavariable we avoid making unnecessarily strong assumptions.

6 Conclusion

Postponing the instantiation of part of a term in a proof is desirable since the middle of the proof is usually more constraining than the beginning. The unknown information is represented by metavariables, and the moment of knowing by the instantiation of the metavariables. We have shown how examples of program reasoning can be helped using such a strategy, within the window inference framework of the HOL system.

Window inferences built on systems that support metavariables allow for proofs with uninstantiated terms. The HOL system does not provide support for metalogical variables, and since HOL window inference translates all derivation steps into basic HOL inferences it does not allow for proofs with uninstantiated terms. Furthermore, adding metavariables to HOL window inference would require that, upon instantiating the metavariables, the whole current stack be inspected, and updated.

As an alternative, we show how higher order variables with special naming conventions can be used to simulate metavariables. When these variables are instantiated, conjectures with their definitions are added and carried till the proof is over, or the window theorem becomes of interest. Then these conjectures can be discharged since the metavariables will not appear free in the conclusion of the window theorem. Alternatively, we can allow metavariables to remain in the final theorem. We then have a result about a *program scheme*, and the metavariable can later be instantiated for different concrete programs.

The discharging procedure has been automated as a general inference rule. This rule can discharge a list of conjectures about metavariables, and can take care of the order of discharging. Circular dependence is also detected. However, the user must be explicit about what metavariables depend on in order to avoid variable capturing. Another restriction is that the inference rule works as a post-processor of the window theorem of interest; we cannot establish a conjecture-free window theorem. Still, the produced theorems are perfectly valid and come at almost no cost since the meta and object levels are not mixed and we do not tamper with the simplicity and security of the HOL system or its window inference.

References

1. D. Basin, A. Bundy, I. Kraan, and S. Matthews. A framework for program development based on schematic proof. In *Proceedings of the 7th International Workshop*

on Software Specification and Design (IWSSD-93), 1993.
2. H. Becht, A. Bloesch, R. Nickson and I. Hayes. Ergo 4.1 Reference Manual. Tech.Rpt. 96-31, Dept. Computer Science, Queensland University, November 1996.
3. M. J. Butler, J. Grundy, T. Långbacka, R. Rukšenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In *Proc. FMP'97 - Formal Methods Pacific*, Discrete Mathematics and Theoretical Computer Science, Wellington, New Zealand, July 1997. Springer-Verlag.
4. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A Program Refinement Tool. *Formal Aspects of Computing*, 10(2), Springer Verlag, 1998.
5. G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide. Tech.Rpt 134, INRIA, December 1991.
6. M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, New York, 1993.
7. J. Grundy. *A Method of Program Refinement.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England 1993. Tech.Rpt. 318.
8. M. Hashimoto and A. Ohori. A typed context calculus. Tech.Rpt. RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, 1996.
9. C. Horn. *The Oyster Proof Development System.* University of Edinburg, 1988.
10. C. Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory.* PhD Thesis, INRIA, 1997.
11. R. G. Nickson and L. J. Groves. Metavariables and conditional refinements in the refinement calculus. Tech.Rpt. 93-12, Dept. Computer Science, Queensland University, May 1994.
12. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.
13. P. Robinson and A. Cheng. Qu-Prolog 3.2 Reference Manual. Tech.Rpt. 93-18, Dept. Computer Science, Queensland University, 1993.
14. P. J. Robinson and J. Staples. Formalizing a hierarchical structure of practical mathematical reasoning. *Journal of Logic and Computation*, 3(1):47-61, 1993.
15. M. Staples. Window inference in Isabelle. In *Proc. Isabelle Users Workshop*, University of Cambridge Computer Laboratory, September 1995.
16. M. Staples. *A Mechanized Theory of Refinement.* PhD thesis, University of Cambridge Computer Laboratory, Cambridge, England 1998.
17. C. L. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112, 1993.
18. J. von Wright. Extending Window inference. In *Proc. TPHOLs'98* (LNCS 1479), Springer-Verlag 1998.