Flexible Interactive Transformational Reasoning

Mark Staples [*][†]

23 March 1996

Abstract

Window inference is a transformational style of reasoning with support for the contextual transformation of sub-terms. Window inference has been successfully used as the basis of various refinement tools. Normal presentations of completed program refinements closely match the presentations of completed window inference proofs. However, in the development of a program refinement, window inference is not as flexible as it should be. Current implementations of window inference allow a user to work on only one sub-problem at a time. While developing a program refinement, a user may wish to work on many sub-problems at the same time—to quickly switch backwards and forwards between working on the sub-problems. This paper describes a design for a window inference system which provides simultaneous access to multiple sub-problems. In the core of the design, access is available to any sub-problem, but constraints can be added on top of the core in order to provide a hierarchical interface implementing window stacks or window trees. A further benefit of this design is that, unlike existing designs, it does not require reflexivity of transformation relations. A prototype implementation of this design in the Isabelle theorem prover is described.

1 Introduction

The refinement calculus of Back [1], Morris [11] and Morgan [10] offers a basis for the methodological development of correct programs from specifications. However, the use of refinement for the development of large programs has been limited, partly because of a lack of suitable tools. Recently, various tools supporting program refinement have been developed; one by Grundy [8], another at the Programming Methods Group of Åbo Akademi [3], and another at the Software Verification Research Centre of the University of Queensland [4]. These tools utilise a hierarchical transformational style of reasoning called *window inference* [15, 7, 9] as the main method of user interaction for the refinement activity.

The normal presentation of scripts of completed refinements is in a linear form, showing the stepwise development of progressively more deeply nested program fragments. Where the development branches, due to a conditional or sequential composition, each branch is refined in turn. Window inference is ideally suited to replaying such a completed refinement script, as it provides a way of progressively decomposing sub-problems for transformation, with the transformation of different branches happening in succession. However, during the exploratory development of a program refinement, a user may wish to consider more than one sub-problem at a time, or quickly alternate between working on different sub-problems. Standard presentations of window inference are not flexible enough to allow this.

The traditional view of window inference is outlined in Section 2. Section 3 describes a design for a system similar to window inference but which provides simultaneous access to multiple sub-problems. On top of this design, a system of suitably constrained transformations can implement a hierarchical interface resembling either window stacks or window trees. Following Grundy [9], an implementation of the design is given in terms of Natural Deduction, in order to show its soundness. Section 4 outlines a prototype implementation of this design on top of the Isabelle theorem prover.

2 Window Inference

Window inference is a hierarchical transformational style of reasoning. As a hierarchical style of reasoning, it provides a mechanism for problem decomposition, and additionally maintains access to the original problem formulation while users work on sub-problems. As

[*]Computer Laboratory, University of Cambridge. email: Mark.Staples@cl.cam.ac.uk
[†]Gonville and Caius College, Cambridge

a transformational style of reasoning, it supports the progressive transformation of terms under various relations. Window inference was first proposed to transform terms under equivalence relations [15] but was later generalised to support preorder relations [7, 9].

Window inference is the main style of reasoning in the Ergo theorem prover [17]. It has also been implemented by Grundy [7] on top of the HOL theorem prover [6], and by the author [16] on top of the Isabelle theorem prover [14].

2.1 Window Stacks

The central element of window inference is the *window*, which contains a term of interest called the *focus*, a relation under which the focus is being transformed, and contextual information which can be used in the transformation of the focus. Windows are stored in a *window stack*. Each window in a window stack corresponds to a particular sub-problem. The sub-problem in a window is contained within problems higher in the stack, with the top window corresponding to the deepest sub-problem. Only the top window in a window stack is visible, and only it can be changed.

There are five main operations involved in window inference.

Initial window: the initial focus and relation it is to be transformed under are established in a new window stack.

Open window: a sub-problem of the top window is pushed onto the window stack. The new window inherits any contextual information available in the parent window, and may also have new contextual information.

Close window: the top window is popped from the window stack, with transformed sub-problem replacing the original in the parent window.

Transform window: the focus of the top window is transformed.

Transform context: the context of the top window is transformed. This transformation is usually justified by a separate window inference proof.

Opening rules justify the transformation of a subproblem in the context of the parent problem. A collection of opening rules is maintained by the system. This collection is indexed by the position of the subterm accessed by each opening rule. When opening a window, a user supplies a sub-term position, and the appropriate opening rule is automatically chosen. The system can provide ways of deriving opening rules from a collection of simple opening rules. In particular, the automatic composition of opening rules should be supported in order to allow the transformation of nested sub-problems.

Window inference is intended to support interactive proof—window stacks maintain a conceptually simple model of a proof suitable for problem decomposition by humans. Conditional contextual rewriting is a noninteractive equivalent to window inference. Conditional contextual rewriting concentrates upon efficient automated proof and simplification, rather than on providing a conceptual model to aid interactive proof.

2.2 Window Stacks as Natural Deduction

Grundy [9] was the to first describe window inference in terms of Natural Deduction. This implementation of window inference allowed an easy soundness argument by appeal to the soundness of Natural Deduction. This representation of window inference also aided in (and was motivated by) the implementation of window inference in theorem provers supporting Natural Deduction. This implementation is briefly outlined below.

The main component of window inference is a stack of windows. Each window is represented by a theorem of the form $R(F_0, F_n)$, where R is a pre-order relation under which we have transformed an initial focus F_0 to our current focus F_n. If a window is a higher window in the stack, it represents a deeper sub-term, with each child window in the window stack corresponding to the state of a proof about the transformation of a sub-term of the parent window.

After the transformation of a sub-window, the transformation of a parent window needs to be justified. Theorems justifying such transformations are called *opening rules.* Each window in the window stack is joined to its parent window by an opening rule. An opening rule is of the form

$$(C \Longrightarrow r(f, f')) \Longrightarrow R(F[f], F[f'])$$

where $F[f]$ is the current focus, f is the sub-problem to be transformed in the surrounding term F, R is the relation under which the initial focus is transformed, r the relation for the sub-problem, and C is additional contextual information which may be assumed in the transformation of the sub-problem. Given such an opening rule, and a sub-problem transformation theorem with conclusion $r(f, f')$, the transformation of window containing a theorem $R(F_0, F[f])$ to resulting window containing a theorem $R(F_0, F[f'])$ implicitly appeals to the transitivity of the relation R.

Window inference was originally developed as a generic style of inference, allowing reasoning in modal and other logics. Describing window inference in terms of a Natural Deduction framework for classical logic does not allow reasoning in modal logics. In particular, it does not allow a representation of the modal contexts of Nickson and Hayes' program window inference [13]. Nickson and Hayes repeat the essence of Grundy's idea of implementing window inference in an established logic, but give a representation for their program window inference in terms of a modal functional logic.

3 Flexible Window Inference

Window inference was originally proposed as a hierarchical style of equational reasoning. Grundy [9] generalised window inference by removing the requirement that relations be transitive, in order to reason about the transformations of terms under preorders such as the refinement relation. Below, window inference is generalised by teasing apart its hierarchical nature, so that it can provide a more flexible basis for transformational reasoning in a graphical user interface (GUI). We will follow Grundy [9] in describing this style of reasoning in terms of Natural Deduction. Thus, we would in a similar way easily get soundness results for this style of reasoning. However, this point is not pursued in further detail here.

3.1 Supporting a Transformation GUI

User interfaces should be based on designs which match the conceptual models of its users in useful ways. Ideas for new styles of user interfaces can place new demands upon underlying implementations. Back's refinement diagrams [2] have been proposed as a way of displaying refinement histories. However, refinement diagrams place new demands upon window inference's ability to provide a logical model behind program refinement GUIs. Refinement diagrams can display the history of a refinement in a way that hides the order of development for independent subcomponents. As part of a development environment, a user would be presented with leaves of the development tree, any of which could be selected for further refinement. Window inference cannot easily reflect this presentation, as window stacks limit the user's attention to one sub-component at a time.

For example, consider Morgan's square-root refinement case study [10], as illustrated in a refinement diagram pictured in Figure 1. Moran's presentation of the the refinement proceeds depth-first, following the numbers given in the figure. However, if such a diagram was progressively created by a user in the course of a refinement, it would be easy to imagine a user choosing to develop the branches of the refinement in the order given as the letters in the figure. So, a user might begin developing one branch, but before completing that development, fully develop a simpler alternative branch. This kind of development is not encouraged by window inference. The user would have to explicitly close back to an earlier parent node, and open on the alternative child in order to transform it.

3.2 Flexible Window Inference

Window inference is especially concerned with providing a facility for the transformation of sub-terms. However, after transforming a sub-term and subsequently closing back to the top-level term, we will have transformed the top-level term.

Thus, instead of thinking of the state of a window inference proof as a stack of windows represented by theorems, we could instead think of the logical state of a window inference proof as just the theorem representing the top-most window. Then, a sub-window is a derived view of the top-level theorem, providing information about its focus and context. We call a rule which derives a sub-window view a *locus*. Instead of maintaining these sub-windows as a stack, we may wish to manage them in a more flexible way, in order to have access to more than one window at a time. This is the core idea of flexible window inference.

The main operations of flexible window inference are as follows.

Initial Window: the initial focus is established in a new top-level window.

Transform at a position: that position and its corresponding opening rule must be present in the current loci. Given a user-supplied theorem showing the transformation of the sub-term at that position, the opening rule is used to justify the transformation of the top-level theorem.

Add a locus: the new locus must not access a position which is already accessed by the current loci.

Remove a locus: a locus is removed from the current loci.

Transform the context of a locus: the context-management function for a locus is updated to represent a transformation of its context.

3.3 Flexible Window Inference as Natural Deduction

An outline of an design for flexible window inference in terms of Natural Deduction is given below.

A flexible window inference proof has a state, which consists of three parts:

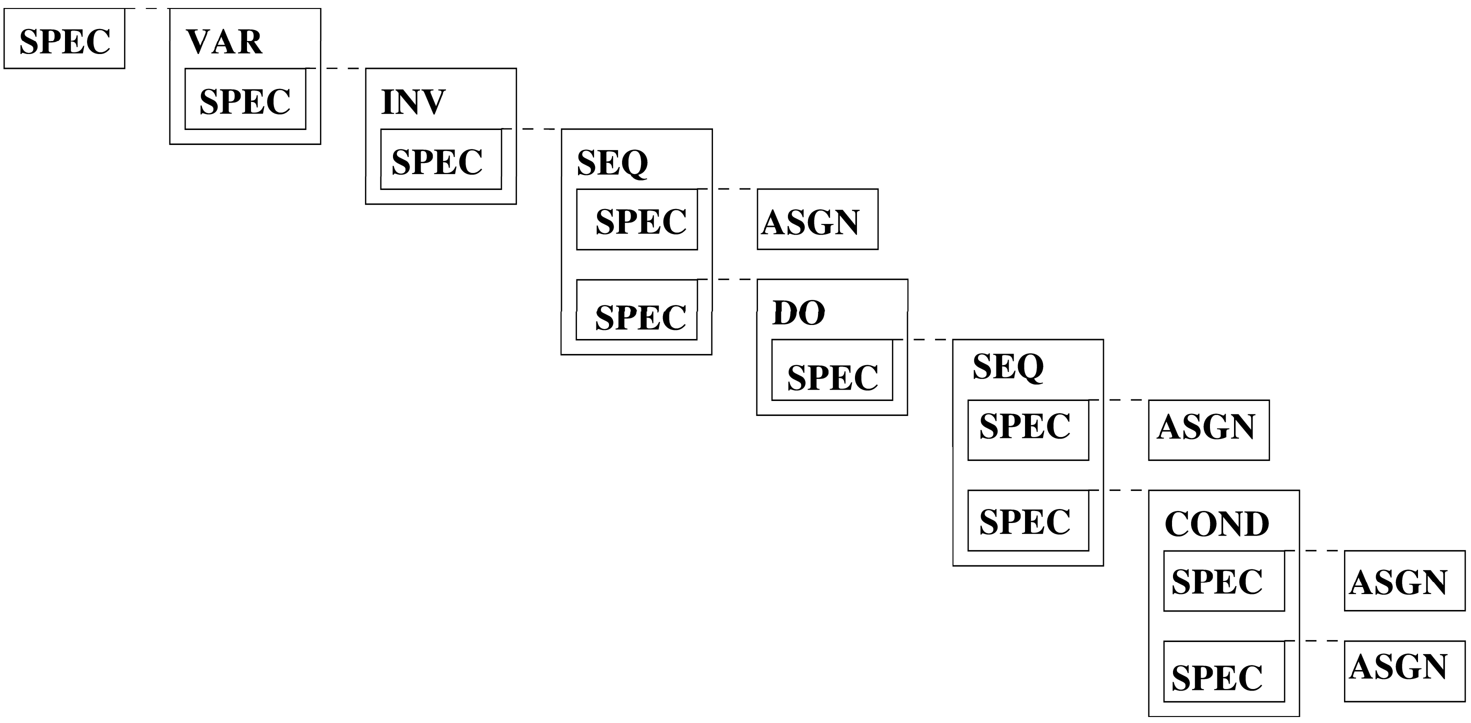

Figure 1: A Back refinement diagram overview of a completed refinement. Numbers indicate Morgan's order of development steps. Letters indicate a hypothetical alternative development order.

1. a *top-level theorem* $R(F_0, F_n)$ describing the transformation of the initial top-level term F_0 to the current top-level term F_n under some relation R;
2. a collection of distinct positions p_i and corresponding opening rules Op_i of the form

$$(C_i \Longrightarrow r_i(f_i, f_i')) \Longrightarrow R(F_i[f_i], F_i[f_i'])$$

   where each position p_i corresponds to the 'term with a hole' $F_i[-]$, and each $F_i[f_i]$ when instantiated, unifies with the current top-level term; and
3. a context management function which transforms an opening rule Op_i with context C_i into an opening rule with context C_i'.

A locus is the combination of an opening rule together with the position to which it provides access. Each locus provides a view upon the state of the proof. The context available through a locus is massaged by the context-management function.

In this general setting, the position of a locus may be contained within the position of another locus. At this level, no action is specified if a locus is rendered ill-formed by the transformation of the state at another locus. Loci management policies may be implemented at higher-levels of the user-interface in order to deal with this problem.

The main operations of flexible window inference are implemented as follows.

Initial Window: for a focus f, the initial state consists of a top-level theorem $f \equiv f$, an empty collection of loci, and correspondingly an empty context management function.

Transform at a position: for a p which is the position projection of some current locus with opening rule

$$(C_i \Longrightarrow r_i(f_i, f_i')) \Longrightarrow R'(F_i[f_i], F_i[f_i'])$$

a user supplied transformation theorem $r_i(f, f')$, an appropriate relation composition theorem

$$R(F_0, F_n) \wedge R'(F_n, F) \Longrightarrow R''(F_0, F)$$

is chosen to change the state $R(F_0, F[f])$ to a new state $R''(F_0, F[f'])$ The loci and context management function are unchanged.

Add a locus: for a new locus whose position projection the is not among the position projections of the current loci, the new locus is added to the current loci. The context management function at that position is the identity function. The top-level theorem is unchanged.

Remove a locus: a locus is removed from the loci, and correspondingly from the context management function. The top-level theorem is unchanged.

Transform the context of a locus: for a rule f transforming the context of an opening rule derived from a locus with position projection p, the current context management function H is overwritten at position p by $f\ o\ H(p)$. The top-level theorem is unchanged.

3.4 Problems and Benefits

The standard difficulty with opening at multiple foci independently is that it is not sound to use context from certain multiple window opening rules simultaneously. For example, consider the opening rules for transforming each side of a conjunction:

$$\begin{aligned} (B \Longrightarrow (A = A')) &\Longrightarrow A \wedge B = A' \wedge B \\ (A \Longrightarrow (B = B')) &\Longrightarrow A \wedge B = A \wedge B' \end{aligned}$$

In order to transform a focus term $A \wedge B$, we cannot both assume A to transform sub-focus B and simultaneously assume B to transform sub-focus A. However, if we maintain sub-windows as a collection of opening rules, then the context for each sub-problem is effectively re-derived after each implicit transformation of the top-level focus. So, for example, after using context B to transform the sub-focus A to A', the derived context for a sub-focus on B would be A'.

Thus the effect of a transformation may be non-local, in that the transformation of a focus may change the context for another focus. This should to be carefully managed by higher-levels of an interface either by restricting the occurrence of side-effecting focus transformations, or by providing appropriate information to a user indicating the occurrence of a side-effecting transformation.

The transformation of the top-level theorem can render a locus invalid by:

- making the position in that locus not a valid position in the top-level term; or by
- making the 'term with a hole' $F[-]$ not match the top-level term.

Such a locus can be automatically removed from the loci, be hidden, or be managed in some other way by the interface.

The design presented above leads to a more relaxed view about the reflexivity requirement of relations in window inference. In Grundy's design for window inference, for every window in the stack with initial focus f, the reflexivity of a relation r is used to create its initial theorem $r(f, f)$. However, in the setting outlined above, we may only represent the top window as a theorem. Hence, we only demand reflexivity for the relation in the top window, and not for relations in sub-windows. Moreover, if we consider more general relation composition theorems instead of transitivity theorems, then we can reason about certain kinds of irreflexive relations. For example, if we wish to transform an initial focus f under the relation $<$, then we can create an initial theorem $f \leq f$, and rely upon the relation composition theorem:

$$(a \leq b) \wedge (b < c) \Longrightarrow a < c$$

3.5 Constraints: Stacks and Trees

This presentation of window inference has abstracted away from the hierarchical nature of traditional window inference. This should provide a more flexible basis for GUIs supporting transformational reasoning. However, some kind of hierarchy of proof may fit better with a mental model of transformational reasoning. We leave this question to Human-Computer Interaction researchers. Certainly as noted above, a hierarchy of window stacks means that contextual information is only used locally. So, window stacks or window trees may turn out to provide the basis for a more usable kind of GUI.

We can implement window stacks or window trees in this general framework by imposing additional constraints on the opening rules.

For example, for a suitable definition of 'deepest locus', we would have an implementation of window stacks if we:

1. force all transformations to operate upon the deepest locus; and
2. only allow the addition of loci at positions inside the deepest locus.

Similarly, for a suitable definition of 'leaf locus', we would have an implementation of window trees if we:

1. force all transformations to operate upon a leaf locus; and
2. only allow the addition of loci at position inside a leaf locus.

Other, more flexible, approaches to accessing and maintaining loci may be required for a graphical user interface. The prototype described below provides a suitable framework for experimenting with such approaches.

4 An Isabelle Prototype

4.1 The Isabelle Theorem Prover

Isabelle [14] is a generic theorem prover in the LCF tradition [5]. Isabelle is written in SML, the type discipline of which prohibits invalid theorems. The only way of constructing a value in the type of theorems `thm` is by appeal to axioms and basic primitive inferences represented by values and functions of type `thm`. This means that we can safely program inference environments on top of Isabelle. The standard backward-reasoning goalstack package supplied with Isabelle is one such inference environment. The flexible window inference prototype outlined later in this section is another.

As a generic theorem prover, Isabelle can represent a wide variety of logics. Isabelle represents both the syntax of its meta-logic, and all the syntax of its object logics, in a term language which is a datatype in SML. The language provides constants, application, lambda abstraction, and bound, free and scheme variables[1].

Isabelle's meta-logic is an intuitionistic polymorphic higher-order logic. Isabelle provides the following constants to represent rules and axioms.

Meta-level universal quantification $\bigwedge$, is used to represent variable-capture side-conditions in the statement of rules or axiom-schemes in an object logic;

Meta-level implication $\Longrightarrow$, is used to represent rules of an object logic. This corresponds to entailment for non-modal logics.

Meta-level equality $\equiv$, is used to represent definitions of an object logic.

The constants construct *propositions*: terms of type `prop`. The statement of a theorem is given by a term of this type.

The semantics of Isabelle's object logics are given by declaring axioms and inference rules. Definitional or axiomatic extensions to a object logics are possible. Theories[2] can be combined under a structured management of theorems and their theories provided in the core system. Isabelle also provides tactics and tacticals, a backward proof interface, advanced parsing and pretty-printing support, and a growing collection of generic decision procedures and other proof tools which are applicable to many object logics.

4.2 Prototype Architecture

Isabelle's meta-logic provides a suitable basis for modelling the generic description of flexible window inference as Natural Deduction.

The three parts of the state of a flexible window inference proof are represented fairly directly by an SML type `winstate`. This type is a triple composed of the following types.

`thm:` of the form $\bigwedge x_1, ...x_m.\ A \Rightarrow R(F_0, F_n)$ describing how we have transformed an initial focus F_0 to the current focus F_n under some relation R using unproven assumptions A and variables $x_1, ...x_m$.

`loci:` which is a list of values of type `locus`. A `locus` is of list of pairs of positions and theorems. The position that a locus provides access to is the composition of the positions in this list, and the opening rule associated with the locus is composition of the theorems in the list. This representation provides a way to support the derivation of opening rules accessing deep sub-terms. All of the loci should be distinct in their position projections.

`position -> (thm -> thm seq):` which is a function transforming the composed opening rule for a locus with the given position projection in the context of the current top-level theorem.

Under Isabelle's higher-order unification, a function variable may sometimes unify with a term in many ways. So for example, if we use an opening rule `arg_cong`:

$$x = y \Longrightarrow f(x) = f(y)$$

to access a top-level theorem with focus $A \wedge B = z$, then x could unify with any of A, B, or $A \wedge B$. So, we must use the position information associated with a locus in order to constrain the composition of opening rules in a locus. The position description used in the Isabelle prototype is similar to Grundy's [7], but is not described further here.

4.3 A Transformational Tactic Environment

In Isabelle's backward proof package, tactics are the mechanism provided for changing the state of the proof. There, the state is of type `thm`, and tactics are of type `thm -> thm seq`. Isabelle's higher-order unification can potentially produce an infinite number of unifiers, and hence the sequence `seq` is a lazy list

[1] Scheme variables are logically equivalent to free variables in Isabelle's meta-logic. They differ in that scheme variables can be instantiated during unification.

[2] In Isabelle, object logics and theories are equivalent.

of possible results of the tactic. It is possible to backtrack over these results, in order to see each result in turn.

Similarly, where the state of a window inference proof is of type `winstate` as described above, wintactics which operate on such states are of type `winstate -> winstate seq`. We can develop specific wintactics which operate on either or both of the theorem and loci component of the state. The wintactic framework is the basis for the implementation of the main flexible window inference operations, following the fairly directly the the description given in Section 3.3. Constraints on the kinds of actions available as wintactics can lead to hierarchical constraints providing window stacks or window trees, as described in Section 3.5.

The presence of scheme variables in Isabelle's term language and Isabelle's support for unification should allow the development of refinement tactics such as those described in [12].

5 Conclusions

This paper has briefly outlined the design for a system of transformational reasoning which allows contextual access many sub-problems at a time. The system is similar to window inference, but is not limited by window inference's hierarchical constraints. The proposed system also generalises window inference's equivalence relations to arbitrary composable relations.

Although standard presentations of window inference can replay the scripts of completed refinements, the more flexible form of window inference described here should be more suited to supporting the exploratory development of a program refinement. It should be especially effective in the support of a graphical interface for Back's refinement diagrams.

Acknowledgements

This work grew in part from discussions with Jim Grundy, Tom Melham, and Joakim von Wright at the 1995 Glasgow Research Festival in Computer Science. My participation at the festival was funded jointly by Gonville and Caius college, the Department of Computer Science of the University of Glasgow, and the Computer Laboratory of the Cambridge University. I am supported by a Packer Scholarship, administered by the Cambridge Commonwealth Trust. Mike Gordon and Chris Owens provided useful comments on earlier drafts of this paper.

References

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

[2] R.J.R. Back. Refinement diagrams. In J.M. Morris and R.C. Shaw, editors, *Fourth Refinment Workshop*, Workshops in Computing, pages 125–137. Springer-Verlag, 1991.

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

[4] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. Refinement in Ergo. Technical Report 94-44, Software Verification Research Centre, The University of Queensland, July 1995.

[5] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[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, *Proceeedings 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. Unpublished note, February 1996.

[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*, volume 828 of *LNCS*. Springer-Verlag, 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. Staples. Window inference in Isabelle. In L. Paulson, editor, *Proceedings of the First Isabelle User's Workshop*, volume 379, pages 191–205. University of Cambridge Computer Laboratory Technical Report, September 1995.

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